CHAPTER 10

DATA-ORIENTED DESIGN

INTRODUCTION

Data-oriented design uses data as the basis for clustering processes, building databases, and identifying potential distribution of the application. In this chapter, we continue the discussion of Information Engineering as the example of data-oriented methodology. Since IE has several 'incarnations' that differ slightly, it is important to note that IE in this chapter is consistent with the Martin [1992], Texas Instruments [1988], and Knowledgeware™[1] versions.

CONCEPTUAL FOUNDATIONS

Information Engineering is the closest to a complete methodology of the methods in common use. It borrows from research and practice to build a complete view of the application and its environment. Structured programming tenets describe the importance of limiting program structure, as much as possible, to selection, iteration, and instruction sequence components. 'Go to' statements should be minimized. Modules should have one entry and one exit. In IE design, these tenets are practiced in structuring the application as well as the program modules.

Subject area database design is based on theories of relational database and practice of data design. Data should be clustered with processes which create the data. Those processes determine 'subject areas' of data. Subject databases are stored in the same database environment and their processes are in integrated applications. These topics were discussed in Chapter 5 and are not repeated here. During analysis, the data entities are normalized and relations are identified (Chapter 9). Normalized data is the starting point for physical database design. Physical database design may automate the normalized relations directly or may denormalize for performance purposes. Also, in organizations with many using locations and potential for distribution of data and processes, a strategy for distribution is defined. These two activities, potential denormalization and distribution, are based on practical guidelines rather than theory.

From practice, we know that there is more to implementing an application than designing program specifications and a database. We need to design screens, a screen dialogue, provide for unauthorized and unwanted damage to the data, provide for conversion from the old to the new method of data storage, design and plan application implementation, install hardware, design and plan application tests,

1 Knowledgeware™ is a product of Knowledgeware, Inc., Atlanta, Ga.

and develop training programs for users. While all of these tasks are discussed in some books on IE, these activities are done regardless of methodology, and to discuss them as pertaining only to IE would be misleading. For this reason, the topics in this chapter include screen dialogue design, hardware planning, and providing for data security, recovery, and audit controls, in addition to procedure and database design. Human interface design, conversion, and training are discussed in Chapter 14; testing is the subject of Chapter 17.

DEFINITION OF INFORMATION ENGINEERING DESIGN TERMS

A full list of the activities in IE design is given here; included are references to chapters in which some topics are discussed.

1. Design security, recoverability, and audit controls
2. Design human interface structure
   - Develop menu structure
   - Define screen dialogue flow
3. Data analysis
   - Reconfirm subject area database definition
   - Denormalize to create physical database design
   - Conduct distribution analysis and recommend production data distribution strategy
4. Develop an action diagram and conduct reusability analysis
5. Plan hardware and software installation and testing
6. Design conversion from the old to the new method of data storage (Chapter 14)
7. Design and plan application tests (Chapter 17)
8. Design and plan implementation (Chapter 14)
9. Develop, schedule, and conduct training programs for users (Chapter 14)

The topics in this chapter are design of data usage, action diagrams (which are program specs), screen dialogues, security, recovery, audit controls, and installation planning. They are discussed in this section in the order above, by the amount of work involved, and their importance to the application.

The first activity in IE design is to confirm design of the database and determine the optimal data location. Invariably, when the details of processing are mapped to specifications, data usage changes from that originally envisioned. To confirm database design, the data is mapped to application processes in an entity/process (CRUD) matrix and the matrix is reanalyzed. (See Chapter 9 for a more complete discussion of entity/process matrices.) The entity/process matrix (see Figure 10-1) clusters data together based on processes with data creation authority. The subject area databases defined by the clusters are stored in the same database environment.

The second step of database design is to determine a need to denormalize the data. Recall that **normalization** is the process of removing anomalies that would cause unwanted data corruption. **Denormalizing** is the process of designing storage items of data to achieve performance efficiency (see Figure 10-2). Having normalized the data, you know where the anomalies are and can design measures to prevent the problems.

The next activity in data analysis is to determine the location of data when choices are present. A series of objective matrices are developed and analyzed. The matrices identify process by location and data by location and transaction volume. These are used to develop potential designs for distribution of data. The application processes and data are both mapped to locations. Cells of the **process/location matrix** contain responsibility information, identifying locations with major and minor involvement (see Figure 10-3). This information is used to determine which software would also be required to be distributed, if distribution is selected.

Two data/location matrices are developed. The first data/location matrix identifies data function as either update (i.e., add, change, or delete) or retrieval

Entities = / Processes	Purchase Order	PO Item	Vendor-Item	Inventory Item	Vendor
Create & Mail Order	CRUD	CRUD	CRU	R	R
Call Vendor & Inquire on Order	RU	RU	RU	R	R
Verify Receipts against Order	RU	RU	RU		R
Send Invoices to Accountant	RD	RD			
File Order Copy by Vendor	R	R			
Identify Late & Problem Orders	R	R	R	R	RU
Identify Items & Vendors			R	R	CRU
Call Vendor to Verify Avail/Price			RU		RU

FIGURE 10-1 Example of Entity/Process Matrix

by location (see Figure 10-4a). The second defines options for data in each location (Figure 10-4b). Together these matrices identify options for distributing data. The options for distributed data are replication, partitioning, subset partitioning, or federation (see Figure 10-5). **Replication** is the copying of the entire database in two or more locations. **Vertical partitioning** is the storage of all data for a subset of the tuples (or records) of a database. **Subset partitioning** is the storage of a partial set of attributes for the entire database. **Federation** is the storage of different types of data in each location, some of which might be accessible to network users. The selection of distribution type is determined by the usage of data at each location.

Then, a **transaction volume matrix** is developed to identify volume of transaction traffic by location. Cells of this matrix contain average number of transactions for each data relation/process per day (see Figure 10-6). In an active application, hourly or peak activity period estimates of volume might be provided. During matrix analysis, the data and processes are clustered to minimize transmission traffic. Then formulae are applied to the information to determine whether the traffic warrants further consideration of distribution.

Finally, subjective reasons for centralizing or for distributing the application are developed. The subjective arguments ensure that political, organizational, and nonobjective issues are identified and considered. Examples of subjective motivations for centralization/distribution relating to Figures 10-4, 10-5, and 10-6 are in Table 10-1. Recommendations on what, how, and why to distribute (or centralize) data are then developed from the matrices and subjective analysis. The recommendations and reasoning are presented to user and IS managers to accept or modify.

After data are designed, the design of the human interface can begin with a definition of interface requirements. The hierarchy diagram is used to determine the structure of selections needed by the application. A **menu structure** is a structured diagram translating process alternatives into a hierarchy

Unnormalized	First Normal Form	Second Normal Form	Third Normal Form	DeRelation
Order Number	Order Number		Order Number	Order
Order Date	Order Date		Order Date	
Order Ship Terms	Order Ship Terms		Order Ship Terms	
Order Payment	Order Payment		Order Payment	
Terms	Terms		Terms	
Customer Number	Customer Number	→	Customer Number	
Customer Name	Customer Name			
Customer Address	Customer Address			Customer
			Customer Number	
*Item Number			Customer Name	
Item Description			Customer Address	
Item Quantity	Order Number	Order Number		Order Item
Item Price	Item Number	Item Number		
Item Extended Price	Item Description	Item Description	→	
	Item Quantity	Item Quantity		
	Item Price	Item Extended Price		X
	Item Extended Price			
		Item Number	→	Inv. Item
		Description		
		Price		

Denormalized Design for Order

ORDER	Order Number Order Date Order Ship Terms Order Payment Terms Customer Number Customer Name Customer Address
Order Item	Order Number Item Number Customer Number Customer Name Item Description Item Quantity Item Price Item Extended Price

FIGURE 10-2 Example of Denormalized Data for an Order

Function	Location A	Location B	Location C	Location D	Location E
Purchasing	\	X			
Marketing	X	X	\		
Customer Service		X	\		
Sales	X	X	\		
Product Development	X	X	\	\	
Research & Dev.			X	X	\
Manufacturing				\	X

Legend:

X—Major Involvement

\—Minor Involvement

FIGURE 10-3 Example of Process/Location Matrix

of options for the automated application (see Figure 10-7). In general, we plan one menu entry for each process hierarchy diagram entry between the top and bottom levels. One level of menus corresponds to one level in the process hierarchy diagram. At the lowest level of the process hierarchy, a process corresponds to either a program or module. Screens at the lowest level are determined by estimating execute units. These functional screens may not be final in menu structure definition because execute

Data Usage by Location Matrix

Subject Data	Location A	Location B	Location C	Location D	Location E
Prospects	All—UR	All—UR			
Customer	All—UR	All—UR			
Customer Orders	All—UR	Subset—Own Products—UR		All—R	All—R
Customer Order History	All—R	All—R	All—R	All—R	
Manufacturing Plans	Subset—own products—R	Subset—own products—R		Subset—own site—UR	All—UR
Manufacturing Goods in Process	Subset—own products—R	Subset—own products—R		Subset—own site—UR	All—UR
Manufacturing Inventory	Subset—own products—R	Subset—own products—R	All—R	Subset—own site—UR	All—UR

U = Update, R = Retrieve

FIGURE 10-4a Example of Data Matrices by Location

Distribution Alternatives by Location

Subject Data	Location A	Location B	Location C	Location D	Location E
Prospects	Replicate—Central Copy	Replicate			
Customer	Replicate—Central Copy	Replicate			
Customer Orders	Central Copy—A data	Vertical Partition by Product		Access central copy with delay	Access central copy with delay
Customer Order History	Replicate Central Copy	Replicate or access central copy with delay		Access central copy with delay	
Manufacturing Plans	Replicate or access central copies with delay	Replicate or access central copies with delay		Subset—own site	Subset—own site with delayed access to D
Manufacturing Goods in Process	Access D and E Databases	Access D and E Databases		Subset—own site	Subset—own site with delayed access to D
Manufacturing Inventory	Access D and E Databases	Access D and E Databases		Subset—own site	Subset—own site with delayed access to D

FIGURE 10-4b Example of Data Matrices by Location

unit design is usually a later activity. Once the menu structure is defined, it is given to the human interface designer(s) for use during screen design (Chapter 14).

The structure is then analyzed further to determine the allowable movement between the options on the menu structure. The **dialogue flow diagram** documents allowable movement between entries on the menu structure diagram (see Figure 10-8). On the diagram, rows correspond to screens and columns correspond to allowable movements. For instance, in the menu structure example (Figure 10-7), *Customer Maintenance* has four subprocesses. A dialogue flow diagram shows how *Customer Maintenance* is activated from the main menu (or elsewhere) and the options for movement from that level. From the *Customer Maintenance* menu, the options are to move to the main menu or to one of the four subprocesses. The dialogue flow diagram is used by the designers in developing program specifications, by the human interface designer(s) in defining screens, and by testers in developing interactive test dialogues.

Next, procedure design begins with analysis of the process hierarchy and process data flow diagrams developed during IE analysis (Chapter 9). Remember, in analysis, we developed one process data flow diagram (PDFD) for each activity. Now each PDFD is converted into an action diagram. An **action diagram** shows procedural structure and processing details suitable for automated code genera-

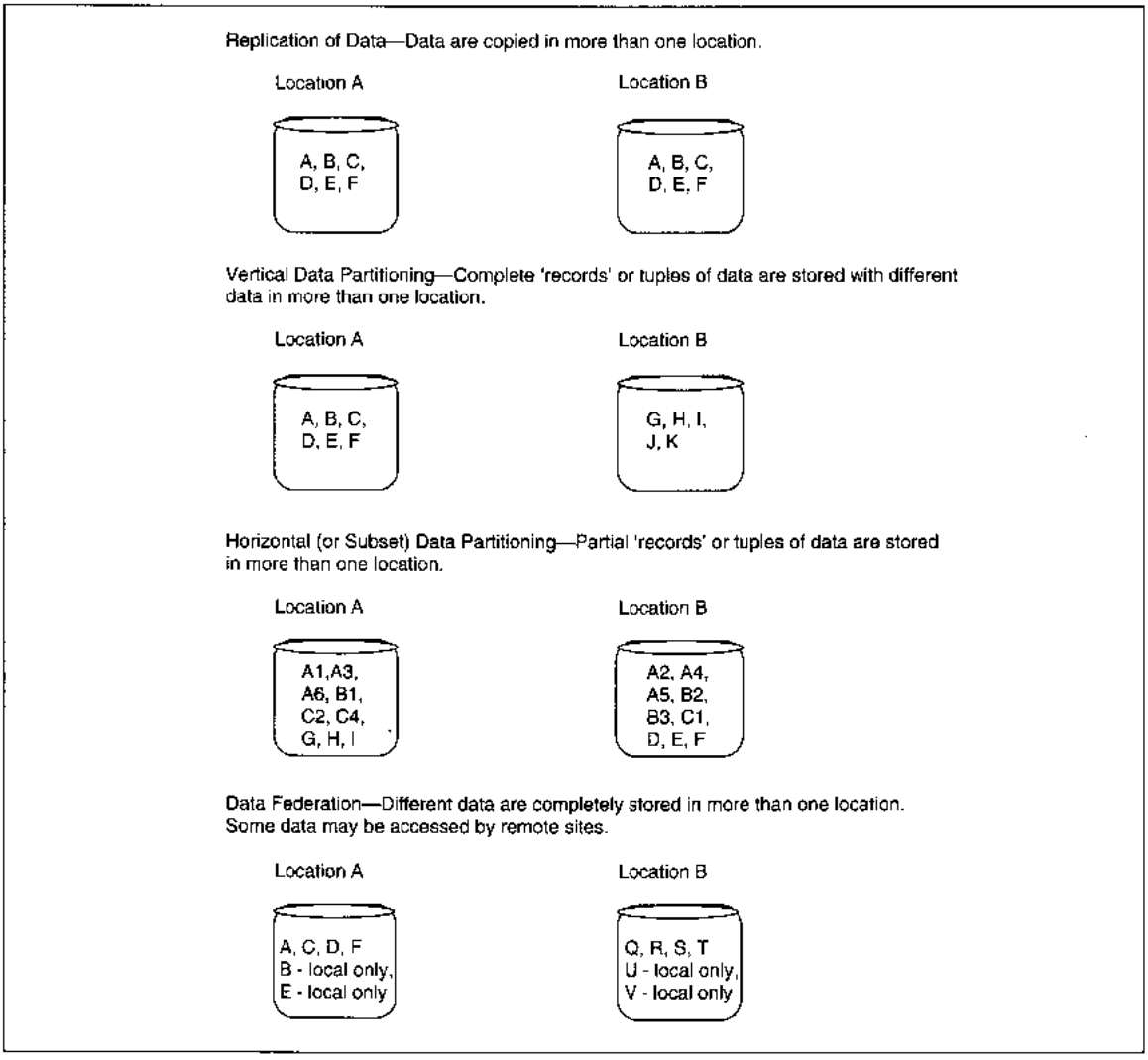


FIGURE 10-5 Data Distribution Alternatives

tion. An action diagram is drawn with different types of bracket structures to show the hierarchy, relationships, and structured code components of all processes.

The first-cut action diagram translates the PDFD into gross procedural structures (see Figure 10-9). Then, using detailed knowledge obtained during the information gathering process, the details of each procedure are added to the diagram to develop program specifications (see Figure 10-10). These program specifications may then be packaged into modules that perform one function. Data entities are added to the diagram at the level they are accessed (see Figure 10-11). Progressively more detail about data usage is provided about data attributes. Arrows are attached to show reading and writing of data (see Figure 10-12).

When the details are completely specified, the action diagram is mapped to procedural templates to determine the extent to which reusable modules

	Subject Database						
Location/Function	Prospect	Customer	Customer Order	Customer History	Mftg. Plan	Mftg. WIP	Mftg. Inven.
A							
Customer Service		100 R 20 U	250 R 400 U	5 R	2 R	2 R	
Sales	50 R 20 U	50 R 30 U	150 R 50 U	50 R	2 R	2 R	15 R
Marketing	15 R	5 R	10 R	50 R	2 R		1 R
B							
Customer Service		250 R 50 U	250 R 400 U	50 R	250 R	250 R	250 R
Sales	25 R 20 U	25 R 5 U	10 R 100 U	70 R	2 R	2 R	15 R
Marketing	20 R	10 R	10 R	50 R		2 R	5 R
D							
Manufacturing					50 R 5 U	50 R 250 U	500 R 2,000 U
E							
Manufacturing					100 R 15 U	200 R 2,500 U	500 R 25,000 U

Legend: U = Create, Update or Delete; R = Retrieve

FIGURE 10-6 Example of Transaction Volume Matrix

can be used in the application, and the changes to the action diagrams required to define modules for reuse. A **procedural template** is a general, fill-in-the-blanks guide for completing a frequently performed process. For instance, error processing and screen processing can be defined as reusable templates (see Figure 10-13). A data template is a partial definition of an ERD or database that is consistent within a user community. For example, the insurance industry has common data requirements for policy holders, third party insurance carriers, and policy information; most companies have similar accounting data needs, and so on. To be a **candidate for template** definition, a process must do exactly the same actions whenever it is invoked, and data must be consistent across users.

After reusability analysis, the action diagram set is finalized and used to generate code. If the application is specified manually, the action diagrams are given as program specifications to programmers who begin coding. If the application uses a CASE tool, automatic code generation is possible. A **code generator** is a program that reads specifications and creates code in some target language, such as Cobol or C. If the application uses a code generator, the action diagram contains the symbols and procedural detail specific to the code generation software. If the application uses a 4GL, the action diagram might contain actual code. If manual programming uses a 3GL or lower, the action diagram contains pseudo-code consisting of structured programming constructs.

The next activity in IE design is to develop security plans, recovery procedures, and audit controls for the application. Each of these designs restrict the application to performing its activities in prescribed ways. The goal of **security plans** is to

TABLE 10-1 Example of Subjective Reasons for Centralization and Distribution

Measure	Argument
General	
D	Geographic distribution by function by product makes centralization difficult
D	Centralized mainframe in a sixth location is not close to distributed sites, nor interested in serving their needs
d	Little product overlap between sites A and B
Location A	
d	General Manager in Location A—smallest needs
d	GM wants 'what is best' for division
C	Little technical expertise in the location; would increase travel expense required to support hardware/software
Location B	
C	Customer service needs fast response to fulfill corporate objectives (90% of requests serviced within one phone call, less than three minutes)
C	Most application expertise in division is located here
C	IS manager, located here, wants the applications and data under his control
Location C	
d	Actions mostly independent of other sites
d	Delays in retrieval of information could be tolerated
Location D	
d	Historically, location controls its own hardware/software
d	Hardware/software not currently compatible with A, B, or C
Location E	
d	Historically, location controls its own hardware/software
d	Historically, software has been successfully developed/bought as joint activity with IS group in Site B

Legend:
D/C = Strong argument for *D*istribution/*C*entralization
d/c = Weak argument for *d*istribution/*c*entralization

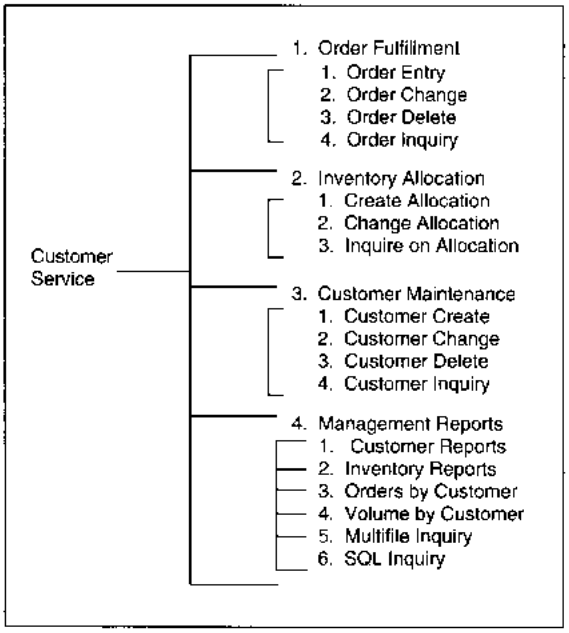


FIGURE 10-7 Menu Structure Example

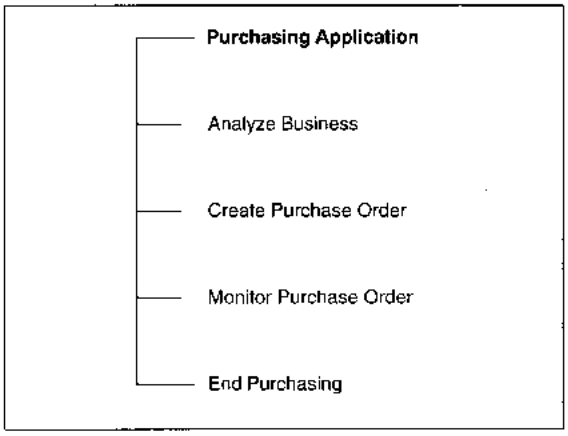


FIGURE 10-9 Action Diagram Example

protect corporate IT assets against corruption, illegal or unwanted access, damage, or theft. Security plans can address physical plant, data, or application assets, all by restricting access in some way. **Physical security** deals with access to computers, LAN servers, PCs, disk drives, cables, and other components of the network tying computer devices together. **Data security** restricts access to and functions against data (e.g., read, write, or read/write). **Application security** restricts program code from access and modification by unauthorized users. Examples of the results of security precautions are locking of equipment, requirement of user passwords, or assignment of a software librarian for program changes.

Recovery procedures define the method of restoring prior versions of a database or application software after a problem has destroyed some or all of

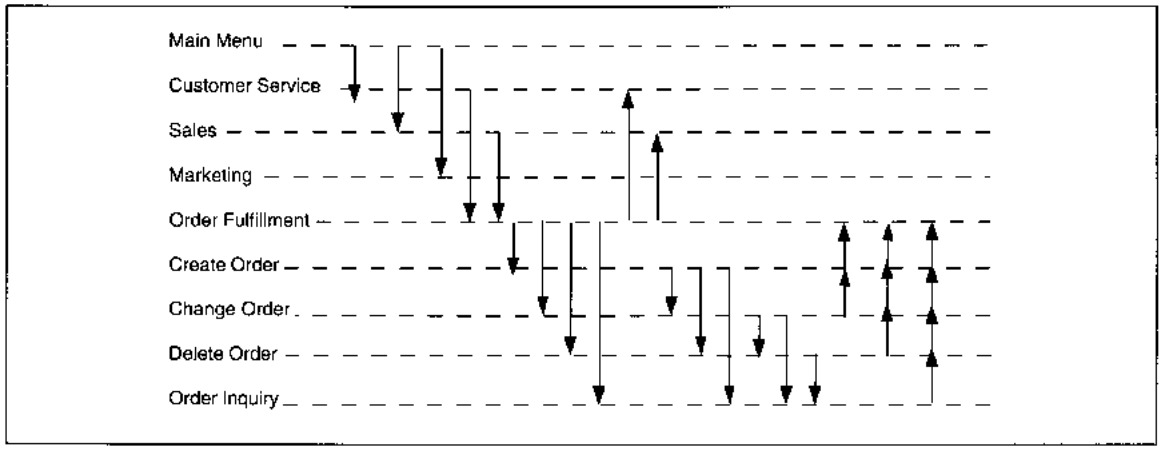


FIGURE 10-8 Dialogue Flow Diagram Example

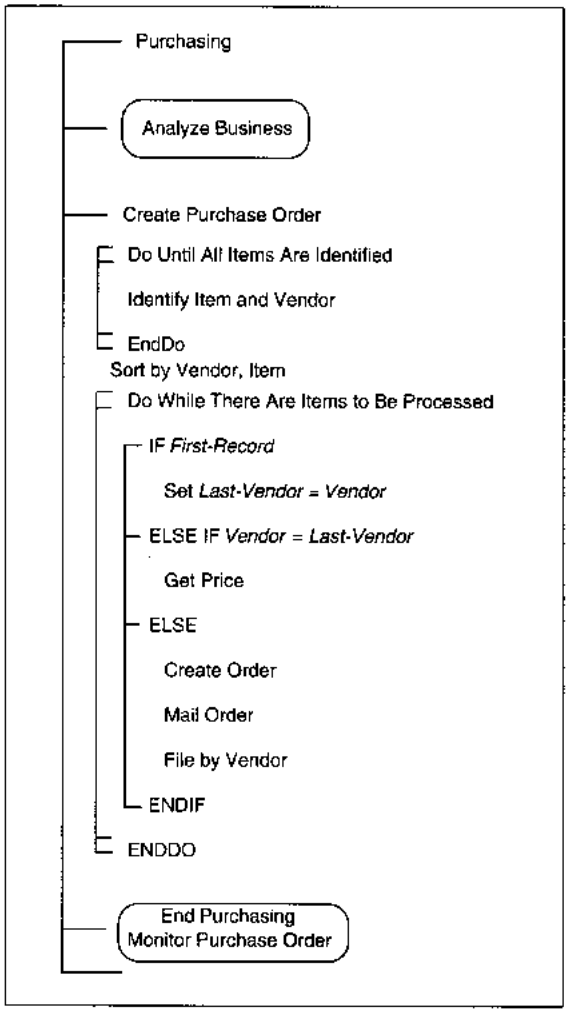


FIGURE 10-10 Action Diagram with Create Purchase Order Process Detail

it. Recovery is from a copy of the item. **Backup** is the process of making extra copies of data to ensure recoverability. Disasters considered in the plan include user error, hacker change, software failure, DBMS failure, hardware failure, and location failure. **Recovery** is the process of restoring a previous version of data (or software) from a backup copy to active use following some damage to, or loss of, the previously active copy. The backup/recovery strategy should be designed to provide for the six types of errors above. Several backup options add requirements to program design that need to be accommodated.

Next, **audit controls** are designed to prove transaction processing in compliance with legal, fiduciary, or stakeholder responsibilities. Audit controls usually entail the recording of day, time, person, and function for all access and modification to data in the application. In addition, special totals, transaction traces, or other special requirements might be applied to provide process audit controls.

Last, hardware installation is planned and implemented, if required for the application. Again, there is no theory or research about hardware installation, but long practice has given us guidelines on the activities and their timing.

INFORMATION ENGINEERING DESIGN

In this section, we discuss each activity in IE design in detail, and relate them to the ABC Video rental application. IE design topics in this section, in order of their occurrence in the application development process, include development of the following:

- data use and distribution analysis
- security, recovery, and audit controls
- action diagrams
- menu structure and dialogue flow
- hardware and software installation and testing plans

Analyze Data Use and Distribution

Guidelines for Data Use and Distribution Analysis

The two activities in this section precede physical database design which is assumed to be performed by a DBA. First, data usage analysis is performed to confirm the logical database design. Then the potential for distributing data throughout the organization is analyzed. The result is a strategy

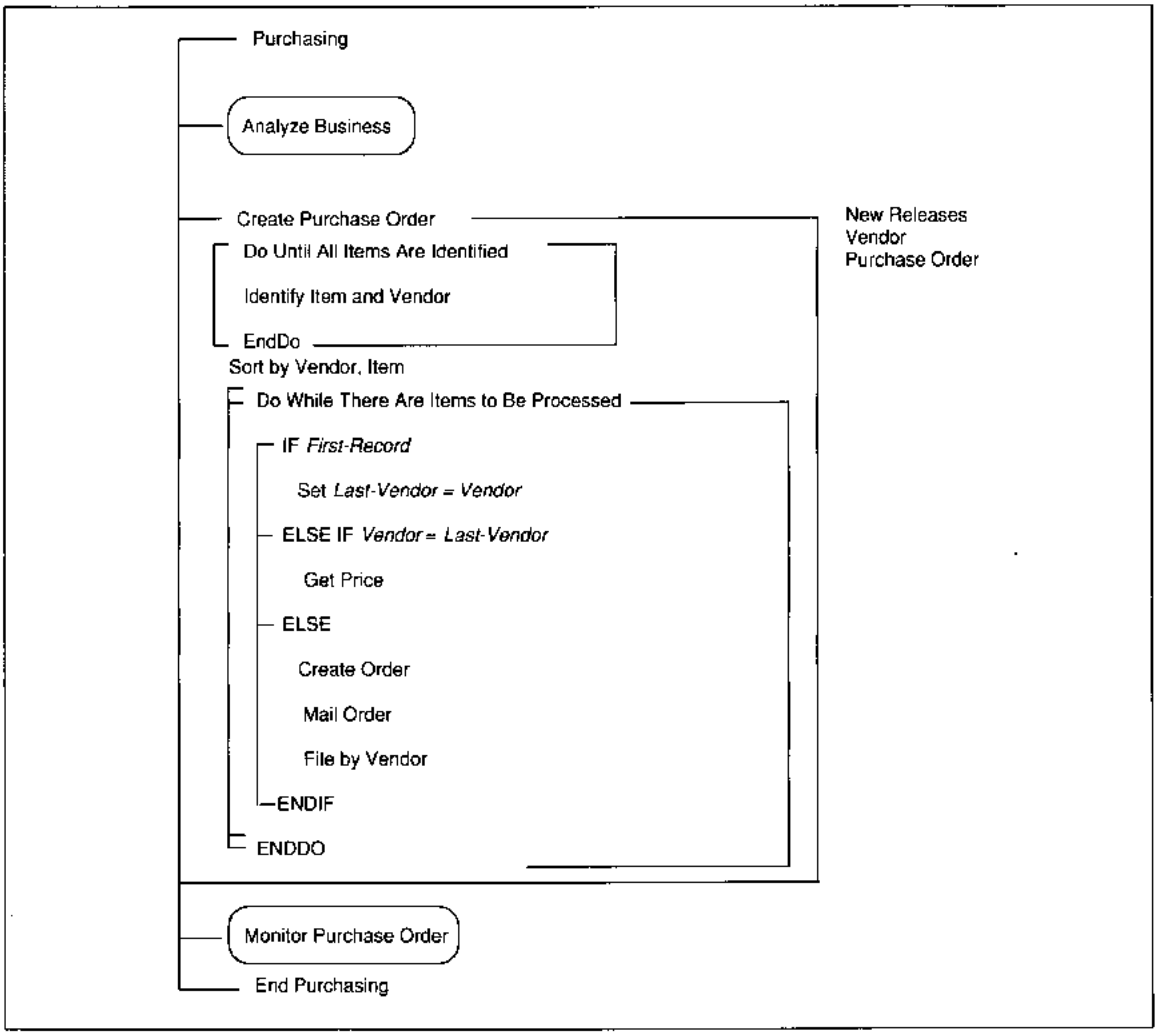


FIGURE 10-11 Action Diagram with Entities

for data and software location that best fits user needs.

The entity/process (CRUD) matrix from IE analysis is reanalyzed and mapped to the completed action diagram. Each process is identified on the action diagram with its associated data items and the related entity. Recall that the clustering of entities and processes on the matrix is primarily based on which processes have create responsibility for the data. The entities and processes are arranged into a *new* entity/process matrix which is compared to the one developed during analysis. If the definition of subject area databases does not change, the distribution analysis can begin. If the definition of subject area databases does change, the logical definition of the databases is redone as discussed in Chapter 9.

The second step to data analysis is to determine the potential for data distribution. Distribution analysis uses three matrices as the objective basis for determining whether data should be distributed.

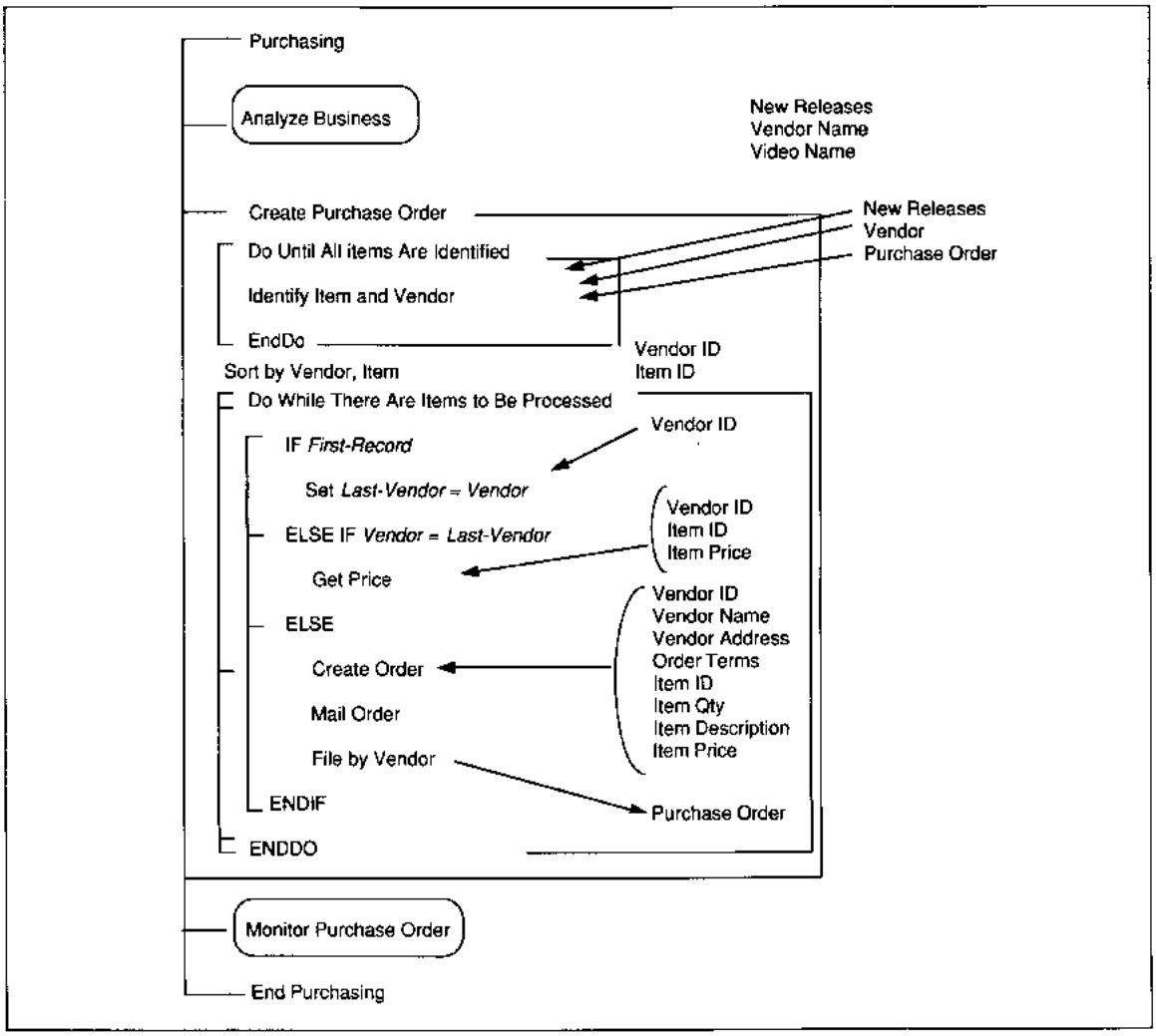


FIGURE 10-12 Action Diagram with Data Detail

First, a location/process matrix is developed to identify major and minor performance of processes in the application (see Figure 10-14). This location/process matrix determines which software is needed at each location to support the functions. The information needed to complete the matrix is provided by the users.

Next, a **data distribution by location matrix** is developed to show creation and retrieval needs by location (see Figure 10-15). This data/location matrix is used to determine the potential *age* of data required by each location. For instance, retrieval data might be down-loaded from a centralized location each day at the close of business, rather than maintained at the remote sites. Created data must be available for creation, and therefore, up-to-date at the creating sites. The information needed to complete the matrix is provided partly from the entity/process matrix from the first data analysis, and partly by the users.

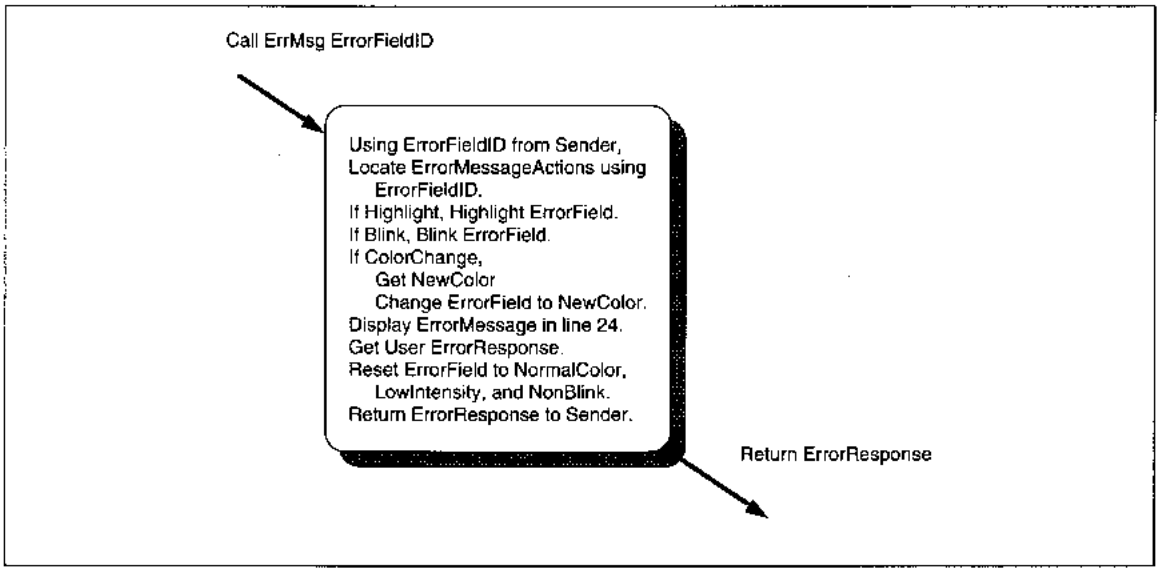


FIGURE 10-13 Procedure Template for Error Message Processing

The next matrix shows **data usage by location** (see Figure 10-16). Recall from above that data can be centralized, vertically or horizontally partitioned, or federated. For instance, a bank branch might create data about customers, but it only accesses information about its own customers on a regular basis. So, for most processing, a vertical partition of the customer database, the branch's customers, could be accessible locally in the branch to speed processing.

The last objective matrix summarizes transaction volume by process by location (from the process/

Function	Location A	Location B	Location C	Location D	Location E
Purchasing	\	X			
Marketing	X	X	\		
Customer Service		X	\		
Sales	X	X	\		
Product Development	X	X	\	\	
Research & Dev.			X	X	\
Manufacturing				\	X

Legend:

X—Major Involvement

\—Minor Involvement

FIGURE 10-14 Process by Location Matrix Example

Subject Data	Location A	Location B	Location C	Location D	Location E
Prospects	All—UR	All—UR			
Customer	All—UR	All—UR			
Customer Orders	All—UR	Subset—Own Products—UR		All—R	All—R
Customer Order History	All—R	All—R	All—R	All—R	
Manufacturing Plans	Subset—own products—R	Subset—own products—R		Subset—own site—UR	All—UR
Manufacturing Goods in Process	Subset—own products—R	Subset—own products—R		Subset—own site—UR	All—UR
Manufacturing Inventory	Subset—own products—R	Subset—own products—R	All—R	Subset—own site—UR	All—UR

U = Update, R = Retrieve

FIGURE 10-15 Data Usage by Location Matrix Example

location table) against each subject database from the data analysis. Two daily transaction volume estimates for each process and location are developed (see Figure 10-17). The first estimate is for transactions that create or update the database. The second estimate is for read-only retrieval processing. Also notice that if no database access is performed by a process, no entry is made. This increases the readability of each matrix.

The analysis of this data is to first identify the location with the highest total transaction count for each database. The example shows a thick box around each such location (see Figure 10-18). If the application were distributed, with centralization of subject databases in one location, the boxes would identify the most likely location for each database. All other transactions, outside the boxes, represent transmission traffic. When the transmission traffic is a high percentage of the total traffic, say over 40%, different types of replication, federation, and partitioning are tried. To analyze the data, first box the transaction numbers for the site(s) representing 50% or more of the total processing. If there is one site boxed in a column, that identifies a centralized database at the location corresponding to the box. We have two of these in the example (Figure 10-18)—the Work in Process and Inventory databases at location *E*. The initial recommendation would be to centralize this data at *E*. Even though *D*'s volume is significantly less than *E*'s, the data usage table shows that each site accesses only its own data, so the option to vertically partition data and provide 'home ownership' could be used to support the business needs.

The other databases all have access competition from two sites (Figure 10-18). Two locations, *A* and *B*, have fairly even usage of the *Prospect* and *Customer*, *Customer Order*, and *Customer History* data. The options from the Data Usage table show that Replication would be the distributed recommendation since the sites both access all data. Customer History processing differs from the other databases in that it is all read-only and it has a much lower volume than the others. Therefore, it could be centralized at either site with an access delay at the other site for retrievals. This option might be chosen if there are hardware configuration differences that favor centralization.

Locations *B* and *E* compete for the *Manufacturing Plan* data (Figure 10-18). Location *B* only retrieves the data, while the location *E* volume of updates is low. The database could either be

Subject Data	Location A	Location B	Location C	Location D	Location E
Prospects	Replicate or Central Copy	Replicate			
Customer	Replicate or Central Copy	Replicate			
Customer Orders	Replicate or Central Copy	Horizontal Partition by Product		Access central copy with delay	Access central copy with delay
Customer Order History	Replicate or Central Copy	Replicate or access central copy with delay		Access central copy with delay	
Manufacturing Plans	Replicate or access central copies with delay	Replicate or access central copies with delay		Subset—own site	Central Copy or Subset—own site with delayed access to D
Manufacturing Goods in Process	Access D and E Databases	Access D and E Databases		Subset—own site	Subset—own site with delayed access to D
Manufacturing Inventory	Access D and E Databases	Access D and E Databases		Subset—own site	Subset—own site with delayed access to D

FIGURE 10-16 Data Distribution by Location Matrix

centralized at *B* to provide fast query access, with delayed access by *E*, or, if politics are involved, the data could be centralized at site *E*, the owner, with delayed retrieval by *B*.

The second part of the analysis is to compute the ratio of data retrieval transactions (D_R) to data update transactions (D_U). If the ratio is greater than one less than the number of locations (L) (or nodes in the network), distribution should be considered (see Table 10-2). In the example, the ratio clearly favors centralization of data (Table 10-2). Keep in mind that centralization here means that each database is stored at *one* location. It does not mean that the databases are all at the *same* location.

If a delay can be introduced for retrieval processing, then the ratio changes. It becomes much easier to argue for distribution. Distribution should be considered when *retrieval* volume is less than the ratio of locations to the delay (D). The delay is for update transactions which are now transmitted in bulk once per period to each other location. In the example, with even a 15-minute delay, the numbers overwhelmingly favor distribution. The rationale for these ratios is given in Table 10-3.

This discussion about distribution is important because it highlights an ethical problem in software engineering. The numbers can be made to argue for distribution regardless of transaction activity. If the transaction ratio of retrievals to updates is large, then the no-delay argument is more likely to favor distribution. If the retrieval to update ratio is less than one, the delay argument is likely to favor centralization.

As an ethical person, you are bound to tell the client about all computations and how the formulae can make either argument.

Last, a subjective list of reasons for and against centralization and distribution is developed for the organization. The exact topic headings for this list are tailored to the company and application environment.

Critical data should be managed centrally
Data is/is not critical to corporation/business unit
Most data can/cannot be stored locally/centrally
Needs/does not need specific DBMS
Requires/does not require larger machine than local sites have
Data ownership is/is not an issue
Data replication needed in one/many locations
Unique data/application in one location
Data affects/does not affect central corporate management
Fast response time important/not important
High availability important/not important
Local staff skilled/unskilled with computers
Application/data security is/is not vital to organization/business unit
Centralized operations is/is not at capacity

	Subject Database						
Location/Function	Prospect	Customer	Customer Order	Customer History	Mftg. Plan	Mftg. WIP	Mftg. Inven.
A							
Customer Service		100 R 20 U	250 R 400 U	5 R	2 R	2 R	
Sales	50 R 20 U	50 R 30 U	150 R 50 U	50 R	2 R	2 R	15 R
Marketing	15 R	5 R	10 R	50 R	2 R		1 R
B							
Customer Service		250 R 50 U	250 R 400 U	50 R	250 R	250 R	250 R
Sales	25 R 20 U	25 R 5 U	10 R 100 U	70 R	2 R	2 R	15 R
Marketing	20 R	10 R	10 R	50 R		2 R	5 R
D							
Manufacturing					50 R 5 U	50 R 250 U	500 R 2,000 U
E							
Manufacturing					100 R 15 U	200 R 2,500 U	500 R 25,000 U

Legend:

U = Create, Update or Delete

R = Retrieve

FIGURE 10-17 Summary Transaction Volume Matrix

	Subject Database						
Location/Function	Prospect	Customer	Customer Order	Customer History	Mftg. Plan	Mftg. WIP	Mftg. Inven.
A							
Customer Service		100 R 20 U	250 R 400 U	5 R	2 R	2 R	
Sales	50 R 20 U	50 R 30 U	150 R 50 U	50 R	2 R	2 R	15 R
Marketing	15 R	5 R	10 R	50 R	2 R		1 R
B							
Customer Service		250 R 50 U	250 R 400 U	50 R	250 R	250 R	250 R
Sales	25 R 20 U	25 R 5 U	10 R 100 U	70 R	2 R	2 R	15 R
Marketing	20 R	10 R	10 R	50 R		2 R	5 R
D							
Manufacturing					50 R 5 U	50 R 250 U	500 R 2,000 U
E							
Manufacturing					100 R 15 U	200 R 2,500 U	500 R 25,000 U

Legend:

U = Create, Update, or Delete

R = Retrieve

FIGURE 10-18 Analysis of Summary Transaction Volume Matrix

Down-loading of yesterday's data would/would not work in local sites
Updates with delay would/would not work in this application environment
Partitioning of data would/would not work in supporting this application
Replication of data would/would not work in supporting this application
Data integrity is/is not paramount to the application
Disaster recovery protection is/is not vital to the application
Operators are/are not at remote sites

Each reason is rated as weak or strong justification of its position. The purpose of list creation is to surface and attempt to objectify objections and arguments from each stakeholder viewpoint regarding distribution of data in the application. An easy analysis is to count the capital and small letters of each type, and compare them. A more elaborate analysis might entail giving a weight to each item and developing a weighted ranking of the central/distributed positions. If the results of this analysis support the objective measures and results, a compelling justification for the result can be developed and presented to user management for approval. If the subjective

TABLE 10-2 Distribution Ratio Formulae

The breakeven point for distribution occurs when

$$\mathbf{D_R/D_U > N - 1.}$$

If the transaction ratio is greater than N – 1, distribute data.

An alternative is to allow a time delay for update transactions with all data replicated at all locations in a network. Then only updates generate network traffic. The breakeven point for distribution occurs with this scenario when

$$\mathbf{D_U < N/TimeDelay} \quad \text{or} \quad D_U * TimeDelay < N$$

If the number of changes is less than the number of nodes divided by the time delay, distribution is favored.

Legend:

D_R = Number of data retrieval transactions

D_U = Number of data update transactions

N = Number of network nodes

D = Total number of data transactions ($D_R + D_U$)

Adapted from Martin (1990), p. 360.

analysis contradicts the objective measures, the user manager/champion might have to do some political maneuvering to obtain the desired result. Of course, if the champion is against the recommendation, the numbers in the traffic table still are useful in determining the size and speed of the machine and telecommunications lines required to service the application's data needs.

ABC Video Example Data Use Distribution and Analysis

ABC's one location simplifies the choices for this analysis. Centralization of data and processes is the only possible choice. For the record, a table of transaction volumes is presented in Figure 10-19.

A secondary issue, if not already decided, is hardware selection. ABC could use a multiuser minicomputer or a LAN. This analysis, too, is simple because ABC is a small company without a high volume of processing. A LAN is cheaper, more easily maintained, more easily staffed, and less costly for incremental upgrades. Therefore, a LAN is the choice. Most multiuser mini-computers allow eight units without major expenditures for an additional I/O controller board. Mini-computers tend to have proprietary operating systems and use packages that tie the user to a given vendor. The strength of

TABLE 10-3 Rationale for Distribution Ratios

If T is the number of traffic units per hour (i.e., transactions), and if all data is centralized at one location (not necessarily the same), then the total traffic units per hours is

$$T_{centralized} = (D_R + D_U) * (N - 1)/N$$

Then, if all data is decentralized (i.e., fully replicated at all user locations), only update transactions generate network traffic, and

$$T_{distributed} = D_U * (N - 1)$$

Fully replicated, decentralized data generates less traffic than centralization if

$$T_{centralized} > T_{distributed}, \text{ or}$$

$$(D_R + D_U) * (N - 1)/N > D_U * (N - 1)$$

This reduces to $\mathbf{D_R / D_U > N - 1}$. This formula means that when the ratio of retrievals to changes ($D_R / D_U = N - 1$) is greater than N – 1, favor distribution. When the ratio is equal to N – 1, either choice is acceptable from a network point of view. When the ratio is less than N – 1, favor centralization.

If changes can be applied with a delay, the equations change. Then the breakeven point occurs when

$$\mathbf{D_R < N/TimeDelay}$$

The greater the delay, the more desirable a distributed strategy can be made to appear.

Legend:

D_R = Number of data retrieval transactions

D_U = Number of data update transactions

N = Number of network nodes

D = Total number of data transactions ($D_R + D_U$)

Adapted from Martin (1990), pp. 360–361.

	Subject Database						
Location/Function	Customer	Video	Item	Customer History	Video History	EOD	Archive
Dunwoody Village Rent/Return	500 R 15 U	500 R 5 U	250 R 400 U	500 R 500 U	500 R 500 U		
Video Maintenance		20 R 5 U	150 R 50 U				
Customer Maintenance	5 R 5 U						
Other					15,000 U/ Once/Mo	1,000 U	15,000 U/ Once/Mo

FIGURE 10-19 ABC Transaction Volume Matrix

multiuser minis is in their added horsepower that allows them to support applications with a high volume of transactions (in the millions per day). A multiuser mini is not recommended here because, for the money, it would be analogous to buying a new Porsche 911 Targa when a used Hyundai would do just fine. To discuss configuration of the LAN, we move to the next section on hardware and software installation.

Define Security, Recovery, and Audit Controls

Guidelines for Security, Recovery, and Audit Control Planning

The three issues in this section—security, recovery, and controls—all are increasingly important in software engineering. The threat of data compromise from casual, illegal acts, such as viruses, are real and growing. These topics each address a different perspective of data integrity to provide a total solution for a given application. Security is preventive, recovery is curative, and controls prove the other two. Having one set of plans, say for security, without the other two is not sufficient to guard against compromise of data or programs. Trusting individuals' ethical senses to guide them in not hurting your company's applications simply ignores the reality of today's world. Morally, *not* having planned for attempts to compromise data and programs, you, the SE, are guilty of ethical passivity that implicitly warrants the compromiser's actions. Therefore, design of security, recovery, and controls should become an integral activity of the design of any application.

The major argument against security, recovery, and audit controls is cost, which factors in all decisions about these issues. The constant trade-off is between the probability of an event and the cost of minimizing its probability. With unlimited funds, most computer systems, wherever they are located, can be made reasonably secure. However, most companies do not have, nor do they want to spend, unlimited money on probabilities. The trade-off becomes one of proactive security and prevention versus reactive recovery and audit controls. Audit controls, if developed as part of analysis and design, have a minimal cost. Recoverability has on-going costs of making copies and of off-site storage. Each type of security has a cost associated with it. Keep the cost issues in mind during this discussion, and try to weigh how you might balance the three methods of providing for ABC's application integrity.

Security plans define guidelines for who should have access to what data and for what purpose. Access can be restricted to hardware, software, and data. There are few specific guidelines for limiting access since each application and its context are different. Those guidelines are listed here:

1. Determine the vulnerability of the physical facility to fire. Review combustibility of construction. Determine adjacent, overhead, and underfloor fire hazards. Determine the status of current fire detection devices, alarms, suppression equipment, emergency power switches, extinguishers, sprinklers, and smoke detectors. Determine the extent of fire-related training. If the facility is shared, evaluate the risk of fire from other tenants.

   Plan for fire prevention and minimize fire threats by using overhead sprinklers, CO_2, or halon. Develop fire drills and fire contingency plans. If no emergency fire plans exist, develop one, reviewing it with the local fire department, and practicing the procedures.
2. Consider electrical/power facilities. Review electrical routing and distribution of power. Review the means of measuring voltage and frequency on a steady-state or transient basis. Determine whether operators know how to measure electrical power and can determine both normal and abnormal states. Define electrical and power requirements for the new application hardware and software. Determine power sufficiency for the computing environment envisioned.

   Correct any deficiencies before any equipment is delivered. For instance, install a universal power supply (UPS) if warranted by frequent power fluctuations or other vulnerabilities.
3. Review air-conditioning systems and determine environmental monitoring and control mechanisms. Evaluate the 'housekeeping' functions of the maintenance staff.

   Correct any deficiencies before any equipment is delivered. For instance, make sure the maintenance staff cleans stairwells and closets, uses fireproof waste containers, and does not use chemicals near computer equipment.
4. Determine the capability of the facility to withstand natural hazards such as earthquakes, high winds, and storms. Evaluate the facility's water damage protection and the facility's bomb threat reaction procedures.

   Design the facility without external windows and with construction to withstand most threats. To minimize bomb and terrorist threats, remove identifying signs, place equipment in rooms without windows, and do not share facilities. To minimize possible storm damage, do not place the facility in a flood zone or on a fault line.
5. Evaluate external perimeter access controls in terms of varied requirements for different times of day, week, and year. Determine controls over incoming and outgoing materials. Evaluate access authorization rules, identification criteria, and physical access controls.

   Plan the security system to include perimeter lights, authorization cards, physical security access, etc. as required to minimize the potential from these threats. Establish procedures for accepting, shipping, and disposing of goods and materials. For instance, shred confidential reports before disposal. Only accept goods for which a purchase order is available.
6. Evaluate the reliability and potential damage from everyday use of terminals and remote equipment from unauthorized employees.

   Plan physical locking of equipment, backup copies of data, reports, etc. to minimize potential threats. Design remote equipment to minimize the threat of down-loaded data from the central database except by authorized users. Usually this is done by having PCs without any disk drives as terminal devices.
7. Evaluate the potential damage from unauthorized access to data and programs.

   Protect programs and data against unauthorized alteration and access.

8. Evaluate the potential damage to the database from unwitting errors of authorized employees.

   Design the application to minimize accidental errors and to be fault tolerant (i.e., recovers from any casual errors).

In general, we consider internal and external physical environment, plus adequacy of data and program access controls. Security evaluation is a common enough event in many organizations that checklists of items for security review are available.[3] An example of general topics in such checklists follows:

- Physical Environment
  - Fire fighting procedures
  - Housekeeping and construction
  - Emergency exits
  - Portable fire extinguisher location and accessibility
  - Smoke detectors located above, under, and in middle of floor areas
  - Automatic fire suppression system
- Electrical Power
  - Power adequacy and monitoring
  - Inspection, maintenance, safety
  - Redundancy and backup
  - Uninterruptible power supply
  - Personnel training
- Environment
  - Air-conditioning and humidity control systems
  - Lighting
  - Monitoring and control
  - Housekeeping
- Computer Facility Protection
  - Building construction and location
  - Water damage exposure
  - Protection from damage or tampering with building support facilities
  - Building aperture protection
  - Bomb threat and civil disorder
- Physical Access
  - Asset vulnerability
  - Controls addressing accessibility
    - Perimeter
    - Building
    - Sensitive offices
    - Media storage
    - Computer area
    - Computer terminal equipment
    - Computer and telecommunications cable

An example of a detailed checklist for building access is provided next.

Facility type: Mainframe, LAN, PC, RJE, Remote, Communications

1. Are entrances controlled by
   _____ locking devices
   _____ guard force
   _____ automated card-key system
   _____ anti-intrusion devices
   _____ sign-in/out logs
   _____ photo badge system
   _____ closed circuit TV
   _____ other _____
2. Are controls in effect 24 hours per day? If not, why?

   _____
3. Are unguarded doors
   _____ kept locked (Good)
   _____ key-controlled (Better with above)
   _____ alarmed (Best with both of above)
4. If guard force, is it
   _____ trained (Good)
   _____ exercised (Better)
   _____ armed
5. Are visitors required to
   _____ sign in and out
   _____ be escorted
   _____ wear distinctive badges
   _____ undergo package inspection
6. If building is shared, has security been
   _____ discussed (Good)
   _____ coordinated (Better)
   _____ formalized (Best)
7. Sensitive office areas, media storage, and computer areas

3 Two IBM-user organizations, GUIDE and SHARE, both have active disaster recovery and security control groups that issue guidelines, checklists, and tutorials on the topic.

_____ Does access authority for each area require management review?

Is access controlled by

_____ locking devices
_____ guard force
_____ automated card-key system
_____ anti-intrusion devices
_____ sign-in/out logs
_____ photo badge system
_____ closed circuit TV
_____ other _____
_____ Are unique badges required?
_____ Do employees challenge unidentified strangers?

8. Control Mechanisms

_____ Do signs designate control/restricted areas?

If locks are used

_____ is key issuance controlled?
_____ are keys changed periodically?

9. Administration

_____ Does management insist on strict adherence to access procedures?

Are individuals designated responsibility for

_____ access control at various control points
_____ authorizing visitor entry
_____ establishing and maintaining policy, procedures, and authorization lists
_____ compliance auditing
_____ follow-up on violations

The probability of total hardware and software loss is low in a normal environment. In fact, the probability of occurrence of a destructive event is inversely related to the magnitude of the event. That is, the threat from terrorist attack might be miniscule, but the damage from one might be total. Each type of threat should be considered and assigned a current probability of occurrence. High probability threats are used to define a plan to minimize the probability. If the company business is vulnerable to bomb threats, for instance, buildings without external glass and without company signs are more anonymous and less vulnerable. Having all facilities locked at all times, with a specific security system for authorizing employees and screening visitors, reduces vulnerability even further.

The major vulnerability is not related to the physical plant in most cases; it is from connections to computer networks. The only guaranteed security against telecommunications invasion is to have all computers as stand-alone or as a closed network with no outside access capability. As soon as *any* computer, or network, allows external access, it is vulnerable to invasion. There are no exceptions, contrary to what the local press might have you believe. Data and program access security protection reduce the risk of a casual break-in to an application. Monitoring all accesses by date, time, and person further reduces the risk because it enables detection of intruders. Encrypting password files, data files, and program code files further reduces the risks; it also makes authorized user access more complex and takes valuable CPU cycles.

The most common security in an application is to protect against unwanted data and program access. Data access can be limited to an entire physical file, logical records, or even individual data items. Possible functions against data are read only, read/write, or write only. Users and IS developers consider each function and the data being manipulated to define classes of users and their allowable actions. Allowable actions are to create, update, delete, and retrieve data. A hierarchy of access rights is built to identify, by data item, which actions are allowed by which class of users. A scheme for implementing the access restrictions is designed for the application.

Backup and recovery go hand-in-hand to provide correction of errors because of security inadequacies. A backup is an extra copy of some or all of the data and software, made specifically to provide recovery in event of some disaster. Recovery is the process of restoring a previous version of data or application software to active use following some damage or loss of the previously active copy.

Research by IBM and others has shown that companies go out of business within six months of a disaster when no backup copies of computer data and programs are kept. In providing for major disasters, such as tornados, **off-site storage**, the storing of backup copies at a distant site, is an integral part of

guaranteeing recoverability. Off-site storage is usually 200+ miles away from the computer site, far enough to minimize the possibility of the off-site facility also being damaged. Old salt mines and other clean, underground, environmentally stable facilities are frequently used for off-site storage.

The disasters of concern in recovery design are user error, unauthorized change of data, software bugs, DBMS failure, hardware failure, or loss of facility. All these problems compromise the integrity of the data. The most difficult aspect of recovery from the first three errors is error detection. If a data change is wrong but contains legal characters, such as $10,000 instead of $1,000 as a deposit, the only detection will come from audit controls. If a data change is wrong because it contains illegal characters, the application must be programmed to detect the error and allow the user to fix it. Some types of errors, such as alteration of a deposit to a bank account or alteration of a payment to a customer, should also have some special printout or supervisory approval required as part of the application design to assist the user in detecting problems and in monitoring the correction process. DBMS software frequently allows transaction logging, logging of before and after images of database changes and assisted recovery from the logs for *detected* errors.

DBMS failure should be detected by the DBMS and the bad transaction should automatically be 'rolled-back' to the original state. If a DBMS does not have a 'commit/roll-back' capability, it should not be used for any critical applications or applications that provide legal, fiduciary, or financial processing compliance. *Commit* management software monitors the execution of all database actions relating to a user transaction. If the database actions are all successful, the transaction is 'committed' and considered complete. If the database actions are not all successful, the commit manager issues a *roll-back* request which restores the database to its previous state before the transaction began, and the transaction is aborted. Without commit and roll-back capabilities, partial transactions might compromise database integrity.

Other data and software backup procedures are either full or incremental. A **full backup** is a copy of the entire database or software library. An **incremental backup** is a copy of only changed portions of the database or library. A week's worth of backups are maintained and rotated into reuse after, for example, the fifth day. To minimize the time and money allocated to backup, incremental procedures are most common. A full backup is taken once each week with incremental backups taken daily. An active database would be completely backed-up daily with one copy on-site for immediate use in event of a problem. Regardless of backup strategy, an extra copy of the database is created at least once a week for off-site storage.

The extensiveness of backup (and recoverability) is determined by assessing the risk of not having the data or software for different periods (see Table 10-4). The less the tolerance for loss of access, the more money and more elaborate the design of the backup procedures should be. The severity of lost access time varies, depending on the availability of human experts to do work manually and the criticality of the application. In general, the longer a work area has been automated, the less likely manual procedures can be used to replace an application, and the less time the application can be lost without

TABLE 10-4 Backup Design Guidelines for Different Periods of Loss

Length of Loss	Type of Backup
1 Week or longer	Weekly Full with Off-site storage
1 Day	Above + Daily Incremental/Full
1 Hour	Above + 1 or more types of DBMS Logging
15 Minutes or less	Above + All DBMS Logging Capabilities: Transaction, Pre-Update and Post-Update Logs

severe consequences. The less important an application is to the continuance of an organization as an on-going business, the less critical the application is for recovery design. An application for ordering food for a cafeteria, for instance, is not critical if the company is an oil company but is critical if the company is a restaurant.

To define backup requirements, then, you first define the criticality of the application to the organization, and the length of time before lost access becomes intolerable. Based on those estimates, a backup strategy is selected. If the delay until recovery can be a week or more, only weekly full backups with off-site storage are required. If the delay until recovery can be one day or less, then, in addition to weekly backups, daily backups should be done. If the recovery delay can be only an hour, the two previous methods should be supplemented with one or more types of DBMS logging scheme. Finally, if a 15-minute recovery delay is desired, all types of DBMS logging, plus daily and weekly backups should be done.

Last, we consider audit controls which provide a record of access and modification, and prove transaction processing for legal, fiduciary responsibility, or stakeholder responsibility reasons. Audit controls allow detection and correction of error conditions for data or processing. As new technologies, greater dependence on ITs, and interrelated systems that are vulnerable to telecommunications attacks all increase, business emphasis on controls also increases. In manual systems of work, control points are easily identified; procedures are observable, errors can be reconstructed, and controls applied by humans. In automated applications, the application *is* the solution, nothing is directly observable, and complexity of functions makes identification of control points increasingly complex.

A **control point** is a location (logical or physical) in a procedure (automated or manual) where the possibility of errors exists. Errors might be lack of proper authorization, misrecording of a transaction, illegal access to assets, or differences between actual and recorded data. Control points are identified during design because the entire application's requirements should be known in order to define the most appropriate control points. Controls are specified by designers in the form of requirements for program validation. For instance, controls for the validity of expense checks might be as follows:

1. Only valid, preauthorized checks can be written.
2. Check amounts may not exceed authorized dollar amounts.
3. Checks may not exceed the expense report total amount.

Application audit controls address the completeness of data, accuracy of data, authorization of data, and adequacy of the audit trail. Detection of processing errors is either through edit and validation checks in programs, or through processing of redundant data. Examples of **controlled redundancy** of data include double entry bookkeeping, cross footing totals and numbers, dual departmental custody of replicated critical data, transaction numbering, and primary key verification. Edit and validation rules are designed to identify all logical inconsistencies as early in the process as possible, before they are entered into the database.

ABC Video Example Security, Backup/ Recovery, and Audit Plans

To design ABC's security, we first review the physical plant and recommend changes to the planned computer site to provide security. The six threats are considered, but the byword from Vic in discussing the possibility of changes is "be reasonable." So, if there is a 'reasonable' chance that a problem will occur, we will recommend a reasonable, and low cost, solution to the problem.

Moving from most to least serious, we consider the six types of threats to application security: location failure, hardware failure, DBMS failure, software failure, hacker change, and user error. For each threat, we consider the potential of occurrence for ABC, then devise a plan to minimize the potential damage. All threats and responses are summarized in Figure 10-21.

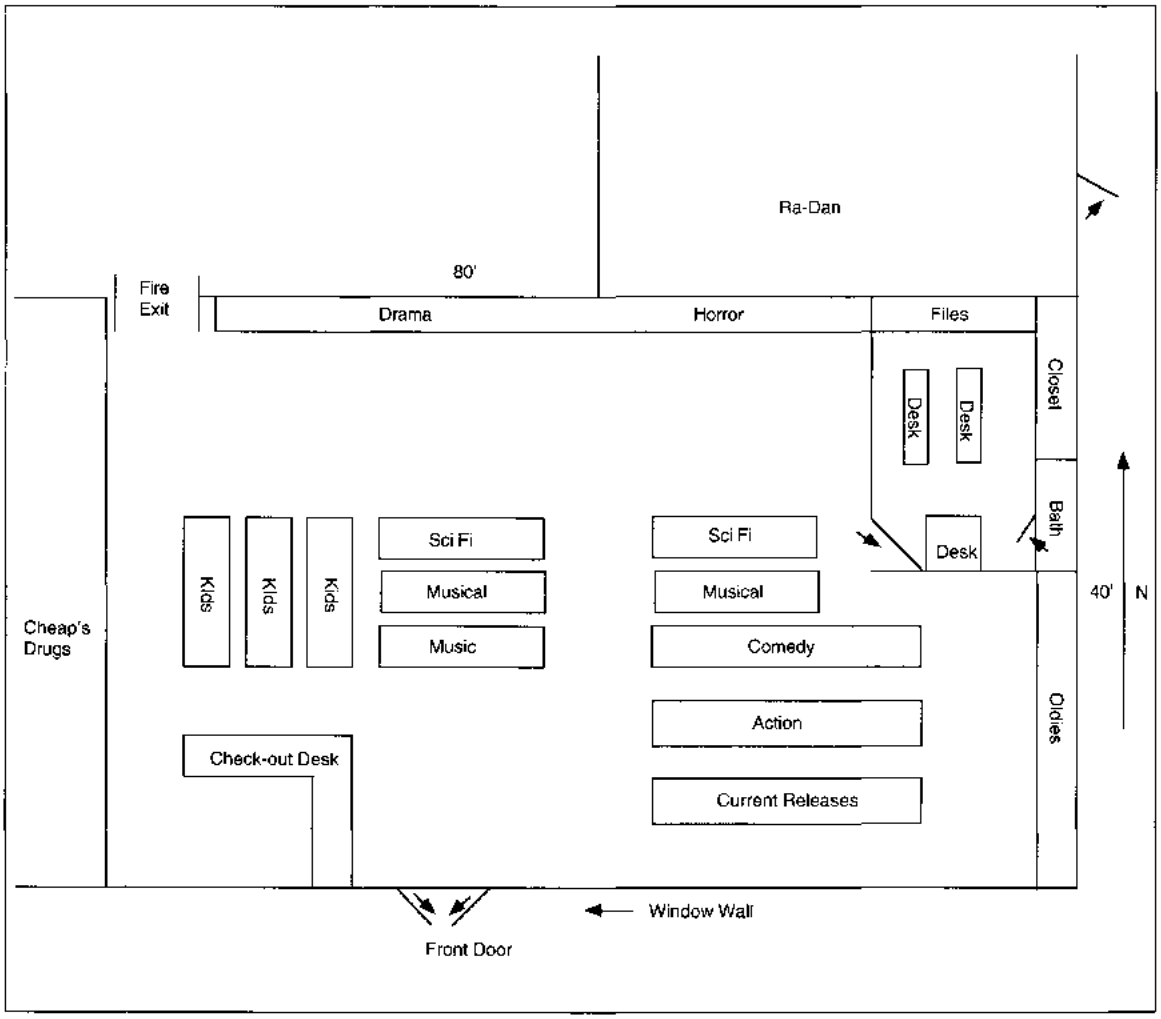


FIGURE 10-20 ABC Current Physical Plant

First, we review the physical plant and relate it to location and hardware failures. ABC Video is located in suburban Atlanta, Georgia, 300 miles from the ocean and 25 miles from the nearest large lake. The company is located in a mall, the Dunwoody Village, a clustering of small shops and offices in open-square buildings containing a plaza in the middle of the square. The company occupies 3200 square feet of 80' × 40' space in the southeast corner of Building A. The adjoining spaces are occupied by Cheap's Drugs and Ra-Dan Hair Salon. A schematic of the space is shown in Figure 10-20. The northeast corner of the area (abutting Ra-Dan's) contains a 12' × 16' office which contains two desks, one supply closet, and a bathroom. The office has no windows and can be locked, although it is frequently empty and unlocked. The supply closet has double doors which do not currently have a lock.

The clerk's checkout counter is near the customer doors on the south side of the building in the western corner. The counter is an 'L' shape with the entry on the short side. A fire door, equipped with an alarm bar, is located in the northwest corner of the area and opens on a short alley behind the building.

Location failure usually results from violent weather, terrorist attacks, or government takeover. The chance of violent weather is the only potential major problem in the area. Tornadoes occur in the area regularly. The expectation is that there is a 20% chance of tornado damage some time in the next 10 years (see Figure 10-21). Tornadoes also imply strong thunderstorms which are common to the area. The chance of damage from a storm is about 30% within five years to the windows, and about 65% within two years for lightning to cause electrical spikes.

The response to location threats is to provide off-site backup of all information, with the site far enough away that it is unlikely to be affected by the same storm (see Figure 10-21). Vic should investigate the possibility of closing in the window wall in the southeast side of the building to minimize storm damage. He can also install lightning rods on the roof of the building to dissipate lightning when it hits.

The next category of problems relate to the hardware selected for the rental/return application. Vendor-cited reliability is 99 years mean time between failure (MTBF) for individual components. When the components are considered as a whole, the probability of component failure is once in two years (see Figure 10-21). The current plan is to have an extra PC in the office that could be moved to the front desk if needed. A hardware service contract with a local company to provide response within 24 hours is recommended.

The planned server location is near the bathroom in the northeast corner of the area. The toilet has a history of overflows during wet spring months. Because of the way the office was constructed, the water is confined to a small area but almost always runs into the supply closet and has been as high as one foot. The probability of component failure to file server and/or disks from water due to toilet overflow is 50% in two years. The answer to this problem is simple, but expensive: Build a new area, specifically for the computer, away from the toilet area to reduce this probability to near zero. Ideally, if the windows are closed in, the office could be moved to the front of the building and the old office removed. A new enclosure for the toilet facilities could be added or the toilet could also be rebuilt in the new location with whatever precautions are needed to preclude the spring overruns.

There is another problem with the planned server location. The planned location—the supply closet—has no ventilation. If the closet doors are open, ventilation for the office is sufficient for the planned equipment, but, ideally, the server closet doors should be locked. If the doors were locked, the probability of server failure due to lack of ventilation is 50% in two years. The solutions possible are to build a new area for the server equipment, or to add ventilation to the planned area to reduce this probability to near zero. Both solutions should be presented to Vic for his decision.

Less serious problems stem from the building location. Glass windows that run along 60' of external front wall and the drop ceiling are accessible from neighboring companies. Theft and break-ins are somewhat common in the area, but the probability of a break-in is 50% in 10 years. Most burglars are looking for money, but some might maliciously tamper with the computer equipment. Therefore, the probability of computer damage during a break-in is 60% according to police estimates.

The recommendations to minimize theft have to address the easy access to the company through windows and ceiling. If the office remains in its current location, a security system with movement sensors in the ceiling and glass-breakage sensors on all windows should be added (whether or not the computer is installed). Long-term, Vic should investigate the possibility of closing-in some or all windows to improve security of the company.

Next, because of the location of the checkout desk at the front of the building, the ability of clerks to monitor approaches to the office is low due to limited visibility. Further, theft of tapes is possible because clerks cannot see down all aisles without moving away from the desk area. For application security, we are concerned with office access; but, as professionals, we can make recommendations that will improve Vic's ability to reduce general theft as well. An easy, but somewhat expensive solution is to move the checkout desk to the center of the floor and assign surveillance duties to clerks. Even if the desk is not moved, mirrors installed in the corners of the

Finding	Recommendation
Location failure—Probability of tornadoes 10% in 10 years. Probability of strong storms causing damage to windows is about 15% within two years. Probability of lightning causing electrical spikes is 15% within two years.	Select off-site storage facility no closer than 200 miles. Investigate closing in the front windows, at least the contiguous 40 feet of windows on the southeast corner. Install lightning rods on the roof.
Hardware failure—Vendor-cited reliability is 99 years MTBF for each component. The probability of component failure is once in two years for some network component.	Move the extra PC in the office to the front desk if needed. A hardware service contract with a local company to provide response within 24 hours is recommended.
Hardware failure from external reasons—Planned server location is near bathroom with history of periodic overflows. Probability of component failure to file server and/or disk is 50% in two years.	Build a new area to reduce this probability to near zero.
Hardware failure from external reasons—Planned server location is a closet in the office area without any ventilation. Probability of server failure is 50% in two years.	Build a new area or add ventilation to the planned area to reduce this probability to near zero.
Hardware failure from external reasons—Current location has glass windows along 60' of external front wall and a drop ceiling accessible from neighboring companies. Probability of break-in is 30% in 10 years; probability of computer damage during a break-in is 60%.	If the office remains in its current location, add security system with movement sensors in the ceiling and glass-breakage sensors on all windows. Long-term, investigate the possibility of closing-in some or all windows, moving the office to the front of the building (away from plumbing).
Physical location vulnerabilities—Ability of clerks to monitor approaches to the office is low because of desk location and limited visibility.	Move the clerks' desk to the center of the floor and assign surveillance duties to clerks. Install mirrors in corners of room to allow monitoring of customers' actions.
DBMS failure—Vendor-stated reliability is two years MTBF. This is one of the best on the market, but each new release is unstable for at least six months.	Do not install latest releases until thoroughly tested using regression test package. Negotiate with vendor for data access software in event of DBMS failure. Include this software access in the vendor contract.
DBMS failure—Other reasons (e.g., electrical spike). Probability is 100% that electrical surges will occur, since they are common in the summer months. Probability of brownouts with reduced power are 30% in two years.	Install a surge protector on the entire ABC electrical system to accommodate spikes (cost is about $100). Install surge protectors on each individual outlet used by computer equipment to further protect the equipment since whole system protectors do not guarantee integrated chip safety in any devices. Install a limited, inexpensive, UPS to provide emergency power in event of electrical failure and for limited use during brownouts (cost about $1,000).

FIGURE 10-21 Security Review Findings and Recommendations

Finding	Recommendation
Software failure—Application failure due to software defects should be less than once in 15 years after the first three months. During the first three months of operation, the probability of application failure is about 75%; no more than one is expected.	The application is designed for 15-minute recovery of all data and programs. Loss of transactions in process will always occur with any failure; they will have to be reentered. Program problems will be fixed within one business day. Any lost transactions will be reentered free of charge by Software Engineers Unlimited.
Hacker change—Outside user access to the system should be zero since no telecommunications capabilities are planned. However, the untended server and occasional lack of clerks at the desk area may provide a local hacker enough time to access and modify the system.	Install security precautions listed above: security mirrors, move desk, assign clerks monitoring responsibility. *Always* lock office door; *always* lock file server door.
User error—The use of computer novices as clerks guarantees user error. Probability is 100% within one week of system operation.	Restrict data and process access to those required to perform each job. Design application to withstand any casual error—hitting any key on keyboard, scanning any bar code type, etc. A report of such errors can be created and printed on demand by Vic to allow retraining (or other action) for repeated errors by one user. Application design also includes validation of all fields such that only valid data can be in the database. On-demand reports of new customer and video entries will allow Vic to monitor the typing skills of employees. New-hire orientation and new-hire mentors should be used to stress the importance of data accuracy.

FIGURE 10-21 Security Review Findings and Recommendations (*Continued*)

room would allow clerks to monitor customers' actions. Both recommendations are made with the understanding that the mirrors should be installed whether or not the desk is moved.

After physical issues are evaluated, we next look at software security and reliability. Vendor-stated reliability for the planned DBMS is two years MTBF. This SQL software is one of the best on the market, but each new release is unstable for at least six months, and those instability figures are not in the MTBF estimates. The company routinely disclaims any responsibility for new release errors and loss of data or processing to using companies. The DBMS does stabilize and is usually reliable after a six-month trial period for each new release. The simple solution to this problem is that unless a feature of a new release is needed, no change from the current stable version should be made. In addition, no software, whether vendor package or customer designed, should be allowed into production use until it is thoroughly tested using the application regression test package that will accompany the system.

A secondary problem with DBMS errors is that, if the DBMS fails, there is no other way to access the data. Part of the contract negotiation should include discussion of such software for the vendor to provide in event of DBMS failure. Other companies have successfully received such commitments from this vendor, although it is not volunteered. Such data

access software should be included in the vendor contract.

Additional problems that might cause DBMS failure are electrical surges and brownouts due to uneven service in the area. Surges generally occur during the summer months when equipment comes on-line to service air-conditioning in the area. The probability of surges is 100% based on local electrical company history. The probability of brownouts with reduced power is 30% within two years, also using electrical history as the basis for the estimate. Problems from both causes can be minimized by a surge protector on the entire ABC electrical system which shuts down power if a particularly large surge is experienced. In addition, one surge protector for each outlet should be installed to further protect the equipment since whole system protectors do not guarantee integrated chip safety. Finally, a limited, inexpensive, uninterrupted power supply (UPS) should be installed to provide emergency power in the event of electrical failure and for limited use during brownouts to supplement reduced electricity from the local provider.

We consider application software failures next. Failure due to software defects should be less than once in 15 years after the first three months of operational use. During the first three months of operation, the probability of application failure is about 75%; no more than one is expected. The application is designed for 15-minute recovery of all data and programs. Loss of partial transactions will always occur with any failure; they will have to be reentered. Program problems will be fixed within one business day. Any lost transactions will be reentered free of charge by Software Engineers Unlimited (Mary's company).

Outside user access to the system should be zero since no telecommunications capabilities are planned. However, the untended server and occasional lack of clerks at the desk area may provide a local hacker enough time to access and modify the system. If the physical security precautions recommended above are provided, such hacker break-ins would be nearly impossible. Therefore, at a minimum the precautions for security mirrors, assigning clerks monitoring responsibility, and locking the office and file server doors should be implemented (see Figure 10-21).

Finally, the use of computer novices as clerks guarantees user errors. The probability of user errors is 100% within one week of system operation. To prevent any application or DBMS damage from user errors (inadvertent or otherwise), the first line of defense is to restrict what users may do and the data they may access as a way to prevent errors. Each job should be defined and a security access scheme developed to allow access to all processes and data required for the job, *and nothing more.*

Second, the application should withstand any casual error—hitting any key on keyboard, scanning any bar code type, and so on. If required, a report of such errors can be created and printed on demand by Vic to allow retraining (or other action) for repeated errors by one user. Application design also includes validation of all fields such that only valid data can be in the database. Such checks are not possible for alphanumeric data, however, so on-demand reports of new customer and video entries will allow Vic to monitor the typing skills of employees.

Application training will use computer-based training (CBT) in entering application data. The CBT will use simulated transactions and should minimize the user errors if taken seriously by clerks. New-hire orientation should include discussion of the importance of accuracy of work, especially with the computer. Further, new hires should be assigned a more senior 'mentor' for learning the application after training.

After disaster recovery is planned, application security must be developed. From the recovery plan, we know that each job should be evaluated to determine the data and processing requirements of the position. ABC jobs evaluated include clerks, owner, and accountant. The owner should be allowed to do any functions on the application and system that he desires. However, many owners do not *want* to become the chief user of the computer. When asked, Vic's reaction is, "Does this mean I can never take a vacation? Do I have to be here in the morning and at night? If so, define a new position that can do most of my functions, just not delete data!" So the position of chief clerk is also considered.

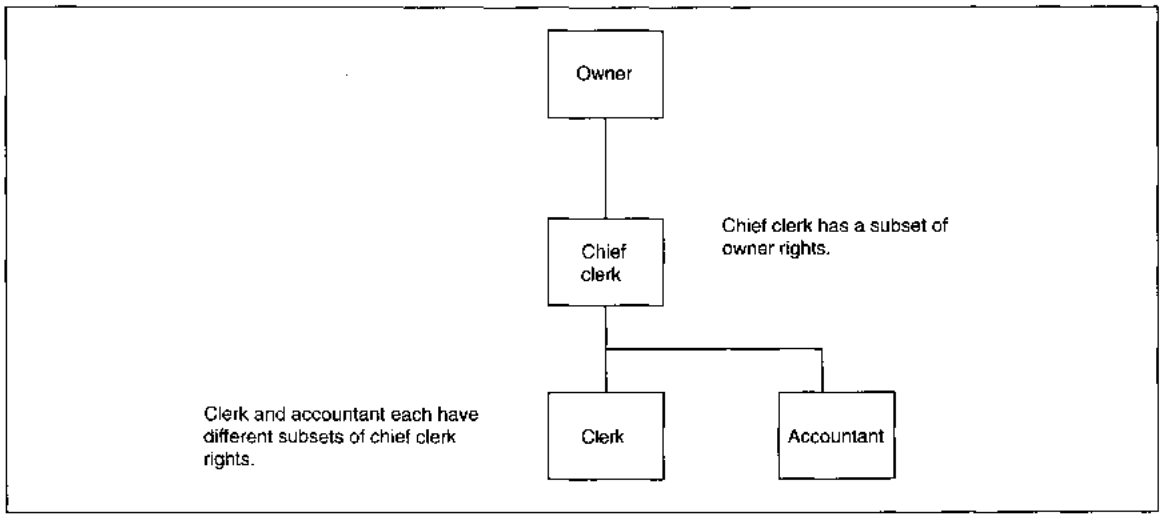


FIGURE 10-22 ABC Data Security Hierarchy of Access Rights

The owner should be the lead person and still be allowed to perform all functions, access all data, and provide security password changes, and so on (see Figure 10-22). The chief clerk, according to Vic's wishes, has all of those functions except deleting information (see Table 10-5). If there were sensitive data in the system, more discussion of the chief clerk's duties and access rights might take place. The clerks have access rights to rent and return videos, and to create and update customers and videos. Finally, the accountant has limited read-only access to several files.

Backup and recovery are considered next. First we decide the maximum tolerable time loss for a computer outage, then select the backup scheme that best fits the time loss maximum. The rental/return application is critical to ABC's ability to conduct business. Vic knows that when he moves all production work to the computer that the clerks will quickly forget the manual way of conducting business. Also, we know that if the databases are not kept up to date, the system is next to useless because the clerks won't know whether to look at manual or automated files for returns, fees, and so on. Therefore, the maximum outage should be less than 15 minutes with recovery of all fully complete transactions. Even at 15 minutes, if an outage were to occur during a peak time, as many as four transactions could need to be reentered and as many as 15–20 transactions would be queued for entry upon system return to production. Ideally, the system should be functional during all business hours.

The recovery requirements imply the most backup protection possible. From Table 10-4, a 15-minute recovery requirement means the use of weekly full backups with off-site storage, daily backups, and logging for transactions, preupdate data items and postupdate data items. Therefore, these are the backup and recovery requirements.

Requirements: Application and system availability during all store open hours, with no more than 15 minutes of down-time from failures of any type.
Backups: Transaction, preupdate, and postupdate logs
Transaction logs maintained one week until weekly backups are verified. Pre- and postupdate logs maintained for 72 hours.
Daily complete database backups with on-site copy plus off-site storage at owner's home.

TABLE 9-5 ABC User Classes and Access Rights

File/Function	Owner	Chief Clerk	Clerk	Accountant
Customer				
Create	X	X	X	
Retrieve	X	X	X	X
Update	X	X	X	
Delete	X			
Video				
Create	X	X	X	
Retrieve	X	X	X	X
Update	X	X	X	
Delete	X	X		
Open Rentals				
Create	X	X	X	
Retrieve	X	X	X	X
Update	X	X	X	
Delete	X			
Video History				
Create				
Retrieve	X	X	X	X
Update	X			
Customer History				
Create				
Retrieve	X	X	X	X
Update	X	X		
Startup	X	X		
Shutdown	X	X		
End Of Day				
Create	X	X		
Retrieve	X	X		X
Delete	X	X		
Initiate End of Month Process	X	X		

Paper copy of transactions maintained for one calendar year in accountant's office.

Weekly complete disk backups with on-site copy plus off-site storage at owner's home and a third copy at

Disaster Prevention Storage
321 Maple Ave.
Somewhere, OK
(618) 123-1234

If ABC's application processed millions of transactions each day, we would do further analysis of the cost of backup and recovery, but here that is not necessary.

Finally, we need to decide about audit controls as summarized here:

Data accuracy and completeness—All edit checks possible will be used as data are entered to prevent errors from entering the

system. Sight verification by clerks and customers will be used to verify alphanumeric information.

Rental transaction accuracy can be verified by customers' signing for all monetary transactions. In case of discrepancy, transaction logs and historical paper copies of transactions can be consulted.

Data authorization—Security controls will provide sufficient authorization for data processing. Only the owner is authorized to perform any delete functions on customer, video, and open rental data. No delete functions for history records are provided.

User ID, date, and time of user to last change data will be maintained in Customer, Video, and Open Rental databases.

Audit trail—A paper trail of receipts should be maintained by the accountant for each calendar year. This is a sufficient trail since ABC is a cash business without any accruals.

Nonmonetary transactions (e.g., return of on-time tapes), have no paper audit trail. If a question about a tape return arises, the database can be checked to verify the information.

All edit checks possible should be used as data are entered to prevent errors from entering the system. To ensure complete editing, we review the data dictionary to check that all nonalphanumeric fields have edit and validation criteria.

On names, addresses, and other alphanumeric fields, little verification can be performed automatically. What cannot be done automatically should be done manually. Procedures for operators should be developed to document clerical 'sight verification' and customer verification standards. An example of such a procedure that would be part of the user manual is shown as Figure 10-23. **Sight verification** means that the person entering information into the computer reads the monitor to verify the accuracy of the information he or she entered. The user, then,

These paragraphs would be part of the user procedures:

Customer Maintenance

. . .

When customers are being added to the system, the clerk should read back all information as shown on the screen to verify its accuracy, as the computer cannot verify mixed alphabetic and numeric information.

. . .

Video Maintenance

. . .

When videos are being added to the system, the clerk should compare all information shown on the screen with the original printed information to verify its accuracy, as the computer cannot verify mixed alphabetic and numeric information.

. . .

Rent/Return Processing

. . .

Users should be encouraged to check the information on the printed rental before they sign it to verify that it is correct.

. . .

FIGURE 10-23 User Sight Verification Procedure

is responsible for data integrity of items that cannot be **computer verified.**

Rental transaction accuracy will be verified by customers' signing for all monetary transactions. In case of discrepancy, transaction logs and historical paper copies of transactions can be consulted. If many discrepancies persist (more than one per week), a special history file of transactions can be added to the application to speed the transaction look-up process.

Security controls can be designed to provide sufficient authorization for data processing. The security scheme should be developed to serve two goals: to provide data access and to provide function access to those who need it. To require several layers of security checking for a simple application does not make sense and wastes clerical time. So, once again the KISS (Keep It Simple, Stupid) method of one security access scheme is best. User ID, date, and time of user to last change data will be maintained in Customer, Video, and Open Rental databases. These attributes are added to affected database relations.

To minimize the extent to which damage can be done to data, only ABC's owner should be authorized to perform any delete functions on customer, video, and open rental data. No automated delete functions for history records are provided without circumventing the application completely. Changes to files will always be somewhat traceable because the historical record will reflect activity. If unauthorized file changes are thought to be a problem, Vic can always request a browsing capability for any of the transaction logs to check on problems.

A manual audit trail should be used for ABC to conserve computer resources. All monetary transactions can be reconstructed through a paper trail of receipts maintained by the accountant. The receipt form is a two-ply preprinted form on which all monetary transactions are printed. For rentals, customers sign the form as proof of rental responsibility. Paper records should be maintained for one calendar year in the accountant's office; this is sufficient since ABC is a cash business without any accruals. If a tape audit trail were to be necessary at some time in the future, it can be added to the system easily.

Nonmonetary transactions (e.g., return of on-time tapes), have no paper audit trail. If a question about a tape return arises, the user ID, date, and time of the return will be on the database and can be checked to verify the information.

Develop Action Diagram

Guidelines for Developing an Action Diagram

An action diagram is a diagram that shows procedural structure and processing details for an application. It is built from the process hierarchy and process data flow diagram developed during IE analysis (see Figure 9-45 for ABC's PDFD). The diagram uses only structured programming constructs to convert the PDFD into a hierarchy of processes that can be divided into programs and modules. First we discuss the components of the diagram, then we discuss how to build an action diagram from the process hierarchy and PDFD.

Action diagrams use different bracket structures to depict the code elements in an application. Basic structured programming tenets—iteration, selection, and sequence—are all accommodated with several variations provided. As Figure 10-24 shows, a **sequence bracket** is a simple bracket. It is optionally identified with a process name and ended with the term ENDPROC to represent a program module consisting of a sequence of instructions.

When a module is designed and detailed in another document or diagram, a rounded rectangle containing the module name is drawn between the brackets (see Figure 10-25). When the module is not yet defined in detail, a rounded rectangle with question marks down the right side is shown. Reusable

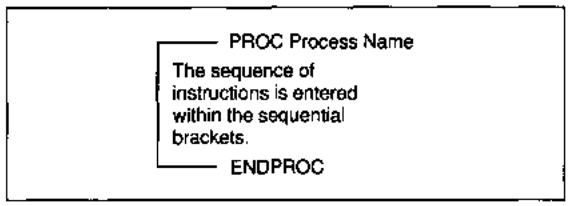


FIGURE 10-24 Simple Sequence Bracket Format

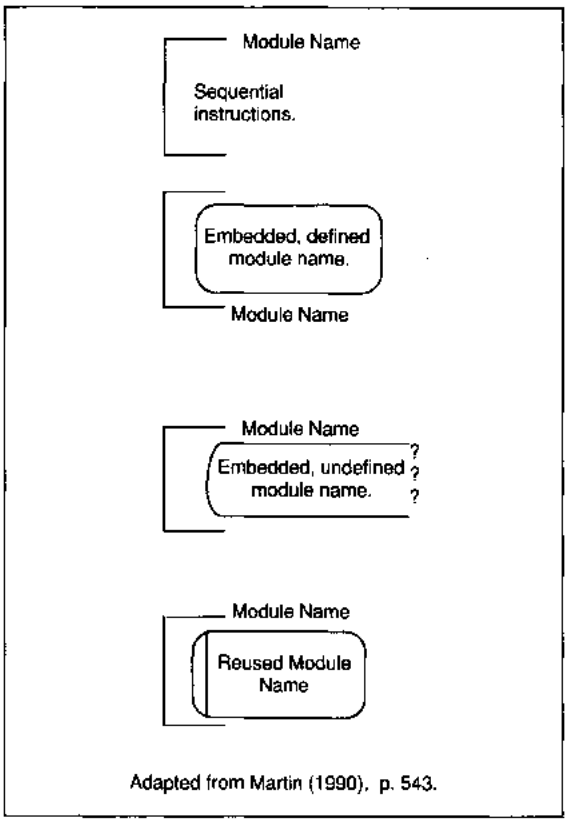


FIGURE 10-25 Module Designation Format

modules are drawn with a vertical bar to represent reuse.

Selection of modules from the PDFD is shown by a **selection bracket** (also called a **condition bracket**) which begins with an *IF* condition and ends with the term *ENDIF* (see Figure 10-26a). If the conditional statement has multiple conditions, two other options are allowed. The condition can be stated as an *IF* statement with one or more *ELSE* conditions (see Figure 10-26b), or a condition can be stated as a mutually exclusive selection list as in Figure 10-26c; this selection list is eventually translated into an *IF* statement.

Repetition is shown with a double bracketed figure. The **repetition bracket** name begins with either *DO* or *DO WHILE* + *condition* (see Figure 10-27). The bracket ends with either an *UNTIL* + *condition* (Figure 10-27a), or *ENDDO* (Figure 10-27b). *DO WHILE* implies that the condition is checked *before* the conditional statements are executed. Do while processing may occur zero times. Conversely, *DO UNTIL* implies that the condition is checked *after* the lower statements are executed. Do until processes occur at least once.

Miscellaneous items include *goto*, *exit*, and concurrency identification. A *goto* is shown by an arrow leaving one level and pointing to the line for the destination level with a goto statement and destination at the right of the arrow (Figure 28a).

An *exit* is shown as an arrow leaving one level and pointing to the line for the destination level with the word *exit* at the right of the arrow (Figure 28b). Unless an exit destination is named with the exit, exit *always* means that the calling module is the exit destination. For example, if *Rent/Return* calls *CustomerAdd*, the exit from *CustomerAdd* returns to *Rent/Return*. Further, if *CustomerMaint* calls *CustomerAdd*, the exit from *CustomerAdd* returns to *CustomerMaint*. That is, the calling module, regardless of what it is, is the return module.

Processes can be sequential or concurrent. **Concurrent processes** execute at the same time. There are two types of concurrent processes: independent and dependent. **Independent concurrent processes** are those which execute at the same time but do not synchronize their process completion. For example, when *Process Payment and Compute Change* is complete in ABC's application, printing and file updates of several types could all be concurrent. If there is no checking on the success of their completions with subsequent action for any failures, these processes are independent. Independent concurrency is shown on the diagram by an arc which connects the module brackets (Figure 10-28). **Dependent concurrent processes** are those which must be synchronized to coordinate further application actions. Dependent concurrency is shown on the diagram by an asterisk (or some other special character) on the arc connecting the modules (Figure 10-28d). Dependent concurrent processes require the development of a synchronization module, if not already in the application, to ensure complete, accurate processing.

Now that you know the bracket symbols used to define action diagrams, we move to discuss the steps

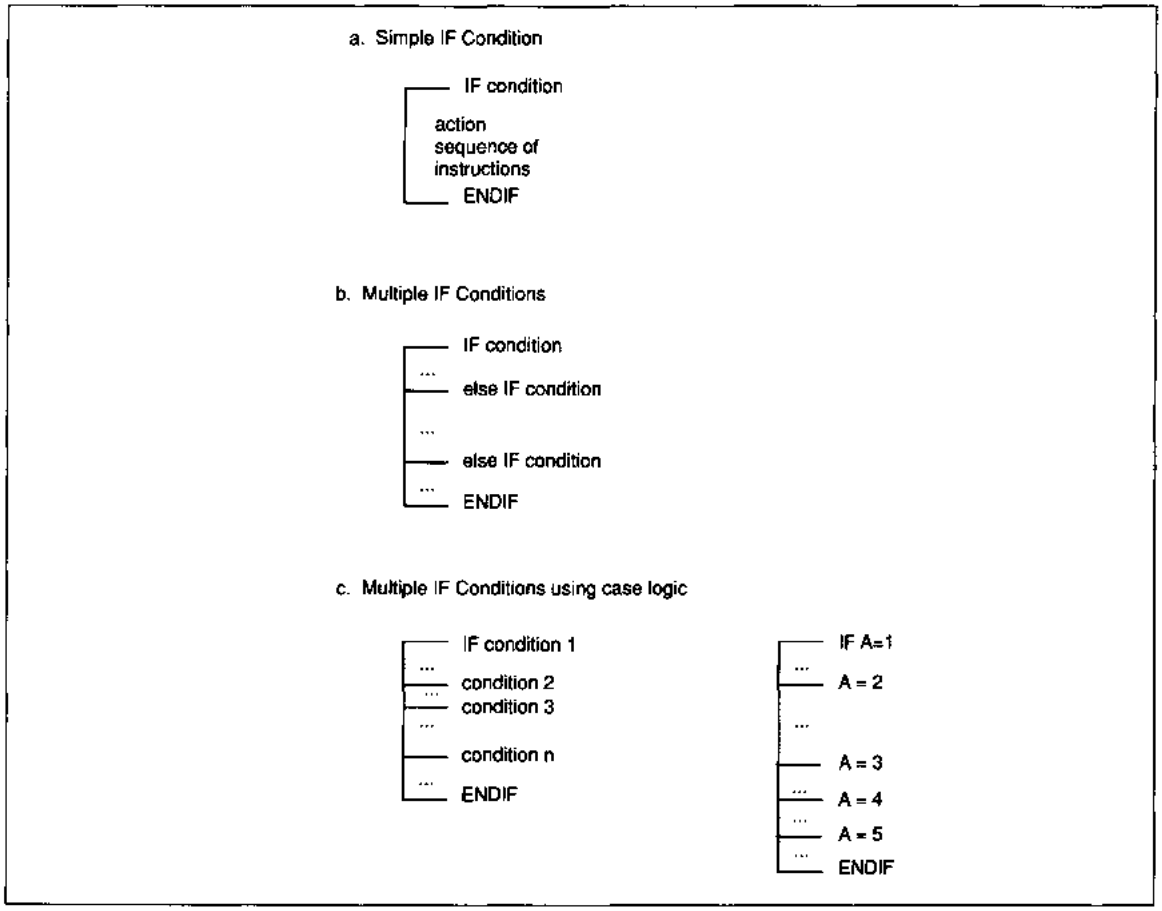


FIGURE 10-26 Conditional Bracket Design Formats

to developing one. The steps to define an action diagram are to translate processes into levels of action using structured constructs, design modules, perform reusability analysis, decide module timing, add data to the diagram, and optionally, add screens to the diagram.

The first step is to translate processes into levels of action. The first-level diagram is developed from the process hierarchy diagram to identify the major activities being performed by the application. The activities themselves are added to the diagram as they are written on the hierarchy diagram. The structured constructs should identify sequence and any selection or conditional processing relating to the activities. Most often, when the diagram is begun at the activity level, the alternative processes are mutually exclusive. When the diagram starts at the process level (Figure 10-29), any construct might apply. The example shows a mutually exclusive selection from among the three alternatives.

Now we shift to the process data flow diagram (Figure 10-30) to add process details to the action diagram. Remember that the processes on the PDFD must match exactly the processes on the hierarchic decomposition diagram. We use the PDFD to translate the structural relationships between the processes correctly. The **structural relationships** are on the PDFD and not on the decomposition; they refer to the sequential, conditional, and repetitive relationships between processes.

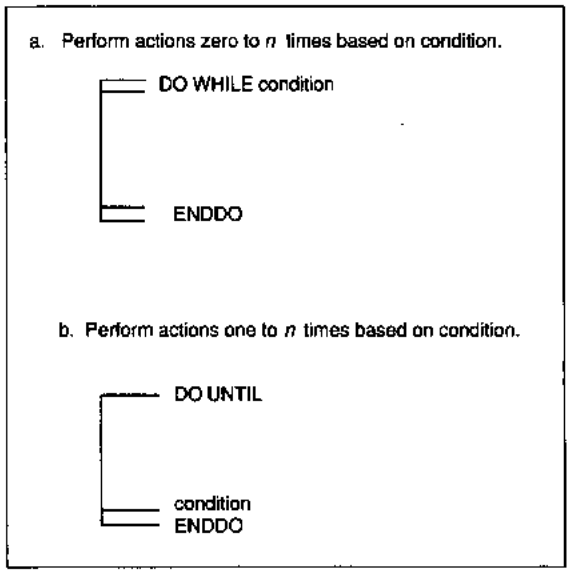


FIGURE 10-27 Repetition Bracket Design Formats

In developing the second-level action diagram, we first add the processes, in sequence, from the PDFD. Then the brackets are drawn to reflect the sequential, conditional, and repetitive structural relationships. In the example (Figure 10-31), the main processes are *Identify Item and Vendor*, *Sort by Vendor and Item*, *Get Price*, *Create Order*, and *Mail Order*. Between these processes, there are two repetitive blocks: one based on *New Releases*, and the other based on *Vendors* (see Figure 10-32). We identify the repetitive blocks by looking at the circular loops and the conditions for repeating the process(es). Notice that the *Sort* is not included in either loop.

Next, evaluate each process grouping. *Identify Item* is alone within its loop. *Sort* is also alone. The last three processes are together and are analyzed. The processes are sequential but according to the PDFD, they are not all processed in sequence. If the vendor has not changed from the previous item, we *Get Price* and *Create Order*. When the *Vendor* changes, we *File* and *Mail* the order. These statements from the PDFD translate into the *IF* conditional statement in the action diagram as shown in Figure 10-33.

The diagram is correct in interpreting the PDFD, but it is incomplete as a program specification. First we need to deal with the *First Vendor*. The *First Vendor* will *not* equal *Last Vendor*, and to file an order for a nonexistent vendor is wrong. Second, think about what an order looks like (Figure 10-34). There are one-time *Vendor* information and variable lines of *Item* information. Where the PDFD says *Create Order*, it really means *Add Item to Order*. When the *Vendor* changes and an order is complete, we want to format *Vendor information* for the new order. Figure 10-35 reflects these details and is ready for the next step. The purpose of this example is to show

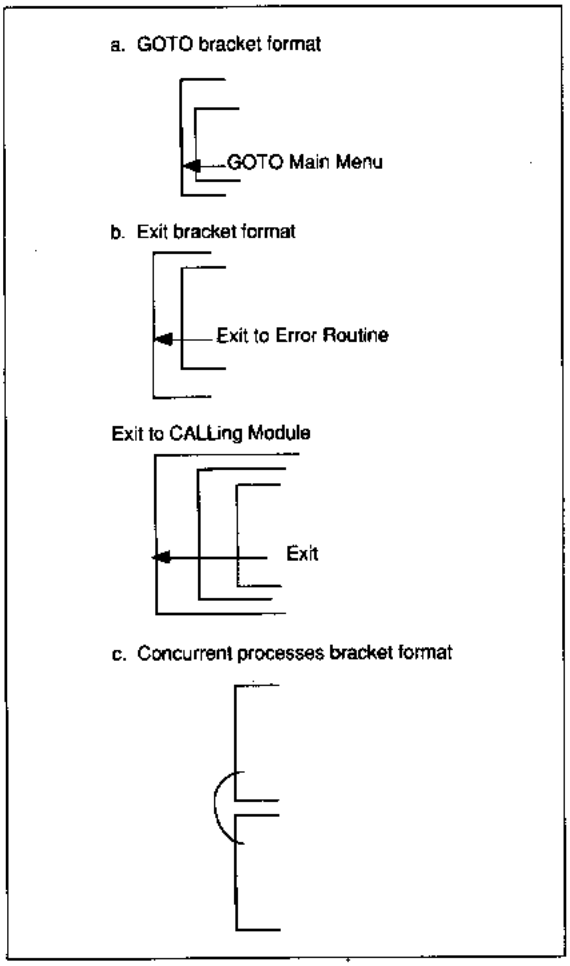


FIGURE 10-28 Miscellaneous Bracket Design Formats

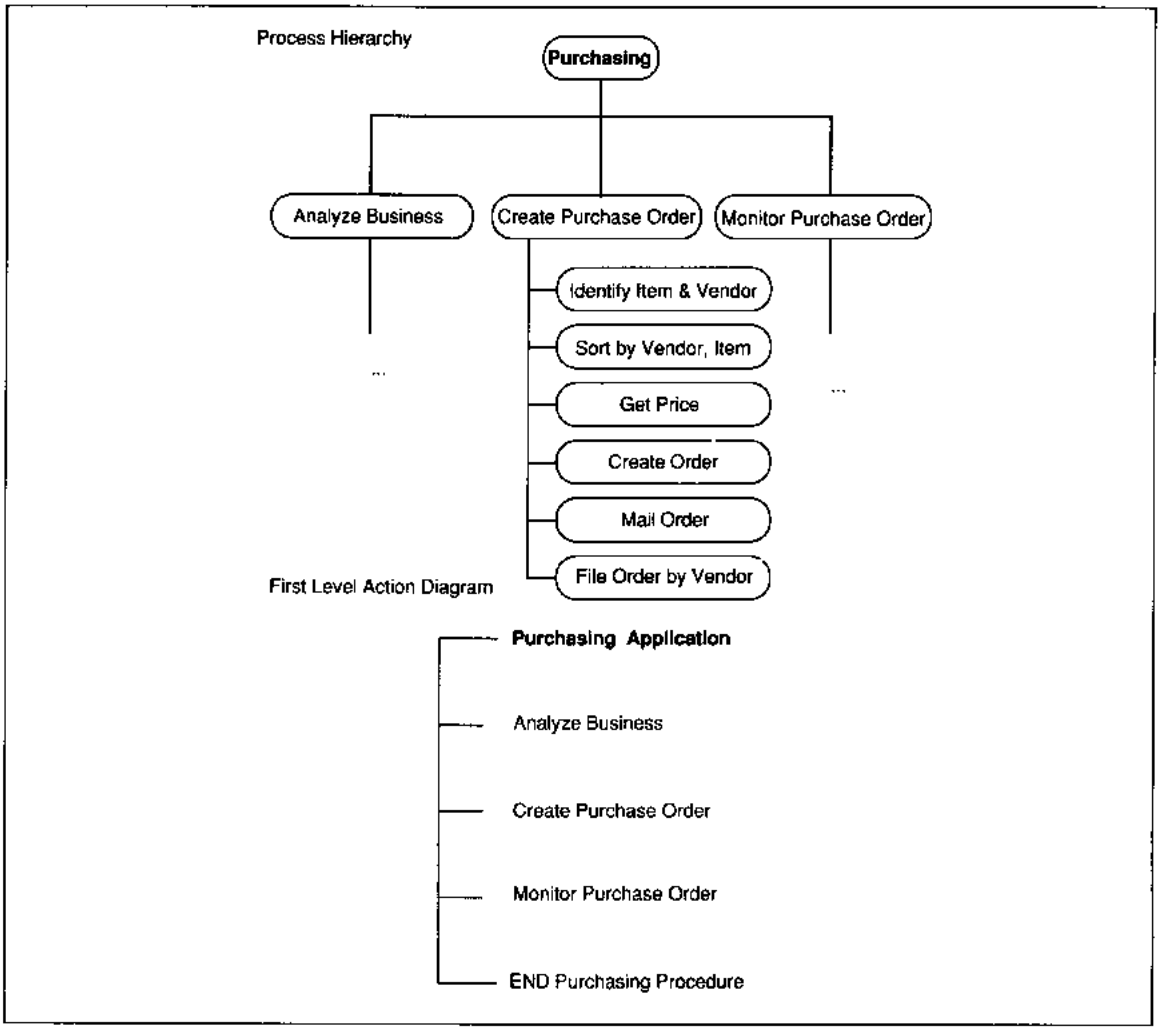


FIGURE 10-29 Process Hierarchy and First-Level Action Diagram

how a correct PDFD may need elaboration to translate into program specifications.

Using the action diagram, modules are defined. There are few guidelines on this aspect of Information Engineering. In general, you should try to define modules that perform one well-defined process and nothing else. The guidelines presented in Chapter 8 for module definition can be applied here. For the example in Figure 10-35, the *IF . . . ELSE IF . . . ELSE* processing is the module's control flow. Within the control flow we have stand-alone processes that conveniently define modules. Figure 10-36 shows the module names, each enclosed in its own rounded rectangular box to indicate that there are more details for each module. The submodules are each further diagrammed or, if fully documented in a data dictionary, refer to the dictionary entry in the module box.

For *Create Purchase Order* processing, then, we have a main module and submodules for *Create Ven-*

dor Info, *Get Price*, *Create Order Item*, *File Order*, and *Mail Order*. Notice that *Create Vendor Info* is used twice.

Next, the action diagram modules are compared to templates already in use to determine whether reuse of existing modules is possible. As reusable modules are identified, the process details are removed from the action diagram and replaced with a call statement. The called module name should indicate whether the reused module is customized for this application or not. The conventional way to identify customized reused modules is by a prefix or suffix on the name. For example, a date compare routine might be used to determine lateness. If not modified, the name of the routine might be *Date-Compare*. If customized, the name of the routine might be *RentDateCompare* or *LateReturnDate-Compare*. In the example in Figure 10-36, *Sort* uses a utility program, a special class of reusable module. The *Sort* statement is removed from the diagram and replaced with a call statement (Figure 10-37). No other modules in this example are general enough for reuse.

When reusability analysis is complete, the action diagram should show the mainline logic of the application with modules for the processes and

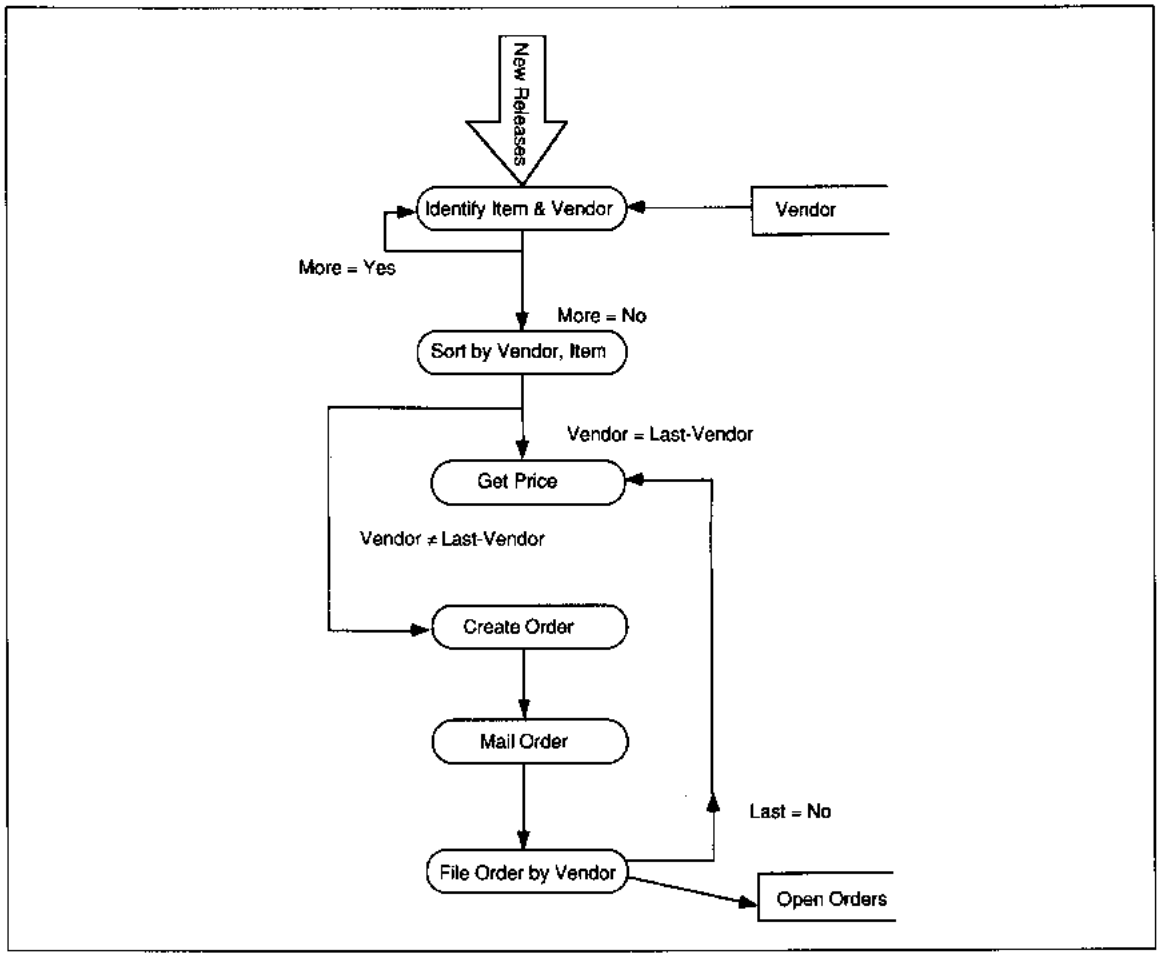


FIGURE 10-30 Sample Process Data Flow Diagram

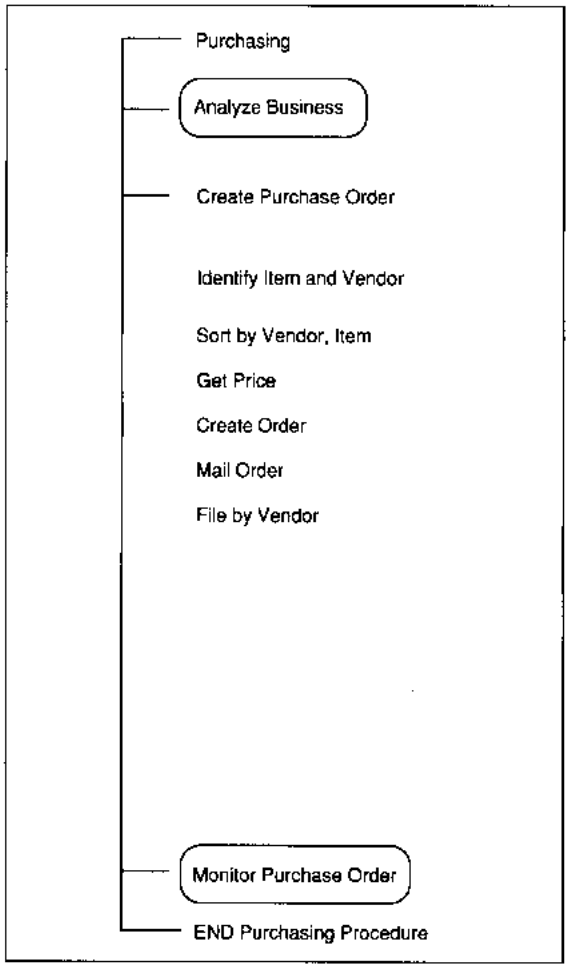


FIGURE 10-31 Second-Level Action Diagram

subprocesses. At this point, timing of processes is decided and added to the diagram. Recall that processes can be sequential or concurrent, and that concurrent processes can be either independent or dependent. Frequently, user requirements will identify required concurrency. If no user requirements identify concurrent operations, a design decision to offer or not offer concurrency is made by the SEs. Concurrency is expensive and adds a level of maintenance complexity to the application that the user might not want.

Optional concurrency is determined by evaluating module interrelationships again. Only groups of sequential modules are evaluated at first. Then the groups themselves are evaluated for possible concurrency. In Figure 10-36, two groups of two or more modules are present. The first is *Get Price* with *Create Order Item*. The second group is *File Order*, *Mail Order*, and *Create Vendor Information on Order*. Working backward, we ask if the modules are dependent on each other. Could we create an order item without knowing the price? In this case, the answer is no, we must know the price. Therefore, the modules are dependent and cannot be concurrent. In the second group, we might perform *File* and *Mail*

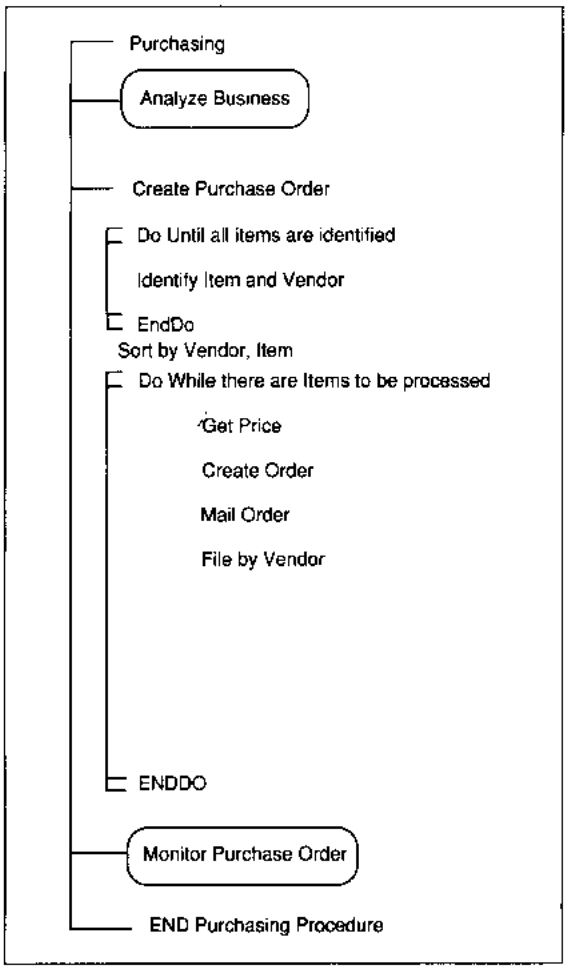


FIGURE 10-32 Repetitive Blocks on Second-Level Action Diagram

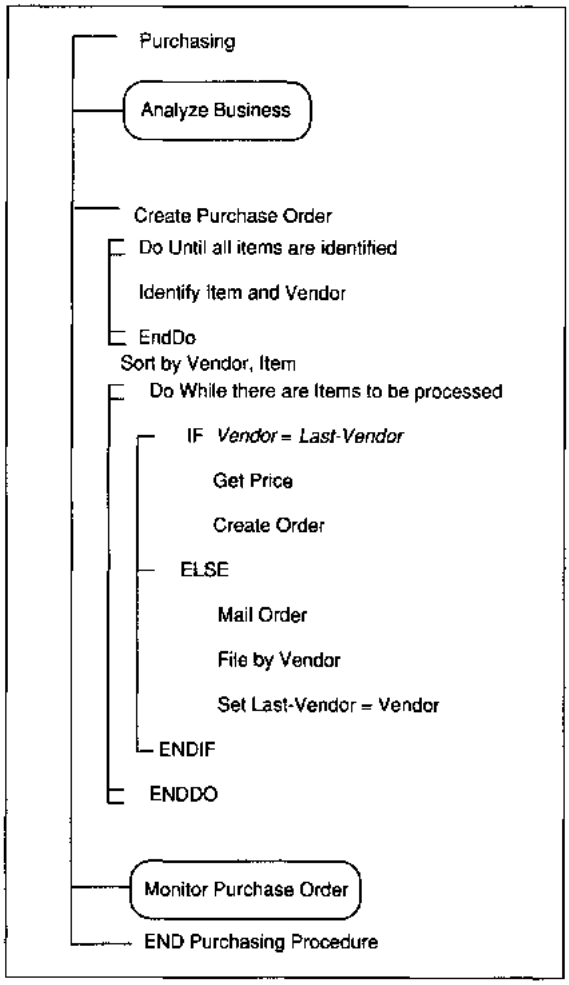


FIGURE 10-33 Conditional Statements on Second-Level Action Diagram

Order at the same time, IF success of the file operation is not an issue. *Create Vendor* cannot be done until the last order is fully processed. To decide on concurrency, we need to know the details of error handling. In this case, we find that errors are checked and handled in the module in which they can occur. If a fatal error occurs, the application does no other processing on this order. This process definition *implies* sequence to the processes. If the processes were concurrent and a fatal error occurred, some undesired processing would occur. Therefore, in this example, concurrency is not an option.

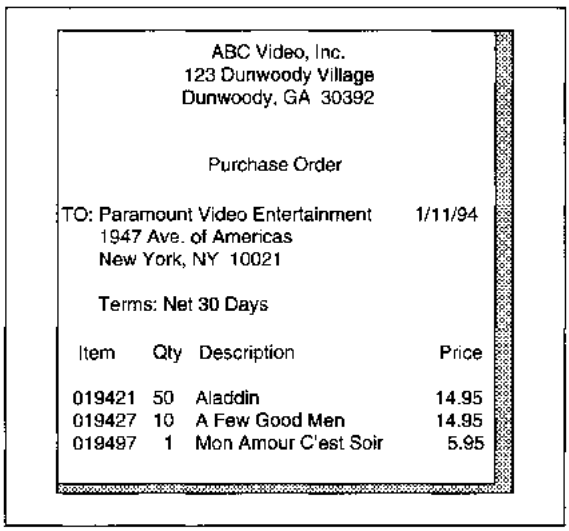


FIGURE 10-34 Order Example

Next, the entities and data elements used by the processes are added to the diagram(s). By the time this action is complete, every attribute of every relation must, at least, have been identified for creation and deletion (Figure 10-37). Any attributes not included in the processing should be reconsidered for elimination from the application. These process definitions should include attributes added to the relations as a result of design activities.

If the action diagrams are developed manually, screen identifiers can be added to the diagram with entities and attributes linked to screens (see Figure 10-38). The diagram then links data sources and destinations to both processes and screens. This type of diagram does manually what linkages in a CASE tool automate.

ABC Video Example Action Diagram

The steps to developing the action diagram are to develop the levels of action using structured constructs, perform reusability analysis, design modules, decide module timing, add data to the diagram, and optionally, add screens to the diagram (refer to p. 434). Only the first-level action diagram includes all of the processes. The lower-level diagrams consider Rent/Return processing and Video Maintenance only. The other processes are left as an exercise.

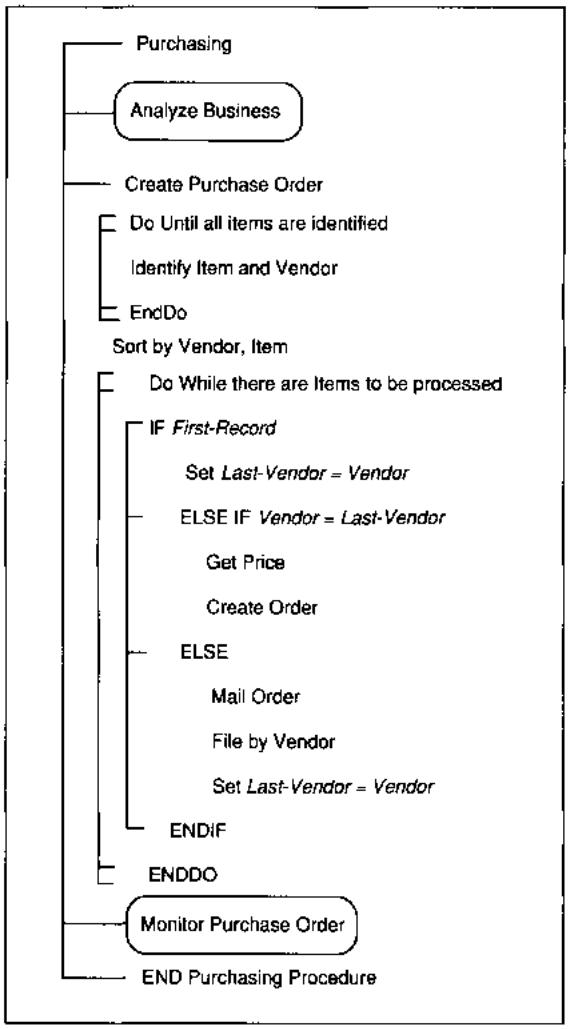


FIGURE 10-35 Order Format Details on Action Diagram

The first-level action diagram is based on the process hierarchy (Figure 10-39). First we draw the general bracket and add the module names, indicating the structural relationships between the modules by the bracket type (Figure 10-40). In the ABC diagram, the processes are all mutually exclusive.

Then, using the PDFD as reference (Figure 10-41), we develop the next level of procedural detail. The subprocess names are added to the diagram as shown in the PDFD (and process hierarchy). For each subprocess, the structural brackets indicating modular control are added.

The subprocesses for *Video Maintenance* are for create, retrieval, update, and delete processing. These processes are all mutually exclusive, so the diagram is simple (Figure 10-42). At the lowest level, we identify modules that refer to the dictionary for process details.

Rent/Return has all of the complexity in the application. Each cluster of modules is discussed separately. First, *Get Request* is always executed whenever *Rent/Return* is invoked (Figure 10-43).

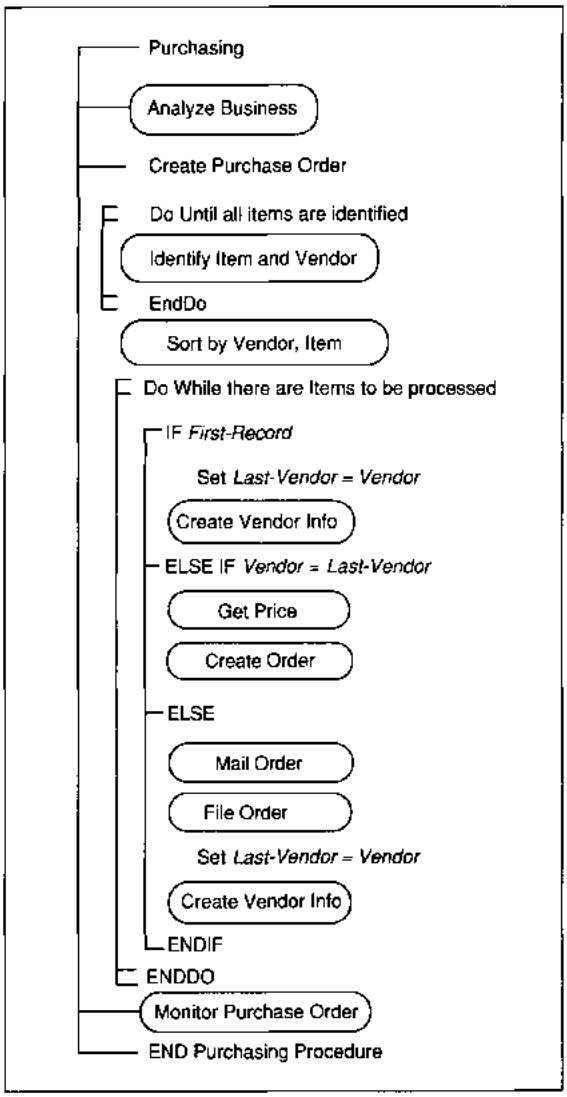


FIGURE 10-36 Module Boxes on Action Diagram

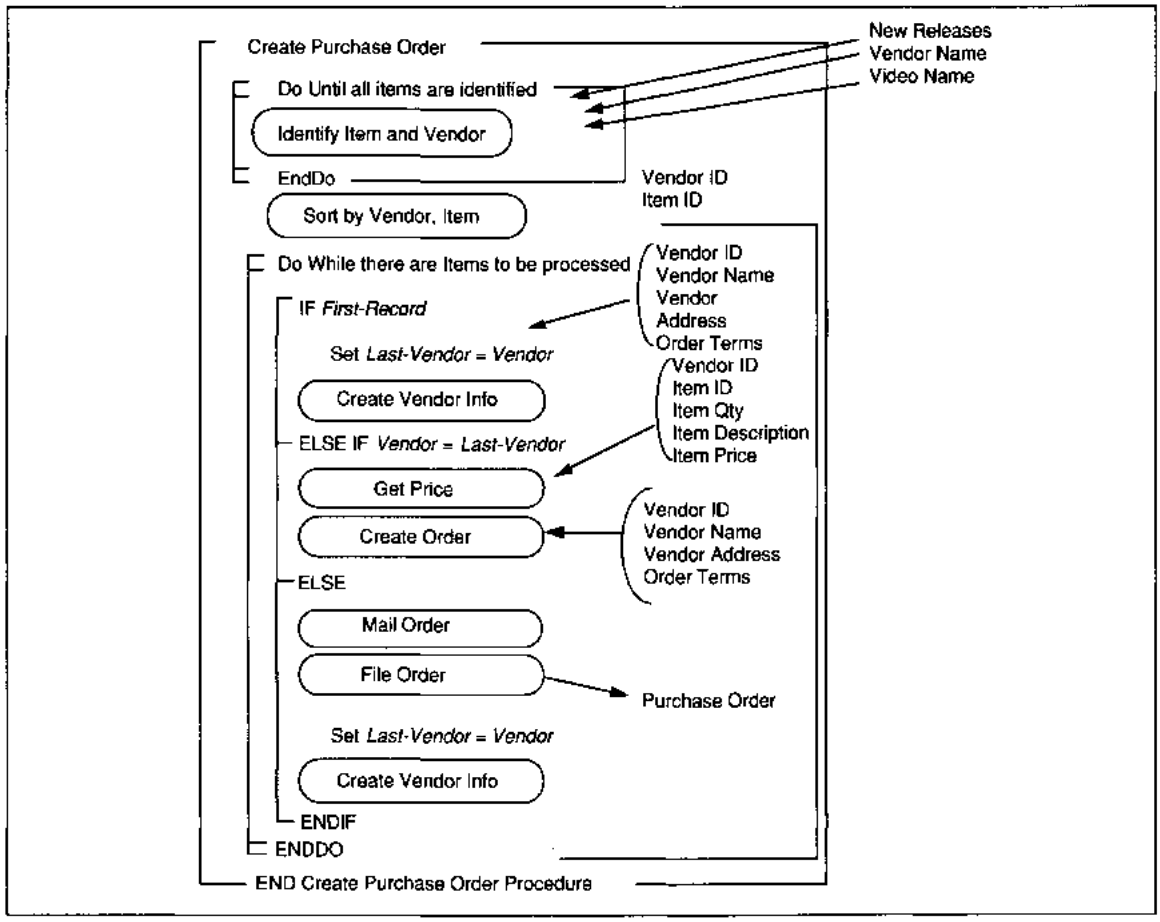


FIGURE 10-37 Data Addition to High-Level Action Diagram

Then the conditional statement for determining the type of request is added (Figure 10-43). The two options are *If Customer* and *If Video ID*, and each has its own processes.

Next, *Open Rentals* are read and displayed until all *Open Rentals* for this customer are in memory (Figure 10-44). The *Open Rental* loop is a simple *Do While* process.

Then video returns are processed using a repetition with a conditional structure (Figure 10-45). Late fees are checked in a repetitive loop for all *Open Rentals* (Figure 10-46). New rental *Video IDs* are entered for all new rentals (Figure 10-47). *Process Payment and Make Change* is a stand-alone module. Then, for all open and new rentals, the *Open Rentals* file is updated; for all of today's returns, history is updated; and if payment is made or a user requests, a receipt is printed (Figure 10-48). The consolidated action diagram is shown in Figure 10-49.

Next, evaluate the diagram to identify program modules. As in the example above, we have naturally identified modules as part of process definition. For instance, *Get Valid Customer* is a small, self-contained module that does one thing only. The module uses a Customer ID to access the Customer relation. If the entry is present, the credit is checked. The name, address, and credit status are returned. The remaining modules, that we originally defined as business processes doing one thing, should each be reviewed to ensure that they are, in fact, single purpose. This is left as a class activity.

In addition, we can now resolve the issue held over from analysis about whether to keep separate or

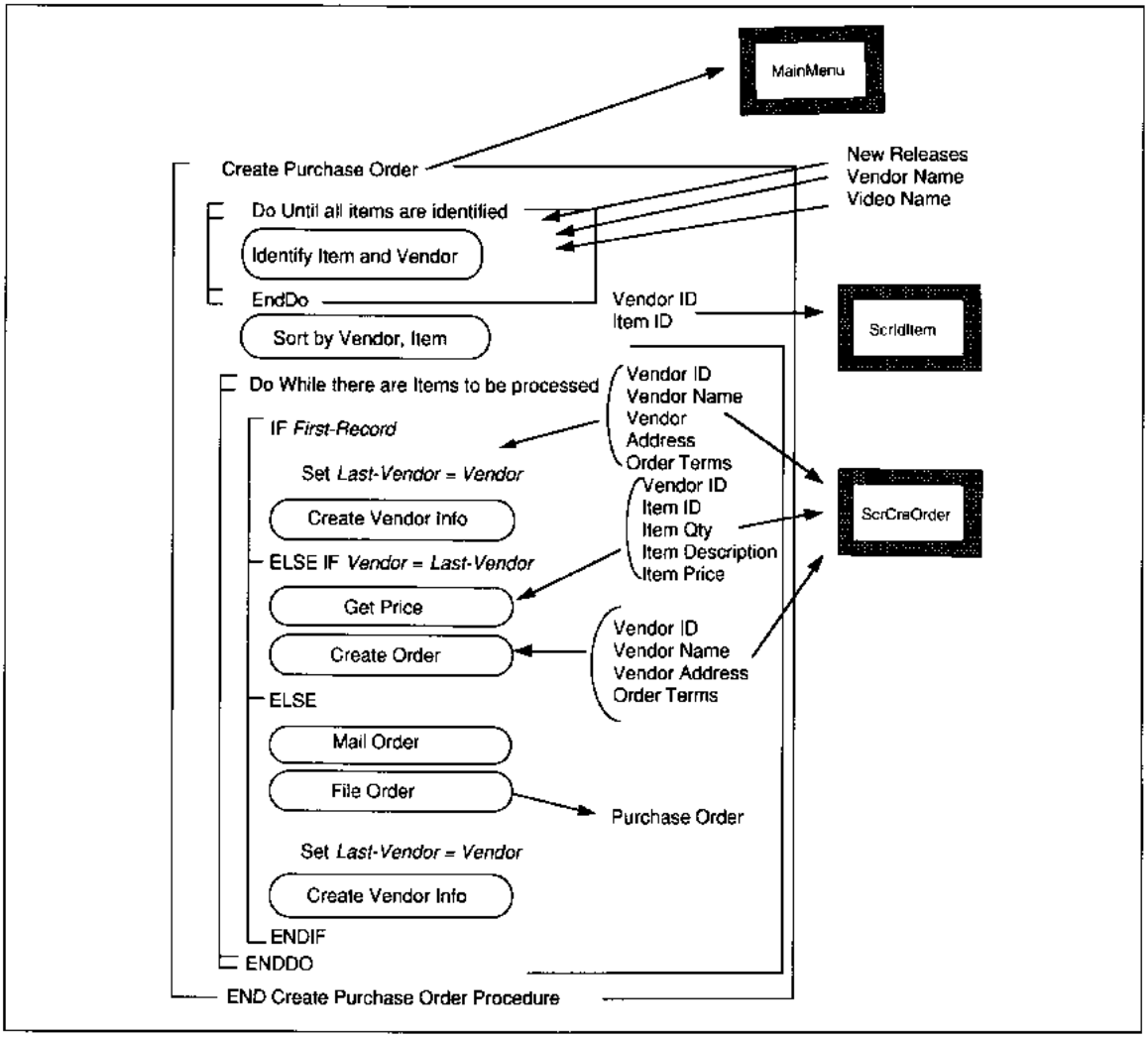


FIGURE 10-38 Optional Screen Processing on Action Diagram

consolidate *Get Open Rentals*, *Add Return Date* and *Check for Late Fees*. Individually, each of these processes is singular (i.e., does one thing). If they are consolidated, they would remain singular but be placed within the same repetition loop. The issue here, then, is which method is easier to program and implement in the intended language, and which provides the better user interface. We need to visualize the user interface and memory processing for each alternative.

If the modules are kept *separate*, all *Open Rentals* are read first and displayed. Then the clerk can be prompted for new videos or for returns. If we prompt for returns every time, many wasted entries to deny return processing will be made. If we prompt for either new or return *Video IDs*, we need a method of knowing which is entered. Assuming we figure that out, we then get all returns and enter today's date for returned videos. Then all entries on the screen are scanned to determine new late fees.

If the modules are *consolidated*, as each *Open Rental* is read, *Late Fees* are computed for tapes with return dates and no late fees (see Figure 10-50). There are two options for this process. Either we

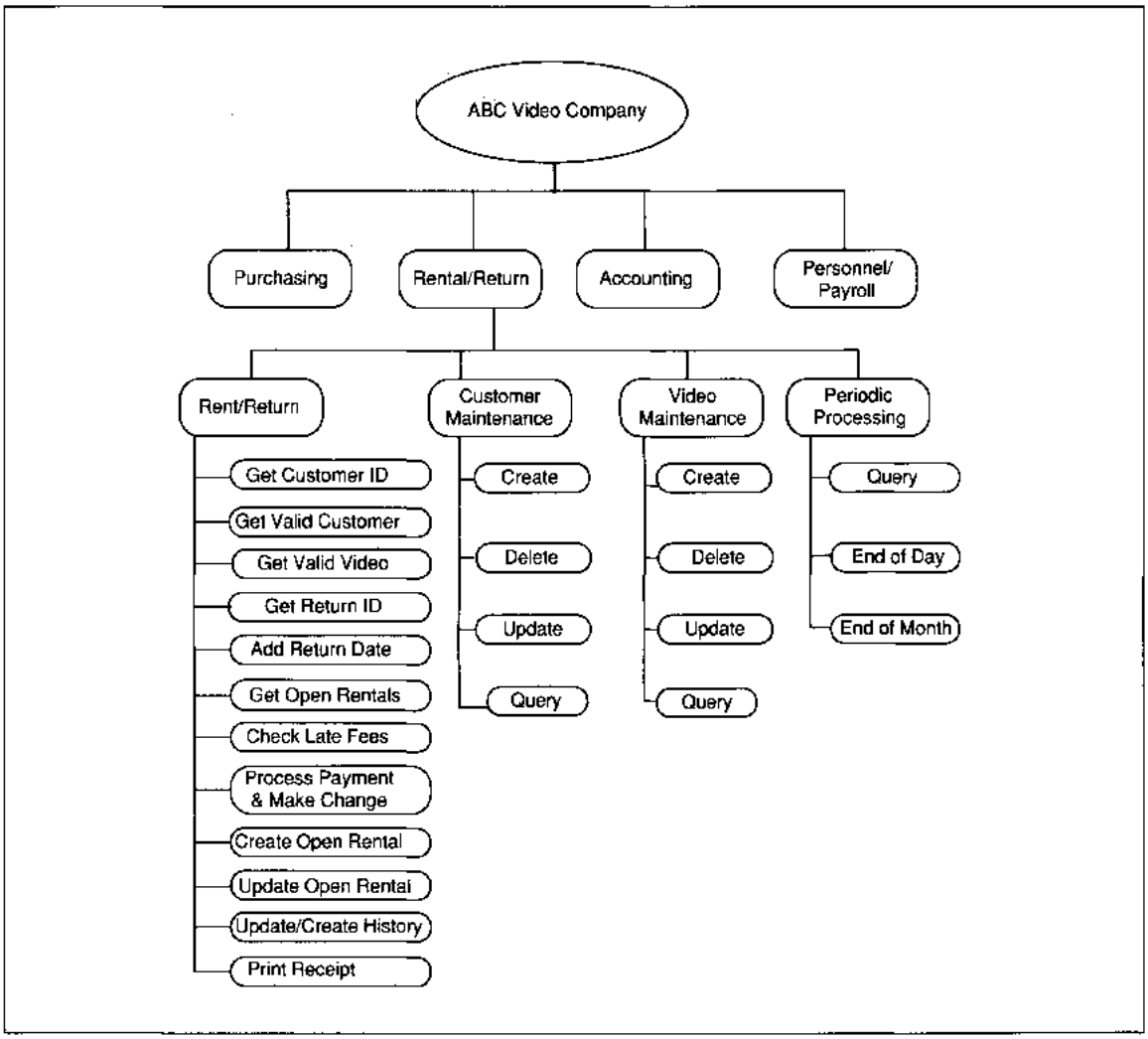


FIGURE 10-39 ABC Video Process Hierarchy Diagram

assume there are no more returns or the clerk must respond to each *Open Rental*. With the first option, the clerk would have a selectable option for more return processing. When chosen, each return *Video ID* is entered and *Late Fees* are computed for that video.

Notice that *both* alternatives have problems. The separation alternative has a problem in dealing with returns, and there will be a slight delay for *Late Fee* processing. The consolidation option actually modifies the processes from the PDFD somewhat for *Late Fee* processing.

Data storage for a rental in memory is the same for both alternatives. We need a location for customer information, a table for open rentals, a table for new rentals, and locations for payment information. We will have three iterations through the table for *Open Rentals* in the separate alternative, and one, or two if returns are present, iteration(s) in the consolidated alternative.

The alternatives are approximately the same in implementation complexity, although three iterations are more likely to contain bugs than one. The human interface design is the same for both alterna-

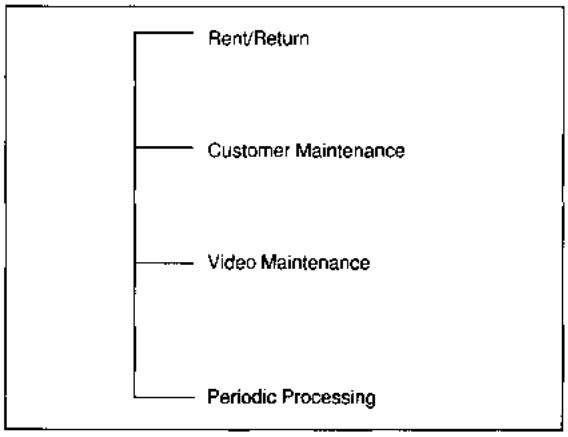


FIGURE 10-40 ABC First-Level Action Diagram

tives. The difference in the human interfaces is the speed and timing for data to appear on the *Open Rentals* lines. In this case the consolidated alternative is slightly faster. The difference in memory processing is the number of iterations through *Open Rental* data. Again, the consolidated alternative is preferred somewhat because it is less likely to contain bugs. With no overwhelming evidence for or against either alternative, this amounts to a judgment call. We will choose the consolidated alternative to minimize the probability of errors and the number of iterations through the data. The action diagram, reflecting consolidated open rental processing, is in Figure 10-50.

The next activity is reusability analysis. ABC has no library of reusable modules to consider since it currently has no computer processing. The types of modules the consultants are likely to have might be relevant to error processing or to screen interactions. For our purposes, we assume no reusable modules.

To assess module timing, we analyze the module clusters. The only modules that could be concurrent are those in the last cluster to update files and print a receipt. Before deciding concurrency, we must decide the details of history processing that were deferred from analysis. We have two types of history files: *Customer* and *Video*. *Customer History* is a separate file that contains the *Customer ID* and all *Video IDs* rented by that customer. No counts, dates, or copy information are anticipated. This description complies with the case requirements in Chapter 2.

Video History contains *Video ID*, *Copy ID*, *Year*, *Month*, *Number of Rentals*, and *Days of Rental* for each entry. This data description also complies with the case requirements in Chapter 2. The issue to be decided is whether or not *Video History* is maintained during on-line processing, or if the current month's activity is kept with *Copy* information. If the second alternative is chosen, we need a monthly process to update the Video History and reinitialize the counts in the *Copy* relation. If the first alternative is chosen, we have two more alternatives. First, we might need update and create processing because, for any one copy, we would not know in advance whether it has a historical entry or not. This alternative requires bug-prone processing that is more complex than keeping counts in the current *Copy* relation. Second, we could create an empty entry for every tape at the beginning of every month. This alternative is not attractive because it generates many empty records on history. Both of these alternatives would require history to be on-line. Keeping current counts with *Copy* relations does not require history to be on-line. The final argument for keeping the counts in *Copy* information is that, to maintain status of a given tape, *Copy* information must be updated upon video return anyway. As long as the tuple is being read, updating it with count information requires adding lines of code rather than a new module. From this discussion, it should be clear that keeping current counts in the *Copy* relation is the preferred alternative. We document this and the other changes in the Data Dictionary.

Now we can discuss module timing for the last group of modules. In this group we create and/or update *Open Rentals*, update *Copy*, and *Print Receipt*. Recall from analysis that Vic does not want file update success to be known to the customers. The receipt should be printed regardless of updating success. This implies that printing could be concurrent with the file processes. The file updates cannot be concurrent because they will all be on the same device. Since there is already contention for the file

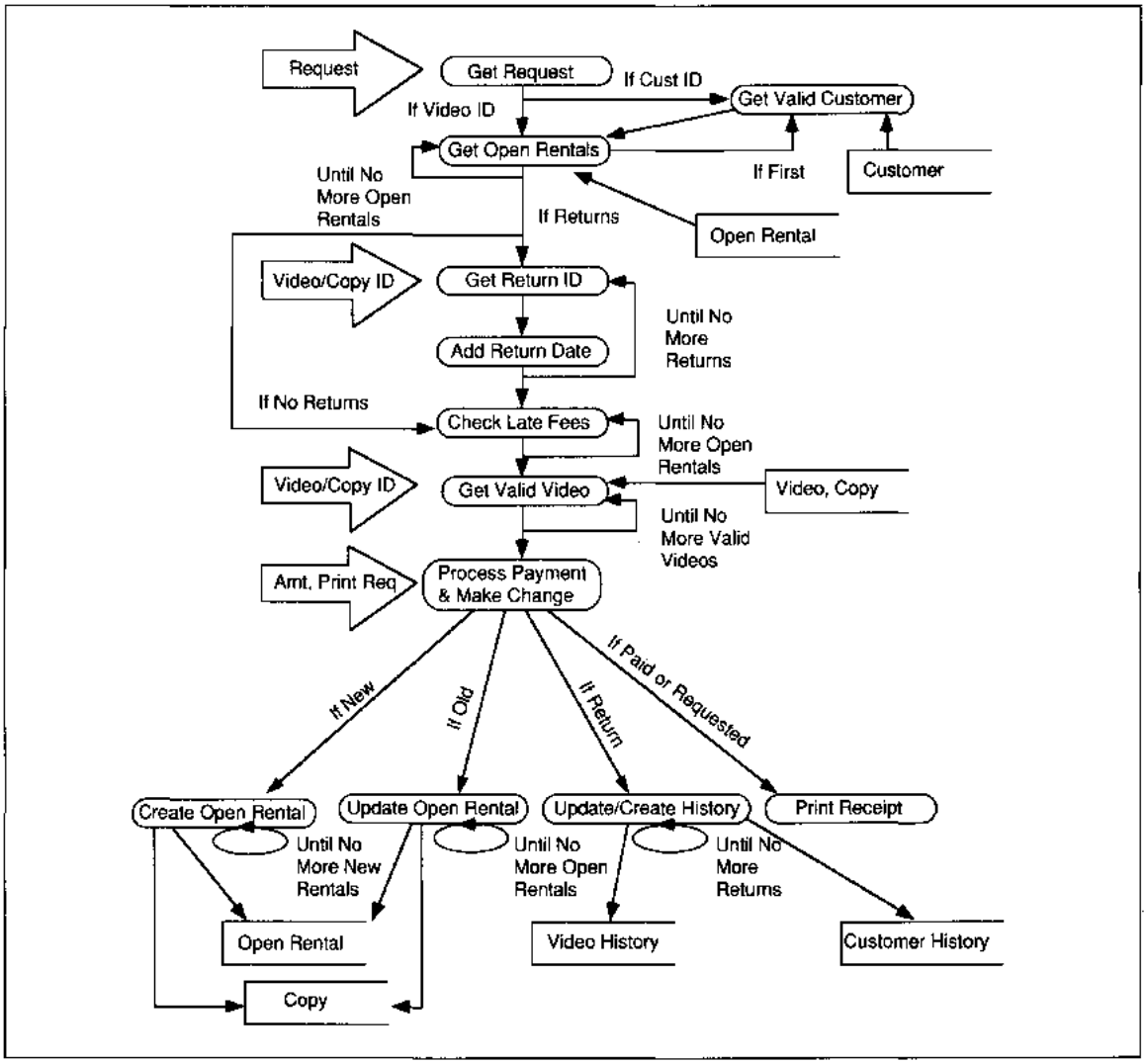


FIGURE 10-41 ABC Video Process Dependency Diagram

among the users, it is unlikely that we would want to increase contention by having the updates concurrent. If printing is the only concurrent process, it is not worth the cost to provide concurrency. Therefore, the processes will be made sequential for production operation. Figure 10-50 is not changed at this point.

The entities and data attributes are added to the diagram next to show input and output processing. Two entities, *EOD* and *Rental Archive*, are still undefined, having been deferred in analysis. These are left as an exercise. The entities referenced in *Rental/Return* processing, *Customer*, *Open Rental*, *Video*, *Copy*, *Customer History*, and *EOD* are all shown in Figure 10-51. When an action diagram arrow is from an entity to a process, it means that the entire tuple is accessed. The final action is to add screens to the action diagram, but they are not yet defined, so this activity will be left as a future exercise.

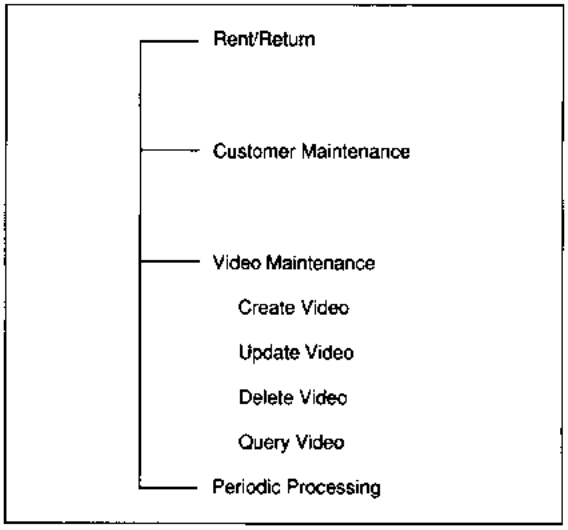


FIGURE 10-42 ABC *Video Maintenance* Second-Level Action Diagram

Define Menu Structure and Dialogue Flow

Guidelines for Defining the Menu Structure and Dialogue Flow

The interface structure includes design of a menu structure and design of dialogue flow within the menu structure. Both designs are based on the PDFD and process hierarchy diagram developed during IE analysis.

First, the menu structure is developed. Recall that the menu structure is a structured diagram translating process alternatives into a hierarchy of options for the automated application. The task hierarchy is analyzed to define the individual processing screens required to perform whole activities, and to identify the other processes and activities in the hierarchy which must be selected to get to the processing screens.

Let's walk through the development of the sample menu structure shown in Figure 10-7. The related process hierarchy diagram is shown as Figure 10-52 with the individual processing screens, selection alternatives, and hierarchy levels identified. For each level in the hierarchy, we identify a level of menu processing. Using simple bracket structures to translate from the top to the bottom of the hierarchy, we first define the options for the first level menu (see Figure 10-53). Next, the menu options for the first process level of the hierarchy are shown in Figure 10-54. Finally, the remaining detailed processes are added to the diagram (see Figure 10-55).

If, for any reason, the hierarchy or lower-level processes are in doubt, review the proposed menu structure with the users before proceeding. If the

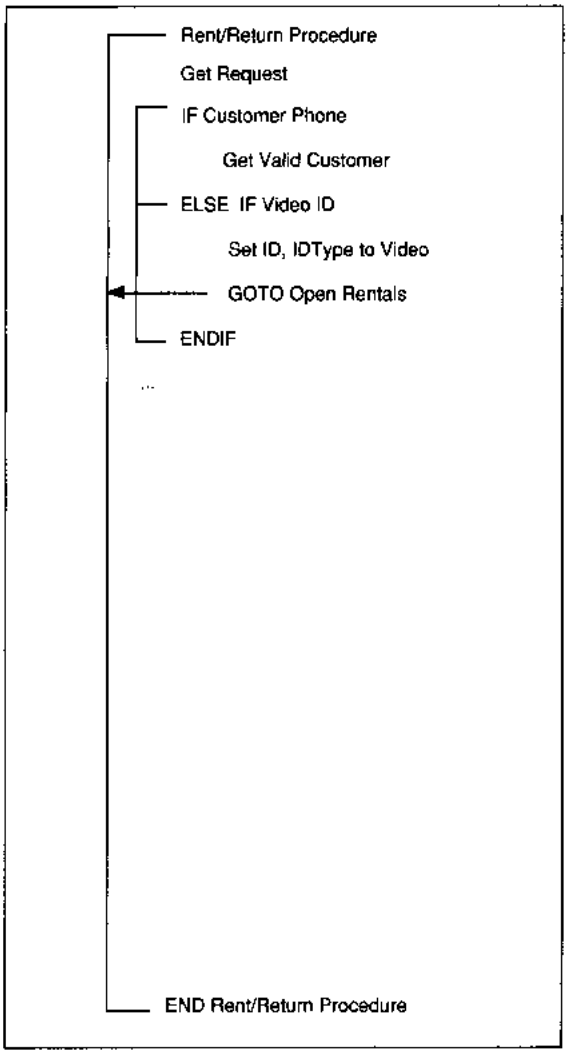


FIGURE 10-43 Request Processing Action Diagram Constructs

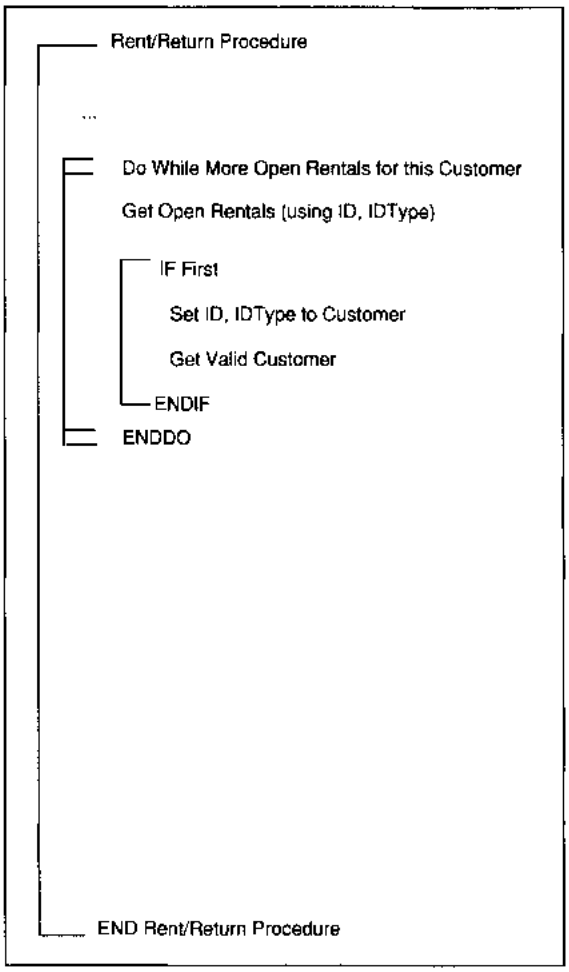


FIGURE 10-44 *Open Rental* Action Diagram Constructs

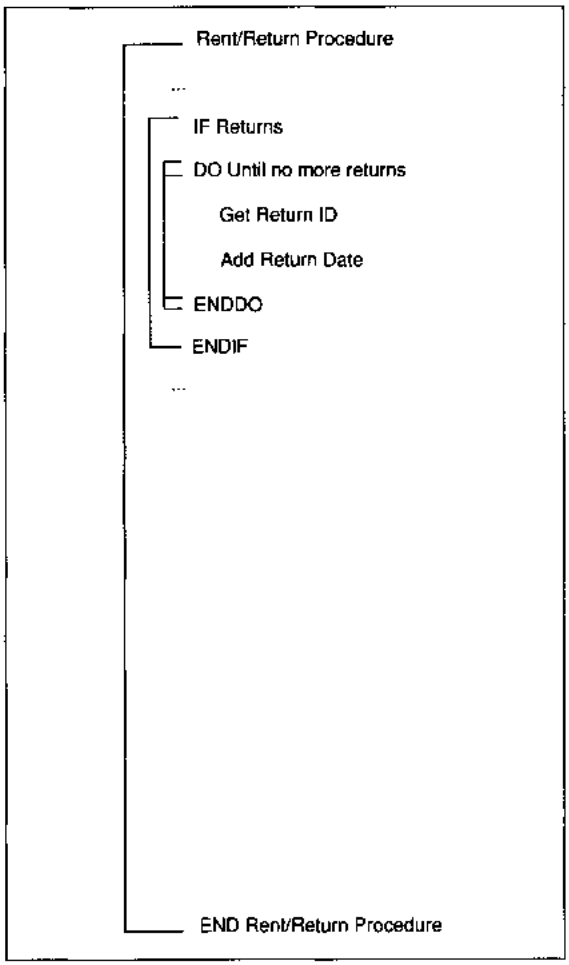


FIGURE 10-45 *Video Returns* Action Diagram Constructs

process hierarchy diagram is accepted as correctly mirroring the desired functions in the application, proceed to the next step, defining the movements between menu items.

Traditionally, applications were constrained to moving top-to-bottom-to-top with no deviation. Anyone who uses such an interface for long knows it is irritating to wait for some menu that is unwanted and to enter choices purely for system design reasons. The decisions should relate to application requirements as much as possible. For instance, security access control requirements can be partially met by restricting movement to functions as part of dialogue flow. The decisions about legal movement should be made by the users based on recommendations by the designers; although frequently, dialogue flow decisions are made by the SEs. In general, if the users are functional experts, an open design that allows free movement should be used. If users are novices or not computer literate, a more restrictive design should be used to minimize the amount of their potential confusion.

Figure 10-56 shows types of arrows used to depict movement between levels of a menu structure.

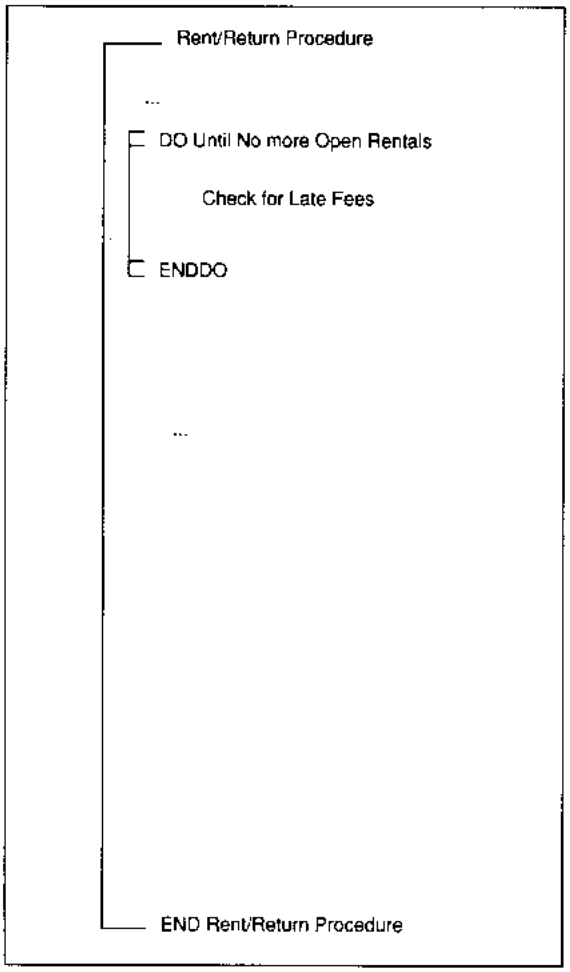


FIGURE 10-46 *Late Fee* Action Diagram Constructs

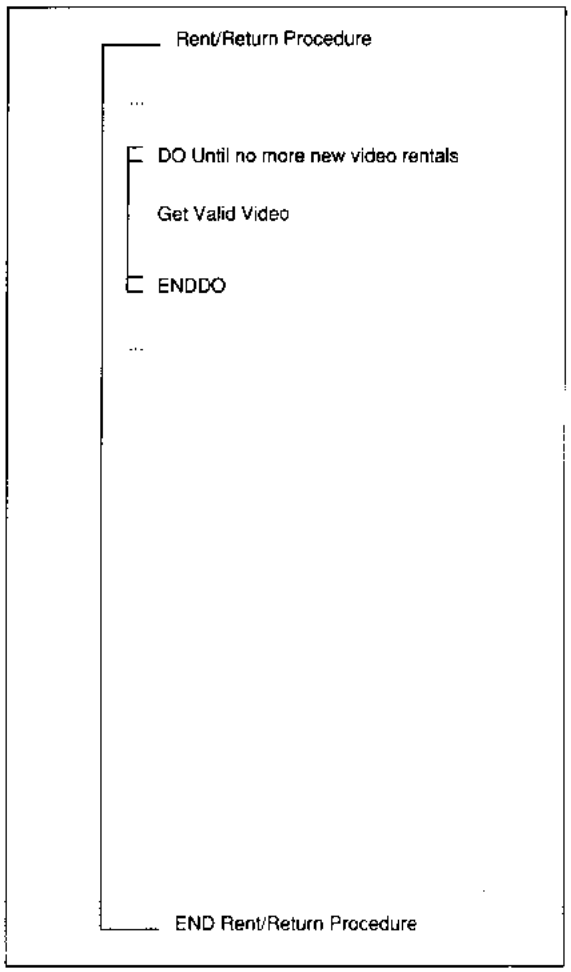


FIGURE 10-47 *New Rentals* Action Diagram Constructs

In a small diagram, with less than ten screens, only single-headed arrows are used, and at least two arrows are drawn for each entry: one entering and one leaving (Figure 10-56a). In a large diagram, with over ten screens, the triple-headed arrows can be added to the diagrams to depict call-return processing (Figures 10-56b and 10-56c).

An example of restricted screen movement that might be designed for novice users is shown in Figure 10-57a. In the diagram, all movement is to or from a menu. The diagram in Figure 10-57b shows that any level of upper menu might be reached from the lower levels. This speeds processing through menus and is preferred to the design shown in Figure 10-57a which only allows a process to return to the menu level from which it was activated. Restrictive dialogue flow (Figure 10-57a) is the type of design that is most likely to waste user time and become annoying.

Experts and frequent users usually are provided more alternatives for interscreen movement because they become proficient with the application. Unrestricted screen movement is desirable for these users. An example of unrestricted movement in screen

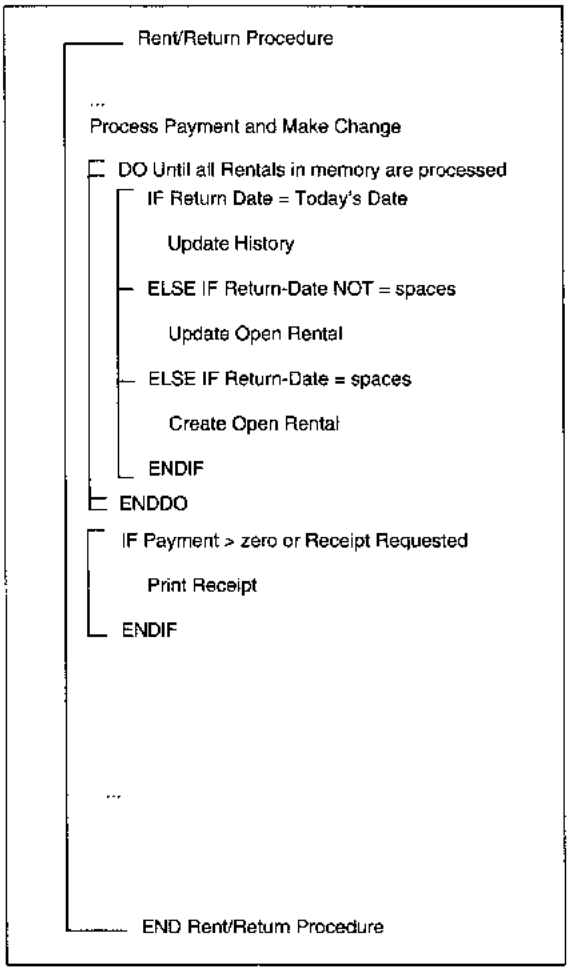


FIGURE 10-48 Payments, File Update and Printing Action Diagram Constructs

design is shown in Figure 10-57c. In the example, the user begins at the main menu and may move down the hierarchy in the same manner as a novice, or may move directly to a process screen, at the user's option. Unrestricted movement requires the design and implementation of a command language or sophisticated menu selection structure that is consistent with the basic novice menu selections, but adds the expert mode.

Unrestricted movement can be costly and error-prone, which are the main reasons why it is not prevalent. The added cost is due to increased access control structure that must accompany an open movement design. The added errors are from a need to provide a specific location on the screen for entry of the expert's direct screen requests. Each request must be checked for access control and legality, plus the current context (i.e., screen and memory information) might need to be saved for return processing.

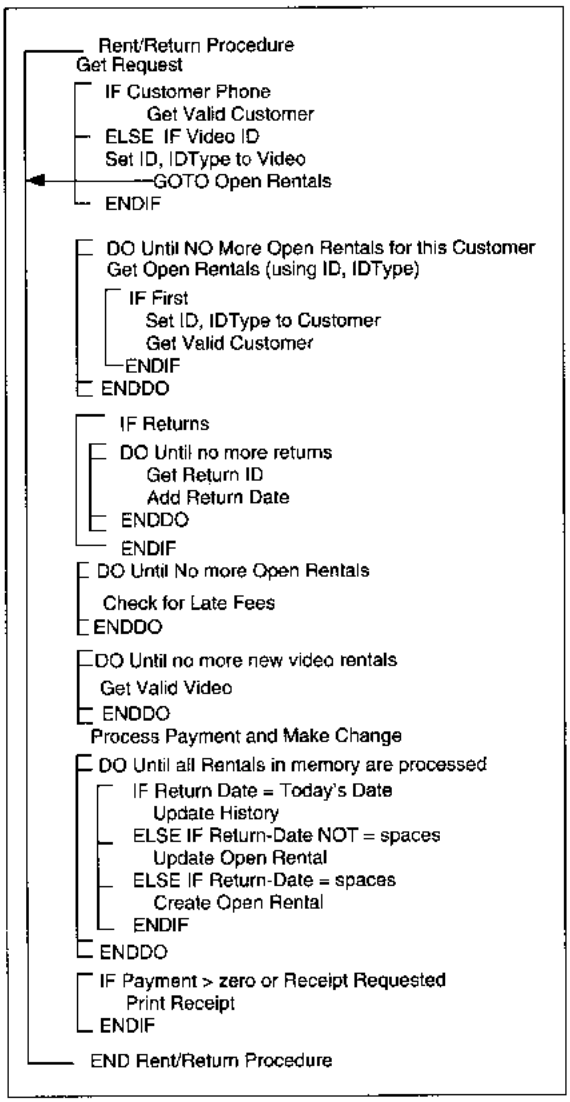


FIGURE 10-49 ABC Consolidated Action Diagram

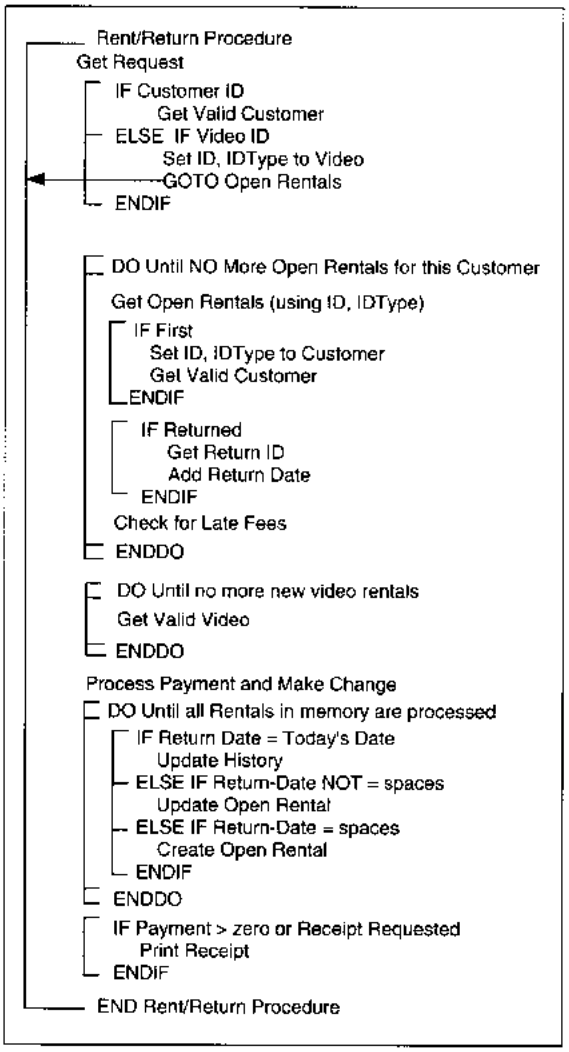


FIGURE 10-50 ABC Action Diagram with Consolidated Open Rental Processing

Upon completion, the menu structure and dialogue flow diagrams are given to the human interface designers to use in developing the screen interface (see Chapter 14). The dialogue flow diagram is also used by designers in developing program specifications. Before we move on, note that even though the menu structure is identified, the human interface may or may not be structured exactly as defined in the menu structure diagram. The human interface designers use the menu structure information to understand the dependencies and relationships between business functions, entities, and processes; they may alter the structure to fit the actual human interface technique used. If a traditional menu interface is designed, it could follow the menu structure diagram.

ABC Video Example Menu Structure and Dialogue Flow

The menu structure is derived from the process hierarchy diagram in Figure 10-58 (reprint of Figure 9-26). First, the activities from the decomposition form the main menu options (see Figure 10-59). The processes are used to develop submenu options. Then, the lowest level of processing completes the simple structure (Figure 10-60).

Notice that all Rent/Return processing is expressed in the first menu option even though we have many subprocesses in the hierarchy. Rental/return has many subprocesses performed as part of the hierarchy diagram. Unlike the other subprocesses, rental/return does not have individual menus and screens for each subprocess. Rather, rental/return requires a complex, multifunction screen with data from several relations and processing that varies by portion of the screen. The subprocesses for rental/return, then, describe actions on portions of the screen. You cannot tell from the decomposition diagram that rental/return has this requirement; rather, you know from application requirements (and experience) what type of screen(s) are needed. An incorrect rendering of the menu structure, such as the one in Figure 10-61, would look weird and should make you feel uncomfortable about its correctness.

Second, notice that we do not indicate access rights for any of the processing options on the diagram. The security access definition is superimposed on the menu structure by the interface designers to double-check the design thinking of the process designers. If there is an inconsistency, the two groups reconcile the problems.

Next we develop a dialogue flow diagram from the menu structure diagram. The rows of the dialogue flow diagram correspond to the entries in the menu structure (Figure 10-62). Rows are entered by level of the hierarchy by convention.

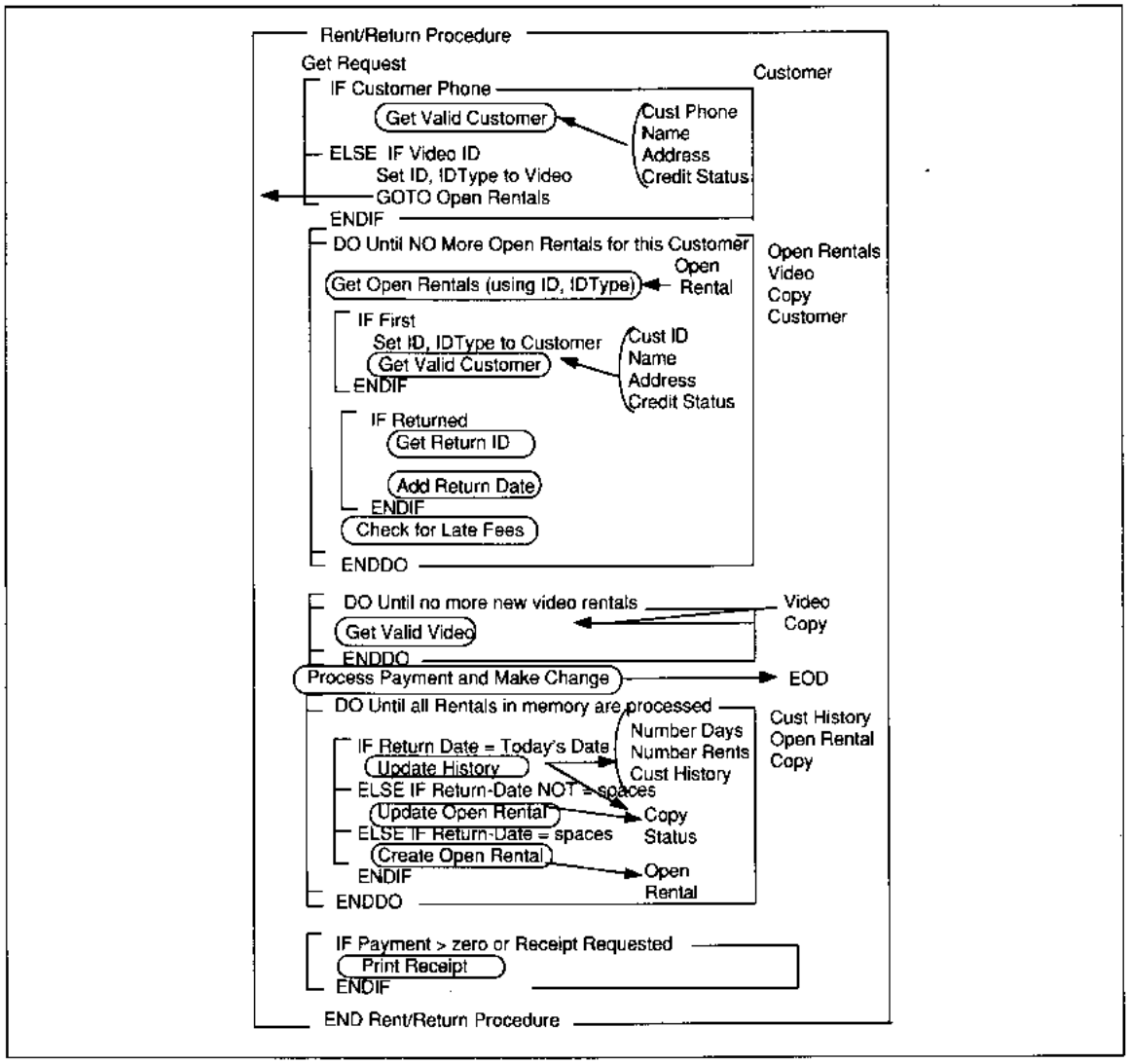


FIGURE 10-51 ABC Action Diagram with Data Entities and Attributes

We need to decide how much flexibility to give users, keeping in mind the security access requirements and the users' computer and functional skills. Users are mostly novices with little computer experience. The average job tenure is less than six months. Data and function access for clerks are unrestricted for customer, video, and open rentals add, change, and retrieve functions. Other options are more restricted in terms of which user class can perform each function.

First we define the options. We could define flexible movement between those options only, and restrict movement to other options through the hierarchy. Top-down hierarchic access is possible. We could allow hierarchic access combined with flexible 'expert' mode movement throughout the hierarchy, constrained by access restrictions.

For each option, ask the following questions. Does Vic have a preference? Which best fits the user profile? Which is the *cleanest*

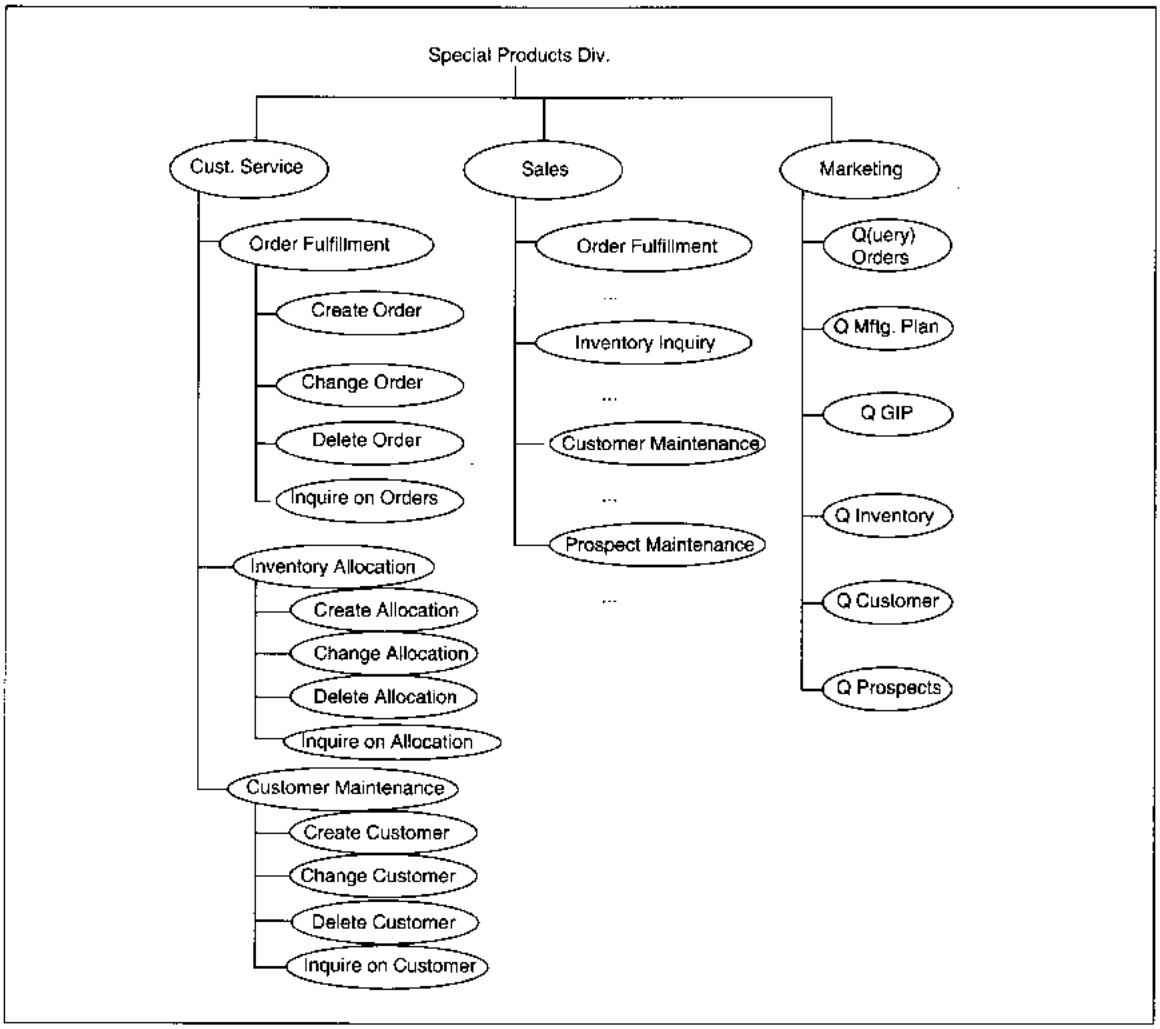


FIGURE 10-52 Example of Process Hierarchy Diagram

implementation, least likely to cause testing and user problems?

Vic, in this case, has no preference. Having never used computers, he has no background that allows him to make a decision. He says, "Do whatever is best for us. I let that up to you. But I would like to see whatever you decide before it is final." This statement implies interface prototyping, which should always be done to allow users to see the screens while they are easily changed.

Most of Vic's employees work there for 1½ years and have little or no computer experience. Therefore, screen processing that is least confusing to new users should be preferred. Usually, novices prefer hierarchic menus, providing the number of levels do not become a source of confusion. Also, the simplest implementation is always preferred; that is, the hierarchic menu option.

Based on the answers to the questions, we should design a restrictive, hierarchic flow. As Figure 10-63

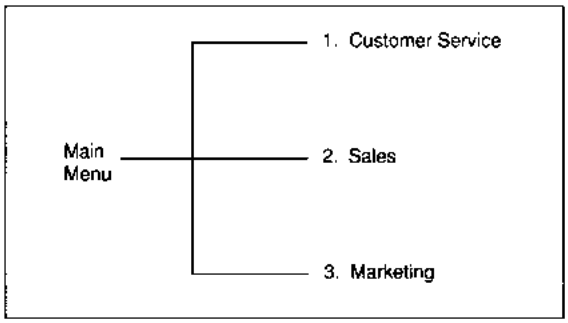


FIGURE 10-53 First-Level Menu Structure

shows, this design is simple and easy to understand. The dialogue flow and screens should be prototyped and reviewed with Vic at the earliest possible time to check that he does not want an expert mode of operation.

You might question whether the movement from rent/return to customer add and video add should be on the dialogue flow diagram. This is a reasonable concern since the process of rent/return does allow adding of both customers and videos within its process. The issue is resolved by local custom. In general, given the option, such flexibility should be shown on the diagram for clarity and completeness. Sometimes, local convention or a specific CASE tool requirement do not allow such completeness.

Plan Hardware and Software Installation and Testing

Guidelines for Hardware/Software Installation Plan

The guidelines for hardware and software installation planning are developed from practice and identify what work is required, environmental planning issues, responsibility for the work, timing of materials and labor, and scheduling of tasks.

Installation requirements should always be defined as far in advance of the needs as possible and documented in a **hardware installation plan**. Installation planning tasks are:

1. Define required work
   Define hardware/software/network configuration
   Assess physical environment needs
   Identify all items to be obtained
   Order all equipment, software, and services
   Define installation and testing tasks
2. Assign responsibility for each task
3. Create a schedule of work

If the SE team has no experience with configuring installations, their work definition should always be checked by someone who has experience. In general,

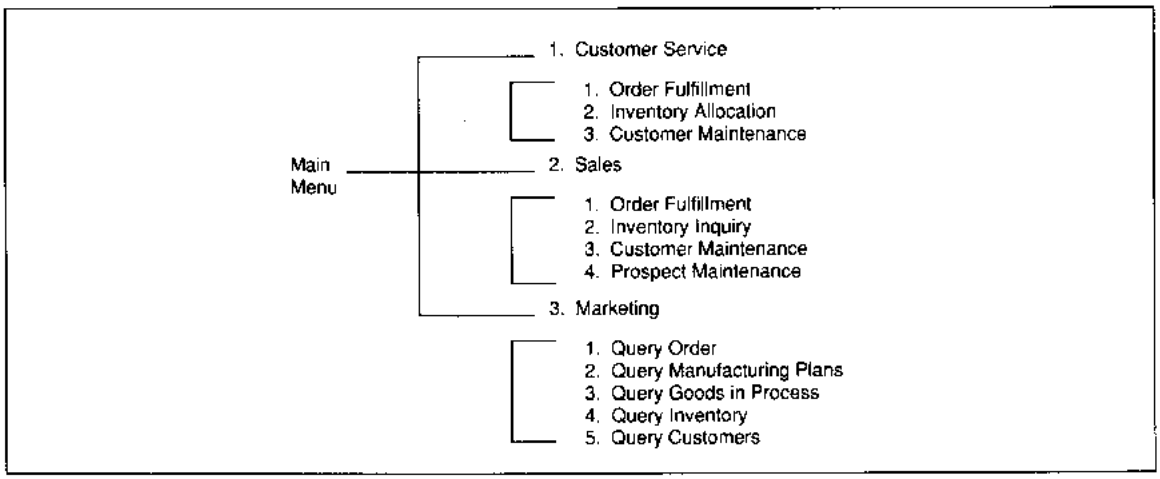


FIGURE 10-54 Second-Level Menu Structure

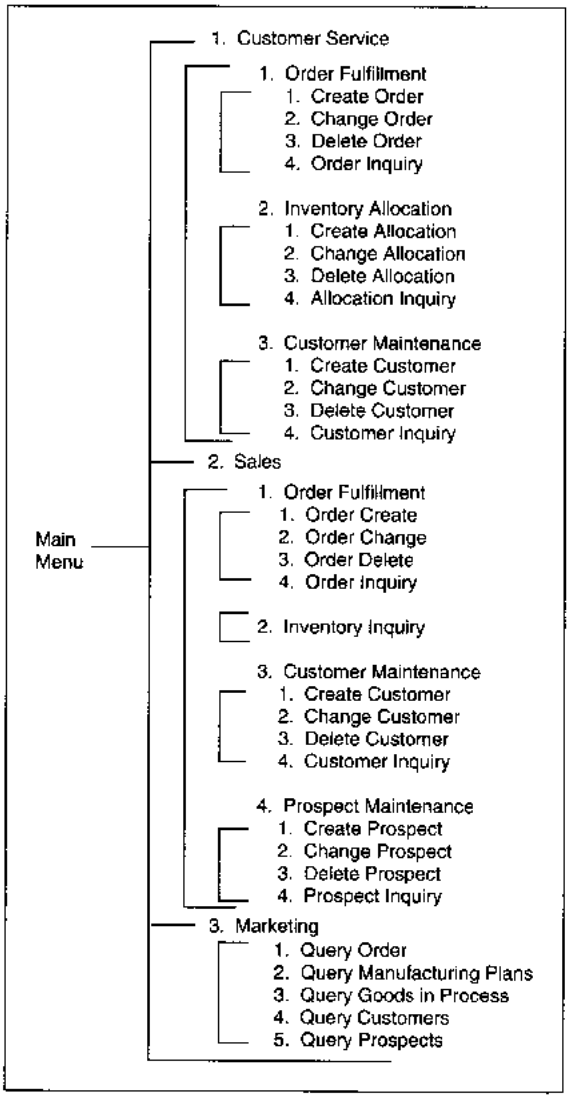


FIGURE 10-55 Final Menu Structure

you define the complete hardware, software, and network configuration needed, match the application configuration requirements to the current installation, get approval for all incremental expenditures, order all equipment and software, and install and test all equipment and software. In a mainframe environment, this task is simplified because the first step, configuration definition, can be abbreviated and done with help from an operations support group. The operations support group also would install and test hardware and install software.

When the configuration is defined, it is matched to the current installation to determine what items need to be purchased. In new installations, the physical installation environment is as important as the equipment. Building, cooling, heating, humidity control, ventilation, electrical cable, and communications cable needs should all be assessed. If you have no experience performing these analyses, hire someone who does. Do not guess. You only do the client a disservice, and chances of making a costly mistake are high.

Once needed items are identified, they should be ordered with delivery dates requested on the orders. The delivery dates should conform to the expected installation schedule which is discussed below. The goal is to have all equipment and parts when they are needed and not before. For capital expenditures, this delays the expense until it is needed. Planning for large capital expenditures should be done with the client and accountant to stagger charges that might be a financial burden.

As items to be installed are identified and ordered, responsibility for installation and testing should be identified. The alternatives for who should do hardware and software installation are varied. Choices include consultants, unions, contractors, subcontractors, or current personnel. In many cases, there are three types of installations being made: software, hardware, and the network, and each has its own installation responsibility.

Software should be installed by system programmers in an operations support group in a mainframe installation, and by the software builders for a PC installation. Contracts, whether formal or informal, should state what work is to be done, timing of work, penalties for failure to meet the time requirements, and price. Other items such as number of hours and dates of access to the site might also be included.

Hardware, in a mainframe environment, is managed, ordered, and installed through an operations department. You, as an SE needing equipment, must know what you need, but must trust the operations department to obtain, install, and test the equipment. Most PC computer equipment is simplified enough that special assistance is not usually required. If

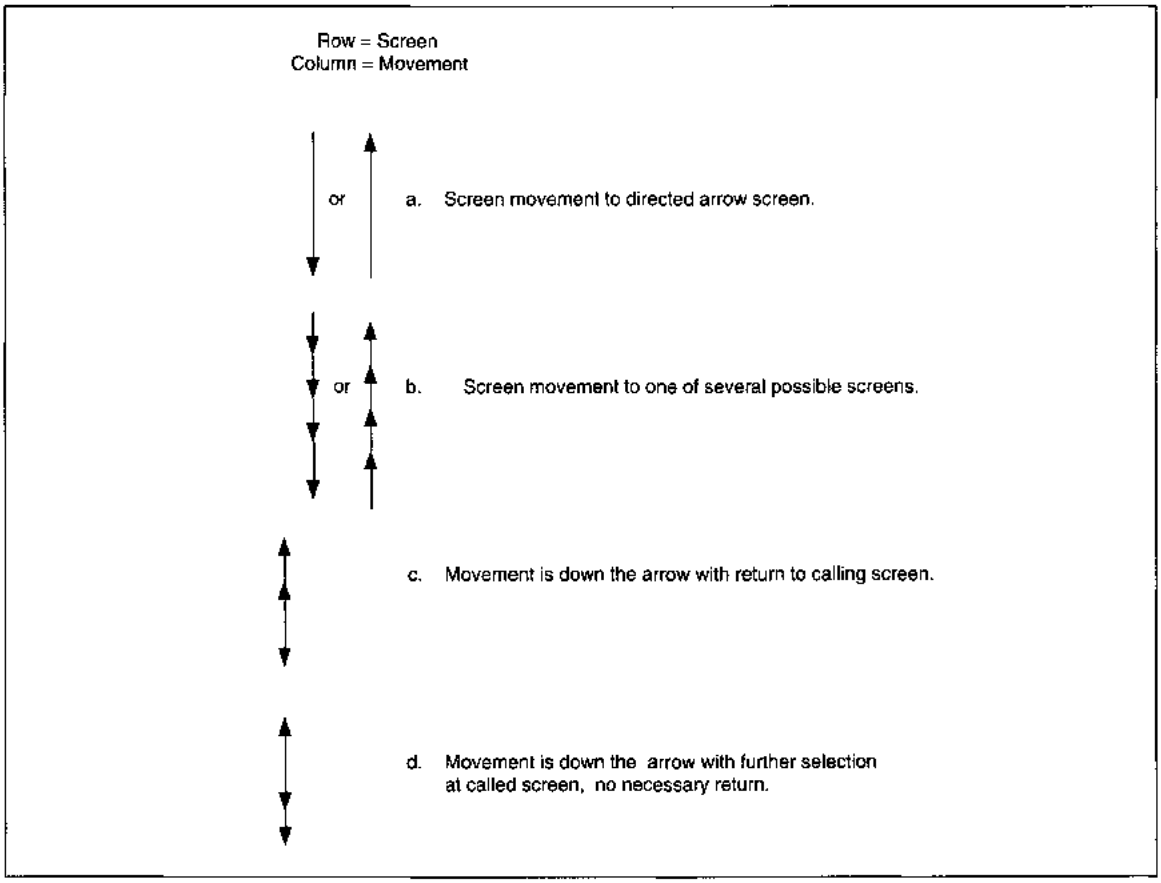


FIGURE 10-56 Dialogue Flow Movement Alternatives

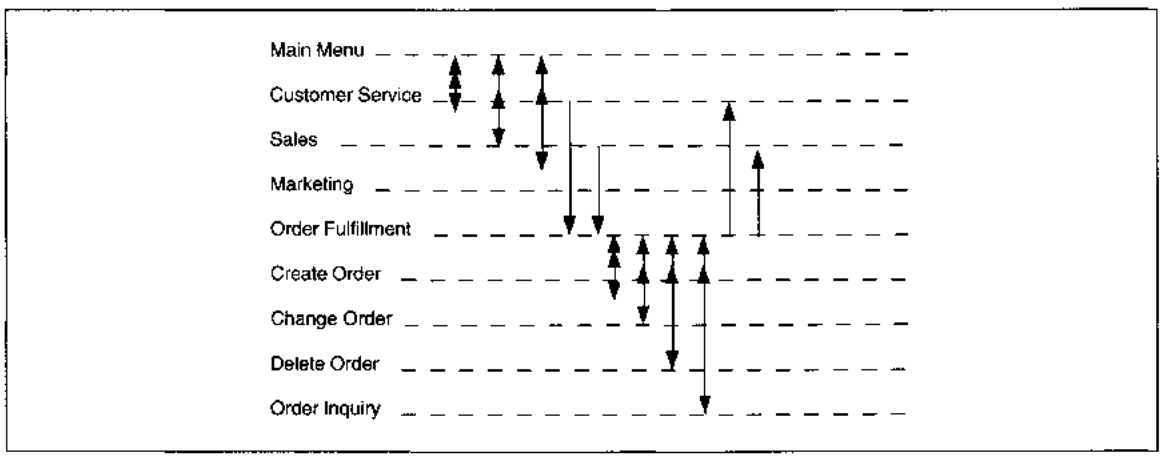


FIGURE 10-57a Example of Restrictive Screen Movement

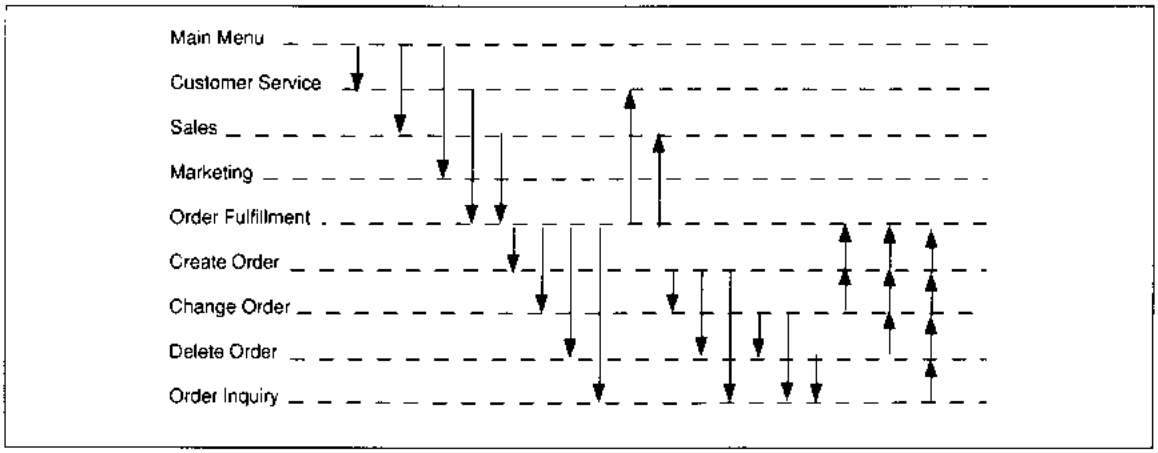


FIGURE 10-57b Example of Less Restrictive Screen Movement

desired, you can usually negotiate with a hardware vendor to *burn-in* equipment and set it up for a small fee. **Burn-in** means to configure the hardware and run it for some period of time, usually 24–72 hours. If there are faulty chips in the machine, 90% of the time they fail during the burn-in period.

At least two terminals or PCs should be configured during installation of network cable for testing the cable. For LAN installation, *hire a consultant if you've never done this before*. The consultant helps you

- define what is to be done
- define required equipment (e.g., cabling, connectors, etc.)
- get permits from the government and building owners
- obtain zoning variances
- identify and hire subcontractors
- supervise and guarantee the work.

As the user's representative, you can prepare the installation for the work to be done. Mark walls

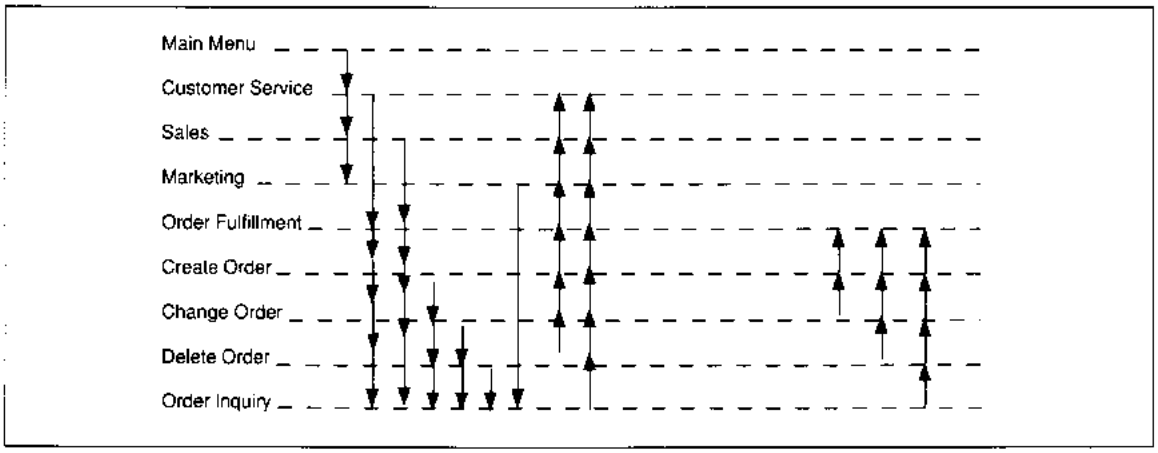


FIGURE 10-57c Example of Less Restrictive Screen Movement

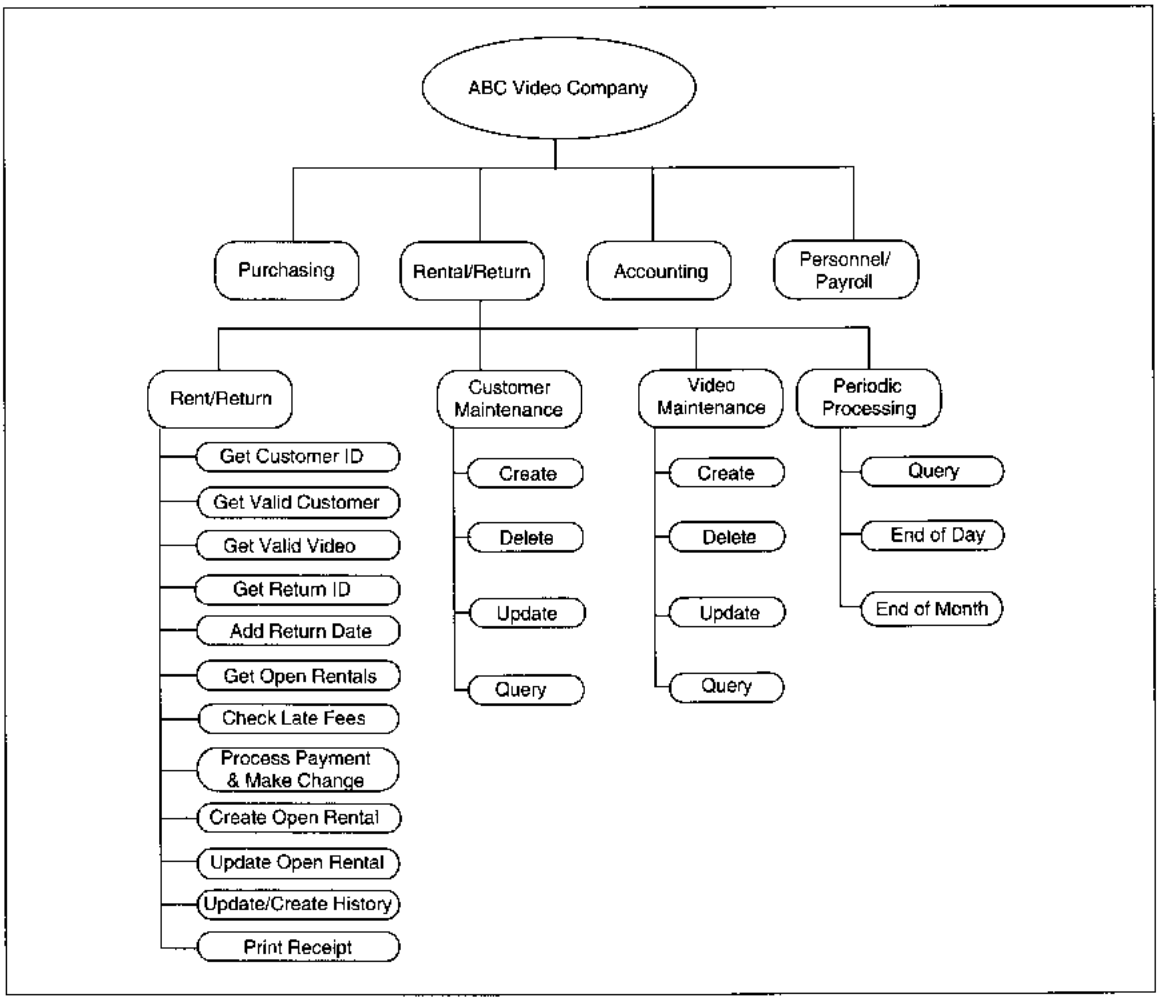


FIGURE 10-58 ABC Process Hierarchy

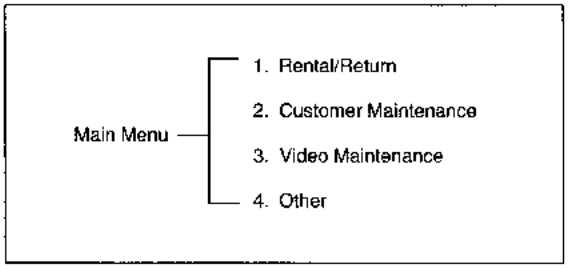


FIGURE 10-59 ABC First-Level Menu Hierarchy

where all wires should be, using colored dots. For instance, you can use blue dots for phone lines, red dots for LAN cable, and green dots for electrical outlets. Number all outlets for identification of wires at the server end. Colored tape shows where cable runs should be placed in false ceilings and walls. Configure one PC, with the network operating system installed, in the location of the file server. As cabling is complete, move the second PC to each wired location, start-up the network, and send messages. Make sure the location is as expected and that

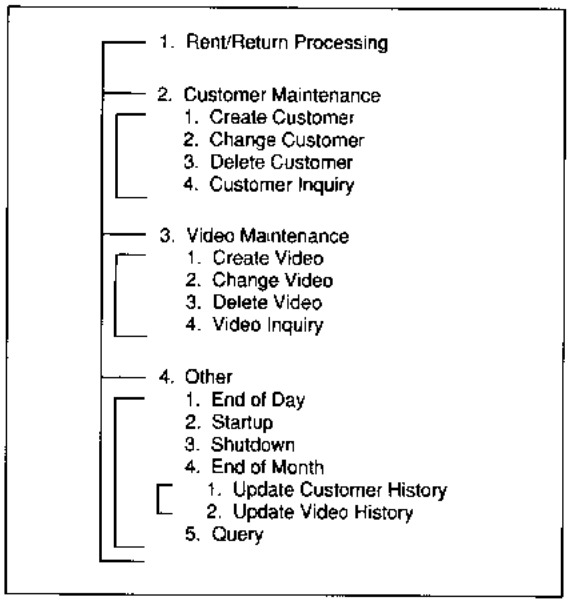


FIGURE 10-60 ABC Menu Structure

the wiring works. Test all wires because *they will be wrong*. Make sure all wiring is correct *before* the electrical contractor is paid and leaves.

The important issue is *to make a choice* of who will do what work long before the work is needed, and plan for what is to be done. Use a lawyer to write all contracts using information provided by you, as the client's representative, and the client.

Timing of installations can be crucial to implementation success. When different types of work are needed, such as air-conditioning and electrical cabling, the work should be sequenced so the contractors are not in each other's way, and in order of need. For instance, a typical sequence might be building frame, building shell, false floor/ceiling framing, electrical wiring, plumbing, air-conditioning, communications cabling, false floor/ceiling finishing, finishing walls, painting, and decorating. Any sequences of work should be checked with the people actually performing the work to guarantee that they agree to the work and schedule.

In general, you want to end testing of all equipment to be available for the beginning of design *at the latest*. This implies that all previous analysis work is manual. If CASE is to be used, the latest possible date for equipment and software availability is the beginning of project work.

Cabling is needed before equipment. Equipment is needed before software. Software is needed before application use. Some minimal slack time should be left as a cushion between dates in case there is a problem with the installation or the item being installed. Leave as big a cushion between installation and usage as possible, with the major constraint being payment strains on a small company.

ABC Video Example Hardware/Software Installation Plan

For ABC, a local area network is to be used. A file server with one laser printer, three impact printers, and five PCs are planned. The LAN will be a

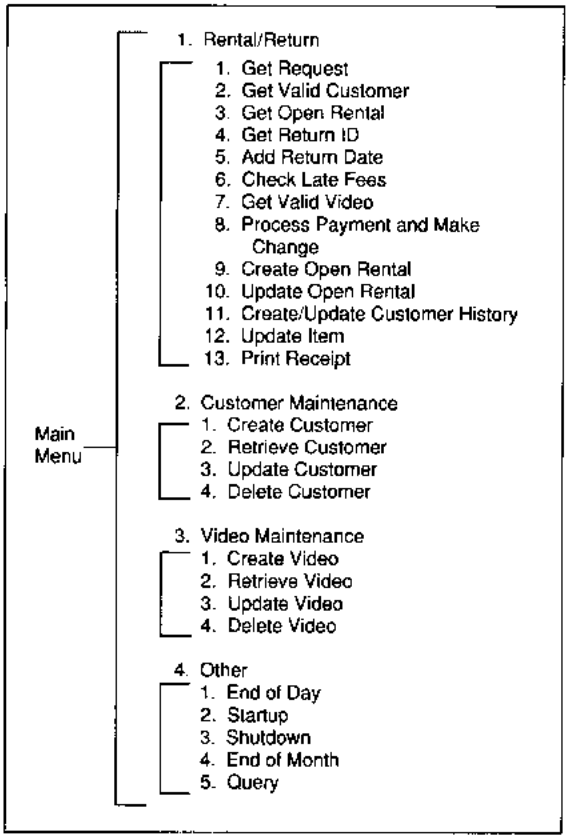


FIGURE 10-61 Incorrect Rental/Return Menu Structure

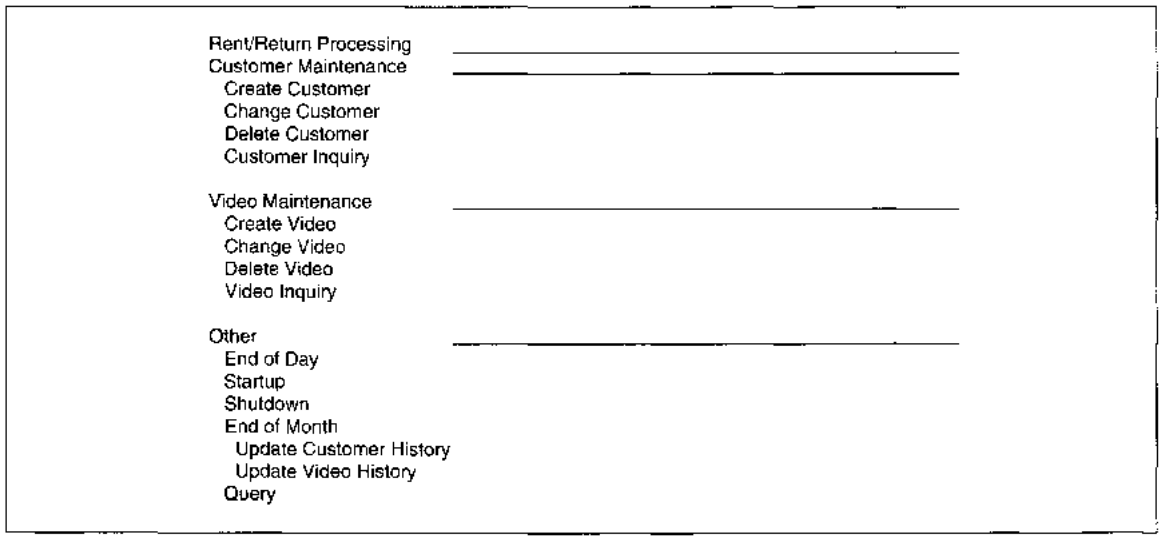


FIGURE 10-62 ABC Dialogue Flow Diagram Menu Structure Entries

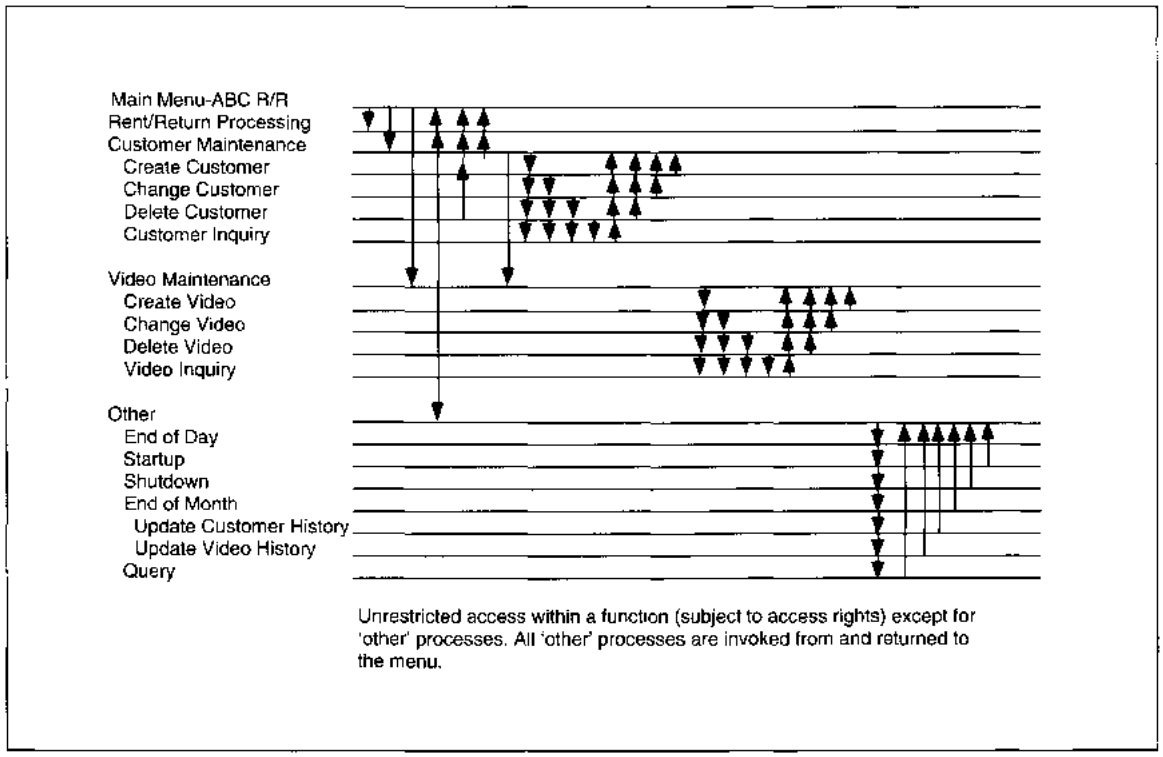


FIGURE 10-63 ABC Dialogue Flow Diagram

Novell ethernet with SQL-compatible DBMS software, Carbon Copy, Word Perfect, Lotus, Norton Utilities, Fastback, and Symantek Virus software. The goal is for all hardware to last at least five years if no other business functions are added to the system. The configuration details are shown in Figures 10-64 and 10-65. There should be adequate capacity to add accounting and order processing software if needed. The current average daily rentals of 600 is expected to double in five years. The current number of customers is 450, and is expected to be 1,000 in five years.

To develop a plan, assume that the current date is January 1, and that the application installation is scheduled for August 1. Design has just begun. The PCs and laser printer were installed five months ago for availability during planning, feasibility, and analysis. The currently installed software includes a CASE tool on two machines, Word Perfect, Norton Utilities, Fastback, the SQL DBMS, and SAM Virus software. The remainder of the software and hardware must be ordered, installed, and tested as part of this plan.

First we determine what we need. A comparison of currently installed items to the list of required items shows the following items need to be planned:

Network cable and connecters
File Server
Novell Software
Network Interface Cards (NICs, i.e., ethernet boards)
Impact printers
Bar Code Reader and Imprinter
Carbon Copy (network version)
Word Perfect (network version)
Norton Utilities (network version)
Fastback
SQL DBMS (network version)
SAM (network version)
Lotus (network version)

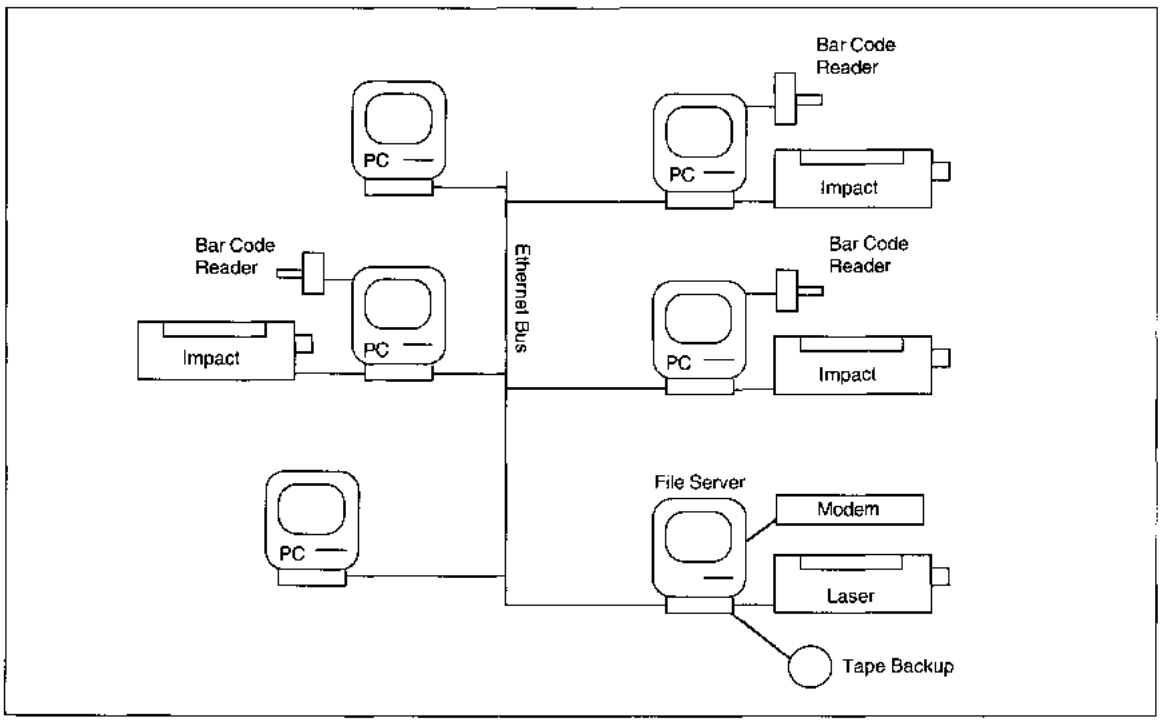


FIGURE 10-64 ABC Configuration Schematic

Hardware Characteristics:	
File server	12 Mb Memory 800 Mb Disk Super 486, SCSI Channel Color monitor
1 Laser printer	8 Page/Minute
3 Impact printers for two-part forms (or 4 cheap lasers with tear-apart forms)	
5 PCs	2 Mb Memory 1.4 Mb Floppy disk for startup No hard disk Local printer (see above)
1 2400 Baud Modem for long distance troubleshooting	
1 Streaming tape backup 100 Mb/Minute	

FIGURE 10-65 ABC Hardware and Software Details

Everything should be ordered as soon as possible to ensure availability. Equipment and software ordering is the first item on the plan.

The group has installed network software before, but not the cable, so they obtain approval from Vic to engage another consultant, Max Levine, from their company to perform that work. Max has been installing mainframe and PC networks for over 20 years and knows everything about their installations and problems. He immediately takes over the network planning tasks. He first obtains a rough idea of the planned locations for equipment, computes cable requirements, and orders cable and connectors. Then, for the plan, he adds tasks for mapping specific cable locations for the installers, for installing and testing the file server, and for installing and testing the cable (see Table 10-6).

At the same time, Mary and Sam work at planning the remaining tasks. Each software package must be installed and tested. These tasks are planned for Sam and one junior person. The tests for all but the SQL package are to use the tool and verify that it works. For the SQL package, Sam and a DBA will install a small, multiuser application to test that the single and multiuser functions are working as expected. Of all the software being used, it is the one with which they are least familiar, so they use the installation test as a means of gaining more experience.

All tasks relating to new equipment and software are scheduled to take place during a six-week period in January and February. This allows several months of cushion for any problems to be resolved; it also allows disruptive installations (e.g., cable) to be scheduled around peek hours and days. The schedule does not show elapsed time, but other work is taking place beside the installations. For instance, design work is progressing at the same time. As the application is implemented and the users have need for the equipment, the PCs and printers are moved to their permanent locations. This occurs in late spring for data conversion. The last stand-alone PCs are scheduled to be added to the network in late July, long before the application implementation date of August 15.

AUTOMATED TOOL SUPPORT FOR DATA-ORIENTED DESIGN

Many CASE tools support aspects of data oriented design (see Table 10-7). Two specifically support IE as discussed in this chapter. The IE CASE tools are Information Engineering Workbench[4] (IEW) by Knowledgeware, Inc., and Information Engineering Facility (IEF) by Texas Instruments, Inc. Both products receive high marks of approval and satisfaction from the user communities. Because of their cost, both products are used by mostly large companies. The products offer enterprise analysis in addition to application analysis, design, and construction (i.e., coding). Both IEF and IEW work on PCs, networks, and mainframes.

A typical IEF installation could include a mainframe version with the centralized repository. Users check out portions of a repository to work with on a PC. Then, when the work is complete and checked on the PC, it is merged with the mainframe reposi-

4 IEW for a OS/2 environment is called the Advanced Development Workbench (ADW).

TABLE 10-6 Installation Plan Items

Due Date	Responsible	Item
1/10	Mary/Sam	Order equipment and software
1/10	Mary/Sam	Order cable and connectors
1/15	ML	Plan cable, printer, PC, server locations
2/1	ML	Install and test file server and one PC
2/1	Sam, Jr. Pgmr.	Install and test impact printers
2/1	Sam, Jr. Pgmr.	Install and test bar code reader and printer
2/5	Sam, Jr. Pgmr.	Install and test Carbon Copy (network version)
2/5	Sam, Jr. Pgmr.	Install and test Word Perfect (network version?)
2/5	Sam, Jr. Pgmr.	Install and test Norton Utilities (network version)
2/5	Sam, Jr. Pgmr.	Install and test Fastback
2/5	Sam, Jr. Pgmr.	Install and test Lotus (network version)
2/5	Sam, Jr. Pgmr.	Install and test SAM (network version)
2/5	DBA, Sam	Install and test SQL DBMS (network version)
2/10	ML, Union Contractor	Install and test cable
2/15	DBA, Sam	Install test application and verify SQL DBMS
5/15	Sam, Vic's LAN Administrator	Move 2 PCs, bar code reader, and 3 printers to permanent locations and test
7/30	LAN Administrator	Move remaining three PCs to permanent locations and test
8/30	Mary, Sam	Remove CASE tools from PCs, remove single user software from PCs and file server

tory for official storage. When the merge takes place, the checked-out items are revalidated for consistency with all mainframe repository definitions. Both products offer automatic SQL schema generation for data. IEF offers automatic code generation for Cobol with imbedded SQL, and can interface to generators for other languages.

IEW and IEF differ in important ways. IEW is more flexible in that it does not require the completion of any matrices or diagrams. However, to take advantage of the interdiagram evaluation software that assesses completeness and syntactic consistency, all matrices and diagrams are required during a given phase. This means that you might not have the diagrams or analyses from planning, but you still can create levels of ERDs within the analysis tool. Similarly, you might not have the analysis tool, so action diagrams can be created directly within the design tool. IEF's strength is that its rigorous adherence to Information Engineering has led to substantive intelligence checking within the software. Both tools easily manage and sort large matrices that result from several of the analyses.

The weakness of the tools differs for each tool. IEW is primarily a PC-based product that can be unstable when used for large projects. IEW also provides DFDs, not PDFDs, and is not a *pure* data methodology tool. A strength of IEW is that Knowledgeware was an IBM partner in its repository defi-

nition; as a result, IEW is compatible with AD-cycle software from IBM.

IEF's strength is also its biggest weakness. IEF requires completion of every table, matrix, and diagram *at this time*.[5] The level of intelligent checking that can be performed is higher than with most other CASE products, but the requirement to complete every table, and so on does not make sense for all projects. TI has recognized the severity of this shortcoming and is increasing the flexibility of the product without compromising its capabilities. The mainframe version of IEF uses DB/2 for repository management and can generate C, Cobol, DB/2, SQL, and other languages' codes.

5 1993

TABLE 10-7 Automated Tool Support for Data-Oriented Methodologies

Product	Company	Technique
Analyst/Designer Toolkit	Yourdon, Inc. New York, NY	Entity-Relationship Diagram (ERD)
Bachman	Bachman Info Systems Cambridge, MA	Bachman ERD Bachman IDMS Schema Bachman DB2 Relational Schema and Physical Diagram
CorVision	Cortex Corp. Waltham, MA	Action Diagram Dataview ERD Menu Designer
Deft	Deft Ontario, Canada	ERD Form/Report Painters Jackson Structured Design (JSD)—Initial Model
Design/1	Arthur Anderson, Inc. Chicago, IL	ERD
ER-Designer	Chen & Assoc. Baton Rouge, LA	ERD Normalization Schema generation
IEF	Texas Instruments Dallas, TX	Action Diagram Code Generation Data Structure Diagram Dialog Flow Diagram Entity Hierarchy ERD Process Data Flow Diagram Process Hierarchy Screen Painter

(Continued on next page)

TABLE 10-7 Automated Tool Support for Data-Oriented Methodologies (*Continued*)

Product	Company	Technique
IEW, ADW (PS/2 Version)	Knowledgeware Atlanta, GA	Action diagram Code generation Database diagram ERD Normalization Schema Generation Screen layout
System Engineer	LBMS Houston, TX	ERD DFD Menu Dialog Transaction Dialog Entity Life History Module Sequence DB2, ADABAS, IDMS, Oracle Table Diagram
Teamwork	CADRE Tech. Inc. Providence, RI	Control Flow Code Generation ERD Process Activation table Program Design Tools Testing Software
vs Designer	Visual Software Inc. Santa Clara, CA	Process flow diagram Action Diagram

SUMMARY

Data-oriented methods assume that, since data are stable and processes are not, data should be the main focus of activities. First, design focuses on the usage of data to develop a strategy for distributing or centralizing applications. Several matrices summarize process responsibility, data usage, type of data used, transaction volumes, and subjective reasons for centralizing or distributing data.

Next, processes from a process hierarchy diagram are restructured into action diagrams in design. The details of process interrelationships are identified from the PDFD and placed on the action diagram. Each process is fully defined either in a diagram or in the data dictionary. Process details are grouped into modules and compared to existing modules to determine module reusability. Modules are analyzed from a different perspective to reflect concurrency opportunities or requirements on the action diagram. Entities are added to the diagram and related to processes. Lines connect individual processes to attributes to complete the action diagram specification of each application module. For manually drawn diagrams, an optional activity is to identify screens and link them to attributes and processes, to give a complete pictorial representation of the on-line portion of the application.

Data-oriented design focuses on the needs for security, recovery, and audit controls, relating each topic to the data and processes in the application.

The menu structure and dialogue flow for the application are defined next. The menu structure is constructed from the process hierarchy diagram to link activities, processes, and subprocesses for menu design. The structure can be used to facilitate interface designers' application understanding. The dialogue flow documents the flexibility or restric-

tiveness of the interface by defining the allowable movements from each menu level (from the menu structure) to other levels of menus and processing.

Finally, installation plans for all hardware and software are developed. A list of tasks is defined, responsibilities are assigned, and due dates are allocated to the tasks.

There are two fully functional CASE tools that support data-oriented methodology as discussed in this chapter, IEW and IEF. They are popular in companies that use data-oriented methods.

REFERENCES

Date, C. J., *An Introduction to Database Systems*, Vol. 1, 5th edition. Reading, MA: Addison-Wesley, 1990.

Finkelstein, Clive, *An Introduction to Information Engineering: From Strategic Planning to Information Systems*. Reading, MA: Addison-Wesley, 1989.

Knowledgeware, Inc., *Information Engineering Workbench™/Analysis Workstation. ESP Release 4.0*. Atlanta, GA: Knowledgeware, Inc., 1987.

Loucopoulos, Pericles, and Roberto Zicari, *Conceptual Modeling, Databases and CASE: An Integrated View of IS Development*. NY: John Wiley & Sons, 1992.

Martin, James, *Information Engineering, Vol. 3: Design and Construction*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1990.

Martin, James, and Carma McClure, *Diagramming Techniques for Analysts and Programmers*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1985.

Texas Instruments, *A Guide to Information Engineering Using the IEF*. Dallas, TX: Texas Instruments, 1988.

KEY TERMS

action diagram
application security
audit control
backup
burn-in
candidate for template
code generator
computer verification
concurrent processes
condition bracket
control point
controlled redundancy
data distribution by location matrix
data security
data usage by location matrix
denormalization
dependent concurrent processes
dialogue flow diagram
$D_R/D_C > N - 1$
$D_R < N/D$
federation
full backup
hardware installation plan
horizontal data partitioning
incremental backup
independent concurrent processes
menu structure
normalization
off-site storage
physical security
procedural template
process/location matrix
recovery
recovery procedures
repetition bracket
replication
security plan
selection bracket
sequence bracket
sight verification
subset partitioning
structural relationships
transaction volume matrix
vertical partitioning

EXERCISES

1. Analyze Figures 10-8 to 10-11 and Table 10-1. Develop and present a recommendation for centralization or distribution. Define all recommended data and software locations. Explain your reasoning for each choice.
2. Complete the action diagram for miscellaneous processing. Define the contents of the *EOD* File.
3. Go visit a local small business such as a video store, restaurant, or supermarket. Assess their security and physical layout. Develop a list of recommendations you would make if installing a computer system for this company. Present your findings to the class and the reasons for your recommendations.

STUDY QUESTIONS

1. Define the following terms:
   action diagram
   code generator
   control point
   controlled redundancy
   recovery
   repetition bracket
   replication
   security
   transaction volume matrix
   vertical data partitioning
2. What are structured programming tenets and why are they important in IE design?
3. What is the purpose of an action diagram?
4. Discuss this assertion: "Normalization to the third normal form and higher is always desirable for a physical database."

5. Define the four types of database distribution.
6. Describe how security, recovery, and audit controls complement each other.
7. There are six types of disasters considered in recovery planning. What are they and what data/application problems do they cause?
8. What are common methods of securing data against unwanted access?
9. What is the purpose of off-site storage? How off-site should off-site storage be?
10. What are the trade-offs in security and recovery design? Why not build a fortress to secure everything?
11. Discuss the differences between full and incremental backup.
12. What features of computers make audit controls difficult?
13. How is a menu structure diagram constructed? What is its purpose?
14. How can dialogue flow diagrams be used to partially provide for access control?
15. What are the structural relationships on an action diagram? Where do they come from?
16. List the steps in developing an action diagram.
17. For what types of applications does concurrency analysis become important?
18. What is reusability analysis? Why is it important?
19. Why, when developing an action diagram, must the processes sometimes change from what is on the PDFD?
20. Describe the matrices and formulae used to determine centralization or distribution of data. In the absence of subjective reasoning, would the matrices and formulae lead to a rational decision? Why or why not?
21. Why is an installation plan important? How can installation be used as a teaching exercise for junior people?
22. What aspects of physical environment should be considered in an installation plan for new equipment?
23. Describe the diagram interrelationships for data and processes from enterprise analysis to analysis to design.

EXTRA-CREDIT QUESTION

1. Analyze the Advanced Office System (AOS) case in the Appendix. Develop all of the distribution matrices and subjective reasoning for/against distribution. Develop recommendations and explain your reasoning for each choice.

CHAPTER 11

OBJECT-ORIENTED ANALYSIS

INTRODUCTION

In this chapter, we reanalyze the requirements for the ABC Video's rental processing application using an object-oriented approach. This approach requires the definition of many new terms and a fundamentally different way of thinking about applications and their components. Keep in mind that object orientation is very much an immature methodology class that is still evolving.

Several distinct schools of thought have emerged on how best to represent object thinking. Since they discuss the same topics, the schools have considerable conceptual overlap. The first school is object orientation that uses many graphical forms paralleling those of other methodologies. Authors using this approach are Coad and Yourdon and Rumbaugh et al. (see References at the end of the chapter). The second school of object orientation is tabular, using mainly tables to list and define objects and their parts. This approach is used by Booch and Berrard. The graphical methodologies lack the reasoning processes of Booch's approach, while the tabular method is not easily communicated because of the extensive detail generated. Therefore, the Booch and Coad and Yourdon approaches are both modified and integrated throughout this discussion. Since few people dispute the need for analytical rigor and graphical richness, this type of object methodology is preferable to either one or the other approach used singly.

CONCEPTUAL FOUNDATIONS OF OBJECT-ORIENTED ANALYSIS

Two key concepts define object orientation: encapsulation and inheritance. **Encapsulation** is a property of programs that describes the complete integration of data with legal processes relating to the data. In addition, encapsulated objects have public and private *selves* (see Figure 11-1). The **public part** of an object defines what data are available in the object and the allowable actions of the object. The **private part** of an object defines local, object-only data and the specific procedures each action takes.

The second major property of object orientation is inheritance. **Inheritance** is a property that allows the generic description of objects which are then reused by *related* objects. Objects are grouped into **classes** that are defined as like objects that have *exactly* the same properties, attributes, and processes. Object **classes** are arranged in **hierarchies** of relationships. Within a hierarchy, objects at lower

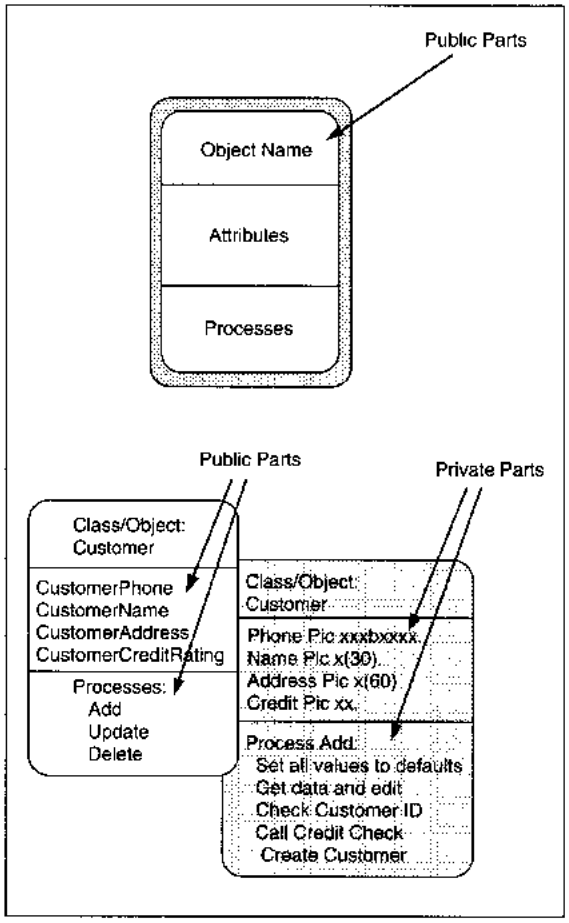


FIGURE 11-1 Encapsulated Object: Public and Private Parts

levels inherit the data and processes of the superior classes. Hierarchies can also be linked to form lattice-like networks of hierarchies of objects.

An example of an object class is *employees* (see Figure 11-2). Each employee has a name, address, social security number, and so forth. Some employees are also managers. Managers are a subclass of the employee class. By subclass, we mean that managers have the same properties as employees (because they *are* employees), and that, in addition, they have additional properties that only managers have. Managers might have an additional subclass of managers who are on a management committee. The management committee subclass is said to have **multiple inheritance** because it inherits the properties, attributes, and processes of employees *and* managers as well as having its own.

Object orientation is an approach to thinking about problems that, when properly applied, represents a substantive improvement in the resulting analysis, design, and code modules. For 30 years, we have known that the key goal of software engineering is to manage the complexity of the problems we automate. We have also known that the best way to manage complexity is to decompose the larger problems into intellectually manageable, small tasks, that hide their internal workings from other modules, and that are coupled only by communicating messages.[1] These are the goals of analysis and design that lead to well-structured and well-formulated programs and modules. Object orientation, when properly applied, appears to come closer to automatically resulting in these desirable outcomes than other ways of thinking.

Thinking in objects requires a paradigm shift. A paradigm is a generally agreed upon way of thinking about a situation. In the process methods we concentrate on *functional* thinking, or the steps taken to perform some procedure. In data methods, we concentrate on *entity* thinking, or the data objects and their interrelationships that dictate much processing. Entity thinking is a difference in degree rather than a difference in kind—a foreground/background shift. We move from processes that change data to emphasizing data that require processing (see Figure 11-3).

1 See the works of CAR Hoare, David Parnas, Nicklaus Wirth, and Edsger Dijkstra. In particular, the discussions are summarized in the following references: Hoare, C. A. R., "The Emperor's Old Clothes," Dijkstra, Edsger, "The Humble Programmer," both in AMC Turing Lecture Awards, NY: ACM Press and Addison-Wesley, 1987, and Parnas, David, "A Technique for Software Module Specification with Examples," *Communications of the ACM*, Vol. 15, #5, May, 1972, pp. 330–336; Parnas, David, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, Vol. 15, #12, December 1972, pp. 1053–1058; and Wirth, Nicklaus, "Program Development by Stepwise Refinement," *Communications of the ACM*, Vol. 14, #4, April 1971, pp. 221–227.

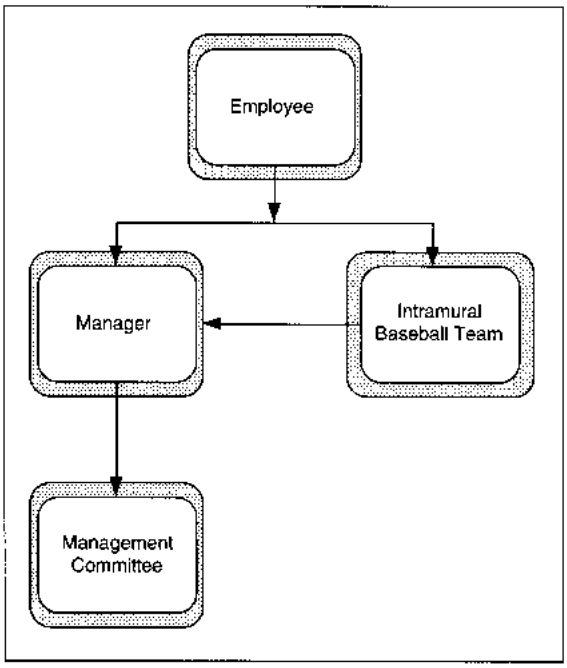


FIGURE 11-2 Example Object Class Hierarchy

In *object* thinking, we can identify data and processes somewhat independently, but they are *married* early on and must be thought of together, forever after, to reason properly about their behavior and contents. The paradigm shift to object thinking is from thinking of data and processes as separate to thinking of data and processes as one.

Several times in this discussion, we have mentioned the term "if properly applied." Object orientation is no different than any other methodology in that it requires consistency and correct reasoning to result in the desirable properties described. When improperly applied, object orientation results in a badly designed application that might actually be less efficient than the same application designed poorly using some other methodology.

DEFINITION OF OBJECT-ORIENTED TERMS

Object orientation is based on the notion of objects which encapsulate both data and processes on that

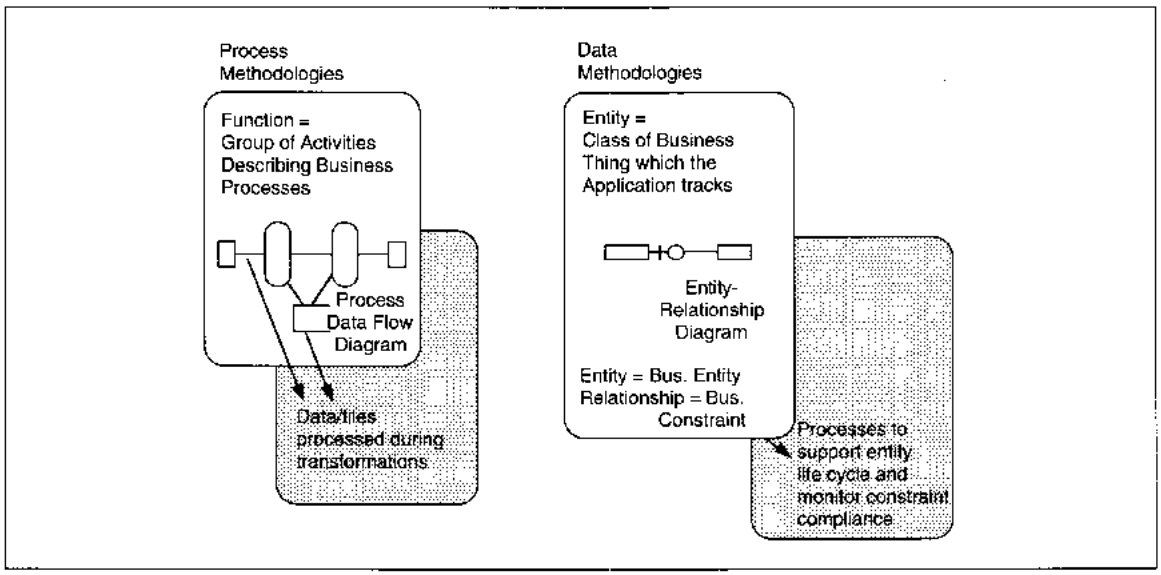


FIGURE 11-3 Process and Data Methodologies as Flip Sides of the Same Paradigm

data. An **object** is an entity from the real world whose processes and attributes (that is, the data) are modeled in a computerized application.

Processes are variously called functions, actions, services, programs, methods, properties, or modules; these terms may or may not have the same meaning to the people using them. For that reason, we stick to the term *process* to mean the transformational program language code that acts on its object data.

An **abstract data type (ADT)** is the name used in some languages (e.g., C) for the new, user-defined data type that *encapsulates* definitions of object data plus legal processes for that data. In this text, we use the terms encapsulated object, object, and abstract data type interchangeably.

The major analysis activities focus on defining objects, classes, and processes. Class/objects are the lowest level of logical design entity. **Class/objects** define a set of items which share the same attributes and processes, and manage the instances of the collection. The **class** defines the attributes and processes; the objects are the instances of the class definition.

There are different types of class-object relationships. First, classes can occur without having any real data associated with them. Classes whose instances are other classes are called **meta-classes**. For instance, we might define a class *Customer* with subclasses for *CashCustomer* and *CreditCustomer*. The class is a meta-class; the subclasses are class/objects which manage the data of *Customer*.

Classes can be composed of class/objects to describe a composition relationship of *whole* and *part*. A **whole class** defines the composed object type. The **part class** defines all the components of the whole class. For instance, a car, as a whole class, contains parts that include motor, wheels, doors, seats, and so on.

Classes can also be defined to allow specialized versions of an item. The meta-class is called a *generalization class*, or gen class for short. The subclasses are called *specialization*, or spec, *classes*. A **generalization class** defines a group of similar objects. For instance, vehicle is a generalization on car. The **specialization class** is a subclass that reflects an *is-a* relationship, defining a more detailed description of the gen class. For instance, a car, truck, or tank are all specializations of the general class vehicle. These could be further specialized themselves. For instance, car could have specializations by type car: full-size, mid-size, or economy.

Each type of class and its subclasses form a hierarchic, lattice-like arrangement of relationships. Through the relationships, the lower-level classes inherit the data and processes of the related higher-level classes. Thus, if we were to refer to an *economyCar* object, we would have information and processing for vehicles, cars, and economy cars all available.

Messages are the only legal means of communications between encapsulated objects. Messages are clear in their intention but not clear in their implementation, which is completely determined by the language (see message types in Figure 11-4). For instance, at the moment Ada does not implement message communication. In this text, a **message** is the unit of communication between two objects. Messages contain an addressee (that is, the object providing the process, also called a service object), and some identification of the requested process.

A major difference between object orientation and other methodologies is the shifting of responsibility for defining the data type of legal processes from supplier (or called) objects to client (or calling) objects. This shift, along with the notions of inheritance and dynamic binding, support the use of **polymorphism**, which is the ability to have the same process take different forms when associated with different objects. Dynamic binding is a language property that selects actual modules to execute during application operation. The concept is completely described in Chapter 12.

A **supplier object** is one that performs a requested process. A **client object** is one that requests a process from a supplier. For instance, I might need to have a date translated from month-day-year format to year-month-day format. As a client object, I request the translation of the supplier object and pass it the date to translate. If the language supports polymorphism, I also pass the data type of the date to be translated.

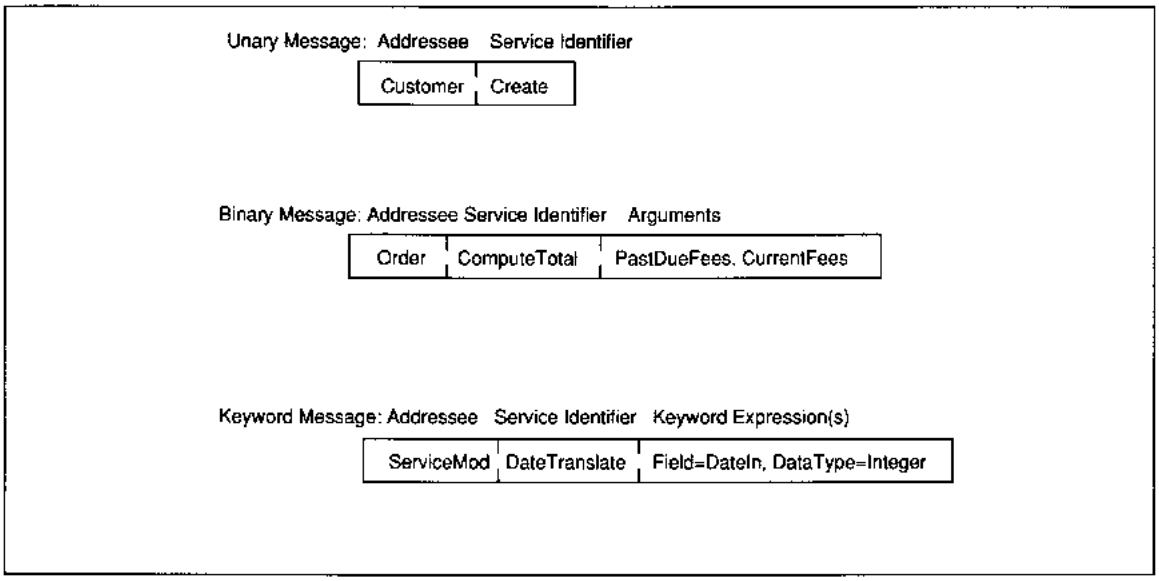


FIGURE 11-4 Example of Message Types

An example of polymorphism is, for instance, a process to perform comparison of two items to identify the 'larger' of the two. One object might be alphabetic, requiring a logical comparison; another object might be decimal numeric, requiring a numerical comparison; a third object might be an array, requiring numerical array comparisons. This polymorphic object has three implementations of its process to *compare and determine the larger of two items*. The client object requests a specific comparison process, here either alpha, numeric, or array.

To summarize the terms, objects are encapsulations of data and processes that have both public and private parts. Objects can communicate via messages which differ by language. Objects are arranged into classes of similar objects, and can belong to more than one class. By the property of inheritance, an object exhibits the attributes and provides the services of the classes of which it is a part. Polymorphism is a desirable property of objects but requires a client-server view of objects along with dynamic binding capabilities.

OBJECT-ORIENTED ANALYSIS ACTIVITIES

The documentation for object-oriented analysis[2] includes a series of tables and graphics (Figure 11-5). The tables are lists that document individual components of the analysis—objects, processes (and their assignment to objects), attributes, and classes. The graphics show relationships between objects and object classes, state transitions of intraobject changes in the application, and time-ordering interobject–event processing. Each documentation representation is elaborated by tracing the object-oriented analysis of ABC Video's rental processing system.

2 The analysis documentation builds primarily on the work of Booch [1983, 1991] and Berrard [1985]. The Class diagrams, subject summary, gen-spec and whole-part diagrams are all from Coad and Yourdon, 2nd ed. [1990].

Summary Paragraph	Provides a brief summary of all major functions to be performed.
Tables/Lists	
Object List	Contains potential objects (nouns) from the paragraph. Each entry is evaluated to determine that it is an object, to classify it as solution space or problem space related, and to assign it a unique, formal name.
Process List	Contains potential processes (verbs) from the paragraph. Each is evaluated to determine that it is a process, to classify it as solution space or problem space related, and to assign it a unique, formal name. All solution space class/objects are tentatively related to processes and the relationships are evaluated.
Object-Attribute List	Contains field name attributes with each object they describe. Each class/object's entries are normalized and other class/objects are created as needed.
Process-Attribute List	Contains formulae, constraints on processing, and state/status changes for each process as required; some processes have no attributes.
Diagrams	
Object Relationship Diagram	Identifies objects with connecting lines showing different types of interobject relationships.
Class Hierarchy Diagram	Shows objects arranged in one or more lattice hierarchies to link shared data/processes and to depict inheritance of those data/processes.
Generalization/Specialization Structure Diagrams	Depicts objects which express *is-a* relationships. This diagram is optional.
Whole/Part Structure Diagrams	Depicts objects which are compositions for which the *whole* class is composed of one or more of the *part* subclasses. This diagram is optional.
Subject Summary Diagram	The highest level of independent classes or class/objects in each leg of a hierarchy are promoted to *subjects* for inclusion in this diagram which provides a summary of the classes in the application. This diagram is optional.
State Transition Diagram	Contains system states (i.e., statuses) and the events (process outcomes) that cause those states to exist.

FIGURE 11-5 Summary of Object-Oriented Analysis Documentation

Develop Summary Paragraph

Rules for Summary Paragraph

The first, and most important, step of object-oriented analysis is to develop a single summary paragraph describing the problem. The purpose of the paragraph is to focus your attention on the most concrete, yet high-level description of the problem. Hidden within a good summary are the main class/objects and the main processes to be provided by the application. In a large application, development will be iterative with a series of more detailed summary paragraphs developed to elaborate the individual sentences from a summary. In a smaller problem, like ABC Video's, we only need one level of summary.

The guidelines for writing the paragraph are as follows:

1. Write only declarative sentences of the form:
   Noun–Verb
   Noun–Verb–Object
   Verb–Object
2. For ease of quality assurance, write each sentence on its own line.

3. Review the paragraph carefully to ensure:
   - All desired functions are represented.
   - All major information and processes are identified.
   - All sentences are at the same level of abstraction, detail, and importance.

These are guidelines because the development of the paragraph is an individual activity performed by the SE with the user, and specific to each application. It is one result of interviews and other data collections that take place before and during analysis. Object orientation assumes that you have the requirements for the application in hand and understand what the application is supposed to do.[3] There are no graphical representations for paragraph information.

ABC Video Example Paragraph

Refer back to Chapter 2 for the description of ABC Video's rental processing requirements. The initial paragraph reads:

> Customers select one to n videos for rental. Customer phone number is entered to retrieve customer data and create an order. Bar code IDs for each tape are entered and video information from inventory is displayed. The video inventory file is updated (decrease the count of available copies by one). When all tape IDs are entered, the system computes the total. Money is collected and the amount is entered into the system. Change is computed and displayed. The rental is created, printed, and stored. The customer signs the rental form, takes the tape(s), and leaves. To return a tape, the video Bar Code ID is entered into the system. The rental is displayed and the tape is marked with the date of return. If past-due amounts are owed, they can be paid at this time; or the clerk can select an option which updates the rental with the return date and calculates past-due fees. Any outstanding video rentals are displayed with the amount due on each tape and a total amount due. The past-due amount must be reduced to zero when new tapes are taken out.

1. Customers select one to n videos for rental.
2. Customer phone number is entered to retrieve customer data and create an order.
3. Bar code IDs for each tape are entered and video information from inventory is displayed.
4. The video inventory file is updated (decrease the count of available copies by one).
5. When all tape IDs are entered, the system computes the total.
6. Money is collected and the amount is entered into the system.
7. Change is computed and displayed.
8. The rental is created, printed, and stored.
9. The customer signs the order form, takes the tape(s), and leaves.
10. To return a tape, the video Bar Code ID is entered into the system.
11. The rental is displayed and the tape is marked with the date of return.
12. If past-due amounts are owed, they can be paid at this time; or the clerk can select an option which updates the rental with the return date and calculates past-due fees.
13. Any outstanding video rentals are displayed with the amount due on each tape and a total amount due.
14. The past-due amount must be reduced to zero when new tapes are taken out.
15. For new customers, the customer information is entered into the system and added to the customers.
16. For new videos, the video information is entered into the system and added to inventory.

FIGURE 11-6 Initial Paragraph in Numbered Sentence Format

> For new customers, the customer information is entered into the system and added to the customers. For new videos, the video information is entered into the system and added to inventory.

The paragraph is reformatted as a numbered list of sentences (see Figure 11-6). This numbered sentence format is recommended because it simplifies discussion, quality assurance, and reviews.

Once the paragraph is drafted, you examine each sentence carefully to make sure all the pertinent information is present and clearly stated. In this paragraph, there is confusion about a 'new order' in sentence 2 and an 'outstanding video rental' in

3 Lorenz [1993] recommends the development of 'use cases' which track all variations of each transaction through its processing. This is, in essence, what you do in interviews with users during a normal data collection activity.

sentence 13. You ask yourself, What do we mean by an 'order'? If you do not know, you may need to ask the client what he means by an order.

Vic wants an order to have information that is linked to video information whenever customers have any videos out on rent, that is, they are an 'active' customer. An order should contain information about all current rentals, dates returned, and late fees. Any other fees owed, for instance, penalties assessed for late payment, should also be present until they are paid. In other words, Vic uses the word *order* to describe what we have termed a *rental*. This confusion is cleared up immediately because different words for the same items always cause confusion. Vic does not mind changing the term *order* to *rental*. He uses the term *order* because he thinks his business is similar to order-entry processing which he managed in an old job. The major differences between these two activities is that Vic has a cash business and order-entry applications are usually used in accrual accounting businesses that link to accounts receivable accounting. Vic is correct; there is similarity between rentals and order processing, but the term rental fits this particular business and will be used.

To be consistent in the use of terms, we modify sentence 2 to read:

2. Customer phone number is entered to retrieve customer data either to create a rental or to retrieve active rentals.

This change also implies a status for *rentals* of *'active'* or *'inactive'* which we will need to further clarify.

The term *video information from inventory* in sentence 3 should be more specific. Knowing the actual fields to be displayed will be helpful in the class analysis and in attribute definition. Upon further conversation with Vic, you change the information to read:

3. Bar code IDs for each tape are entered.
3a. Video name and rental price from inventory are displayed.

The next unclear issue is: When is money collected for new rentals? Can a customer rent a video, pay past-due fees, and pay for the current video rental upon its return? Again, we go back to Vic, the client, and ask him what he wants.

Vic says, "I would like as little bureaucracy as possible in this system. Since 80% of videos are returned on time, I want new rentals paid in advance—when they are rented. About 90% of my customers return their videos through a slot in the door during nonworking hours. Any videos that have late fees are checked in, and a note of past-due fees must be made.

"For legal reasons, I must be able to prove how past-due fees are derived. To meet this obligation, the past-due fee amount, rental date and return date must all be maintained.

"Also, I do not want to encourage 'deadbeats' who do not pay for their rentals, so I insist that any outstanding fees be paid before any new rentals."

With the above information supplied by Vic, we evaluate the sentences dealing with payments. Although they remain somewhat ambiguous, they would be sufficient if we chose not to change them. The information is clearer if sentences 13 and 14 are moved between sentences 2 and 3 and are renumbered 2a and 2b for the present.

One remaining ambiguity might be computations for the *'total'* and *'change.'* If the computations are understood, they are not required in the paragraph. We do not *need* the computations for the paragraph, but we do need it soon. So, if the computations are not understood, you again go back to Vic and ask how the computations are performed.

Vic: "There are two basic totals: one for settling past-due fees and one for the current rental. They may be computed together as the rental total equal to the sum of all past-due items, fees, taxes, and current rentals. Change is computed as the rental-total less amount paid."

Vic's definition of the rental-total raises a new question about the paying of late fees and sentence 2b. If past-due fees must be settled before any current rentals are allowed, how can you add the information together to create the rental-total?

Old #	New #	Sentence
2.	2.	Customer phone number is entered to retrieve customer data either to create a rental or to retrieve an active rental.
2a.	3.	Any outstanding video rentals are displayed with the amount due on each tape and a total amount due.
2b	Note	The past-due amount must be reduced to zero when new rentals are made.
3.	4.	Bar code IDs for each tape are entered.
3a.	5.	Video name and rental price from inventory are displayed.
5.	6.	When all tape IDs are entered, the system computes the total (= $\sum$ past-due fees + $\sum$ other fees + $\sum$ current video rental fees).
6.	7.	Money is collected and the amount is entered into the system.
7.	8.	Change is computed (= amount entered—order-total) and displayed.
	9.	If the change amount is negative, that is, the customer did not pay for all fees, the clerk asks for more money.
	10.	If the customer gives the clerk more money, return to step 7, else, when the clerk presses an order complete key, the system 'pays-off' the fees on a first-in-first-paid order until the amount entered is used up. The rental is redisplayed. Past-due items 'paid-off' are marked paid and the status of the current video rentals are either paid or due.
	11.	If the amount entered paid for one or more current rentals, they are updated as paid and the videos are given to the customer; else when the clerk presses the rental complete key again, the current rentals not paid for are removed and placed back in stock.
4.	12.	When the clerk presses a rental complete key (to be defined by the system), this order is complete and the video inventory file is updated (decrease the count of available copies by one).
8.	13.	The rental is stored and printed.

FIGURE 11-7 Partially Renumbered Paragraph

"Oh," says Vic, "I meant that the clerk should not give the customer the video tapes until all of the past-due fees plus current rental fees are paid. They can still process the current rentals on the computer at the same time. Remember, my motto is no burcaucracy."

This new information does change at least the order of sentences 2 through 8 (see Figure 11-7). At the end of the paragraph, add the following so the information is not lost.

> 2b. NOTE: The amount paid less change must be equal to the rental-total or the clerk should politely refuse to give the customer the current tapes.

The new sentences 9, 10, and 11 add needed information to our understanding of the problem, but now they are at a different level of detail from the other sentences. They constitute *processing* that accompanies *change*. So, to keep the level of abstraction consistent, they should be removed from this paragraph and kept for use during the next iteration of *change processing*. To indicate that other steps are needed to process change, modify sentence 8 to read:

> 8. Change is computed (= amount-entered—rental-total), displayed, and further processed by the clerk as required.

At the moment, the final paragraph for ABC Video's rental processing system should read like the

one in Figure 11-8. All major functions, data entities, information sources, and destinations are identified. All sentences are at the same level of abstraction, detail, and importance.

Identify Objects of Interest

Rules for Identifying Class/Objects

The next step is to identify and analyze all of the class/objects of interest. The items are called class/objects because they identify a collection (class) of like instances (objects). The rules are summarized here:

1. Underline all nouns in the summary paragraph.
2. List the underlined verbs on a separate sheet of paper, using the exact same sequence and spelling as in the paragraph.
3. Evaluate each noun to make sure it *is* an object. (Common errors are to include attributes objects, that are not of interest to the solution of *this* problem, or physical objects we do *not* keep information about).
4. Determine whether the object is in the *solution space* (must be present both to describe the problem *and* to develop a solution) or the *problem space* (must be present to describe the problem).
5. Name each unique object in the solution space. Ignore the processes in the problem space. Use the convention '*=name*' to identify duplicates of already named objects and to show that you know it is a duplicate.

The mechanics of the identification are to underline the nouns in the paragraph. Once the underlining is done, make a list of the nouns on a separate sheet of paper. When making the list, keep the nouns in exactly the same sequence as they occurred in the paragraph and use exactly the same spelling as occurred in the paragraph!

Next, evaluate each noun to make sure it *is* an object. Evaluate similar criteria for identifying entities in the data methodology: people, places, events, applications, organizations, or other abstractions about which the application must keep information

To rent tapes,

1. Customers select one to *n* videos for rental.
2. Customer phone number is entered to retrieve customer data either to create a rental or to retrieve an active rental.
3. Any outstanding video rentals are displayed with the amount due on each tape and a total amount due.
4. Bar code IDs for each tape are entered.
5. Video name and rental price from inventory are displayed.
6. When all tape IDs are entered, the system computes the total (= Σ past-due fees + Σ other fees + Σ current video rental fees).
7. Money is collected and the amount is entered into the system.
8. Change is computed (= amount entered – order-total), displayed, and further processed by the clerk as required.
9. When the clerk presses an 'order-complete' option key (to be defined by the system), this rental is complete and the video inventory file is updated (decrease the count of available copies by one).
10. The rental is stored and printed.
11. The customer signs the order form, takes the tape, and leaves.

To return a tape,

12. The video bar code ID is entered into the system.
13. The rental is displayed and the tape is marked with the date of return.
14. If past-due amounts are owed, they can be paid at this time; or the clerk can select the 'order-complete' option which updates the rental with the return date and calculates past-due fees.

To add a customer:

15. Enter customer information.
16. Create customer.

To add a new video:

17. Enter video information.
18. Create video inventory.

NOTE: The entire amount owed must be paid before any rentals are allowed. That is, the amount paid less change must be equal to the rental total or the clerk should politely refuse to give the customer the current tapes.

FIGURE 11-8 Final Paragraph for ABC Order Processing

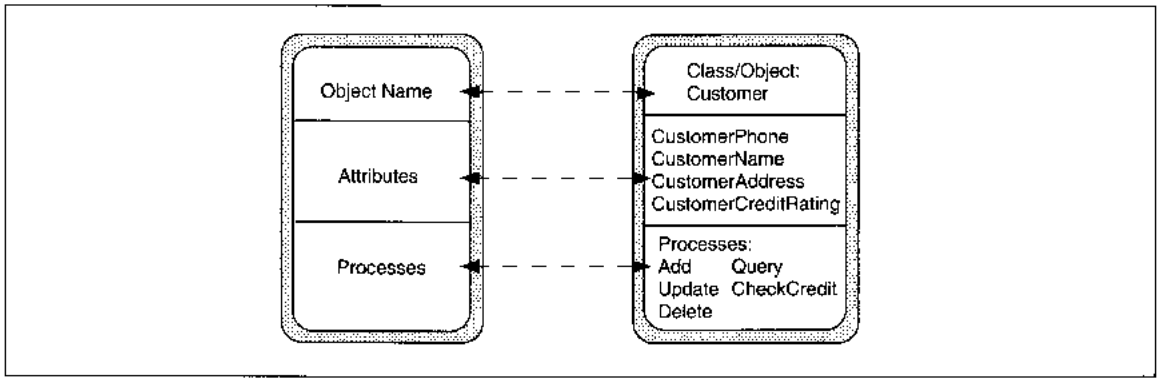


FIGURE 11-9 Class/Object Diagram Format

or for which processing is required. If the items in the list fit any of these criteria and pass the other tests, keep them on the list.

There are no hard and fast rules for this process, only heuristics or rules of thumb. Ask yourself the following sets of questions. Does the noun identify something from the real world you want to store information about? If so, keep going. If not, it is not an object in this system, so cross it off.

Does the noun identify something that takes on values itself, for instance, a social security number, balance, or rental total? If so, these are attributes (or fields) describing an object. Cross them off this list and put them on a list of attributes somewhere. If not, then keep going.

Does this name uniquely identify a set of things with the same attributes? If so, keep going. If not, if it identifies one unique thing, it may still be an object but you should look for commonalities and combine with some other class/object.

Once you have crossed off all nonobjects in this application, you are ready for the next analysis on objects: Determine if it is in the problem space or in the solution space. The **problem space** includes objects that are required to describe the problem but are not required to describe the solution. For instance, you might need to know something about IRS reporting requirements to properly define the length of time you need to keep an accounting file of transactions. But the IRS does not factor into the solution, nor do you keep any information about the IRS in the application. In this example, the IRS would be a problem space object.

The **solution space** includes objects that are required both to describe the problem *and* to develop a solution. In ABC Video, 'customer' is necessary to both the problem definition and to the automated application solution. So, it is in the solution space.

When you are done evaluating all entries in the list, the solution space objects are given a class/object name by which they are known for the life of the application. During this step, we eliminate duplicates of each object. By convention, the name in the list is entered as either *ObjectName* or *=ObjectName*. The format *ObjectName* identifies a unique class/object. The format *=ObjectName* identifies a synonym of a class/object. The *=ObjectName* ensures quality assurance reviewers that you have accounted for all objects and have considered every entry on the list.

Finally, a class/object diagram is begun. A class/object is a collection of like *things* in a class; the objects are the individual *instances* of the *things* in the class. Class/objects are drawn as a rounded vertical rectangle with a shadow rectangle. The class/object is divided into three parts to depict the name, attributes, and processes (see Figure 11-9). The three areas identify public information relating to the class/object. Eventually other details are added for private information during design. Now, let us return to ABC's application to develop the object list.

ABC Video Example Object List

First, we underline the nouns from the paragraph (see Figure 11-10). Objects represent people, organizations, events, applications, or other abstractions from the real world about which we need to keep information. These are all identified by *nouns*. The underlined nouns represent all of the *potential* objects from the paragraph. If the paragraph is complete, this action should result in the identification of all major objects relating to the application.

Next, list the objects *exactly* as they are spelled and ordered in the paragraph. The first-cut object list is shown in Figure 11-11. The dispositions for each object are discussed here.

The first analysis is to eliminate attributes from the list. In the first-cut object list, attributes are crossed out and their respective objects are listed. Attributes change value for each related object instance. To identify an attribute, we ask, Can this *name* take on a value? If the answer is yes, it is an attribute. Attributes are set aside for use in a future step.

Figure 11-11 shows Rental attributes including *AmountDue*, *TotalAmountDue*, *RentalTotal*, *Amount*, and *Change*. Attributes of Videos on Rentals include *RentalPrice*, *ReturnDate*, and *PastDueFees*. Video attributes include *BarCodeId* and *VideoName*. Finally, *PhoneNumber* is an attribute of Customer.

Next, we evaluate remaining nouns to determine if they are objects. The nouns that are clearly objects are the following:

customers
videos
rental (4 times)
tape (4 times)
money
clerk (3 times)
video inventory file
rental form
system

The objects in the above list do not take on values of their own. They are material and distinct, and they are of interest to the application. Therefore, they are objects.

To rent tapes,

1. Customers select one to *n* videos for rental.
2. Customer phone number is entered to retrieve customer data either to create a rental or to retrieve an active rental.
3. Any outstanding video rentals are displayed with the amount due on each tape and a total amount due.
4. Bar code IDs for each tape are entered.
5. Video name and rental price from inventory are displayed.
6. When all tape IDs are entered, the system computes the rental total (= Σ past-due fees + Σ other fees + Σ current video rental fees).
7. Money is collected and the amount is entered into the system.
8. Change is computed (= amount entered – order-total), displayed, and further processed by the clerk as required.
9. When the clerk presses a 'rental-complete' option key (to be defined by the system), this rental is complete and the video inventory file is updated (decrease the count of available copies by one).
10. The rental is stored and printed.
11. The customer signs the rental form, takes the tape, and leaves.

To return a tape,

12. The video bar code ID is entered into the system.
13. The rental is displayed and the tape is marked with the date of return.
14. If past-due amounts are owed, they can be paid at this time, or the clerk can select the 'rental-complete' option which updates the rental with the return date and calculates past-due fees.

For new customers,

15. Enter customer information.
16. Create customer.

For new videos,

17. Enter video information.
18. Create video.

FIGURE 11-10 Underlined Nouns

At this point we are not concerned that there are duplicates on this list, or that we will not keep automated information about all entries on this list. The less obvious, remaining entries we need to evaluate are:

Noun from Paragraph	Disposition
Customers	Object
videos	Object
~~Customer phone number~~	Attribute of Customer, Rental
customer data	Object
rental	Object
active rental	Object
outstanding video rentals	Object
tape	Object
~~total amount due~~	Attribute of Rental
~~Bar code IDs~~	Attribute of Video, VideoOnRental (VOR)
tape	Object
~~Video name~~	Attribute of Video
~~rental price~~	Attribute of Video, VOR
~~tape IDs~~	Attribute of Video, VOR
system	Object
~~rental total~~	Attribute of Rental
Money	Object
~~amount~~	Attribute of Rental
~~system.~~	What we are creating
~~Change~~	Attribute of Rental
clerk	Object
clerk	Object
~~'rental complete' option key~~	Event trigger
rental	Object
video inventory file	Object
rental	Object
customer	Object
rental form	Object
tape	Object
~~video Bar Code ID~~	Attribute of Video, VOR
~~system~~	What we are creating
rental	Object
tape	Object
~~date of return~~	Attribute of Video on Rental
~~past due amounts~~	Attribute of Rental, VOR
~~they~~ (meaning past due amount)	Attribute of Rental
clerk	Object
~~'rental complete' option~~	Event trigger
rental	Object
~~return date~~	Attribute of Video on Rental
~~past due fees.~~	Attribute of Video on Rental
~~customer information~~	All attributes of customer
customer	Object
~~video information~~	All attributes of video
Video	Object

FIGURE 11-11 Initial Object List for ABC Rental Processing

active rental
outstanding video rentals
'rental complete' option key (2 times)
customer information
video information

'Active' is an adjective describing a state of a rental. As soon as we say *describing* we know this is an attribute of some sort. The allowable states most probably are 'active' and 'inactive,' in which case this is the status of a rental, an attribute. We may want to reevaluate what an active/inactive rental is to make sure this is correct. Active, in the sense used here, appears to mean *open rental with rentals*, based on the paragraph. Then inactive would imply *no rentals outstanding*. If this status were to remain in the application, it would be appropriate to change the wording to be more precise to open/closed rental. At some point, the analysis should be reviewed with Vic. So, for the active rental issue, for instance, we might ask Vic the following:

> We have talked about active rentals. Does *active* really mean an open rental? If not, what other kinds of rentals are there? If yes, do we need to keep that status separate or is it implicit? For instance, is an open rental any for which a rental is not returned or is returned with late fees owed?

The next action on active rentals is based on the answers to these questions. Vic decides that *active* does mean *open rentals* and that a specific status is

not required as long as he has access to *open rental* information.

Outstanding video rentals is also an adjectival description of videos on a rental that appears to be a status. Other statuses of videos on rentals that we might identify so far are combinations of:

outstanding/returned
on-time/late
paid/not paid.

We note these for the attribute list and eliminate them from further discussion here.

Last is the *rental complete option key*. This is a noun phrase describing an implementation detail—a key on the keyboard to be pressed to indicate the end of rental processing. It is not an object because it has no attributes, and we do not keep data about it in the application. It is an event trigger that will initiate some processing, but it does not enter into this level of analysis so it is eliminated from the object list.

Last are *customer information* and *video information*. These two items are similar in that they both reference a collection of attributes describing two entities. As such we could either list their attributes (then omit them from the list because they are attributes) or call them objects. We opt for calling them 'collections of attributes' and eliminating them from the object list.

Now we return to the objects we did find to decide if they are in the problem space or the solution space. Problem space objects are required to describe the task domain but not to develop an automated solution. Solution space objects are required to describe both the task domain and the automated solution. Once problem space objects are identified, they drop out of the remaining analysis. We decide which space each object describes (see Figure 11-12).

The last stages are to name each object with a unique name by which it will be known in the system and to eliminate duplicate names for the same object. When we find a duplicate, we indicate the name by an equal sign ('=') appended to the front of the name to signify that the name already appeared once.

During this exercise, we have two options for dealing with repeating information and relationship objects which describe one-to-many relationships. We can define them for later normalization or we can define them as fully as possible now. We opt for more completeness now because it usually means

Object	Space	Justification
Customers	S	Need automated customer information
Video	S	Need automated video information
Rental	S	Need automated rental information
Tape	3 S, 1 P	Three references are tape information to be maintained in the system. One reference is to the tape taken home by customers; this reference is in the problem domain.
Money	P	Real money is outside of the system. We are concerned with the amount which is data entered into the system and related to rental.
Clerk	P	We do not keep statistics or other information on clerks in the system.
Video Inventory File	S	Need automated video information.
Rental Form	P	Just a different media than 'rental' . . . not relevant by itself to the solution.
System	P	This is irrelevant because 'system' is what we are building.

FIGURE 11-12 Object Space Justification

Noun from Paragraph	Solution or Problem Space	Object_Name
Customers	S	Customer
videos	S	VideoInventory
rental	S	Rental
active rental	S	=Rental
outstanding video rentals	S	VideoOnRental
tape	S	=VideoOnRental
tape	S	=VideoOnRental
rental	S	=Rental
video inventory file	S	=VideoInventory
rental	S	=Rental
rental	S	=Rental
tape	S	=VideoOnRental
rental	S	=Rental
customer	S	=Customer
video	S	=VideoInventory

FIGURE 11-13 Object List for ABC Rental Processing

less reworking later. For example, a rental has one or more related videos. We could define both of these as 'rental,' or we could define *Rental* and *VideoOnRental* separately. We opt for the normalized form because it results in a more complete analysis. This results in four class/objects: *Customer*, *Rental*, *VideoOnRental*, and *VideoInventory*.

Figure 11-13 shows the class/objects from this analysis in their final form (for this step). Notice the objects are still in order by their sequence in the paragraph, all have a space designation, and all solution space objects are named.

Finally, we depict class/objects from this list. We switch from the term object to the term *class/object* to acknowledge both the shared attributes and processes *and* the instantiation of them. ABC has four class/objects corresponding to *Customer*, *VideoOnRental*, *Rental*, and *VideoInventory*. The four class/objects are depicted in Figure 11-14 for further elaboration in future steps. Information that we know at this point is also in the diagram.

Identify Processes

Rules for Identifying Processes

The next step is to identify processes. The rules for identifying processes are summarized as follows:

1. Circle all verbs in the summary paragraph.
2. List the circled verbs on a separate sheet of paper, using the *exact* same *sequence* and *spelling* as in the paragraph.
3. Evaluate each verb to make sure it *is* a process. (Common errors are to include status, physical actions, or comments.)
4. Determine whether the process is in the *solution space* or the *problem space*.
5. Name each unique process in the solution space. Ignore those processes in the problem space. Use the convention '=*name*' to identify duplicates of already named processess and show that you know it is a duplicate.
6. Assign objects to verbs if the object is transformed by the process or if the object data is read by the process.
7. Evaluate the object assignments:

   If there is only one object assigned to a process, continue.

   If all objects are read-only, continue.

   For processes with more than one object transformation, evaluate the transformation process:

   If all processes are exactly the same, and all data types acted on are exactly the same, then mark the process for creation of a reusable module.

   If all processes are exactly the same, but all data types are not the same, mark the process for polymorphic module creation.

   If all processes are not exactly the same, redevelop the paragraph to more specifically define the processing.

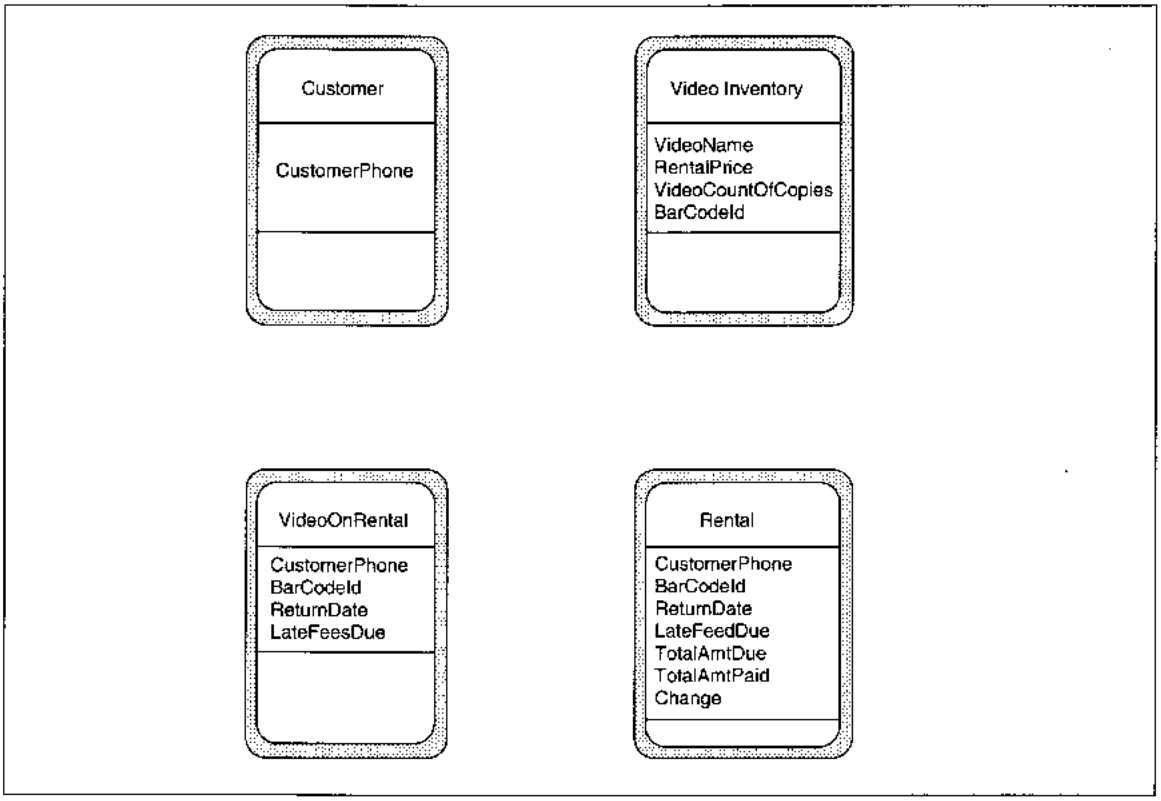


FIGURE 11-14 ABC Class/Objects

Processes are actions described by verbs. We identify the verbs in the summary paragraph, circling them to distinguish them from the nouns. Once the circling is done, make a list on a separate sheet of paper of the verbs. When making the list, keep the verbs in exactly the same sequence and use exactly the same spelling as occurred in the paragraph!

Then, evaluate each verb to make sure it *is* a process. Ask yourself if the verb is a process that the *application* must provide. If yes, keep going; if not, cross the verb off. For instance, if the paragraph said "The clerk enters the customer's phone number into the system," the clerk has been removed as a problem space object. But, the verb *enters* as applied to the customer's phone number *is* required data entry to begin the rental entry process. So, *enters* remains in the system. If we had included the terms *To rent a tape* or *To return a tape* in the list, these are summary descriptions of entire procedures and the verbs *rent* and *return* would be excluded as nonprocesses.

After the first evaluation, review each verb again to determine if it is in the solution space or the problem space. The meanings of solution and problem space are the same as for class/objects. Problem space means the process is required to define the problem but not the automated solution. Solution space processes are required both to define the problem and to define the solution.

Next, review each verb carefully and give it a meaningful name. Try to define meaningful process names that indicate *both* the process and the class/object on which it acts. So, for *enter a customer phone number*, the process name might be *enterCustPhone*.

For any processes that use the same verb descriptor, or that you think are *exactly* the same, mark with an asterisk for further evaluation in the design phase.

Include an asterisk on processes that work on objects with different data types. Name them the same verb appending a unique identifier for each instance. These unique names make recognizing these processes in the next step easier. One possible naming convention[4] is to describe the situation, such as *enterTapeIdRental*, *enterTapeIdReturn*, and *enterTapeIdRenew*. The idea is to assign names that you can live with for the entire life of the object and its processes. In design, if these processes are all defined as the same, we simply truncate the names to *enterTapeId*.

The last step in identifying processes is to assign class/objects to operations. List each object with all processes that *use* or *transform* it. When this identification is done, reevaluate all processes with more than one object assignment.

The three questions you ask in this evaluation are summarized in Figure 11-15. First, ask if only one object is actually transformed by this process. If the answer is yes, go to the next process to be evaluated. If the answer is no, then continue with the evaluation.

Next, for the processes being transformed, does the *exact* same processing occur to each object? That is, are the data types *and* the process steps identical? If the answers to these questions are all yes, no further analysis is required. You have identified a candidate for development as a reusable module. If the answer is no, then you must identify the specific differences with the next set of questions.

Third, are the data types different or identical? Are the processes different or identical? If the data types are different and the process is the same, these process-object combinations are candidates for polymorphic module creation and should be noted with an asterisk. If the processes are different, then you must refine your paragraph to define the specific processes for each object, and redo this part of the analysis from the beginning.

When you have evaluated all of the multiobject processes and resolved any inconsistencies, you are ready to perform the next step. Next, we identify the processes for ABC Video's rental application.

4 A convention is a locally agreed upon way to do some activity.

1. Is only one object actually transformed by this process?

   If yes, this process is complete.

   If no, continue.

2. Does the exact same processing occur for each object? This means the same steps and the same transformations.

   If no, go to step 3.

   If yes, are all object data types the same?

   If yes, this process is complete; create one reusable module for this process.

   If no, mark for polymorphic module creation.

3. Redefine the sentence(s) to identify the specific processing of each object. Then, reevaluate the processes beginning at step 1.

FIGURE 11-15 Multiobject Process Evaluation

ABC Video Example Process List

The steps we follow here are to circle the verbs, evaluate them as processes of interest, define solution and problem space processes, assign class/objects to processes and evaluate those object assignments (refer to the summary list on p. 473).

The first step is to return to the paragraph and circle the verbs. Analyze each verb to ensure that it *is* a process. For instance, if you include in your list the terms 'To rent tapes' and 'To return a tape,' the verbs 'to rent' and 'to return' are omitted from the list because they are identifying the entire process, but are not processes in the system. All verbs in the paragraph are processes. Figure 11-16 shows the verbs circled in the final paragraph.

Next, list verbs and identify their *space*. Remember, problem space identifies processes needed to describe the problem but *not* the solution; solution space processes are needed to describe both the problem and the solution. Figure 11-17 identifies the space of each process listing a reason for exclusion of problem space items. The problem space processes all refer to physical actions which are not tracked by the application. The verb *is complete* is the only nonprocess in the list. *Is complete* refers to

1. Customers select one to *n* videos for rental.
2. Customer phone number is entered to retrieve customer data either to create a rental or to retrieve an active rental.
3. Any outstanding video rentals are displayed with the amount due on each tape and a total amount due.
4. Bar code IDs for each tape are entered.
5. Video name and rental price from inventory are displayed.
6. When all tape IDs are entered the system computes the rental total (= Σ past-due fees + Σ other fees + Σ current video rental fees).
7. Money is collected and the amount is entered into the system.
8. Change is computed (= amount entered − rental-total), displayed, and further processed by the clerk as required.
9. When the clerk presses a 'rental-complete' option key (to be defined by the system), this rental is complete and the video inventory file is updated (decrease the count of available copies by one).
10. The rental is stored and printed.
11. The customer signs the rental form, takes the tape, and leaves.

To return a tape,

12. The video bar code ID is entered into the system.
13. The rental is displayed and the tape is marked with the date of return.
14. If past-due amounts are owed, they can be paid at this time; or, the clerk can select the 'rental-complete' option which updates the rental with the return date and calculates past-due fees.

To add a customer:

15. Enter customer information.
16. Create customer.

To add a new video:

17. Enter video information.
18. Create video inventory.

FIGURE 11-16 Paragraph with Verbs Circled for ABC Rental Processing

a rental status in the procedure which signals different processing. This status is an attribute of the process that we will deal with in the next step.

Next we name solution space processes, eliminating duplicates. Figure 11-18 shows the list of solution processes with names. The duplicate actions are *EnterBarCode*, *DisplayRental*, *DisplayVideoOnRental*, *RetrieveRental*, *RetrieveVideoOnRental*, and *WriteRental*.

Several actions deserve further comment. Sentence 5 for tape rental says, 'Video name and rental price from inventory are displayed.' This sentence implies that name and prices are retrieved from inventory, so the sentence should be modified to reflect this action. Sentence 13 for tape return is similar in saying 'The rental is displayed. . . .' The rental cannot be displayed until it is retrieved. The word 'tape' in the same sentence is ambiguous. Does this refer to the *VideoOnRental* or to *VideoInventory*? In fact, both are affected by this action. The *VideoOnRental* is updated with the return date and the *VideoInventory* is updated to add one to a count of available tapes (the opposite of the action in sentence 9). The sentence should be rewritten to reflect these differences. The new sentence now reads:

> 13. The rental, related video(s) on the rental, and video(s) in inventory are retrieved and displayed. The return date is added to the video(s) on the rental. One is added to the count of available tapes in inventory. Inventory is updated.

Verb from Paragraph	Disposition
select	P—Customer physical action—delete
is entered	P—process (could be more meaningful if called, e.g., read-from-terminal)
to retrieve	S—process
to create	S—process
to retrieve	S—process
are displayed	S—process
are entered	S—process
are displayed	S—process
are entered	status-attribute
computes	S—process
is collected	P—Clerk physical action—delete
is entered	S—process
is computed	S—process
displayed	S—process
processed	P—Clerk physical action—delete
presses	P—Clerk physical action—delete
is complete	status—attribute
is updated	S—process
is stored	S—process

Verb from Paragraph	Disposition
printed	S—process
signs	P—Customer physical action—delete
takes	P—Customer physical action—delete
leaves	P—Customer physical action—delete
is entered	S—process
is displayed	S—process
is marked	S—process
are owed	Rental status—attribute
can be paid	P—optional physical action—delete
can select	P—Clerk physical action—delete
updates	S—process
calculates	S—process
enter	S—process
create	S—process
enter	S—process
create	S—process

FIGURE 11-17 Process Dispositions for ABC Rental Processing

A similar ambiguity is present in sentence 14 which states that 'amounts . . . owed . . . can be paid.' This process, can be paid, refers to sentences 6–8 in the tape rental process. Because these processes are present, we do not need to change the paragraph, but we must reference those sentences so the actions are clear. Sentence 14 now reads:

> 14. If past-due amounts can be paid at this time (repeat sentences 6–8 above); else the past-due fees are calculated and the rental is updated.

This new sentence omits the extraneous information previously present. Both the object list and the process list are reevaluated to reflect these changes. The verbs in sentences 6–8 are also reviewed to ensure identical processing and are added in the proper sequence to the process list. The old verbs are replaced with 'are calculated' and 'is updated.' We review that the nouns from sentences 6–8 and 14 are accounted for in the object list.

The last step is to review the sentences once more, using the object list as reference to assign objects to processes. Figure 11-19 shows the result of this activity. The rule for performing this activity is that any object that is read or acted on by this process is identified.

All processes relating to multiple objects are reanalyzed to determine if they are the same processes. *RetrieveRentalVOR* is identified in the figure as requiring two actions which we discuss here. The processes dealing with *Rental* and *VOR* take information that is separate and process it as if it were integrated. The *Rental* information identifies the customer and the *VOR* describes a video. There is one *Rental* per transaction and one *VOR* per video. The question then becomes one of definition: Is it necessary to maintain this *Rental*, or can it be added to each *VOR* and eliminated?

As in the other methodologies, the *Rental* information and the *Customer* information are essentially

Verb from Paragraph	Space	Process Name	Object Assignment
is entered	S	EnterCustPhone	
to retrieve	S	ReadCust	
to create	S	CreateRental	
to retrieve	S	RetrieveRentalVOR	
are displayed	S	DisplayRentalVOR	
are entered	S	EnterBarCode	
are retrieve	S	RetrieveInventory	
are displayed	S	DisplayInventory	
computes	S	ComputeRentalTotal	
is entered	S	EnterPayAmt	
is computed	S	ComputeChange	
displayed	S	DisplayChange	
is updated	S	UpdateInventory	
is stored	S	WriteRental	
printed	S	PrintRental	

FIGURE 11-18 Named Process List for ABC Video

duplicates. If the company operates on a cash basis and simply needs to know videos outstanding for a customer, then we do *not* need *Rental*. If the company operates on an accrual basis and needs to be able to exactly reconstruct individual transactions, then we need *Rental*. Video rental is a cash basis business; therefore, we do not need *Rental* but we do need to carry its information in *VOR*.

Next, we consider Vic's potential need to differentiate between rentals for a customer or to maintain information beyond the rental's life. Once again, the software engineers return to Vic to find the answer.

Vic: "I have customers sign a copy of a rental and I keep those. I use them to resolve disputes, to find errors, and to provide accounting records. I don't care how you identify rentals because I don't have a need, at the moment, for any analysis. I would like to add trend analysis in the future."

From this discussion, we know there is no business requirement to separate the two objects. A side issue to the decision is whether separation or joining of the objects impacts processing time. For ABC, there is no process time impact. If there were an impact, we would probably opt for the faster solution. We could choose consolidation of *VOR* and *Rental* to simplify processing. In this case, *Rental* would be removed from the list and declared in the object list as =*VOR*. Another option is to leave it as it is. A third option is to think about *Rental* as *Transaction* since attributes, such as *TotalAmountDue*, apply to a specific grouping of videos for a customer at a point in time. There is no 'right' answer to this question, and we do not have enough information to make a final decision although transaction sounds like an idea we will need in design. For now, we will

Verb from Paragraph	Space	Process Name	Object Assignment
is entered	S	EnterBarCode	
is retrieved	S	RetrieveRentalVOR	
is displayed	S	DisplayRental VOR	
is added	S	AddRetDateVOR	
is added	S	Add1toVInv	
is updated	S	UpdateInventory	
can be paid	S	=ComputeRentalTotal =EnterPayAmt =ComputeChange =DisplayChange	
are calculated	S	ComputeLateFees	
is updated	S	WriteRentalVOR	
enter	S	EnterCustomer	
create	S	CreateCustomer	
enter	S	EnterVideoInventory	
create	S	CreateVideoInventory	

FIGURE 11-18 Named Process List for ABC Video (*Continued*)

change the name of *Rental* to *TempTrans* to reflex this thinking and will revisit the need for this class/object again during design. There are no other multiobject processes. The final process list is Figure 11-20.

Define Attributes of Objects

Rules for Defining Object Attributes

An **attribute** is a named field or property that describes a class/object or a process. Each object is a collection of attributes which take on values. A set of specific attribute values describes an object or **instance.** Each object is identified by a **primary key** which is a unique set of values comprised of one or more attributes. A primary key in object-orientation may not actually be used to identify stored objects; physical addresses are most often used.

To define the attributes of an object, we identify all of the information *about* objects. First, attributes that were set aside during object definition are now assigned to a class/object. All items from the original object list that we deleted because they were attributes are now listed with the class/objects they describe.

The original description of the project is rechecked to identify any adjectives or adjectival phrases describing nouns that are now objects in the solution space. In our case, we reread Chapter 2's description of the case and rewrite any attributes identified there that are missing from the object list. These attributes are added to the list.

Next, evaluate the rewritten paragraph to find any data requirements underlying what is stated in the paragraph but not already known. For instance, a status is implied in the statement 'Retrieve all open rentals.' The adjective 'open' implies a status of open/closed. Any *qualified* class/objects should be

Verb from Paragraph	Space	Process Name	Object Assignment—Action Actions are (R)ead, (W)rite, Data Entry (DE), (D)isplay (P)rocess in memory, (PR)int
is entered	S	EnterCustPhone	Customer (DE)
to retrieve	S	ReadCust	Customer
to create	S	CreateRental	Rental (R)
to retrieve	S	RetrieveRentalVOR	Rental (R), VideoOnRental (VOR, R), (NOTE: This requires two *different* actions because the primary keys and read processes are different. We are keeping these together for now for simplicity. All processes marked . . . Rental VOR fit this requirement.)
are displayed	S	DisplayRentalVOR	Rental, VOR (D)
are entered	S	EnterBarCode	VOR (DE)
are retrieved	S	RetrieveInventory	VideoInventory (R)
are displayed	S	DisplayInventory	VideoInventory (D)
computes	S	ComputeRentalTotal	Rental (Process)
is entered	S	EnterPayAmt	Rental (DE)
is computed	S	ComputeChange	Rental (P)
displayed	S	DisplayChange	Rental (D)

FIGURE 11-19 Class/Object Assignments to Processes for ABC Video Processing

evaluated to determine if the qualification is identifying an attribute. When evaluating the paragraph, ask what information is needed to perform, document, or track each action taken. When you identify new information, create attributes for each piece of information.

Next, normalize each set of attributes to third normal form (3NF).[5] For any newly normalized sets of objects, any process-object encapsulations should be reexamined to determine that they encompass both the original object and new objects resulting from the normalization process.

When all attributes are listed with an object, identify a primary key identifier. A primary key provides a unique identification for the object and is composed of one or more attributes. Compare objects to determine if any have identical primary keys. If the answer is yes, consolidate the objects, or change the object with the incorrect primary key. Now, let us walk through attribute identification for ABC.

ABC Video Example Object Attribute List

All items from the original object list that we deleted because they were attributes are first listed with the

5 Recall that normalization includes the following:
1NF—Removal of repeating groups of information
2NF—Removal of partial key dependencies
3NF—Removal of nonkey dependencies.
If you have problems with this activity, refer to Chapter 9 to refresh yourself on this activity.

Verb from Paragraph	Space	Process Name	Object Assignment—Action
is updated	S	UpdateInventory	VideoInventory (P)
is stored	S	WriteRental	Rental, VOR (W)
printed	S	PrintRental	Rental, VOR (PR)
is entered	S	EnterBarCode	VOR (DE)
is retrieved	S	RetrieveRentalVOR	Rental (R), VOR (R)
is displayed	S	DisplayRental VOR	Rental (D), VOR (D)
is added	S	AddRetDateVOR	VOR (P)
is added	S	Add1toVInv	VideoInventory (P)
is updated	S	UpdateInventory	VideoInventory (W)
can be paid	S	=ComputeRentalTotal =EnterPayAmt =ComputeChange =DisplayChange	
are calculated	S	ComputeLateFees	Rental (P), VOR (P)
is updated	S	WriteRentalVOR	Rental (W), VOR (W)
enter	S	EnterCustomer	Customer (DE)
create	S	CreateCustomer	Customer (W)
enter	S	EnterVideoInventory	VideoInventory (DE)
create	S	CreateVideoInventory	VideoInventory (W)

FIGURE 11-19 Class/Object Assignments to Processes for ABC Video Processing (*Continued*)

class/objects they describe. We refer to Figure 11-14 to find those items. A partial list of the attributes from our paragraph is shown in Figure 11-21.

Next, we review the Chapter 2 description of the case and rewrite any attributes identified there that are missing from the object list. These attributes are added to the list as shown in Figure 11-22.

Next, we reconsider our paragraph to find any hidden attributes that are implied by other information such as statuses. We have open and closed rentals, but we might not require a specific attribute for the status. We know a rental is open when it has a *RentalDate* without a *ReturnDate*, or when it has late fees owing. We can check those attributes in lieu of carrying a specific *RentalStatus* attribute. Keeping this attribute requires a judgment call. If junior people are doing the programming, a *RentalStatus* attribute is simpler. If senior people are doing the programming, either method is acceptable. As a matter of choice, we will carry the *RentalStatus* to make sure that future maintenance programmers can also easily understand the processing.

Figure 11-23 shows the initial attribute list for each object. We evaluate each, in turn, to determine its completeness and primary key.

Customer[6] appears complete in its information required to perform rental processing. *VideoOnRental* is considered next. We know we need a

6 Note that if *Rental* had been retained, it would have had the same primary key as Order and would have been eliminated in this step rather than the earlier one.

Verb from Paragraph	Space	Process Name	Object Assignment—Action Actions are (R)ead, (W)rite, Data Entry (DE), (D)isplay (P)rocess in memory, (PR)int
is entered	S	EnterCustPhone	Customer, Data entry (DE)
to retrieve	S	ReadCust	Customer
to create	S	CreateRental	TempTrans (R)
to retrieve	S	RetrieveRentalVOR	TempTrans(R), VideoOnRental (VOR, R)
are displayed	S	DisplayRentalVOR	TempTrans (D)
are entered	S	EnterBarCode	TempTrans (DE)
are retrieved	S	RetrieveInventory	VideoInventory (R)
are displayed	S	DisplayInventory	VideoInventory (D)
computes	S	ComputeTempTransTotal	TempTrans (Process)
is entered	S	EnterPayAmt	TempTrans (DE)
is computed	S	ComputeChange	TempTrans (P)
displayed	S	DisplayChange	TempTrans (D)
is updated	S	UpdateInventory	VideoInventory (P)
is stored	S	WriteVOR	VOR (W)

FIGURE 11-20 ABC Final Process List

Customer Phone to tie rentals to customers and a *Video ID* to tie rentals to inventory. From Chapter 2, we also need rental and return dates. The question is how much fee information we need. Vic supplies the information that he needs to know that regular fees, late fees, or other fees have been paid and the amount of the fee. Therefore, we add those attributes to the list and it also appears to be complete.

The *VideoInventory* is not normalized. While we are normalizing, we can also evaluate the impact of Vic's nebulous desire for promotions on inventory objects. Refer to Figure 11-23's list of the fields and definitions relating to videos in inventory. Repeating information is indented. Primary keys of each part of the information are underlined. The 3NF result of normalization is four relations (see Figure 11-24): *VideoInventory*, *BarCodeVideo*, *VideoPromo*, and *PromoVideo*.

The distinct definition of *VideoPromo* means we can omit it after this analysis because promotions are a future requirement. The separation of *BarCodeVideo* from *VideoInventory* means we need to reevaluate the object and process lists to define related changes. Since *VideoInventory* and *BarCodeVideo* are always accessed together, we can just add *BarCodeVideo* to the lists anytime *VideoInventory* is present. We may want to consolidate the two objects later in the design, for convenience of processing, if we can accommodate repeating information.

The final object attribute list is shown in Figure 11-25 and omits the *VideoPromo* Promo Type objects as discussed above. The attribute list shows the class/objects with their attributes. The process-object figure is corrected to reflect the new *BarCodeVideo* class/object. The objects are all 3NF and appear complete for ABC rental processing.

Verb from Paragraph	Space	Process Name	Object Assignment—Action
printed	S	PrintTempTrans	TempTrans (PR)
is entered	S	EnterBarCode	TempTrans (DE)
is retrieved	S	RetrieveVOR	TempTrans, VOR (R) VideoInventory (R)
is displayed	S	DisplayTempTrans	TempTrans (D)
is added	S	AddRetDateTempTransVOR	TempTrans (P), VOR (P)
is added	S	Add1toVInv	VideoInventory (P)
is updated	S	UpdateInventory	VideoInventory (W)
can be paid	S	=ComputeTempTransTotal =EnterPayAmt =ComputeChange =DisplayChange	
are calculated	S	ComputeLateFees	TempTrans (P), VOR (P)
is updated	S	WriteVOR	TempTrans, VOR (W)
enter	S	EnterCustomer	Customer (DE)
create	S	CreateCustomer	Customer (W)
enter	S	EnterVideoInventory	VideoInventory (DE)
create	S	CreateVideoInventory	VideoInventory (W)

FIGURE 11-20 ABC Final Process List (*Continued*)

Define Attributes of Processes

Rules for Defining Process Attributes

Attributes of processes define formulae, constraints, or status processing performed by or on processes in the application being developed. In particular, **process attributes** define:

- how the process is performed in the system (that is, formulae performed by the process, for example, the formula computing change for a video rental)
- status changes resulting from the process execution (for example, a customer changes from an overdue status to a current status when late fees are paid)
- constraints on the process (that is, prerequisite, postrequisite, time, structure, control, and inference limitations; for example, a prerequisite of video rental is that all late fees must be paid).

The steps to define process attributes are similar to those for object attributes.

1. Assign attributes which were set aside during object or process definition to a class/object.
2. Review the original problem description and any notes from data collection to find attributes.
3. Review the summary paragraph to find implied attributes, such as statuses a process can take.

We use the original description of the problem and the paragraph to determine process attributes.

Object Name	Attribute Name
Customer	CustomerPhone
TempTrans	CustomerPhone BarCodeId ReturnDate LateFeesDue TotalAmtDue TotalAmtPaid Change
VideoOnRental	CustomerPhone BarCodeId ReturnDate LateFeesDue
VideoInventory	VideoName RentalPrice VideoCountOfCopies BarCodeID

FIGURE 11-21 A Partial List of Attributes from the Paragraph

Status attributes identify state changes due to a process's successful completion. The status attributes will, during design, be assigned to a class/object. The purpose of identifying them with processes is that they are more obvious and less likely to get lost.

The constraints are identified to ensure that the procedural code generated during design includes the constraints. The formulae are included as process attributes because they provide some of the logic detail that is also included in the process design.

One inadvertent consequence of process attribute identification can be the definition of artificial constraints on processes. For instance, in the ABC Video rental process, we know that customers must return and pay for prior rentals before taking out new rentals. But consider this situation:

> A customer has several tapes on loan. The customer returns all but one video and wants to rent two others. The customer could pay for all past rentals, the new rentals, and late fees up to the current date for the tape still on loan.
>
> *Or* the customer could pay for all past rentals and the new rentals. The remaining tape, because it is not returned, is left unchanged.

Both of these solutions might be acceptable, but the first places the prerequisite that 'all rental fees be up-to-date' on the customers. This requirement is slightly different than 'all late fees must be paid before new rentals.' The difference is in how late fees are defined; that is, do customers incur late fees when the due date is past the current date or when a video is returned and it has been kept out past the expected return date? In keeping with Vic's edict of the least bureaucracy placed on the customer, the latter definition would be preferred, and he verifies this preference. With this discussion, let us turn to defining the attributes for ABC Video.

Object Name	Attribute Name
Customer	CustomerPhone CustomerLastName CustomerFirstName CustomerAddress CustomerCity CustomerState CustomerZip CustomerCreditCardType CustomerCCNumber CustomerCCExpDate CreditRating CustEnrollDate
TempTrans	CustomerPhone BarCodeId ReturnDate LateFeesDue TotalAmtDue TotalAmtPaid Change
VideoOnRental	CustomerPhone BarCodeId ReturnDate LateFeesDue
VideoInventory	VideoName RentalPrice VideoCountOfCopies BarCodeId TypeVideo Vendor DateReceived

FIGURE 11-22 Additional Attributes from Chapter 2

Object Name	Attribute Name (Primary key is underlined, Repeating information is indented.)
Customer	CustomerPhone
	CustomerLastName
	CustomerFirstName
	CustomerAddress
	CustomerCity
	CustomerState
	CustomerZip
	CustomerCreditCardType
	CustomerCCNumber
	CustomerCCExpDate
	CreditRating
	CustEnrollDate
TempTrans	CustomerPhone
	TotalAmtDue
	TotalAmtPaid
	Change
	BarCodeId
	RentalDate
	FeesPaid
	ReturnDate
	LateFeesDue
	LateFeesPaid
	FeesDue
	FeesPaid
VideoOnRental	CustomerPhone
	BarCodeId
	RentalDate
	FeesPaid
	ReturnDate
	LateFeesDue
	LateFeesPaid
	FeesDue
	FeesPaid
VideoInventory	VideoName
	RentalPrice
	VideoReleaseDate
	VideoCountOfCopies
	TypeVideo
	Vendor
	DateReceived
	PromotionType
	PromoOnDate
	PromoOffDate
	PromoPrice
	BarCodeId
	BarCodeRentalCount
	BarCodeRentalDays

FIGURE 11-23 Initial Object Attribute List for ABC Rental Processing

ABC Video Example Process Attribute List

First, we list all processes down the left margin of a page (see Figure 11-26). Then, we examine each process to determine whether it is constrained in any way. To identify constraints we return to the original description of the problem and the final paragraph to determine processing formulae, constraints, and statuses.

The obvious process attributes are the formulae used to compute rental total and to compute change. Each of these are entered in the table (see Figure 11-26). To ensure proper payment processing, a postrequisite that Change be greater or equal to zero is defined. If this postrequisite is not met, payment processing is performed again.

The first entry in the table for *RetrieveRental-VOR* is a prerequisite that the *Customer* information must be retrieved and a *Rental* able to be developed. If this process is not successful, it is due to a new customer and the *EnterCustomer* process is initiated.

Several status attributes which were set aside during process identification are defined here. Two statuses were identified for knowing when all video data entry is complete and when all transaction processing is complete. Both of these prerequisite statuses are listed with related processes in Figure 11-26. Notice that for the constrained processes, we listed the type of constraint and the details of processing relating to the constraint. Also, notice that many processes have no specific attributes.

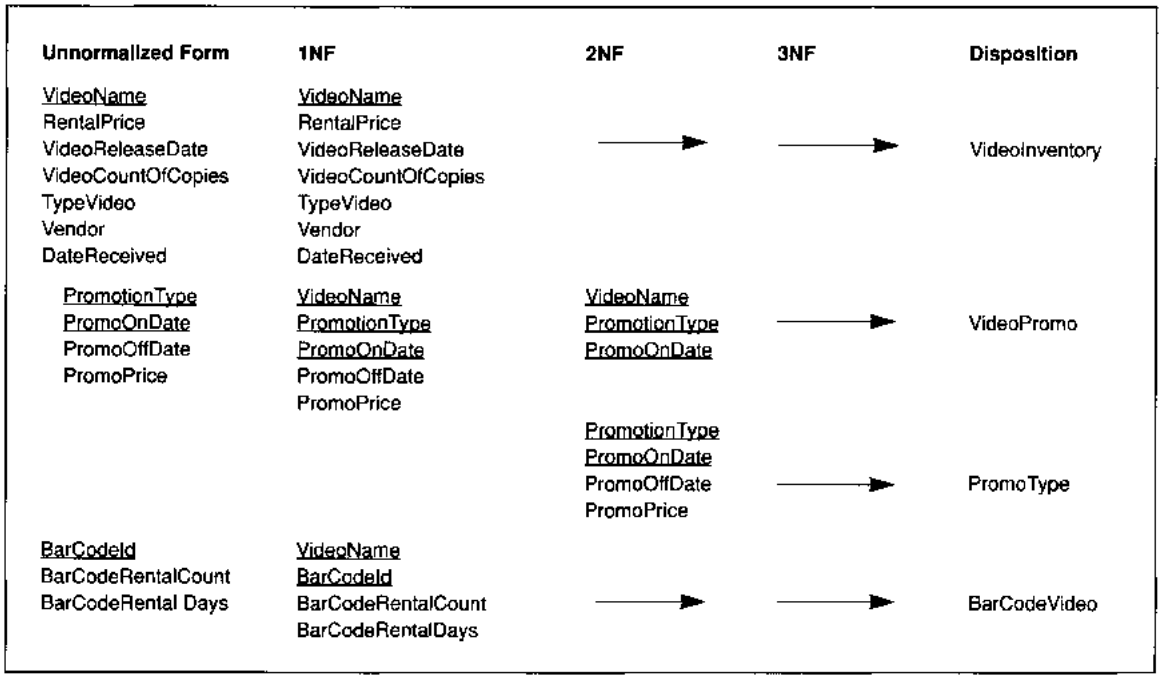


FIGURE 11-24 Normalization of ABC Inventory Information

Perform Class Analysis

Rules for Analyzing Classes

This step is conceptually one of the more difficult steps in object-oriented analysis. It is also crucial to defining the class relationships properly. You have already learned to define entities, relationships, and class hierarchies in Information Engineering, so many of the ideas are not new. What is new is the notion that not just data is inherited: Both data and processes are inherited and considered in this analysis.

The goal is to define classes of class/objects and their relationships. A class defines the attributes and processes that are shared by one or more class/objects. All objects are members of at least one class. When multiple objects share attributes, or share processes, we extract the attributes and processes in common, and create a **superset** class. The important issue is to ensure that the class does, in fact, relate in exactly the same way to all of the multiple class/objects. The class has no objects of its own; it is simply identifying shared data and processes.

Classes are similarly evaluated for commonly shared attributes and processes to create layers of classes. The notation for such a relationship is similar to that of an entity-relationship diagram with directed arrows indicating the direction of the relationship and small numbers indicating the cardinality (i.e., number) of the relationship (see Figure 11-27). Recall that cardinality can be one-to-one (1:1), one-to-many (1:*m*), or many-to-many (*m*:*n*).

To **instantiate** means to define the values of a specific occurrence of an object. (Keep in mind that processes are the same for all instances.) For example, the class/object *Customer* has one instance object for each customer. At the analysis level, an instance is analogous to a tuple in a relation or a record in a file. In an order entry example, illustrated in Figure 11-28, *Customer* class has no specific data; it is an abstract class. The *Cust* class/object instantiates, that is, defines the data values for the customer

class. The *Order* class/object inherits the data and processes in the *Customer* class.

Inheritance relationships identify shared data and processes. The object at the arrow-headed end shares or inherits from the other object. Inheritance relationships identify hierarchical networks of relationships.

Booch [1991] also recommends the design of classes for class/objects whose data or processes are used by another class/object. For instance, an order *uses* information about inventory items. Therefore, another class would be created shared inventory information (see Figure 11-29). This notation is the same as for general classes.

A fifth type of class, a meta-class, can also be defined, but is usually developed during design. The meta-class relationship defines a class whose instances are themselves classes. For instance, customers contain *CustomerName* which defines a subclass 'character string,' which defines a subclass 'character.' Customer is a meta-class representing its character string contents. In general, *all* classes and class/objects from analysis are meta-classes that are elaborated during design.

Coad and Yourdon [1990] recommend looking for classes by evaluating each class/object for special cases and creating *generalization* classes for *specialization* class/objects. For example, cash and credit customers might be specialized class/objects of the general class *customer* (see Figure 11-30). Coad and Yourdon customize their notation for generalization-specialization relationships, although

Object Name	Attribute Name (Primary key is underlined, Repeating information is indented.)
Customer	CustomerPhone
	CustomerLastName
	CustomerFirstName
	CustomerAddress
	CustomerCity
	CustomerState
	CustomerZip
	CustomerCreditCardType
	CustomerCCNumber
	CustomerCCExpDate
	CreditRating
	CustEnrollDate
TempTrans	CustomerPhone
	TotalAmtDue
	TotalAmtPaid
	Change
TempTransDetail	CustomerPhone
	BarCodeId
	RentalDate
	FeesPaid
	ReturnDate
	LateFeesDue
	LateFeesPaid
	FeesDue
	FeesPaid
VideoOnRental	CustomerPhone
	BarCodeId
	RentalDate
	FeesPaid
	ReturnDate
	LateFeesDue
	LateFeesPaid
	FeesDue
	FeesPaid
VideoInventory	VideoName
	RentalPrice
	VideoReleaseDate
	VideoCountOfCopies
	TypeVideo
	Vendor
	DateReceived
BarCodeVideo	VideoName
	BarCodeId
	BarCodeRentalCount
	BarCodeRentalDays

FIGURE 11-25 Final Object Attribute List for ABC Rental Processing

Process	Attribute
EnterCustPhone	
CreateTempTrans	
RetrieveRentalVOR	Prerequisite: CreateTempTrans process must be successful to continue rental processing. If not successful, goto EnterCustomer process.
DisplayTempTransVOR	
EnterBarCode	Status: Bar code entry finished.
RetrieveInventory	
DisplayInventory	Postrequisite: All rentals are entered.
ComputeRentalTotal	Formula = ∑LateFeesDue + ∑VideoPrice by CustomerPhone
EnterPayAmt	
ComputeChange	Formula = TotalAmountDue – Total Amt Pd by CustomerPhone
	Postrequisite: Change must be ≥ zero to successfully complete this process. If change < zero repeat payment process.
DisplayChange	
UpdateInventory	Prerequisite: TotalAmountDue=zero, and processing is complete.
WriteRental	Prerequisite: TotalAmountDue=zero, and processing is complete.
PrintTempTrans	Prerequisite: TotalAmountDue=zero, and processing is complete.
EnterBarCode	Status: Bar code entry finished.
RetrieveRentalVOR	
DisplayTremTransVOR	
AddDateToVOR	
UpdateInventory =ComputeTempTransTotal =EnterPayAmt =ComputeChange =DisplayChange	
Write Rental	
ComputerLateFees	Formula = ∑ LateFees by CustomerPhone
EnterCustomer	
CreateCustomer	
EnterVideoInventory	
CreateVideoInventory	

FIGURE 11-26 Process Attribute List for ABC Rental Processing

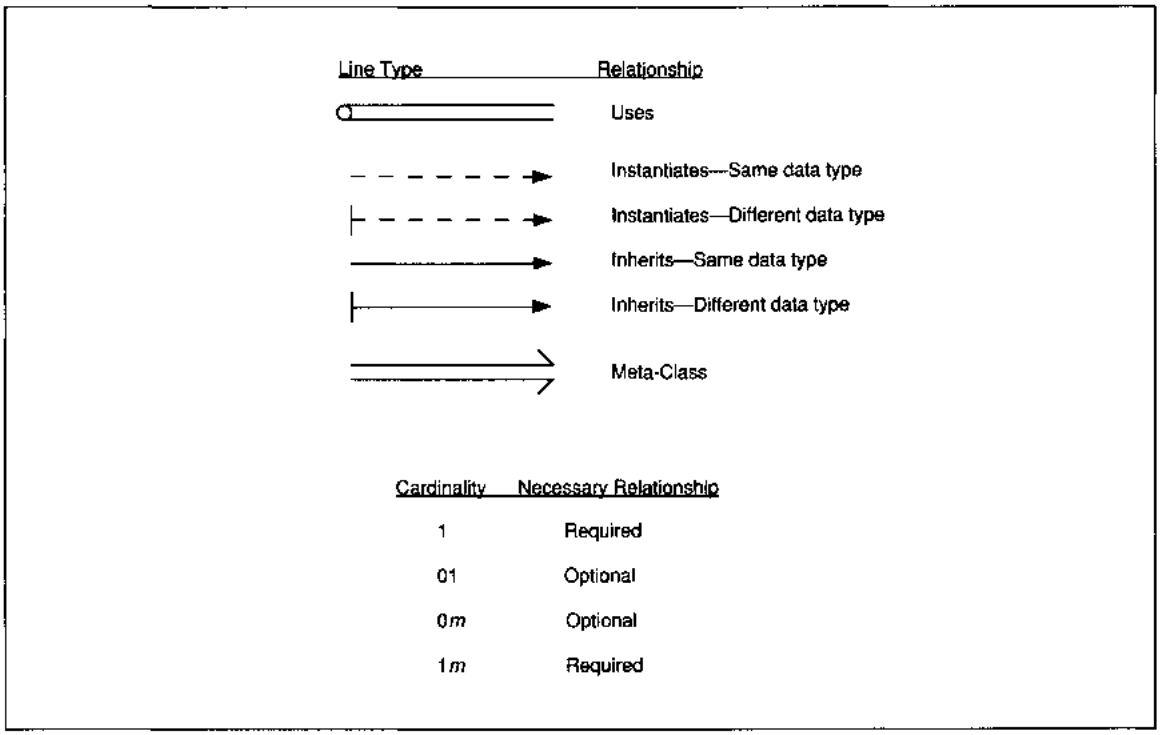


FIGURE 11-27 Relationship Types and Cardinality for Object Class Diagram

it is not necessary to do so unless using their CASE tool. Figure 11-30 shows two alternative *generalization-specialization* notations.

Coad and Yourdon also recommend that classes be created to express *whole-part* relationships. For example, in manufacturing, finished goods are assemblies of other goods; the *whole* class might be for the finished product, while the *part* class/objects define each component (see Figure 11-31). Again, Figure 11-31 shows two notations, a customized version of whole-part as expressed by Coad and Yourdon, and the more general notation used in manual drawings and other CASE tools.

To summarize, we have five types of relationships that we evaluate for specifically. First, we look for shared attributes and processes across class/objects to define inheritance classes. Then we evaluate the class/objects for specialization and for component part relationships. Next, class/objects which *use* the attributes or processes of another class/object are identified to create a class for the common class/object items. Finally, we define meta-classes as abstract classes whose instances are themselves classes.

To create less cluttered diagrams, elevate the highest independent class or class/object on each diagram to define *subjects*. A **subject** is the most abstract class represented in an application. The purpose of subjects is to provide a summary identifier that represents the cluster of subordinate relationships which inherit from the class (see Figure 11-32).

Finally, we reevaluate and, as necessary, redefine both process-object assignments, class, and class/object definitions again. We reevaluate to ensure that all definitions accurately reflect the application requirements, and are 'clean,' that is, all processes relate to all data with which they should be encapsulated.

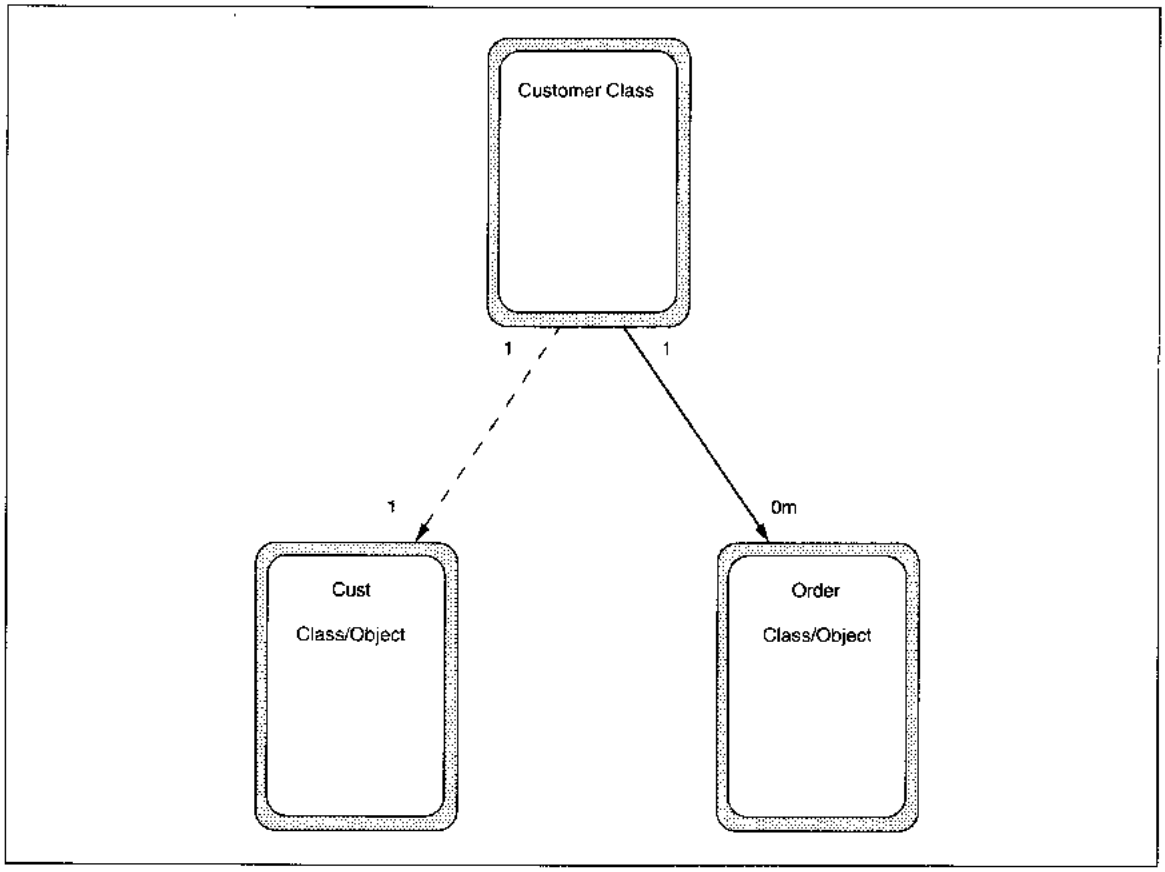


FIGURE 11-28 Order Entry Example of Customer Class

ABC Video Example Class Analysis

The class diagram for ABC rental processing is fairly simple (see Figure 11-33). First we draw the object classes: *Customer*, *VideoOnRental*, *Video-Inventory*, *BarCodeVideo*, and *TempTrans*.

Next, we evaluate the relationships between them. Referring back to the attribute list, we see that *VideoOnRental* (*VOR*) contains information from *Customer*, *BarCodedVideo*, and *VideoInventory*. The question is, Is this an inheritance relationship or a using relationship? In other words, are the data and processes also shared by *VOR* or does it simply use the data? The answer is found in the process descriptions. For all three class/objects, if the object does not exist while rental processing is going on, the rental class/object is supposed to be able to add new customers and new videos. Therefore, the processes for adding and reading the information from all three class/objects are shared and should be inherited. If *VOR* simply used the data, the using relationship would have been more appropriate. *BarCodeVideo*, *Video-Inventory*, and *Customer* are drawn as classes because they will not actually store data. They manage the shared processes.

Next, we consider the relationship of *VideoOn-Rental (VOR)* to *TempTrans*. There is considerable overlap since VOR gets all new objects from *Temp-Trans*, and *TempTrans* gets all information about open rentals from *VOR*. In this example, neither can

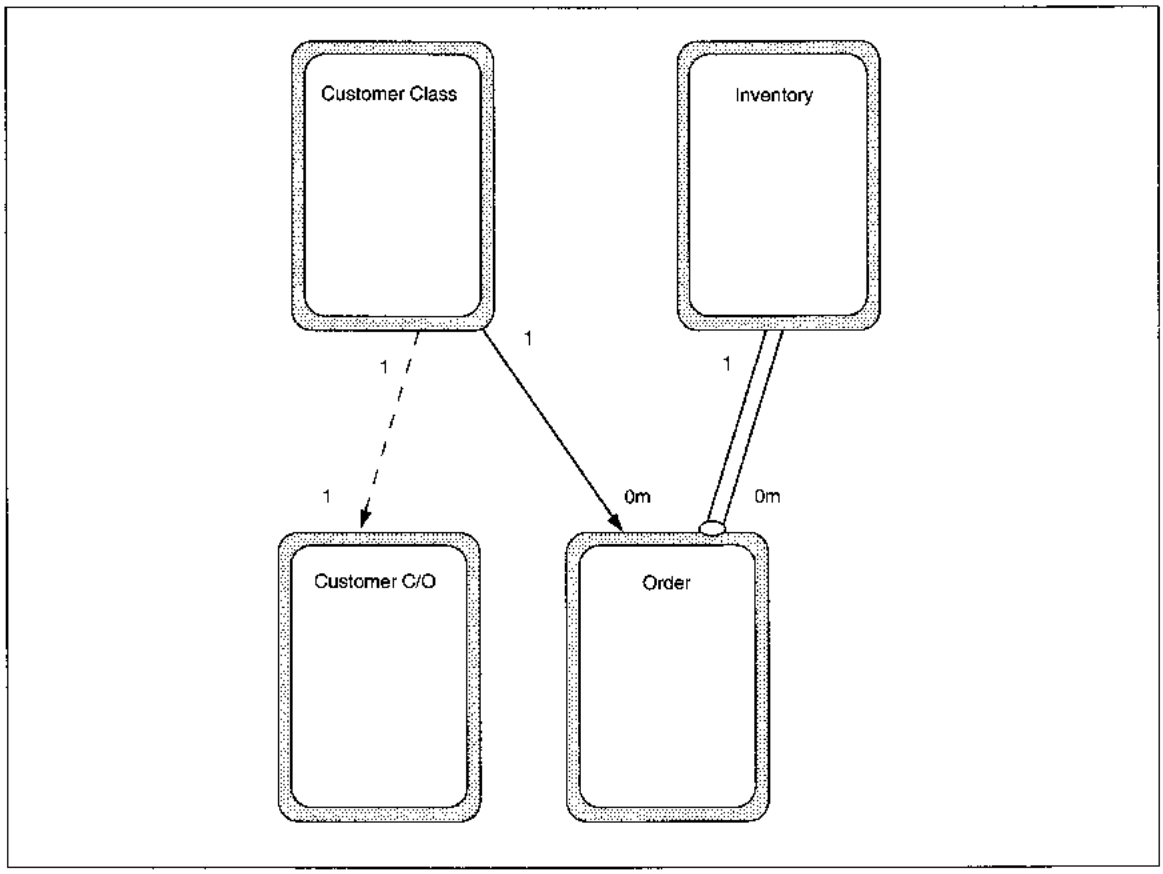


FIGURE 11-29 Example of Using Class

inherit the processes of the other. Since they both use each other's data, they have reciprocal using relationships which are expressed in the diagram (see Figure 11-33).

Then, we create new class/objects for attributes and processes not shared or inherited by *VOR* (see Figure 11-34).

Next, we consider the relationship between *BarCodedVideo* and *VideoInventory*. *VideoInventory* defines the characteristics of a group of inventory items. For instance, there will be one object with the value *Terminator 2* in the Video Name, but there might be many *BarCodedVideo* objects which refer to that name. That is, there are many copies of the movie, each with its own bar code. Therefore, the characteristics of *VideoInventory* appear to be inherited by *BarCodedVideo*.

Next, we ask if the processes of *VideoInventory* also apply to *BarCodeVideo*. For instance, when we add a *BarCodeVideo*, do we need to know or do processing for *VideoInventory*? One attribute of *VideoInventory*, a count of the number of videos in stock, is created and updated every time *VOR* is created or used during rental processing. Therefore, a class for *VideoInventory* that includes the attribute(s) and processes that are shared is required. Now we have two classes dealing with *VideoInventory* and one class/object that will contain the data. The diagram reflecting these final data and processing requirements is shown in Figure 11-34.

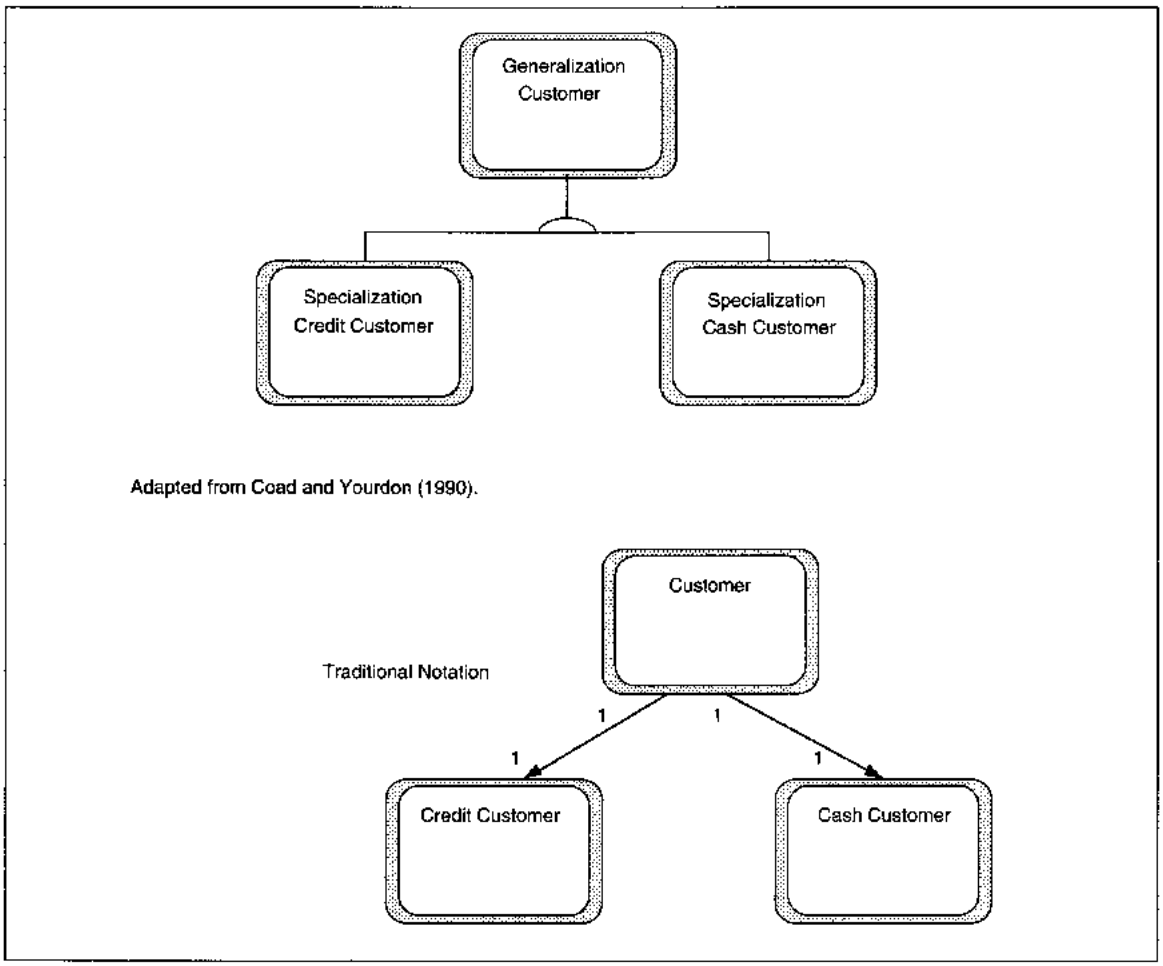


FIGURE 11-30 Example of Generalization-Specialization Classes

Draw State-Transition Diagram

Rules for Drawing a State-Transition Diagram

A **state-transition diagram** defines allowable changes for data objects. Specifically, for each change of data content for an object, we identify the initial state, the event that causes the change, the process by which the change occurs, and the resulting state. A **state** is a set of values an object can have while a **transition** is an event causing a change to the set of values.

There are two subtly different types of state-transition diagrams known as the *Mealy model* and the *Moore model*. The **Mealy model** defines all state changes and associates each with an action; it is used in this text. The **Moore model** defines all actions and associates each with a state. Theoretically, both models lead to the same definitions, they take different perspectives. For novices, the Mealy model is simpler because it is easier to identify and verify state changes than it is to identify and verify that all actions are present.

The icons used in drawing a state transition diagram are shown in Figure 11-35 as a circle and directed line. The rules for developing a state-transition diagram are as follows:

1. Draw one diagram for each object/class and each class.
2. Identify the possible states the class/object can take.
3. Draw circles on a diagram labeling each with a state.
4. Connect the states to show transition from one state to another. Use directed arrow lines to show the direction of state change (i.e., from . . . to . . .). Each state should lead to one or a small number of other states.
5. Label the transition lines to identify the events that initiate the change. Write the event names above the lines.
6. Label the lines with the processes that manage the event. Write the process names under the lines.

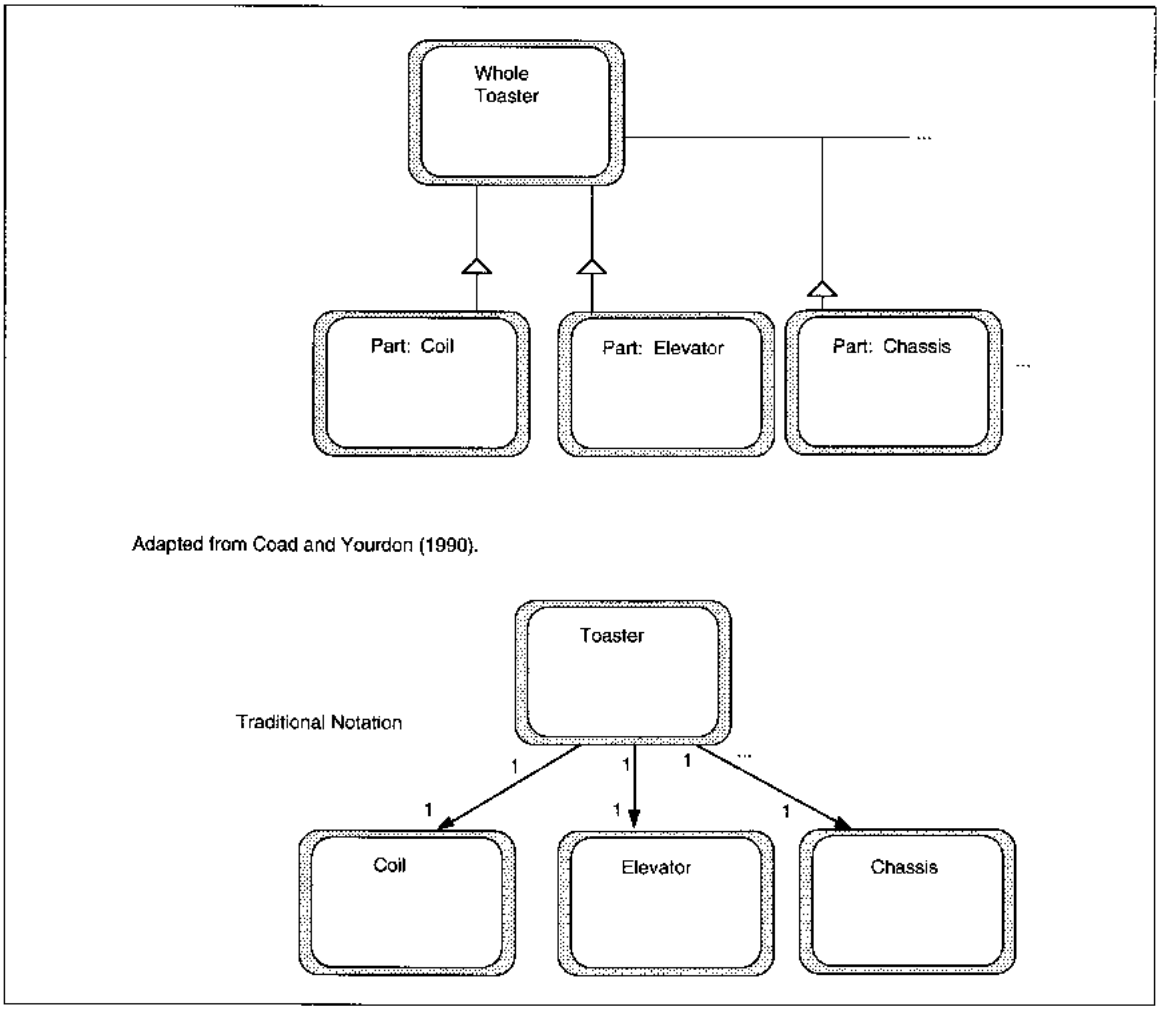


FIGURE 11-31 Example of Whole-Part Class

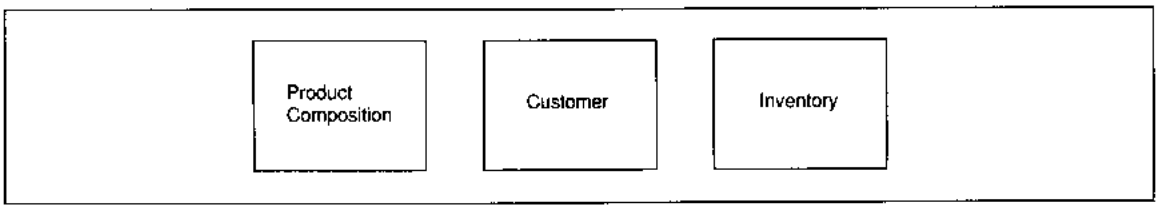


FIGURE 11-32 Example of Subject Diagram

7. Examine the diagram. If there are any recursive state changes, reanalyze that part of the diagram in more detail to remove the recursion or to specifically label the state and its processes as recursive.
8. Walk through the diagram with other team members until it is complete and accurate.

The circle identifies the states of the object. Directed lines signify transitions and lead to the resulting state. The *event* causing the transition is written on top of the directed line. The *process* that changes the state is written under the directed line. The names of states should be unique, but the names of events and actions need not be unique if they, in fact, relate to more than one state. Events can spawn more than one process. Conversely, object states can require more than one event to be changed. If many events are required to initiate a state change, they are shown with separate lines leading to the state. If *any* of several events can initiate a state change, the lines converge into one line entering the state. Each *class* and *class/object* in an application has a state-transition diagram developed for it.

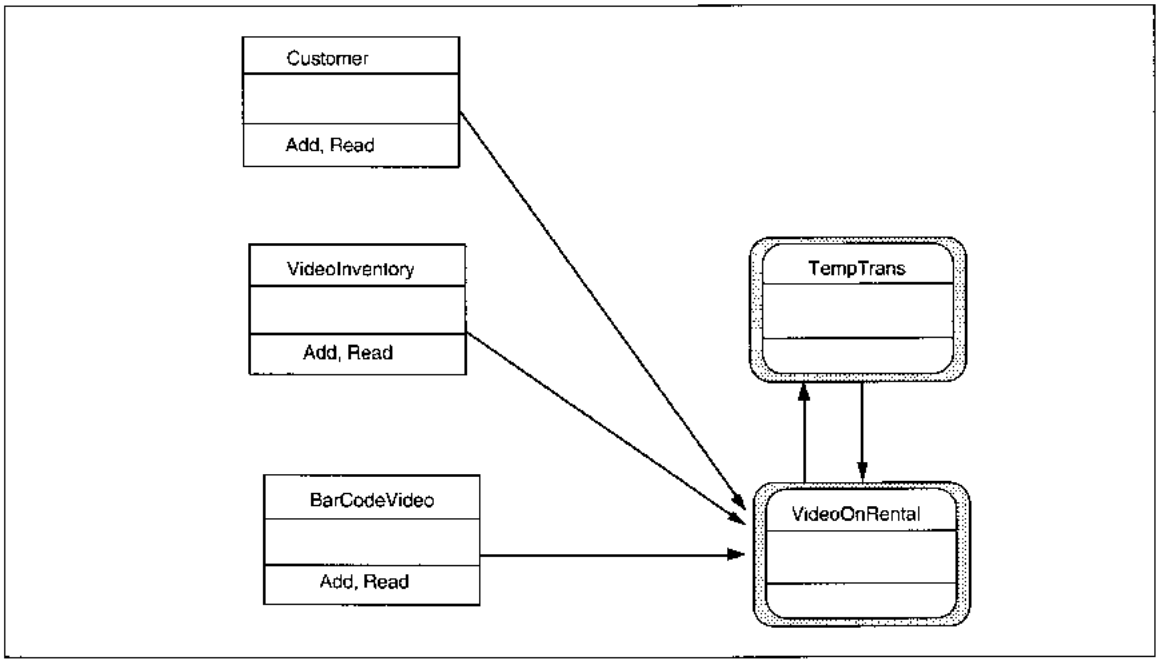


FIGURE 11-33 Class Diagram for VideoOnRental

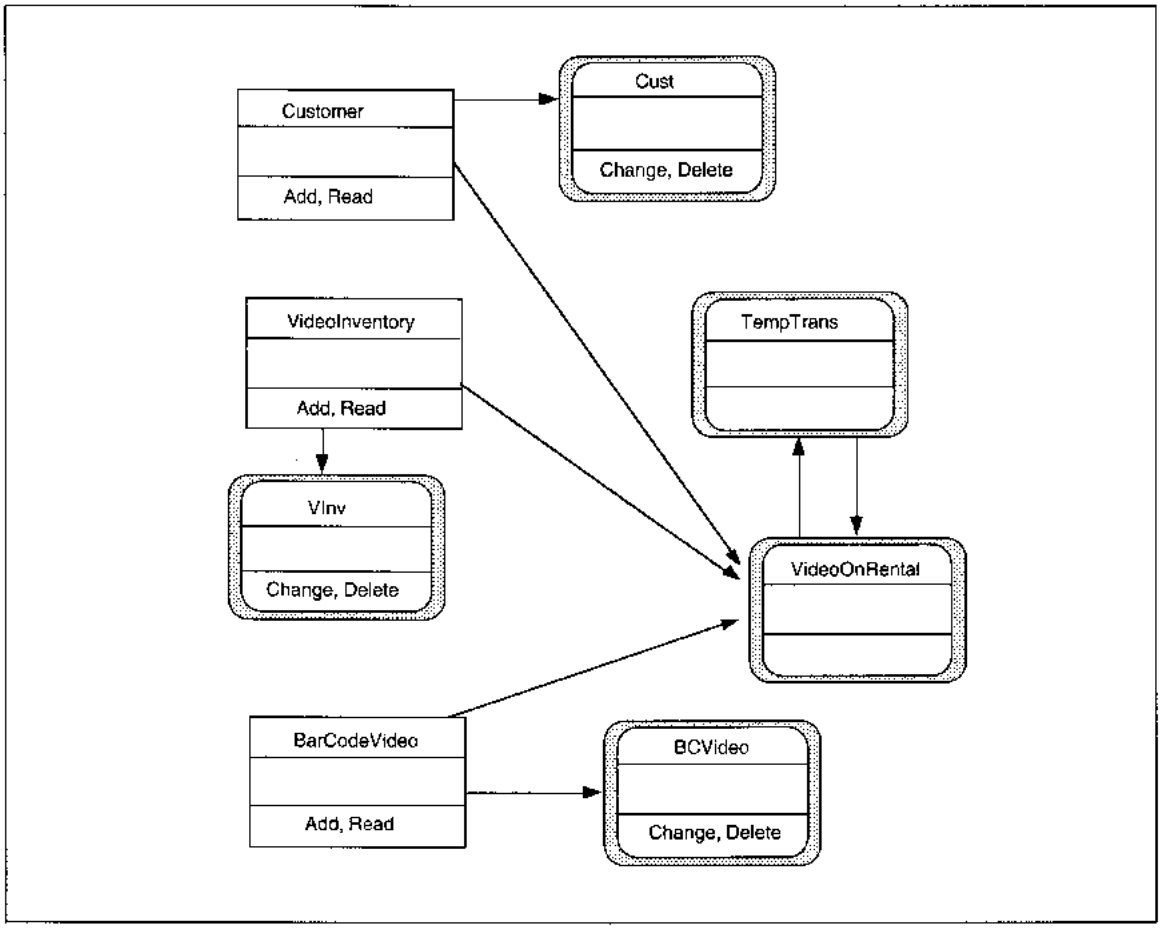


FIGURE 11-34 Class Diagram for ABC

State-transition diagrams are optional representations in object orientation. They are useful for diagramming the behavior of systems with

- multiple message types
- complex processes
- synchronization requirements.

Different diagrams, such as *fence diagrams*,[7] are often substituted for state transition diagrams when there are less than 20 states.

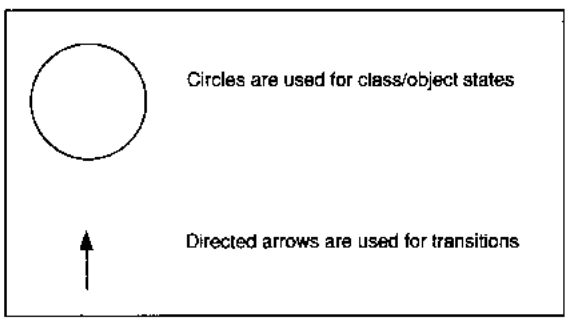


FIGURE 11-35 Icons Used in State Transition Diagrams

7 See Martin and McClure, 1985, for a further discussion of different substitute representations.

ABC Video Example of State-Transition Diagram

The steps to developing a state-transition diagram are to draw circles for each state that an object can take. Then connect the circles with lines showing which states lead to which next-states. Label the lines with the event triggering the change on top and the associated process from the application under the line. Rental *VOR* objects are the most complex in the ABC Video rental processing task, so they are discussed here. Development of state transition diagrams for the other objects is left as a student exercise.

In its most simple form, a rental is either open or closed (that is, no rental). So, the first iteration of the state transition diagram will begin with those two states. The high level diagram is in Figure 11-36. For each path between these two states, we ask ourselves the question, What causes the change? First, what causes the change from no rental to an open rental? Open rentals are created when a customer *requests a rental*; this is the event for the line from no rental to open rental. The process accompanying this event is to *create an open rental.*

Second, what causes the change from open rental to no rental? Return of the video(s) and payment of late fees can cause an open rental to be closed. There are two events in this statement, so now we ask ourselves about the events' timing. Are all returns and payments performed at the same time? If not, what is different about them? From the description of the rental process, we know that returns can be made without any payment taking place. So, we separate these events.

Consider returns first. When a video return takes place, what process is performed? The answer is that we update the rental with the return date. The rental does not change from open to closed when a return is performed, however; so, we draw a recursive line from *open rental* to *open rental* and mark it with the event and process. *This recursive line identifies a need for another level of detail on rental states because each state should have its own circle for clarity of expression.*

Finally, we evaluate the other event, payment of rental fees. This event causes a rental to become closed. The directed line connects open rental to no/closed rental, the event is *pay late fees*, and the process is *close rental.*

We already know we have to create another level of detail to this diagram to be more specific about return date processing, but we also want to evaluate this diagram to see what else is required. Does this diagram account for *all* rental states? The answer is no. It does not account for situations when late fees are owed (in other words, if there is already an open rental), and it does not account for updates for fees paid. So, we redraw the diagram to include these states.

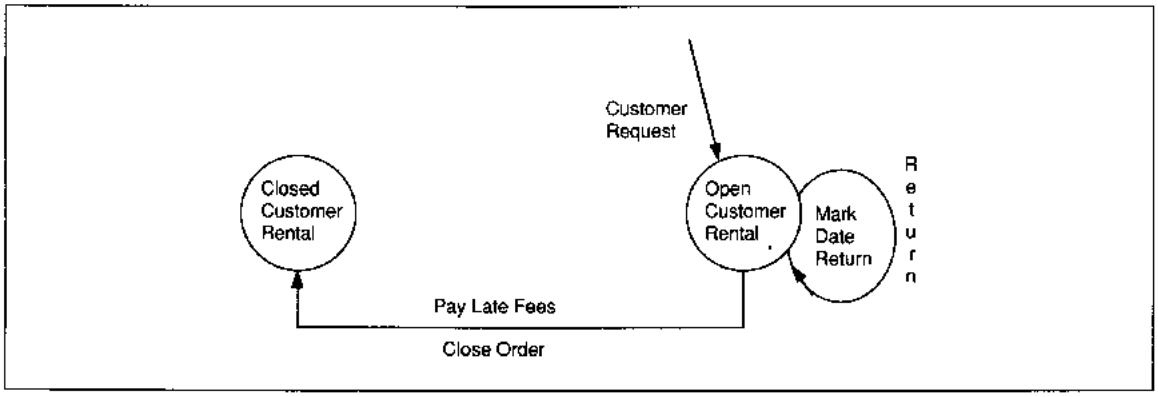


FIGURE 11-36 High-Level State-Transition Diagram for ABC Rental Processing

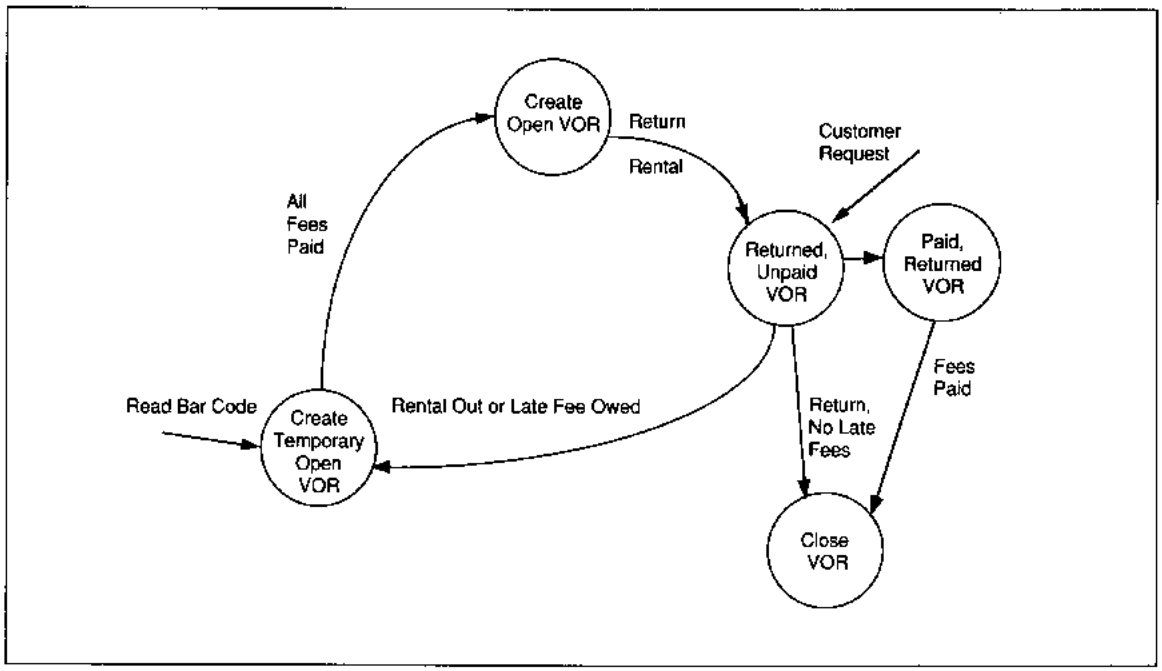


FIGURE 11-37 State-Transition Diagram for ABC Rental Processing

In the revised diagram (see Figure 11-37), we continue the thought process we used to draw the first diagram while accounting for the details we omitted from the first diagram. Now, we try to identify the states through which a rental proceeds from its opening to its closing. The states are:

- open
- temporary, new rental in memory, until fees paid
- unpaid, returned *VOR* may have late fees
- paid, returned VOR
- closed rental with return date and all fees paid

Next we draw the lines showing how each of these states comes to exist. Notice that a customer request triggers a search of open rental and will result in either the temporary rental status or the add-on rental status, depending on whether or not a rental for this client exists. The remaining events are *return-Rental* and all fees paid.

AUTOMATED SUPPORT TOOLS FOR OBJECT-ORIENTED ANALYSIS

Object orientation is less than five years old in its use in business. Yet the number and variety of support tools and environments available attests to its growing popularity and legitimacy. The tools presented here represent both partial and complete support for one or another method of developing object views of the world (see Table 11-1). Many tools include code generation capabilities which automatically generate C++ or other object-oriented code objects from the logical definitions provided in object analysis and design.

SUMMARY

Object orientation is a methodology that alternates evaluation between objects and processes to develop

TABLE 11-1 Automated Support Tools for Object-Oriented Analysis

Product	Company	Technique
DSEE, HP/Softbench	Apollo/Hewlett-Packard	Integrated CASE Product Supporting OO Analysis
Excelerator	Index Tech. Cambridge, MA	State-Transition Diagram Matrix Graph (RTS)
Object View	KnowledgeWare Atlanta, GA	Application Prototyping Software Using 4GL or SQL Code
Object Vision	Borland International	Visual Application Development System
OOA Tool	Object International, Inc. Austin, TX	Coad's Tool Supporting Object Analysis Using Coas & Yourdon Graphics
ProMod	Promod, Inc. Lake Forest, CA	Control Flow Diagram State-Transition Diagram Module Networks Function Networks
Software Backplane Cohesion	Atherton Technology/ Digital Equipment Corporation Maynard, MA	Integrated CASE Product Supporting OO Analysis
SW Thru Pictures	Interactive Dev. Env. San Francisco, CA	Control Flow State-Transition Diagram
Teamwork	CADRE Tech. Inc. Providence, RI	DFD Control Flow State-Transition Diagram Process Activation Table
Telon	Pansophic Systems, Inc. Lisle, IL	State-Transition Diagram
Visible Analyst	Visible Systems Corp. Newton, MA	State-Transition Diagram
vs Designer	Visual Software Inc Santa Clara, CA	Booch Diagram Visual RD Diagram Ward-Mellor Diagram

a complete view of an application. Objects are entities to be automated. They are encapsulated with processes which operate on them or which read them.

Encapsulated class/objects may be identified for creation of reusable, normal, or polymorphic modules. Reusable modules perform the same action on the same data type class/objects, but are called by more than one class/object. Normal modules perform one action on data from one object. Polymorphic modules perform one action on data from many objects of differing data types. Object-process capsules are evaluated to determine their interrelationships. The interrelationships usually describe a

hierarchic network of relationships for which the lower-level capsules inherit both the data and processes of the higher capsules. Encapsulated class/objects with multiple relationships have multiple inheritance from related higher capsules.

The declarative steps performed to develop an object analysis include identification of class/objects, identification of processes, class and hierarchy definition, definition of attributes of operations, definition of interobject messages, and class/object state-transition definition. The procedural evaluations within each step consist of questions to be answered and actions to be taken based on the answers to the questions.

REFERENCES

Berrard, E. V., *An Object Oriented Design Handbook for Ada Software*. Frederick, MD: EVB Software Engineering, Inc., 1985.

Booch, G., *Object Oriented Design with Applications*. Redwood City, CA: Benjamin/Cummings Publishing Company, Inc., 1991.

Coad, P., and E. Yourdon, *Object-Oriented Analysis*. Englewood Cliffs, NJ: Prentice-Hall, 1990.

Coad, P., and E. Yourdon, *Object-Oriented Design*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

Graham, Ian, *Object-Oriented Methods*. Reading, MA: Addison-Wesley, 1991.

Taylor, David, *Object Orientation and Information Systems: Planning and Implementation*. NY: John Wiley & Sons, 1992.

KEY TERMS

abstract data type (ADT)
attribute
class
class hierarchy
class/object
client object
encapsulation
generalization class
inheritance
instance
instantiate
Mealy model
message
meta-class
Moore model
multiple inheritance
object
object-oriented analysis
part class
polymorphism
primary key
private part (of a class/object)
problem space
process attribute
public part (of a class/object)
solution space
specialization class
state
state-transition diagram
subject class
superset class
supplier object
transition
user object
whole class

EXERCISES

1. Complete the state-transition diagrams of the ABC Video rental processing application. Walk through the diagrams in class and discuss the difficulties and alternatives you found in developing the state transition diagrams.
2. Perform an object-oriented analysis on the Eagle Rock Golf League in the appendix. Develop all lists, tables, diagrams, and pictures required to document the requirements of the problem.
3. Split the class into three teams. Have each team develop a second-level analysis of ABC *CustomerOnVideo* maintenance using object-oriented analysis. Compare the resulting views of the application.
4. Debate this assertion: Object orientation is more likely than process or data methodologies to lead to well-defined modules which automatically deal with problem complexity by hiding information, being single-purpose, and having minimal coupling.

STUDY QUESTIONS

1. Define the following terms:
   class
   class/object
   encapsulation
   inheritance
   meta-class
   multiple inheritance
   object
2. Describe the sequence of events during analysis.
3. Compare the differences between the major forms of documentation in structured analysis and object-oriented analysis.
4. Compare the differences between the major forms of documentation in information engineering and object-oriented analysis.
5. Why is the summary paragraph in object-oriented analysis so important?

6. Compare and contrast the definitions of objects, processes, and encapsulated objects.
7. List the documents and graphics created in object-oriented analysis and describe how they are related to each other.
8. What are the decisions you must make in object-oriented definition of object hierarchies? Why are they important?
9. What rules in object-oriented analysis simplify quality control and review?
10. How do you determine that the allocation of objects to processes is correct? What are the questions asked, and why are they important?
11. What is polymorphism? What is its importance in object orientation?
12. What is the purpose of a state-transition diagram?
13. Describe the development of a state-transition diagram.
14. What is the relationship between a state-transition diagram and objects, processes, or encapsulated objects?
15. What is the purpose of a graphical class diagram?

EXTRA-CREDIT QUESTIONS

1. What are the rules for identifying objects? Can you think of others that might be useful?
2. The steps that use nouns and verbs to identify objects and processes, respectively, have been criticized as too simplistic. Can you think of a different approach to identifying objects and processes, perhaps borrowing from another methodology, that improves on the process?

CHAPTER 12

OBJECT-ORIENTED DESIGN

INTRODUCTION

Object-oriented analysis defines classes and class/objects, processes, and the assignment of objects to processes, resulting in encapsulated objects. In object-oriented design (OOD), we continue this analysis of the problem domain to assign the encapsulated objects to one of the four subdomains, elaborate component definitions to include service processes, design module interactions, and define the required messages and their type.

CONCEPTUAL FOUNDATIONS

Encapsulation and inheritance are the basis for OOD, just as they were for object-oriented analysis. In addition, the scope of the thinking process is expanded. The design approach is holistic, designing people, hardware, software, and data as the four components of object-oriented systems.[1] As Figure 12-1 shows, the four components all relate to each other and can all communicate with each other. This design methodology makes a valuable contribution to methodological thinking by integrating the components, many of which are frequently ignored. As multiprocessor computing, such as in client/server, increases, this type of classification will be required of *any* methodology. Object method developers have led the thinking about multiprocessor applications because of the closeness between object-oriented applications and operating systems from which many concepts are borrowed.

OOD explicitly uses an iterative approach to detailed design that Booch calls **"round-trip gestalt"** [Booch, 1991, p. 188], meaning the incremental development of whole applications. Each prototype

1 This concept of four components is from Coad, Peter & Edward Yourdon, *Object-Oriented Design*. Englewood Cliffs, NJ: Prentice-Hall, 1992.

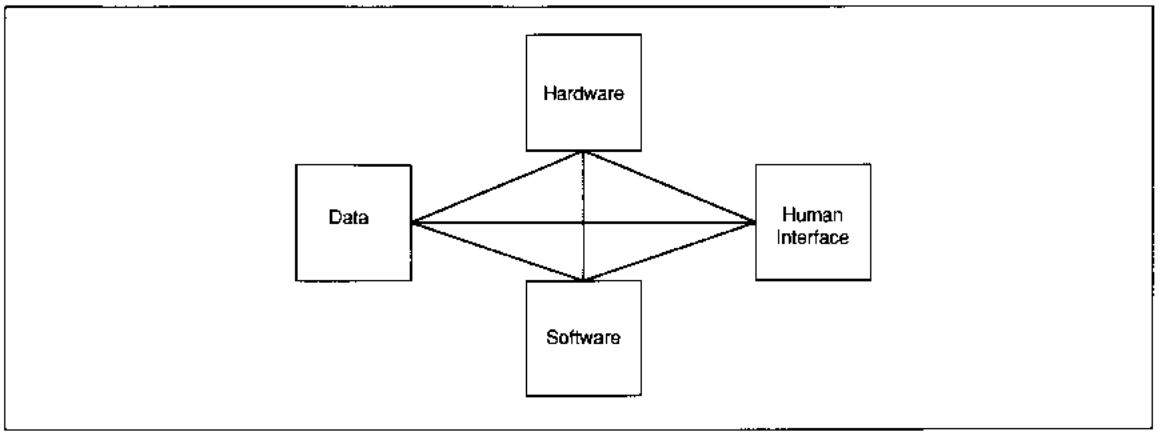


FIGURE 12-1 Object-Oriented Subdomains

is the entire application as currently defined. As the prototype is examined, further details of operation are explicated for incorporation in the next iteration of the prototype. Following the format of previous chapters, we first define terms used in the OOD process, then move on to developing guidelines for each step and an example of the step and thinking processes for ABC Video's rental application.

DEFINITION OF OBJECT-ORIENTED DESIGN TERMS

The seven steps to performing an object-oriented design are:

1. Allocate objects to four subdomains, including human, hardware, software, or data.
2. Develop time-event diagrams for each set of cooperating processes and their objects.
3. Determine service objects to be used.
4. Develop Booch diagrams.
5. Define message communications.
6. Develop process diagram.
7. Develop package (i.e., module) specifications and prototype the application.

In this section, we define the terms used in these steps, again integrating and extending the work of Booch with that of Coad & Yourdon. Keep in mind that while the terms are fairly well-defined, the manner and order of implementing the steps is not. The documentation created by these steps is summarized in Table 12-1.

In the first step, problem domain objects are assigned to one of the human, hardware, software, or data subdomains. The **human subdomain** defines human-computer interaction in the form of dialogues, inputs, outputs, and screen formats. A **dialogue** is interactive communication that takes place between the user and the application, usually via a terminal, to accomplish some work. A dialogue defines actions of users and actions of the application and hardware. Inputs (i.e., data entry), outputs (e.g., reports), and screens are the three modes of communication used for a dialogue. The task being performed is usually a transaction relating to a business event (e.g., sale of goods), but could also relate to application-generated events, such as sensor readings in process control or a data request in a query application. A screen design alone is a static definition of field formats while the dialogue is a series of interactions that takes place via a dialogue.

The **hardware subdomain** defines object assignment to physical processors or firmware.[2] The hard-

2 Firmware refers to software that is permanently on a programmable chip and that processes significantly faster than memory-resident software program code.

TABLE 12-1 Object-Oriented Design Documentation

Tables	
Process Assignment to Object Table	Contains all solution space objects and, for each, the processes that act on the object
Subdomain Allocation Table	List of processes and subdomain assignments
Message Table	Contains, for each process, the calling object, the called object, the input message contents, the output message contents, and the object to which control is returned
Diagrams	
Subdomain Allocation Diagram	Optional graphical depiction of process-subdomain assignments
Time-Ordered Event Diagram	Depicts required sequencing of processes
Booch Package Diagram	Depicts objects and message flow for the entire application. Lower-level Booch diagrams, one per processor, are created to show objects and processes with message flow.
Process Diagram	Shows hardware configuration and process assignment to processors

ware interface is significant as we develop applications using more firmware, mainframes augmented by local intelligent devices, and distributed processing. To support these types of processing, allocation of tasks to hardware must explicitly be part of the methodology.

The **software subdomain** defines service control and problem-domain objects. **Service control objects**, also known as **utility objects**, manage application operations. Depending on the complexity of the application, synchronizing, scheduling, or multitasking services to control object/process work might be required. **Problem-domain objects** are the class/objects and objects (hereafter, both are referred to as *objects*) defined during analysis and describing the application functions.

The last subdomain relates to **data**, which are the actual instances of the objects in the solution set. During the data design, data are normalized and redesigned to accommodate operational efficiencies. Depending on the 'purity' of the object implementation, the physical data storage may or may not implement encapsulated data and processes in the database. The most common variation of data storage is a template definition that uses physical address pointers to reference the physical data store for data and processes. The template is analogous to the *File* and Working-Storage *Sections* of a COBOL program, but includes a process template as well as a data template.

The second step for all processes, regardless of their subdomain assignment, is to develop time-event diagrams. **Time events** are the business, system, or application occurrences that cause processes to be activated. Time-event diagrams show sequences, concurrency, and nesting of processes across objects. The **time-event diagram**, then, shows the relationships between processes that are triggered by related events or have constraints on processing time. Process relationships are either sequential or concurrent, determining the types of service objects required in the application. Processes that are not concurrent are sequential and related only by data or parameters passed between the processes. **Concurrent processes** operate at the same time and can be dependent or independent. Dependent concurrent processes require synchronization of some sort.

Above, we defined service control objects as managers of application operations. The third OOD step is to determine which service objects are needed to control the application. There are three broad categories of service objects: synchronizing, scheduling, and multitasking.

Synchronizing is the coordination of simultaneous events. **Synchronizing objects** provide a rendezvous for two or more processes to come together after concurrent operation (see Figure 12-2).

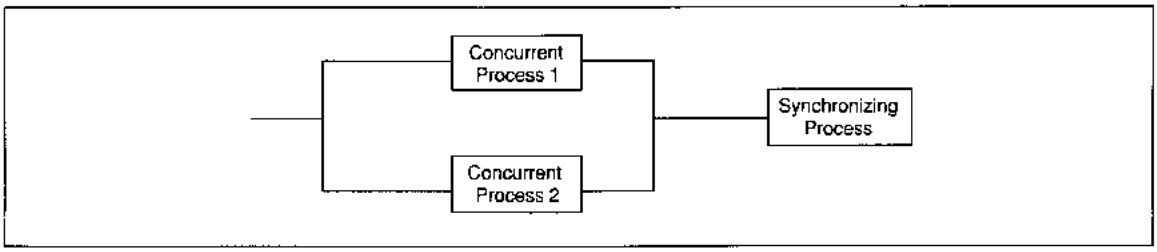


FIGURE 12-2 Diagram of Synchronizing Object Functions

Scheduling is the process of assigning execution times to a list of processes. **Scheduling objects** can be for sequential, concurrent-asynchronous (i.e., independent), or concurrent-synchronous (i.e., dependent) processes. In the terminology of COBOL, scheduling objects are analogous to a mainline routine (see Figure 12-3), but the scheduler performs many functions beyond those of a COBOL mainline.

Multitasking is the simultaneous execution of sets of processes (see Figure 12-4). Each set of concurrent processes is called a **thread of control**. These threads are initiated by the scheduling objects and controlled by multitasking objects. **Multitasking objects** track and control the execution of multiple threads of control and can be in both the problem domain and the service control domain. These three types of service control objects provide the structure within which problem domain objects execute.

Service object definition is based on time-event diagram analysis. If all objects are sequential and used one at a time, then only scheduling objects are required. If concurrent processing takes place, synchronizing and scheduling objects are used. If many users are supported concurrently, multitasking objects are added to the other types.

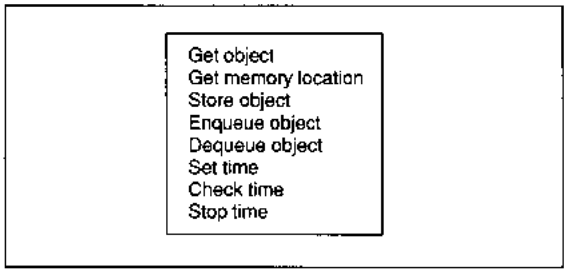


FIGURE 12-3 Scheduling Object Functions

After service objects are identified, the next step is to begin to develop a Booch diagram. A **Booch diagram** depicts all objects and their processes in the application, including both service and problem domain objects. First, a draft diagram is created. Then, several message passing schemes are evaluated. After a message passing scheme is identified, message contents are defined.

The basic graphical forms used are rectangles and ovals (see Figure 12-5). Vertical rectangles signify a *whole* package. A **package** in OOD is a set of modules relating to an object that might be modularized for execution. Service packages are single purpose and do not usually have subparts that are visible to the rest of the application. Service objects have no visible data, that is, no oval identifying a data part to the object. Problem-domain packages have data identifiers for objects and processes. The object in the oval and the process names are each in their own horizontal rectangle (see Figure 12-5). In Figure 12-5, the lines connecting modules show allowable paths for messages.

Next, messages are defined. A message is the only legal means of communications between encapsulated objects. Messages are clear in their intention, but not clear in their implementation which is completely determined by the language. For instance, at the moment, Ada does not implement message communication. In this text, a **message** is the unit of communication between two objects. Messages contain an addressee (that is, the object providing the process, also called a service object), and some identification of the requested process.

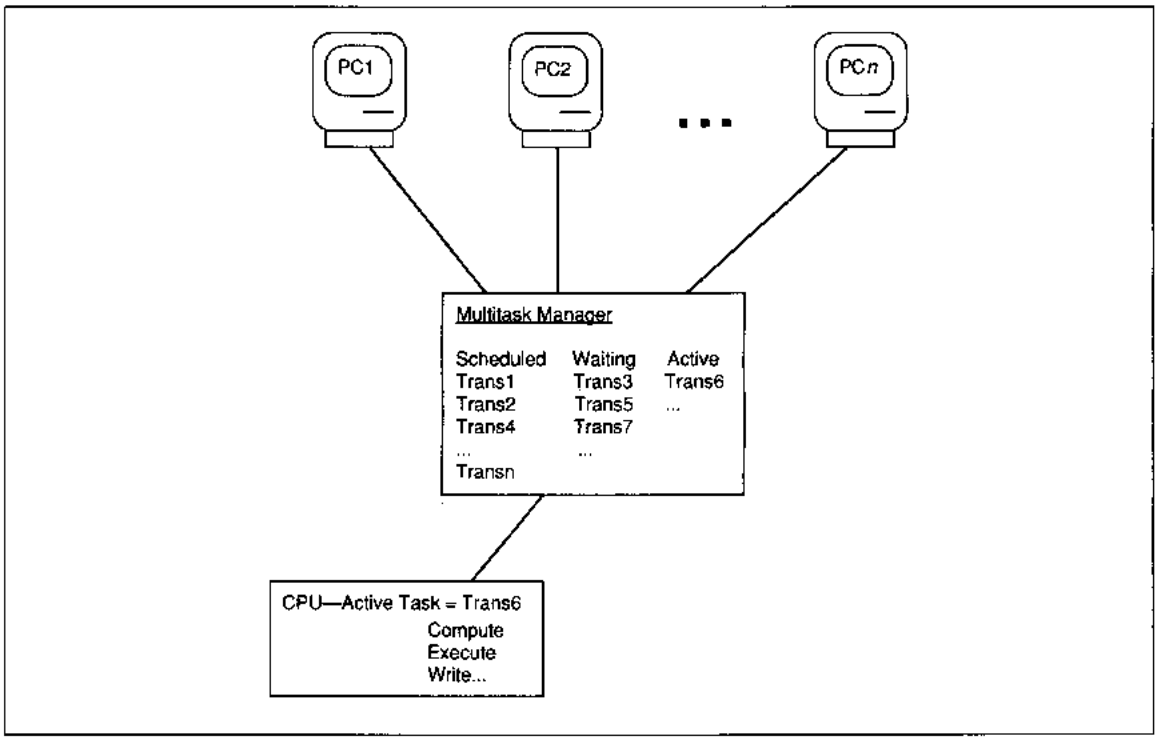


FIGURE 12-4 Multitasking Management of Multiple Threads

Messages may be unary, binary, or keyword (see Figure 12-6). **Unary messages** contain only an addressee and service identifier. **Binary messages** contain addressee, service identifier, and two arguments (that is, variable object names or addresses upon which the service is performed). **Keyword messages** contain addressee, service identifier, and one or more keywords, each with an argument

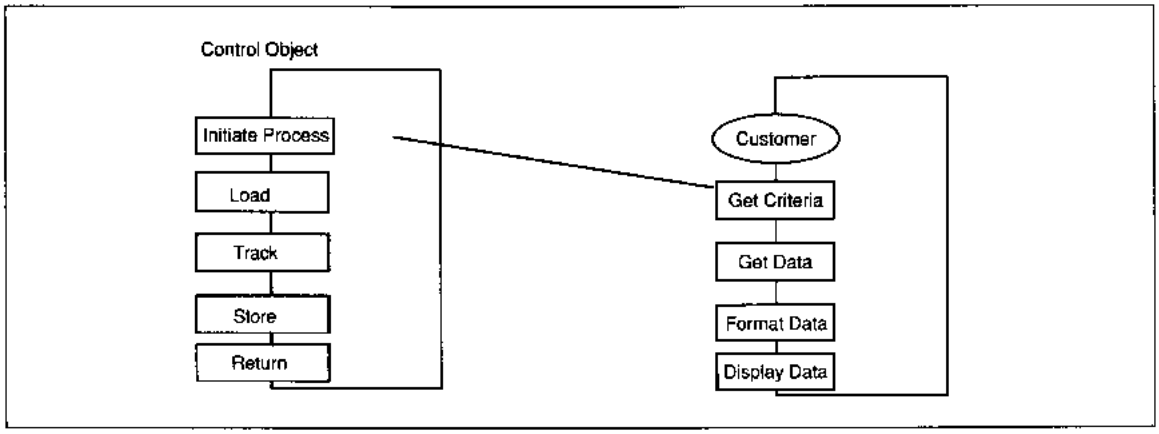


FIGURE 12-5 Sample Booch Diagram—Simple Inquiry Process

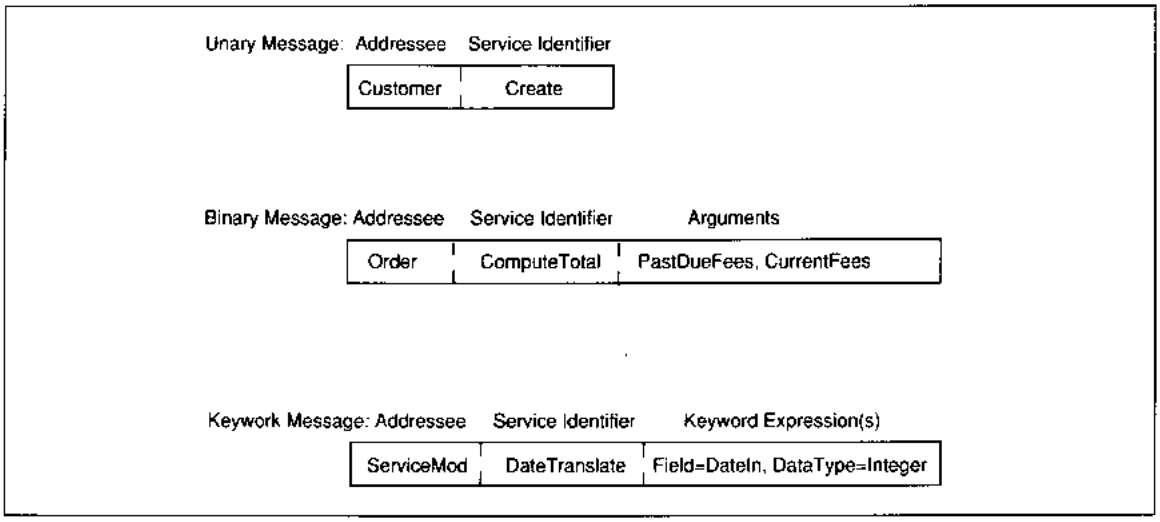


FIGURE 12-6 Example of Message Types

to show optional process selection. Message definitions probably will expand as languages capable of expressing and processing object-oriented designs develop.

The next step is to develop a **process diagram** that defines the hardware environment and shows process assignments to hardware. The first activity is to draw a hardware configuration showing processors (shadowed boxes in Figure 12-7) and devices (plain boxes in Figure 12-8). Lines connecting processors identify allowable message paths. At this summary level, multiple messages may travel each path.

When the process diagram is complete, the Booch diagram is divided and redrawn for each processor in the configuration. These subdiagrams show the extent of replication in the application and may identify new service object needs to control interprocessor communications. The message list is reexamined to ensure that all interprocessor messages are accommodated and complete. For multiprocessor applications, the timing of processes is reverified to ensure correct definition.

Using the information from the problem domain analysis and the OOD diagrams describing object interrelationships and timing, the next step is to develop package specifications and prototype the application. These are not the last steps in the design, only the last steps in an iteration of the design process which may have several iterations. As a result of prototype development, other service objects might be recognized as needed. Iterating requires review of all design steps and redoing analysis as required to support development of a complete application prototype for each iteration.

Package specifications define the public interface for both data and processes for each object, and define the private implementations and language to be used. The **public interface** is that part of the data and process definitions visible to all objects in the application. The **private interface** describes the physical data structure and actual functions (i.e., data manipulations, calculations, or control processes) to be coded for the application. Multiple implementations of the same function that operate on different data types might be required. The function that has one name but multiple implementations is called polymorphic. **Polymorphism**, is the ability to have the same process, using one public name, take different forms when associated with different objects.

One item in a package specification is a definition of the language to be used. Process timing (i.e., sequential or concurrent) and a need for polymorphism determine the type of implementation language required. Some languages are more con-

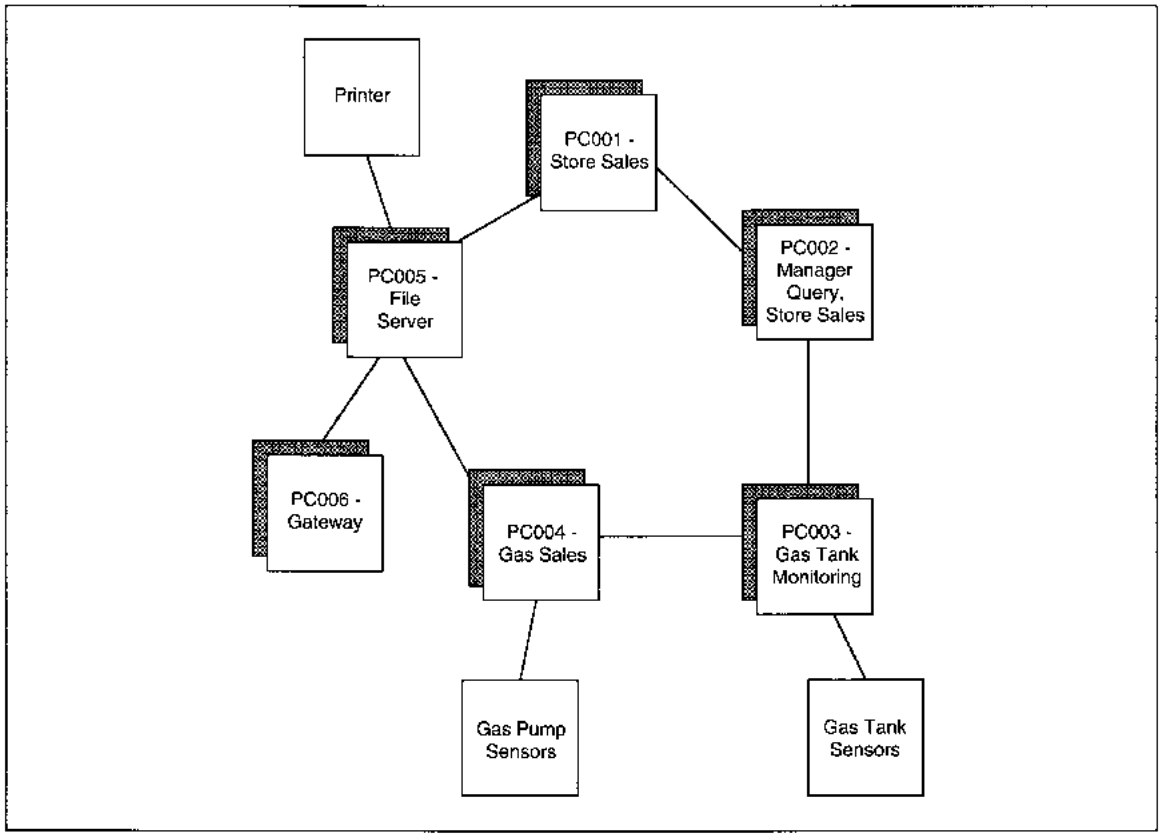


FIGURE 12-7 Process Diagram Example of Convenience Store/Gas Station Network

straining than others. To understand these language differences, binding and client/server relationships should be understood. **Binding** is the process of integrating the code of communicating objects. Binding of objects to operations may be **static** (fixed at compile time), **pseudo-dynamic** (parameter driven and decided at the beginning of a session), or **dynamic** (decided for each object while the system is executing, that is, at run time).

A major difference between object orientation and other methodologies is the shifting of responsibility for defining the data type of legal processes from server (or called) objects to client (or calling) objects. A **server object** is one that performs a requested process. A **client object** is one that requests a process from a supplier. For instance, you might need to translate a date from month-day-year format to year-month-day format. As a client object, you request the translation of the supplier object and pass it the date to be translated. If the language supports dynamic binding, you also pass the data type of the date (for example, binary string or packed matrix). This shift, to client/server logic, plus the notions of inheritance and dynamic binding, support the use of polymorphism.

Let's return to the idea of binding and work our way through these ideas and how they work together. In most business applications, we think of processes as always operating on the same type of data. For example, items on an order have an order quantity (for example, 2), quantity type (for example, each or dozen), and price (for example, $1.20) that is expected to match the quantity type. To compute the line item total, we multiply quantity times price for a given quantity type. But what if the type quantity is not known beforehand and the formula must change

based on the type? Then, we have three choices. First, we could write many routines that are all resident in the compiled code as static binding requires. This is the most common COBOL solution.

Second, we could write many routines that use information passed to the computation procedure to identify which routine to use for the session (for instance, only dozens will be processed in one session). This is called pseudo-dynamic binding (e.g., in Ada at the moment).

Third, we can write many routines and pass the quantity type to the computing object in the request message to dynamically bind and select the routine it needs to compute the total (as in Assembler, C++, or Smalltalk). Dynamic binding is done *on-the-fly* at run time. When the computation is complete, the quantity type code no longer is kept in the computer's memory.

Binding time is a function of the language used and the application's requirements. If the application is batch, single-thread, and sequential, there is no *need* for any but static binding. If the application is anything else (multithread, concurrent, real-time), dynamic binding is desirable, but many languages only support pseudo-binding. Then, the application requirements, in the form of business needs for response time or process time, should drive the language selection decision.

We no longer assume that a called object can do only one thing in only one way; instead, a called object can do only one thing but it can do it in many ways. This ability to do one thing many ways *is* polymorphism. Polymorphic processes take different forms when associated with different objects, but a process *always* takes the same form with a given object. Client-object message requests contain both the process and the form of the process. The polymorphic process then loads its correct process code to service the request via the *dynamic binding mechanism* of the implementation language. An example of pseudocode for polymorphic pairwise item comparison is shown in Figure 12-8.

This discussion summarized the major terms, diagrams, and procedural steps in object-oriented design. Next, we discuss the steps of OOD in detail, including allocation of objects to the subdomains, developing time-event diagrams, determining service objects, developing Booch diagrams, developing process diagrams, and developing module specifications. Prototyping is beyond the scope of this text.

```
Pairwise Compare—                Pairwise Compare—
Two Numbers                      Two Matrices

If A = B                         Set sub = 1
  return-code = 1                Set return-code = 0.
else                             Perform compare
  return-code = 0.                 varying sub by 1
Return return-code.                until sub = 1st-entry.
                                 Return return-code.

                                 Compare.
                                 If A(sub) not = B(sub)
                                   return-code =1.
                                 Compare-exit. Exit.
```

FIGURE 12-8 An Example of Polymorphic Descriptions for a Comparison Process

OBJECT-ORIENTED DESIGN ACTIVITIES

In ABC's rental application, we are using off-the-shelf software in an off-the-shelf hardware environment. In the environment, the operating system, network, database, and form of human interface are all given. Because of our choices—PCs, MS-DOS, Novell Netware, and a SQL DBMS—the application does not easily lend itself to object-oriented design that assumes none of the services and functions provided in our target environment. Because of the differences, we will discuss ABC at two levels: one for SQL DBMS which becomes unobject-like, and one for a Unix/C++ environment that stays object-like. First, we follow ABC through the process of design keeping in mind that the off-the-shelf software will be used. Think of this design as **object-based**, that is, based on object thinking, but decidedly *not* object-oriented in implementation. Object-based design is what is practiced by most novice object-designers, and is what most CASE tools being retrofitted for object orientation will be. In the chapter appendix, we present a second design for a Unix/

C++ environment that is completely object-oriented. Without both discussions, the view of object orientation that you would get is not complete, and some of the discussions would be inaccurately stated for object-oriented design.

Allocate Objects to Four Subdomains

Heuristics for Allocating Objects to Human, Hardware, Software, and Data Subdomains

The first step is to allocate the problem domain processes to one of the subdomains: hardware, software, data, and human interface. Each process and the data it requires from its object[3] are examined to determine whether they are best implemented as part of the human interface, hardware, software, or data subdomains. There is no particular order to the allocation process. It is recommended to allocate the software domain last, because it is the default for all processes not allocated elsewhere. Since these implementation alternatives are usually not broken apart by other methodologies, and since hardware is usually completely ignored, the consideration of these subdomains and explicit allocation of objects to them provides useful detail that is explicitly documented for maintenance. Also, since hardware options are becoming more numerous and common (e.g., automated teller machines have local intelligence and some of the application code for deposit and withdrawal processing), this mechanism accommodates hardware and firmware in design decisions.

We will discuss data first, because current guidelines demonstrate some of the shortcomings of current OOD writing. Booch suggests that standard database activities should be assumed to be under the control of the data domain, including create, retrieve, update, and delete processes (i.e., CRUD). All other data manipulations or computations are allocated 'somewhere else.' Coad & Yourdon, and most authors published after 1992, assume the use of a DBMS and usually an object-oriented one that includes the properties of persistence, inheritance, and abstract object-oriented data definition. Some authors assume use of an SQL-compatible database with an equally unobject-like language, recommending that the data functions should be separated from the application which will maintain its object-like properties for all non-data operations.

Keep in mind that this is an inexact process that is highly dependent on the implementation language and the implementation environment. For example, if we were using Smalltalk, in which *everything* is an object, separation of data access and manipulation is usually more efficient than keeping the functions all together. Conversely, if an OODBMS, such as Gemstone, were used, the DBMS object performs the physical CRUD actions and the application objects usually control the logical CRUD functions that are grouped by object. The key idea is that judgment on allocation of functions is required and needs to be done with knowledge of the entire implementation environment.

If the application *needs* to use a nonOODBMS, then evaluating whether data integrity, security, and access controls can be adequately maintained by *not* using the DBMS language is required. If the application can both perform the functions faster, and provide for integrity and so forth, then there should be a real analysis of where the functions should be. The application requirements for execution and response time may force use of a programming language when constraints are tight, and default to use of the DBMS language when there are no constraints.

Table 12-2 summarizes this discussion, showing that allocation of physical and logical read, write, and delete actions and the control over security, integrity, and access be tied to constraints and the type of database environment used. If no DBMS is used, the alternatives are either to allocate DBMS functions to each object, or to design data control objects that perform DBMS functions, or to design a polymorphic reusable object that performs all DBMS functions.

We said before that DBMSs illustrate the problem of all authors in object-oriented design. For the most part, OO authors do not work in commercial business and do not build commercial applications; they

3 Superset objects, class/objects, and objects are all assumed in the use of the term *object*.

TABLE 12-2 Heuristics for Data Allocation Processes

Type Database	OO	OO	Non-OO	Non-OO	None
Functional or response time constraints	Y	N	Y	N	—
Allocate CRUD to DBMS	Phys.	All	*Phys.	Phys. *Log.	—
Allocate CRUD to Object or generic	Log.	—	*Phys. Log.	*Log.	All
Allocate security, integrity checking, access control to DBMS	—	All	—	*All	—
Allocate security, integrity checking, access control to Object or create generic objects	*All	—	All	*All	All

Legend:

Phys.	=	Physical functions (read, write)
Log.	=	Logical functions (edit)
*	=	Requires analysis and judgment
All	=	Physical and logical
Y	=	Yes
N	=	No

work in defense-related businesses and build real-time, embedded applications which function as part of some larger system. For instance, defense applications might include building a guidance system for a missile, a monitoring system for airplane radar, or a reporting system on the Hubble microscope. These applications all have *no persistent data*; rather, they work on sensor data and pass on the information they filter for processing or feedback by other systems.

The problem with applying embedded-system thinking to persistent object problems is that there is little overlap in designing for temporary and persistent data. Persistent data and, in particular, DBMS-stored persistent data, have entirely different thinking processes that the computer-scientist authors of most object-oriented methods do not recognize. Because of this lacking recognition, these heuristics on object allocation are more crude than those of, say, process methods which have been tried for the last 20 years.

A similar problem occurs in the hardware domain. Object-oriented authors most often are designing state-of-the-art hardware as part of their application design including customized operating systems and software. Most business applications use off-the-shelf hardware that is generalized in function and has many user features. The only custom development in most business applications is the application software itself. So, the design problem with hardware is opposite that of DBMSs. For hardware, the methodology authors do more detailed levels of development than is necessary in most business applications. You will see this problem again when we discuss service object definition.

Now let's consider allocation of functions to the other subdomains. The human interface is exactly what you think it is, the interactions with people, usually through a terminal device, that provides the essential inputs and outputs of the application. The human interface is discussed poorly in the OOD books that do exist (including all of those in the ref-

erences of this chapter) because of the traditional lack of human users in object-oriented applications. Because of this lack, they are discussed in Chapter 14 as one of the 'forgotten activities' of systems analysis and design.

In general, the activities that provide human interface control, such as screen interactions, are recommended to be relegated to the human component of the application. Again, there are no *compelling* reasons for blindly making this decision, therefore it is subject to analysis. Activities that can be grouped across objects, such as line control, error message display, and screen reads and writes can all be abstracted out of the individual objects and placed in reusable, generic objects. The actual editing of data from screens should remain with the original object unless there are sufficient similarities across screens and data items to warrant abstracting them out as well, or unless the functions will be assigned to human interface hardware. To perform this abstraction requires listing all the detailed, primitive actions required of screen interactions for each object, identifying which actions are performed automatically by the DBMS or other application software and removing them from the list, and re-evaluating the remaining items to determine whether or not there are commonalities across objects.

This primitive level of detail may be deferred automatically when you relegate all *screen interactions* to the human interface. This deferral allows you to build the interface during prototyping even though you may not know all of the primitives during the first iterations. In other words, allocating screen interactions to the human interface is a means of deferring *detailed* design decisions until initial prototype development.

The more distributed devices and processors, the more likely that processing might be allocated to firmware embedded in otherwise unintelligent devices. For instance, automatic teller machines include some intelligence for editing magnetic strip information from the cards used for withdrawal and deposit of funds from banks. They can, for instance, tell what type of card, such as Visa, is being used, and whether or not the personal ID number (PIN) is a valid combination of digits. They cannot tell whether or not the PIN matches the card number entered because that requires access to a database that is not stored locally. In addition, specific hardware functions, such as accepting a deposit envelope, are functions that would be allocated to hardware.

Allocation of processes to hardware/firmware is determined by the need for fast response time, minimum communication delay, and minimum processing time. Whenever any of these three constraints are present in an application's functional specification, hardware process allocation should be investigated. Some authors recommend that allocation to hardware can include functions to be performed by the resident operating system. When there is access to these functions and they can be used as generics, this is a useful, time-saving idea. So, for instance, in systems such as Unix and Smalltalk, where the environment, operating system, and application are essentially inseparable, thinking of operating systems and hardware as one simplifies design thinking.

Finally, we have allocation of processes to software. This allocation assumes that all problem-domain processes not already allocated elsewhere will be implemented in software in the software domain. This allocation includes remaining service and problem domain objects after the other allocations are complete. Now, let us turn to ABC Video to see what allocation means in this application.

ABC Video Example of Subdomain Allocation

ABC's rental application will be an interactive, multithread set of processes which will service up to six threads of control, with growth to some higher number. Therefore, the concurrent processing requirements of the application should be considered when allocating processes to subdomains to ensure that timing requirements will be met.

To refresh your memory, we had decided to use an SQL-compatible database to implement the application. We can interface the SQL language with other languages, but, as is typical of most DBMS software, all data accesses must go through the DBMS. This implies that the create, retrieval, update, and delete (CRUD) functions will all be allocated to the data subdomain as discussed above.

By doing this allocation, we explicitly are deciding what is and is not object-oriented. SQL is not object-oriented. Therefore, any functions performed in SQL are not object-oriented. The design can proceed in an object manner until the primitive level is reached, then the design is completed in SQL.

If we look at the output from the analysis where we allocated objects to processes, we can identify all those processes relating to these functions. Each object has simple CRUD functions as well as a need for CRUD functions on a user-view of the database that incorporates *Customer*, *Inventory*, and *VideoOnRental*. Eventually, for SQL implementation, we will collapse the superset objects back with the class/objects and will control the use of add and read functions by logic in the SQL DBMS application code. Any access control on superset objects is controlled by the DBMS.

Figure 11-20 processes are listed in Table 12-3 with their subdomain allocations. First, consider the data subdomain. From Table 12-2 we know that we *can* allocate the data functions based on application requirements. We are using a non-object DBMS and have no constraints on processing. Part of the attraction of the fourth generation database is its ease of use, therefore, anything that can be allocated to the DBMS should be. As Table 12-3 shows, all CRUD functions are allocated to the data function. Similarly, printing, which interfaces with external devices, is allocated to hardware. Print control is allocated to hardware because in a LAN, spooling and printing are network operating system functions that are not under application control.

All data entry functions are allocated to the human interface for design and control. Remaining processes are allocated to the software subdomain.

Draw Time-Order Event Diagram

Rules for Drawing a Time-Event Diagram

A time-event diagram graphically depicts the timing constraints and events that trigger related objects, showing sequences of processing, concurrent processes, and nesting of processes across objects. Time-ordered event diagrams show neither flow of control nor if-then-else logic. These diagrams are showing what *can* happen in time, including required timing. The time-order event diagram becomes the basis for decisions about concurrent processes and is helpful in identifying service-object needs of the application.

The diagram is a two-dimensional graphic with objects listed down the left axis and time, broken into segments corresponding to events in the application, along the horizontal axis. For processes that might run concurrently, multiple lists of the objects are shown. Synchronization of concurrent events is shown by the divergent lines returning to one event at some point (see Figure 12-9).

Two formats for time-event diagrams are used. One shows deviations from an otherwise horizontal line with events and critical times demarcated by vertical bars (see Figure 12-10). The other format shows rising steps to mark events and critical time slots within the main object (see Figure 12-11). If one diagram per transaction is created, the rising step method is preferred because it is easy to see the points of change. If one diagram per application is drawn, the information can be presented more compactly with the horizontal line method.

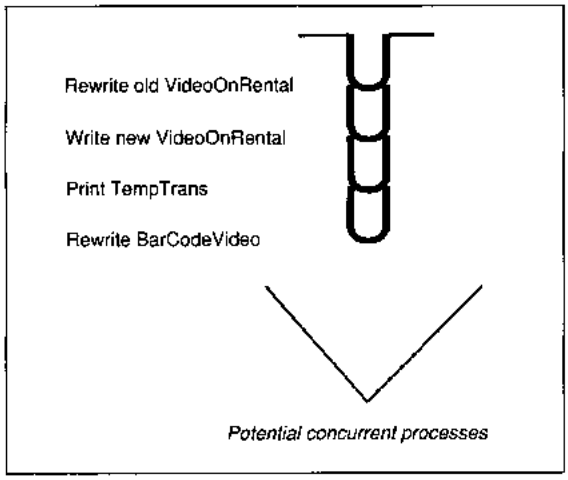


FIGURE 12-9 Potentially Concurrent Processes

TABLE 12-3 Process Subdomain Assignments

Process Name	Subdomain			
	Data	Hardware	Process	Human
EnterCustPhone				X
ReadCust	X			
CreateTempTrans			X	
RetrieveVOR	X			
DisplayTempTrans				X
EnterBarCode				X
RetrieveInventory	X			
ComputeTempTransTotal			X	
EnterPayAmt				X
ComputeChange			X	
DisplayChange				X
UpdateInventory	X			
WriteVOR	X			
PrintTempTrans		X		
EnterBarCode				X
RetrieveVOR	X			
DisplayTempTrans				X
AddRetDateTempTransVOR			X	
Add1toVInv			X	
UpdateInventory	X			
ComputeLateFees			X	
WriteVOR	X			
EnterCustomer				X
CreateCustomer	X			
EnterVideoInventory				X
CreateVideoInventory	X			

Diagram segments are defined as event-driven or clock-driven. For time-constrained segments of the diagram, the allowable maximum time is labeled along the horizontal axis (see Figure 12-12). For event-driven segments, the event is identified on the horizontal axis. Actual drawing requires knowledge of the problem domain requirements for processing.

The steps to creating a time-event diagram are:

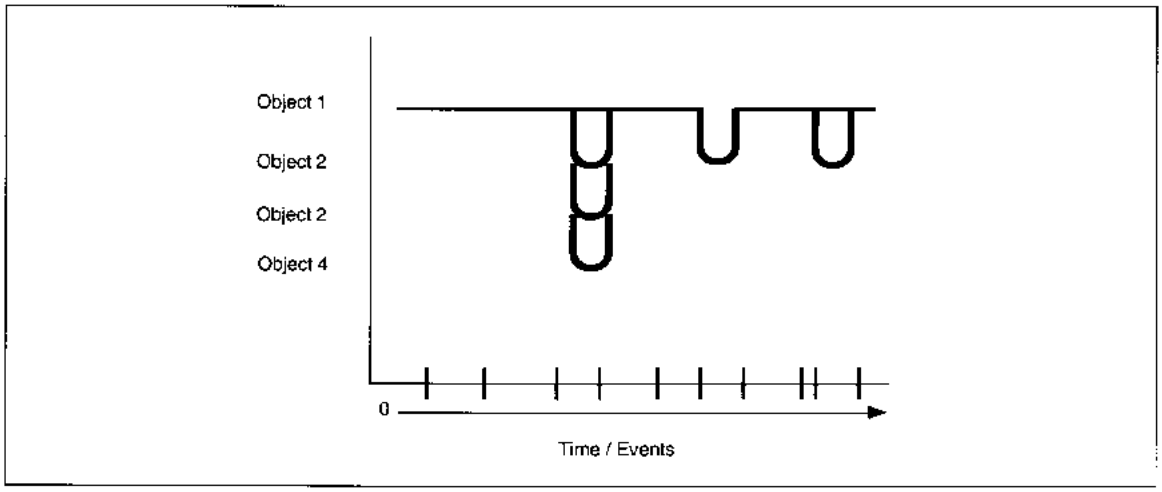


FIGURE 12-10 Horizontal Time-Event Diagram

1. Define all allowable transactions in the application.
2. Define the processing steps for each transaction.
3. For each transaction, design a time-event diagram reflecting the dependence or independence of processing steps.

ABC Video Example of a Time-Order Event Diagram

For ABC, Table 12-4 shows the transactions allowed in the application. The transactions should have no surprises by this stage of design, and should be closely related to the processes defined for each

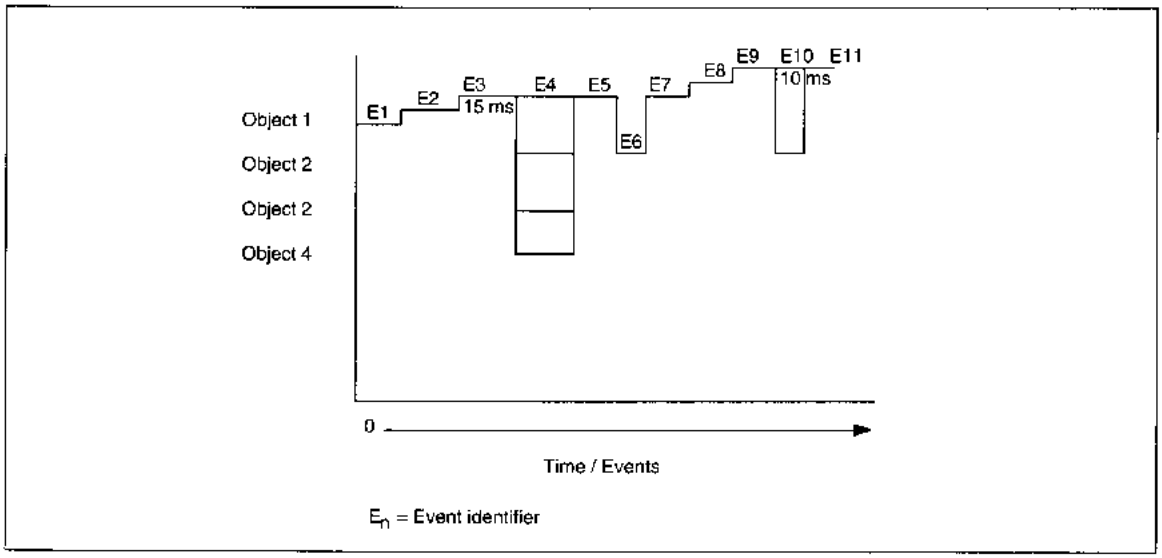


FIGURE 12-11 Rising Step Time-Event Diagram

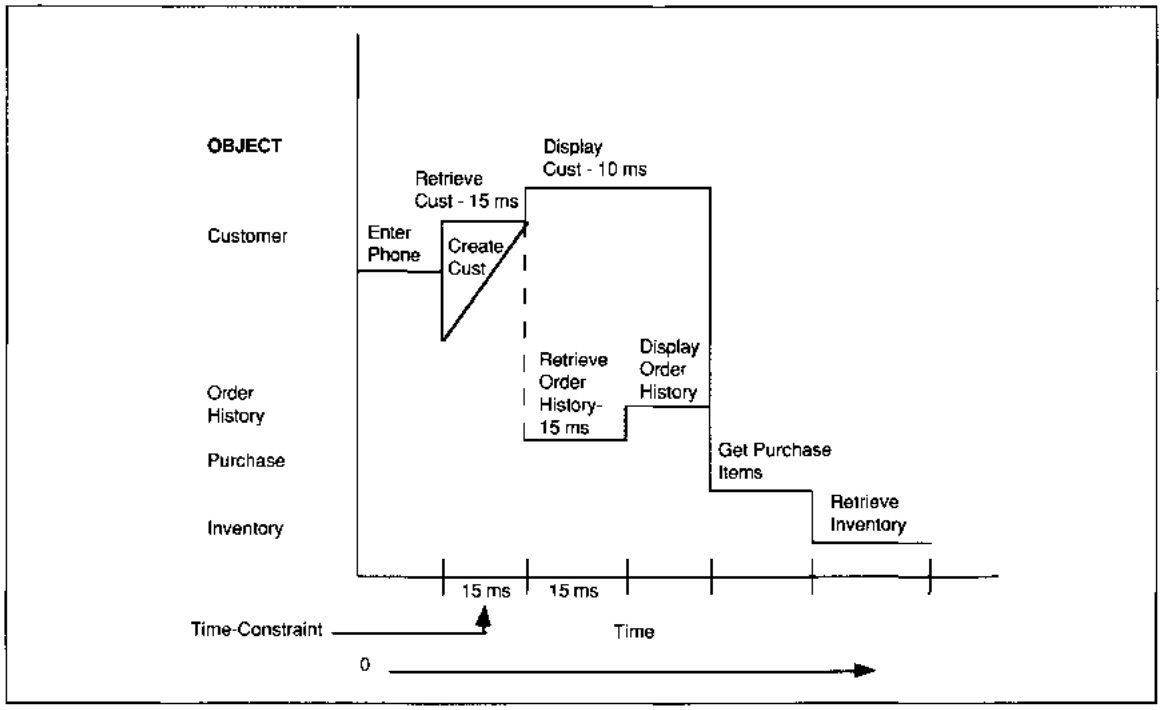


FIGURE 12-12 Diagram Segments Identified as Time-Driven or Event-Driven

object. Some objects, such as *TempTrans*, have processes that relate to more than one transaction, while other objects each have processes that reflect one transaction, such as for *Customer*.

Of the transactions shown, we will discuss two that are representative of the others: video inventory additions and rental processing.

First, we describe what happens for a *VideoInventory* addition. This step requires detailed knowledge of the specific processing to be performed. This knowledge comes from user interviews, study of current procedures, and so on. Subprocess details should be based on the process-object assignment list (Figure 11-20). If there are discrepancies between the use of objects here and the list, the list should be revised to reflect this more detailed level of thought. The steps to adding inventory are:

1. Enter a new *VideoId* and remaining information for a particular film.
2. When the *NumberOfCopies* is entered, add the new video information to *VideoInventory*. Begin prompting for *BarCodeId* until the number of bar codes is equal to *NumberOfCopies*.
3. As each *BarCodeId* is entered, add the new *BarCodeVideo* entry to the database.
4. When the number of *BarCodeIds* entered is equal to *NumberOfCopies*, signal completion of the transaction to the clerk and end processing.

Figure 12-13 shows the time-event diagram for the processing steps about video inventory creation. Notice that two objects are involved: *VideoInventory* and *BarCodeVideo*. Even though *VideoInventory* is begun first, its processing is completed before *BarCodeVideo* processing takes place. The processes are related in that the *VideoId* is passed to the *BarCodeVideo* process, but they are otherwise

TABLE 12-4 ABC Transaction List

Object	Transactions
Customer	Create Retrieve Update Delete
VideoInventory	Create Retrieve Update Delete
BarCodeVideo	Create Retrieve Update Delete
VideoOnRental	Rental without Returns Rental with Returns Returns without Rental Returns with Rental
Video History	Create Retrieve
Customer History	Create Retrieve
EndOfDay	Create Retrieve Delete

independent. There is no necessary concurrency within the transaction.

The rental transaction shows that several processes might be concurrent. First the steps to completion of a rental process are:

1. Get the entry and determine its type (either *CustomerPhone* or *VideoId*).
2. If the entry is *CustomerId*, get all relevant customer information (e.g., name, address, and so on).
3. If the entry is *VideoId*, get the corresponding *VideoOnRental* and place it in memory.

   Use *CustomerId* to get all relevant customer information (e.g., name, address, and so on),
4. Get all current outstanding rentals (i.e., either unpaid late fees or unreturned rentals).
5. Compute *LateFees* on returned tapes.
6. Compute *TotalAmountDue*.
7. Display all information.
8. Process returns and redo steps 5–7 until no more returns.
9. Get *VideoIds* of new rentals until end of transaction is signaled. For each, get *VideoInventory* and *BarCodeVideo* information; format and display the relevant information; recompute and display *TotalAmountDue*.
10. At transaction end, process payment and make change until *TotalAmountDue* equals zero.
11. Write new *VideoOnRental* entries; update and rewrite old *VideoOnRental* entries; print *TempTrans*; update and rewrite *BarCodeVideo* as required; end transaction.

The first event, data entry, results in one of two possible processes being invoked. These are shown with dotted lines on the diagram to show that only one is running at a time. If the *VideoId* is entered, then we have a choice to either nest getting the customer or transfer control. If we transfer control, the video information must have been stored in memory for the first *VideoOnRental* to avoid passing unnecessary data. If we do not transfer control, and nest retrieval of customer information, then the customer information is unnecessarily passed through the retrieval process for *VideoOnRental*. The best object-oriented decision would be transfer control to maximize information hiding here, but we can treat these accesses as one if the DBMS supports a user view that links the relevant information. SQL DBMS does provide user views and we select that option. (Make sure you read the appendix for true object-oriented design of this information. It is significantly different.) Once *VideoOnRental* is accessed, then, the related information from *VideoInventory*, *BarCodeVideo*, and *Customer* are all present automatically (see Figure 12-14).

Eventually, we loop through getting all current outstanding rentals from *VideoOnRental*. This itera-

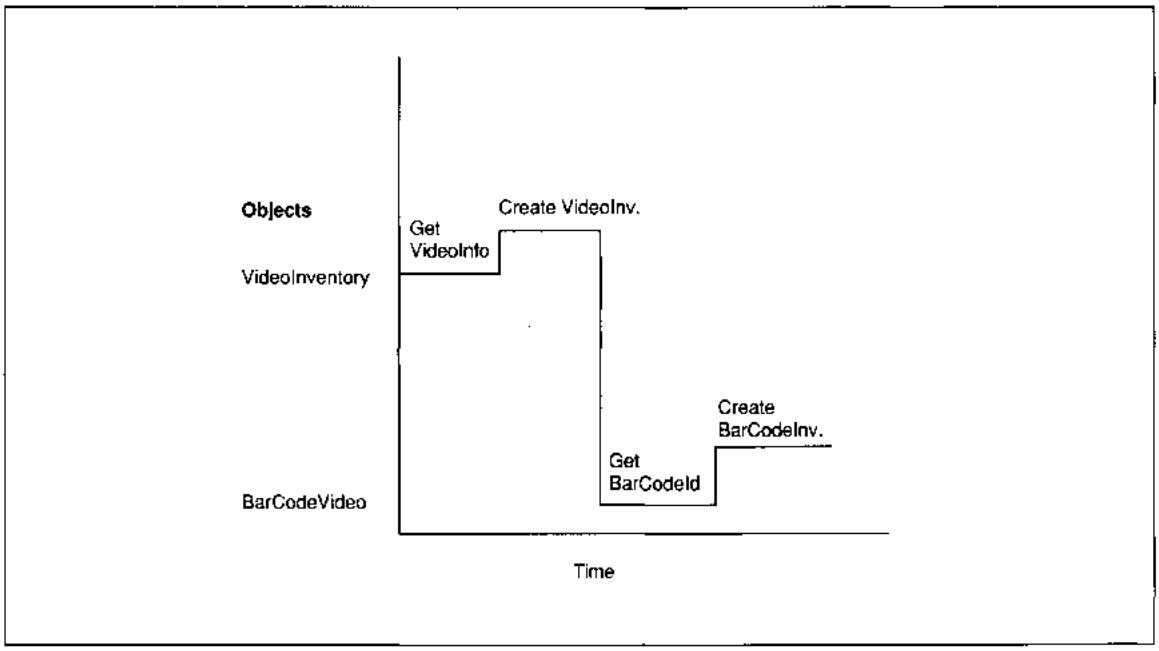


FIGURE 12-13 Time-Event Diagram for Inventory Creation Transaction

tion can be programmed to run until a return code indicating no more videos on rental are present. This return code, then, becomes the event to trigger the next step of the process.

Control is passed to compute Late Fees on returned tapes that will require a count of the number of *VideoOnRentals* in memory to be maintained and passed to control this process. Having processed late fees until this count is reached triggers the next step to compute *TotalAmountDue*. This is a one-time event at this point, and its completion leads to display of all current customer and rental information on the user screen.

At this point, if there are new rentals, the *BarCodeIds* are entered. This triggers obtaining *BarCodeInventory* and *VideoInventory* information. To simplify memory processing, we have a choice similar to that above for customer and *VideoOnRental* in step 3. In this case, the decision is between treating *BarCodeVideo* and *VideoInventory* as separate and independent or nested or the same. In order to treat them the same, we must be accessing a user view that contains the relevant information. Again, SQL allows user views, and we use the user view that collapses this activity from two to one. As each video's information is displayed, the *TotalAmountDue* is recomputed and redisplayed.

Upon receiving the trigger that the rentals, or returns, are complete, payment processing takes place and continues until *TotalAmountDue* equals zero. At that time, all of the *VideoOnRentals*, *BarCodeVideo* locations, and video history counts (for returns) are updated. These are once again assumed to be in the same object as a result of having user view capabilities in SQL.

Determine Service Objects

Guidelines for Determining Service Objects

Service objects perform background scheduling, synchronizing, and multitasking control for the application. The activities performed by some service

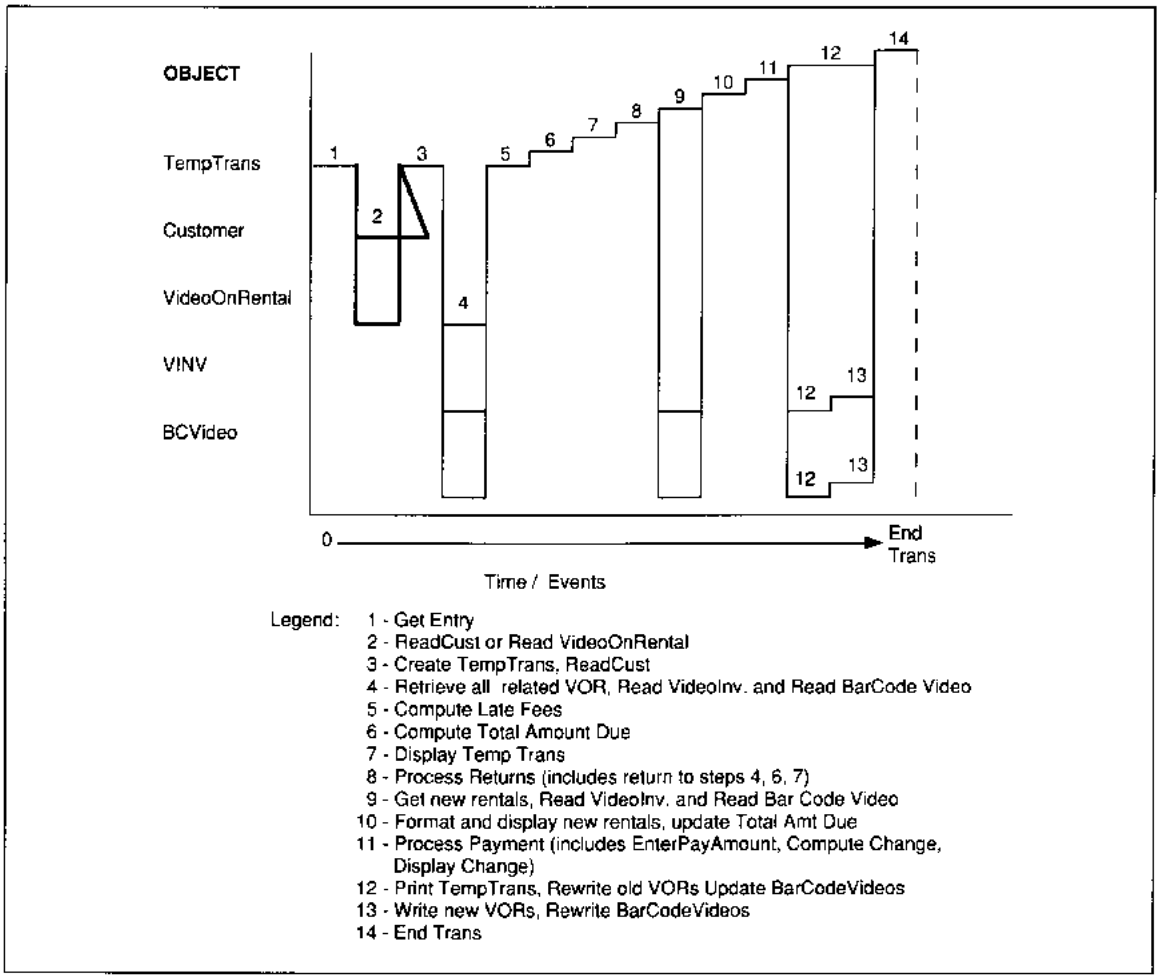


FIGURE 12-14 Time-Event Diagram for ABC Video Rental Transaction

objects are analogous to those of an operating system in a mainframe environment which provides job management, task management, memory management, I/O management, and data management. For that reason we will digress a minute to discuss these operating system functions, relating them to service objects.[4]

Job management routines initiate processing for individual applications. In multitasking applications, that means that the first scheduling tasks are loaded and turned over to the task management routines for execution. In mainframes, there are multiple jobs, sometimes as many as 50, executing concurrently. The job management routines keep track of all jobs active in the system.

The task manager monitors and tracks individual steps within a multistep set of sequential processes (i.e., a job). Task management is similar to monitor-

4 This discussion is necessarily short. For further information see Per Brinch Hansen, *The Architecture of Concurrent Programs*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1977.

ing done for multiple threads of control for concurrent processes. The work of job and task manager routines are similar and include:

- Load, schedule, execute
- End, abort
- Get/set process attributes
- Create/terminate process
- Wait for time
- Wait/signal event
- Get/set process attributes for jobs, files, or system data

Multiple-thread management requires both job and task management. Think of individual transactions as analogous to jobs to be managed, and of individual steps to completing a transaction as tasks, or processes in OOD terminology. The job management, transaction routines manage whole transactions, and task management routines manage atomic processes to perform the transaction.

Monitoring of individual processes (or transactions) and sequences of processes, one per thread, is accomplished either by stacks (sometimes called heaps) or queues, depending on the operating system software. The stack commands are *push* to add something to the stack and *pop* to take something off the stack. The queueing commands are *enqueue* and *dequeue*, to add and delete items, respectively. The stack (or que) items, in multithread control, include the name of the task, its current execution status (i.e., running, idle, or waiting), and the address of the next command to be executed. One set of stacks is managed for each transaction, and one set is managed for each process. Stacks operate on a last-in, first-out principle while queues are first-in, first-out.

Similarly, the I/O manager and data managers act together to perform physical inputting and outputting of information to central processing unit (CPU) memory. The I/O manager interacts with terminals, printers, and other devices that are moving information physically into and out of the computer. The data manager interacts with secondary storage devices, such as disks. The activities performed by these managers include file manipulation and device management. The key activities include:

File Manipulation:

- Create/delete file
- Open/close
- Read, write, reposition
- Get/set file attributes

Device Management:

- Request/release
- Read, write, reposition
- Get/set device attributes

These tasks are usually provided in primitive form by the operating system and in a more abstract form by a DBMS. The more sophisticated the software environment, like a DBMS, the more likely the services are provided by the environment.

Finally, memory management keeps track of the location of each item, in random access memory (RAM). Recall that all data and programs must be memory-resident to be executed. In dynamic applications in which modules and data are being moved into and out of memory constantly, memory management is a crucial function. The main functions provided by the memory manager include:

- Allocate/delete memory (can be dynamic or static)
- Track used and free memory location by task
- Track used and free memory within each task's allocation
- Garbage collection (identify and erase or write-over unused objects)

All operating system management is accomplished by cooperating processes that use event-driven interrupts to provide services in the system. Interrupts at the operating system level are called **supervisor calls** (SVCs). The implementation of SVCs differs across operating systems.[5]

5 For a more complete treatment of this information, see any operating systems text. Some good ones include A. J. van de Goor, *Computer Architecture and Design*, Reading, MA: Addison-Wesley Publishing Company, 1989; Anthony P. Sayers, *Operating Systems Survey*, NY: Auerbach, 1971; J. Peterson and A. Silbershatz, *Operating System Concepts*, Reading, MA: Addison-Wesley Publishing Company, 1983.

Now, let's relate this operating system information to applications. All of these functions are required for the three types of control provided by service objects. If you are working in a Unix or Smalltalk environment which already have been used for application development, many of these functions should already be available for reuse. If you have to write your own, you need to test and retest these functions *very thoroughly* to ensure proper application functioning. In any case, you need to decide which of the service object functions are needed and provide them for your application.

The steps to identifying the service objects are:

1. Examine the event diagram and identify each process as sequential or concurrent, and, if concurrent, as independent or cooperating.
2. Define the service needs for loading the object, processing the object, synchronizing the process to others, and sending any messages the object might generate.
3. Compare this list to one specific to the target operating environment that identifies reusable service objects that can be used by this application.
4. Enter the name, language, and any other information needed to identify the reusable object. For all service objects, make sure that the class, object, event, and/or process using the service object are identified.
5. When all reusable objects have been identified, the remaining service objects included in the remaining tasks are divided among the four subdomains as appropriate for module specification.

In general, all applications need scheduling objects (see Table 12-5). The need for synchronization and multitasking are determined by the time-event diagram and whether or not the objects are concurrent and multiuser. Table 12-5 shows that concurrent, single-user processes need synchronization while concurrent and multiuser objects need synchronizing and multitasking services. Multiuser, sequential processes, like ABC, require both scheduling and multitasking services.

TABLE 12-5 Decision Table for Service Object Type Requirements

Problem Domain Object Characteristics:				
Sequential	Y	Y	—	—
Concurrent	N	N	Y	Y
Multiuser	N	Y	N	Y
Service Objects Required:				
Scheduling	X	X	X	X
Synchronization	—	—	X	X
Multitasking	—	X	—	X

ABC Video Example of Service Objects

First, we examine the time-event diagram to identify each related process as sequential or concurrent, and independent or cooperating.

There are three possible sets of concurrent processes within one rental transaction shown on Figure 12-15 as circled and numbered sets. The other processes are sequential. Our decision on concurrency, then, is based on the implementation environment. Let's say that SQL supports multithread but not multitasked processing, therefore, we need to decide sequential ordering of the processes and how the processes will be performed in SQL.

Next, for each process, define the service needs for loading the object, processing the object, synchronizing the process to others, and sending any messages the object might generate. SQL supports user views. By creating user views to link *VideoInventory* to *BarCodeVideo*, and *VideoOnRental* to *Customer*, *VideoInventory*, and *BarCodeVideo*, the opportunity for most concurrency disappears in one database access that retrieves all the related information.[6]

6 See Chapter 12 appendix discussion of ABC in which the service object discussion results in a different outcome.

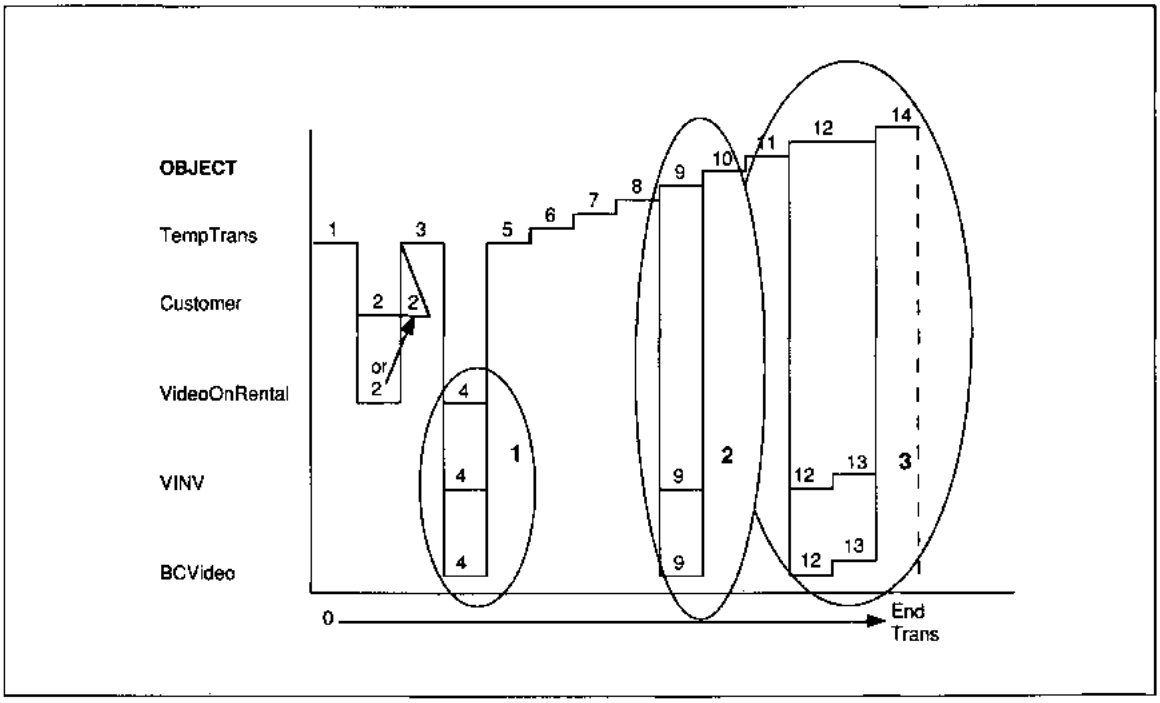


FIGURE 12-15 Potential Concurrent Sets of Processes

Even though we have removed concurrent object processing from the diagram, we still have both transaction level and process level service object requirements. Transactions and processes all need scheduling, including processes that load, store in memory, initiate, terminate, monitor events, and possibly provide message communications between objects.

This list is compared to our target operating environment: SQL on a PC LAN running Novell Netware.™ The services are all provided transparently by the operating environment and are not needed to be developed in primitive form for ABC's application. Even though the target environment is not object-oriented, the need for service objects disappears because these are all services provided in the operational environment.

The next step is to examine a current library of reusable objects for use as problem domain processes. Since ABC's environment is new, there is no reusable library; therefore, any modules would need specification and development.

Develop Booch Diagram

Guidelines for Developing Booch Diagram

Booch diagrams, also called **module structure diagrams**, provide a graphical summary of the class and object information in the entire application. The icons for drawing the diagram are shown in Figure 12-16 with service objects in vertical rectangles with no other detail beyond their name, and problem domain objects in vertical rectangles with smaller ovals to identify the object and horizontal rectangles to identify the individual processes. One diagram connecting the domains as required is drawn; then one Booch diagram for each subdomain (or for the whole project if it is small) is developed.

The steps to drawing a Booch diagram are:

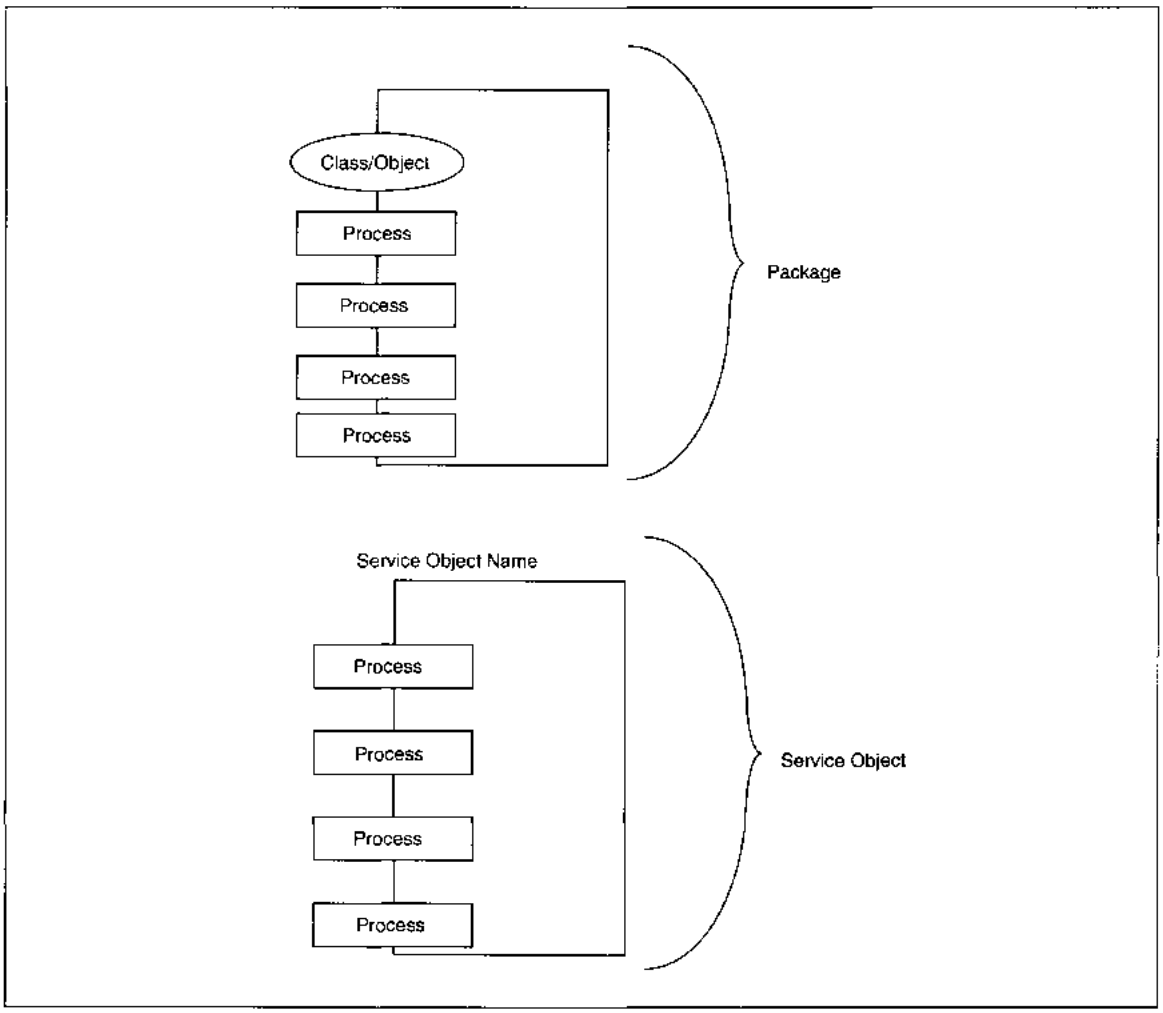


FIGURE 12-16 Booch Diagram Icons

1. Draw the Booch icons (see Figure 12-16) relating to service and problem domain objects.
2. Evaluate and choose a scheme for connecting the objects via messages.
3. Draw lines between objects to signify the legal message connections.
4. Define message processing scheme.

Service objects selected for controlling application operations are arranged by personal preference, but can be grouped by function performed: scheduling, synchronizing, and multitasking within subdomain. The service objects described in the previous section are shown with subdomain grouping in Figure 12-17.

Problem-domain objects are obtained from the process-object assignment list developed during analysis. This table is now reversed with the information arranged by object for this diagram. During the reversal process, a reevaluation of process-object assignment should be made to ensure that the processes are associated correctly with their necessary objects. Subdomain groups may be maintained on

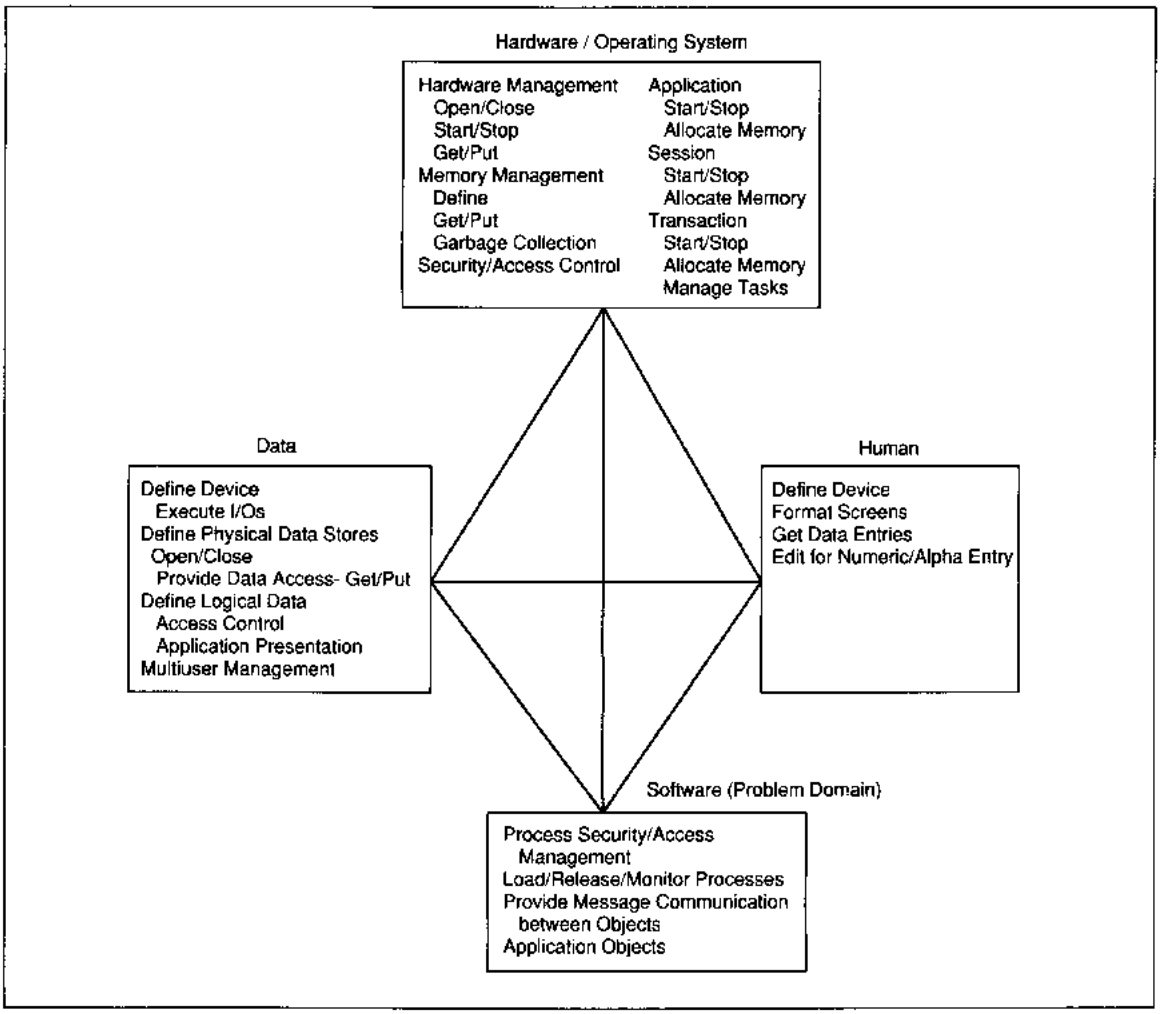


FIGURE 12-17 Service Objects by Subdomain

the diagram which means that we may have new superset objects to define the split between objects for subdomain processing.

Processes that are candidates for generic, reusable object development should be marked consistently in some way, for instance by **bold** or *italic* print to identify them visually. A quick glance at the diagram gives the viewer a sense of the extent to which reusable objects and processes are being leveraged in the application.

After the icons are drawn, they are *played with* to evaluate different message passing schemes. There is no one *right* way to do message passing, but there are definitely some methods that are better than others. We will walk through a reasoning process for message passing definition in the ABC Video example. In general, the goal of messages are

1. To accomplish the application's tasks.
2. Pass minimal information and pass only to objects requiring information.
3. Minimize the potential for bottlenecks.
4. Maximize the potential for application throughput.

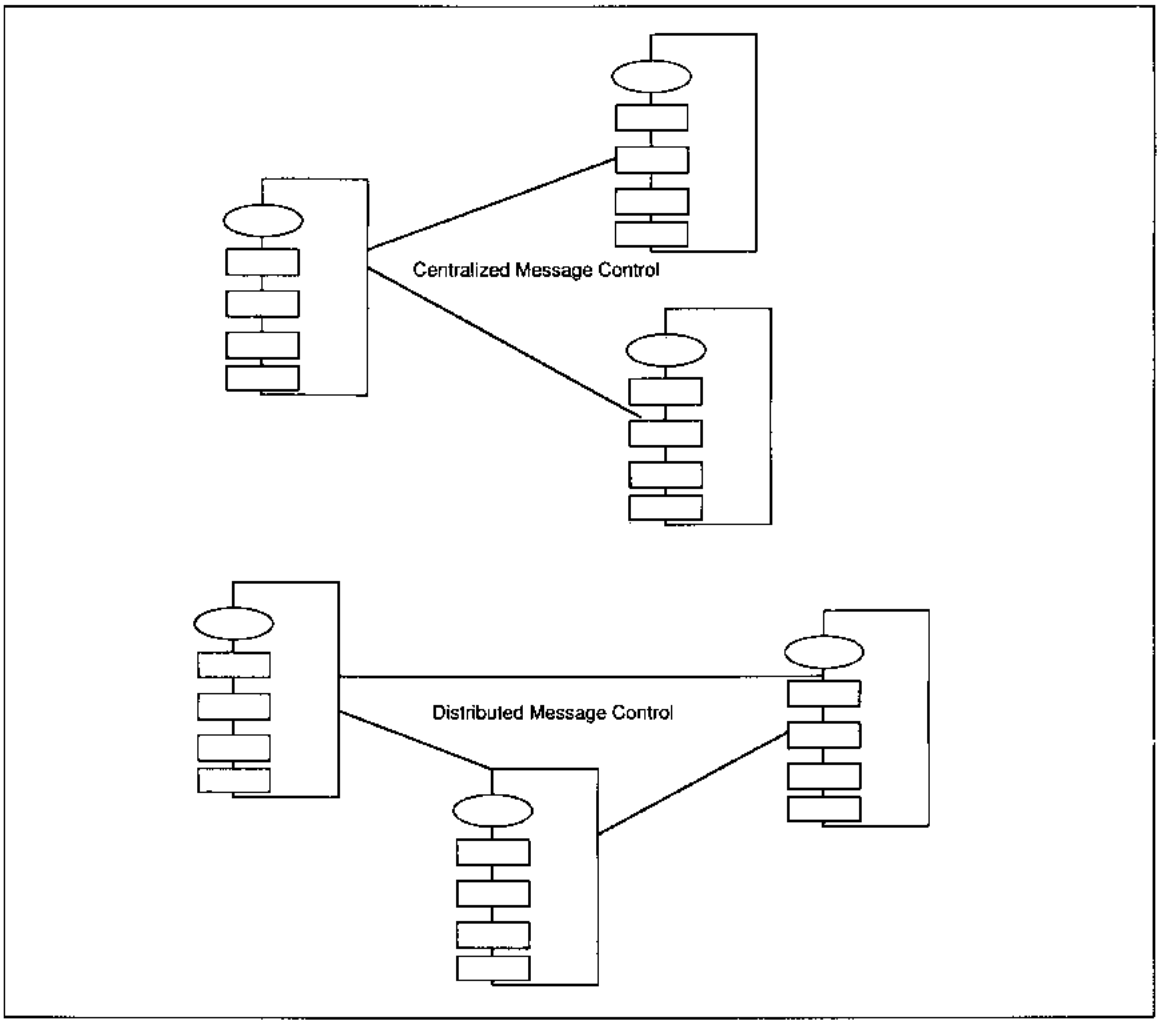


FIGURE 12-18 Sample Configurations of Object Message Passing

The evaluation of alternatives is to determine the best throughput scheme of message passing without creating bottlenecks, while accomplishing the first two goals. Booch suggests a **3x5 approach** to this evaluation in which, rather than drawing the diagram icons on paper, the information for each object is written on a 3" × 5" card. The cards are arranged spatially in different configurations on a large piece of paper with lines drawn to signify the required inter-object message communications. When a configuration is identified that might be useful, it is annotated for further analysis. Figure 12-18 shows two different configurations for a simple application. You can see how, if you have 20 or 30 objects, the 3" × 5" method simplifies evaluation of message passing schemes.

All further alternative configurations are evaluated to determine message traffic. **Message traffic** is the number and direction of messages in the system. Overall, the goal is to minimize the number of messages passed for any single transaction, while not overloading any single object with message traffic

related work.[7] The minimum number of messages is $n-1$, where n is the number of packages needing to communicate in the application. That is, once initiated, each package must communicate with at least one other package. The centralized message control scheme shown in Figure 12-18 shows an example of $n-1$ messages. The arrangement with the best message traffic configuration is selected for prototype development, and the design process continues.

ABC Booch Diagram

Before we can develop a Booch diagram, we need to digress and redefine some application needs to fit the SQL environment.[8] The drawing of packages normally assumes no consolidation of functions or data via user views, but we have collapsed our processing to take advantage of SQL features. Therefore, Table 12-6 shows the effects of user views on data domain processes: the 11 data processes are now eight consolidated processes. The remaining subdomains are not affected by the data changes.

First, we will draw the packages based on what we now know to be the design of the application (see Table 12-6). There are four data packages: *Customer*, *VideoInventory*, *UserView1* which includes VideoOnRental, VideoInventory, BarCodeVideo and Customer, and *UserView2* which includes VideoInventory and BarCodeVideo (see Figure 12-19). The related processes for those data objects are placed in horizontal rectangles in their respective packages.

There is one scheduling service object (which we may not need because of the environment) that includes initiation and termination of the application, user sessions, and transactions. There is an interface service object to provide all display and input from personal computers (see Figure 12-19). The hardware service object contains only one process for printing TempTrans. Finally, the *TempTrans* object contains the data and problem domain processes that are the core of rental processing.

Next, we try different configurations of the objects to develop a message passing scheme that will provide necessary processing and information to called objects, while minimizing the communications overhead in the application. Figure 12-20 shows one reasonable message passing scheme that follows the logic of processing. The scheduling object passes control to the interface object which has some choices. The interface object could pass, for instance, a *CustomerPhone* to either *TempTrans* or *Customer* to initiate rental processing. If the pass is to *Customer*, it could return and pass the customer information to *TempTrans*, or *Customer* could continue and initiate *TempTrans* directly. You see how the options can build and get complex. We will opt for a fairly traditional scheme in which the *Interface* will pass any rental transaction data to *TempTrans* which will determine what to do with it. This decision is reflected by the line connecting *HumanInterface* with *TempTrans*.

TempTrans then initiates one of three data retrievals: *Customer*, *UserView1*, or *UserView2*. The data is returned and *TempTrans* continues processing. This method of passing provides the most information hiding between objects, but could result in a bottleneck within *TempTrans* which is controlling all of the interobject communication for the problem (e.g., software), hardware, and data subdomains. This is a potential problem that would be checked during prototype development.

The *HumanInterface* object also communicates directly with *Customer* and *VideoInventory* for create processing which does not require *TempTrans*. All completed transactions, regardless of type, return to the *Scheduling* object to terminate the transaction.

Define Message Communications

Rules for Defining Messages

The next step after the Booch diagram is to actually define message contents to provide interobject interfaces for the application. A table is created to

7 This would cause a bottleneck.

8 Don't forget to read the Chapter 12 Appendix for a complete discussion of object-oriented design using an object-oriented development environment.

TABLE 12-6 Consolidated Process Subdomain Assignments for Oracle

Process Name	Subdomain			
	Data	Hardware	Process	Human
EnterCustPhone				X
ReadCust	X			
CreateTempTrans			X	
RetrieveVOR (includes VideoInventory, BarCodeVideo, and Customer)	X			
DisplayTempTrans				X
EnterBarCode				X
Retrieve BarCodeVideo (includes VideoInventory)	X			
DisplayInventory				X
ComputeTempTransTotal			X	
EnterPayAmt				X
ComputeChange			X	
DisplayChange				X
WriteVOR	X			
PrintTempTrans		X		
EnterBarCode				X
DisplayTempTrans				X
AddRetDateTempTransVOR			X	
AddItoVInv			X	
Rewrite VOR data	X			
ComputeLateFees			X	
WriteVOR data	X			
EnterCustomer				X
CreateCustomer	X			
EnterVideoInventory				X
CreateVideoInventory	X			

document the specific requirements of each message (see Table 12-7). The objects that act as clients are listed in the *Calling Object* column, service objects are in the *Called Object* column. This information should come from the Booch diagram coupled with the Process table generated during analysis that identifies objects with the processes that act on them. The *Input Message* column describes the data that is sent

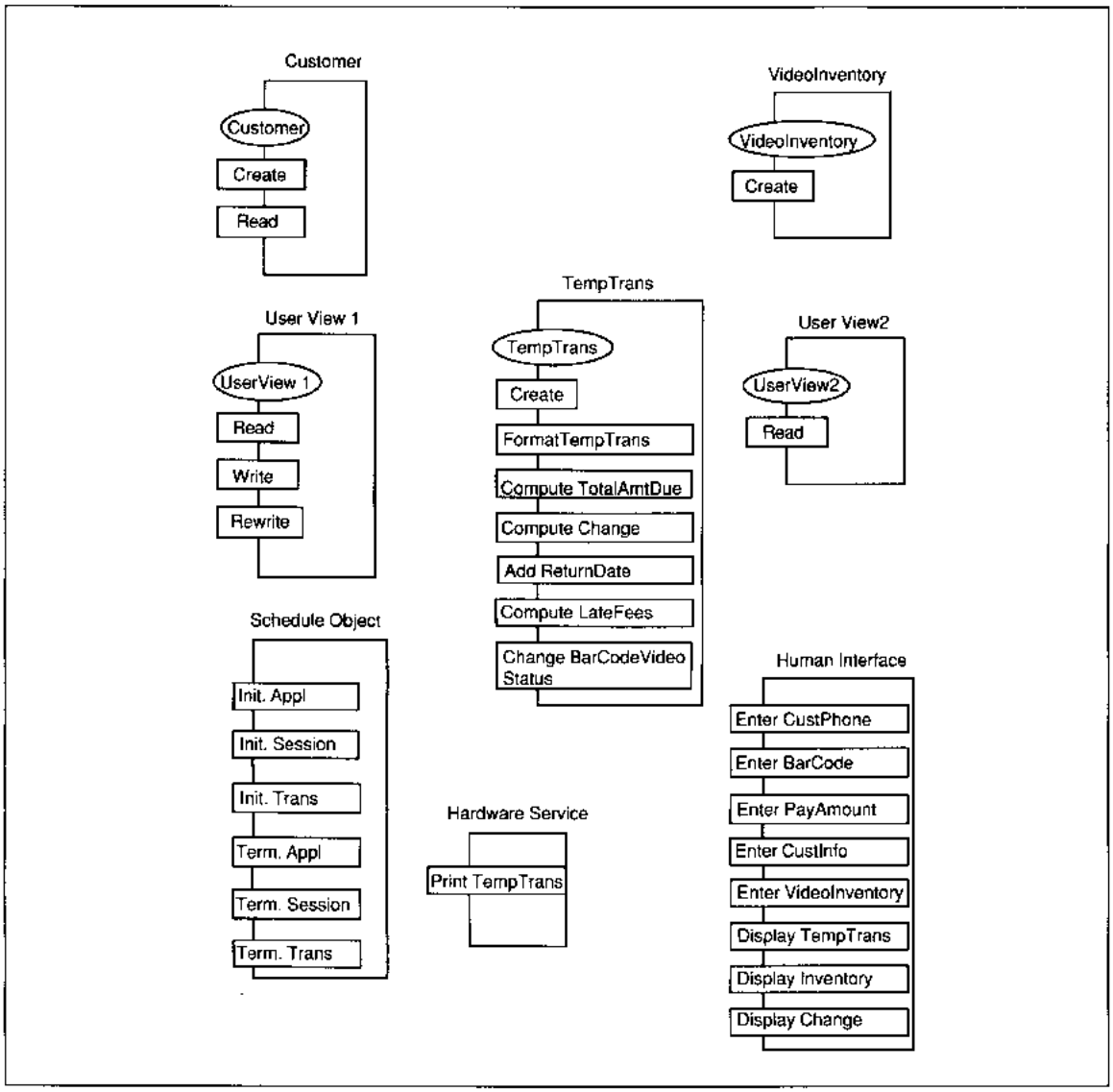


FIGURE 12-19 ABC Rental Booch Diagram Objects

as part of the calling object message to be processed. The output message is the result data that is sent on (or returned) by the called object after processing. The columns *Action Type* and *Return Object* are optional. The action type describes the process to be performed in terms of CRUD or other processing. The return object provides continuity of processing logic when the called object does not return directly to the calling object.

For each process-object pair defined in the Process Definition List, we will have one input message to initiate processing and, if needed, an output message which reports the results of processing. The message list contains one column for each of the

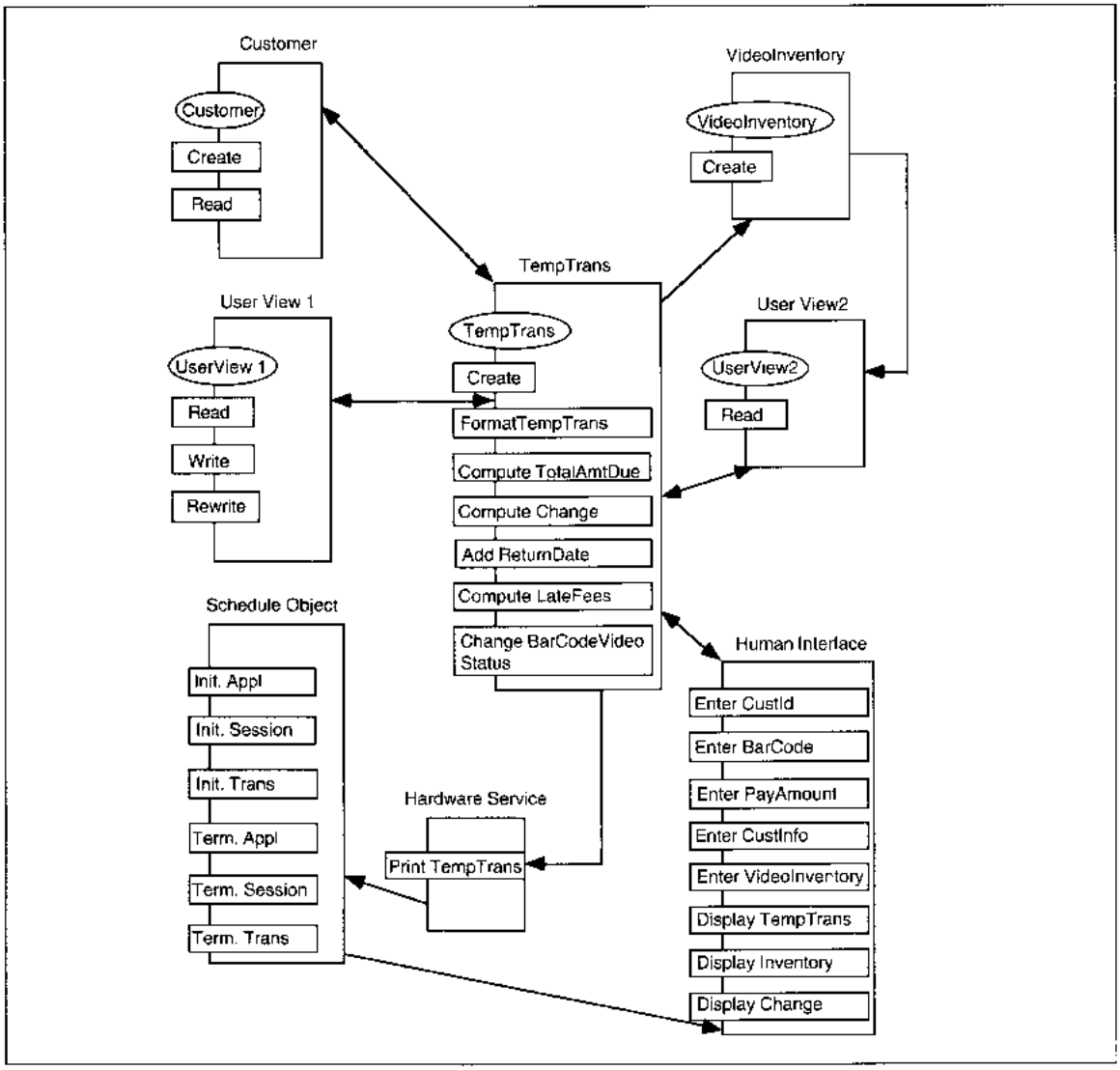


FIGURE 12-20 ABC Booch Diagram Message Passing Scheme

types of information shown in Table 12-7. The steps to creating the message list are:

1. Make a table with headings as listed in Table 12-7.
2. Refer to the list of all object-process combinations. The objects from that list are listed in the 'Called object' column. The processes from the process list are placed in the 'Input message' column.
3. Next, decide both the 'Calling object' and other 'Input message' entries.

These two definitions seem to go together because as we define the input message, we know the information required to perform the process. Once we know the information to perform the process, we

TABLE 12-7 Message List Contents

Header	Contents
Calling object	Identifies the client.
Called object	Identifies the server.
Input message	Identifies the process to be performed and any input parameter data needed to perform the process, for instance, the data type for polymorphic processes.
Output message	Defines the output to be passed, if any.
Action type	Defines the process as Read, Read/Write, Write, Display, or Print.
Return to	Identifies either the object to which the result is returned or a nested object for further processing, if any.

decide which object has that information to pass it on. This step determines much of the logical process flow from one encapsulated object-process to another. The **logical process flow** defines the sequence of processing in the application.

4. Define the 'Output messages' by determining what type of information is required next from each process as it completes. For data entry type processes, frequently the output message is only an acknowledgement of processing (ACK = successful, NACK = unsuccessful). For some processes, no response is required.
5. Complete the 'Action type' column.

The action type summarizes the type of processing for designers to determine possible implementation consolidation of activities, or to decide on further allocation of processing to hardware, software, or firmware.

6. Define the return object column.

This column usually refers to the calling object which is ordinarily the object to which control returns, but some nested subprocess might take place. When subprocessing occurs, the return object column identifies the next object entered to help other software engineers understand the logic flow.

Completeness and correctness review of the message list is done to ensure that each process-object pair has an associated message in the table and that the calling/return objects are correct.

ABC Video Example of Message List

First, we make a table with the above headings. Then, referring to the process list that we used to draw the Booch diagram, we list all object process combinations. The objects from that list are listed in the 'Called object' column. Make sure that all process-object pairs have one entry in the table.

Next, we decide both the 'Calling object' from the Booch diagram and the 'Input message' for each entry (Table 12-8 shows the completed list). Then the 'Output message' is completed for each entry. As the output message is complete, we complete each line with the 'Action' and 'Return Object' definitions.

Table 12-8 shows the message list for ABC's application. It reflects the consolidated data objects, the messages decided during the development of the Booch diagram, and the details of information that must be provided for each object-process. Notice that many processes are called from within an object itself. This localizing of processing is desirable to simplify interobject communication and ensure information hiding, but it also can encourage development of nonobject-oriented designs. Make sure that each message contains all, and only, the information required to perform the process. Make sure that each message returns only the information required by the client object.

Develop Process Diagram

Guidelines for Developing the Process Diagram

A process diagram depicts the hardware configuration and the allocation of processes to processor

TABLE 12-8 Message List for ABC Video Rental Processing

Calling Object	Called Object	Input Message	Output Message	Action Type	Return Object
Human Interface	Customer	Customer Information	CustomerPhone	Create	Human Interface
Human Interface	Video Inventory	Video Information	VideoId, # BarCode Videos Created	Create	Human Interface
Schedule	Schedule	Application Id	Queue Address	Execute Init Appl	Schedule
Schedule	Schedule	UserId	Memory Address or Logoff	Execute Init Session	Schedule
Schedule	Schedule	Session Id, Menu Selection for Rental	None or Quit Session	Execute Init Session	Human Interface
Human Interface	Human Interface	No data (Initiate Request)	Trans Request Data Memory Address	Enter Request	TempTrans
Human Interface	TempTrans	Trans Request data	Data access key	Create TempTrans	UserView1 or Customer
TempTrans	Customer	Data access key	Customer Info	Read	TempTrans
TempTrans	UserView1	Data access key	Customer, VideoOnRental, VideoInventory, BarCodeVideo	Read	TempTrans
Customer or UserView1	TempTrans	TempTrans Info	TempTrans	Format	TempTrans
TempTrans	TempTrans	Memory Location, VideoOnRental, Rent/Return Date	Ack	Compute Late Fees	TempTrans
TempTrans	TempTrans	Memory location (Amounts Due) and End of rentals/returns when present	Ack	Compute Total Amount Due	TempTrans
TempTrans	Human Interface	TempTrans Info and End of rentals/returns when present		Display	Human Interface

TABLE 12-8 Message List for ABC Video Rental Processing (*Continued*)

Calling Object	Called Object	Input Message	Output Message	Action Type	Return Object
Human Interface	Human Interface	No data (Execute Request)	Prompt BarCode or End of Rentals/ Returns	Prompt	TempTrans
Human Interface	TempTrans	BarCode (Rental) or End of rental	None	Format	UserView2 or TempTrans
Human Interface	TempTrans	BarCode (Return) or End of return	None	Format	TempTrans
Temp Trans	User View2	Bar Code (new rental)	Video Inventory, BarCodeVideo	Read	TempTrans
UserView2	TempTrans	TempTrans Info	TempTrans	Format	Human Interface TempTrans
Human Interface	Human Interface	End of Rentals/ Returns	Payment Amount	Data Entry	TempTrans
Human Interface	TempTrans	Payment Amount	Change or Payment Due	Compute Change	Human Interface
Temp Trans	Human Interface	Change or Payment Due	End of Trans	Display	TempTrans
Human Interface	Temp Trans	End of Trans	None	Change BarCode Status	User View1
Temp Trans	User View1	Video on Rental Information	Ack	Rewrite	TempTrans
Temp Trans	User View1	Video on Rental Information	Ack	Write	TempTrans
Temp Trans	Hardware Services	TempTrans	None	Print	Schedule
Hardware Services	Schedule	Trans Id		Terminate Trans	Schedule
Schedule	Schedule	Session Id		Terminate Session	Schedule
Schedule	Schedule	Appl Id		Terminate Appl.	System

platforms in a distributed environment. There are two types of icons used in the diagram: processor and device. A **processor** is any intelligent device that performs data, presentation (i.e., monitor display), or application work. A **device** is any dumb device that is part of the hardware configuration supporting application work. Processors are shown on the diagram as a shadowed cube; devices are shown as transparent cubes (see Figure 12-21). This diagram is a crude equivalent of a system flowchart used before process methods were developed. It is crude because devices and processors are all treated as the same, the only immediate visual knowledge the user gets is the configuration size and the extent to which intelligent processors are used.

The methodology assumes that hardware configuration decisions are not part of the SE task and that the hardware decisions are known. Similarly, there are no guidelines for allocating processes to processors. This is an artifact of the development of OO in a defense environment in which the application developers are working from specifications developed by government employees in another city. In the absence of guidelines from the methodology, we can borrow the distribution decision techniques from information engineering and apply them to this decision. In any case, the processes are listed in small print next to the processor in which they will operate.

One shadow cube is drawn for each processor. Individual processes are allocated to each processor. Lines are drawn to show communications capabilities between the *processes*, not between the processors (i.e., the processors are assumed to be networked whether or not the application processes communicate). Only one line per set of processors is drawn, since the details of messages are documented elsewhere. The lines only have directional pointers to show one-way communication.

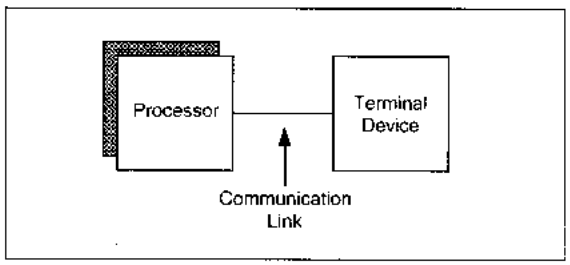


FIGURE 12-21 Process Diagram Icons

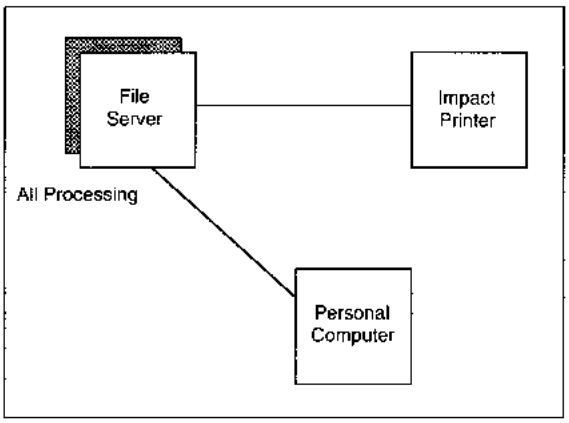


FIGURE 12-22 First-Cut ABC Process Diagram

Next, for each processor, draw the terminals, printers, disk drives, and other peripheral devices that are attached to it. If there are more than one disk drive in the configuration, a list of the classes, class/objects, and objects is made near each drive that will contain data used by the application.

Finally, the diagram is compared to the message list to ensure that all messages are accommodated in the diagram and accurately depict communications between processes. The Booch diagram or the message list can also be used to validate the accuracy and completeness of processes allocated to processors, and of the data allocated to storage devices.

ABC Video Process Diagram

The most simple form of ABC's process diagram shows the file server as the processor and the PCs and printers as terminal devices (see Figure 12-22). This allocation of work is a problem because it does not take advantage of PC intelligence and, therefore, is suboptimal in terms of benefits to be gained from using PCs. Having said this, the allocation is constrained by the software environment. If SQL sup-

ports multilocation processing, then the comment stands. If SQL does not support multilocation processing, then the figure is complete. As it is currently, SQL does not support multilocation processing although it does support distribution of databases.

An alternative process distribution is shown in Figure 12-23. Even with SQL, we could distribute editing, the hardware management functions, payment and change processing, and printing of the rental copy to the local PCs. This is a more complex application because the multiple sites now require synchronization and intraprocessor scheduling in order to coordinate their work, but, if bottlenecks show up in a prototype of the first-cut process distribution, this is a likely candidate for the second iteration of design and prototyping. As it is, we select the simple design because it is significantly easier to implement and maintain, having no synchronization overhead. If it works and is robust to additional users, the first prototype will be completed and placed into production.

Develop Package Specifications and Prototype

Guidelines for Package Specifications and Prototyping

At this point in the design, the functions to be performed are translated into package specifications for translation into program code. A **package** is an encapsulated definition that contains both data and process specifications that define an execute unit. The data might be defined in the form of a class, class/object, or object, with specific attributes and identification. There may be one or more process in a package; they result in individual module specifications and are independently executed under the control of service objects.

Packages have both public and private parts which are specified. The **public package part** identifies the data and processes to the application without any indication of how they are physically implemented. The **private package part** defines the

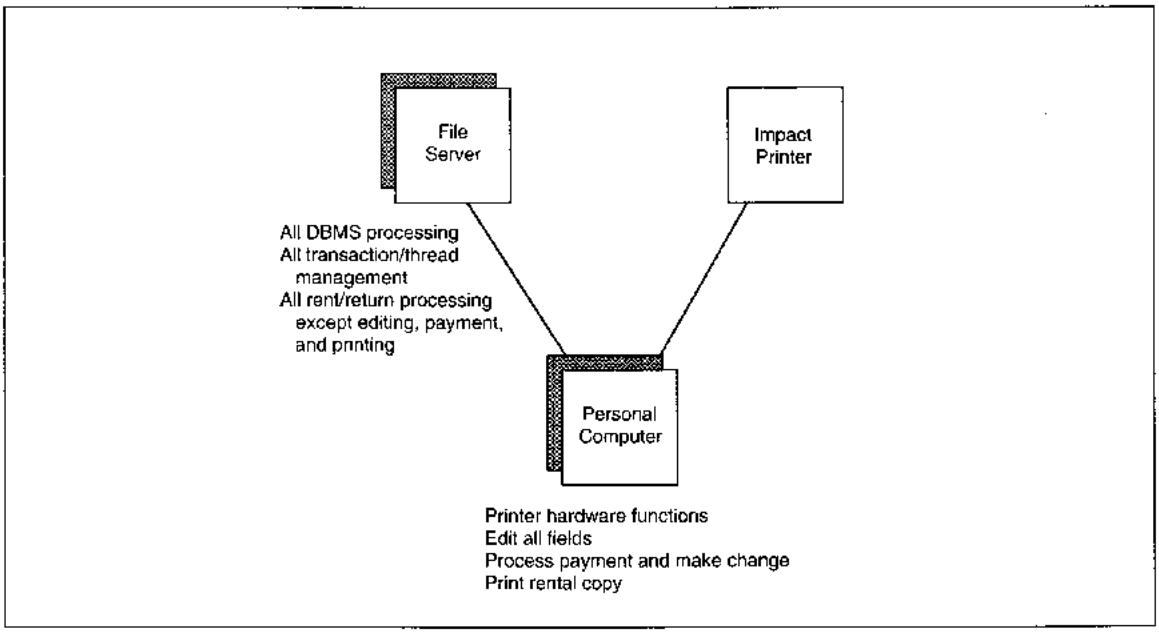


FIGURE 12-23 Alternative ABC Process Diagram

physical implementation. If there are polymorphic definitions of a function, each version of the function is defined separately, and the control mechanism for interpreting the message and activating the appropriate function is defined. Service objects should be used for this interpretation and activation if at all possible.

The steps to package specification are:

1. Review the diagram/list set.
2. Redraw a subset of Booch diagrams, one per processor in the process diagram, to depict objects and processes by processor.
3. Document packages.
4. Design physical database if not already designed.
5. Develop pseudocode specifications for all processes and messaging handling routines.

ABC Video package specifications are not created for this step as it is beyond the scope of this text.

What We Know and Don't Know from OOA and OOD

Object orientation, based on the contents of tables and diagrams, provides a detailed, reasonably complete view of an application. Exceptions to this view are human interface design and specific attention to database, input, and output design. Object-oriented design is distinguished by three characteristics: detail, all potential environments are accommodated, and the need for an object-oriented implementation environment to obtain the payoff from the exercise.

The extensive detail generated in object-oriented design leads directly to module specification which should be straightforward since the definition of process details, the class/object data, constraints, and message communications are all completely defined.

Object orientation, as seen by the exercise in the chapter, can accommodate even nonobject-oriented environments. The benefit of OOD's ability to accommodate any application environment is that, for on-line, object application environments, the methodology *does* lead to information hiding, minimal coupling, and maximal cohesion by virtue of the thinking processes. Object orientation requires good understanding of operating system concepts, object thinking, and interactions between services and applications. The design process, as the chapter appendix shows, requires iteration and prototyping to get required levels of detail and to ensure efficient processing of message traffic. Most important, object thinking IS NOT the same as entity thinking or as process and data methodology thinking. Object orientation requires a paradigm shift to be done correctly.

Object orientation is not very object-oriented in an SQL implementation environment. The choice of SQL changes the entire design from what it would be in an object environment to be object-based. Like COBOL, the methodology can be made to do anything. Is this the *best* use of OOD? Not in my opinion. Unless an application is at least on-line and will be in an object-oriented environment, the work required for object-oriented design is not worth the effort. Especially with a fourth-generation DBMS, like SQL, the undesign that must be done wastes tremendous time and could result in a worse design than use of some other methodology. While this compromise is acceptable for a small, on-line application such as ABC, it would not be acceptable for applications with real-time or more complex processing requirements. Much of the effort to develop an object-oriented design is wasted when the implementation environment is not object-oriented. Therefore, the choice of methodology should be driven by the expected implementation environment.

Automated Support Tools for Object-Oriented Design

There are a vast number of object-oriented CASE tools that have all come on the market in the last few years. Some are more complete in life cycle coverage than others. Some environments, such as 001 Tool Suite, cover most of a development life cycle,

TABLE 12-9 Automated Support Tools for Object-Oriented Design

Product	Company	Technique
001 Tool Suite	Hamilton Technologies, Inc.	Full life cycle multiuser OOA, OOD, and code generation tool for C or Ada
Actor	Symantec Cupertino, CA	OOD environment for client/server applications. Links to C and SQL databases.
Aide-De-Camp	Software Maintenance and Development Systems Concord, MA	Configuration management software with support for OO languages.
BOCS	Berard Software Engineering, Inc.	Berard object and class specification
C/Spot/Run	Procase, Corp. Santa Clara, CA	Interactive, GUI environment for C language development on Sun, HP, and Apollo hardware
Design/1X0, Design/IDEF, Design/OA	Meta Software Corp.	Data and behavior modeling expressed in OO C-language tool
DSEE, HP/Softbench	Apollo/Hewlett-Packard Palo Alto, CA	Integrated CASE Product Supporting OO Analysis
Excelerator	Index Tech. Cambridge, MA	State-transition diagram Matrix graph (RTS)
IPSYS OOA/RD Tool Suite	IPSYS Software	Shlaer-Mellor OOA and Recursive Design
Object View	KnowledgeWare Atlanta, GA	Application prototyping software using 4GL or SQL code
Object Vision	Borland International Scotts Valley, CA	Visual application development system

(Table continues on next page)

in this case, from analysis through code generation. Some tools, such as ObjectView, are more object-based than object-oriented. Some, like Software Through Pictures, try to shield the user from code altogether by sophisticated graphics that generate objects for that environment. Their existence attests to the object revolution that is beginning to be felt in business organizations.

SUMMARY

Object-oriented design (OOD) requires detailed development of all required functionality in the operating system and how it interacts with an application. In this chapter we developed the seven steps to object-oriented design, linking them to the tables developed during object-oriented analysis. First, the

TABLE 12-9 Automated Support Tools for Object-Oriented Design (*Continued*)

Product	Company	Technique
ObjectMaker	Mark V Systems	Full life cycle structured analysis using Ward-Mellor extensions tool with code generation for Ada, C, and C++
OMTool, OMT/SQL	GE Advanced Concepts Center	OOA and OOD with schema compilation compatible with Oracle, Ingres, and Sybase
ProMod	Promod, Inc. Lake Forest, CA	Control flow diagram State-transition diagram Module networks Function networks
Smalltalk/V	Digitalk Los Angeles, CA	32-bit Smalltalk for OS/2 hardware
Software Backplane Cohesion	Atherton Technology/Digital Equipment Corporation Maynard, MA	Integrated CASE Product Supporting OO Analysis
Software Thru Pictures	Interactive Dev. Env. San Francisco, CA	Control flow State-transition diagram
Teamwork	CADRE Tech. Inc. Providence, RI	DFD Control flow State-transition diagram Process activation table
Telon	Pansophic Systems, Inc. Lisle, IL	State-transition diagram Code generation
Treed4C, Tree4Fortran, Tree4Pascal, TreeSoft1	1 Software Engineering Camarillo, CA	Program code reengineering products for Sun hardware
Visible Analyst	Visible Systems Corp. Newton, MA	State-transition diagram
vs Designer	Visual Software Inc. Santa Clara, CA	Booch diagram

objects are allocated to four subdomains: human, hardware, software, and data. The split of processing into these four areas accommodates the use of, for instance, firmware, distributed computing, DBMSs, and intelligent interfaces in what would otherwise be a monolithic development of an application.

The second step of OOD is the development of time-event diagrams for all processes and all objects. The purpose of a time-event diagram is to allow the analysts to identify independent, sequential, concurrent, independent, and concurrent, dependent processes. Usually, several alternative ways of looking at the timing of processes emerge from this analysis, one of which is selected for development.

Once the types of process are defined, their service object needs are identified. Service objects closely parallel operations performed by an operat-

ing system (OS). OSs have five main functions to manage: memory, job, task, I/O, and secondary storage. The memory, I/O, and secondary storage management functions are directly translatable into object thinking. Job management functions are analogous to those performed at the control level for an entire application and/or user. Job management is more appropriately called session, or user, management in object terms. Similarly, tasks are individual steps of a job and are analogous to transaction-related modules when thinking in objects. Therefore, the term used here for task functions is transaction management. Each type of management function requires its own type of processing and the processes selected are particular to the application and implementation environment.

The fourth step of OOD is to develop a Booch diagram to summarize the objects—both application and service—and their interactions. Booch recommends a 3" x 5" approach for which each object and its processes are shown as a package on a 3" x 5" index card. The set of cards is moved into different configurations and message connections are drawn. The purpose of this exercise is to choose a message-passing scheme that minimizes the potential for bottlenecks and that provides information hiding and minimal coupling. The final configuration selected is documented for the application.

The message connections decided during design of the Booch diagram are elaborated in the next step, which is to define message communications. Each called object and its calling object, input message, output message, action type, and return object are identified.

At a higher level of abstraction, the next step is to develop a process diagram that shows the distribution of functionality and equipment for the application being developed. A process diagram depicts processors, for example, computers, and devices, that is, limited-intelligence equipment such as a disk drive. All equipment and their interconnections are identified. Multiprocessor interconnections show allowable message movement throughout a network, while the device connections show hardware configuration. The functions performed at each processor in a multiprocessor configuration are also on the diagram.

The last step of OOD is to develop package, or module, specifications for programming. The information from the various tables and graphics is rearranged to show the relevant information for each particular module. Also, details of each module's logic, if not already documented in a dictionary, are defined in the package specifications.

OOD CASE tools come in several varieties: object-oriented life-cycle development, object-oriented design without code support, object-oriented coding without design support, or object-based thinking through adaptation of existing methods.

REFERENCES

Booch, Grady, *Software Engineering with Ada*, second ed. Menlo Park, CA: Benjamin/Cummings Publishing Co., Inc., 1987.

Booch, Grady, *Object Oriented Design with Applications*. Redwood City, CA: Benjamin/Cummings Publishing Co., Inc., 1991.

Coad, Peter, and Edward Yourdon, *Object-Oriented Analysis*, second ed. Englewood Cliffs, NJ: Prentice-Hall, 1990.

Coad, Peter, and Edward Yourdon, *Object-Oriented Design*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

Graham, Ian, *Object-Oriented Methods*. Reading, MA: Addison-Wesley Publishing Co., 1992.

LaFore, Robert, *Object-Oriented Programming in Turbo C++*. Emeryville, CA: The Waite Group Press, 1991.

Peterson, J., and A. Silbershatz. *Operating System Concepts*. Reading, MA: Addison-Wesley Publishing Company, 1983.

Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

KEY TERMS

3" x 5" approach
binary message
binding
Booch diagram
client object
concurrent processes
data subdomain
device
dialogue
dynamic binding
hardware subdomain
human subdomain
keyword message
logical process flow

message
message traffic
module
module structure diagram
multitasking
multitasking objects
object-based
package specification
package
polymorphism
private interface
private package part
problem-domain objects
process diagram
processor
pseudo-dynamic binding
public interface
public package part
round-trip gestalt
scheduling
scheduling objects
server object
service objects
software subdomain
static binding
supervisor call (SVC)
synchronizing
synchronizing objects
thread of control
time events
time-event diagram
unary message
utility objects

EXERCISES

1. Continue with the exercise begun in Chapter 11. Design the application for Eagle Rock Golf League.
2. Design all *Customer* processing for ABC's application. Why is it different from that of *VideoInventory*? If we add multiple members to a household, how does that change the design?
3. Compare the SQL and C++ designs for ABC rental processing. If there are bottlenecks in processing for the two designs, where are they likely to be? How might they be removed? Which design gives you better control over the computer and its resources?

STUDY QUESTIONS

1. Define the following terms:
   message
   object
   polymorphism
   problem domain
   round-trip gestalt
   service objects
   synchronizing
   thread of control
   time-event diagram
2. Define the four subdomains and the type of objects found in each.
3. What benefits accrue from the allocation of processes to hardware, software, database, and human subdomains?
4. Why are service objects needed? When are they needed and when not?
5. What is multitasking? Why is it important in application design?
6. What is the purpose of a Booch diagram?
7. List and compare three types of message formats.
8. What is the purpose of a process diagram?
9. Describe client/server computing and how it relates to object orientation.
10. What is binding? What types of binding are possible? How do you know what type is used in an application you are developing?
11. Describe an example of polymorphism.
12. What are some of the problems associated with allocation of processes to subdomains?
13. What does the configuration ⊤ on a time-event diagram mean?
14. Describe how to interpret a time-event diagram.
15. Describe how operating systems relate to service objects.
16. Describe the kinds of activities managed by the task manager.
17. What are the control levels in object orientation that are analogous to job and task management in an operating system? Distinguish between them and the tasks they manage.
18. What is memory management and why is it necessary?
19. List the steps to defining service objects. Describe some of the problems related to this activity.
20. What is the purpose of a Booch diagram?
21. Describe the steps to developing a Booch diagram. What information is shown on the diagram?
22. What is a package? What are its contents on a Booch diagram? What are its contents in a working application?
23. Booch recommends the use of 3" × 5" cards to create and 'play' with the Booch diagram contents. What is the playing for? Why are 3" × 5" cards helpful to that process?
24. List three design goals of messages. Create an example of message passing in an object-oriented application. Describe different types

of messages to illustrate good and poor message designs.
25. What information is placed in the message table to document message traffic in an application?
26. Why is message definition a difficult activity?
27. Describe the icons used in a process diagram and their purpose.
28. How many Booch and process diagrams are drawn for an application?
29. Describe the validation processes used throughout an object-oriented design process. Why is each validation step where it is in the process and what is the purpose of each validation?
30. Discuss the statement: "There is no such thing as a one-shot object-oriented design."
31. What information is provided for package specification documentation? How do you decide what is public and what is private information to an object?
32. What is the role of prototyping in object orientation?

★ Extra-Credit Questions

1. Research queue or stack management. Write a two-page paper to describe the functions of that type of management. Then, design the object-oriented class/objects and processing routines that would accomplish these functions.
2. Booch discusses primitive processes in detail and names several different types of primitive processes. Research these types of processes and discuss their importance to object-oriented design. How important is it to have a name for each type of *thing* in a design?

Appendix: Unix/ C++ Design of ABC Video

Although the Chapter 12 presentation of ABC Video's design began as object-oriented, it ended as a hybrid: part-object and part-not, because of the implementation environment. This appendix is the same design with a discussion of the decisions and alternatives from a purely object-oriented perspective. Chapter 12 presented a consistent discussion of the implementation throughout the text and shows what happens when you deobjectify the application to fit a particular language environment. This appendix, then, gives you a basis for contrasting what would happen if you designed a purely object-oriented application. Each stage of the process is presented with enough comment for you to see the differences between the hybrid and object designs. Package specifications and a prototype are still beyond the scope of this discussion, but we present a partial package specification so you can contrast the levels of detail for OOD to the other methodologies.

A few terminology differences exist with the Unix, C++ environment and we start with them. Class structure is similar in C++ to the discussion in the chapter. Data in C++ is defined by *structures*. A structure that contains both data and functions is called a *class*. Classes were defined in the chapter as having public and private parts. In C++ classes have public, private, and protected parts. The public part is that part accessible by the rest of the system. The private part is not directly accessible by any other classes. These two definitions have not changed from the chapter. A **protected** part specifies what may be inherited, that is, processes that are accessible by member processes in its own class or in any class derived from its own. A **derived class** is one that has multiple inheritance and is made up of its own, and its inherited, data and functions. Class inheritance is implemented by having processes that have a *protected* status. Thus, in C++, the manner of implementing inheritance is to provide the protected part of an object and to distinguish inheriting objects by calling them derived classes.

The term process refers to **functions** in C++. Functions can be part of a class (i.e., a member) and restricted in use, or they can be stand-alone entities that are independent of a class. At least one independent function, **main()**, is required to initiate processing of a program or application. Many functions are provided in a library of reusable functions that are link-edited to compiled code for execution. We will not spend much effort on functions since they are most evident at the code level.

Individual language operators are analogous to other languages. Polymorphism is termed operator **overloading** but the meaning is the same. **Virtual functions** are the method used to provide run-time binding for polymorphic functions. Other function types beyond the typical ones associated with classes include **friend functions**, that have *read only* access to the private data of a class, and **static functions**, that operate on the class level rather than at the object (i.e., instance) level. Borland's Turbo C++ provides an entire set of classes with functions and inheritance as the basis for developing applications. The 'container' classes, for instance, include several types of arrays, associations, hash tables, lists, stacks, and queues. The container classes are important because they provide a means for implementing service objects. Next, we discuss the object-oriented design (OOD) activities.

Allocate Objects to Subdomains

In object-oriented analysis (OOA), we defined classes, class/objects, and superset classes needed to properly define all of the interrelationships among objects in the application. This diagram and the table matching processes to their objects are the basis for this activity. The allocation in Table 12-3 has no change here (see Table 12-A1).

In allocating the data handling functions to the data subdomain in C++, we commit to designing generics to handle all files. This means that we need a new object for DB actions. Also, there will be no collapsing of data objects as in SQL. Object-access control will be implemented as a superset of functions to mirror the object relationships. To implement the generics, a fixed message type that accommodates all of the processing for all of the data objects is required. Such a message's minimal contents are: From-Object, To-Object, Action, Object, Return-code, Physical-Location-Key, Length-of-Data, and Data.

While the subdomain allocations do not change, the handling of them does. Once functions are allocated to a DBMS, all developers need to know all allowable interactions. Those interactions must be *defined* and *designed* manually when no DBMS is used. A partial list of functions required includes:

Locate Data (transform key to physical location)
Get Data (may include a prechange write to a log for recovery)
Rewrite (may include a postchange write to a log for recovery)
Write (may include a postchange write to a log for recovery)
Delete (may include a postchange write to a log for recovery)
Space Management
Queue Management (including service requests and service responses)
Backout Management
Commit Management
Lock Management
Access Control Management
Error processing for such problems as data not found, out of space, hardware error, or unsuccessful read, write, rewrite, or delete.

These functions can be defined and incorporated into documentation at subdomain allocation time or during service object definition.

The human interface definition is also going to be different. In the main text of this chapter we designed the system for a 4GL, in which a screen is *painted* and the programmer only needs to know the fields, their format, and desired characteristics. The 4GL software manages all of the formatting and setting of field attributes. In a lower level language, such as C++, screen format, line, starting position, length, field attributes (e.g., blink, reverse video, or color), and field contents are all managed by the programmer and, therefore, require design.

Another choice we make is to have full-screen, line-at-a-time, field-at-a-time, or character-at-a-time interactions. Selection of input method is application specific. In ABC's case, we decide that using a method that will not slow down users the least during peak periods is best. Since actual data entry is limited to *CustomerPhone*, *VideoBarCode*, and *money amounts*, for rental processing, and since rental processing is the most used function, we choose field-at-a-time entry. If the application had thousands of users and millions of transactions each day, we might have field-level entry for rent/return processing and screen entry for customer and video

TABLE 12-A1 Process Subdomain Assignments

	Subdomain			
Process Name	Data	Hardware	Process	Human
EnterCustPhone				X
ReadCust	X			
CreateTempTrans			X	
RetrieveVOR	X			
DisplayTempTrans				X
EnterBarCode				X
RetrieveInventory	X			
Display Inventory				X
ComputeTempTransTotal			X	
EnterPayAmt				X
ComputeChange			X	
DisplayChange				X
UpdateInventory	X			
WriteVOR	X			
PrintTempTrans		X		
EnterBarCode				X
RetrieveVOR	X			
DisplayTempTrans				X
AddRetDateTempTransVOR			X	
Add1toVInv			X	
UpdateInventory	X			
ComputeLateFees			X	
WriteVOR	X			
EnterCustomer				X
CreateCustomer	X			
EnterVideoInventory				X
CreateVideoInventory	X			

maintenance, because they are more data-entry intensive activities. Whichever input 'chunking' method is chosen, we must intercept start and stop characters from the keyboard and bar code reader to synchronize processing between the input devices and the computer.

With field-level input, we could choose field-level interactions, having local, PC-based intelligence

simulating a 4GL that checks alphabetic/numeric contents and beeps on errors. This greatly complicates the application and is decided against. At some future date, if the number of users begins to tax the file server, we could revisit this decision to speed processing by off-loading work from the server.

Draw a Time-Event Diagram

The time-event diagram also does not change and is presented here as Figure 12-A1. Now we will pay more attention to the potential for concurrency, because we must be able to prove the processing and that implies monitoring of the success of all write, rewrite, and print actions.

The choices for concurrent processing all relate to data I/O, and the consequences of deciding for concurrency must be considered. First, consider consequences of concurrency if we opt for read/write concurrency. At the hardware level, the affected databases must be on separate buses (on a PC) or channels (on a mainframe) to ensure that the processes are not contending for the same hardware disk access time. Second, management and synchronizing modules to reunite multiprocesses within a thread and to verify processing are required. This implies a need for queues for each process and for each thread. For each process we need process ID, thread ID, and return code. For each thread, we need all concurrent processes' IDs and return codes from processing. Side effects of potential errors must be considered. For instance, if *WriteVideoOnRental*, *RewriteVideoOnRental*, *PrintReceipt*, and *WriteHistory* objects are all active at the same time, we need to decide acceptable combinations of successful/unsuccessful processing *and* actions taken for each possible combination.

Concurrency decisions should be based on business constraints and needs for processing or response time. There should be some attempt to compute how long a transaction will take and to determine response time. For example, ABC rental transactions have an approximate processing time of 8.6 seconds (8566 ms; see Table 12-A2) during nonpeak time and about 11 seconds during peak processing times. From this table, which the SEs generate, we see that input and output from the terminal account for 8.1 seconds of the total and actual internal processing is about 506 ms or slightly over one-half second. If the internal time were over two seconds, we would opt for concurrency to minimize the internal strain on processing. With under a half-second processing time, we can continue thinking of sequential processing as we did with the SQL solution. The differences in using SQL versus an object-oriented language are not yet apparent. The major difference so far has been the level of detail of the reasoning process to make concurrency and data-related decisions. This level of detail is similarly lower for the other OOD reasoning processes as well.

Determine Service Objects

In this section, we list the required service object functionality to show the level of detail and complexity required of true object systems, but without much explanation. We will assume that the Unix/C++ environment being developed for ABC will employ reusable code objects for many service functions. 'Free' code is one of the benefits of using consultants who come with their own implementation modules for many functions. We still need to determine which modules are needed, however. Referring back to Table 12-5, ABC is a sequential, multiuser application with needs for scheduling and multitasking management, in addition to I/O, user, transaction, thread of control, memory, startup/shutdown, and data management. Table 12-A3 lists high level service objects required to support ABC's application.

Input/output is straightforward. There are four I/O functions to design: keyboard, bar code reader, display screen, and printer. We assume that all input interactions are from the keyboard or bar code reader, which read slightly differently. The keyboard is read one character at a time until a field is complete. The bar code reader reads the entire code, or field, at once. Thus, we can use polymorphic modules to *GetField* and possibly for other functions as well. Likewise, we assume all output interactions are to the display screen and printer. The basic actions for all four devices is to start, synchronize (abbreviated synch from now on), get/put, wait, or stop.

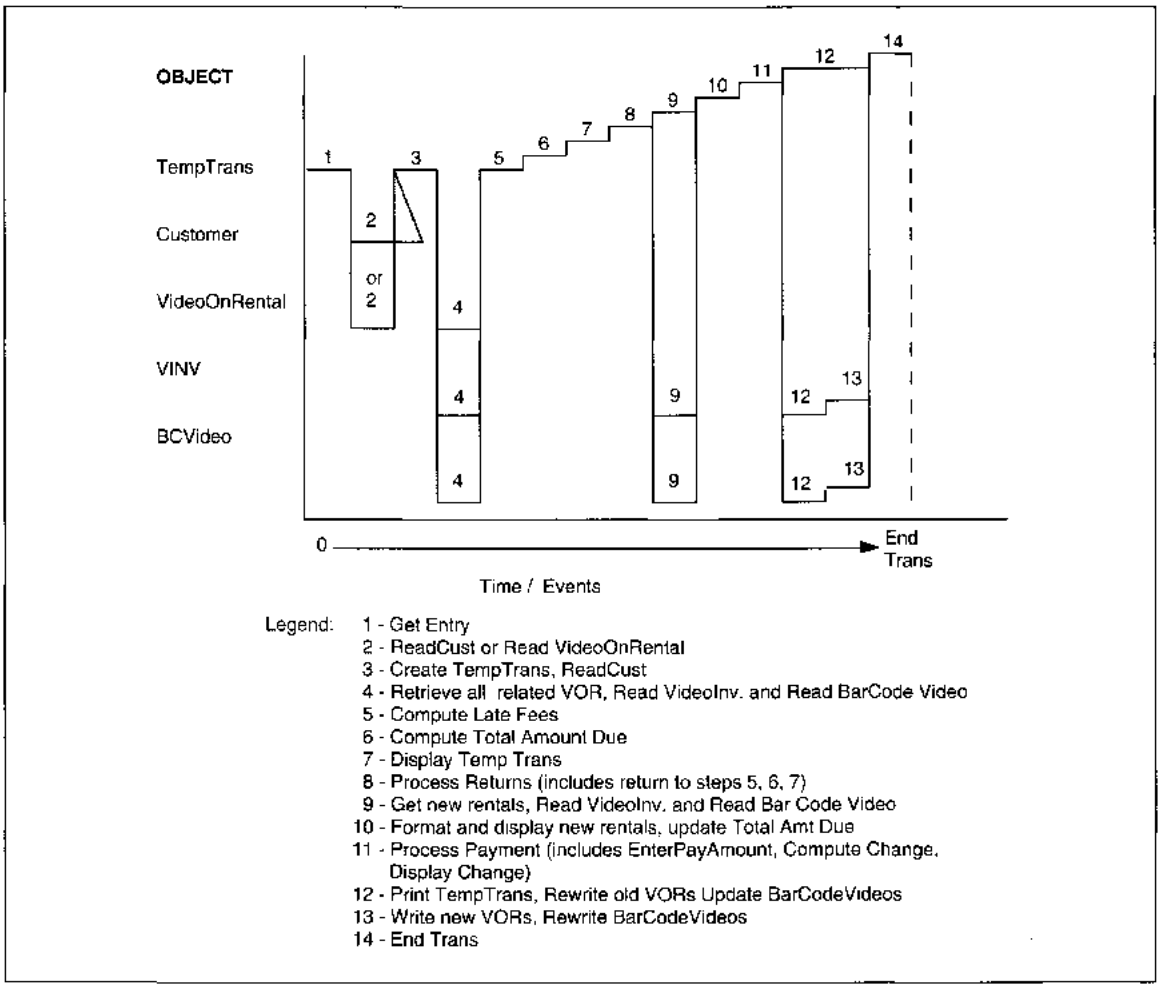


FIGURE 12-A1 ABC Time-Event Diagram

Waiting requires a queue to manage multiple waiting requests.

User routines initiate an application session and verify user access. The 'put' commands all interface to the screen I/O manager, handing off the message to be displayed. Similarly, the 'get' commands all interface with the keyboard or bar code routines of the I/O manager. The purpose of user logon routines is to identify physical terminal address (*TermID*) and user (*UserID*).

The transaction object and its routines manage individual transactions selected from menus. Information is directed to a specific device based on the *TermID* and *UserID* passed from the User routines. For instance, customer *Maintenance* has four transactions: create, delete, update, and retrieve. Job routines then display menus and alter menu contents based on user logon and access codes. As above, 'puts' interface with the screen or printer routines of the I/O manager objects and 'gets' interface with the keyboard or bar code reader routines. The information passed to the command object for use in process control includes *TermID*, *UserID*, and *TransCode*.

Thread of control is handled by a command object and routines which manage atomic processes,

TABLE 12-A2 Rent/Return Transaction Processing Time Estimate

Instruction	Input*	Internal Process	Output	Total
Get	1000			1000
Read (average 3) 30 ms each plus data transfer of 6 ms each		96		96
Compute late fees		30		30
Compute amount due		10		10
Display (average 20 lines, 150 ms/line)			3000	3000
Get Returns (30% of transactions)	1000			1000
Retrieve VOR (average 3)		96		96
Compute late fees and amount due (10 ms each)		30		30
Display 3 lines			450	450
Get Rental (assume one)	1000			1000
Retrieve 3 DBs		96		96
Compute amount due		10		10
Display rental line, amount due line			300	300
Process payment—enter amount	1000			1000
Compute change		10		10
Display new amount due, change			300	300
Print (assumes automating queuing and time to transfer queue address)			10	10
Rewrite (average 3)		96		96
Write (average one)		32		32
Subtotal (nonpeak time)	4000	506	4060	8566
Time in queue (average .33 trans waiting during peak times transaction time)				2855
Total peak processing time				11421

*All times are in milliseconds.

TABLE 12-A3 Service Objects Required for C++ ABC Application

I/O Manager			
Keyboard Processes	Get character until end of field		Put password prompt
	Ready to receive (Sync keyboard)		Get password
	Start keyboard entry		Verify password
	Reset keyboard		Put password error
	Send entry to screen formatter	Transaction object	Put menu
Bar Code Reader	Start reader		Get selection
	Sync reader		Verify selection
	Get bar code		Get memory
	Send bar code to calling routine		Release memory
Display Screen	Identify screen location and type interaction		Set up global user area
			Release global user area
	Format screen protected lines		Call defrag for user area
	Format screen data lines	Thread of control— Command Object	Get memory address of data
	Put keyboard entry in field		Get memory
	Set field attributes		Set status
	Check allowable value		Queue instructions for execution (i.e., call object/process)
	Get error message		Transfer control to TempTrans or Data
	Send entry to calling routine		Enqueue transaction
	Put screen		Dequeue transaction
	Put screen line		Execute instruction
Printer	Sync printer		Check status
	Start print		Create status
	Put lines until end of print		Delete status
	Stop printer		Release memory
	Get print lines until end of print	Memory Manager	Allocate memory
	Wait to print		Deallocate (free) memory
	Store print lines for 60 seconds		Defrag memory (i.e., defragment)
	Queue address, length of print information		Queue memory request
User Object	Put logon prompt		Dequeue memory request
	Get logon		
	Verify logon		
	Put error		

(Table continues on next page)

that is, they supervise execution of code modules. The object reads code into memory, passes one instruction at a time to the CPU for execution, and interfaces to the other manager routines to perform I/O, memory, and data management. The command object uses the fields passed from the transaction object and adds to it the task and task status.

Memory management is designed simply to allocate the maximum amount of space for a transaction to any request. The largest transaction is a rental/return which is estimated to take 13,860 bytes as follows:

Design Element	Bytes
Screen 80 × 22	1,760
Max fields 100 bytes × 10 lines	1,000
Attribute bytes three/field	300
Miscellaneous data area	800
Code	10,000
Total	13,860

TABLE 12-A3 Service Objects Required for C++ ABC Application (*Continued*)

Start/shut	Set up all memory	Read DB
Main()	Initiate managers	Write DB
	Load application code	Rewrite DB
	Allocate transaction code locations	Position DB
	Store application code	Determine physical location
	Get DB indexes	Request Read
	Store DB indexes	Wait read
	Start DBs	Request Write/Rewrite
	Close DBs	Position Index
	Transfer to User	Read Index
Data Manager	Open DB (Open Index, Read Index into memory, Position Index, Open DB files)	Wait write/rewrite
	Close DB (Write Index, Close Index, Release Locks, Backup DB, Backup Indexes, Close DB files)	Check item locks
		Enqueue item lock
		Dequeue item lock
		Wait for item lock

While this over-allocates memory, the alternative, to size memory to each transaction, is more complex. If memory becomes scarce, the change to transaction size allocation can be made. To contrast the amount of memory required, a Customer Create transaction takes approximately 5K memory.

Startup and shutdown could be handled as part of the user object, but a cleaner implementation is to design them as separate. This start/shut object allocates memory, initiates application and DB processing, including bringing all transaction code and DB indexes into memory. In C++ implementation terms, the start/shut object will be the main() routine that initiates ABC processing.

Last, data management could be by file or by function. By file is simpler and easier for novices to maintain, but it also requires much more code and, therefore, more maintenance. Here we will define one set of generic CRUD functions for the data object with each requiring the specific DB name and data. If necessary, polymorphic processes for the CRUD functions can be customized for each database.

After the services objects are developed, they are allocated to the four subdomains of data hardware, software, and human interface as shown in Table 12-A4. Allocation of keyboard and bar code to hardware would be a possible choice. They are left with

TABLE 12-A4 Service Object Allocation

Data	Hardware	Process	Human
Data Manager	I/O—Print	User Manager	I/O—Keyboard, Display, and Bar code reader
		Memory Manager	
		Transaction Manager	
		Command Manager (Thread of Control)	

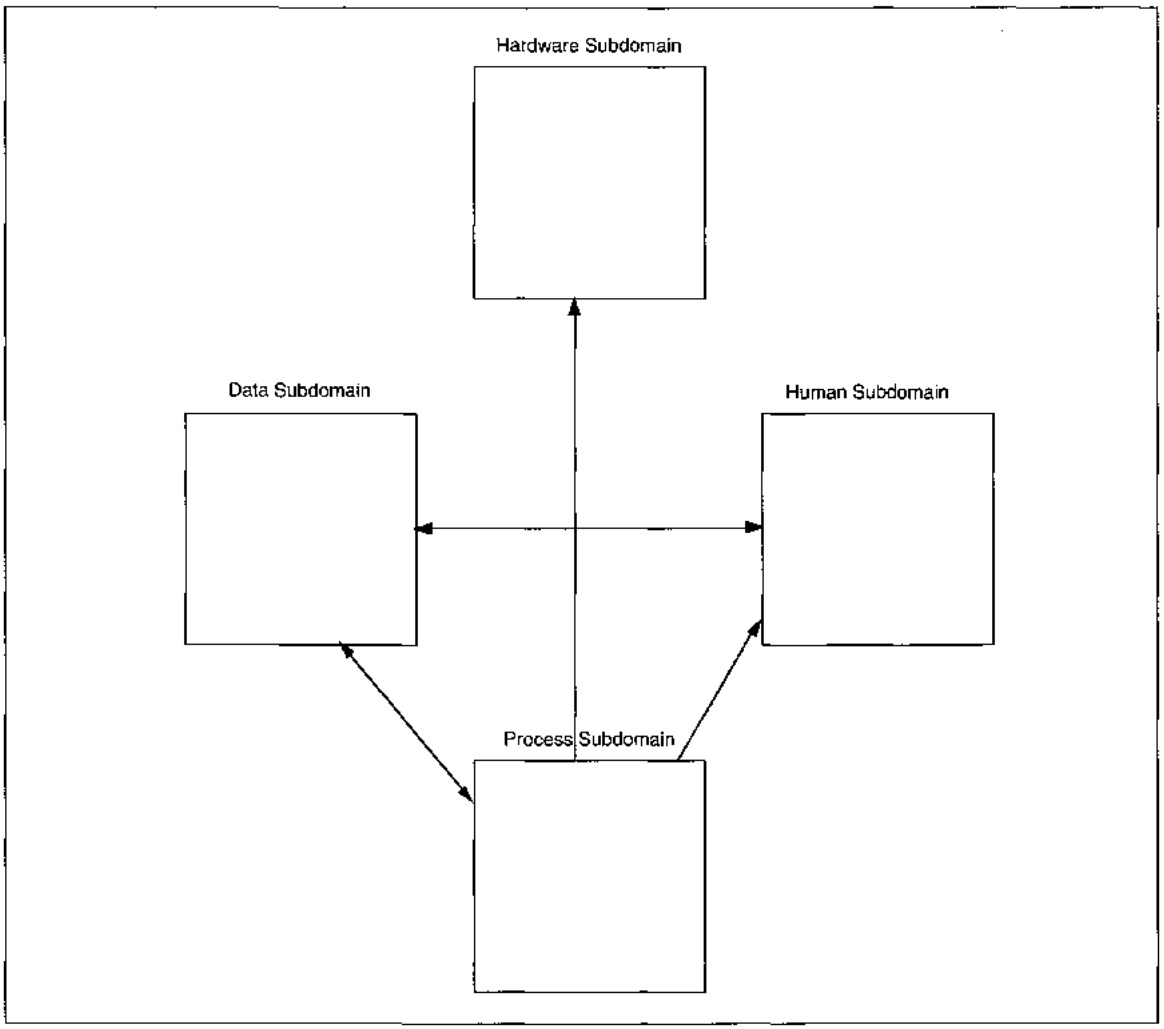


FIGURE 12-A2 Subdomain-Level Booch Diagram

the human interface because they are closely related to the display processes which mirror all of their input. Keeping these processes together reduces the object-switching overhead required to change from one object context to another.

Develop a Booch Diagram

The first Booch diagram in Figure 12-A2 shows the subdomain-level communication. To simplify the communications in the system, based on the subdomain message interchanges, we will define a generic message for use in most communications. The second Booch diagram, shown in Figure 12-A3, is at the object level and is obviously more complex than the SQL solution.

There are several major differences between the SQL and C++ designs. First, the schedule in SQL is a mainline routine that determines the next code to execute and is a centralized controller of the application. That function is performed to some extent by the command manager objects in the C++ design,

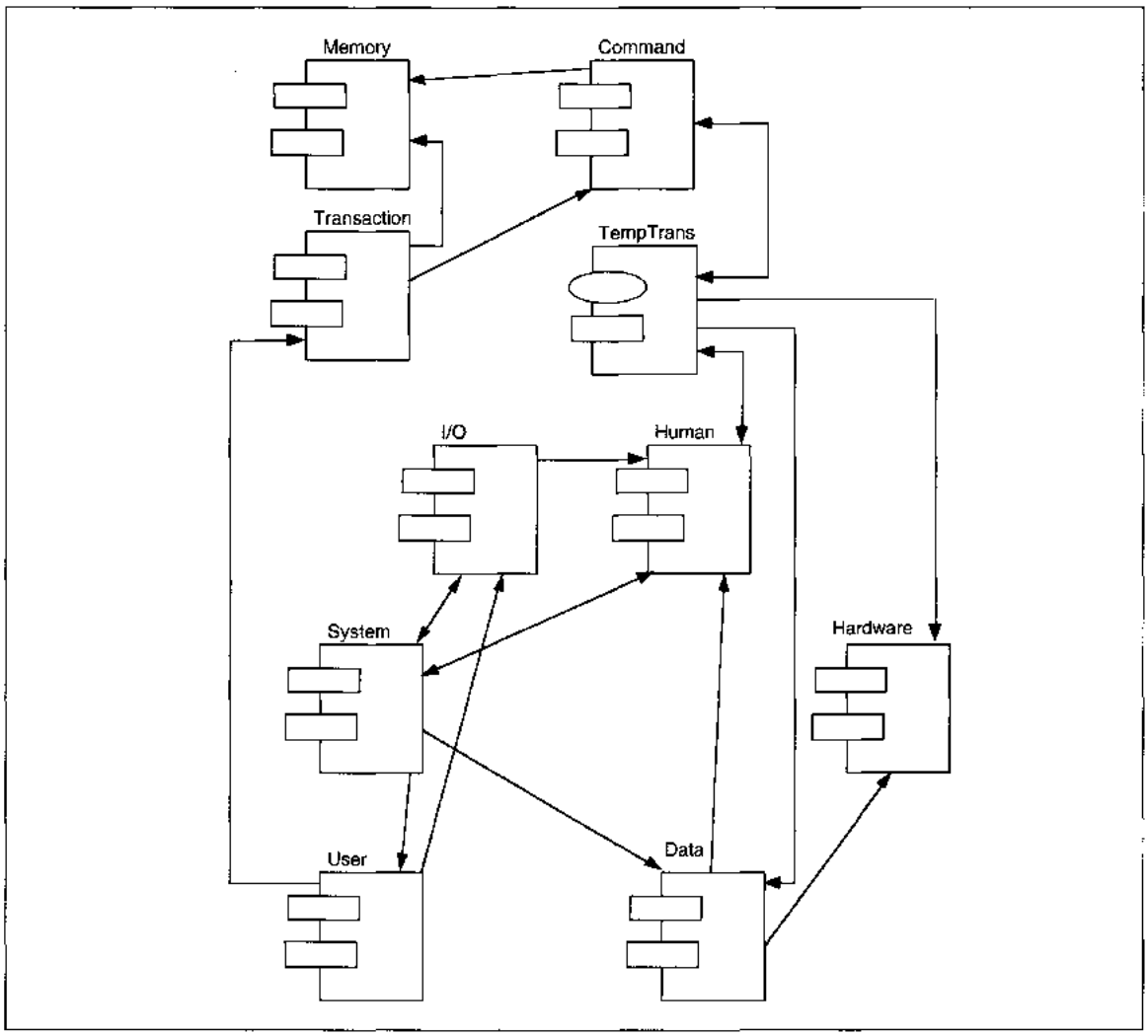


FIGURE 12-A3 Object-Level Booch Diagram

but the scheduler functions are at a lower level and spread over the service objects. At this level, the specific processes are not shown because the diagram would be more complex than necessary. Instead, we have shown the service and data objects only. To implement the application, we would complete that detail.

The design as shown in Figure 12-A3 is still incomplete for the data part of the processing. In Figure 12-A4 the next lower level of detail to show the complexity of the data objects is developed. Based on this diagram, we might decide to denormalize the data to provide minimal accessing of databases during rental processing. For instance, we might replicate all *VideoInventory* information in each *BarCodeVideo* object to eliminate the need to access another object as part of rental processing. Similar denormalization might be done with *Customer* and *VideoOnRental*. Before a prototype could be built, a second design iteration on all objects and complete design of the details is required.

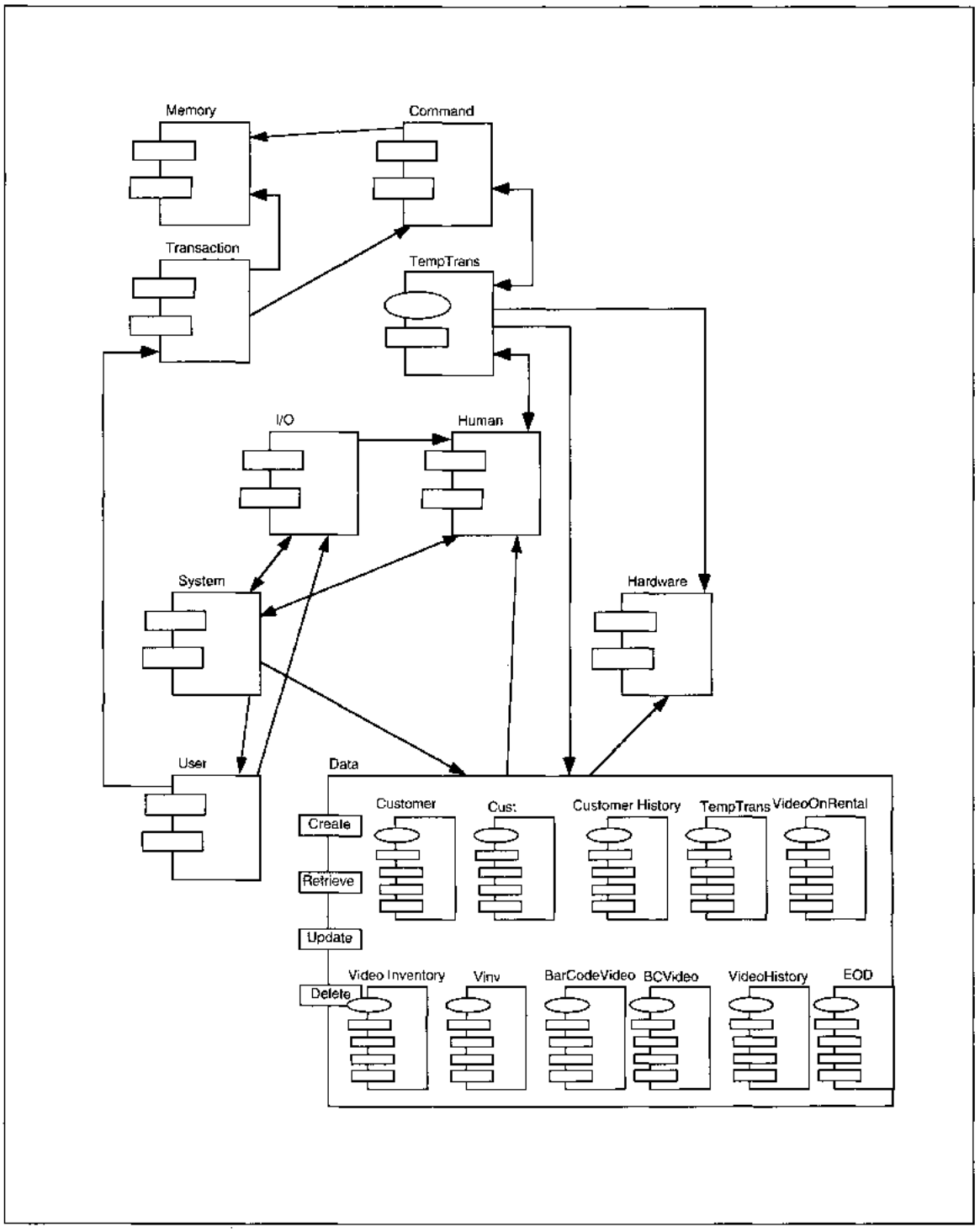


FIGURE 12-A4 Object-Level Booch Diagram with Data-Object Detail

Define Message Communications

The message list is shorter than that of the SQL solution if we use a generic message as described above. The generic message list for the C++ Booch diagram is shown as Table 12-A5. If we do not use a generic message, the number of connections increases from the SQL number of about 30 messages to over 170 messages for C++ as shown in Figure 12-A5, which depicts *all* connections in the Booch diagram, summarizing the processing for Command and I/O manager objects. In Figure 12-A5, the processes with no specific arrows have multiple calling routines and return to the caller. The other routines with arrows are chained as shown.

In the SQL design, the network operating system and SQL shielded the application programmer from most of the complex elements—the service objects. With C++, the increased number of connections also increases the application's complexity. If we cannot use DB user views, there are more data objects on the diagram. If we do not have a sophisticated operating system to monitor execution and physical I/O aspects of the application, the capability must be part of the application. By using generic messages, we reduce the complexity somewhat by reducing object abends for wrong message type and by allowing generic code for message reception and interpretation.

Develop Process Diagram

The process diagram has no changes from Figure 12-22, which is redrawn here as Figure 12-A6.

Develop Package Specifications and Prototype

Package specifications for SQL would be simple compared to those of C++. One package description/program specification is shown below for customer data. The specification identifies public and private parts, plus the processing to be performed. Following the specification is an example of a C++ code module to read the customer file based on a location that is passed to the read module.

Customer Specification

Item:	Description
Name:	Customer
Documentation:	The customer database contains information about legal customers for ABC. All access is through the data manager routines. All data is passed to using routines.
Visibility:	Private
Cardinality:	400–600
Hierarchy:	
Superclass	Customer
Class	Cust
Metaclass	None
Generic parameters:	&custloc &custrec
Interface-Implementation:	
Public:	Only through passed parameters
Protected:	Uses:Customer class Fields = char custphon [10]; char custln [50]; char custfn [25]; char custadd1 [50]; char custadd2 [50]; char custcity [30]; char custstat [2]; char custzip [10]; char cctype [1]; char ccno [17]; date ccexp [8]; date entrydat [8];
Operations:	Add (put) Seek (read) Update (put) Delete
Persistence:	Static

TABLE 12-A5 C++ Design Message List for ABC Rental Processing

Calling Object	Called Object	Input Message	Output Message	Action Type	Return Object
Temp Trans Start/Shut	Data Manager	Task ID, Terminal ID, Thread ID, Database ID, Type Request, Data	Task ID, Terminal ID, Thread ID, Database ID, Type Request, Return Code, Data	CRUD, Open, Close	Caller
Print Term Trans	Hardware-Print	Data Address, Type Print	None	Print	None
Temp Trans, Start/Shut	I/O-Bar Code Reader, I/O-Keyboard	Task ID, Terminal ID, Thread ID, Database ID, Type Request	Task ID, Terminal ID, Thread ID, Database ID, Type Request, Return Code, Data	Input	Caller
Start/Shut, User Mgr, Trans Mgr, Human Interface, Data Mgr	I/O-Display	Task ID, Terminal ID, Thread ID, Database ID, Type Request, Data	ACK or Task ID, Terminal ID, Thread ID, Database ID, Type Request, Return Code	Display	Command
System	Start/Shut	Begin	Non until shut down	Process	User Mgr
Start/Shut	User Mgr	Term Id	I/O-Display—Logon screen request (no message return to caller)	Put Prompt	I/O-Display
Command	Temp Trans	Task ID, Terminal ID, Thread ID, Database ID, Type Request, Data	Depends on next called routine, either Task ID, Terminal ID, Thread ID, Database ID, Type Request, Data or Task ID, Terminal ID, Thread ID, Database ID, Type Request, Return Code, Data	Process	Either Command, Human Mgr, Data Mgr, HW-Printer I/O Mgr

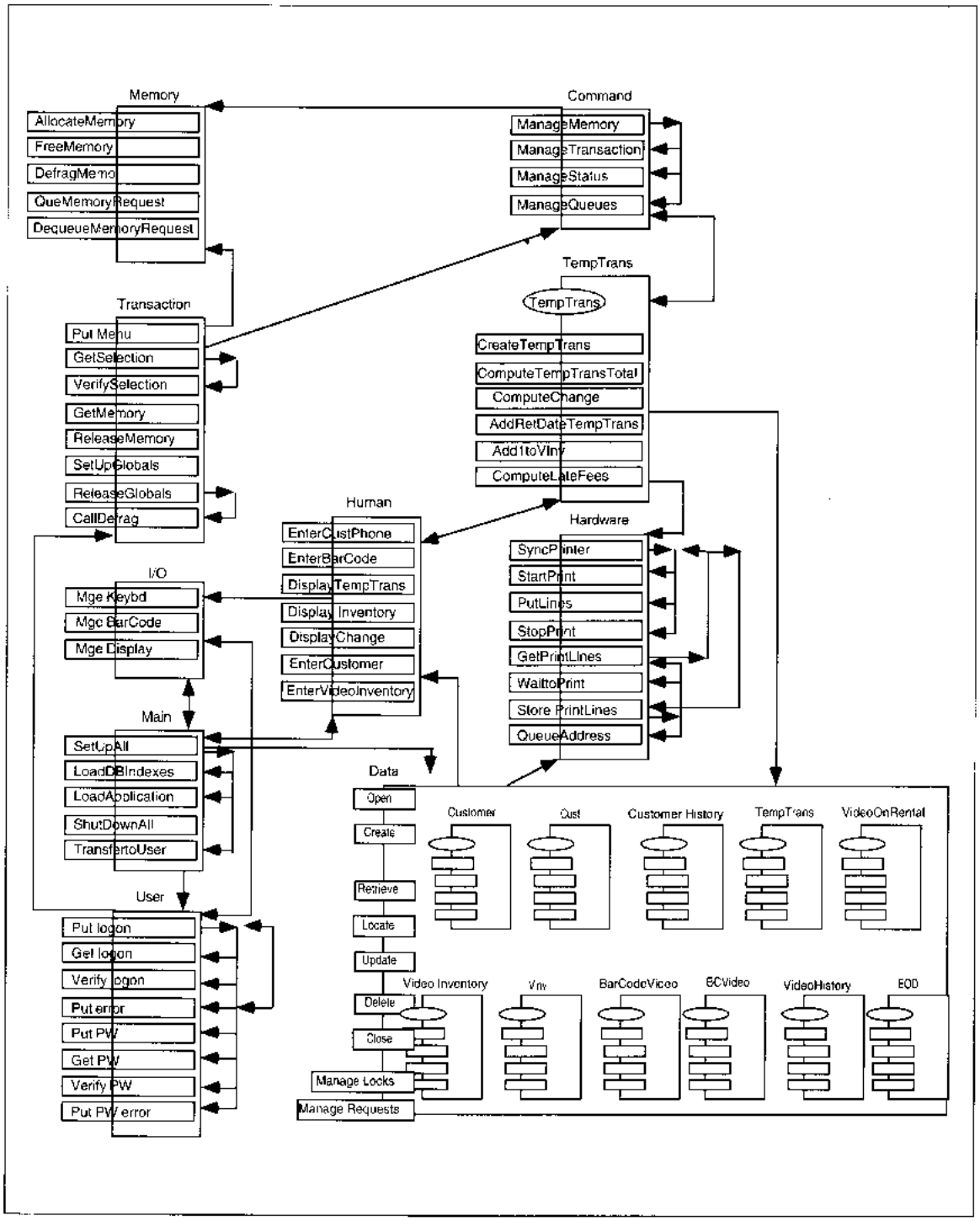


FIGURE 12-A5 ABC Process Diagram

Program fragment to read the customer data:

```
//seekc.cpp
//read particular customer using
passed customer location
#include <fstream.h>  //file stream
class customer
{
protected:
  char custphon [10];
  char custln [50];
  char custfn [25];
  char custadd1 [50];
  char custadd2  [50];
  char custcity [30];
  char custstat [2];
  char custzip [10];
  char cctype [1];
  char ccno [17];
  date ccexp [8];
  date entrydat [8];
public:
  void custdb();
};
void main(custloc& custloc)
  //customer location passed
{
person cust;
  // establish customer object
ifstream cust;
  // establish customer file
infile.seekg(0,ios:end);
  //go to 0 bytes from end
int endposition=cust.tellg();
  //find file position
int n=endposition/sizeof(cust);
  //number of customer on file
int position=(custloc-1) *
sizeof(cust);
  //relative location # * record
size locates individual record
cust.seekg(position);
cust.read((char*)&cust,sizeof
  (cust));
  // read customer information
}
```

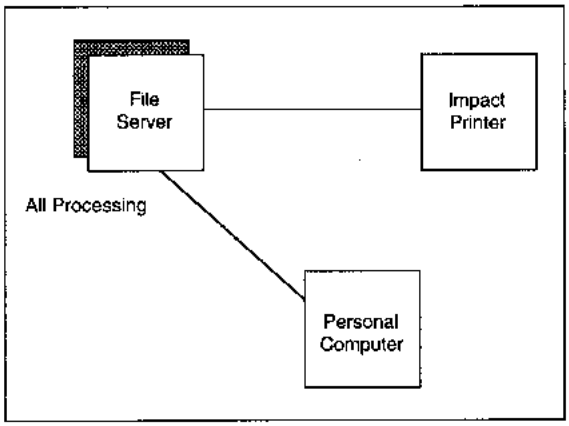


FIGURE 12-A6 ABC Process Diagram

CHAPTER 13

SUMMARY AND FUTURE OF SYSTEMS ANALYSIS, DESIGN, AND METHODOLOGIES

INTRODUCTION

There are an unlimited number of ways in which the methodologies discussed in the preceding six chapters might be compared and analyzed. In addition, significant research is proceeding on individual methods as well as on integrating different methods. To confuse matters, new technologies introduced daily profoundly impact our ability to develop applications and will require equally profound changes in methodologies to be used efficiently and effectively. In this chapter, we first compare the three methodologies to get a fix on their completeness and ability to be used to analyze and design applications. Next, computer-aided software engineering tools (CASE) are critiqued and summarized. The deficiencies and usefulness of CASE are discussed and related both to development of current applications and to the future applications that companies now desire to build. Then, the changes in organizational and technological environments that will require continuous evolution of methodologies are described and related to problems in application development.

COMPARISON OF METHODOLOGIES

In this section, we take two different approaches to summarizing the usefulness and sophistication of the three methodologies discussed in the preceding six chapters. In the first analysis, the phases, information developed, characteristics, and decisions made in the three classes of methodologies are traced following the work of Olle et al. [1988] and expanding the information analyzed for each of the methodologies. Then, Watts Humphrey's maturity framework is described and applied to the methodologies to describe which, if any, might be appropriate for use in a maturing IS organization. In the concluding remarks in this section, we summarize the findings

and discuss the future of the methodology classes and, in particular, the three methodologies discussed in this text.

Information Systems Methodologies Framework for Understanding

In their classic work, Olle et al. [1988], developed the **information systems methodology framework** to compare methodologies, discuss the representation forms, and identify information supported in methodologies available for use in the mid-1980s, including the process methods and data methods analyzed in this text. Here, we summarize the framework to analyze activities and phases supported by the three representative methodologies. Then we extend the analysis to evaluate the phases in which information becomes known, the general capabilities of the methodologies, and the sophistication of resulting designs. Before the evaluation, please be cautioned that these analyses are not intending to condemn or otherwise pass a value judgment on the methodologies presented in this text. If they were not the best of their class, they would not have been selected in the first place. Rather, any shortcomings in the methodologies only point out that an organization must compensate for the lacking activities, phases, or decisions by providing its own guidelines and methods, or by hoping that their analysts have the requisite skills to perform these tasks on their own.

Activities and Phases

This section analyzes the phases of application development work that may begin at the organization level to develop information systems plans (ISPs) based on business objectives. An **ISP** is an analysis of both data and processes that includes manual or automated work to capture a snapshot of the work performed in an enterprise. The ISP is modified to provide the basis for organizational reengineering analysis as discussed in Chapter 5 (which is not part of Olle et al.'s work). Work proceeds according to the framework to include business process, entity and feasibility analysis for a given application. Analysis and design are discussed in terms of the orientation of the majority of tasks performed during those phases. Support for human interface design, allocation of work to hardware or firmware, and DBMS design are all included. Maintenance, the final phase of a project's life, is considered in the extent to which it is supported in the methodology.

Table 13-1 shows the ratings of the process, data, and object methodologies from Chapters 6–12 on these activity and phase criteria. The process method, including the work of DeMarco and Yourdon & Constantine, is most focused, including only analysis, design, and program development techniques and methods.

The information engineering (IE) data methodology is the most complete, covering all phases of the life cycle except maintenance explicitly, and covering all design items to some extent (see Table 13-1). The support for hardware/firmware design is limited to allocation of tasks and data to distributed environments. There are no decisions in IE for how to allocate work to hardware or firmware as in object orientation.

The enhanced Booch and Coad & Yourdon object-oriented (OO) approach ignores front-end tasks, including organization level, business analysis of entities and process, and feasibility analysis. Rather, it assumes that these tasks have been performed before object-oriented methods begin to be used. Object orientation is more specific in its approach to analysis and design than process orientation, and, for some items, than data orientation. OO examines and selects the objects and processes of interest in developing the application during the analysis process. These are then subsequently refined and further defined until design primitives are developed. Object design explicitly discusses the control structure of the application in the form of service objects which can support either batch, interactive, or real-time applications with any number of users, in addition to providing for distributed computing through the development of process diagrams. The other two methodologies do not specifically address design differences that relate to timing or number of users for an application.

TABLE 13-1 Methodology Comparison: Activities and Phases

Knowledge	Process	Data	Object
Business objectives as basis for applications	No	Yes	No
Organization Level Analysis	No	Yes—Information Systems Plan (ISP) or Organizational Reengineering	No
Business Process Analysis	No	Yes	No
Business Entity Analysis	No	Yes	No
Feasibility Study	No	Yes	No
Analysis	Process-Oriented	Balanced Data and process analysis	Objects incorporate both Data and Processes and are defined during Analysis
Design	Process-Oriented	Balanced Process data integration	Encapsulated Object-Oriented
Program Development	Program design has some heuristics but relies on personal expertise of SEs	Program design has some heuristics but assumes use of CASE which generates code	Iterative prototype development is an integral part of the methodology . . . some methods are oriented to specific languages
Human Interface Guidelines	No	Yes	No
Hardware/Firmware Attention	No	Distribution analysis	Yes
DBMS Design Attention	No	Yes—Assumes 3rd normal form relational DBs	No
Maintenance Support	No	No	Assumes independent modules which *should be* easily maintained

To summarize, information engineering (IE) covers more phases of the life cycle and more specific activities as identified by the Olle framework. Object orientation (OO) has more depth to the design phase by providing for design of problem domain, hardware, *and* service object activities. The guidance provided by IE for distributed computing decisions is significantly more detailed than the heuristics provided by object-oriented design for allocation of work to processors.

Where Information Becomes Known

Next, we evaluate the phases in which information becomes known by classifying data, processes,

relationships, and module information at different levels of detail.

Table 13-2 shows that both data and object methodologies provide analysis of all the items but some items are completed in different phases. Major entities and processes can be known during the information systems planning (ISP) activity of IE, if it is conducted. In addition, the current automation state of the entities and processes is identified during ISP as well. The same items, using the term object for entity, are defined during object-oriented analysis and are subject to refinement during object-oriented design. There is no explicit identification of cur-rent automation status for any of the items in OO methods.

Business events and processing triggers are both identified in IE and object orientation. The timing of events, via event diagrams, is analyzed in more detail in object-oriented design, providing a basis for concurrent processing decisions. In IE, events are used to identify triggers for processing and to show where external data entry is performed in the application. Process methods identify necessary data flows into and out of the application, but they are not specifically tied to business events or triggers. The event/trigger distinction is important because it identifies necessary and sufficient inputs whereas data flow identification leads to continuation of past data interactions without consciously reflecting on their need.

The process method does not provide for data relationship analysis, nor is data structure analyzed at either the logical or physical levels. The process method explicitly ignores timing and interprocess relationships.[1] The lack of relationship analysis means that the resulting designs will be less likely to mirror the business requirements of the application. Even Yourdon's 1989[2] update to the methodology fails to integrate data with process analysis.

Object orientation appears more complete for real-time and database applications in explicit analysis and decisions for system, database, or software-specific attributes and processes that might be required of the application. The event diagram more explicitly identifies opportunities and requirements for concurrency than the other methodologies. The reliance of both process and data methodologies (with or without extensions) on designer knowledge and experience leaves too much to chance and puts pressure on designers to remember these tasks (i.e., concurrency analysis and software-specific data design).

General Capabilities

In this section, the methodologies are compared according to the extent to which they support analysis and design of the application characteristics described in Chapter 1: inputs, data, outputs, and constraints. In addition, processes and management of different sources of complexity are analyzed to complete the general description of an application. Inputs include the extent to which information and events that trigger processing are included in the analysis and design of the application. Data are internal, computerized representations of facts about entities in the real work that are stored in the database for the application. Outputs are information that leaves the computer system either to a display or to paper or some other (e.g., image) media. Processes describe the activity being automated, for instance, transaction, decision, or inferential processing.

Constraints define restrictions on objects, entities, data, relationships, or processes within an application. Constraint types include prerequisites, temporal, inferential, structural, and control constraints.

Although not explicitly defined in Chapter 1, the ability of the methodology to facilitate management of problem complexity is a key concern to developers. **Complexity** stems from several sources, including management of the number of elements in the application; the degree and types of interactions, and the need to support novelty and ambiguity.

1 This explicit ignoring of process timing and relationships is in DeMarco and Yourdon & Constantine. In extensions of process methods for real-time systems, these are both analyzed explicitly. For a discussion of the real-time extensions, see Ward, P. T., and S. J. Mellor, *Structured Development of Real-Time Systems* (three volumes). NY: Yourdon Press, 1985.

2 See Yourdon, Edward, *Modern Structured Analysis*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1989.

TABLE 13-2 Methodology Comparison: General Capabilities

Knowledge	Process	Data	Object
Entities/Objects	Feasibility—Begun Design—Complete Terminology differs	During ISP if done Feasibility—High level fully known Analysis—Complete	Analysis—May be revised during iterations
Entity Attributes	Feasibility—Begun Design—Complete Terminology differs	Analysis Design—Complete	Analysis Design—Complete
Entity Identifiers	Design Terminology differs	Analysis	Analysis Design—Complete
Entity Class/Object Structure	NA	Design	Analysis, subject to change during Design
Data Relationships	No	Analysis—Entity Hierarchy	Analysis—Object Lattice Hierarchy
Specific attributes required of operating system, DBMS, or software	Design—Required knowledge of designers, not part of methodology	Design—Required knowledge of designers, not part of methodology	Design—Specifically part of the methodology
Physical Data Design	Design, Programming	Design, Programming	Design, Prototyping
General Processes	Feasibility—Begun Design—Complete	During ISP if done Feasibility—High level fully known Analysis—Complete	Analysis
Detail Process Logic	Feasibility—Begun Analysis—Complete	Analysis Design—Complete	Design
Data relationship to processes	Design	Analysis Design—Complete	Analysis Design—Complete
Events, Triggers	None—Analysis includes identification of external entity inputs only.	Design-Process Triggers on PDFD	Design-Event Diagrams State Transition Diagrams
Process relationships	No	Analysis Design—Complete	Process Timing defined in Analysis with State-Transition and in Design with Event/Triggers
Module Structure	Design	Design	Design
Module Specifications	Design	Design	Design

TABLE 13-3 Methodology Comparison: General Capabilities

Knowledge	Process	Data	Object
Inputs	None	Trigger Identification; Screen Design Heuristics	Event Analysis State Transition Analysis
Data	Minimal	Entity Relationship Diagram, DBMS, Normalization	Object Analysis Object Attribute Analysis
Output	None	Screen Design Heuristics	None
Prerequisite Constraints	None	Yes	Yes
Temporal Constraints	None	Limited	Yes
Inferential Constraints	None	None	None
Structural Constraints	None	Data only	Hierarchic inheritance for data and processes
Controls	None	Problem domain	Includes both problem and service domains
Complexity Management	Top-down perspective	Top-down perspective	Round-trip Gestalt perspective
	Relies on SE skill for proper manual decomposition	Relies on SE skill for proper manual decomposition	Allocate processes to hardware, software, DBMS, and human interface; treat as four separate elements
Management of Novelty	None	None	None
Management of Ambiguity	None	None	None

As Table 13-3 shows, none of the methodologies are complete in providing for analysis of all types of design criteria. None of the methodologies support design of inputs or outputs, even though both data and object methods identify the *need* for inputs via event/trigger identification.

None of the methodologies deal with inferential constraints (see Table 13-3). Remember, the fact that constraints might be missing from a methodology does not mean that they cannot be in the resulting application, only that they must be remembered and designed outside of the methodology and rely on designer skills. Process methods are the most limited in providing no constraint identification and processing as part of the methodology. In contrast, object-oriented analysis specifically provides a step to identify and define the constraints on processing and structural constraints as they relate to both data and processes. IE and data methods are in the middle with prerequisite constraints shown on action diagrams, while structural constraints are limited to those expressed in a class hierarchy for data. Controls are explicitly provided for in both data and object methods and are absent from process methods.

Complexity management is similar in data and process methods since both take a top-down

perspective and are controlled through SE skills. IE decomposition is somewhat easier when an ISP is performed, because the software decomposition follows from primitive business processes which translate into computer processes. The SE skills required, then, are for further decomposition of computer processes into modules and execution units that provide for desired software characteristics such as minimal coupling, maximal cohesion, and so on.

The OO design perspective of round-trip gestalt and explicit use of iterative prototype development supports complexity management to some extent by providing increasingly detailed abstractions of the application with each iteration. OO design also manages complexity through inheritance which minimizes the replication of both data and processes and by allocation of processes to hardware, software, DBMS, and human interface. Through the allocation of objects and processes to each subdomain, the subdomains can be considered independently, even by different design groups. The only need for intergroup coordination is for interprocess message definition.

For complexity management of ambiguous or novel requirements, none of the methodologies provides guidance.

None of the methodologies guide input/output design. Process and object methods are unlikely to be useful in identifying conversion requirements of an application, since they do not differentiate automated from manual data as IE does. Similarly, process and object methods are not likely to lead to well-defined databases since the methods do not provide guidelines for database design.[3] IE provides explicitly for normalization and logical database design while recognizing the need for physical design based on data usage requirements.

None of the methodologies are perfect at complexity management. Object orientation appears to facilitate complexity management more than the other methodologies through its support for inheritance and allocation of processes to subdomains. Novelty and ambiguity of requirements are not addressed by any methodologies.

Sophistication in Explicit Design Decisions

Sophistication means "developed in form or technique,"[4] complex, or worldly. In this section, we rate the methodologies in their ability to guide the development of sophisticated modules, programs and applications to exhibit characteristics of reusability, modularization, information hiding, maximal cohesion, and minimal coupling. The issue is not can the methodologies use or result in modules with these characteristics—the answer is absolutely yes, they can. The issue is the extent to which the methodologies explicitly provide guidelines and validation heuristics for reaching designs that exhibit these characteristics.

Neither data nor process methodologies provide for information hiding, maximal cohesion, or minimal coupling beyond somewhat arbitrary heuristics. Only object orientation specifically can result in a clean design (see Table 13-4), but it can also be corrupted if the designers significantly change intraobject and class/object structures or relationships during design. By early encapsulation of objects and processes during analysis, object orientation automatically imbeds cohesion in the application. By only allowing communication via minimal messages, object orientation automatically provides minimal coupling and information hiding. When implemented using object-oriented DBMSs and languages, object designs should have these properties.

Problems and a loss of minimal coupling and information hiding *will* occur if nonobject languages or software are used to implement OO designs. For instance, COBOL is the antithesis of object orientation. COBOL assumes global data and cannot manage encapsulated objects because it assumes separation of data and process. Therefore, if COBOL is the target language, object orientation would not be a good choice of methodology, all other things considered.

3 Attempts by Booch (1991), for instance, to design databases into an OOD and by Yourdon (1989) to integrate entity-relationship and data analysis in *Modern Systems Analysis* are incomplete and cursory.

4 From *Webster's New World Dictionary*, pocket edition. NY: Popular Library, 1973, p. 544.

TABLE 13-4 Methodology Comparison: Explicit Design Decisions

Knowledge	Process	Data	Object
Extent of Information Hiding	NA	NA	Analysis—Begun Design—Complete
Extent of Modularization	Heuristics rely on SE skill	Uses Process-design heuristics and SE skill	Forces design until primitives, highly dependent on implementation language. Relies on SE skill and prototyping.
Extent of Maximal Cohesion	Heuristics rely on SE skill	Heuristics rely on SE skill	Analysis—Begun Design—Complete
Extent of Minimal Coupling	Heuristics rely on SE skill	Heuristics rely on SE skill	Forced by the methodology but could be subverted by SE errors.
Supports reusable object design	No	No	Heuristics and procedure for identifying reusable objects
Supports reusable module/object use	Yes	Yes	Includes heuristics and limited procedure for identifying reusable objects
Extent of Reusability	Relies entirely on SE skill	Relies entirely on SE skill	Can be 80%+ Organization dependent

The other measure of sophistication is the extent to which the methodologies support reusability and reusable module/object design. Only object orientation provides for explicit identification of potential reusable processes and objects. Once the reusable items are identified, object orientation does not provide further guidance in how to actually design reusable modules; nor should it necessarily provide such guidelines.

IE covers the whole life cycle, something both process and OO methodologies need to provide for application development. The IE data methodology provides more human interface design guidance and is the only methodology that covers the complete life cycle of an application. IEs' disadvantage is that many activities rely on SE skill and experience to know the activity should be performed rather than incorporating the need for the activity in the methodology. When data is complex, nonobject software (either DBMS or language or both) are used, or if human interface design is paramount, information engineering would be the choice.

Structured analysis and design, the process methodology, is the least prescriptive in telling users how to perform the various activities, and it has the least activities in the methodology.

Overall, object-oriented methodologies would be expected to lead to a design that more closely

resembles the functional requirements, if the functional requirements are adequately stated before OO analysis begins. The lack of front-end activities in OO hinders its usefulness in business. Keep in mind that just because object orientation is the most explicit methodology, it is weak in actual data design, human interface design, and must be used with object-oriented languages in order to realize the benefits from its use. Also, every author has a *different* OO methodology with *different* notation and *different* reasoning. As a result, the fledgling OO methodology will change and be refined over the next decade. Large-scale commitment to OO without attaining some consensus and stability of methods certainly adds risk to application development.

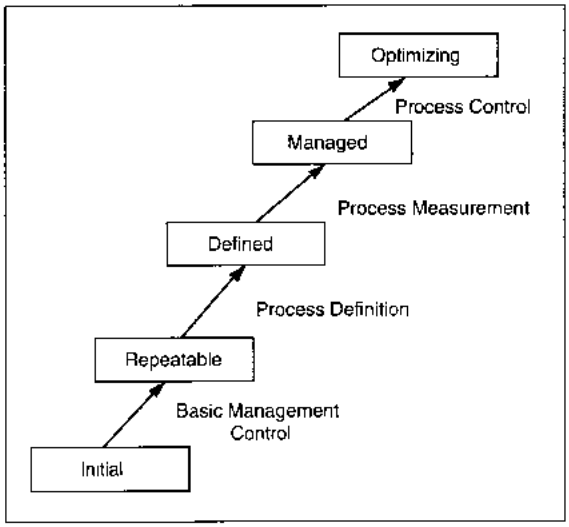


FIGURE 13-1 Humphrey's Five Levels of Maturity

Humphrey's Maturity Framework

The **Humphrey's maturity framework**[5] was developed for the Department of Defense as a self-assessment framework that identifies levels of computing and application development *process* maturity. The goal of the framework is to provide a means of assessing and accelerating technology transfer from research to practice throughout the Department of Defense. According to Humphrey, the ideal software process is predictable, consistent, measurable, and monitored according to objective standards. The maturity levels are initial, repeatable, defined, managed, and optimizing (see Figure 13-1).

At the **initial level**, neither measures (i.e., statistical control) nor orderly progress are possible. This is the level at which organizations operating under no methodology and no life cycle operate. Managerial oversight for quality, productivity, and change control to provide some stability to project schedules are required organizational supports that must be present to even attain the initial level.

At the **repeatable level** the organization has introduced managerial controls in the form of project management cost, schedule, and change controls. Project team members are expected to commit to their tasks and be measured against their commitments. While never actually saying the words, the repeatable level implies the recognition of *both* a life cycle and a methodology, that is, a repeated set of global level tasks with deliverable *products* that implicitly become the measures of schedule and cost performance and that are performed within a definable *process*. Humphrey's reason for having a life cycle/methodology is to provide a framework within which to address the risks to a development project from new tools, methods, and/or technologies. Organizational support in the form of providing for walk-throughs, formal design methodologies, configuration management for code, and application testing standards and methods are required at the repeatable level to continue to the next stage. Humphrey argues the need for a **process group** (i.e., a Standards group) which defines the steps to making orderly progress in project work and that provides a nucleus for transferring the process knowledge to the working groups.

The **defined level** requires the definition of the software development process, which defines the methodology in sufficient detail to guide the work

5 See Humphrey, Watts, *Managing the Software Process*. Reading, MA: Addison-Wesley Publishing, Inc., 1989.

process and define detailed subphase products that collectively become the phase deliverables needed to further manage the tasks. Each deliverable product has process and product measures of quality and productivity that are aggregated to the phase and project level for managerial oversight and assessment. At this stage, a quality assurance group that performs independent analysis of product and application quality is formed to report to management on a product-by-product basis. At the defined level, a *process database* is established and all SEs are trained in the use of the information to provide history for the organization on the use and productivity of each project and tool.

At the **managed level** the organization initiates "comprehensive process measurements, beyond those of cost and schedule" [Humphrey, 1988, p. 302]. The managed process requires analysis of the process database measures to ensure that comparable statistics are available and can be universally interpreted, and that project-specific data that highlight unique characteristics or aspects of application development projects are stored and interpreted properly. At the managed process level, the data for the process database should be gathered automatically and used to modify the process to "prevent problems and increase efficiency" [Humphrey, 1988, p. 306]. Humphrey takes pains to point out that the database should not be used to penalize either project teams or individuals, but that type of use by managers can be taken. One example of measures is function points.

The **optimizing level** is one at which the organization continues improvements begun at the managed level and starts development process optimization. The optimizing level, ideally, allows SEs to identify many types of errors in advance of their causing delays and problems on a current project by analyzing and identifying the patterns of mistakes from other projects based on information in the process database. In my opinion, this is truly an ideal at this point in time since our ability to detail the steps to what appear to be random incidences of Murphy's Laws is rudimentary, at best, and nonexistent, in practice.

While Humphrey's framework is useful for discussing key differences between methodologies, it is not without problems. First, it is based on Humphrey's and others' experiences in the field but has never been subjected to empirical validation of its definitions. Humphrey asserts that the maturity framework "represents the actual ways in which software-development organizations improve" [Humphrey, 1988, p. 307]. The stages are presented as distinct and sequential, with the implicit understanding that to attain, for instance, the optimizing level, an organization must have moved through all previous levels. There is no basis for this supposition. In fact, the framework represents Humphrey's ways of attaining software development maturity without recognizing that it may not fit all situations. The second drawback to the framework in analyzing methodologies is that many of the requisite support activities are organizational, not methodological. For instance, walk-throughs, configuration management software, and testing standards are outside the scope of methodologies. We assume they are not an issue in this discussion.

Having said these criticisms, the framework is still useful for discussing problems with methodologies that relate to the extent to which they define development activities and support phase work.

Table 13-5 shows my subjective ratings of the methodologies with respect to Humphrey's framework. None of the methodologies has a uniformly high rating in all of the categories.

In general, process methods are the least predictable, consistent, measurable, or monitorable because they leave so many activities to SE skill and omit specific activities from the methods. At worst, process methods are at Humphrey's initial stage; at best, they are repeatable. Because the focus is on process, I would assume that consistency and measurability of processes should be medium, that is, different people should arrive at similar analyses. In fact, we think they are low to medium. Designs would be expected to vary most because the heuristics are vague. Data analysis, data design, and human interface design, which some authors add on as an afterthought, would all be expected to vary significantly across different SEs because they are not explicitly part of the methodology.

Measurability is low to medium. Assuming function point metrics, measurability is low because

TABLE 13-5 Methodology Comparison: Humphrey's Framework

Knowledge	Process	Data	Object
Predictable	Low	Medium-High	Medium
Consistent	Low-Medium	Medium-High	Low-Medium
Measurable	Low-Medium	Medium	Medium-High
Monitored	Low-Medium	Medium	Low-Medium

function points concentrate on externals (e.g., numbers of interfaces, files, I/Os, and so on) and not on processing complexity.

The ability to monitor the methodology-defined tasks is probably about medium. The ability to monitor process-oriented applications is low when only methodology-supported phases and tasks are monitored and would be inconsistent if monitored tasks were defined by project.

The data methodologies have slightly better overall ratings. In Humphrey's framework they are, at worst, repeatable and, for some activities, reach the defined level. IE is reasonably predictable in having a set of activities defined into phases for ISP, feasibility, analysis, design, and program design. If using, for instance, Texas Instruments' version of IE, there are many more tasks that are not all necessary for a given application; thus the activities are not completely predictable across projects. The activities should provide a level of consistency across SEs who should be expected to define the same entity-relationship diagram and the same activities even though details would probably differ. Therefore, consistency should range from medium to high. The extent to which IE analyses and designs are measurable is ranked as medium. If function point analysis is used and baselines for the company have been defined, the measurability is probably medium since IE analyzes the major function point items. The extent to which IE can be monitored is medium. IE defines more tasks and activities and follows more phases of the application life cycle; therefore, its ability to be monitored is greater than that of process and object methods. However, all projects are subject to unforeseen problems that require unplanned time, and monitoring cannot assist in foreseeing those problems. Therefore, not all tasks and activities can be monitored to the extent that they eliminate problems during the development process. If a CASE tool, such as IEF, is used for development, monitorability is high because the entire life cycle has well-defined stages, products, and reports on status that can be tracked for all phases.

Object orientation, in the form of the enhanced and integrated Booch/Coad & Yourdon methodology is similar to IE in predictability and measurability. Consistency is lower and varies from low to medium because individual SE skill is required to define the calling sequences and ultimate operational structure of the application, even though the definition of the object *pieces* is fairly well described. The difference between a good calling sequence and message set and a bad one is difficult to define in abstract, procedural terms, but can only be noticed through prototyping and actual comparison of different schemes. Monitorability is less because of the ill-defined nature of service-object identification and of language-specific OO requirements. Moving targets, like OO, are hard to measure. OO is repeatable at best in Humphrey's framework.

The bottom line on methodologies and Humphrey's framework is that the methodologies alone do not offer enough guidance to support the *defined level* of application development management, let alone get to the optimizing level. For this reason, more work on methodologies, life cycle, and development activities are needed to accommodate the variety of work for different types of applications. Having said this, we also need to be realistic about just how much predefinition of decision

processes can, in fact, be imbedded in methodologies. Two things seem obvious. One is that we *can* define some of the methodology-driven activities more completely. The other is that the engineering nature of the SE task is that each application *will* require unique characteristics and design that *cannot be codified!*

In summarizing this section, no single methodology appears to be complete and sufficient for all the tasks and activities performed during an application development. There is no *silver bullet* that will solve our application development problems or provide a complete cookbook for the development process. For these reasons, there will always be a need for SE expertise in application development. There is also a need for continued definition of tasks needed during application development and the continuous evolution of techniques that are integrated into the various methodologies to guide those tasks.

COMPARISON OF AUTOMATED SUPPORT ENVIRONMENTS

There is a marked degree of consensus on many design features of the ideal CASE environment. Table 13-6 summarizes many features and functions that Pressman, Gane, Booch, Martin, and McClure recommend. The curiosity is that the vendors do not seem to listen. Take three general requirements as an example: integration, intelligence, multiuser support.

CASE integration is the absence of barriers between one graphical or text form and others. The experts agree that the most useful CASE should support all project life-cycle activities within an integrated environment. The rationale for this position is that tools that support only application development, even if they include project management, address only a small, possibly noncritical, portion of the SE discipline. Further, the integration should be **seamless**, that is, transparent to users. Transparent integration includes the automatic conversion of diagrams and design text into other forms of documentation or program code with little or no manual intervention. The integration should be both between tools and between life-cycle phases. This level of integration implies that some resolution of fundamental semantic and syntactic differences between phases is required. Specifically, differences between analysis and design should be eliminated through CASE use. To reach this sophisticated level of integration, the methodologies require some redesign to remove their own built-in lack of seamlessness between phases activities. For instance, in process methods, one major intellectual stumbling block is the transition from data flow diagram (DFD) in analysis to structure diagram in design. Many people ask, Why not develop a structure diagram in analysis instead? Or, conversely, Why not carry DFDs through to design?

Next, intelligence in tools is desirable. **Artificial intelligence (AI) in CASE** facilitates reusability and provides consistency and completeness checking within and between graphical and text forms. AI routines can be used to implement the concepts of reusable analysis, design, program specifications, and code. The routines can locate, retrieve, and select specifications matching design parameters and can identify specification fragments that do not match what is required. Other applications of AI are the analysis of completeness and consistency of requirements or code. Other checking is between phases to match logical design to physical design to code. This use of AI is technically feasible and not particularly difficult. What we don't know about AI for these uses is what to match, how to match it, and when the best time for matching occurs. New meta-language descriptions of analysis and design requirements will be required to fully exploit AI in CASE. These meta-languages must also be consistent and no additional burden to the other integration and multiuser support requirements of CASE.

One consistently recurring theme in all CASE products and research is concern over the replacement of one sort of complexity with another sort of complexity. The solution to software development productivity, quality, and reliability problems is to build tools that, in hiding some complexities of the development process, are necessarily complex themselves. The hidden complexities require absolute

TABLE 13-6 Desired Computer-Aided Software Engineering Features and Functions

- Project Management:
  - Work breakdown
  - Cost estimation
  - Person/task scheduling
  - Monitoring allocated vs. actual times
  - Budget creation
  - Monitoring budget vs. actual money spent
- Documentation for all Work
  - Word processing editor functionality
  - Integration of text and graphics
  - Nesting of text, graphics, and so on with recall at all levels
  - Document templates—predefined and customizable
  - Query capabilities to all parts of the graphical and text definitions
  - Version/release control support
  - Change control support
- Analysis
  - Graphical and text support for specific methodology
  - Intelligent syntactic evaluation of completeness and correctness
  - Repository (i.e., dictionary) support for all graphic and text information with nesting and linkage within and between levels
  - Support for reusable component recognition, definition, use
  - Human interface definition support
  - Prototyping support
  - Customizable reporting facility
- Design
  - All analysis functions above
  - First-cut of *next step graphical form* from analysis via automated functions
  - Support for program definition language (PDL) with interface to code generators for several languages
  - Bi-directional interface to analysis and code from design
  - Sensitivity analysis on designs
- Code
  - All above plus
  - Source code templates
  - Source code syntax checking and comparison to requirements
  - Automated code generation
  - Automated third normal form database definition from repository data definitions
  - Automated minimal test set definition . . . with generation of test data
  - Integration to software configuration management tool
- General
  - Consistent interface with function keys having identical uses across phases
  - On-line documentation, suggestions for problems
  - Adaptability to local conventions for methodology use
  - Support on any operating system, hardware platform, DBMS generation, and if not, machine independence of designed application
  - Interfaces to other tools and products

accuracy and reliability themselves to make their use worthwhile; the systems will have to reveal themselves upon request so users may understand internal processing. With AI routines, that, for instance, learn to predict what is required for code based on design specifications, these revelations are crucial to guaranteeing CASE's continued use.

The integration of phases and tools must also be multiuser. **Multiuser CASE** support implies some sort of centralized repository of information about the application that is accessible by any number of people concurrently. Warnings to users when a component is changed and automatic version control are desired features. Multiuser support extends to group work collaboration, scheduling, tracking, sensitivity analysis, and electronic meeting support.

Now, let's first examine the extent to which the methodologies themselves exhibit the properties thought to be desired for CASE, then extrapolate from that to determine the level of support for these features we can realistically expect from CASE products.

First, integration across phases and graphical forms is important to building intelligence into CASE. If we examine the three methodologies described in this text, structure analysis and design (SA), information engineering (IE), and object orientation (OO), we would find the most integration in

OO with less in IE and even less in SA. OO begins with tables that are increasingly elaborate but whose contents can be traced from the beginning of analysis through to development of module specifications. There is no *shift* in thinking required once the data and processes become encapsulated, because they continue to be encapsulated throughout the remaining steps.

IE has less integration because there are two fairly distinct paths of thought in IE, one for data and one for processes. Within each path, the level of integration is consistent and high, but between paths, the integration is less consistent and there are few guidelines for integrating the two. One example of this lack of consistency is that, depending on the author, IE should not have data files or entities shown on action diagrams; action diagrams should remain a process sequencing and event trigger identifying graphical form. If this line of reasoning is followed, data and processes are integrated at the program specification level. Program specification work is micro-design that could then miss major global problems because of the lack of data-process integration.

SA is even less integrated than IE because data are not specifically addressed in the methodology. The analyst is supposed to know what 'data stores' are required and the appropriate contents of those data stores. Some authors[6] assert that a data store can refer to a group of related normalized relations, while others[7] assert a data store is a third normal form relation. When data analysis is not an official activity, by definition it cannot easily be integrated into the methodology. Similarly, there are numerous texts that describe how to use SA for developing real-time applications[8] and that provide a foundation for several of the graphical forms used in OO. But close analysis of the Ward & Mellor methodology, for instance, identifies a very different approach to developing applications from the original DeMarco and Yourdon & Constantine approaches.

Given the levels of integration as low for SA, medium for IE, and medium to high for OO, the greatest potential for CASE to provide *seamless*, complete integration of functions seems most likely for object orientation. Further, the higher the level of integration, the greater the intelligence that can be built into the software, once again, identifying OO as the most likely to provide extensive use of AI. Does that mean that AI cannot be used for the other methodologies? Absolutely no! It means that *sophisticated* AI that recognizes reusable analysis, design, or code fragments and that performs significant *semantic* analysis of the contents of diagrams and the interdiagram relationships is *most likely* in OO. Anyone using *any* CASE tool today knows that they provide fairly extensive syntactic evaluation intelligence that will tell you, for instance, if your connections on a data flow diagram (DFD) are all legal, or that the external entity interactions from the context diagram are all accounted for in the DFD.

From the discussion of the previous two issues, you should be able to figure out that multiuser support in products also lags behind the desire for its sophistication in industry . . . and it will continue to do so for at least five years. Multiuser support adds a level of underlying complexity because of the need for locking mechanisms, access security, and concurrent multiplatform hardware support that impedes vendor development. Since there are no competitive reasons for developing multiuser capabilities, that is, no other vendors have it either, vendors are not spending their resources on multiuser support. Current tools with a central repository allow segmenting of repository items, such as an ERD. When multiple users want to change the ERD, they check out segments and work on their respective segments individually. The completed checked-out segments are checked-in to a reconciliation procedure that frequently fails because of inconsistencies that are then manually reconciled. In a truly concurrent environment, locking mechanisms would support multiple concurrent users without segmenting and check-out processing, but with locking mechanisms similar to those used in DBMS software.

6 See Gane, Chris, *Computer-aided Software Engineering: The Methodologies, The Products, and The Future*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1990.

7 See Yourdon, 1989.

8 Ward, P. T., and S. J. Mellor, *Structured Development for Real-Time Systems* (three volumes). NY: Yourdon Press, 1985, is one of the most commonly used.

What does the state of integration and AI mean for CASE? CASE tools are necessarily limited in the number of processes, number of entities, number of attributes, complexity and detail of description, and so on. These limitations are higher candidates for removal by vendors than are these three more abstract concepts: integration, intelligence, and multiuser support. The CASE industry has entered a *push-pull* stage of product development. The *push* comes from the ever increasing desire of client companies to develop ever more complex and sophisticated applications, and their recognition that CASE *can* be used to deploy ITs to their competitive advantage. The *pull* comes from the products on the market and their growing sophistication. As soon as one vendor provides a feature or function, others feel obligated to offer it too, or risk losing market share. Many vendors try to support as many methodologies as they can, frequently without regard to underlying differences in mental thought processes required to comply with the methodologies. So, for instance, DeMarco's SA and IE analysis might both be advertised as supported by the same vendor. But DFDs are *not* action diagrams and vice versa, nor will they ever be. So, when vendors claim multimethodology support, beware of the claim.

RESEARCH RELATING TO ANALYSIS, DESIGN, AND METHODOLOGIES

There are two growing bodies of research[9] relating to methodologies and the application development process. The first research is attempting to reconcile the differences in methodologies to develop an improved hybrid. The second type of research studies the decision processes that occur in analysis and design activities. Both of these lines of research are described in this section and related to future changes that we might expect in methodologies and application development.

The methodology research consists of normative and descriptive writing on the procedures and application focus in analyzing application problems. From this body of work, we have over 60 identifiable methodologies with primary concentrations, such as SA, IE, and OO, described in this text. Unfortunately, the value of these methodologies has not been studied. There is no evidence that *any* of these methodologies is better than any other of these methodologies. Nor is there any evidence that *any* methodology is more appropriate for a particular problem domain than any other. Intuitively, they can't all be best in all situations. Current research is taking two directions to follow on this idea: First, one line of research attempts to integrate methods to create an improved hybrid; second, the other line of research is trying to determine when and which methodologies are appropriate for different types of problems.

Current research in building hybrid methodologies is primarily applied. All authors, so far, are seeking to integrate OO notions and notations with some other methodology, including structured analysis, Jackson systems design, information engineering, and others.[10] This research is purely prescriptive, of the form: "If I were going to put OO together with structured analysis, here's what I would do." While this research is promising, the lack of researcher attention to the differences in reasoning and thinking processes of the methods needs to be resolved. Also, these authors will need to offer evidence of the synergy they promise but for which they currently offer no evidence.

The second type of research discusses methodology learning by novices. Having learned COBOL or another procedural language, novice learning of structured analysis is easier and more accurate than learning of other methodologies.[11] Since there is less to learn, this is not surprising. In addition, this research notes that the thought processes of OO are decidedly different that those of SA and IE. We would conclude then that novices who learn Ada

9 See Adelson & Soloway, 1985; Guindon & Curtis, 1988; Guindon, Krasner, & Curtis 1987; Pennington, 1987; Vessey & Conger, 1993.

10 See for example, Sanden, 1989; and Ward, 1989.

11 See Vessey and Conger, 1993.

first, for example, would have an easier time learning OO than structured analysis, and their OO designs would be more accurate. This is a promising line of work that needs much more study, including analysis of real analysts doing real work before any results applicable to business use of methodologies can be expected.

The study did find that analysts' development of a mental model is crucial to complete solution of a task. The process followed by successful analysts includes development, expansion, and simulation of a mental model that uses personal problem-solving plans that are used to elaborate constraints, and notemaking as a means of deferring work until a later time. Many of these skills in Chapter 2 recommended for you to think of while studying the text were identified through this research.

Also, some comments about easy and hard features of methodologies can be developed. The easy features of OO are those that automatically lead to information hiding, minimal coupling and maximal cohesion, the traceability of information throughout the process, and the essential continuity of the method (i.e., building tables and progressively adding details to the information). The hard OO features are the extensive experience in operating systems required to determine service object requirements and the significant coupling between the implementation language and the application design.

The easy features of IE are entity analysis, full-life cycle approach including enterprise through maintenance phases, the methods for deciding distribution, and the balanced thinking given to both data and processes. The hard IE features are the mental shift required to move from design to program specification and from an action diagram to its components. The decisions about the size and content of components is left to the SE.

The easy feature of SA is the simplicity of the thought process which is easily grasped by most people. The hard SA features are the disjoint phase relationships moving from DFD to structure diagram and decomposing the structure diagram into modules. These actions, like similar ones of IE, are left to SE skills and have few guidelines.

To summarize the application development literature, we know that skills needed seem to vary by activity both across and within phases of a system development life cycle, that task domain facilitates the process of building a mental model of the problem solution, and that different types of domain knowledge exist, including methodology and task domains.

For SEs, this research has several implications. First, the entire field of methodology research is in its infancy. As it matures, both the methods and the way we use them should be expected to change. Second, hybrid methodology that attempts to integrate methodologies requiring different mental models of a problem, for instance, structured analysis and OO, are unlikely to be very productive. Rather, we need to identify which methodological orientation best fits different problem domains, concentrating on methodology improvement and use in the appropriate domains.

Last, since methodologies do not provide complete analysis of all aspects of problem domains, by definition, CASE tools based on the methodologies will also provide partial task coverage. The more complete the methodology, the more complete the CASE tool. Some vendors add completing tasks to support, for example, code generation; these CASE tools are even more complete than those that are only methodology-based. The most notable example of a more complete tool is Texas Instruments' Information Engineering Facility (IEF).

Applying Humphrey's framework to research in IS, methodologies are in either the initial stage or the defined stage. CASE tools help methodologies attain the defined stage, but sometimes impose such rigidity in doing so that usage is constrained and might not fit either the way SEs work or the work itself.

Business and Technology Trends that Impact Application Development

There are several trends in application management and development that will change dramatically the way business computing is performed in the next ten

years. The trends are both technological and business related, including management of legacy systems and data, client/server computing, development of repositories and data warehouses, multimedia application development, and the business globalization. Each of these trends are briefly described with their impact on application development and software engineering.

Legacy Systems

Legacy means handed down as from an ancestor. **Legacy systems** are applications that are in a maintenance phase but are not ready for retirement. Legacy systems are most often mainframe, COBOL applications that were probably built using no methodology and no life cycle. Such applications are frequently referred to as 'held together with spit and glue' because they are **fragile**, that is, susceptible to introduction of errors caused by unrelated changes. In short, they are a liability. The reason these systems are not all rewritten and done away with is because of the tremendous investment in their development.

A related concept is **legacy data** which is data used by outdated applications that are required to be maintained for business records. Legacy data are as much as 50% incorrect and may be in an unusable form without considerable expenditures of time and money. In short, they are a liability. The reason legacy data are not reformatted in some new DBMSs that can optimize storage and access time is the inherent cost of correcting the data which could be ten times or more than the cost of reformatting.

The impact of legacy data and systems is to inhibit and slow the integration of data across organizations and applications, and to inhibit the integration of technologies for application use. Ultimately, companies with significant legacy problems will be forced, for competitive reasons, to spend the money to transform the systems and data into useful items or to abandon them and write off the expense.

The impact of legacy systems and data on SE is to continue to inhibit new application development by requiring attention. The new *pulls* from industry include need for reengineering data, methods, and software that support data *scrubbing* to remove anomalies and errors. These are nontrivial needs that will divert some industry resources away from methodologies toward these very practical and real problems.

Repositories and Data Warehouses

A related issue is the notion that organizations no longer want to discard data. For instance, the maintenance of legacy data sometimes is mandated by the government. The means to store unlimited, continuously growing databases currently are called **data warehouses**.

Similarly, all of this data must have meta-data that defines each attribute and its related entities (or objects), the applications and software allowed to access the data, and the allowable using organizations. The meta-data definitions are in a **repository** which, in its most sophisticated form, is a data dictionary for data, processes, hardware, and software.

Repositories control and centralize management of data as an organizational resource. Distributed repositories will be developed in the future but are currently only available as one-user chunks of a centralized repository that must be reintegrated with the centralized, official data.

Both repositories and data warehouses have significant overhead (i.e., human) costs associated with managing and tracking all of the information actually managed by the software. Because of this overhead expense, companies must choose carefully those items they really want to maintain indefinitely. The luxury of being a 'data packrat' has currently unknown costs.

The impact of data warehouses will be felt in the need to design time-dependent databases[12] that have associative relationships and to migrate legacy data to the warehouse. **Associative data relationships** are irregular, dictated by data content rather than abstractions such as normalization. An example

12 Time-dependent databases are also referred to as temporal databases and have an entire body of research associated with their definition and use.

might be in an image document that describes an insurance policy. That policy needs to be related to the insured, the owner, the beneficiaries, and its value *over time* so that a complete reconstruction of its status at any single point in time can be determined. Existing database products can support temporal databases but are not specifically designed for temporal data. This implies the development of a specialized temporal database type, or the extension of existing database products to accommodate temporal data definitions.

Client/Server

Client/server computing describes a situation in which multiple processors share responsibility for managing pieces of an application. Currently, the pieces include data, presentation software for the human interface, and application. For a given processing request, one processor acts as a client requesting that a processing service be provided; the other processor is the server that executes the request. In this context, examples of a service request are to access data, perform a routine, or display data on a terminal screen. In a true client/server environment, any processor can be a client and any processor can be a server. The same processor might be a client for some actions and a processor for others. Therefore, the client/server environment, in its truest form, is describing a **peer-to-peer networking** scheme in which intelligent sharing of resources and data across multiple processors is taking place.

The state of client/server development changes almost daily, so by the time you read this, Figure 13-2 will be out-of-date. Don't worry, it is only to give an example of the alternatives and confusion in the client/server marketplace. The figure shows the alternative configurations of presentation software, data (and DBMS), and application software with traditional, centralized mainframe resource management on the upper left of the diagram. Moving down the diagonal to full distributed client/server processing, we have first presentation software that resides both on the mainframe and on a PC. The PC software interfaces to the mainframe presentation software and is translated for use by the application. At the next level of sophistication, the presentation software is offloaded to the PC completely. Then data is partitioned (i.e., split by columns or rows or both), and accessible via DBMS in both places. Next, the data are moved fully to the distributed environment, possibly with replication (i.e., multiple data copies). At the next stage, some application functions are performed on a PC and others on a mainframe. In its most advanced state, all functions (or pieces of each) are stored both on mainframes and PCs and with access determined by the closest processor with available CPU time.

In client/server's most advanced form, for example, simple functions might be on a LAN and complex processing functions on a mainframe. The data might be anywhere. The application part closest to the request decides type of processing to be performed and ships the request off to be executed in the most efficient place. If that location is busy, its software might forward the processing request to another processor until idle CPU cycle time is found. The executing processor would obtain the nearest version of the data and perform the requested service. The result is sent back to the requesting processor.

Client/server processing is sometimes confused with downsizing. **Downsizing** is the shifting of processing and data from mainframes to some other, less expensive environment, usually to a multiuser mid-size machine, such as an IBM AS400, or to a LAN of PCs. Downsizing can occur with or without client/server computing. The reasons for buying mainframes are diminished with the availability of client/server computing, but the compelling argument for maintaining an existing mainframe environment is to obtain the most benefit from the tremendous start-up and maintenance costs associated with them. Downsized environments also have large start-up costs that sometimes are equivalent to mainframe start-up cost.

The impact of client/server computing on SE is here now. There is tremendous demand for SEs who know how to integrate data, applications, and presentation software over multiple processors and networks. The large accounting companies, such as Ernst & Young, who also do consulting, have found a niche in providing leading-edge services of this type. But the need is in every size of company, even

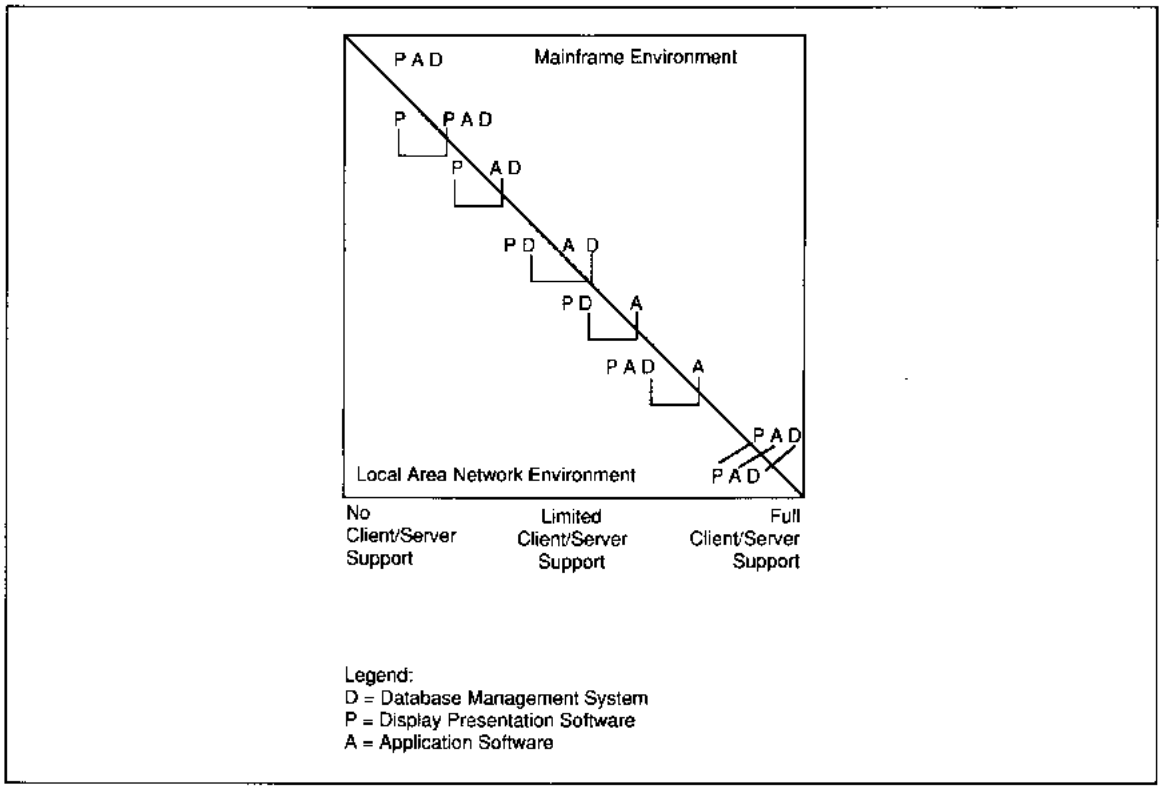


FIGURE 13-2 Client/Server Alternatives

those that cannot afford a large consulting company's fees. The pressure on SE professionals then is to develop the integration skills to develop and support these applications as fast as possible.

Multimedia

Multimedia is a term that describes the integration of object orientation, database, and storage technologies in one environment. By the 21st century, multimedia will transform both applications and the way we interact with them. New technologies must be able to be incorporated into traditional application processing to be useful in business organizations. By defining equipment as objects and storing the object definitions in a database repository, integrating new equipment and technologies in traditional applications becomes not just possible, but fairly easy.

SEs developing multimedia applications require new skills for authoring the contents of multimedia systems, and for developing the applications that make the information accessible in a meaningful manner. For graphic design, video direction, and so on, one strategy has been to hire graphics artists or movie school graduates, for instance, to be multimedia authors rather than to teach an SE about video production. This splitting of duties still requires SEs to develop skills in integrating multimedia in applications. At present, the skills required include OO analysis and design, media knowledge, and human interface design incorporating moving and still-motion video, graphics, text, and data in the same interface.

Globalization

Globalization is the movement of otherwise local businesses into world markets. In 50 short years, business organizations worldwide have evolved

from national to multinational to global enterprises. As with all trends, there are forces that both ease and inhibit movement into global markets. In general, information technologies enable globalization; and, in general, cultural differences and history inhibit globalization. The technology enablers are application and communications technologies that remove the barriers of geography and time, while providing equal access to multimedia applications. The historical and cultural barriers inhibit cross-cultural exchange of ideas, technologies, and methods of work. Dealing successfully with both the technological and cultural issues is a challenge to information systems professionals and business managers. Preparing yourself for deploying globalizing technology is the challenge to SEs today.

There are three main social barriers to globalization of businesses: infrastructure differences, technology transfer differences, and political and cultural differences. **Infrastructure** usually refers to the installed base of equipment and services for communications, transportation, and services of a geographic entity (i.e., a country). Infrastructure relates to computers, telecommunications, and supporting software, including, for instance, database and networking software.

There are two infrastructure challenges to SEs. The first challenge is technical, learning both current and past technologies, and devising sometimes messy ways to integrate them. The second challenge is social, developing and presenting alternatives and trade-offs for imaginative, practical, cost-effective applications in developing countries.

Technology transfer is a large scale introduction of a new technology to some previously nontechnical environment. Transfers of computing and communications technologies to all developing countries in Eastern Europe, Asia, Latin America, and Africa are needed. History leaves me pessimistic about such transfers taking place easily, smoothly, or soon. Broadscale transfers for such disparate technologies as farming methods, birth control, building of dams, and water purification have failed simply because technologists fail to contend with cultural differences and resistance.[13] Technology transfer suffers from the same bias that diffusion of innovation theory in general suffers: If the technology is not accepted, there is something wrong with the intended user, not the transfer agent or the technology. Naively, we think our way of implementing and using are the *right way* as if no other way is as good. The concept of **equifinality**—many paths lead to the same goal—eludes most Westerners. We fail to evaluate the technology *within the context* of the intended cultural structure. We assume stupidity on the part of the users and also assume this stupidity can be corrected by sufficient education. What we forget is that projects fail when planning is incomplete, potential difficulties are not assessed or are misassessed, and cultural impacts of projects are insufficiently analyzed. The challenge to SEs is not to oversimplify projects and circumstances of their implementation that inhibit technology transfer, but to attend to the cultural aspects of implementations.

In any technology transfer project, it is imperative that the sensitivity to local differences is maximal. Teaching and training in a different culture does not mean making the target audience the same as you. Equifinality must be allowed. SEs' roles change from doer to facilitator, with less control than usual over outcomes. Successful globalization of applications and technologies requires considerable breadth experience for SEs; for those who can develop and integrate the necessary business skills with their technical skills, the rewards will be huge.

Client/server and multimedia are technologies that enable globalization and require different ways of thinking in a global context. Most effective placement of data, database software, software, storage media, and computers is the main issue. Distribution of data and functionality will require new decision criteria. Before distributed applications, decisions were based on what the software and hardware could do. Constraints drove the decision process. Now we can have anything anywhere. The decision criteria shift from being technologically driven to being business driven. Why do we need data x for y PCs in location z if we can have a data for b PCs in location c? *What business requirements* demand *this* placement of data, hardware, and so on? The extent of distributed multimedia access and enabling of peoples in far-off locations that takes place will become a conscious business decision.

13 See Hirschman, A. O., *Development Projects Observed.* Washington, D.C.: The Brookings Institution, 1967.

Ethical, political, and practical issues inform distributed media placement decisions.

Multimedia applications, because they support data, graphics, photos, audio, and video images, also have a significant cultural component in a global application. Design of culture-free or culturally-rich applications becomes a decision. Is it truly possible to design culture-free applications? My feeling is no, all applications have cultural assumptions at least implicit in their design. Multimedia will make obvious our assumptions about appropriate words, pictures, and ideas for users. Biases that surface will relate to information system developers, user designers, and manager approvers. When applications go global, assumptions that survive in the United States, in all likelihood, will be inappropriate globally. The assumptions will require development of the same application with different media components to fit the using culture. SEs will need to learn how to surface cultural assumptions of application developers and how they carry over to the finished product. SEs will need to make assumptions explicit, then use the assumptions to design cultural diversity into applications.

In summary, business and technical trends are pointing toward breadth and depth of skill levels in SEs in many different areas. Methodologies do not support these trends today. Therefore, continued evolution and change to methodologies can be expected.

SUMMARY

Two methods of analyzing methodology classes were used in this chapter. The first, the information systems methodology framework, was extended to include the characteristics of applications from Chapter 1 and the desirable characteristics of applications. From the analysis we know that both information engineering (IE) and object orientation (OO) are more complete in describing applications than structured analysis (SA), but each addresses different phases of the life cycle. IE is more complete in coverage of organization level information systems planning and analysis, both of which precede design and implementation. OO is more detail- and programming-oriented, resulting in a deeper level of design by the end of the design phase. SA is so process-oriented that data, input, output, and other detailed aspects of the application are left to SE skill and are not specifically addressed by the methodology.

The second analysis of methodologies used the Humphrey's maturity framework to discuss the maturity of methodologies. Humphrey discusses the initial, repeatable, defined, managed, and optimizing levels of maturity. The results of this analysis show that no methodologies are currently beyond the defined level and that SA is only at the initial level. There are too many activities that are not addressed by SA to reach the repeatable level for all requisite tasks. At the repeatable level different people would arrive at the same design. IE is at worst repeatable, and, when completed in a CASE tool, may reach the defined level. OO is at the repeatable level for many early activities, but is at the initial level for package and message communication design.

CASE tools were discussed in their ability to provide three key design objectives: integration, intelligence, and multiuser support. The ability of CASE is hampered by methodologies that are not themselves integratable because of shifts in thinking that must be made from one phase of work to another. In general, SA and IE characterize such shifts and have relative difficulty in CASE interphase integration of work. In contrast, OO is more consistent in the thinking and documentation forms both within and between phases, thus, the CASE tools supporting OO are more highly integrated and represent the ever more detailed thinking required in OOD, and do so within similar graphical and text forms throughout the CASE tools.

Next, business and technology trends that impact application development were discussed, including legacy systems, repositories and data warehouses, client/server computing, multimedia applications, and business globalization. Legacy systems and data are historical leftovers from premethodology days that may have errors and structural flaws that make their conversion to new environments costly and difficult. In particular, client/server, data warehouses, and repositories are three emerging technologies to

which companies want to migrate the legacy systems and data. Client/server environments provide for storage and processing of data wherever it is most needed by the organization in a peer-to-peer network. Data warehouses are storage technologies that provide for massive amounts of historical data. Repositories are versatile means of storing information about data, applications, hardware, and software that provide the definitions of interchangeable technology components. Multimedia applications will use repositories to define the integration of object orientation, database, and storage technologies in one application environment.

Globalization is the movement of businesses into worldwide markets. Global application developers must deal with difficulties in development due to infrastructure differences and technology transfer difficulties. Technology transfer is the large-scale introduction of new technology to a new environment, usually a developing country. Problems in technology transfer relate to cultural and political differences more than to the new technology. SEs developing global applications will need to attend to the culture and politics to be successful. Client/server technology enables global applications. Multimedia was discussed as one type of application with a significant cultural component.

REFERENCES

Adelson, B., and E. Soloway, "The role of domain experience in software design," *IEEE Transactions on Software Engineering, SE-11*, Vol. 11, 1985, pp. 1351–1360.

Bergland, Gary D., "A guided tour of program design methodologies," *IEEE Computer*, October 1981, pp. 13–37.

Card, David N., Frank E. McGarry, and Gerald T. Page, "Evaluating software engineering technologies," *IEEE Transactions on Software Engineering*, Vol. SE-13, #7, July 1987, pp. 845–851.

Conger, S. A., "Teaching globalization in information systems courses," in *Global Information Technology Education: Issues and Trends* (M. Khosrowpour and K. D. Loch, eds.). Harrisburg, PA: Idea Group Publishing, December 1992, pp. 313–353.

Datamation, "The best in client/server computing," Special Issue, October 1, 1991, pp. 1–24.

Dunsmore, H. E., W. M. Zage, D. M. Zage, and G. Cabral, "Building an empirical case for CASE," Software Engineering Research Center Report SERC TR-8-P, Lafayette, Indiana, December 16, 1987.

Episkopou, D. M., and A. T. Wood-Harper, "Towards a framework to choose appropriate information systems approaches," *The Computer Journal*, Vol. 29, #3, 1986, pp. 222–228.

Gane, Chris, *Computer-Aided Software Engineering: The Methodologies, the Products, and the Future*. Englewood Cliffs, NJ: Prentice-Hall, 1990.

Guindon, R., and B. Curtis, "Control of cognitive processes during software design: What tools are needed," *CHI Proceedings*. ACM: 1988, pp. 263–268.

Guindon, R., H. Krasner, and B. Curtis, "Breakdowns and processes during the early activities of software design by professionals," in *Empirical Studies of Programmers—2nd Workshop* (G. Olson, E. Soloway, S. Sheppard, eds.). Norwood, NJ: Ablex Publishing Co., 1987, pp. 65–82.

Hirschman, A. O., *Development Projects Observed*. Washington, D.C.: The Brookings Institution, 1967.

Humphrey, Watts S., "Characterizing the software process: A maturity framework," reprinted in *Milestones in Software Evolution*, Paul W. Oman and Ted G. Lewis, eds. Washington, D.C.: IEEE Press, 1988, pp. 301–307.

Humphrey, Watts, *Managing the Software Process*. Reading, MA: Addison-Wesley Publishing, Inc., 1989.

Iivari, Juhani, "Levels of abstraction as a conceptual framework for an information system," *Proceedings of IFIPS WG 8.1: Information Systems Concepts: An In-Depth Analysis*, Belgium, October 18–20, 1989, pp. 122–151.

Kelly, John C., "A comparison of four design methods for real-time systems," *ACM SIGSOFT Software Engineering Notes*, Vol.12, 1987, pp. 238–251.

Keys, Paul, "A methodology for methodology choice," *Systems Research*, Vol. 5, #1, 1988, pp. 65–76.

McClure, Carma, *CASE Is Software Automation*. Englewood Cliffs, NJ: Prentice-Hall, 1989.

Olle, T. William, Jacques Hagelstein, Ian G. McDonald, Colette Rolland, Henk G. Sol, Frans J. M. Van Assche, and Alexander A. Verrign-Stuart, *Information Systems Methodologies: A Framework for Understanding*. Wokingham, England: Addison-Wesley Publishing Company, 1988.

Panzi, David J., "A method for evaluating software development techniques," *The Journal of Systems and Software*, Vol. 2, 1981, pp. 133–137.

Pennington, N., "Stimulus structures and mental representations in expert comprehension of computer programs," *Cognitive Psychology*, Vol. 19, 1987, pp. 295–341.

Pressman, Roger S., *Making Software Engineering Happen: A Guide for Instituting the Technology*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

Sorenson, Paul G., Jean-Paul Tremblay, and Andrew J. McAllister, "The metaview system for many specification environments," *IEEE Software*, March 1988, pp. 30–38.

Wand, Yair, and Ron Weber, "On the deep structure of information systems," *Information Systems Research*, Vol. 4, #2, 1993, pp. 23–45.

Ward, P. T., and S. J. Mellor, *Structured Development for Real-Time Systems* (three volumes). NY: Yourdon Press, 1985.

Yourdon, Edward, *Modern Structured Analysis*. Englewood Cliffs, NJ: Prentice-Hall, 1989.

KEY TERMS

AI in CASE
associative data relationships
CASE integration
client/server
complexity
data warehouse
downsizing
equifinality
fragile applications
globalization
Humphrey's defined level
Humphrey's initial level
Humphrey's managed level
Humphrey's maturity framework
Humphrey's optimizing level
Humphrey's repeatable level
information systems methodology framework
information systems plan (ISP)
infrastructure
legacy
legacy data
legacy systems
multimedia
multiuser CASE
peer-to-peer network
process groups
repository
seamless CASE
technology transfer

EXERCISES

1. Write a three- to five-page paper describing some new technology—distributed database (e.g., Informix or Sybase), Multimedia, Simple Network Management Protocol (SNMP) (networking protocol), imaging. Predict how the technology will change in use in applications in the next five years. Predict IS and user organizational changes as well as design changes.
2. Discuss globalization of businesses and other changes to software engineering activities that might be required.
3. Compare the methodologies using your own technique. What are the important methodology issues to you? How easy or hard do you find the work involved in describing the ABC application in each methodology? How easy or hard is it to really learn each methodology? Which are you most likely to continue using? How likely do you think these methodologies are to be useful for the emerging technologies of client/server and multimedia? How would you change any or all of the methodologies to make them more usable? How might methodologies become less tied to technology? (Please send your responses to the author.)

STUDY QUESTIONS

1. Define the following terms:
   client/server
   downsizing
   equifinality
   globalization
   Humphrey's maturity framework
   legacy data
   repository
2. What phases of application development are in the Olle et al. information systems methodology framework?
3. Describe the features of the Olle et al. approach to comparing methodologies and identify the sophistication of the three methodologies on each feature.
4. Why do you think the ISP was left out of the process methods of Tom de Marco and Ed Yourdon? (You might refer back to Chapter 1's historical discussion for a hint.)
5. Object-oriented methodologies all ignore the front-end tasks of feasibility and data collection. Why? Can they continue to ignore those actions and still be useful in business applications? Why?

6. The Olle et al. framework was expanded to analyze the phases within each methodology where information is expected to become known. Describe this framework extension and identify, for data, processes, relationships, physical database model, and event triggers, where this information is known in each of the three methodologies.
7. What is the position of process methodologies with respect to data and data modeling? What is the significance of this position? How integrated is data to process description? What is the significance of this level of integration?
8. List three sources of application complexity. How does each source add to the complexity of an application?
9. Which methodology handles complexity the best and why? What is deficient about the other methodologies' handling of complexity?
10. To what extent do the three methodologies discussed guide input/output design? What is the significant of this?
11. Rate the three methodologies on desirable application characteristics: minimal coupling, maximal cohesion, and information hiding. Justify your ratings.
12. What is Humphrey's maturity framework? How is it used to assess IS organizations? How is it used to assess IS methodologies for application development?
13. What are three shortcomings of Humphrey's framework? How might they be eliminated?
14. List and describe the five levels of maturity in Humphrey's framework.
15. Do many organizations or methodologies reach the optimizing level of Humphrey's framework?
16. Describe the three methodologies in terms of Humphrey's framework.
17. If you have access to a CASE tool, use Table 13-6 to analyze the sophistication of your tool. List five ways in which the tool you use could be improved to contain more of the desired CASE features and functions.
18. Three issues in CASE are discussed: integration, intelligence, and multiuser support. How does the author view current products on the market? How does a CASE tool you use rate on these three criteria? What changes might be made to the tool you use to improve its integration, intelligence, and multiuser support?
19. Describe the research that seeks to integrate the best of all methodologies into a new, improved hybrid. Critique the utility of such a methodology and identify three of the problems with this approach. What benefits might accrue from a hybrid methodology? Why is it such a popular topic of research?
20. Describe the research that studies novice analysis of problems and relate this research to that which seeks to integrate the best of all methodologies into a new, improved hybrid. How can the analyst research be used to improve methodologies? What effect will hybrids have on novice learning?
21. What impact do legacy systems and data have on the use of new methodologies and CASE tools?
22. Define and discuss the issues of legacy systems and data.
23. Define a data warehouse and why companies are moving toward implementation schemes of this concept.
24. What is an associative data relationship and why does it impact data storage techniques?
25. Define client/server computing and downsizing. Discuss how they relate.
26. What is multimedia and how does it relate to application development and methodologies?
27. Describe some of the cultural issues in global information systems development.
28. What are the main issues in deploying global applications?

★ EXTRA-CREDIT QUESTION

1. Change the scenario for ABC Video. Assume ABC is an international organization that not only rents videos but also sells concert tickets, CDs, and other related entertainment and musical merchandise. What cultural assumptions are

in the case description of ABC Video that need to be reexamined for an application to be used in locations all over the world? What other changes might be required for worldwide use of the rental application? Don't concentrate on merchandise; concentrate on the cultural and equipment differences. If each of 3,000 stores in 60 countries send information to a single site in, let's say, Los Angeles, once each day, what technology considerations might be required?

CHAPTER 14

THE FORGOTTEN ANALYSIS AND DESIGN ACTIVITIES

INTRODUCTION

The forgotten activities of systems analysis are design of the human interface, conversion/implementation process, and user documentation. This chapter concentrates on human interface because the guidelines are not context specific and are based on research as well as practice. Rules of thumb for the other activities are discussed. Both the human interface and conversion are planned for ABC Video's rental processing application.

HUMAN INTERFACE DESIGN

The presentation of information for selection and data entry is the single most important design item in an application. The format, type, size, color, and content of the display all are important to a user locating, controlling, entering, or monitoring information. A badly designed screen makes a user tire faster, make more mistakes, and miss information that might have disastrous effects on decision making. Misrepresented data can have the same effects. The user's perception of the application and how it helps or hinders in performing his job is directly related to the human interface. If a user perceives the application as helpful and facilitating productivity, the application will be used with a high degree of satisfaction. If a user perceives the application as difficult, obscure, or reducing productivity, the application will not be used voluntarily and user satisfaction will be low.

Interface design is one of the most intensely researched areas of computing, yet much of the research has not found its way to business application design. In this section, we try to remedy that situation. First, the conceptual foundations of interface design are reviewed briefly. Then the options and guidelines for each major activity during interface design are presented. Following each section, we discuss how to apply screen design guidelines to ABC rental processing.

Conceptual Foundations of Interface Design

A combination of research, theory, and practice blend to provide the guidelines for interface design. In general interface design needs to answer questions about when, what, and how to enter data into, and present data from, applications.

First, when to collect data has been resolved through long experience and research. The ideal data entry point is at the data source. There should be no

creation and collection of paper from which data is then keyed into a machine. The more people who touch a transaction, the more errors it will have. Therefore, eliminate all middle men, enter data at its source, and errors are greatly reduced.

Second, which data to collect and display are also issues. The general answer, based on practice, is all data required for *business* reasons. Data may be expanded to include company specific requirements. Also, data items IS staff think might some day be necessary, but for which users have no current or future business need, *should not be collected or displayed.*

Last, and most complex, is how human–computer interactions should be structured and presented to ease learning, minimize errors, and facilitate use. Research and theory on physical and cognitive aspects of memory, information processing, pacing of work, color perception, icon perception, and keystroke effectiveness all are used to determine guidelines for interface design. The results of applying the research versus not applying the research are increased productivity and reduced errors. Since the research is so voluminous, it is presented in the context of the chapter.

With all the choices and research recommendations, deciding how to actually design functional screens can be a confusing exercise. In the next sections, practical guidelines from research and practice are developed. Information from the analysis phase is used to define the display requirements of the human interface. The analysis information is used to define a task profile for the application. Then, a profile of users is developed to identify screen requirements that relate to users rather than to functions of the system. The task profile is matched to guidelines for the application type to define and select the general interface as menus, windows, or commands. Application type also suggests functional screens as forms oriented, questions and answers, or direct manipulation. Once the general and functional interfaces have been defined, individual field presentation is defined and formatted for the screen. Finally, extra field characteristics, such as color, are decided and added to the design. Each of these topics is summarized below and addressed in the following sections.

1. Define task profile.
2. Define user profile and application design response.
3. Choose option selection screen type.
4. Choose functional screen type.
5. Design option selection interface.
6. Design functional screen interface format.
7. Choose field format options for normal, abnormal, alert, and alarm data conditions.
8. Design on-line user documentation, error messages, and abnormal processing for all interfaces.
9. Design reports as required.

Develop a Task Profile

Guidelines for Developing a Task Profile

The first activity is to develop a **task profile** which summarizes work requirements of the application. The level of detail in developing a task profile depends on the type of application being developed. The first task, then, is to classify the application as either transaction, query, DSS, ESS, or process monitoring and control (a special type of TPS). Since transaction processing is the most frequent application type in businesses, they are discussed here. The level of detail and activities for task profile develop-ment are summarized in Table 14-1 for the above application types.

For each activity, a hierarchy of processes is defined. This is the basis for screen navigation design. The top activities identified become selection options on a menu. Upon selection, the entries at the second hierarchic level are presented, and so on until a functional work screen is presented. The level of detail for the hierarchy should match the level of processing detail for the application type.

Next, required and optional data are defined for each task (see Table 14-1). For business applications, following the methodologies discussed in Chapters 7–12, required and optional data for entities should have been defined and documented in the data dictionary. For most business applications, this information can be developed at the entity/relation level rather than the attribute/field level. The idea is to identify multivariate dependencies which, in real-

TABLE 14-1 Task Profile Development Activities

Activity	Transaction	Query	DSS	ESS	Process Control
Define Task Hierarchies	Process Level	Activity Level	Activity/ Process Level	Activity/ Process Level	Process Level
Define Required/ Optional Data	Transaction/ Field Level	Entity Level	Entity Level	Entity Level	By input source
Define Data Precision	Only if greater than 2 decimal places for numbers	Only if greater than 2 decimal places for numbers	Only if greater than 2 decimal places for numbers	Only if greater than 2 decimal places for numbers	For each field
Define Data Source	Process/ Transaction Level	Activity Level	Activity/ Process Level	Activity Level	Field Level
Define Purpose	Entity/ Transaction Level	Entity Level	Entity Level	Entity Level	Field Level
Define Accuracy	Only if it varies from 100%	Only if it varies from 100%	Only if it varies from 100%	Only if it varies from 100%	Field Level
Define Domain	Field Level	Field Level	Field Level	Field Level	Field Level
ID Specific Display Criteria	Field Level	Field Level	Field Level	Field Level	Field Level

time systems, may need to meet synchronization and timing constraints.

If not already defined, precision requirements should be specified, by field, for all numeric fields (Table 14-1). **Precision requirements** specify the number of decimal places and special display characters required for numeric information. Precision is very important in mathematical, statistical, and process control applications. Precision beyond two decimal places is frequent in business applications dealing with large financial transactions. Banking applications, for instance, frequently require precision to five decimal places for computing interest due and paid. Specific maximum field size, need for sign (e.g., +), and need for debit/credit indicators [e.g., CR or ( )] should all be defined. For text fields, the maximum length should be defined, if not already done. Possible edit characters for numeric and text fields might be blanks, commas, or slashes. These definitions limit the number of data fields on a line while defining specific screen contents.

The source of data for each process should be identified next (see Table 14-1). Data source can be user-provided through data entry, measured data entry, or system-derived through computation. The key to identifying source, if it is unknown, is to determine where users go when they have a question about data on a screen. The answer might define a user, instrument type, or application as the information source. When user data entry is the source, training needs and help facilities are required to ensure proper entry. Edit checks for entry errors are required. When instrument measures are the source of data, the signal-to-noise ratio should be analyzed to determine the need for filtering devices or software. Fields for which the application is the source are

called derived fields for which data entry is not allowed.

Next, the purpose of every entity or field should be defined, depending on the type of application. Possible choices for purpose are forms completion, information, alert, or alarm. Business applications data purposes are usually form completion and information. Rarely are data items used to alert or alarm the user. Because alarms are rare in business, entity level checking is sufficient for all but critical applications. For each entity, then, the task profile identifies needs to send alert or alarm signals to the user based on data changes or system process outcomes. For critical and process control applications, each data element should have its purpose defined since the task of process control is to monitor changes in a system and correct any abnormal or undesired processes. Alerts to changed conditions and alarms to abnormal conditions are an integral part of process control interface design.

The need for accuracy for each task and, if less than 100%, of the data processed should be assessed. In business applications, this definition should be provided only when it varies from 100%. Typically, variation in business is in query or ESS applications for which ballpark numbers are acceptable for many types of processing.

For instance, a marketing person may want to target a product to one or more specific demographic groups. If the target mailing is 1,000,000 pieces, the marketer needs to know how many groups he needs to meet this goal. A sample based on selection criteria (e.g., age, education level, and zip code) can be used to project the size of the population ±5%. In this case, a 0.1% sample might be sufficient. Rather than read a 20,000,000 record file, only 20,000 records are needed.

The last two pieces of task information—domain and display criteria—are defined if not already complete. The domain is the set of allowable values for each field. Special display criteria might include translations of data to text (or vice versa), or a special color for some field, and so on.

All of these task characteristics are used to determine the type of interface in system terms, and to determine training needs for users.

ABC Rental Application Task Profile

There is no special complexity in the ABC rental application, so completion of the task profile is relatively simple. We are using the Information Engineering analysis in Chapter 9 as the basis for this discussion. The first action is to create the task hierarchy. Using activity level as the top of each hierarchy, we rearrange the processes as the next level and their subprocesses as the third level, continuing until all processes are elementary (see Figure 14-1). This diagram is the basis for navigation between screens. Each leg of each level on the hierarchy is translated eventually into a menu selection list.

As of the analysis, all data were required for all entities (see Table 14-2). Precision for money fields is two decimal places. All other numeric fields are dates or integers. The source of *Customer*, *Video*, and *Copy* data is user data entry, so extensive edits will be needed in the entry programs to ensure that only correct data enters the system. *End of day*, *Video History*, and *Customer History* are all derived by the system and have no human interaction. The derived relations identify testing requirements for specific verification. The *Rental* relation is a combination of entered and derived data which identifies both edit and testing requirements.

The purpose of the entities is either forms completion or information with one *Rental* relation exception. The credit field will be used to deny rental privileges to customers who have a poor credit rating. Some special processing may be desired to highlight bad credit ratings. The possibility of highlighting bad credit rating information should be discussed with Vic and his approval obtained. No decision is made at this time.

Accuracy for all maintenance, rental, return, and query tasks is assumed to be 100% (see Table 14-2). If Vic, while performing ad hoc querying, chooses to sample the data rather than read the entire database, that is okay, but not of interest for this definition.

The domains of each field are in the data dictionary. No special display criteria are identified at this time.

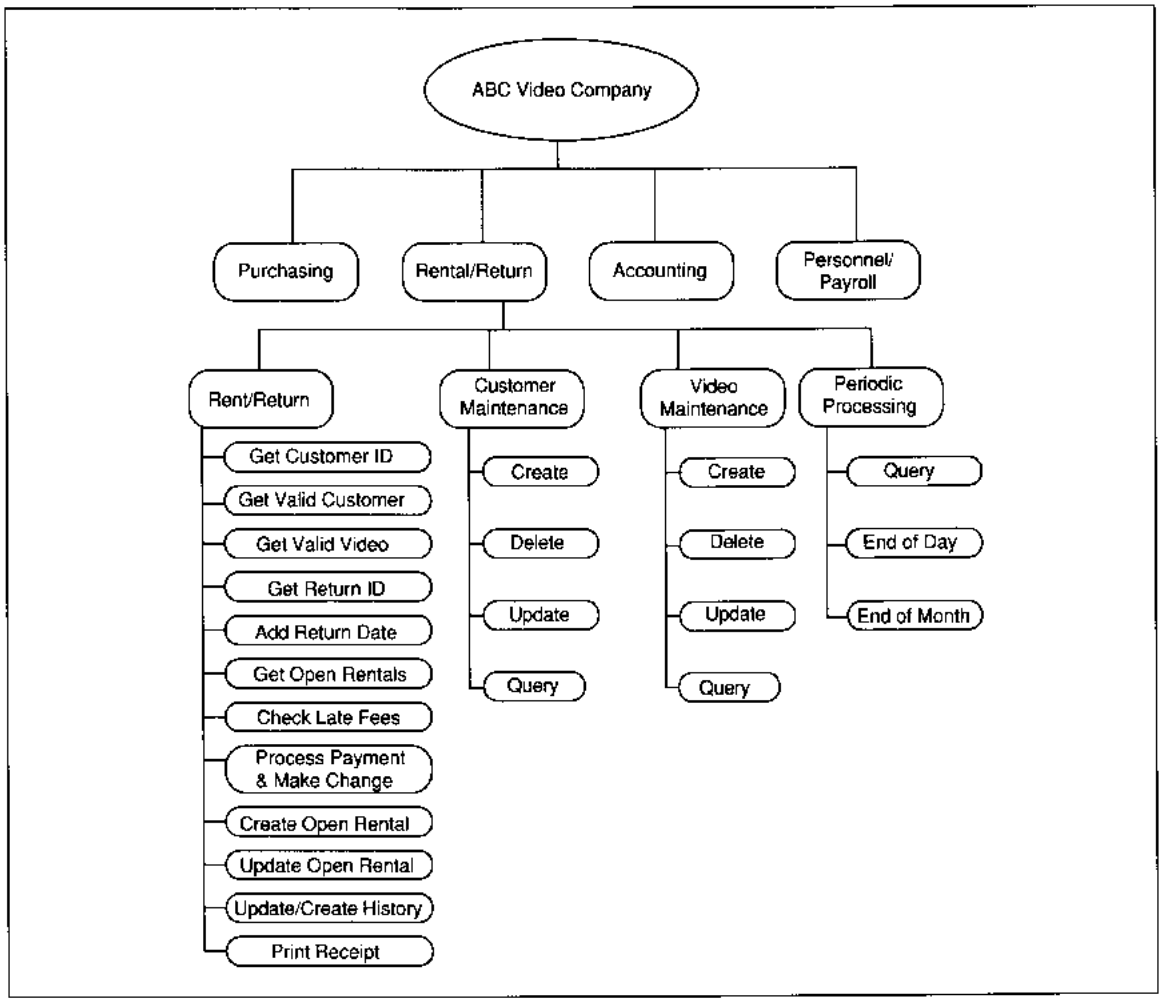


FIGURE 14-1 Process Hierarchy Chart for ABC Rental Application

Develop a User Profile

Guidelines for User Profile Development

A **user profile** is developed to determine the need for special interface design requirements that relate to the user rather than the task. User profile criteria include physical, educational, computer, and task capabilities (see Table 14-3). At the same time the user profile is developed, a matching profile for the application and how it will address the user needs is also developed.

Information in the user profile is obtained from users through interviews, questionnaires, or personnel file searches. If personnel file searches are performed, only average ratings of user skills should be computed unless each employee gives permission to use his or her information. Use of employee records for other than personnel purposes without permission is considered an unethical violation of privacy rights.

TABLE 14-2 ABC Rental Task Profile

Activity	Transaction	ABC Rental
Define Task Hierarchies	Process Level	See Figure 14-1
Define Required and Optional Data	Transaction/Field Level	All data required
Define Data Precision	Only if greater than 2 decimal places for numbers	None. Dollar amounts have 2 decimal places.
Define Data Source	Process/Transaction Level	User Entry and Derived, See Data Dictionary
Define Purpose	Entity/Transaction Level	Form Completion, Information
Define Accuracy	Only if it varies from 100%	100%
Define Domain	Field Level	See Data Dictionary
ID Specific Display Criteria	Field Level	() Required for Change field negative values. No other special requirements.

In critical applications with possible life threatening consequences, *each individual user* should be profiled and reviewed for proper skills, computer experience, and task expertise before being assigned to use the new application. Education can take care of some deficiencies in skill levels, but with some critical applications, people may be reassigned to other jobs when their knowledge does not match the application requirements. For noncritical applications, the profile can *average* user skills for each characteristic. User profile is used to determine sophistication of the interface and training needs.

Physical skills include color perception, typing skill, and physical disabilities that might be present in the user population. Color perception problems mean that reds and greens might not be perceived. If colors are used, users should be screened to ensure that they can recognize the selected colors. Also, color selection should relate to conventional meanings for each color used. For instance, red is the usual alarm-signaling color. In an application using red to signal an alarm condition, then, all users should be screened for their ability to perceive the color red.

Typing is the other typically used physical skill. If user typing skills are low, either the application must be designed not to require typing, or typing training should be provided to users.

Education and math profiles can be either individual or average analyses (see Table 14-3). Education level determines the level of writing required to explain errors. For math-intensive or numerical control applications, specific math skills might also be necessary of users. When this is the case, math skills needed are defined for each task (e.g., one task might need algebra, one might need the ability to interpret geometric drawings, and so on). Users whose profile does not match the required skill levels are trained or reassigned. Many companies, such as Texas Instruments, Chevron Oil, and others, retrain their employees in math skills needed to manage complex computerized manufacturing equipment.

When the average education and math levels are lower than high school-graduate level, the application interface must be designed as simply as possible. Instructions and text help must be written using sentences under 25 words and use words averaging less than three syllables. Different indexes can be

TABLE 14-3 User Profile and Application Response

User Characteristic	Description	Application Response
Physical Skills:		
Color Perception	Red/Green/Blue Color Perception	Either design application without the problem colors or reassign the users.
Typing	Ability in words/minute	Either design the application to fit the skill level or schedule typing training to increase skill level.
Disabilities	Sight, hearing, or physical impairment that might change application hardware, software, or interface design	Either design application to accommodate impairments or reassign the users.
Educational Skill:		
Education Level	Average or actual level of highest degree	For both education and math, design application help and training to ensure users can learn and use the application.
Math Proficiency	Average or actual level of math proficiency	
Language		
Native	All native languages not the same as intended implementation language	International applications should use language native to the region for the application interface.
Proficiency with application language	Average or actual level of proficiency	Training and text descriptions in application can be no more difficult than the average level of proficiency. Training should be provided to ensure that all users attain the average level (i.e., the average becomes the minimum).
Computer Proficiency		
Average Proficiency	Average or actual level of proficiency in years of experience	Design the application help, messages, and user documentation to ensure understanding of all functions, messages, and menu options.
Number of packages	Number and type of packages with which users are familiar	Define training method and requirements. Define level of supervision after training is complete.
Job Characteristics:		
Turnover	Average % new employees per year	Determine interface option selection type.
Experience	Average years task experience	Determine level of help and location (automated vs. manual and immediate screen message vs. requested help).

used to compute reading level of text. For instance, the software RightWriter©,[1] provides the Kincaid reading grade level (scale of 1–16), Flesch index of readability (scale of 1–10), and a fog index (ratio of nouns and verbs to total words in a sentence) as measures of text difficulty.

Information about native language is important to determining the language of the interface. As globalization of the economy and development of global organizations increases, the need to implement the same system worldwide will become commonplace. When applications are implemented in other countries, the native language should be used as much as possible. From research we know errors are reduced and some user satisfaction comes from working with applications in one's native tongue. Sometimes, this requirement is government imposed. For instance, in the early 1980s, the King of Saudi Arabia declared that as of 1990 all communications, documentation, and application interfaces used in the kingdom would be in Saudi language. This posed a tremendous challenge to every company doing business with Arabia because Arabic is read right to left, frequently omits vowels, and has as much as 50% of every sentence in a local dialect. At the time of the declaration, there was no one recognized Arabic dictionary for the Arab world. Rather, each country had its scholars map the language for their country. In general, the more critical the application for controlling some potentially catastrophic process, the more important native language processing becomes. I would not like to think of a person who barely speaks English as the controller of a nuclear power plant with all systems and manuals in English!

Next, computer experience is profiled. The average and range of number of years experience, number of software packages used, type of software (e.g., spreadsheet), and whether the individual develops his or her own software are all important to know. The level of computer experience, coupled with the skill level required of the application, determine the type of training that is most effective. For applications that are complex, critical, or have many variable activities, classroom and hands-on training would be indicated. For applications that are simple and have few activities, classroom, computer-based training or on-the-job training are sufficient. Assignment of new staff on the job might require close supervision for a period of time to ensure that they possess the skills to use the system properly. Close supervision should be used for all critical applications regardless of complexity or method of training.

The level of task turnover in the next rating category determines which of the training methods is actually used. If turnover is low, classroom or computer lab training reach the most people at once and are the cheapest. If turnover is high, some method of individual training is required. Some alternatives for individual training are on-the-job, programmed instruction manuals, or computer-based programmed instruction. All can be effective means of training.

Finally, task experience is estimated. If the average user has a high level of task experience, the labels for fields can be more abbreviated, less text is needed to guide data entry, and an expert mode of operation might be preferred. If the average user has a low level of task experience, or experience is variable, novice and expert modes might both be needed.

Task experience and turnover information together determine the mode of interface as novice or expert, and the extent to which on-line help should be provided. Figure 14-2 shows that with low experience levels, novice-only modes are required. With a high experience level, either a mixed mode or an expert mode-only are required.

Figure 14-3 shows that the type of message and extent of on-line assistance also varies with experience and turnover. Low experience with low turnover suggests use of meaningful text error messages with on-line help to elaborate on the error messages. With high turnover, the on-line help should include information on menu options, fields to be completed, and error messages for data entry errors. With high experience levels, the on-line messages can be abbreviated (or eliminated with use of a beep instead of any text message), and with high turnover, supplemented with a paper manual documenting errors and error recovery.

Last, effective training for the application type, user education level, and experience level can be

1 RightWriter is a copy-protected product of RightSoft, Inc.

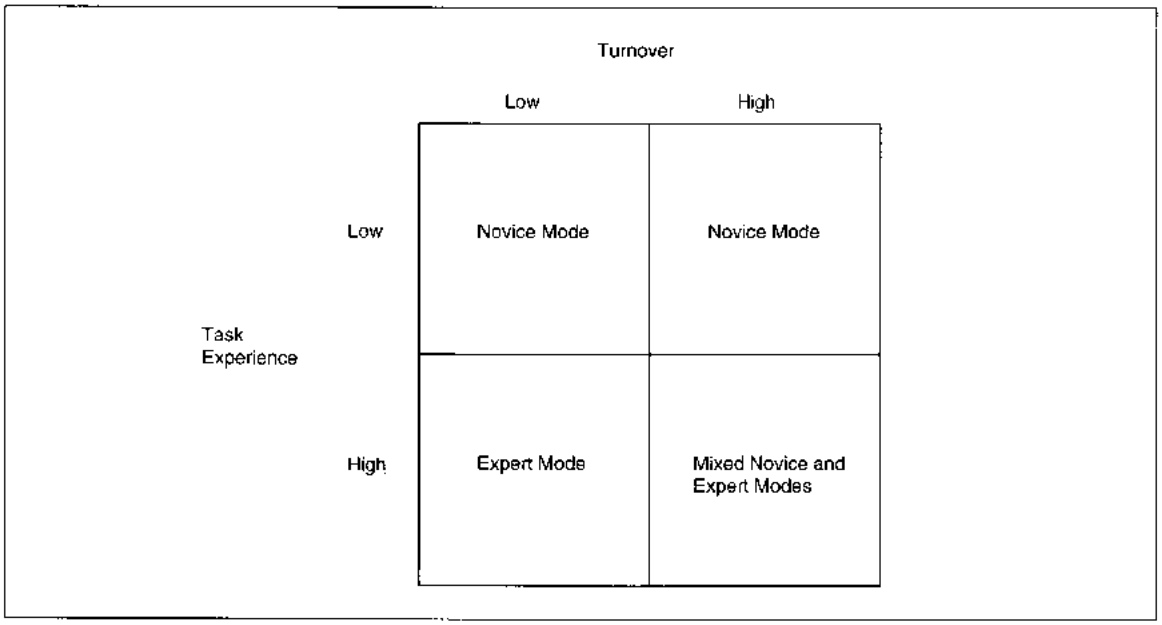


FIGURE 14-2 Turnover and Task Experience Determine Mode of Processing

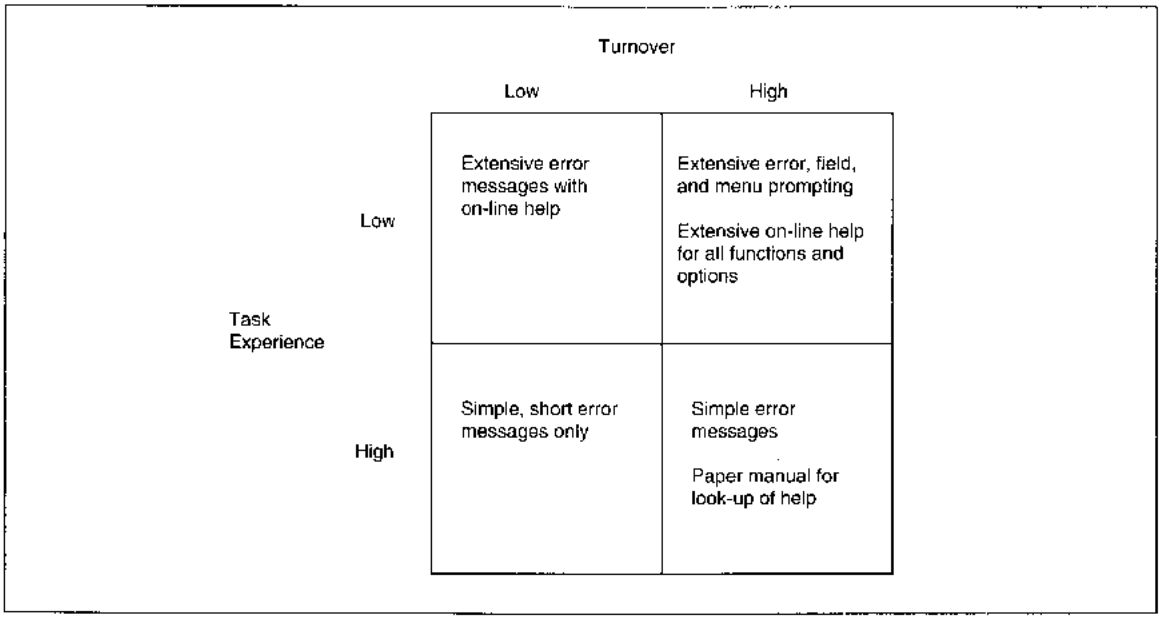


FIGURE 14-3 Turnover and Task Experience Determine Level of On-line Assistance

decided. Training choices include **classroom instruction, computer-based training** (CBT), or **on-the-job training** (OJT). Classroom training is the most cost-effective for groups of students. Students can ask questions and receive personalized training while a number of people are being trained simultaneously. The disadvantages of classroom training are high cost and the fact that training cannot be repeated without additional cost.

CBT is most effective for training one or a small number of people simultaneously and at different rates. CBT is self-paced, low pressure, and does not require a senior person to monitor the training. The major disadvantage of CBT is its cost, which is steadily dropping. Much training in business will be computer-based by the year 2000 because, by then, it will be cost-effective for most business uses.

On-the-job training is cheap but requires a senior person to teach trainees. The senior person is assumed to be a good teacher who can explain all necessary variations to someone else. These assumptions may not be valid. If OJT is used, some manager or senior staff person should monitor training and privately correct the teacher if a problem arises.

ABC Rental User Profile

Video stores hire younger people, who are frequently in high school. The turnover is high because it is part-time work with mostly evening hours (prime date time) and because the business is somewhat cyclical in video rental patterns. Since the specific users are not known, the average user is estimated based on the four current ABC employees. The analysis is summarized in Table 14-4.

In the ABC example, current employees have no physical impairments and none are anticipated. Typing skill is expected to be low. No particular prohibitions on color or special equipment will be needed except to compensate for the lack of typing skills.

The application will use a bar code reader, as suggested by Vic, to replace the need to type most information. The bar code reader minimizes the key strokes required of users. The reader will scan user IDs, if they are used, and video bar codes to enter the information to the computer. If user IDs are not used, the phone (or other ID) number will be typed. Another typed entry is the total amount paid. This should not be too error prone because most people pay in even dollars, receiving change. If the need to enter a few numbers really worried Vic, user ID cards can replace the need to type user IDs, or, alternatively key pads are less error prone than typewriter keyboards and could be used.

The average education and math levels of employees is expected to be at the 10th-grade level. This means that algebra is the most abstract level of math skill. The system design criteria are KISS—keep it simple, stupid—so the 10th graders can do the work easily. The math level should be acceptable since the only skills required are to enter the amount paid and to make change.

The language of employees and the language of the application is expected to be English.

Task turnover is high and task experience varies from low to high. Vic has one employee who has worked there four years and two who have been there two months. The task experience of the longer employee is significantly greater than the other two. While the video rental business is not complex, the two newer employees cannot be expected to perform all functions. The system design criteria in response to high turnover and variable task experience is to provide a simple interface with message help on request for all selections, fields to be completed, and error messages.

Computer experience is also expected to be variable but generally low. Number of years' experience for the three employees ranges from zero to two years. Number of software packages ranges from zero to three. The software used is word processing by two people, and database and spreadsheet by one person. One person wrote his own software.

With little computer experience, high turnover, low task experience, low task complexity, and 10th-grade education, two alternatives are recommended. First, individual, self-paced, computer-based instruction (CBT) is recommended because the students can come in on their own time to train whenever it is convenient. When the store is not busy, they might continue their on-the-job training using the CBT. The method would be to give the person one each of the different transaction types. The

TABLE 14-4 ABC User Profile and Application Response

User Characteristic	Description	Application Response
Physical Skills:		
Color Perception	No Problems	None
Typing	Less than 15 WPM	Design to minimize data entry by using bar code reader for Video ID, Copy ID; data to be entered Customer Phone, Amount Paid
Disabilities	None	
Educational Skill:		
Education Level	10th Grade	On-line help
Math Proficiency	Algebra	Needs no special design. Users must be able to make change.
Language:		
Native	English/Spanish	None unless Vic wants to verify user ability to read all display text
Proficiency with application language	High	English will be the implementation language.
Computer Proficiency:		
Average Proficiency	Low, 0–3 yrs. Average = 1 Yr.	Training in basic computer skills, startup, shutdown, etc., required.
Number of packages	0–3, Lotus, WP	
Job Characteristics:		
Turnover	65% Yr.	Use extensive on-line help for all options, entry types, data types, forms fields. Provide expanded on-line help to supplement messages for errors.
Experience	Low to High, Average = Low	Provide extensive training in all transaction types, beginning with turning on the machine. Monitor performance for first week on the job to ensure that training was sufficient.

person would enter the information and the computer would automatically do all subsequent processing. Then, the person would do several of each type of transaction completely. The system would intercept their entries and prompt them for correction, displaying reasons for the correction when they made errors.

Second, if CBT is too costly, on-the-job training (OJT) with a senior person monitoring and assisting the trainee should be sufficient. If this is the chosen alternative, the trainees should learn rental and return processing first. This can be followed with less important tasks after several days. If OJT is the preferred training method, Vic should monitor the trainer(s) and trainee(s) closely for several days to ensure that the trainers cover all alternatives, pace the instruction to fit the person, and make no assumptions about the trainees' skills.

Option Selection

Once the user profile is complete, the general form of the human interface is decided by mapping the user and task to the implementation environment. When this activity is complete, all interface recommendations are presented to the user for discussion and decision. Two choices are made from the mapping of user and task to implementation environment. Either or both of the choices may be constrained by particular hardware and software if these are already known. The choices are for general option selection screens and general functional screens. Each of these are summarized in Table 14-5. Each set of alternatives and guidelines is followed by a description of how to apply the information to screen design for ABC rental.

TABLE 14-5 Summary of Interface Choices

Interface Level	Alternatives
Option Selection	Menu Window Command Language
Functional Screen	Form Question & Answer Direct Manipulation
Data Presentation	Analog Binary Digital Bar Chart Column Chart Point Plot Pattern Display Mimic Display Text Text Form
Screen Item	Color Size Type Font Type Style Blink

Option Selection Alternatives

Choices for interface **option selection** design are menus, command languages, and windows for getting *to* some functional screen. **Menus** are lists of items from which a selection is made. **Command languages** are high-level programming languages that communicate with software to direct its execution. **Windows** are a form of direct manipulation environment that combine full screen, icon symbols, menus, and point-and-pick devices to simplify the human interface by making it represent a metaphorical desk environment.

In general, menus and windows are novice modes of operation, while command languages are expert modes. Windows are the interface design most recommended because they simulate an office desk and present the most familiar interface to users. The next section presents design guidelines for the selection level of processing. Details of design for menu and window design are presented in the following sections.

General Option Selection Guidelines

General design guidelines relate to the development of a consistent, standardized interface, consisting of a header, a body, and a footer (Figure 14-4). The screen may include error message lines and command entry lines as well. Many companies have standards for screen design, so much of the work is already complete.

The **header section** of the screen should contain an identifier of the application, function, date, time, screen ID, and program ID. An example is shown in Figure 14-5.

The **body of the screen** contains variable information (see Figure 14-4). In hierarchic menu processing applications, the body contains menu selection, forms for completion, graphics output, or graphical monitoring measures. The body of the screen is subject to many other guidelines which are discussed in the next section.

IBM standards also suggest a user message line and an error message line (see Figure 14-6). Defining user commands and error message lines as fixed may take too many lines away from the screen, so these are optional.

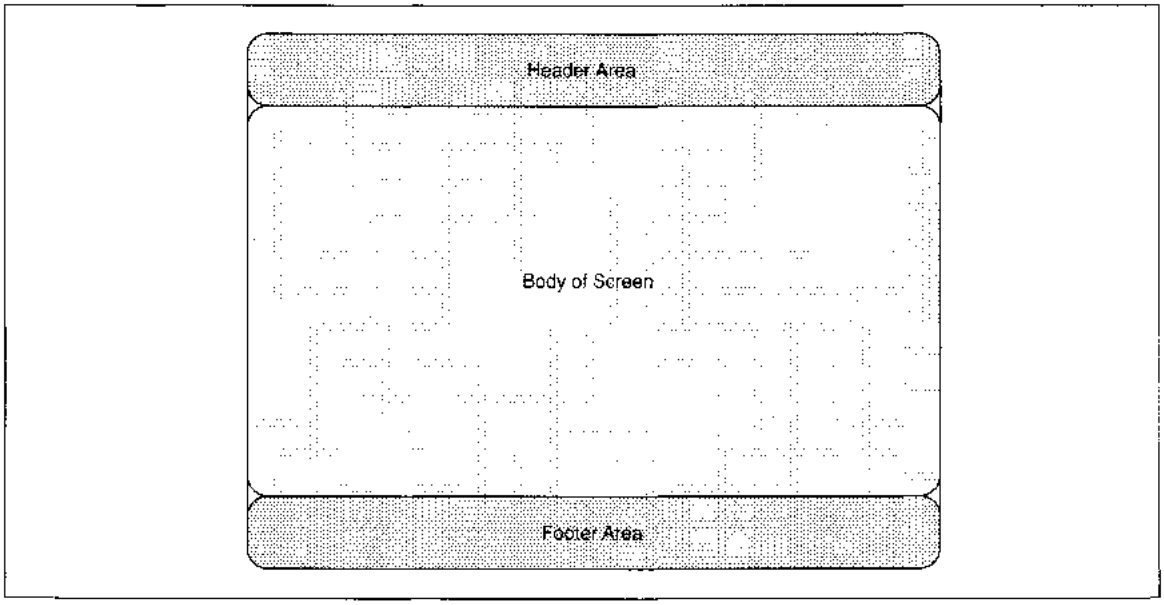


FIGURE 14-4 General Screen Design

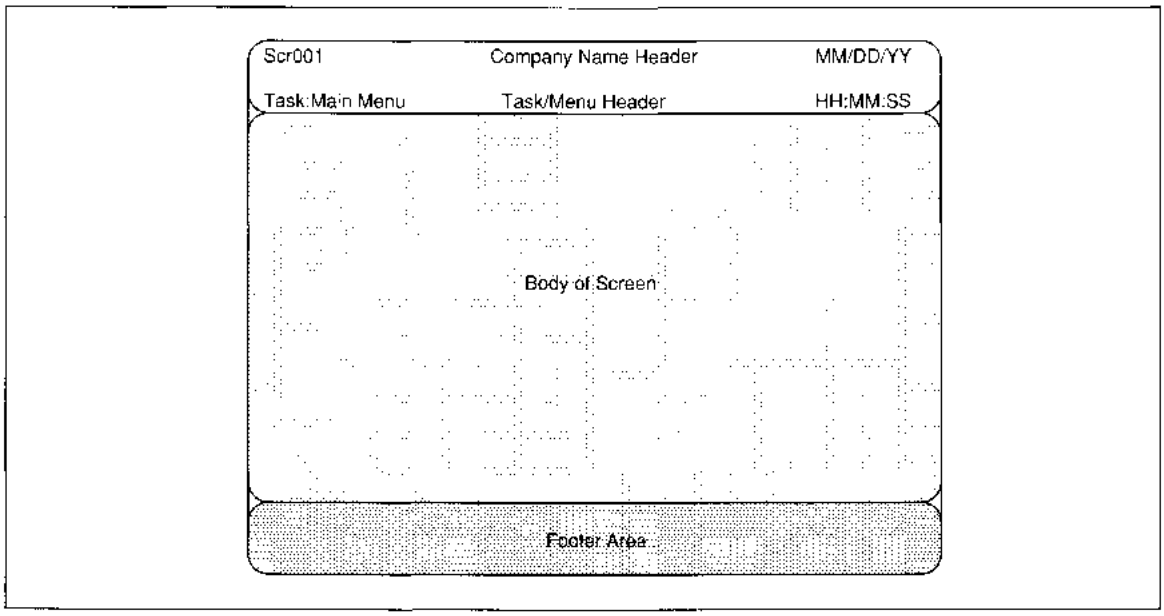


FIGURE 14-5 Screen Header Example

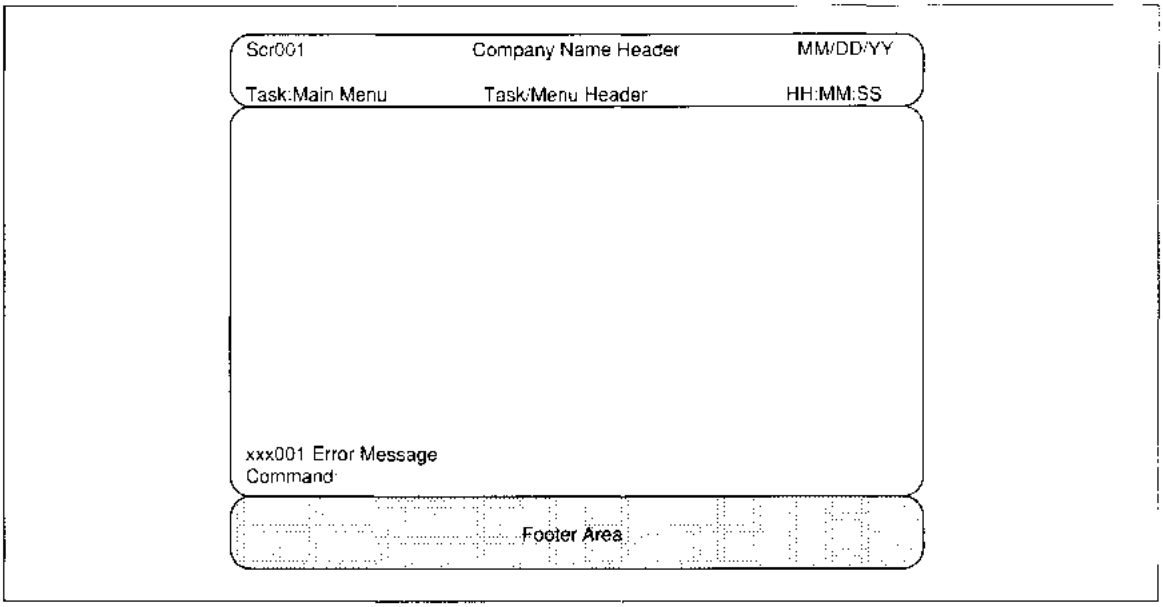


FIGURE 14-6 Command and Error Line Examples

The **footer screen section** contains indicators of navigation choices. Navigation choices should identify which key to select for each allowable movement option. Movement can be within a screen, between screens, or between menus and functional screens. Usually, screen navigation actions are taken by using special keys: escape (ESC), delete (DEL), or programmed function keys (PF or F keys). The allowable actions should be identified at the bottom of the screen in a manner similar to that shown in Figure 14-7. The identifiers should always contain a connector (such as colon) between the key label and the action label. The action labels should be concise, clear, and consistent across the entire application (see Figure 14-7). Ideally, only actions allowed from the current screen should be shown. Others might be blanked out or muted to indicate that they cannot be chosen here.

Menu Standards

The research on menu processing has given us guidelines for location and ordering of menu options. User/SE choices prevail for menu option names and option selection technique. First, based on the number of items on the menu, location is decided. If the number of options is less than 10, the items should be centered as a left-justified list of options. If numbers or letters are assigned to the options, they should be right-justified, followed by a period, and two spaces to the left of the corresponding choice (see Figure 14-8).

When the number of options is 10 or greater, you should experiment with different layouts to make the menu simple and easy to use. If the options are all independent, separating sequences of four or five options by blank lines enhances understandability (see Figure 14-9). If list options are interrelated, then experiment with segmenting the screen into different areas with each area containing an area ID and a centered, justified list of options for the area (see Figure 14-10).

The options for menu selection are entry of an option ID without cursor movement, point and pick, or entry of an option ID with cursor movement. Either of the first two are recommended and selection should be based on user preference (see Figure 14-11). The third option requires more key strokes

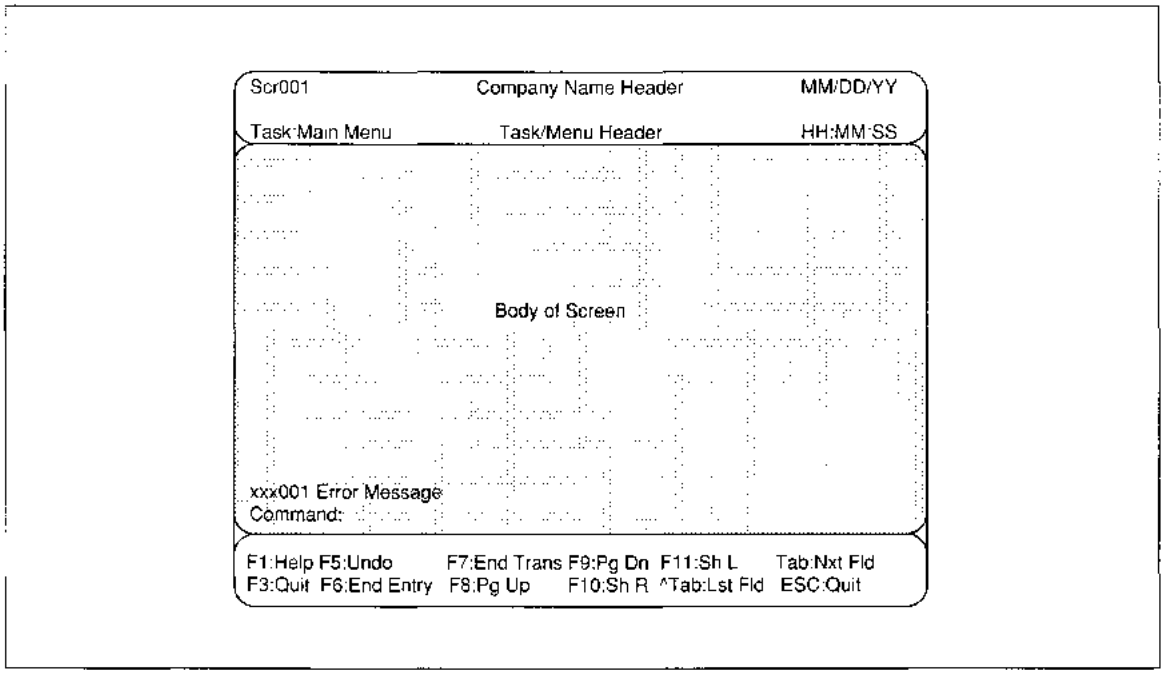


FIGURE 14-7 Screen Footer Example with Function Keys

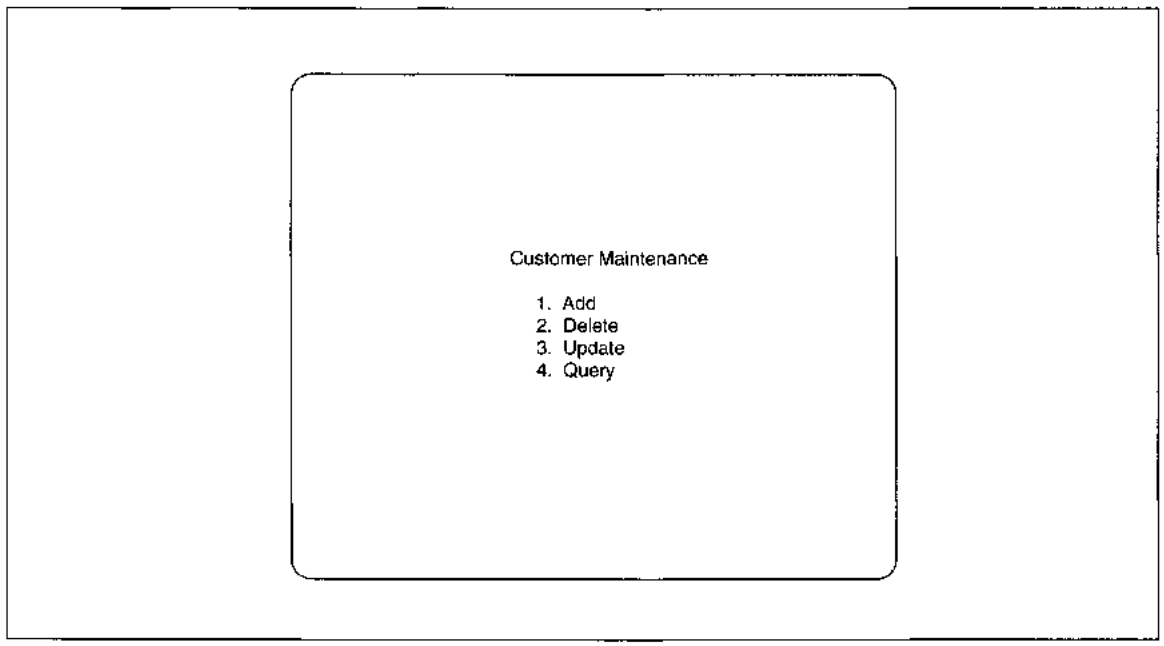


FIGURE 14-8 Numbered Menu Option List, Less than 10 Choices

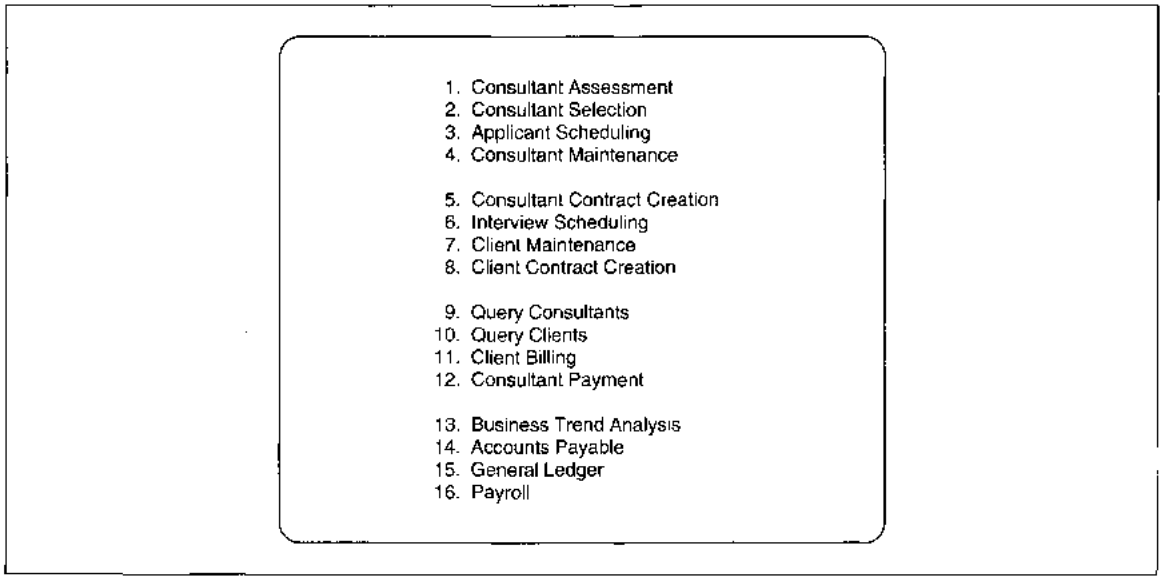


FIGURE 14-9 Menu Option List, More than 10 Independent Choices

and is more error prone; therefore, it is not recommended. Option IDs can be alphabetic or numeric; alphabetic options can be the first letter of the option or letters assigned from the alphabet in sequence. Again, there is no one right answer and user preference should prevail. If a point-and-pick

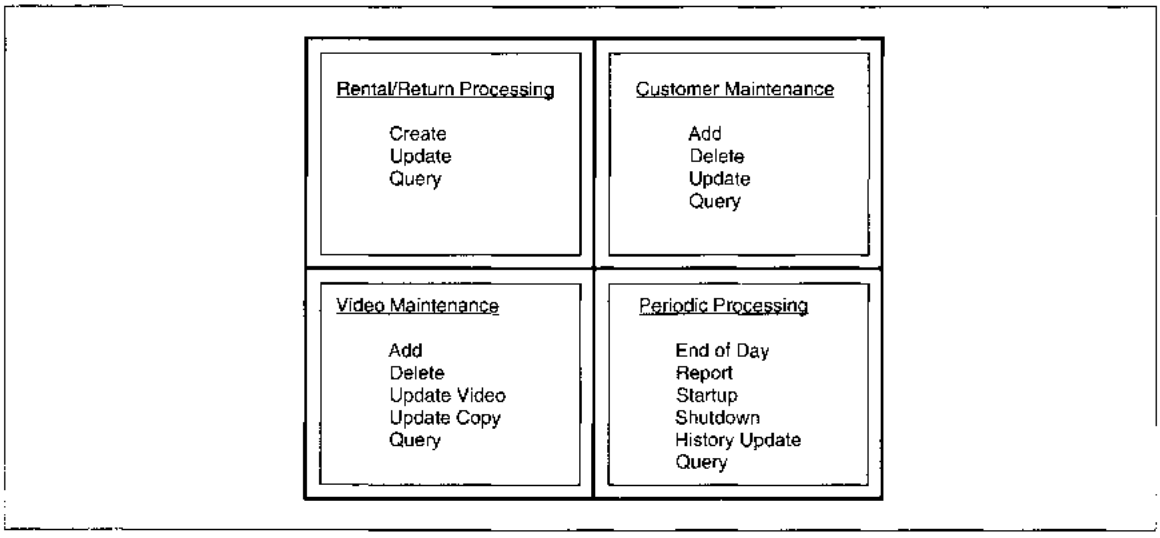


FIGURE 14-10 Menu Option List, More than 10 Interrelated Choices

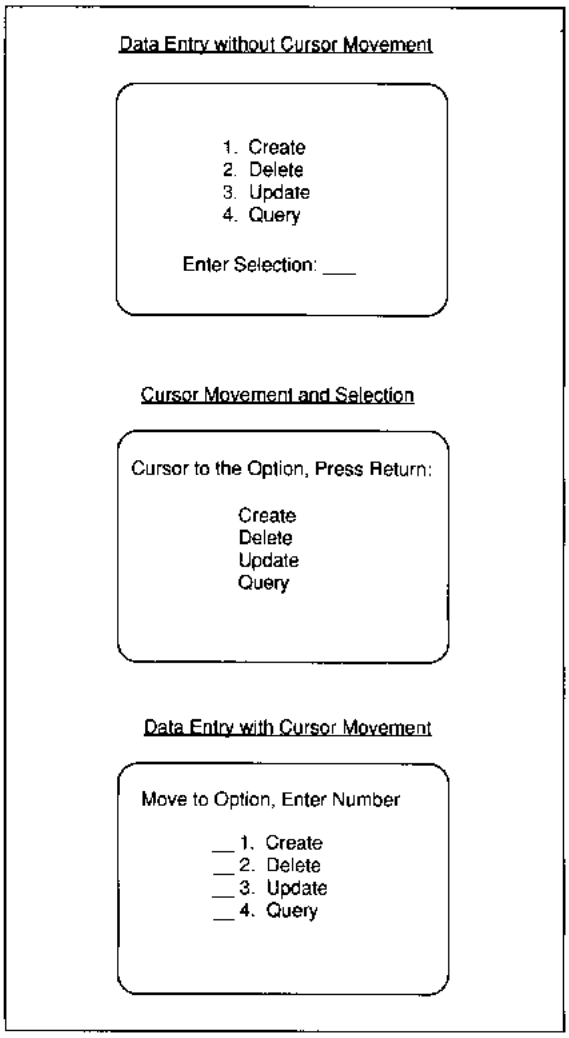


FIGURE 14-11 Menu Selection Options

device, such as a mouse, is used, no option IDs are required.

In all cases, when entry of a selection option is used, the message requesting the data entry should be centered on the screen, two lines under the last menu item, and should be in this location on all screens. This means that the location of the entry line should be two lines under the longest list in the entire application, and that it is always displayed on that line.

The listing of options within the menu should be based on frequency of choice when point-and-pick selection is used, and should be based on alphabetic order of choices when entry of a selection ID is used. Frequency listing is used for point-and-pick selection because the cursor should be positioned automatically at the most frequent choice (see Figure 14-12). The positioning by frequency of use minimizes keystrokes when moving to other choices. Alphabetic sequence of choices is used when a selection ID is entered, because users can read and understand an alphabetic list faster than a random list (see Figure 14-13). Both alternatives assume a novice user who does not know the options from memory.

The last issue in menu design is option names. Some authors[2] recommend specific names even if it means repeating some information (see Figure 14-14). Other authors[3] recommend concise but meaningful names with no repetitive information (see Figure 14-15). Combining these guidelines, we can design screens that are easily understood and used. First, the option names should be listed to completely define the process and entity(s) (as in Figure 14-14). Then, any information repeating in *all* entries should be removed and placed in a header for the menu list (see Figure 14-16). The result is the concise list from Figure 14-15 with a short header providing the additional information from Figure 14-14.

To summarize, menu applications should be designed in the context of a standard screen format that is used throughout the application. Menu items should be centered, selection action should be obvious, and minimal information should be in the body of the screen.

Window Standards

Windows are rectangular screen areas used to display information. Window displays differ from

2 For instance, Banks & Weimer [1992].

3 For instance, Galitz [1981]; Thomas [1982].

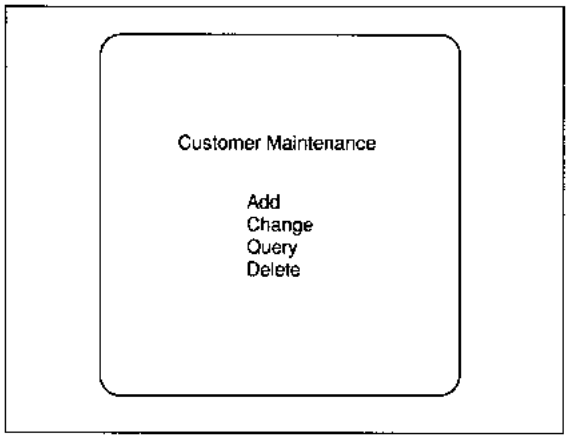


FIGURE 14-12 Menu Options Listed By Frequency

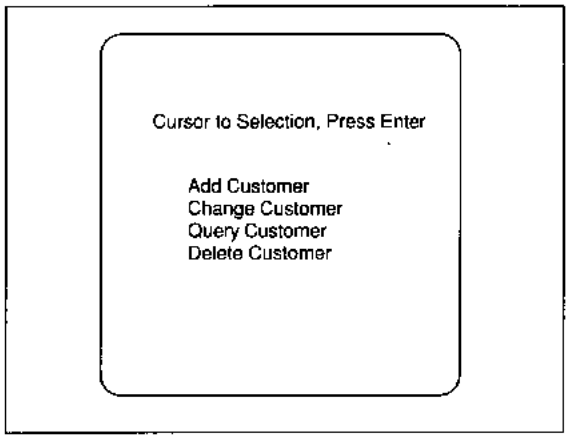


FIGURE 14-14 Complete Menu List

menu-driven full screen displays because users can view different, possibly unrelated information at the same time in different windows. For instance, in ABC's rental application, we might be looking for rental information for Sarah Cropley. We can begin a query function, then type, for example a '?' in the Customer Name field to indicate a look-up. A new window opens up and shows customer names. We select Sarah Cropley, the window closes, the name is moved to the first window, and we continue the query. Look-up and selection of information from a window is simpler than a menu system which uses the entire screen for one thing at a time. Because windows are different from menus, they have different guidelines and standards for their use.

A typical window can have the components shown in Figure 14-17. A **Close Box** stops processing and is similar to an F3 key use defined for a menu. The **Title Bar** names the window the same as the header line in the header portion of a menu. **Location ID** and **status indicator** identify where the user is in the window and whether or not processing

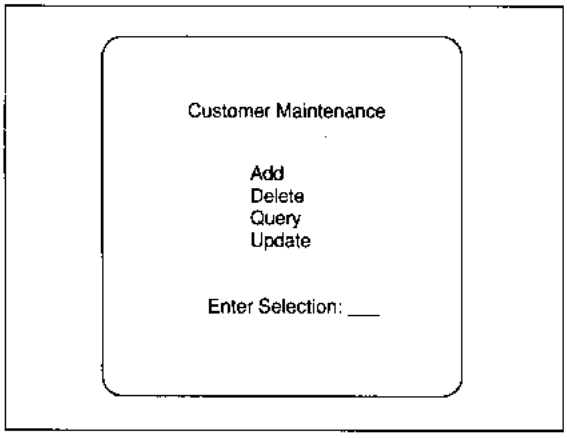


FIGURE 14-13 Menu Options Listed Alphabetically

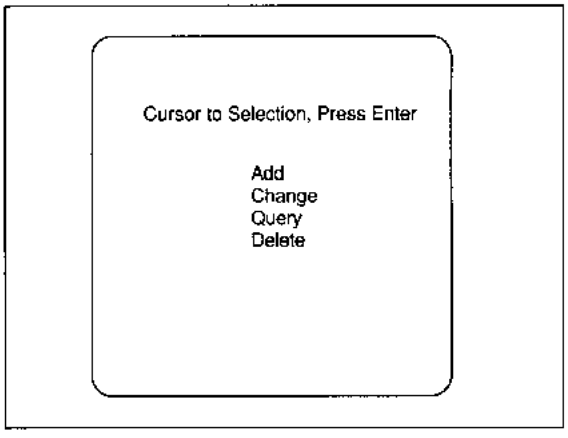


FIGURE 14-15 Concise Menu List

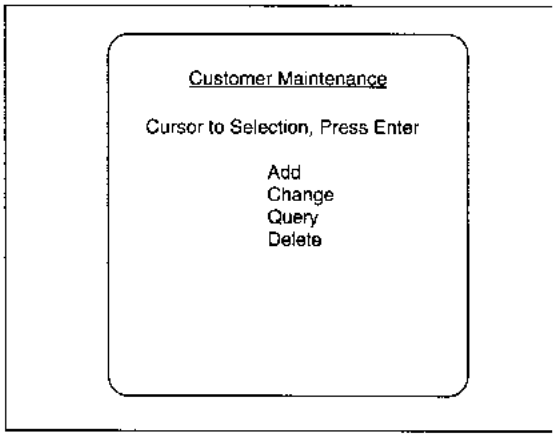


FIGURE 14-16 Combined Menu List

is normal. The **zoom box** and **resize box** both are used to change window shape. Zoom toggles between current size to full screen and back. Resize allows the user to customize the desired width and height to the window. **Scrolling elements**, arrows, bars, and boxes are used to move vertically and horizontally in the window, and are similar to function keys F8-F11 we defined for the menu system. A **scroll box** is dragged to move a variable distance, while a **scroll bar** pages up or down depending on where it is touched, and a **scroll arrow** moves one line at a time. Most window elements are available for use in a windowing application, such as Paradox, but usage is selected by the programmer. All are recommended if the application contains multiscreen forms completion. At least one type of scrolling element for each dimension should be provided.

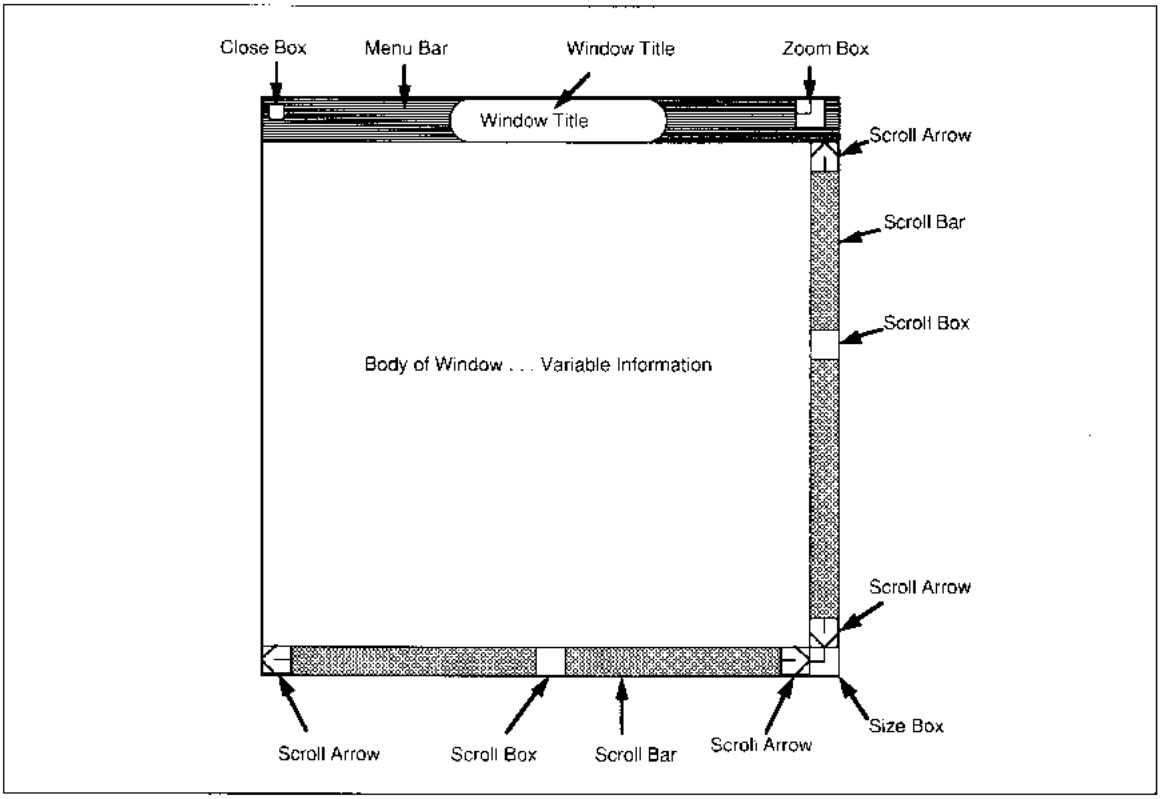


FIGURE 14-17 Window Components

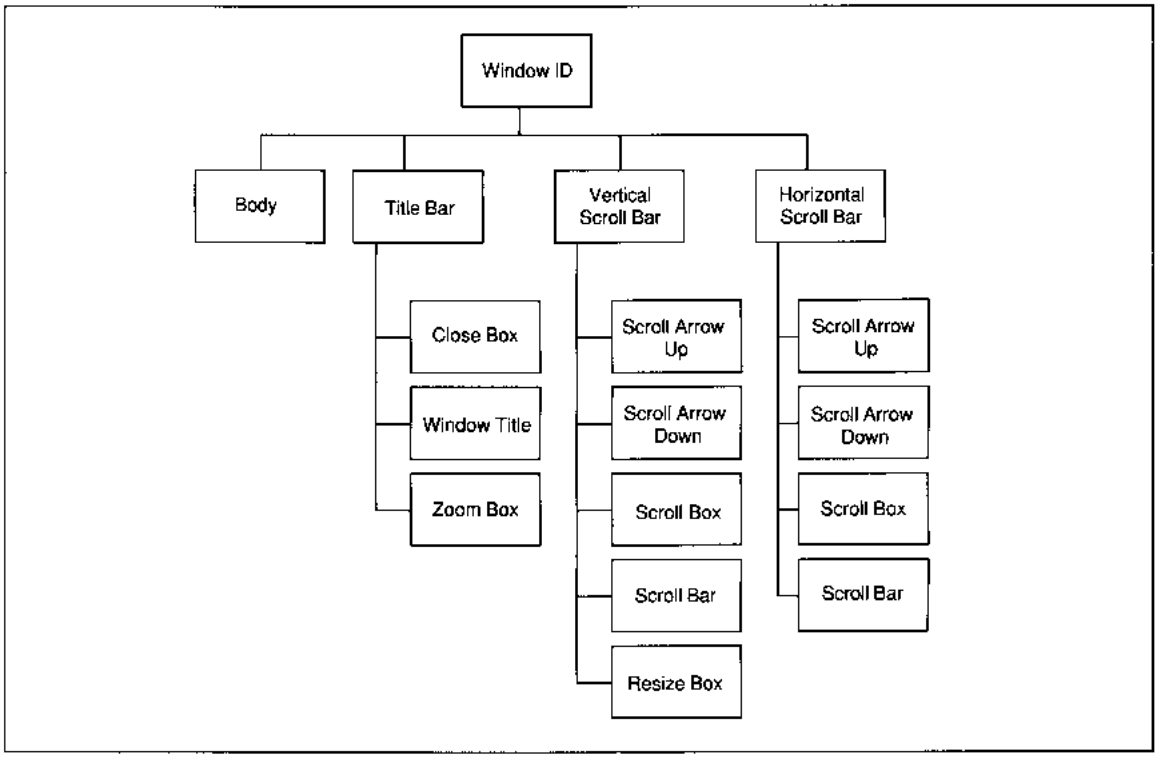


FIGURE 14-18 Window Component Hierarchy

Windows have two basic varieties: tiled and overlapping. **Tiled window systems** only create nonoverlapping windows. These work best for process control and nondata intensive applications. When many functions and types of data may be active at once, overlapping windows might be desired. **Overlapping windows** layer windows as opened, one on another, until the application maximum. To move from one window to another, the user clicks on the edge of the desired window to bring it to the front of the stack.

Windows are defined as hierarchies of objects for management. Figure 14-18 shows the hierarchy for the window components in Figure 14-17. As new windows are opened, a new hierarchy is built. All of the window hierarchies are managed by a screen manager which links all hierarchies.

Windows should be set off from each other and from the background by thick, easily recognized borders. Tiled windows should provide blank space, if it is available, between windows. In current windowed systems, the user has little choice about positioning of selected options for title bar and scroll bars, for example, but, if choice is allowed, the design should be consistent in all tasks. One of the best features of the Macintosh environment is that Apple Computer requires any software operating on the Mac to use exactly the same interface definition as the Apple operating system. All software seems familiar before it is even used. Finally, if no other features beyond a window space are used, scrolling to allow viewing of all window accessible information should be provided.

Window menu styles include horizontal pull-down, Lotus-style horizontal pop-up, and vertical pop-up. **Horizontal pull-down menus** show the top-level selection choice across the top of the screen, taking the least screen space of all menu options (see Figure 14-19). When a menu is activated, by having the cursor moved to its location, the

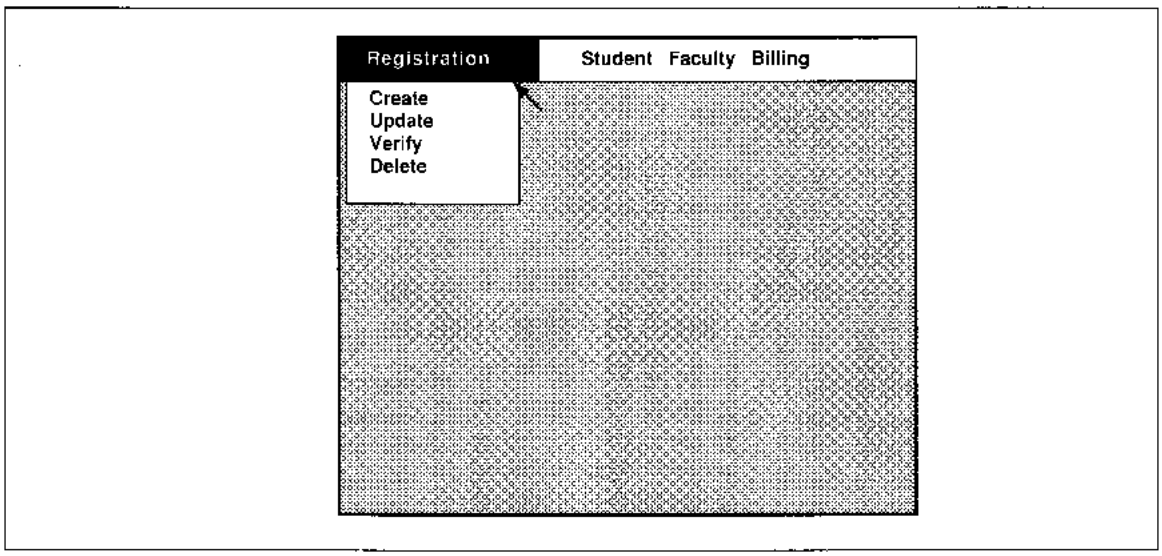


FIGURE 14-19 Horizontal Pull-Down Menu Example

second-level menu is *pulled-down* from the original entry. To make a selection, the cursor is moved to the desired option and activated. Activation is either through a return key or by pressing a mouse button.

Lotus-style horizontal pop-up menus present a second level of options shown as menu items (see Figure 14-20). The main difference is that pop-up selection continues to show between pull-down and pop-up menus the second level actions, whereas pull-down menus disappear as a selection is made.

Vertical pop-up menus are long lists that contain a portion of the list in a scrollable window (see

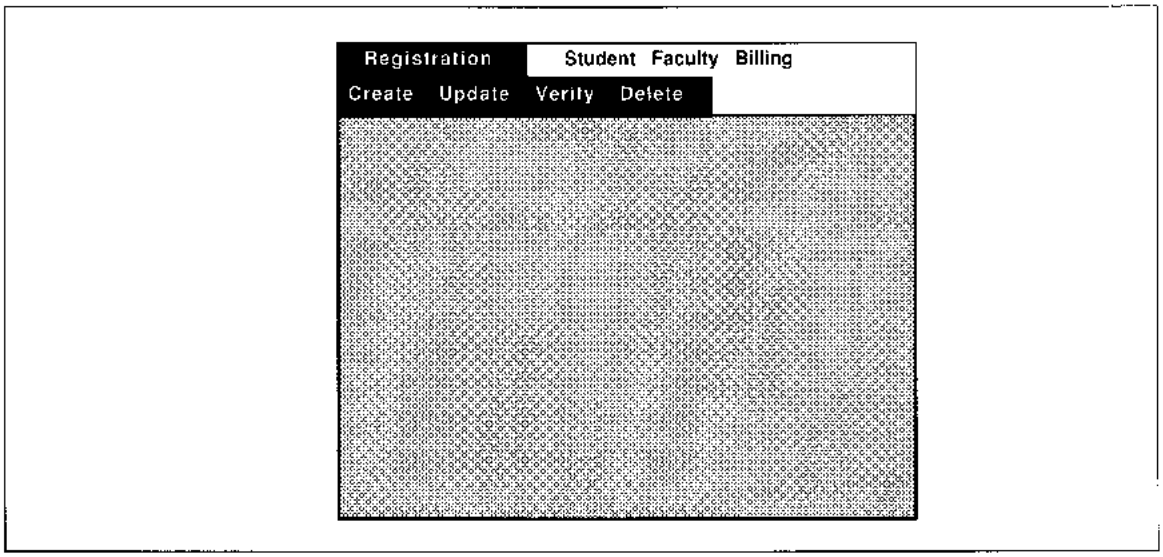


FIGURE 14-20 Lotus-Style Horizontal Pop-Up Menu Example

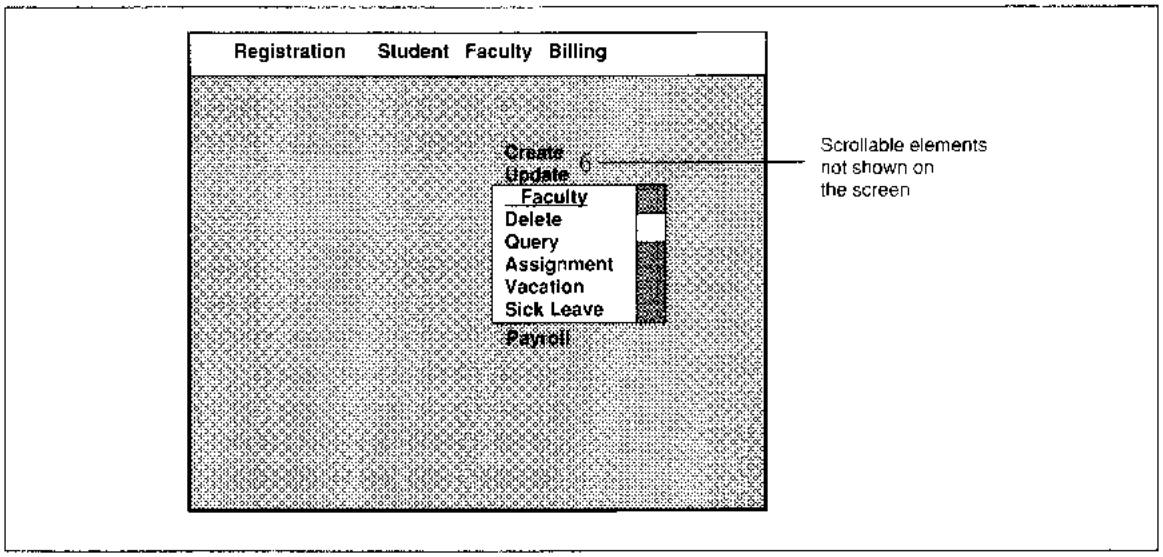


FIGURE 14-21 Vertical Pop-Up Menu Example

Figure 14-21). To select an action not currently showing, the menu is scrolled until the desired action is visible. Then it is activated. Vertical pop-up menus also disappear once an action is activated. In Figure 14-21, the items that would not be showing on the screen are in the gray area.

There is no research on the effectiveness of these three types of menus. In general, though, we know from past research that familiarity with the interface type leads to greater satisfaction with the software. Both horizontal pull-down and Lotus-style pop-up screens are familiar to most PC users. Vertical pop-ups remain useful for long lists.

Both pull-down and vertical pop-up menus offer a simple means for providing expert and novice modes of work. Command keys can be defined for specific functions and shown on a menu for optional use (see Figure 14-22). Novices can use the menu without paying attention to the commands, while experts can learn commands as they need them, becoming proficient in some areas and remaining a novice in others. This option, plus the office desk metaphor that people easily relate to, make windowed environments the preferred development screen style.

ABC Rental Option Selection

The ABC rental application is mostly transaction processing with some query processing. Both windows and menus are recommended for transaction systems, with windowed query development recommended for query applications. Both graphical and digital presentation are recommended. If hardware has not already been chosen, these recommendations imply math and graphic capabilities for the workstations. Standard displays should be sufficient unless Vic wants many graphics, in which case, one display should be high-resolution for graphical use.

The key screen design decision is between windows and full screen menus for selection. There is no one best choice in this decision. When software is chosen before screen design, software sometimes dictates the interface. For instance, mainframe software, for the most part, does not support windows as this text is written. The most advanced screens require a full-screen menu interface. Conversely, some PC software does not support anything but menu bars and windows. To use full-screen menus in this software is cumbersome and costly. User pref-

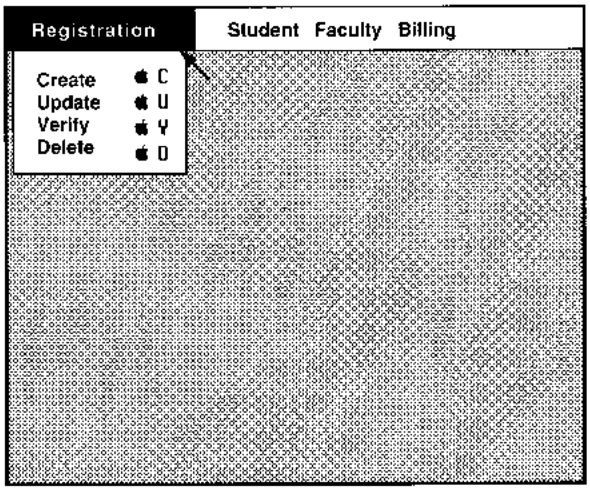


FIGURE 14-22 Function Keys on Pull-Down Menu for Expert Use

erence for selection tends to be strong and should be the deciding factor.

Assume no software is selected yet. To give Vic an informed choice we should sketch both window and menu screen and let Vic choose which he likes best. To do both, we have to design the interface to accommodate the application. For windows, the menu bar should include each major entity and/or process. The menu bars and subchoices for ABC rental processing are shown in Figure 14-23. This design might change with software selection, such as dBase IV, so a sample menu bar with subchoices for dBase is also shown as Figure 14-24. Next, a hierarchic menu system is defined for contrast (see Figure 14-25). The hierarchy menus mirror the task hierarchy defined above. One menu is present for each activity and for its successive levels of subactivities until the functional screens are reached.

The recommended design uses windows. Vic selects windows with the Figure 14-23 menus to be used. He dislikes the dBase menu because none of the functions relate to his applications. Finally, Vic requests a 'quick look' at the screens on the computer to confirm his choice.

Functional Screen Design

Functional Screen Design Alternatives

Once all navigation through menus or commands is complete, the functional level of screen is presented for the real work of the application. Functional level screen choices are direct manipulation, question and answer, and form filling. **Direct manipulation** interactions are those in which the user performs an action directly on some display object. CAD/CAM, CASE, and some computer-based training (CBT) systems have direct manipulation interfaces.

Question and answer (Q&A) interfaces are those in which progressively more focused dialogue takes place based on responses to preceding questions. Artificial intelligence applications and some CBT systems are the most common uses of the Q&A format.

Form-filling interfaces are most common in transaction processing applications but can be used for any application needing to collect discrete, single values for variables. **Form-filling** interfaces present the user with labels and indicators of where data is to be entered. Users are led through the form

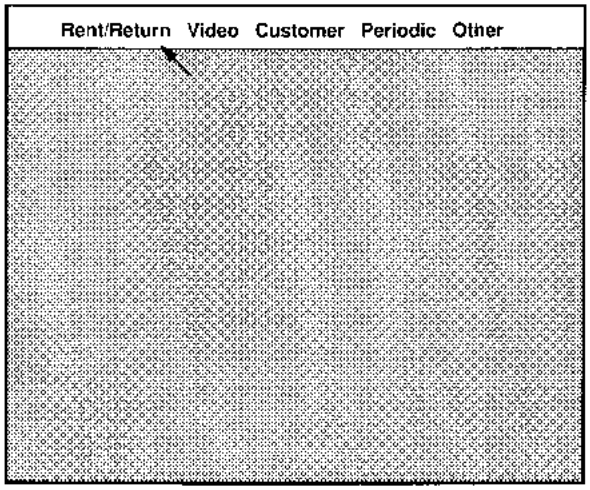


FIGURE 14-23 Menu Bar for ABC Rental Processing

completion process by cursor movement and messages from the software.

Functional Screen Design Guidelines

In general, the application type determines the most appropriate functional screen design. Recommended interface designs are shown in Table 14-6 for all application types. Windows are the preferred method of selection presentation because they can be layered to keep track of thinking processes during long selection sequences, and because their pop-up action matches the way people think more closely than menus. Command languages are not preferred for

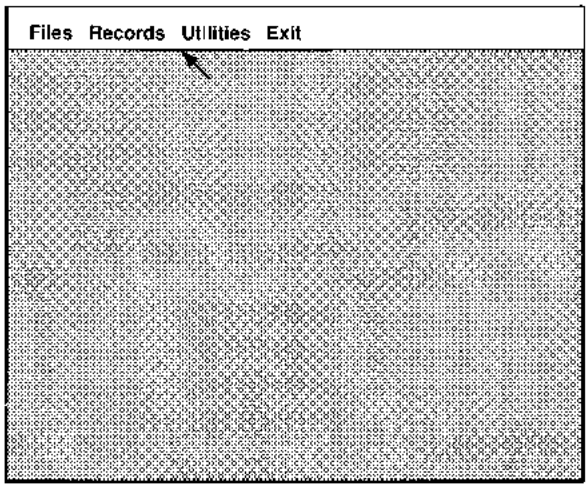


FIGURE 14-24 dBase IV Menu Bar for ABC Rental Processing

Option 1: All Menu Choices on One Screen

Rent/Return	Customer Maintenance Create Update Delete Query
Video Maintenance Create Update Delete Query	Periodic Processing End of Day History Update Query Startup Shutdown

Option 2: Individual Menus for Each Level of Choice

SCR01 ABC Video mmddyy
Rental Processing Application
Main Menu

Move the cursor to your choice,
Press Enter

Rental Processing
Customer Maintenance
Video Maintenance
Periodic Processing

F1:Hlp F3:End

SCR02 ABC Video mmddyy
Rental Processing Application
Customer Maintenance Menu

Move the cursor to your choice,
Press Enter

Create
Update
Delete
Query

F1:Hlp F3:End F5: Main

SCR03 ABC Video mmddyy
Rental Processing Application
Video Maintenance Menu

Move the cursor to your choice,
Press Enter

Create
Update
Delete
Query

F1:Hlp F3:End F5: Main

SCR04 ABC Video mmddyy
Rental Processing Application
Periodic Processing Menu

Move the cursor to your choice,
Press Enter

End of Day
History
Update
Query
Startup
Shutdown

F1:Hlp F3:End F5: Main

FIGURE 14-25 Hierarchic Menu Set for ABC Rental Processing

TABLE 14-6 Interface Design by Application Type

Application Type	Selection	Function	Display
AI	Window Menu Command Language	Q&A Form	Text short answer is usual display; could also include graphic results.
DSS and ESS	Window Menu	Forms Windows	Graphical-bar column, point plot Digital Need help and cautionary comments for inappropriate output form use.
Process Monitor/Control	Window Menu Command Language	Analog display Mimic display for multivalued or multidimensional data Digital display for specific numbers with symbols, numbers or indicators (e.g., alert) Command Language Direct Manipulation	N/A
Query	Window Menu Command Language	Window Form Command Language	Graphical-bar column, or point plot Digital
Transaction Processing	Window Menu Command Language	Forms	Forms

DSS and ESS because the users of these applications are usually managers who should not be expected to know a command language. DSS and ESS may be used infrequently and the interface should *chauffeur* and lead the user as much as possible. Command languages are the third choice for all application types because they assume expert level knowledge both of the task and of the computer system doing the task. Ideally, a combination of windows with optional expert commands should be provided.

For transaction applications, forms completion screens are preferred for functional processing. Q&A is much less efficient for transaction applications (TPS) than forms because line-by-line entry takes longer and is fatiguing. Direct manipulation is inappropriate for TPS.

For query applications, all options can be used for selection *and* query generation. Query generation is the functional processing in a query application. For query generation, windows with query criteria are preferred. For experts, direct command language use is preferred. Query results can use graphical or digital styles of presentation.

DSS and ESS should use a consistent interface until data results are presented. Either window selection with window request formulation or menu selection with form request formulation are recommended. Results screens can combine any graphic and digital presentation styles, although warning messages for inappropriate display selections might be desirable.

Artificial language applications usually result in a Q&A format. Each AI language environment uses its own method. For instance, Turbo Prolog™[4] uses a combination of windows and command language to initiate processing. A text answer which may have an associated probability of correctness is the usual AI output. Some AI language environments also support limited graphical display.

Last, in process control applications, the functional display *is* the results display. Analog, mimic, and graphical display are all common in process control, sometimes on the same screen. The display usually has a command line at the bottom of the screen. Commands are limited to requesting additional information about a certain measurement or part of the system being monitored, or requesting a different display. The most flexibility and sophistication of design are required in process control applications because they are most likely to be critical in terms of having life-sustaining responsibility.

ABC Functional Screen Selection

ABC rental processing is a TPS and will use forms for the data entry functions. The forms screens for data entry include rental, return, customer maintenance, video maintenance, periodic, and query selection processing. These screens should not change regardless of which option selection interface is selected. Therefore, they could be designed at the same time the general interface is being decided. In any case, the forms screens should be presented to Vic to get general comments and to correct any design he might dislike before a prototype is built.

Presentation Format Design

Once the general form of the interface is decided, details of display are decided. The first set of choices are for data presentation based on the type of data. The second set of choices are for specific field formats. Presentation format describes the method of displaying data on a screen.

Presentation Format Design Alternatives

The options for presentation format include analog, digital, binary graphic, bar chart, column chart, point plot, pattern display, mimic display, text, and text forms.

ANALOG. **Analog displays** are for continuously variable data (see Figure 14-26) and are usually used in direct manipulation interfaces. Analog displays use a pointer of some kind to show a position that is *analogous* to a value the position represents. Analog displays all should have a scale, pointer, a **direction indicator** of increasing/decreasing measure, and an indicator of normal/abnormal measures (see Figure 14-26). For instance, analog display is effective for the pounds per square inch of pressure (psi) to show a measure of exerted force. Another example from manufacturing is the continuous flow of various densities of oil from a cracking plant which is effectively conveyed via analog display.

The **scale** is a numeric indicator of the item measures. A **pointer** indicates the current position on the scale. Pointers might be arrowheads or needles and may be fixed or moving. The indicator of increasing/decreasing direction is usually a combination of arrows and text to indicate the meaning of direction of pointer movement. **Normal and abnormal measures** can be indicated by a shaded section of the scale, different colors to scale numbers, a change in color of the pointer, a tone for abnormal measures, or

4 Turbo Prolog is a trademarked product of Borland International.

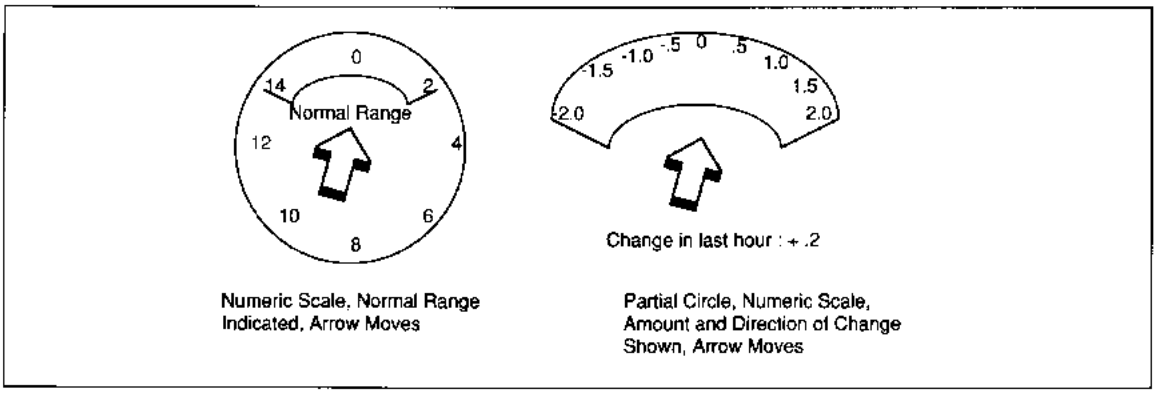


FIGURE 14-26 Examples of Analog Displays

some means of showing expected and unexpected numbers.

The guidelines for analog displays are summarized as follows:

- Display Contents
  - Scale to which the measure applies
  - Pointer to indicate position on the scale
  - Indicator of increasing/decreasing direction
  - Normal/abnormal measures indicated
- Display Design
  - Use conventional user mental model of item
  - Use moving points on fixed scales
  - Use same analog design for all analog measures on display
  - Use design method—circular scale or open, partial circle scale—to facilitate user recognition
- Usage
  - Rate of change
  - Range of values for continuous data
  - Determine acceptable operation

In general, the most effective displays fit the users' mental model of the measure, use moving pointers on fixed scales, and are consistently designed when more than one analog measure is used. If numeric analog values must be tracked, a semicircular open scale using a fixed pointer with a moving scale allows faster numeric recognition.

Analog displays are best used for monitoring rate of change, monitoring a range of analog values, or for determining ranges of acceptable operation. Examples of rate of change are the flow of oils in a cracking plant or the voltage fluctuation in cables. A monitoring example is a speedometer for speed limit. Pressure gauges in a nuclear power plant or bond ratings selections that must fall within company guidelines are examples of ranges of operation.

DIGITAL. A **digital display** is used to convey exact numerical information. Digital displays are most effective when used for variables that have one value at a time. Each value requires a label to identify the data value.

Guidelines for digital data and an example are shown in Table 14-7. In general, only that data of required precision for accuracy should be displayed. Field size should provide for the maximum and minimum values. If data displayed changes frequently, as in a stock trading application, the data should stay on the screen long enough for comprehension, about five seconds, before being changed. If the user is monitoring change, an arrow, plus/minus signs, or other indicator of direction of change might be shown.

BINARY. **Binary** means having two parts. A **binary display** shows some graphic to indicate a two-value selection option. Usually, we think of binary items as having on-off, or yes-no, or zero-one values.

TABLE 14-7 Guidelines and Example of Digital and Binary Data

Display Contents:
a. 'Y' or 'N' or other character
b. o or •
c. 'On' or 'Off'
d. 1 or 0 (One or zero) or other numerals
e. √ or blank

Display Design
If text form, use contents a, c, or d above
If analog display, use b or e
If in a menu list, use b or e

Usage
To indicate an item that is 'turned on' or 'turned off'
To indicate a two position setting

Example of digital time display

07:15

Binary interface information can be presented using text or graphics in several ways (see Figure 14-27). The binary item can be displayed in text using the words yes-no or on-off, or with letters 'y', 'n'. A menu can list the option with a check mark to indicate an 'on' condition. A graphical button, or circle, can be used—when the button is empty, the item is not on; when the button is filled in, the item is on.

By itself, binary indicator selection may not be a major decision. It becomes important when used with other information on the screen at the same time. If used in a menu, a check mark, change of color intensity, or change of color can all be used to effectively indicate an 'on' condition without using any extra characters. If used within a line of text, text presentation (e.g., 'y' or 'n') is more effective.

BAR CHART. A **bar chart** summarizes numeric data as one or more horizontal bars whose lengths correspond to the values of related variables (see Figure 14-28).

By convention, bar charts show increases in value as the chart is read left to right. Bar charts are effectively used to show task plans over time, percentage of task completion, comparisons of item values (i.e., *item 1* value vs. *item 2* value), and cyclic data (e.g., product sales over a fixed period). In business applications, bar charts are rarely used on screens with other graphic displays; they are generated by applications as summary output for managers, and can be easily generated on-line by many software packages.

COLUMN CHART. A **column chart** is a bar chart using vertical bars rather than horizontal ones. Bar charts are most often used when time is a fixed period (or is not relevant). Column charts are most often used when time varies and is shown on the x-axis (across the bottom). For instance, cyclic data is most effective in a bar chart when comparing a fixed period (see Figure 14-29). When comparing cyclic data over periods, a column chart is more effective.

The general rule is to use column charts for multiple time periods, to compare different items on the same scale, or for consistency with cultural conventions which assume a vertical scale (e.g., plotting temperatures, times, revenues, sales).

POINT PLOT. A **point plot** is a column chart that shows the x-y points on the diagram with or without a line connecting them (see Figure 14-30). Point plots might have trend lines generated to show the direction of change. A **band chart** is a special type of point plot that plots several variables on the same diagram. Band charts use shaded areas of the diagram to show variable participation. Bar charts are most effective for showing cumulative variable participation or percentage of participation of each variable (see Figure 14-31).

PATTERN DISPLAY. Pattern recognition is a human strength. When designing displays that are monitored for change in complex systems, patterns

Example of Alphabetic Listing Using Y/N Indicators

Name	Sex	Married ?	Deceased?
Jones, Sandra	F	N	N
Andrews, Darcy	F	Y	Y
Lane, Bruce	M	Y	Y

Example of Menu List
Using • or Blank Indicators

Font

10 Pt.
12 Pt.
• 14 Pt.

Cairo
Helvetica
• New Century
Times Roman

FIGURE 14-27 Examples of Binary Indicators

are effective. **Pattern displays** repeat the same graphic several times with identical 'normal' displays (see Figure 14-32a). When a change to one portion of the pattern occurs (see Figure 14-32b), it is easily perceived by users. These are not very common in business applications.

MIMIC DISPLAY. A **mimic display** shows a schematic or other replica of a system to allow the user to monitor its functioning (see Figure 14-33). Because mimic displays are usually symbolizing complex systems, the information presented should be kept to a minimum needed to control, monitor,

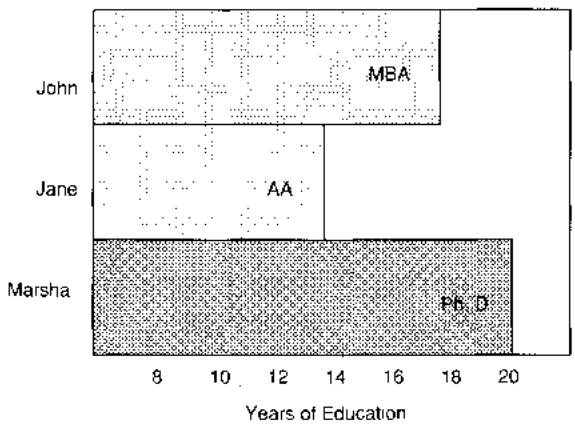


FIGURE 14-28 Example of Bar Chart

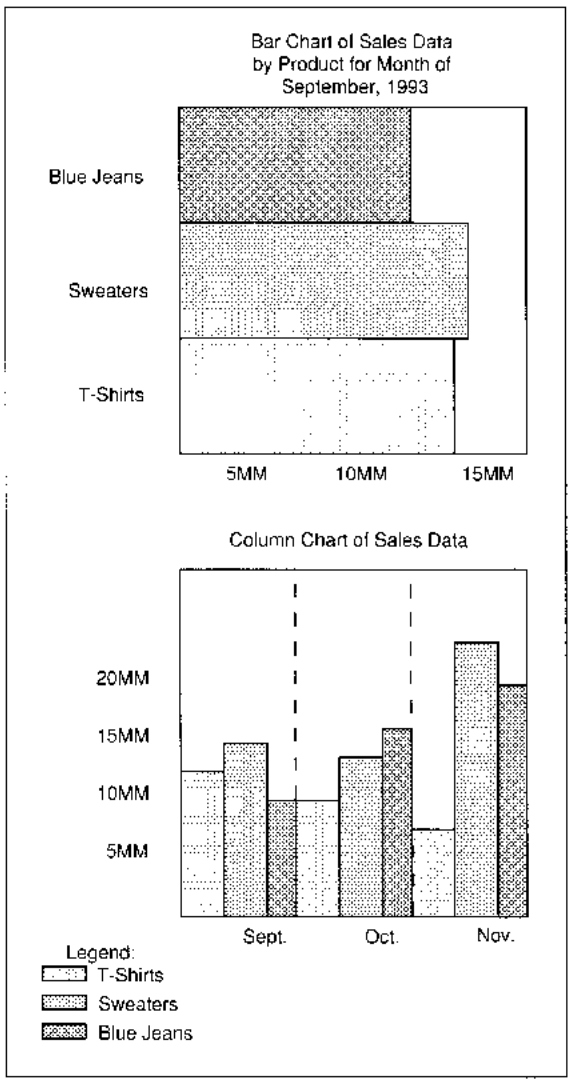


FIGURE 14-29 Bar and Column Charts of Sales Data

or obtain information needed. The symbols, spacing, and relative sizes of symbols used in the display should conform to business conventions to convey immediately meaningful information. For example, Figure 14-33 shows an electrical diagram, not a plumbing diagram; therefore, the users should be electricians or electrical engineers.

Mimic displays are best used when a monitoring application requires a view of the whole system. They provide understanding of system component relationships and can be more easily understood than other types of graphics for the same information. Colors can be used to highlight abnormal functioning of components. In business applications, mimic displays are effective for monitoring network components, telecommunication linkages between networks, and even for tracking problems in application interfaces.

NARRATIVE TEXT. **Text** is verbiage in which words, rather than numbers or symbols, are used to describe the intended information. Text is hard to read, time consuming to understand, and requires a high skill level of the user. Ideally, text is minimized; but some applications require comments or special, noncodable instructions that must be in text format. Some guidelines for text usage are the following:

- Use no more than 60 characters per 80 character line.
- Wrap text as a word processor does. Do not require the user to change lines.
- Use abbreviations common to the work context, and use abbreviations sparingly.
- Allow users to scroll, change paragraphs, and control the text creation process.

TEXT FORMS. One of the major uses of displays in business applications is for data entry that corresponds to a *form*. **Form screens** present a series of labeled fields of information for which some information is completed by the user and some information is generated by the application. Forms screens simulate paper forms that they replace or automate. Because forms automate information from paper, the format, sequence, spacing, and information to be completed should mirror that of the analogous paper form.

Forms screens should have standard header, instruction, body, and footer information that differs from the general screen format (see Figure 14-34). The different areas should be clearly delineated, grouping information that is related (e.g., the header) or that repeats. The header should contain an application identifier, function identifier, date, time, and a

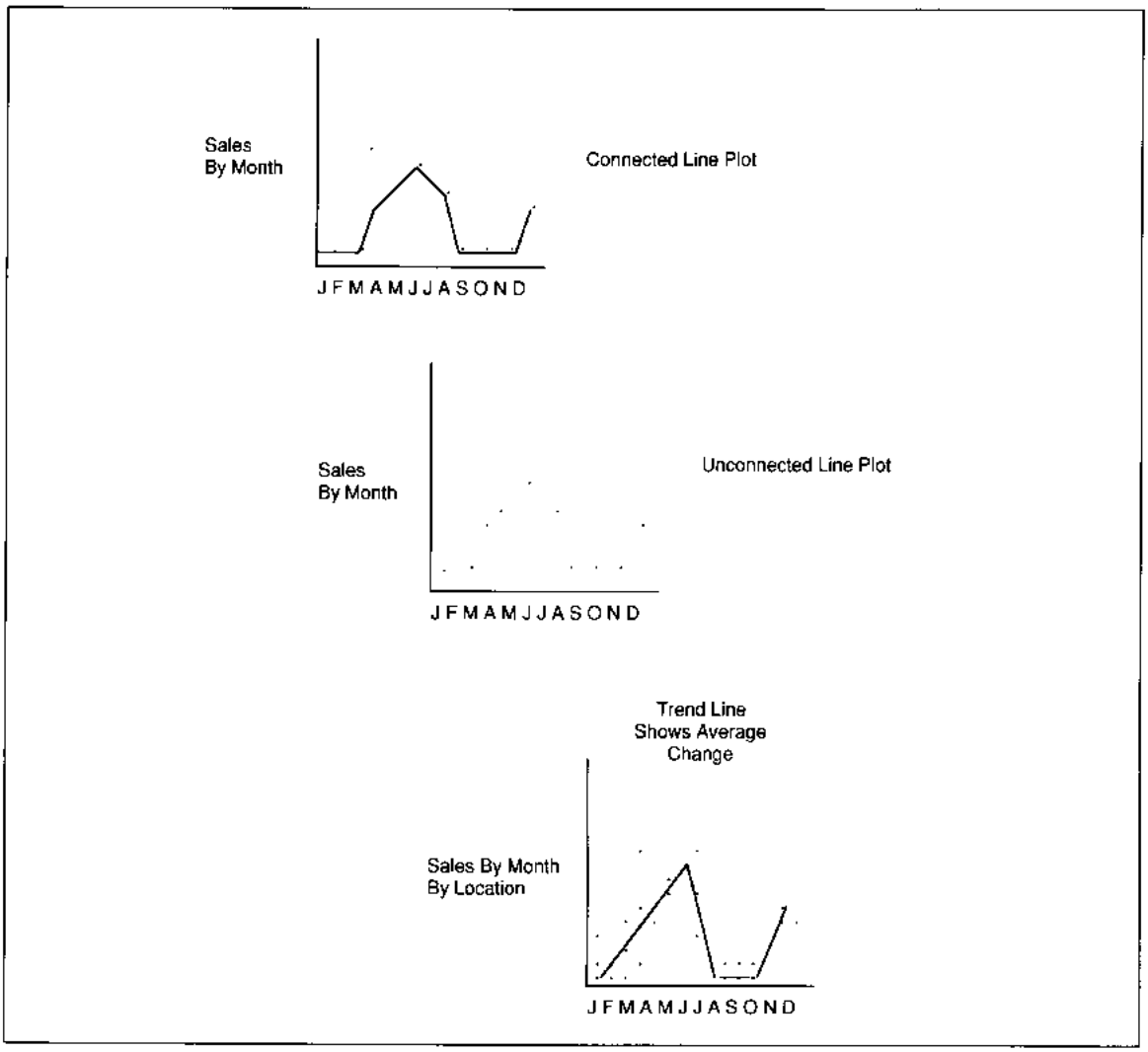


FIGURE 14-30 Example of Point Plots

screen/program ID as discussed above. The header may be the same as the general screen header.

The instructions can be in the form of screen text, help availability, or a short description of expected action. As much as possible, the screen should provide intuitive guidance. Instructions should lead the user to supply information to get to the next step.

The body of a form contains the labeled fields to be entered in an easily understood, contextually related format. The **footer** should provide screen summary totals or other summarizing informations. Footers and instructions are optional. The body, then, is the main focus of attention.

The body of the form should be partitioned or windowed to mirror sections of data to be entered (see Figure 14-35). The screen in Figure 14-35 shows a simple *Customer Add* screen for ABC Video. All information relates to the customer and there is no additional family member information in the application. If additional family members are

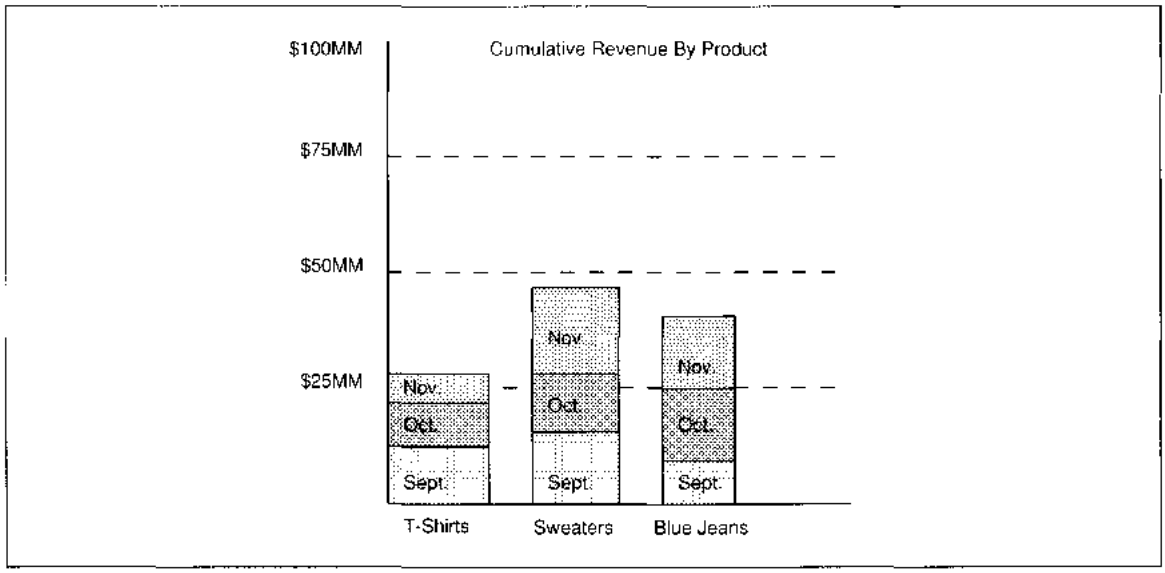


FIGURE 14-31 Example of Band Chart

added to the membership, the *Customer Add* screen might look like Figure 14-36 which shows two sections, one for general customer information and one for additional family members.

Each field or group of fields should be clearly labeled to identify the required information. Customer preferences are needed to design identification for some fields. For instance, three variations of name and address information are shown in Figure 14-37; all three conform to different, good design guidelines. The first variation shows each field labeled. The second shows major fields labeled and minor fields with understood labels. The third shows one heading for all fields; this heading minimizes the text on the screen. No one of these is preferred over the others. Rather, the customer should be allowed to choose the preferred design.

Labels, and any codes designed as well, should be designed to be familiar, less than five characters long, and include letters and numbers. For instance, Figure 14-38 shows four possible codes for a *Customer ID*. The first alternative, 913-8041, is a phone number. It is low in recognition for the clerks in the store, but the highest of any choice for the customer. Who doesn't know their own phone number? For that reason, high customer recognition, a phone number, is a good choice for *Customer ID*.

The second choice, CONG001, is a combination alpha and numeric code. The first four characters are the first four letters of a last name and the last three characters are a sequential number. This is also high in recognition for both customers and clerks. It is less recognizable than a phone number, but a good choice in any case. The next code, 03001 uses '03' to denote 'C' and a sequential number '001' to denote sequence within the Cs. The purely numeric code is

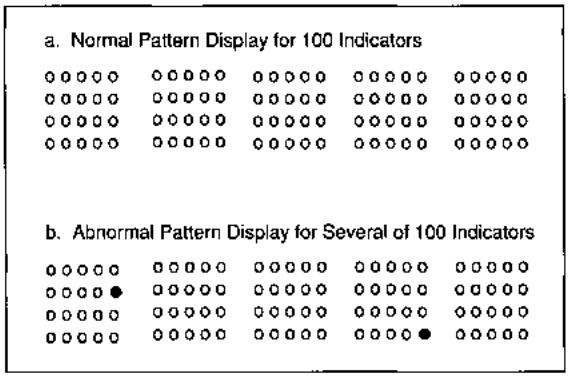


FIGURE 14-32 Normal (a) and Abnormal (b) Pattern Displays

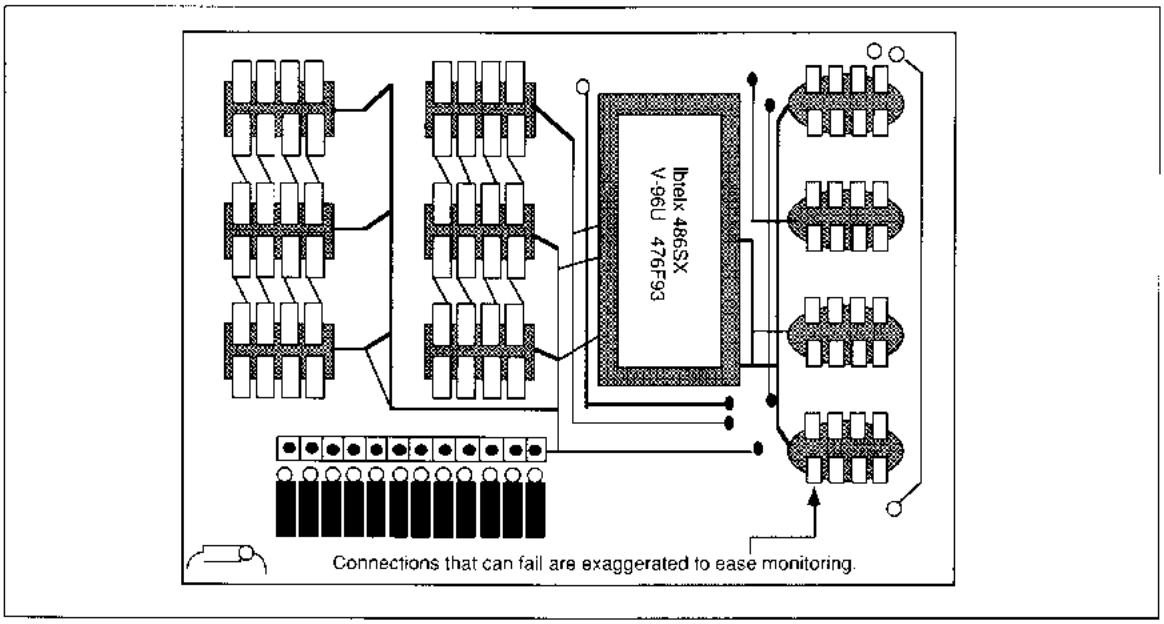


FIGURE 14-33 Mimic Display for Electrical Monitoring System

cryptic but short. It is less useful than the first two choices.

Text information, such as names, should always be left-justified. Ideally, they should be long enough to provide for the maximum length of the information. This is difficult with names, especially hyphenated names. Each application defines its own maximum; but, in general, over 90% of names in the United States are shorter than 35 characters. If disk storage space is tight, shortening fixed-length text fields is one way to conserve space; another is to define a variable length field that does not store unused spaces.

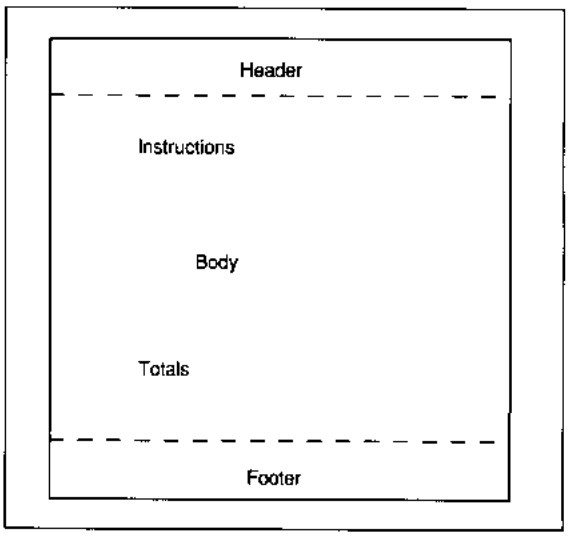


FIGURE 14-34 Sections of a Form Screen

After the individual labels, fields, and field codes are defined, the next task is to position them on the screen. The design is context related and should group fields that logically go together. From cognitive psychology research we know human brain capacity is limited to holding 5–7 bits of information called 'chunks' in our short-term memories. **Short-term memory** (STM), also called 'active' memory, is what is in your head while you are thinking. STM is measured in nanoseconds of response time for processing and is analogous to the arithmetic/logic unit (ALU) on a computer where all processing takes place. In designing presentation formats, we try to group items to take advantage of the chunking phenomenon. For instance, in the *Customer Add* screen

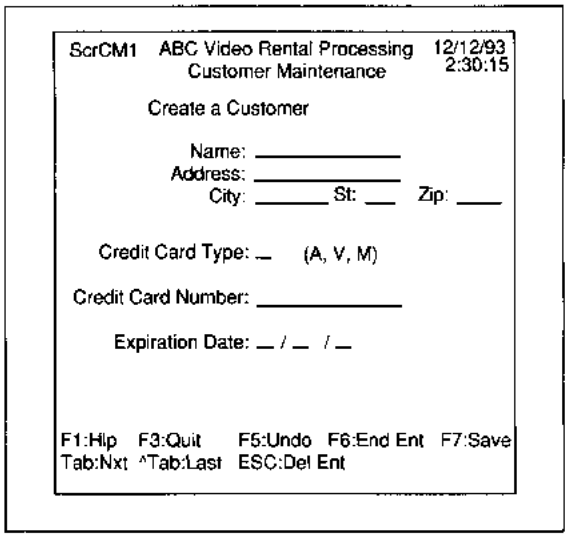


FIGURE 14-35 Customer Add Screen

in Figure 14-35 above, address and credit card information form two natural groupings of information that should be on the screen as a group. Another aspect of short-term memory chunking is to position required fields first, followed by optional fields. This placement should allow users to signal completion of data entry without having to tab through or touch unneeded fields.

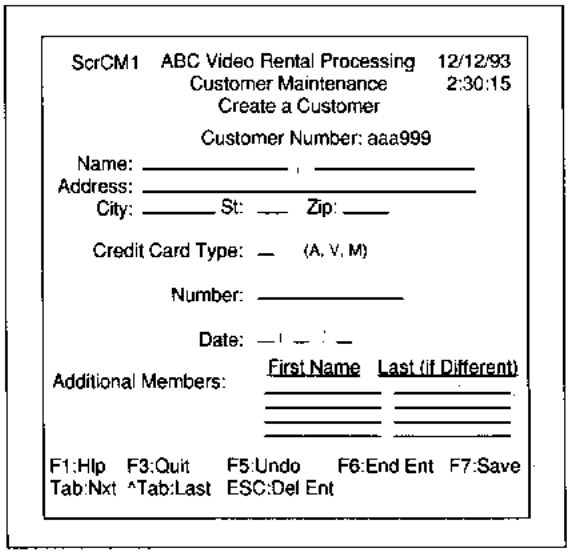


FIGURE 14-36 Customer Add Screen with Additional Family Members

We must also account for long-term memory processing in screen design. **Long-term memory** is what is stored in your brain, similar to disk storage for a computer. Retrieval of stored information uses a schema, or mental model, of what are effectively primary and secondary keys for retrieving information. Retrieval time is measured in 100s of milliseconds or slower. When chunking cannot be done, screen items should be spatially separated to allow users to switch contexts as they move their eyes from one section of the screen to another.

When positioning information on screens, you should also consider possible reusability for screens. For instance, the *Customer Add* screen above could also be used for delete verification, updating, and individual customer query.

When positioning is complete, each screen should be given a system name that is added to the task hierarchy to relate screens to tasks.

It used to be thought that the shortest possible terminal interaction time was desirable, but this is not true any more. Research shows that we need to pace work so that 'psychic overload' does not occur. Chunking items for data entry that logically go together is one way of pacing work. Another is in pacing the response time for different types of work. Long transactions can take a relatively long time, up to 20 seconds, while short transactions should take a short time, less than five seconds. Keystroke response is a simple, direct interaction and should have immediate response from the computer. A query, request to activate a function, and selection of a menu item are all examples of simple interactions. Examples of complex interactions are a database update, saving a word-processed document, or sending a facsimile transmission of several pages. Delays of up to 20 seconds are acceptable *if* the user is kept informed on the status of the processing. Some methods of telling the user the system is working are a message, '. . . *Working* . . .', a clock icon with hand movement synchronized to different percentages of completion, or a whirring sound from the equipment.

Other field definitions for forms relate to character entry and default values. Guidelines for

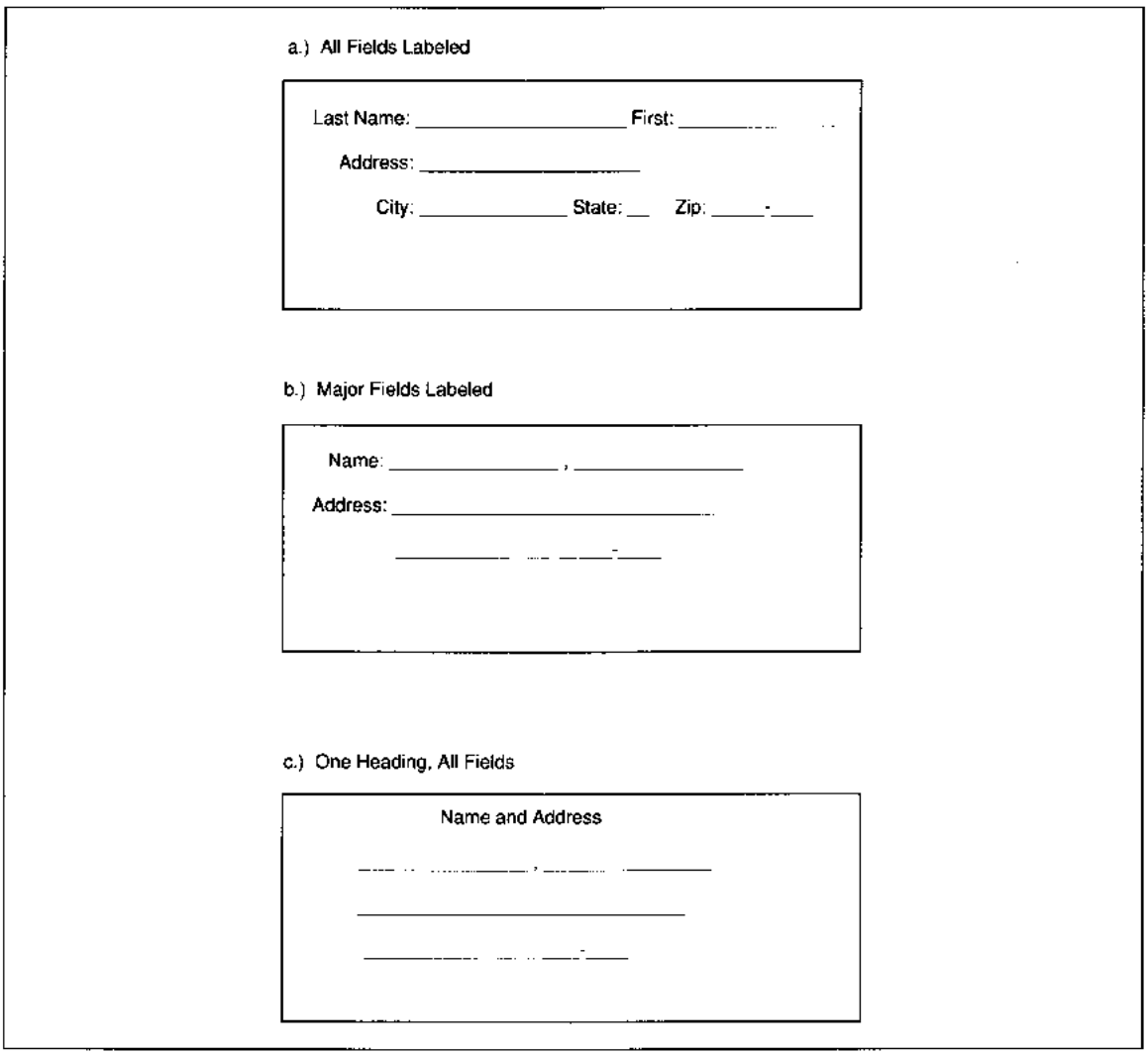


FIGURE 14-37 Label Variations for Name and Address Information

character entry are listed below; examples of field guidelines are shown in Table 14-8.

- Always display keyed information.
- Never require delimiters to be keyed. For instance, in a social security number, provide dashes to split the numeric parts: xxx-xx-xxxx.
- Do not require entry of leading zeros for numeric fields or of following blanks for text fields.
- Make areas of the screen not used for input inaccessible to the user.

Guidelines for default values are:

- Display all defaults before any data entry begins.
- Confirm defaults by tabbing past the field.
- Default replacement should not alter current default value. For instance, if the default date is today's date, and the operator places yester-

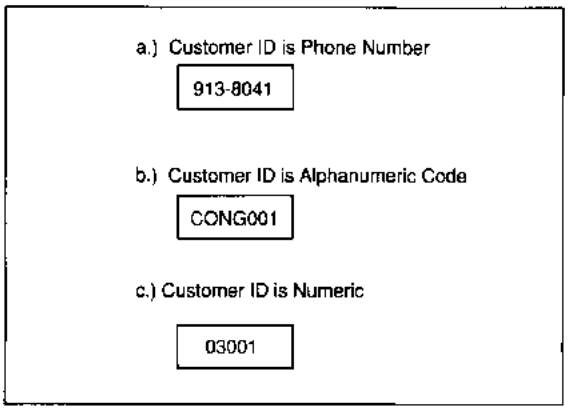


FIGURE 14-38 Variations for Customer ID Code

day's date in the field, the next transaction should still have the default of today's date.

ABC Rental Presentation Format

First, design the standard interface for all functional screens in the application. This should include header, date, time, screen ID, and program ID (see Figure 14-39).

Next, design the keys for navigation, error correction, and help and design the footer to identify them and their functions. The standard used here is fairly common. Program keys and their meanings are shown in Figure 14-40.

We need to know when a portion of processing is done, for instance, when returns are complete (F6), and we need to know when the transaction is complete for inputting the total amount paid (F6). The F8-F11 functions are used for retrieval and query processing to browse through multiscreen output (F8-F9) that is longer than 80 characters (F10-F11). The other keys are for changing actions during data entry.

The designations for F1, F3, and F8 through Escape (ESC) are IBM standards that have been followed by many PC applications. The remaining keys: F2, F4–F7 are open to definition. F2 and F4 are not used here and can be used for future changes. We could have assigned the End Entry type and End Transaction functions to F2 and F4 as easily as to F6 and F7 (see Figure 14-40). F2 and F4 are not used to minimize the probability of hitting the wrong key and canceling a good transaction. If either of these keys is pressed accidentally, it should have no effect.

Finally, we design the detail form screen for rental/return processing. The periodic processing and customer and video maintenance screens are left as assignments at the end of the chapter. Rental/return processing includes chunks for *Customer* information, *Open Rental* information, *New Rental* information, and *Payment* information. Corresponding to the chunks of information, the screen can be thought of as having four sections. The middle two sections are identical except that *New Rentals* cannot have return dates, late fees, or other fees applied. So, we design three different sections. Each section is designed separately, keeping in mind that there are 20 usable lines on the screen and that we want about 75% blank space. For this screen design, we assume a screen size of 24 lines by 80 characters per line.

The sections of screen information should be prioritized for condensation and crowding if it becomes necessary. For ABC rental processing, the priorities are highest to lowest: rentals, payment information, returns, and customer. Since new rentals are generating the payment information, they are most important. Payment information is second because it must be accurate and easily understood for the clerk to handle money properly. Returns are a low priority here because 90% of returns are on time. Customer information is only important for the clerk to verify the customer name. If necessary, the remaining customer information could be condensed onto one line for display.

The first section of the screen is for *Customer* information. The information to be included is name, address, city, state, zip, phone number, and credit status.

The first issue to be decided is what type of field labels to use. For example, the options for *Customer* are individual field identifiers, only a *Customer* identifier, or some combination of the two (see Figure 14-41). To minimize information on the screen, we use only the word *Customer* (Option 2, Figure 14-41). This also makes sense since the *Customer ID* probably is to be scanned to minimize data

TABLE 14-8 Field Format Guidelines

Content	Poor Design	Better Design
Do not intersperse letters and numbers	A1B1C1	ABC001
Use alpha mnemonics that are meaningful, predictable, easy to remember, distinct	ZXCVB001	Video001
Try not to mix special characters with letters and numbers	User types: $123.45	Preformatted $_ _ _._ _ User types: 12345
Break long codes into groups of three and four digits	277426631	277-42-6639
Do not use frequently confused letters in codes	o and 0 l and 1	Use zero, 0, only Use one, 1, only
Identify maximum number of spaces for item data entry; replace space marker as data is entered.	Enter Vid-ID	Enter Vid-ID: _ _ _ _ _ after three char. Vid-ID:123_ _
Labels	**Poor Design**	**Better Design**
Use abbreviations and contractions	Video Identification	Video ID
Try to keep labels less than eight characters long	Customer name and and address	Customer:
Design abbreviations to be less than five characters	Ident	ID
Separate mnemonics by hyphens	VidID	Vid-ID
Place label to left of single occurrence field	Name: Sam Jones	Name: Sam Jones
Place label over column of repeating information	Name: Sam Gerry Leonard Jesus	Name: Sam Gerry Leonard Jesus

entry and the *Customer* information is displayed automatically.

The second issue is format of the information. The options in Figure 14-41 all follow a conventional post office address format. The address need not be formatted in that manner, but the high recognizability of addresses in this format is a strong inducement to keep it the same. Unless screen space is a major problem, the post office format will be kept.

Two fields remain: *Customer ID* and *Credit Status*. *Customer ID* is an important field as the identifier of the information and should be positioned in a way that highlights its presence. Conversely, *Credit*

TABLE 14-8 Field Format Guidelines (*Continued*)

Error Messages	Poor Design	Better Design
Use upper and lower case if possible	ALL UPPER CASE IS DIFFICULT TO READ	Mixed case is preferred to enhance readability.
Only use asterisks in extreme situations	*****This ***** is *****very ***** distracting *****.	*****ALERT***** The database may have been destroyed.
Error IDs should be in a consistent location	PF001 Error 001 Error 002 PF002	PF001 Error 001 PF002 Error 002
Should be brief	Numerics were expected by the application but you entered some nonnumeric information.	Numerics expected.
Should be positive	You entered an illegal date format.	Enter date format mm/dd/yy
Should be constructive	You idiot! This mistake should NEVER occur.	Reconstruct database and begin again.
Should be specific	Illegal entry or ?	Enter data format mm/dd/yy
Should be comprehensible	FAC DB 29081230123	Database error. Call the DBA at x3456.
Should allow the user to feel as if they control the system rather than the system controlling them.	?	To undo, press F5.
Provide levels of messages with less detail for error message and more detail for requested help.		

Status is only important when it is the cause of a canceled request. So, *Credit Status* needs some sort of 'alert' design but, otherwise, can be positioned to conserve space. Several alternatives for *Customer ID* and *Credit Status* formats are shown in Figure 14-42. All alternatives are acceptable; the third option is selected because it minimizes labels and has credit in an easy-to-spot location—the upper right corner of the screen.

The second section of the screen is for *Open Rentals* information. The information needed on the screen includes *Video ID*, *Copy ID*, *Description*, *Rental Prices*, *Rental Date*, *Return Date*, *Late Fees*, and *Other Fees*. By convention, a typical bill,

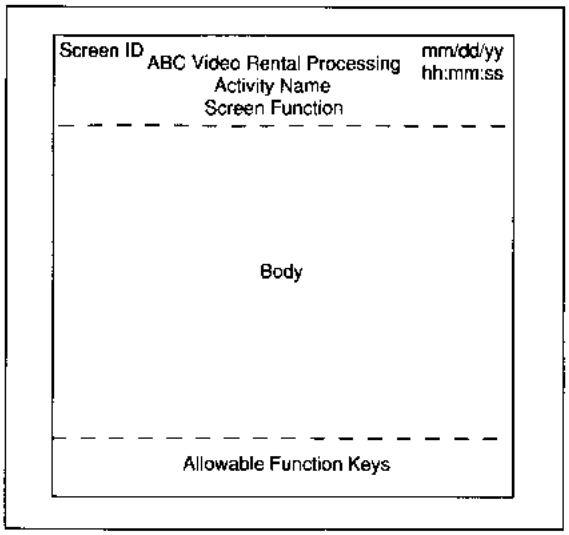


FIGURE 14-39 Standard for ABC Video Functional Screens

invoice, purchase order, or shipping papers list the item identifier followed by its description. We follow this convention for ABC. Two basic alternatives for fees and dates are shown in Figure 14-43. Since the same line design will be used for the *New Rentals* screen section, the alternatives as they would display for new rentals are also shown.

Key	Functions
F1	Help
F2	Not Used
F3	Quit/No Save
F4	Not Used
F5	Undo Last Entry
F6	End Entry
F7	End Trans/Save
F8	Page Forward
F9	Page Back
F10	Shift Page Right
F11	Shift Page Left
DEL	Delete Character
ESC	DEL/Cancel Field
TAB	Go to Next Field
Shift/Tab	Go To Last Field

FIGURE 14-40 Program Keys and Functions

The alternative which is easier to read and understand should be selected. If neither is obviously easier to read, the user should be consulted. The choice here is the first alternative. Keeping the dates together allows fast understanding of a tape's lateness, while keeping the rental information and return information separate allows fast understanding of rental fees owing. Vic has stated that no rentals are made without payment of rental fees, so the second option loses some appeal. The first option is selected then on the basis of keeping like things together—dates with dates and money with money. When returns are processed, the default of today's date should be placed in the *Return Date* field.

The third section of the screen is for *New Rentals* information. For this section, we use the *Open Rentals* line definitions and blank out the fields for return dates, late fees, and other fees (Figure 14-43a). A default of today's date should be placed in the *Rental Date* field. The only issue is how many tapes should a customer be allowed to rent at any one time. There are arguments for any number one can select and they all are determined by opinion. Therefore, Vic should select the number of allowable tapes out on rent at any one time.

When asked, Vic wants no restrictions at first. Then, he reconsiders. "If I allow unlimited tapes, someone could theoretically give me a stolen credit card as identification, rent many tapes, leave town, and I'm out the tapes. Maybe I should limit the number. But, one or two does not seem enough. What if they are short, like music videos? What if they want to watch movies all day? Why should I stop them? Hmmmm. I think someplace between 10 and 20 is probably okay because most people would never rent that many. My biggest customer is George Anderson and he takes out about six tapes at a time. So, I guess 10 is a reasonable limit."

With ten tapes as the limit, the screen needs no scrolling because all information will fit on one screen. Because this choice turns out to be an important design decision, Vic should be reconsulted and told that scrolling will not be available for rent/return processing. If he chooses to change the number, or asks for scrolling, it should be provided.

The fourth section of the screen is for *Payment* information. For payments, the fields are the *Total*

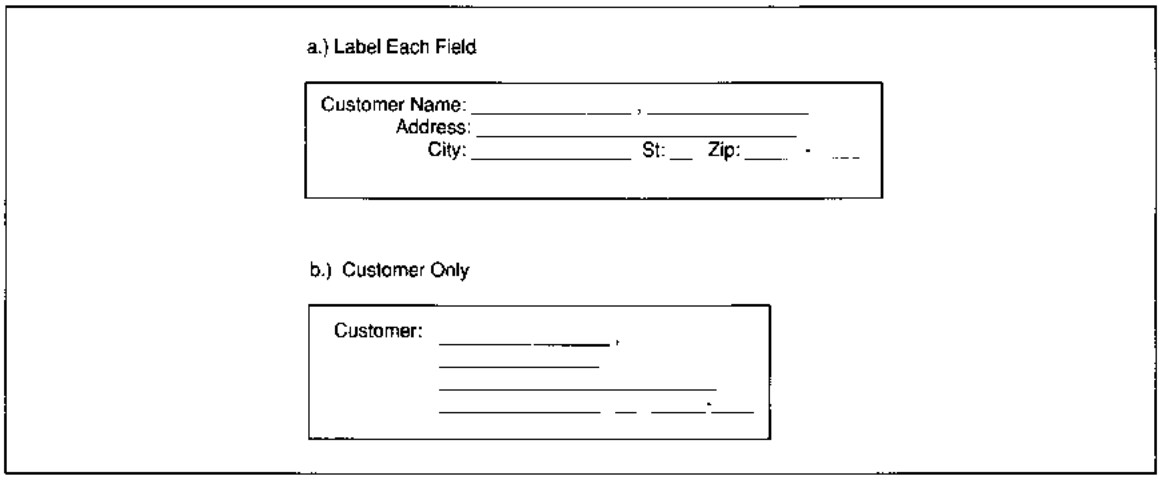


FIGURE 14-41 Customer Name Screen Options

Amount Due, *Total Amount Paid*, and *Change*. These could be on one line, two lines, or three lines as shown in Figure 14-44.

The choices for payment should be first, readability and understandability, and second, space available. For ABC, all information can fit on the screen with three-line spacing and still have room left over. So, the last alternative (Figure 14-44c) is selected as most easily read. The money fields should be right-justified with one set of numbers on the rental/return lines. The title fields should be right-justified for the group of three lines.

Last, we consider placement of the entire screen in the blank area between the standard screen header and footer. So far we have 22 lines accounted for in the rental screen: two standard header, one screen header, two footer, four customer, ten rent/return, two rent/return header, and three total lines. There

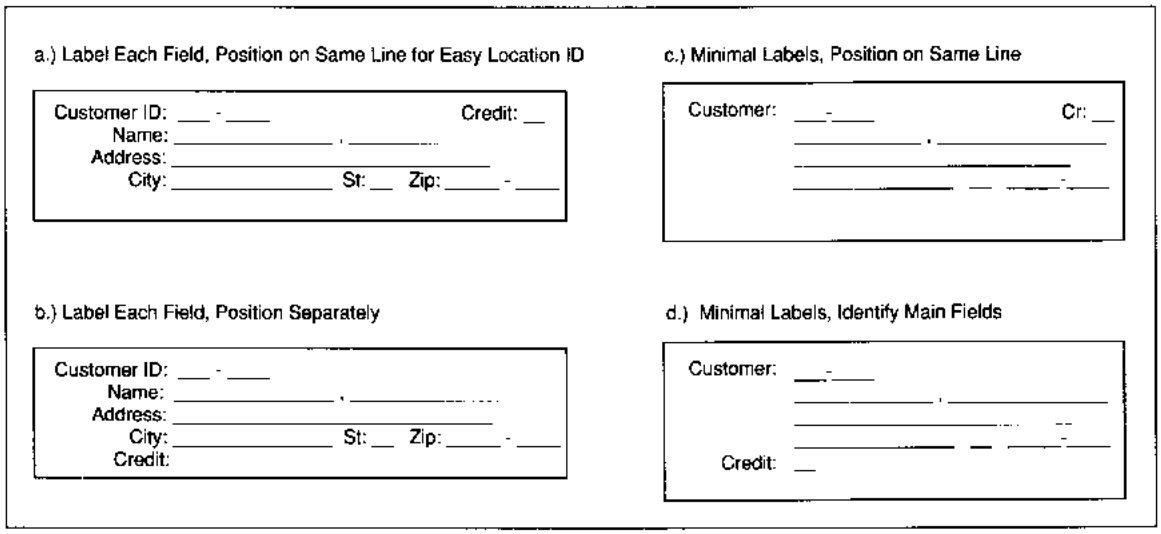
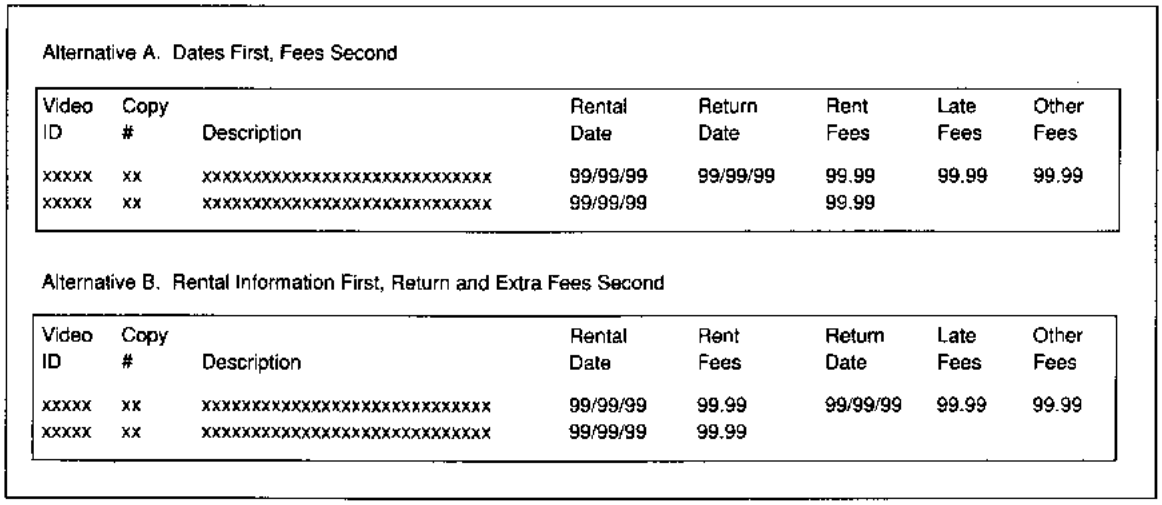


FIGURE 14-42 Alternatives for Customer ID and Credit Status



Alternative A. Dates First, Fees Second

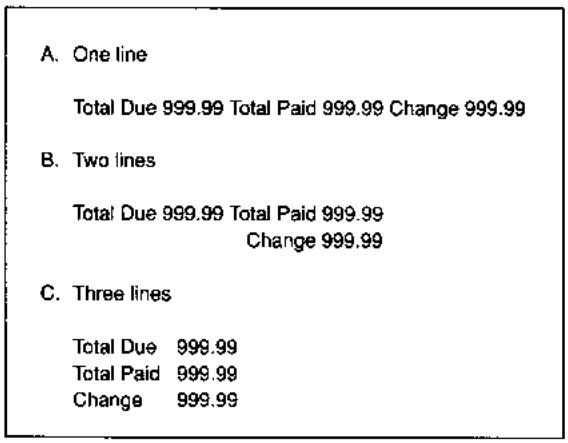
Video ID	Copy #	Description	Rental Date	Return Date	Rent Fees	Late Fees	Other Fees
xxxxx	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	99/99/99	99/99/99	99.99	99.99	99.99
xxxxx	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	99/99/99		99.99		

Alternative B. Rental Information First, Return and Extra Fees Second

Video ID	Copy #	Description	Rental Date	Rent Fees	Return Date	Late Fees	Other Fees
xxxxx	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	99/99/99	99.99	99/99/99	99.99	99.99
xxxxx	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	99/99/99	99.99			

FIGURE 14-43 Alternatives for Dates and Fees

are no extra lines on the screen (see Figure 14-45). Ideally, one blank line should separate the header and footer from the body. Also, one blank line is desired to separate the rental/return information from customer information. To provide blank lines, we either delete a header line or change the arrangement of information on the screen. According to our priorities, customer information should be condensed onto fewer lines to gain the blank lines. The *Customer ID* can be added to the customer name line and given its own label to specifically identify it (Figure 14-46a). This makes reading the *Customer ID* somewhat more difficult but adds to the readability of the rental information. A better choice is to redesign the standard header and make it two lines, with the second line identifying the function, and only display function keys available and use one line. This screen (Figure 14-46b) is preferred and recommended. In the end, Vic should select *his* preferred screen and it should be the final design. Vic selected the recommended screen for the same reasons that informed its design.



A. One line

Total Due 999.99 Total Paid 999.99 Change 999.99

B. Two lines

Total Due 999.99 Total Paid 999.99
Change 999.99

C. Three lines

Total Due 999.99
Total Paid 999.99
Change 999.99

FIGURE 14-44 Alternatives for Payment Information

Field Format Design

Field Format Alternatives

Field format design selects the characteristics of individual fields or values of fields on a screen. The alternatives for field format design include size, font, style, color, and blink for individual field values, and include coding options for field labels.

SIZE. Size is an issue in field attribute definition when it is selectable. For many software platforms, the size, spacing, and selection of characters is fixed within the application. Size of characters is mea-

SCRR01 ABC Video Rental Processing 12/02/94
Rent/Return Processing 02:03:15
Rentals and Returns

Customer: #xxx999 Cr: x
xxxxxxxxxxxx xxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxx, xx 99999

Video ID	Copy #	Description	Rental Date	Return Date	Rent Fees	Late Fees	Other Fees
xxxxx	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxx	99/99/99	99/99/99	99.99	99.99	99.99
xxxxx	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxx	99/99/99	99/99/99	99.99	99.99	99.99
xxxxx	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxx	99/99/99	99/99/99	99.99	99.99	99.99
xxxxx	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxx	99/99/99	99/99/99	99.99	99.99	99.99
xxxxx	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxx	99/99/99	99/99/99	99.99	99.99	99.99
xxxxx	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxx	99/99/99		99.99		
xxxxx	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxx	99/99/99		99.99		
xxxxx	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxx	99/99/99		99.99		
xxxxx	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxx	99/99/99		99.99		
xxxxx	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxx	99/99/99		99.99		
				Total Due:	999.99		
				Amount Paid:	999.99		
				Change:	999.99		

F1: Hlp F3: Quit F5: Undo F6: End Ent F7: End Trans F8: Pg Up F9: Pg Dn F10: Sh R F11: Sh L
Tab: Nxt Fld ^Tab: Lst Fld ESC: Cncl

FIGURE 14-45 Alternative 1 for ABC Rental Screen

sured in points. A **point** is a measure of type that is approximately 1/72 of an inch (about 2.8 mm). In general, the size of characters should be no less than 10 points and no more than 14 points unless an alert or alarm situation is being shown. These sizes are in the range of normal printed point sizes for display processing. An example of the range of point sizes is shown in Figure 14-47.

The default in most applications is 12-point type. As you can see from Figure 14-47, the larger the point size, the fewer characters fit on a screen. At 18 inches, the minimum point size should be about 9 and a comfortable point size is 12. The further away from the screen the user is, the larger the point size should be. At 30 inches, the minimum point size should be 10 points and either 12 or 14 points print size are acceptable. At 10 feet, the size should be about 72 points, or one inch.

FONT. Most software applications have a fixed default for type font as well as type size. Most applications default to a serif style such as that used in this text. A serif font has been proven easier to read and faster to comprehend than a sans-serif style such as this. If fonts are selectable, the rule of thumb is to select one or, at most, two fonts and use them consistently throughout the application for obvious distinctions. For instance, use one font for all field labels and another font for all information entered by the application user. Do not mix fonts for the same purposes or users will get confused and error rates will increase.

STYLE. Type styles might include regular, **bold**, *italic*, outline, reverse video, SMALL CAPS, ALL CAPS, underline, or ~~strike through~~. While the options make for interesting reading, interchanging

a.) Customer ID on Customer Name Line

```
ID: xxx999  Customer: xxxxxxxxxxxxx xxxxxxxxxxxxx               Cr: x
                      xxxxxxxxxxxxxxxxxxxxxxxxxxx
                      xxxxxxxxxxxxx, xx   99999-9999
```

b.) Recommended Screen Design

```
SCRR01                                ABC Video                                        12/02/94
                               Rent/Return Processing                                  02:03:15

Customer: #xxx999                                               Cr: x
          xxxxxxxxxxxxx xxxxxxxxxxxxx
          xxxxxxxxxxxxxxxxxxxxxxxxxxx
          xxxxxxxxxxxxx, xx   99999
```

Video ID	Copy #	Description	Rental Date	Return Date	Rent Fees	Late Fees	Other Fees
xxxxx	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	99/99/99	99/99/99	99.99	99.99	99.99
xxxxx	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	99/99/99	99/99/99	99.99	99.99	99.99
xxxxx	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	99/99/99	99/99/99	99.99	99.99	99.99
xxxxx	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	99/99/99	99/99/99	99.99	99.99	99.99
xxxxx	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	99/99/99	99/99/99	99.99	99.99	99.99
xxxxx	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	99/99/99		99.99		
xxxxx	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	99/99/99		99.99		
xxxxx	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	99/99/99		99.99		
xxxxx	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	99/99/99		99.99		
xxxxx	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	99/99/99		99.99		
				Total Due:	999.99		
				Amount Paid:	999.99		
				Change:	999.99		

F1: Hlp   F3: Quit   F5: Undo   F6: End Ent   F7: End X   Tab: Nxt Fld ^Tab: Lst Fld   ESC: Cncl

FIGURE 14-46 Alternative 2 for ABC Rental Screen

the styles on a form to be completed make it much harder to comprehend and will increase error rates. In general, regular print is acceptable in all applications for text display. For general purpose, noncritical text, regular print is recommended.

Bold print and reverse video are useful to call attention to a field if it is warranted. For instance, bold type style is effective for alert field values on a monochrome screen. A common use of reverse video is to show cursor position. The character at which the cursor is positioned is shown in reverse video and switches back to normal as soon as the cursor is moved.

Italics and outline are not generally used because they are harder to read and, therefore, increase comprehension time. Strike-through and underline are used mostly in word processing applications and can be effective in that context. For most forms-

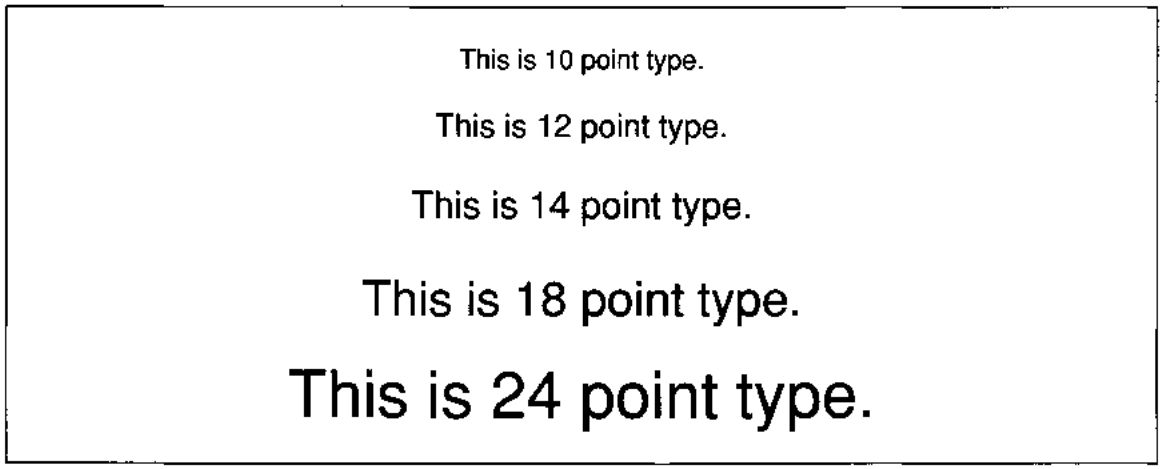


FIGURE 14-47 Sample Point Sizes

completion TPS applications, neither of these is recommended. Finally, research studies have shown that use of all capital letters increases comprehension time and they are not recommended.

COLOR. Color can be an effective addition to screen design, or it can seriously detract from the understandability and readability of the information. For indicating binary or ternary conditions, color is faster and easier to comprehend than any other coding scheme.

Research provides clear guidance on appropriate and inappropriate uses of color for application displays. Color is most effectively used for search tasks in which the goal is to find one or two objects (of the same color) that differ from surrounding objects. This type of search does not occur often in business applications. Color coding also is effective for:

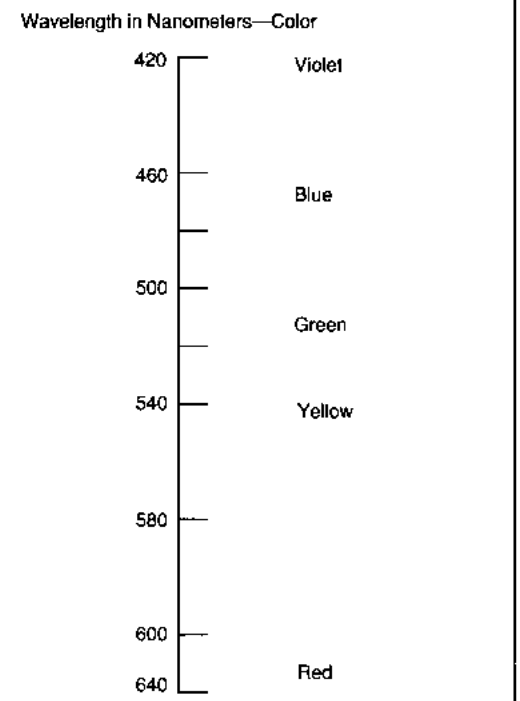
- unformatted display of information
- symbols which may be within a high density of information on the screen
- tasks in which the position of the item to be identified is not known but the color is
- screens for which color relates to the task
- user tasks involving search and recognition of differences in symbol color

Color is least effective for tasks in which a large number of colors are indiscriminately used, for which colors selected do not differ sufficiently to enable distinction, and for tasks in which the goal is to identify large numbers of objects (of the same color) when surrounded by a large number of objects of other colors. These ineffective color uses result in problems of discrimination. Research findings show that performance deteriorates with more than six colors on a screen. Many writers suggest using no more than four colors at any one time for business tasks.

Research on color selection recommends selection by wavelengths, ensuring sufficient contrast to speed comprehension. For instance, Figure 14-48 shows common colors on a spectrum by wavelength. Poor choices would be blue, blue-green, and green for different meanings on the same screen. Good choices would be red, yellow, and blue, because they are sufficiently different to facilitate understanding.

Because color blindness and other color perception problems are common, user profiles and user testing should be used to guarantee that all users can recognize all colors on a screen. Bold or odd colors of any type, for example, olive-green, should be avoided.

Common meanings ascribed to colors should be used in the application, and the common meanings which change by culture should be adapted. The government recommends using red only for alert conditions, yellow for warning, and green for normal



Adapted from Banks, William W., & Jon Weimer, *Effective Computer Display Design*. Englewood Cliffs, NJ: Prentice-Hall, 1992, p. 128.

FIGURE 14-48 Color Spectrum

because that is the common, conventional use for these colors. The use of a flashing red signal should be limited to an emergency condition requiring immediate action.

BLINK. Blinking characters or 'flashing' is a useful attention-getting device for monochrome or limited color displays. Blinking is considered more annoying than color codes by most users and should be limited to no more than one field at a time or one meaning at a time. An example of effective flashing would be to flash all data entry fields in error. As errors are corrected, flashing stops.

Blinking rates need to be monitored for the **flash rate** or speed of blinking. The optimal flash rate is 2–3 times per second with equally spaced intervals for on and off. Rates of 8–12 flashes, while discernable, can cause nausea and even seizures in people with photo-epilepsy. For those of us over age 30, a phenomenon called **flicker fusion** causes us to see constant light when the flash rate is very high, over 50 times per second.

Guidelines for Field Format Design

Assignment of field format characteristics is a judgmental activity based on SE experience and common sense. Follow the tenet 'less is more' in defining field formats that add formatting options. The use of these options diverts attention, causing a delay in the thinking process. If delay and attention shift are not desired, the result will increase error rates and reduce productivity.

Effective uses of color, blink, or audio sound for directing attention should be considered; however, user approval should be obtained before adding formatting changes to the screens.

ABC Field Format Design

One field on a rental screen, credit standing, might be worth highlighting. In addition, when processing takes place, several other items might be highlighted. In particular, data entry errors and insufficient payments, late tapes, and special fees should be considered for use of color, blinking, or bold type. These items are chosen because they represent all of the abnormal conditions that occur during rental processing.

A customer's credit standing is acceptable unless it is specifically changed by Vic during an update process. Since its change requires management action, a customer with a poor rating should probably be denied rental rights. This process has never been discussed with Vic and needs verifying. If he approves, the credit standing for poor ratings only could be displayed as a red or a blinking field to highlight credit status.

Data entry errors can also be highlighted. Since red is being used to signify denial of rental rights, a

different color should be chosen. If data entry errors are highlighted, the recommended colors are either yellow or blue to make them distinct from the red used for credit standing.

Insufficient payment occurs when the *Change Amount* is a negative number. The current design calls for moving the cursor to the payment field which is updated with the new *Total Amount Due*. Since this is not an expected occurrence, clerks might miss the cursor movement and complete the transaction even though insufficient payment has been made. Some method of highlighting is also desirable to ensure against such mistakes. The recommendation is to blink all money fields and move the cursor to the new *Total Amount Due*.

Late tapes might cause a justifiable denial of rental rights, but this has also never been discussed with Vic. The number of days that constitutes significant lateness needs to be defined. If monitoring of lateness is desired, a red, blinking value in the rental date field could be used to represent significant lateness.

Last, special fees, which require management update, might also be highlighted and a cause for rental denial. The use of special fees is not well-defined to the project team at this point. Presumably Vic is using special fees for lost or damaged tape assessments. Perhaps if the fees are over a certain amount, to be defined, Vic would want the field highlighted and, unless paid, rentals would be denied. If Vic wants this highlighting, a red, blinking field, consistent with other rental denial fields, would be suggested.

A long conversation with Vic resolves all of these issues. The recommendations for errors, credit problems, and insufficient payments are all accepted. Vic likes the idea of denying rental rights when tapes are over 10 days late. He questioned the use of the same blinking red signal, however, thinking that white blinking might be more effective. The SE explained that if one signal, blinking red, is used for rental denial regardless of reason, it will be more easily learned by the clerks. Vic agrees with the recommendation. He does not want special fees highlighted, nor does he want rental denied. He is using special fees for the two purposes described, but he also is using it for tapes purchases with money still owing, a usage never before defined.

Design of Report Output

In many companies, formal reports are no longer produced from application systems. Instead, users are provided with a query language and told to develop ad hoc reports as they are needed. When formal reports are required, they usually are based on queries of the same information. The guidelines for reports, then, follow similar guidelines for screens.

1. Design a standard header and footer and be consistent in the general format on all reports.
2. Keep report body as close to query screens as possible.
3. If query screens are not present for the specified reports, follow the design guidelines for screens. Define clearly identifiable areas for grouping information that is related or that repeats. Follow reasoning for individual fields on a report that parallels the reasoning used for screen design.

The ABC rental receipt is shown in Figure 14-49 as an example of a report that follows the design of its related screen. Notice that while the receipt has a header, it is preprinted and differs from that of the screen. Preprinted information is most effective when it is printed in some unobtrusive color, such as turquoise, which users can ignore when they become familiar with the report format.

CONVERSION

Conversion of applications is a systems analysis and design in miniature. The activity is only concerned with transforming data from its current format and storage media into a new application's format and storage media. Conversion is usually concurrent with design and done as a side activity by a small group of one to three people who report to the PM

ABC Video Rental
5930 Preston Rd.
Atlanta, Ga. 30303

Customer Information: #xxx999 MM/DD/YY
xxxxxxxxxxxxx xxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxx, xx 99999

Video ID	Copy #	Description	Rental Date	Return Date	Rent Fees	Late Fees	Other Fees
99999-	999	xxxxxxxxxxxxxxxxxxxxxxxxxxxx	mm/dd/yy	mm/dd/yy	99.99	99.99	99.99
99999-	999	xxxxxxxxxxxxxxxxxxxxxxxxxxxx	mm/dd/yy	mm/dd/yy	99.99	99.99	99.99
99999-	999	xxxxxxxxxxxxxxxxxxxxxxxxxxxx	mm/dd/yy	mm/dd/yy	99.99	99.99	99.99
99999-	999	xxxxxxxxxxxxxxxxxxxxxxxxxxxx	mm/dd/yy	mm/dd/yy	99.99	99.99	99.99
99999-	999	xxxxxxxxxxxxxxxxxxxxxxxxxxxx	mm/dd/yy	mm/dd/yy	99.99	99.99	99.99
99999-	999	xxxxxxxxxxxxxxxxxxxxxxxxxxxx	mm/dd/yy	mm/dd/yy	99.99	99.99	99.99
99999-	999	xxxxxxxxxxxxxxxxxxxxxxxxxxxx	mm/dd/yy	mm/dd/yy	99.99	99.99	99.99
99999-	999	xxxxxxxxxxxxxxxxxxxxxxxxxxxx	mm/dd/yy		99.99		
99999-	999	xxxxxxxxxxxxxxxxxxxxxxxxxxxx	mm/dd/yy		99.99		
99999-	999	xxxxxxxxxxxxxxxxxxxxxxxxxxxx	mm/dd/yy		99.99		
99999-	999	xxxxxxxxxxxxxxxxxxxxxxxxxxxx	mm/dd/yy		99.99		
				Total Fees Due:	999.99		
				Total Paid:	999.99		
				Change:	(99.99)		

Accepted By: ______________________________

FIGURE 14-49 ABC Rental Receipt

and work with the DBA to define and populate the new database environment. The activities of conversion are:

1. Identify current and future locations for all data items.
2. Define edit and validate criteria for all attributes.
3. Define data conversion activities.
4. Define options for data conversion.
5. Recommend and gain approval for data conversion strategy.
6. Develop a schedule for data conversion based on estimates of time to convert one data item.
7. Define options for application conversion and implementation.
8. Recommend and gain approval for implementation strategy.
9. Develop a schedule for application implementation.

Identify Current and Future Data Locations

The first task is to identify the data being converted. A matrix listing every relation with its attributes/fields is developed. Then, in one column, the present location of each attribute is identified. An automated data field entry has the current file, relative address in the logical record, length, type characters, and current data name. A manual field entry identifies the data source and person responsible for data accuracy.

A third column is created to identify specific conversion errors if they are known.

Define Attribute Edit and Validate Criteria

For attributes that are simply being moved from one location to a new location, the edit and validate criteria should already be defined in a data dictionary. If this information is not already defined, the conversion team defines and documents necessary edit and validate criteria.

When attributes are being encoded to use a shortened storage format, the encoding scheme must have been defined. If a coding scheme is not already defined, the conversion team works with the design team to define and document the encode-decode scheme.

Define Data Conversion Activities and Timing

Three major issues relate to data conversion. First, the automation status is either automated or manual; second, is data accuracy and reliability; third, is the ease of mapping from the old data storage technique to the new data storage technique (see Figure 14-50).

The extent to which data is already automated, clean, and has a simple mapping from the old to the new data storage technique, makes conversion simple. When data are manual, inaccurate, or not easily mapped, conversion is difficult. When data are all three—manual, inaccurate, and not easily mapped—conversion becomes a critical task that may define the critical path for the application development.

Manual data that must be automated require extensive edit and validation criteria in the data entry program to prevent bad data from getting into the database. Data that are not easily mapped may have no simple way for conversion staff to verify accuracy of processing, therefore, testing and test verification with user assistance become critical tasks in determining data conversion success.

Data that are inaccurate require two things. First, the conversion team must define what the possible correct data values are. Second, the conversion team and user must define the mapping from incorrect values to correct values. Then, any new values that might change the mapping from old to new storage technique must be reviewed with the systems design team to ensure that the application design is still valid. Third, an army of clerks must be hired to correct the errors. This means that special training for data correction is required. Fourth, training for the new application must address the data inaccuracies, the new values, and their interpretation for all current data users.

Data that have combinations of problems require multiple skills of conversion team members and complicate the conversion process. Data conversion planning should be complete early in the design stage. The planners should know which types of these problems are present and how the conversion team is planning to minimize their impact.

Select and Plan an Application Conversion Strategy

The methods of conversion are direct cutover and gradual conversion. Both methods may or may not be supplemented by continuing **parallel execution** of the old application to allow comparison of results and verification of processing.

Direct cutover means that on the set day, the old way of work is abandoned and the new way begins to be used. This is a risky method since few applications work perfectly the first time. There is no way to compare results and verify correctness of the new processing.

Gradual, or **incremental**, **cutover** is a conversion approach in which the new application is implemented in some piecemeal form. The implementations may be geographic, functional, iterative, or some combination of these. **Geographic** conversion is an approach in which the entire application is implemented in each location, one location at a time. The application that is used to account for pay telephones in the United States, COIN, has several different versions in operation across the country at a time. As a new version is implemented, one of the locations volunteers to be the first to use it. It is

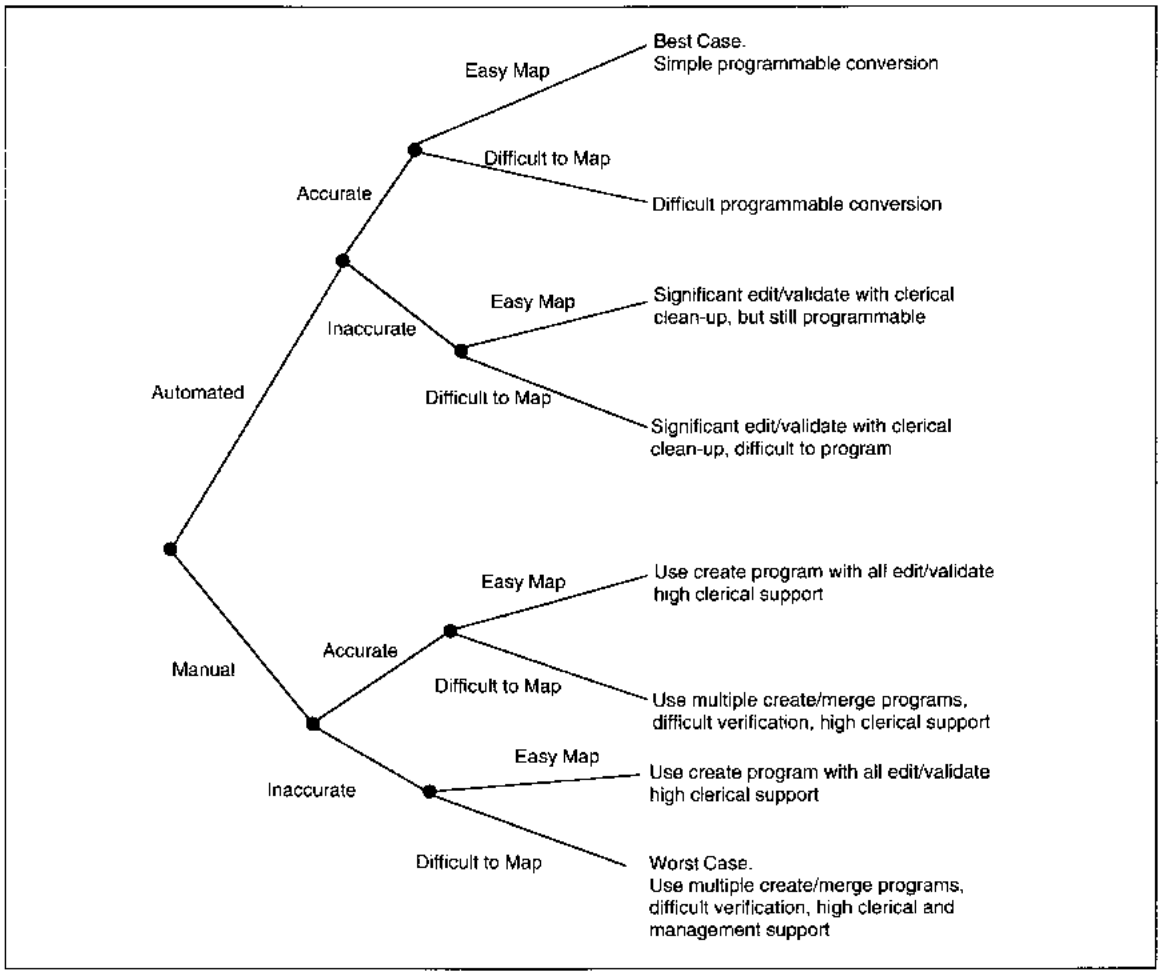


FIGURE 14-50 Decision Tree on Ease of Data Conversion

implemented in that one geographic location for six months. Then another location is added. After another six months, a third location is added. The timed geographic technique keeps the lives of the implementers relatively stable and allows the distributed companies using the software to choose their own implementation times.

Functional conversion has three variations. First, work functions can be cut over one at a time to the new application. This is a local version of the geographic conversion method. Second, **incremental software development** can place specific work functions into production use as soon as they are tested. Third, small numbers of transactions or one type of transaction might be implemented first using **transaction conversion**. Then, as the users gain experience and the application stabilizes, more transactions are cut over until all are in production. In the first variation, the entire application is implemented in one department or group at a time. In the second, pieces of the application are implemented one at a time, and may be in production company-wide or by group. In the transaction variation, the whole application is complete, but it is implemented piecemeal by transaction type.

When a new application changes the old method of work, or when a specific problem is highlighted during feasibility or analysis for immediate implementation, some form of functional, incremental conversion is useful. Both of these circumstances occur in large business applications. Small applications may not have enough functionality to allow iterative conversion, requiring the complete application to be placed into production at one time.

Gradual conversions can not always be done. When the new application is automating a previously manual process, gradual conversion may be difficult unless unrelated transactions can be identified. When this occurs, the project team should develop a final test using live data that parallels daily production and can, therefore, be checked for accuracy.

Parallel conversion means that the new and old methods of work, including any applications work, are both done every day for some period, usually one or two cycles of processing. Parallel conversions only work if the new application produces the same outputs as the old application and has comparable formulae and processing on the data. In the parallel method, the people using the application would do their jobs in the new way and follow it by doing the work in the old way *with the same data*. That is, the same information is processed twice. If the formulae, processing, or outputs are very different, parallel processing might not work. Parallel conversion is also difficult when the number of people doing the work is insufficient for processing the double volume of work. Then, if parallel conversion is desired, some gradual method should be coupled with parallel execution.

ABC Conversion Strategy

Conversion in ABC is from a totally manual to a totally automated application. This means that the planning for conversion should follow the need for data. Each relation is examined individually to determine its criticality for processing on the first day of Rental/Return use (see Table 14-9).

Of the seven relations in the application, four (i.e., Rentals, Customer History, Video History, and End of Day) are derived from processing and need no conversion. The other three—Customer, Video, and Copy—are manual and needed the first day of operation. All could have the same priority because the application cannot be tested without all three relations. The customer relation is given the highest priority because it has accurate data from the card file, and therefore, should be more easily converted. Another reason for choosing the customer relation first is because if it turns out to be error-ridden, the other two files can be assumed to be as bad or worse. Customers tend to overestimate the quality of their data, and errors become known when the method of processing changes.

The strategy then is convert the customer file from the existing card files, followed by the video and copy information. The next issue is who is to do the data entry. The clerks might enter *Customer* information during nonbusy work hours or could be hired for extra hours of work. The estimate of conversion for customer information is approximately 70 hours (4 minutes * 1,000 customers / 60 minutes in an hour). This assumes four minutes of data entry time for each of 1,000 customers. The ideal solution is to hire clerks for extra work so their entire attention is only on conversion at the time. This speeds the process and minimizes errors that might occur from interruptions during the work day.

TABLE 14-9 ABC Rental/Return Data Relations and Conversion

Relation	Status	Priority
Rental/ Return	Derived from Processing	0
Customer	Manual/Clean	1
Video	Manual Clean if known	2
Copy	Manual Need a count	3
Customer History	Derived from Processing	0
Video History	Derived from Processing	0
End of Day	Derived from Processing	0

One alternative for doing the data conversion is to hire the current staff to work more hours. If three ABC clerks each worked two extra hours each day, and all work a five-day week, the customer conversion would take between ten days and two weeks. This alternative is attractive because the current clerks know the data. The disadvantage of this alternative is that the clerks don't type and the four minute estimate might be very low for them. Another disadvantage is that because the clerks' typing skills are low, name and address errors, which are very difficult to identify via computer, might get into the file.

A better alternative is to hire an experienced data entry person(s) from a temporary agency. The cost is not too high, $10–14/hour, and their accuracy will be greater. For an experienced typist, the four minutes is probably a high estimate.

The next relations to be converted are *Video* and *Copy*. One issue in this conversion is the high amount of time for bar coding each copy of a video. Assignment of bar codes affects database design. Alternatives are to use the bar code to identify each tape uniquely and duplicate video information in the copy relations, or identify each video with a portion of the bar code and identify each copy by a unique sequence number within bar code. The preferred solution from a data perspective is to generate one *Video ID* bar code that is the same for all copies of a tape. Database storage and typing time are minimized, and retrievals will be faster. This solution is recommended. The only advantage to the other alternative is that no sorting of the physical inventory is required. The disadvantage of the unique base code for each tape alternative is that video information is replicated a number of times thus increasing the time for data entry, error rates, and retrieval time.

The related issue in video-copy conversion is the physical inventory identification of all copies of each video for entry into the application. The scheme we chose of one *Video ID* bar code for all copies of the same tape makes data entry easy but makes the physical work more difficult. The people doing this work must sort all of the tapes by video, assign the *Video ID*, and generate and affix the bar codes to each copy. Last, each copy's bar code must be entered into the system. Since we chose one *Video ID* bar code for all copies, we can enter the video information and a count of copies and have the application generate all *Copy* relations. Part of the change procedure for a video, then, must include changing the number of copies. Increasing the number poses no problems. Decreasing the number means that a check for outstanding or past rentals must be made and, if present for a number to be removed, the number may not be removed. These maintenance requirements should be discussed with the design team to ensure that they treat video processing in this way.

The last issue to decide about data conversion is who should do the video and copy conversion data entry. The estimated time for a complete physical inventory is about 28 hours. This number assumes six seconds of inspection time per tape for 10,000 tapes, plus four seconds overhead for extra movement of tapes to make room for the sorted ones (i.e., 10 * 10,000 / 60 seconds per minute / 60 minutes per hour = approximately 28 hours). This includes sorting the tapes by title alphabetically and keeping them in that order until the data are completely entered. Tapes out on loan must be included in each day's conversion process to ensure 100% conversion coverage. Once the tapes are in sequence, the clerks putting tapes back into inventory are assumed to alphabetize them automatically, adding no extra time to the conversion.

The data entry for each tape, because of the coding scheme defined, should take only about two minutes per tape for a total time of about 33 hours (i.e., 2 * 10,000 / 60). The total conversion time for the ABC rental/return application is about 120 hours, or about three weeks.

Again, the clerks, who know the inventory best, could be hired extra hours to work on conversion sorting and data entry, or Vic might hire outside workers to come in daily for 8–10 hours for several days.

If Vic wants to use his current clerical staff to use otherwise idle time, the amount of time for conversion is 120 hours divided by the number of idle hours per day. If the three clerks are idle a total of six hours per day, the conversion will take approximately 20 days. This is a long period of time and usually, the longer conversions continue, the greater the likelihood of errors. The recommended approach is to hire

temporary data entry clerks to sort the tapes, assign bar codes, and enter the data into the system.

The alternatives and recommendations are presented to Vic for his approval. He chooses to hire two temporaries for two weeks to work full-time on converting all data. His rationale is that he really wants his clerks to concentrate on customers, and he decides they can help with the physical inventory sort in their spare time. The remainder of the time they should be working at helping customers. If videos are missed during the inventory sort, they will be found as they are rented and their information will be entered into the application then.

USER DOCUMENTATION

Mix of On-Line and Manual Documentation

User documentation is important because it is usually the first information about an application that new employees are given. Therefore, it should be developed and maintained to disclose accurate usage information about an application. User documentation is started after analysis and can be a parallel activity to design. Some researchers and practitioners recommend developing the user documentation before design begins. The application is then designed to meet the requirements of the user documentation.

Frequently, *users* develop the manual documentation and define what they would like for on-line help and messages. At the least, users should participate in developing user documentation. The arguments for having users develop their own documentation are:

- Users are less likely to assume knowledge that SEs take for granted (e.g., how to start an application).
- Users know what to do better than SEs.
- Users who develop their own documentation require less training because they already know how the system will work.

With complete novices who have never used a computer system, having them develop the user manuals is NOT a good idea.

Contents of the user documentation vary with each project and company. In general, the writing style should not be patronizing, but should take the users' general level of computer expertise into consideration. This means that documentation written for experts can be concise, use jargon, and have less explanatory information about how to get started. Documentation written for novices should begin at an elementary level, for example, "The button to turn on the machine is located. . . ."

An outline for general contents of user documentation is provided in Table 14-10. First, any document should contain a table of contents. A system overview describing the scope of processing is next. Assumed level of user and expected system-user interactions should be included in the overview. Diagrams should be frequent and 'understood by your mother.' Also in the overview, include information about whom to call for help and what kind of help they offer. For instance, Operations provides assistance if the terminal malfunctions, or the Information Center assists in developing ad hoc queries.

Describe the hardware, software, and at a very high level, how the equipment is connected. This is especially important when LANs, distributed applications, or PCs hooked to mainframes are being used and some functions are local and some remote. Be specific about what work is performed in what location and how to determine problems.

Next, describe the general formats for screens and functions. Begin the details of system operation with startup and shutdown, including security information, without documenting security codes! Describe all function keys and what they do.

Then, for each screen in the application, present the screen and the required/optional entries made by the operator. Be specific about the type of data to be provided. Present an example of a correct screen and of an incorrect screen with error messages. Sequence this information by logical groupings of activities. For instance, for ABC, there would be four functional description sections: rental/return, customer maintenance, video maintenance, and

TABLE 14-10 User Documentation Contents

Introduction
- Application Overview
- Special Features
- Format of Document
- Support Group Services, Contacts

General System Information
- Obtaining a User ID
- Starting the Machine
- Shutting the Machine Down

System Access Procedures
- Logon Procedures
- Logoff Procedures

General Data Entry Information
- Menus and Menu Selection with examples of all screens
- Data Entry Screen Format with one example screen
- Function Key Assignments

Rent/Return Procedures
Customer Maintenance Procedures
Video Maintenance Procedures
Periodic Processing Procedures
Backup/Recovery Procedures
Error Recovery Procedures
Error Messages

For each section:
- List screen(s)
- Required entries
- Optional entries
- Procedure for screen completion

periodic processing. For each screen, describe normal, error, optional, and required processing.

Include backup and recovery information if the user is expected to perform those activities. Be specific about what actions are performed and the sequence of actions. If recovery must be activated from a specific terminal, for instance, begin the instructions with something like the following. "At Terminal 011, located on the 2nd floor of 235 West Covina in the southwest corner, and labeled 'MAIN OPERATOR TERMINAL,' enter the following."

In an appendix, provide a list of all error messages, by message ID with a detailed description of how to correct the error. Format the appendix to correspond to the sequence of functional sections in the body of the report.

AUTOMATED SUPPORT FOR FORGOTTEN ACTIVITIES

Many products are available to support the activities in this chapter. For screen design, screen 'painters' and application generators both provide screen design. **Screen painters** are forms-oriented design tools that allow fast prototyping and layout of screens that then generate coded descriptions of the screens. A user identifies that screen design is desired; if the relation is described in the tool, the fields can be listed to provide screen design guidance, and the user 'paints' the screen by placing labels and field names on the screen in the target location. When complete, the screen can be called up to allow printing and viewing of the screen as it would be presented to the data entry clerk. Screen painters can be stand-alone software packages but are more frequently a function of CASE environments.

A second type of software support for screen design is available in application generator software. The screens for menus are designed first with menu entries typed in by the software user. Then as functional screens are reached, the program code to generate the requisite screen interaction (e.g., SQL) is coded. If custom form design for data entry is required, some packages include that activity, too; others require the designer to generate the code within the package.

Conversion software support is mostly in the form of utility programs that allow easy reformatting of data to move from a current automated file to one or more new files. Merging of information from two sources to create new composite files is sometimes provided but requires more complex software coding.

Manual-to-automated data conversion ideally uses the application code for data creation to further test it and increase estimations of reliability. Sev-

eral application generator packages, for example, Focus™,[5] provide automatic screen generation with no underlying edit or validation for 'quick and dirty' data entry. This is useful in prototyping and demonstrating prototypes, but should not be used for the production application. Focus generates the screen by sequentially listing the fields as defined in the database. As a line fills up with data, a new line is generated. This automatic screen utility only works on files with no repeating information and cannot join files for combined data entry.

Help packages are now plentiful in the marketplace. Help used to be totally manual and all messages had to be in the user documentation. As help has moved to become an on-line function, more messages are documented on-line than in manuals. The advantage of a Help package that is independent of specific software is that it, and its messages, can be used across applications and software environments. This cross-application use can help ensure that definitions are consistent throughout the company and can make data administration standards compliance easier to monitor.

The automated packages supporting the screen design, conversion, and help processing are summarized in Table 14-11.

SUMMARY

In this chapter, human interface, conversion, and user documentation were discussed as three required activities during analysis and design that are omitted from many methodology discussions.

Human interface design focuses on screen interactions between users and the application. Using a task profile and user profile to guide the design process, first the option selection method is chosen. The alternatives for option selection are menus, windows, or command languages. Then, the presentation format(s) most effective for the data to be displayed are decided. Presentation formats include analog, digital, text, text form, bar chart, column chart, point plot, pattern, and mimic displays. Within the presentation format, each screen item's characteristics of size, type font, style, color, and blink rate are defined. In designing forms, decisions about the chunks of data to be presented and formatting of chunks on the screen are required.

Conversion alternatives are direct conversion or incremental conversion. Incremental conversion may be geographic or functional (by transaction, by department function, or by application function). Direct conversion has the highest risk of failure because the old method disappears at conversion; therefore, when an alternative is present, it is usually recommended. Incremental conversion type selected is determined by the context of the application.

Reports are designed following the same general guidelines as those of screens. Whenever a report is of displayed information, both screen and report should use the same format.

User documentation is an important introduction to an application for many new employees. As such, it should be easy to read, oriented toward the education and computer experience level of the reader, and should include all information for normal and abnormal processing of an application. Lists of contacts for different types of problems should be identified.

5 Focus is a product of Information Builders, Inc., New York.

REFERENCES

Bailey, R. W., *Human Performance Engineering: Using Human Factors/Ergonomics to Achieve Computer System Usability*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1989.

Banks, William W., Jr., and Jon Weimer, *Effective Computer Display Design*. Englewood Cliffs, NJ: Prentice-Hall, 1992.

Carter, R. C., "Visual search with color," *Journal of Experimental Psychology: Human Perception and Performance*, Vol. 8, 1982, pp. 127–136.

Christ, R. E., "Review and analysis of color coding research for visual displays," *Human Factors*, Vol. 17, 1975, pp. 542–570.

Cohen, Barbara F. G. (ed.), *Human Aspects in Office Automation*. New York: Elsevier, 1984.

Galitz, Wilbert O., *Human Factors in Office Automation*. Atlanta, GA: Life Office Management Association, Inc., 1980.

TABLE 14-11 Automated Support for Interface Design, Conversion, and On-Line Documentation

Product	Company	Technique
APS Dev. Center	Sage SW Rockville, MD	Screen/Form/Report Painters
Deft	Deft Ontario, Canada	Form/Report Painter
Easytrieve	Ribek, Inc. Tacoma Park, MD	Data Conversion Utility
Focus	Information Builders, Inc. New York, NY	Prototyper Screen Generator Application Generator
Foundation	Arthur Anderson & Co. Chicago, IL	Prototype Generation Screen Design Version Control
IEF	Texas Instruments Dallas, TX	Dialog Flow Screen Design
IEW, ADW(PS/2 Version)	Knowledgeware Atlanta, GA	Screen Design
PacBase	CGI Systems, Inc. Pearl River, NY	Screen Flow
Teamwork	Cadre Technologies Inc Providence, RI	Screen Painter
Telon and other products	Pansophic Systems, Inc. Lisle, IL	Screen/Report Layout
Visible Analyst	Visible Systems Corp. Newton, MA	Screen Painter/Prototyper

Galitz, Wilbert O., *Handbook of Screen Format Design*. Wellesley, MA: QED Information Sciences, Inc., 1981.

Martin, James, *Design of Man-Computer Dialogues*. Englewood Cliffs, NJ: Prentice-Hall, 1973.

Mayhew, D. J., *Principles and Guidelines in Software User Interface Design*. Englewood Cliffs, NJ: Prentice-Hall, 1992.

Morland, D. Verne, "Human factors guidelines for terminal interface design," *Communications of the ACM*, Vol. 26, #7, July 1983, pp. 484–494.

Powell, James E., *Designing User Interfaces*. San Marcos, CA: Microtrend Books, 1990.

Olsen, Dan R., Jr., *User Interface Management Systems: Models and Algorithms*. San Mateo, CA: Morgan Kaufmann Publishers, 1992.

Schneiderman, Ben J., *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Reading, MA: Addison-Wesley, 1987.

Thomas, John C., "User interface design," *Proceedings of NYU Symposium on Human Factors*, New York, NY, May 1982.

Tullis, T. S., "Screen design," *Handbook of Human Computer Interaction*, Mark Helander (ed.). New York: Elsevier, 1988, pp. 377–411.

KEY TERMS

analog display
band chart
bar chart
binary
binary display
body of form
body of screen
classroom instruction
close box
column chart
command language
computer-based training (CBT)
derived field
digital display
direct cutover
direct manipulation
direction indicator
field format
flash rate
flicker fusion
footer screen section
form screen
functional conversion
geographical conversion
gradual cutover
header screen section
horizontal pull-down menu
incremental cutover
incremental software development
location ID
long-term memory
Lotus-style horizontal pop-up menu
menu
mimic display
normal/abnormal measures
on-the-job training (OJT)
option selection
overlapping windows
parallel conversion
parallel execution
pattern display
paint
point
point plot
pointer
precision requirements
question & answer format
resize box
scale
screen painter
scroll arrow
scroll bar
scroll box
scrolling elements
short-term memory
status indicator
task profile
text
tiled windows
title bar
transaction conversion
user profile
vertical pop-up menu
window
zoom box

EXERCISES

1. Complete the screen design for *Customer* and *Video* data entry for ABC Video. For video data entry, keep in mind how conversion defines the add function to automatically provide for *Copy* relation creation. Specifically, identify reused portions of screens or whole screens for different functions. Discuss why complete reuse of *Create Video* screens is not possible for *Video Update* processing.
2. For the CCD Medicaid case described in Appendix A, design windowed menus for the application. Design the screen for Patient Information Creation. How much scrolling is necessary? What colors, type, style, font, and so forth, do you recommend for each field?

STUDY QUESTIONS

1. Define the following terms:
   analog display
   field format
   flash rate
   form
   horizontal pull-down menu
   OJT
   scrolling elements
   user profile
   task profile
2. Why is the data source the best location at which data should be entered into automated applications?
3. Why should screen design guidelines be followed?
4. Describe a task profile and how it is used in the application development screen design and conversion.
5. When should individual users be profiled and when can average user information be used?
6. Describe how novice/expert modes of operation should be determined.
7. Describe how extent and type of on-line messages and help are defined.
8. Describe the option selection choices and how you decide which to use.
9. Why is command language use by itself rare?
10. What is a screen window and why are they popular?
11. How many scrolling options are available? What is the minimum scrolling that should be provided in an application?
12. What are the differences between tiled and overlapped windows?
13. Why should function keys be consistent?

14. Describe general screen design contents.
15. What is direct manipulation interface?
16. What application types use forms as the most common functional screen design?
17. List and define five data presentation alternatives. For each alternative, describe one possible business application use.
18. When are bar and column chart use recommended?
19. How are fields positioned on a screen? On a line?
20. Why are short-term memory (STM) and long-term memory (LTM) important in screen design?
21. When is color effective in screen design? How many colors should be used on screens at any one time?
22. How can type font be varied for effective screen design?
23. What are three options for incremental conversion? How do you choose which to use?
24. Discuss issues in data conversion.
25. Why should users do user documentation? Why should application developers do user documentation?
26. Discuss how contents of user documentation can be varied to match user skills and computer expertise.

★ EXTRA-CREDIT QUESTION

1. Define a poorly designed menu and functional screens for ABC Customer Maintenance. Use at least 10 bad design elements. Then, fix the design problems and define effective screens for the same function. Describe the guidelines followed in defining each element of the good screens. Write a paragraph discussing the kind of errors that users might make from using the poorly designed screens.

PART IV

IMPLEMENTATION AND MAINTENANCE

The five chapters in this section discuss implementation and maintenance issues. An application is never completed until it is retired. After analysis and design, we must be able to implement the design on computer hardware using computer software or our work is useless. The first three chapters in this section relate to implementation issues: selecting a computer language; evaluating and selecting hardware, software packages, or consulting services; and testing/quality assurance of the finished product.

Chapter 15 defines characteristics of languages, to allow us to distinguish between ten languages that are evaluated. Then, the languages are matched to the application types discussed in Chapter 1 and to the methodologies discussed in Chapters 7–12. Language selection, rather than code structure, is emphasized because of the increased use of computer-aided software engineering (CASE) tools to generate code. The language selected must be able to support the application requirements. In Chapter 15, we first describe identifying characteristics of languages. Then, the implementation of each characteristic is described for ten languages. Based on the language characteristics, we define the types of applications for which each language is best suited.

Similarly, outsourcing and use of software packages are growing in all industries because it is frequently cheaper to *buy* rather than *build* an application and/or its environment. In Chapter 16, we discuss the evaluation process and highlight the types and alternatives for soliciting bids from vendors. Sections and contents of a request for proposal (RFP) are defined and developed for the ABC case to show what they look like. Hardware, software, and consulting services might all be contracted for in the same request, or could individually be the subject

of RFPs. Examples of RFP expectation criteria for each type of work are provided to give a sense of the level of detail to which work is defined in an RFP. Then, vendor proposal evaluation alternatives are defined and discussed in relation to ABC Video's application.

Regardless of the development product—packaged software, generated CASE code, or manually programmed code—proving that the software works by testing it at various levels of detail and aggregation is required. Chapter 17 defines the different strategies for testing and types of testing performed. Test types are matched to strategies to develop an effective overall strategy for testing applications. For each level of testing, key issues in test case development are identified. Based on research on testing errors found, guidelines for deciding when to stop testing at each level are provided. The ABC case is then analyzed to demonstrate how the theories apply in practice.

The last two chapters relate to change. Chapter 18 discusses application change management that all take place throughout the life of a project. Change is a way of life in computing and application development is no exception. In Chapter 18, we first discuss how to design for reusability by using templates and reusable modules. Then, change management techniques that apply to documents, decisions, software, and application configurations are presented. The automated tools section includes software representative of each type of change management.

Documentation for project work can be thousands of pages long. Since errors in code usually begin to be traced through documentation, it is important to identify changes to facilitate the error tracing process. Also, users and maintenance personnel who might only infrequently review documentation should be directed to the new information rather than having to read entire documents each time. The techniques for identifying change easily are identified in Chapter 18.

Similarly, application decisions might provide a useful trace of the considerations and discarded ideas throughout a project's life. Few project teams keep such a decision trace because, historically, to do so meant maintenance of more thousands of pages of paper. With automated decision support and sophisticated word processing, keeping a record of decision history is now feasible and can be useful in organizations with rapidly changing management or on projects that support business functions that are subject to rapid industry change.

Software changes and application configuration management are the other major topics of Chapter 18. A recent buzzword identifies *software reengineering*, also called *reverse engineering*, as the backward design of undocumented programs and applications that were probably built without the team having followed a methodology to guide the work. Also called *spaghetti code*, such applications can be maintained beyond a useful life. In the chapter, we describe how to decide when to reverse

engineer, reengineer, or retire applications and/or individual programs. Once the decision is made to maintain software, management of the software maintenance process is an important task in determining that the correct configuration of modules, functions, programs, and so on, is in production. The issue of configuration management is more complicated when multiple versions of software, such as a DOS and MVS versions, exist. Techniques and management practices for configuration management are described in the chapter.

Finally, your career is important and requires management by you for your working life. It is difficult to plan a career without having a sense of what opportunities and expectations are available. First, the typical job levels and types of jobs found in businesses are described. Then, one way to plan a career by thinking through your wants and requirements for technical, job, company, geography, and opportunities for advancement is developed. A method for defining your chances of job success is defined next. Trends of IS jobs over the last five years by geography, salary, and industry are discussed. Part of developing yourself into a professional and having a career is to maintain your professional status. Techniques for maintaining professional status and building on knowledge areas including education, professional association membership, accreditation, and reading are all defined, with suggested approaches to applying the information to your own situation.

CHAPTER 15

CHOOSING AN IMPLEMENTATION LANGUAGE

INTRODUCTION

In this chapter, we discuss the selection of a language for implementing an application. **Programming** is the process of designing and describing an algorithm to solve a class of problems. As any programmer knows, any activity *can* be programmed in *any* language . . . just not necessarily as effectively or completely in each language. When working on an application, we do not always have a choice of the language we use. But with the selection of the wrong language, we constantly compromise the requirements to fit the constraints of the language. In this chapter, we discuss characteristics of languages and how to select a programming language based on requirements of an application so that, if there is a choice to be made, an appropriate language can be selected. The activity of programming is not discussed in this text because, with CASE environments and tools, much program code is automatically generated.

First, the characteristics of languages are defined. Then 10 computer languages—SQL, Focus, BASIC, COBOL, Fortran, C, Pascal, Ada, PROLOG, and Smalltalk—are evaluated according to the characteristics. These languages represent the major programming paradigms, including procedural (Fortran, COBOL, BASIC, Pascal), object orientation (Smalltalk, Ada), declarative processing (SQL, PROLOG), fourth-generation languages (4GL, Focus), and expert systems (PROLOG). They also represent the most popular languages in use in business organizations today and in the years to come. Then, languages are matched to different types of applications and methodologies. Finally, automated support for programming is discussed. First, we develop the characteristics that distinguish between languages.

CHARACTERISTICS OF LANGUAGES

To differentiate languages, we must evaluate how each language deals with data definition and processing, mathematical and logical processing, control, conditional, array, input/output, and subprogram processing in addition to nontechnical assessment of each language's ease of use, portability, and maintainability. Finally, available automated development aids such as CASE and code generators are noted.

Data Types

Each language supports some data types. A **data type** is a language-fixed definition of data. All languages support variables and constants for numeric

Data Type	Example
Integers	1, 2, 3
Real	−1.01, 3.21
Character/String	Abc12;'.

FIGURE 15-1 Examples of Universal Data Types

and character data. The universally supported data types are integers, real numbers, and character strings. Example of each are shown in Figure 15-1. **Integers** are whole numbers such as one, two, or three. **Real numbers** include positive and negative continuous numbers, including all decimals. **Character strings** are any legal combination of alphanumeric characters.

Fewer languages support one or more of logical, Boolean, pointer, object, bit, date, or user-defined data types. **Logical data types** are notation providing for nonnumeric comparison including *and*, *or*, or *not* processing (see Figure 15-2 for example). Also, the comparison operators used in logical data types include all variations of equality and inequality operators (see Figure 15-2).

Data Type	Example
Logical	And, Or, Not, <, >, =, ≤, ≥, ≠
Boolean	True, False
Pointers	16F26 (where 16F26 is a valid memory address)
Object	Customer=12346, Add, Change, Delete, Inquire
Bit	0, 1
Date	02 28 93

FIGURE 15-2 Examples of Nonuniversal Data Types

Boolean operators generate binary true/false indicators based on some logical comparison (see Figure 15-2). **Pointers** are addresses of other program or data constructs that are used for reference within a program.

Objects are programmed encapsulations of data with methods. The example in Figure 15-2 shows only the names and ID of an object with the names of the methods or program modules that can manipulate the data. In actuality, an object contains all of the data and all of the program code for the methods.

A **bit** is an individual binary digit (see Figure 15-2). Bit manipulation is highly desirable in programs using binary status indicators. In an eight-bit character set, use of one bit rather than eight to indicate a single value can save millions of characters of storage space.

Date data types define combinations of months, days, and years that support only legal date entries (see Figure 15-2). Rather than writing routines to validate dates, the language may have built-in validation processing.

Finally, **user-defined** data types are data definitions that become fixed within a program or application. User-defined data types can be for any application-specific combination of legal characters. A common user-defined data type is for a date construct when the language does not provide a date data type.

Data Type Checking

Data type checking refers to the extent to which a language enforces matching of specific data definitions in mathematical and logical operations. There are four levels of type checking, ranging from typeless to strong checking. Which level is required is dependent on the application type. In general, the more stringent the requirements for accuracy and consistency of processing, the more desirable strong type checking becomes. With object methodologies, strong checking is desirable because with polymorphism, the ability to have multiple modules processing the same function but on different data types,

```
01   COBOL-INFO.
     05    EXAMPLE-NUMBER        PIC 9(5).
01   TARGET-INFO.
     05    TARGET-NUMBER         PIC 9(5).
     ....
PROCEDURE DIVISION.
     Move 'A124X" to COBOL-INFO. *** Causes no
       errors ***
     Move COBOL-INFO to TARGET-INFO.
       *** Causes no errors ***
     Move EXAMPLE-NUMBER to TARGET-
       NUMBER. *** Abend—Illegal data in
       EXAMPLE-NUMBER ***
```

FIGURE 15-3 Cobol Typeless Checking

the probability of errors is reduced with strong type checking.

Typeless checking means that there is no explicit checking performed. In typeless languages, such as BASIC or COBOL, alphanumeric characters are allowed in an integer field, but might cause an abend if the field is referenced as an integer (See Figure 15-3). Operations using typeless fields are not guaranteed to execute successfully. Typeless field processing is not consistent across languages or compilers.

The next level provides **automatic type coercion** in which mixed data types are allowed, but conversion of incompatible types occurs when used together. Also called **mixed mode type checking**, different data types within a category (e.g., numeric) are converted to a single target type for mixed mode operations. In Fortran, for instance, mixing a real and integer number in a mathematical operation leads to unpredictable results because the target type is determined by the result field definition (see Figure 15-4). If the result field is defined as real, the process will yield a real number. In Fortran, the first character of a field determines its data type. Names beginning with A–H and O–Z are real; names beginning with I–N are integer. In Figure 15-4a, the result field begins with B; therefore, the result field is a real number. If the result field is defined as integer, the process rounds the answer and the result is integer. In the example in Figure 15-4b, the answer is either zero or one depending on the computer system and how it rounds integers. Obviously, without detailed knowledge of the internal language processing, programming errors can result.

Pseudostrong type checking, the third level of data type checking, permits operations only on data objects of the same data type when they are defined in the same module. But, unlike strong type checking languages, there are language inconsistencies, or *undocumented features*, that allow programmers to mix data types. Pascal is a pseudostrong type checking language in that it supports strong typing *within* modules, but has no type checking *across* modules. So, data passed from one module to another for processing may be combined in the called module with another data type with no penalty.

At the highest level of data type checking, languages with **strong type checking** permit operations only on data objects of the same, prespecified data type whether in the same or other modules. If a module contains an illegal data type, the application would stop processing and issue an error message. Ada provides strong type checking.

Language Constructs

Language constructs determine what and how operations on data are carried out. They provide for sequencing, iteration, selection, and data structure

a. The formula is: I/A = B
   5/10.0 = 0.50

   The data are converted to real because B is a real name.

b. The formula is: I/A = J
   5/10.0 = 1.0 *or* 0.0

   Data are converted to integer and rounded. Results vary depending on the computer system.

FIGURE 15-4 Mixed-Mode Data Type Checking

processing, and differ for each language classified. In general, the richer the language, the more these constructs will be present. However, with the richness comes a trade-off in language complexity that forces users to learn more language details to become proficient.

The need for rich language constructs depends somewhat on the language paradigm. For instance, **SQL** is a declarative, set processing language that does not need iteration because iteration is embedded in the language. In a declarative language, you code *what* you want to do, not *how*. With *set* processing, you identify the database and the language controls all file manipulation. The more procedural the language, the richer the language constructs need to be. The more detailed the application, the richer the language of the application should be.

Sequencing occurs between and within commands. Between-command sequencing is controlled by you as the programmer who defines the order of commands. Intracommand sequencing is part of language definition and is called operator precedence. **Operator precedence** is the prioritizing of symbols to manipulate data. All languages have at least four arithmetic symbols in common: + for add, – for subtraction, * for multiplication, and / for division. Most languages also have many other symbols and operations supporting unary and binary operations including relational processing (e.g., "less than," "less than or equal," etc.), logical processing (e.g., "and," "or," or "not"). A list of operators available in different languages is provided in Figure 15-5.

Operator	Symbol
Add	+
Subtract	–
Multiply	*
Divide	/, ÷
Exponent	**, ^
And	AND
Or	OR
Not	¬
Equal	=
Less	<
Greater	>
Less or equal	≤, =<, <=
Greater or equal	≥, =>, >=

FIGURE 15-5 Language Operators

Control language constructs support iteration, sequential or selection processing via loops, exits, conditional statements, or case constructs. **Loops** provide iterative, repetitive processing and are usually supported through structured programming notations such as "do while . . ." or "do until. . . ." **Conditional statements** support "if . . . then . . . else" processing. Conditional statements are used in some languages to control iterative loop processing. Common loop notations are shown in Figure 15-6.

Case statements allow identification of code segments that combine to identify the "case," for example, in Focus file maintenance processing you can code screen processing cases for add, change, and delete cases. This simplifies the thought processes involved in programming by "chunking" case contents.

Exits leave the current code module and return to the calling module or to some other named module. Exits can be simple returns to the calling module, such as *Return*, *Cut*, or *Exit* statements (see Figure 15-7); exits can indicate the nature of the end as in PROLOG's *Fail* exit, or exits can return to a named module in a *Goto* statement.

Arrays, or **tables**, are a third type of language construct that may or may not be supported by a language. Linear arrays, or lists, are one type of data that are relatively simple to support (see Figure 15-8). When higher dimension arrays are supported, the maximum number of dimensions are identified. Occasionally a language will support *n*-dimensional arrays, with a user-defined maximum.

Next there are four possible alternatives for **physical input and output (I/O)** of information to and from automated files or data entry fields. First, specific I/O statements (e.g., read/write) for externally stored data may be one of three types: record-oriented, set-oriented, or array-oriented. **Record-oriented I/O** reads (or writes) a physical record of

BEGIN . . . END
BLOCK
DO . . . ENDDO
FOR . . .
FOR . . . END FOR
ifFalse . . .
ifTrue . . .
INDEX . . .
LOOP . . . ENDLOOP
REPEAT . . .END
REPEAT . . .
WHILE . . .
WHILE . . . ENDWHILE
whileFalse . . .
whileTrue . . .

FIGURE 15-6 Loop Notations

information that may contain one or more logical records. Recall from database class that records (or tuples in relational terminology) are groupings of related fields. Record-oriented I/O requires opening and closing of files, reading or writing of records, and user management of all file processing, such as checks for end-of-file. COBOL, Fortran, Assembler languages, and Ada are record-oriented.

Exit Type	Processing
Return	Return to Calling Module
Cut	Return to Calling Module/Instruction
Exit	Return to Calling Module
Fail	Go to Calling Module/Instruction with Boolean indicating process failure
Goto	Go to Named Module

FIGURE 15-7 Exit Types

Linear Array, List

1
2
3
4
5

Two Dimensional Array of Months and Days

January	31
February	28
March	31
April	30

Three Dimensional Array of Sales By Year By Month

Year	Month	Sales
1996	January	220,000
1996	February	250,000

Year	Month	Sales
1995	January	150,000
1995	February	170,000

Year	Month	Sales
1994	January	100,000
1994	February	100,000

FIGURE 15-8 Types of Arrays

Set-oriented I/O assumes that all *records* (or tuples) are treated the same and that some selection criteria, when applied, identify the desired information. The language controls all file and read/write processing according to user-defined selection criteria. At the end of a procedure, the set of records (tuples) resulting from the procedure are stored in memory for printing or display. SQL is set-oriented.

Implicit I/O is similar to set-oriented I/O. Implicit I/O is used in 4GLs in which reading and writing of data is hidden from the user. The user specifies the type of process, for instance, TABLE FILE . . . , and the language infers the type of file processing required from the command. Set-oriented I/O is

more rigorously defined and has provably correct contents based on mathematical set theory which underlies relationship processing. Implicit I/O, on the other hand, is in languages which predate relational theory and do not have provably correct results.

Array-oriented I/O reads and writes strings of fields that are assumed to be some sort of array. The user is responsible for defining and manipulating the nature and data type of array. The language simply reads or writes until the end of the array. Pascal is an array-oriented language.

List-directed I/O is a variant of array-oriented I/O. **List-directed I/O** is used in Fortran to define a list of variable names to which items are *directed* as they are read. The language reads until the list is full, then continues processing until the read is again executed. Data items are not specifically formatted, rather the format is implicit in the variable names.

The extent to which data formats and I/O processing can be defined and controlled distinguishes languages as I/O-oriented versus CPU-oriented in their processing. The more elaborate the I/O processing, the more I/O-oriented the language. The more primitive the I/O processing, the more CPU-oriented the language. Fortran is an example of a CPU-oriented language, while COBOL is an example of an I/O-oriented language.

Modularization and Memory Management

The extent to which modularization and memory management are supported is an indication of language sophistication. **Modularization** is the creation of subprograms or stored functions. Languages differ in the manner in which the subprogram and their data are supported. First, the ability to define subprograms or functions is important to attaining desirable program characteristics such as maximal cohesion. Not all languages allow subprograms. In particular, set-oriented languages (SQL) do not easily support subprograms.

Second, how data in modules is managed is important. Data can be local or global. **Local data** storage defines data variables and constants that are only used within a given module. **Global data** are accessible to any module in the application. The ability to have local data is important to attaining information hiding and minimal coupling. The extent to which global data is required limits the quality of resulting programs by limiting information hiding and cohesion.

Subprograms' activation is similar across languages. Called modules are referenced by module name. For instance, "CALL FACTORIAL, 5" might be a subprogram call that passes the value five for factorial computation. Modules must reside in a library that is linked to the calling module via control language (e.g., JCL). Options for call processing include passing of variable data either by name, by address, or directly, by value. Value passing requires local data definition while passing data by name or address is used with either local or global data.

Generally, when using subprograms, a main module calls the subprogram which performs its processing and returns to the calling module. The ability to support subprogram processing requires one or more entry and exit points. Exit and return processing are also important when passing control of processing between modules. In general, the more opportunities to enter and exit a given module, the more proficient the programmer needs to be to ensure proper processing. According to structured programming tenants, a well-designed module should have one entry and one exit point. Some languages, such as Smalltalk and Ada, enforce this idea by allowing only one entry and one exit per module. One entry–one exit modules are less error-prone than modules that allow many alternatives.

The next level of sophistication is the extent to which programmers have control over their own memory management. **Memory management** refers to the ability of a program to allocate more computer memory as required. This is an option frequently desired in variable list processing and real-time applications that manage multiuser resources. Memory in less sophisticated languages is **static**: The program is assigned a maximum at the time it is initiated for processing. If more memory than that allocated is needed, the program abends, more memory is requested manually via job control language, and the program is rerun.

With **dynamic memory management** capabilities, the program monitors its own use of storage and allocates more memory as needed. In sophisticated languages, the capability to dynamically allocate memory is present.

Exception Handling

Exception handling is the extent to which programs can be coded to intercept and handle program errors without abending a program. This capability adds to both the complexity and the range of usefulness of a language. This capability ranges from none to some. For instance, COBOL allows you to intercept data errors such as overflow or divide by zero, but not others, such as invalid data definition or read past end-of-file. In contrast, Smalltalk allows the interception of any error.

Multiuser Support

The extent to which language constructs for memory management, global/local variables, and subprogram management are available, determines the extent to which a language can support multiple users. There are three levels of support for multiple users that relate to program modules having the properties of reusability, recursion, and reentrancy. **Reusability**, also called serial reusability, is a property of a module such that many tasks, in sequence, can use the module without its having to be reloaded into memory for each use (see Figure 15-9). To accomplish this level of program, any changes to local variables must be reset to their original contents before the completion of processing and return to the calling module. The easiest way to develop reusable programs is to provide global variables that can change contents and local variables that either cannot change or are always reset after the module's use. Reusable programs can support sequential or interactive processing, but not multiuser or real-time processing.

Recursiveness is a property of modules such that they call themselves or call another module that, in turn, calls them. An example is factorial multiplication in which the same process is performed on a different number of variables a number of times (see Figure 15-9). Processing with recursion is explicitly outlawed in some languages, while it is considered a main strength of others, such as PROLOG. Recursion requires serial reusability of programs in addition to the ability to maintain a queue (or stack) of outstanding requests to be completed. This queueing support provides for multiple uses of the module by one user.

Reentrancy is a property of a module such that it can be shared by several tasks concurrently. There is a constant part and a variable part to each reentrant

Reusable Pseudo-code

```
Factorial (N, Nfact)
End=0
If N=0 or 1
        go to exit.
Loop.
        If N=1
                go to exit
        else
                Nfact = N * (N-1)
                N = N-1
                go to Loop.
Exit. Exit.
```

Recursive Pseudo-code

```
Function FACT (N)
Begin
        If N =0
                Then Factout = 1
        Else Factout = N * FACT(N-1)
End {Function Fact};
```

FACT is a function that recurs continuously until N = 0.

Reentrant Pseudo-code

```
Load N, Nfact, First-Exec
If N = (0 or 1) and First-Exec = 0
        Then Nfact = 1
Else
        If N > 1
                Nfact = N * (N-1)
                N = N-1
                First-Exec =1
                Save N, Nfact, First-Exec.
```

FIGURE 15-9 Examples of Reusable, Recursive, and Reentrant Modules

module. The constant part is loaded into memory once and it services tasks in a serially reusable manner until it is overwritten by another program. A copy of the variable part is activated for each task when it is initiated (see Figure 15-9). A queueing mechanism keeps track of the user's identification, the location of the variable part, program status word, and register contents for the task. This information is swapped into (or out of) the active area as the user becomes activated (or interrupted). Only one task is active at a time, but several tasks might be in various stages of task completion. Only the property of reentrancy allows true real-time processing and support for multiple concurrent users. Both serial reusability and recursiveness are required to achieve reentrancy in programs.

To summarize, programming languages differ in the extent to which they support alternatives for defining data types, input/output processing, mathematical, relational, logical, bit, control, array, subprogram, and memory processing. The less extensive the language constructs supported, the simpler the language, but the more restricted the domain of problems to which it is amenable. The more extensive the language constructs supported, the more complex the language, and the more extensive the domain of problems to which it is appropriate.

NONTECHNICAL LANGUAGE CHARACTERISTICS

Nontechnical characteristics are at least as important as technical characteristics when selecting a language. The nontechnical characteristics evaluated here are uniformity, ambiguity, compactness, locality, linearity, ease of design to code translation, compiler efficiency, and portability. The availability of CASE tools, availability of code generators, and availability of testing aids also add to a language's attractiveness, and are discussed in a later section.

Uniformity is the use of consistent notation throughout the language. An example of nonuniformity in Focus is the use of single quotes for customized report column titles and the use of double quotes for customized report page titles. This type of inconsistency hinders the learning of the language and almost guarantees that novices and infrequent users will make mistakes.

Ambiguity of a language refers to the extent to which humans and compilers will differ in their interpretation of a language statement. Ideally, humans' thinking should be identical to compiler interpretation, *and* that compiler interpretation should be intuitive to humans. Unfortunately, ambiguity may be inherent to some problems, such as artificial intelligence applications which reason through a process. As new rules and inferences are added to an AI application, interpretation of existing data and rules might also change, thus introducing ambiguity into a previously unambiguous application.

Compactness of a language is its brevity. The presence of structured program constructs, keywords and abbreviations, data defaults, and built-in functions all simplify learning and programming. Contrast SQL or Focus, both fourth-generation languages, with COBOL, a third-generation language. A report that takes three to five lines in 4GL procedure code requires 50–150 lines of COBOL code (see Figure 15-10). That learning time is considerably shorter for Focus than COBOL, partly due to the compactness of the language.

In turn, compactness implies **locality** in providing natural "chunks" of code that facilitate learning, mental visualization of problem parts, and simulation of solutions. Locality is provided through block, case, or other similar chunking mechanisms in languages. Chunks might be implemented via a performed section of code in COBOL, a case construct in Focus, or an object definition in Smalltalk. In all three of these examples, a user's attention is focused only on the chunk of the code present. By being able to ignore other parts of the code, learning of the chunk is simplified.

Linearity refers to the extent to which code is read sequentially. The more linear a language, the easier it is to mentally "chunk" and understand the code. Linearity facilitates understanding and maintainability. In Figure 15-10, the COBOL code chunks in paragraphs and performed sections; these

```
4GL—Focus
        TABLE FILE SALES
        HEADING CENTER 'SAMPLE SALES REPORT'
        SUM SALES
                BY REGION
                ACROSS MONTH
                BY YEAR
        ON YEAR SUMMARIZE
        ON YEAR PAGE-BREAK
        END

3GL—COBOL
...
        WORKING-STORAGE SECTION.
        01 CONTROL-TOTALS.
                05      LINE-COUNT              PIC 99          VALUE 55.
                05      END-OF-FILE             PIC 9           VALUE ZERO.
                        88      EOF                                     VALUE 1.
                05      CURRENT-REGION PIC 99 VALUE ZERO.
                05      SUM-SALES.
                        10      JAN-SUM         PIC 9(5)        VALUE ZEROS.
                        10      FEB-SUM         PIC 9(5)        VALUE ZEROS.
                        10      MAR-SUM         PIC 9(5)        VALUE ZEROS.
                        10      APR-SUM         PIC 9(5)        VALUE ZEROS.
                        10      MAY-SUM         PIC 9(5)        VALUE ZEROS.
                        10      JUN-SUM         PIC 9(5)        VALUE ZEROS.
                        10      JUL-SUM         PIC 9(5)        VALUE ZEROS.
                        10      AUG-SUM         PIC 9(5)        VALUE ZEROS.
                        10      SEP-SUM         PIC 9(5)        VALUE ZEROS.
                        10      OCT-SUM         PIC 9(5)        VALUE ZEROS.
                        10      NOV-SUM         PIC 9(5)        VALUE ZEROS.
                        10      DEC-SUM         PIC 9(5)        VALUE ZEROS.
        01      REPORT-HEADER.
                05      FILLER                  PIC X(48)       VALUE SPACES.
                05      HD1                     PIC X(19)       VALUE
                        'SAMPLE SALES REPORT'.
        01      COL-HEADER1.
                05      FILLER                  PIC X(132)      VALUE
                        'REGION         MONTH'.
        01      COL-HEADER 2.
                05      FILLER                  PIC X(132)      VALUE
                '       JAN     FEB     MAR     APR     MAY
                JUNE    JULY    AUG     SEPT    OCT     NOV     DEC'.
        01      REPORT-DETAIL.
                05      FILLER                  PIC XXX         VALUE SPACES.
                05      REGION                  PIC XX          VALUE SPACES.
                05      FILLER                  PIC X(10)       VALUE SPACES.
                05      SALES                   PIC X(84)       VALUE ZEROS.
```

FIGURE 15-10 4GL versus 3GL Language Compactness

```
        05      SALES-NUMERICS REDEFINES SALES.
          10    JAN-SALES              PIC ZZZ,ZZZ    VALUE ZEROS.
          10    FEB-SALES              PIC ZZZ,ZZZ    VALUE ZEROS.
          10    MAR-SALES              PIC ZZZ,ZZZ    VALUE ZEROS.
          10    APR-SALES              PIC ZZZ,ZZZ    VALUE ZEROS.
          10    MAY-SALES              PIC ZZZ,ZZZ    VALUE ZEROS.
          10    JUN-SALES              PIC ZZZ,ZZZ    VALUE ZEROS.
          10    JUL-SALES              PIC ZZZ,ZZZ    VALUE ZEROS.
          10    AUG-SALES              PIC ZZZ,ZZZ    VALUE ZEROS.
          10    SEPT-SALES             PIC ZZZ,ZZZ    VALUE ZEROS.
          10    OCT-SALES              PIC ZZZ,ZZZ    VALUE ZEROS.
          10    NOV-SALES              PIC ZZZ,ZZZ    VALUE ZEROS.
          10    DEC-SALES              PIC ZZZ,ZZZ    VALUE ZEROS.
        ...
PROCEDURE DIVISION.
...

PERFORM SUMMARY-CONTROL THRU PRINT-REPORT-EXIT.
...


SUMMARY-CONTROL.
        IF REGION = CURRENT-REGION
                GO TO PAGE-CONTROL
        ELSE
                MOVE SUM-SALES TO SALES-NUMERICS
                MOVE YEAR TO REGION
                WRITE REPORT-DETAIL AFTER 3.
                ADD 3 TO LINE-COUNT.
PAGE-CONTROL.
        IF LINE-COUNT > 50 OR REGION NOT = CURRENT-REGION
                WRITE REPORT-HEADER AFTER PAGE
                WRITE COL-HEADER1 AFTER 2
                WRITE COL-HEADER2 AFTER 1
                MOVE 4 TO LINE-COUNT.
        MOVE REGION TO CURRENT-REGION.
PRINT-REPORT.
        MOVE CORRESPONDING INPUT-SALES-SUMMARY TO REPORT-DETAIL.
        WRITE REPORT-DETAIL AFTER 1.
        ADD 1 TO LINE-COUNT.
PRINT-REPORT-EXIT.
        EXIT.
```

FIGURE 15-10 4GL versus 3GL Language Compactness (*Continued*)

language features facilitate COBOL program understandability.

The ease with which program specifications are **translated** into code is also important in language selection. In general, more declarative languages, such as SQL, are considered easier to code than more procedural languages such as Fortran. However, PROLOG and other inferential languages, while declarative and simple in developing single rules, are *not* simple when trying to determine whether the rules aggregate to the proper knowledge structures.

Compiler efficiency is the extent to which a compiled language generates efficient assembler code. Compiler efficiency varies by vendor and by language. Compiled code efficiency is important especially when programming for small computer systems or for embedded applications that interact with other system components as part of a larger system.

Along with efficiency of executable code, portability of code is important. **Portability** is the ability to transplant the code without change to a different operating platform that might include hardware, different operating system, or different software environment. A hardware platform may be a single-user personal computer, a workstation, or a mainframe. Each of these might run the same operating system, for example Unix, or might use a different operating system. The more code that must be changed to accommodate a specific hardware or operating environment, the less portable the language. As global and distributed applications become more prevalent, the need for language portability will increase. Ideally, programs should be able to be developed anywhere for execution on any hardware or operating system platform.

In summary, when technical characteristics do not distinguish languages for application use, nontechnical characteristics of languages become important to their selection. The nontechnical characteristics evaluated here include uniformity, ambiguity, compactness, locality, linearity, ease of code development, compiler efficiency, portability, and availability of automated development tools. In the next section, we discuss ten popular programming languages and the extent to which they contain the language constructs above. Then we discuss application characteristics and how they map to the languages.

COMPARISON OF LANGUAGES

Ten languages are evaluated in this section to highlight the differences across paradigms and language generations for all of the characteristics defined above. The ten languages selected were chosen because of their current and expected future popularity either in academic circles (e.g., Pascal) or in industry. The languages include SQL, COBOL, Fortran, BASIC, Focus, C, Pascal, PROLOG, Ada, and Smalltalk. Each language is discussed briefly below to highlight the characteristics that make it popular and unique. Table 15-1 summarizes the 10 languages on all of the characteristics described above.

SQL

As the American National Standards Institute's standard for database query language, SQL has enjoyed a successful life. SQL pervades any database course taught in North America and is a query language front-end to virtually every database package on the market regardless of machine size, number of users supported, or complexity of the database. SQL's virtues are mostly nontechnical: ease of learning, compactness, uniformity, locality, linearity, portability, and availability of automated tools (see Table 15-1). The simplicity of the language is evident in the small number of hours of learning time it takes novices to begin using the language. A novice might begin writing queries in literally minutes. Proficiency, of course, takes longer, but time to become proficient is shorter than most database languages.

Many CASE environments that support analysis and design also support logical database design through the process of normalization. Those products also generate SQL database definitions as the logical DB design output. Many of the same

(Text continues on page 656)

TABLE 15-1 Comparison of Languages

	SQL	Focus	BASIC	COBOL	Fortran
Data Types					
Real	Yes	Yes	Yes	Yes	Yes
Integer	Yes	Yes	Yes	Yes	Yes
Character	Yes	Yes	Yes	Yes	Yes
String	No	No	No	Yes	No
Boolean	No	No	No	No	No
Date	No	Yes	No	No	No
User-Defined	No	No	No	No	No
Pointer	No	No	No	No	No
Bit Identification	No	No	No	No	No
String-Mask	No	No	No	No	No
Data Type Checking					
Typeless			X	X	
Automatic type coercion	X				
Mixed mode		X			X
Pseudostrong					
Strong					
Operator Precedence	0^ */±	0^ */±	0^ */±	0^ */±	0^ */±
Binary and Unary Operators	Yes	Yes	Yes	Yes	Yes
Arithmetic +,–,*,/	Yes	Yes	Yes	Yes	Yes
Relational <,=,>,≤,≥	Yes	Yes	Yes	Yes	Yes
Logical and,or,not	Yes	Yes	Yes	Yes	Yes
Bit	No	No	No	No	No
Type Conversion	No	Yes, Limited	No	Yes, Limited and Inconsistent	Yes, Limited and Inconsistent
Control					
Loops	No	No	FOR . . . NEXT	PERFORM . . . UNTIL	FOR . . . CONTINUE
Exits	No	EXIT, GOTO	EXIT, GOTO	EXIT	EXIT, GOTO
Conditional Statements	WHERE	IF . . . ELSE	IF . . .	IF . . .THEN . . . ELSE	IF . . .
Case Statements	No	Yes (not in query language)	No	COBOL 88 only	No
Arrays					
Linear Arrays	No	No	Yes	Yes	Yes
Multiple Dimensions	No	No	Up to 2	Up to 3	Up to 3

(Table continues on next page)

TABLE 15-1 Comparison of Languages (*Continued*)

	SQL	Focus	BASIC	COBOL	Fortran
Input/Output					
I/O of Records	No	No	Yes	Yes	Yes
I/O of Arrays	No	No	No	No	Yes
Implicit I/O	Yes	Yes	No	No	No
Format Control	Automatic or Programmed	Automatic or Programmed	Programmed only	Programmed only	Programmed only
Data-directed I/O	No	No	No	No	Yes
Subprograms					
Subroutines	Nested	Yes	Yes	Yes	Yes
Functions	Limited	Yes	Limited	Limited	Limited
Local/Global Storage	No	Yes	Limited	Programmed only	Yes
Static/Dynamic Storage	No	No	No	No	No
Entry Points	No	Yes	Yes	Yes	One
Pass Parameters	No	Yes	Yes	Yes	Yes
Call by Address	No	No	No	No	No
Call by Value	No	No	No	No	No
Call by Name	No	Yes	Yes	Yes	Yes
Reusability	No	Yes	Yes	Yes	Yes
Reentrancy	No	No	No	No	No
Recursion	No	No	No	No	No
Concurrency	Only when used with DB2	Yes	No	No	No
Exception Handling	No	Limited	Limited	Limited	Limited
Nontechnical					
Uniformity	High	Medium-High	Medium	Medium	Medium
Ambiguity	Low-Medium	Low-Medium	Medium	Medium	Medium
Compactness	High	High	Medium-High	Low	Medium-High
Locality	High	High	Programmed only	Programmed only	Programmed only
Linearity	High	High	Low-Medium	Low-Medium	Low-Medium
Ease of design to code	High	High	Low-Medium	Low-Medium	Low-Medium
Compiler Efficiency	Yes, when used as embedded language; otherwise SQL is interpreted	Medium, Mostly Interpreted	Medium, Mostly Interpreted	Medium-High	Medium-High
Source code portability	High	High	Medium	High	High

TABLE 15-1 Comparison of Languages (*Continued*)

	SQL	Focus	BASIC	COBOL	Fortran
Nontechnical, cont.					
Availability of					
CASE tools	Yes	Yes	No	Yes	No
Code generators	Yes	No	No	Yes	No
Testing aids	Yes	No	Yes	Yes	Yes
Maintainability	High	Medium-High	Low-Medium	Low-High	Low-Medium

	C	Pascal	PROLOG	Ada	Smalltalk
Data Types					
Real	Yes	Yes	Yes	Yes	Yes
Integer	Yes	Yes	Yes	Yes	Yes
Character	Yes	Yes	Yes	Yes	Yes
String	Yes	Yes, Limited	Yes	Yes	Yes
Boolean	No, but can be user defined	Yes	No	Yes	Yes
Date	No	No	No	No	No
User-Defined	Yes	Yes	No	Yes	Yes
Pointer	Yes	No	No	Yes	Yes
Bit Identification	Yes	No	No	Yes	Yes
String-Mask	No	Limited	Yes	No	No
Data Type Checking					
Typeless	X				
Automatic					
Mixed mode			X		
Pseudostrong		X			
Strong			TurboProlog	X	X
Operator Precedence	( ) [ ] –> + – (unary) ++ — ! ~ * & size of (type) * / % + – << >> <= >= != == & ^ && || ?: = op=,	not */ div mod + and – or =<> < <= > >= <in |	( ) + – unary mod div * / + – binary relational operators	** not abs * / mod rem + – unary + – & binary relational logical short-circuit	unary binary keyword
	No exponent operator	No exponent operator	No exponent operator	No exponent operator	
Operators					
Binary and Unary	Yes	Yes	Yes	Yes	Yes

(Table continues on next page)

TABLE 15-1 Comparison of Languages (*Continued*)

	C	Pascal	PROLOG	Ada	Smalltalk
Operators, cont.					
Arithmetic +,–,*,/	Yes, also % for modulus	Yes	Yes	Yes	Yes
Relational <,=,>,≤,≥	Yes	Yes	Yes	Yes	Yes
Logical and,or,not	Yes	Yes	Yes	Yes	Yes
Bit	Yes	No	No	Yes	Yes
Type Conversion	No	No	No	No	No
Loops	DO	WHILE . . . FOR . . . REPEAT . . . END	Simulated via REPEAT . . . WHILE . . . INDEX . . .	BEGIN . . . END WHILE . . . FOR . . . BLOCK LOOP . . . END LOOP	ifTrue ifFalse whileTrue while False
Exits	RETURN	RETURN GOTO	FAIL CUT RETURN	EXIT GOTO	
Conditional Statements	IF . . . ELSE	IF THEN BEGIN . . . END . . . ELSE . . .;	None	IF . . .THEN . . . ELSE ELSEIF CASE	ifTrue ifFalse whileTrue whileFalse
Arrays					
Linear Arrays	Yes	Yes	Only as LIST	Yes	Yes
Higher Dimensional Arrays	No limit to number of dimensions	No limit to number of dimensions, Some dynamic allocation support	No	No limit to number of dimensions, Dynamic allocation support	No
Input/Output					
I/O Statements	Only using defined function	No	TurboProlog, else No	Yes	Yes
I/O of Arrays	Only using defined function	Yes	No	No	No
Implicit I/O	Only using defined function	No	TurboProlog, else No	No	No
Format Control	Only using defined function	Limited	Yes	Yes	Yes
Data-directed I/O	No	No	No	No	No

TABLE 15-1 Comparison of Languages (*Continued*)

	C	Pascal	PROLOG	Ada	Smalltalk
Subprograms					
Subprograms	Yes	Yes	TurboProlog, else No	Yes	Yes
Functions	Yes	Yes		Yes	Yes
Local/Global Storage	Both	Both	Both	Both	Both
Static/Dynamic Storage	Both	No control	Both	Both	Both
Entry Points	One per function	One per routine	One per program	One per routine	One per object
Parameters					
Call by Address	Yes	No	No	Yes	No
Call by Value	No	Yes	No	Yes	No
Call by Name	Yes	Yes	Clause name as subgoal	Yes	Yes
Reusability	Yes	Yes	Yes	Yes	Yes
Recursion	Yes	Yes	Yes	Yes	Yes
Reentrancy	No	Yes	No	Yes	Yes
Concurrency	No, unless C++	Concurrent Pascal only	Depends on version	Yes	Yes
Exception Handling	Yes	No	Yes	Yes	Yes
Nontechnical					
Uniformity	Low-High	Medium-High	Medium-High	Medium-High	Medium-High
Ambiguity	Low-Medium	Low-Medium	Medium-High	Low-Medium	Low-Medium
Compactness	Low-High	Medium-High	Low-High	Low-High	Low-High
Locality	Low-High	Low-High	Low-Medium	Low-High	Low-High
Linearity	Low-High	Low-High	Low-High	Low-High	Low-High
Ease of design to code	Medium-High	Medium-High	Medium	Medium-High	Medium-High
Compiler Efficiency	High	High	Usually interpreted	Medium-High	High
Source code portability	High	Medium-High	Low	Medium-High	Low
Availability of CASE tools	No	In academia, yes	No	Yes	Yes
Code generators	No	No	No	No	No
Testing Aids	Yes	Yes	No	Yes	Yes
Maintainability	Low-High	Low-High	Low-High	Medium-High	Medium-High

products also provide code generation of Cobol with embedded SQL providing DB access. Examples of CASE products are ADW™ and IEF™. These products have their own code generators and can interface to code generation software.

In terms of technical capabilities, SQL is limited. It is assumed that complex programming is done in some other language with SQL embedded as described above. SQL can define and modify databases, perform simple mathematical processing on fields for reporting, and generate default or customized reports.

Focus

As a fourth-generation language, **Focus** consists of a database engine with its own query language, SQL compatibility, a full-screen processor, and language subsets for graphical, statistical, file maintenance, and intelligent processing. Focus DB supports relational, hierarchic, and network files as well as providing an interface to many popular mainframe DBMSs, such as IMS, IDMS, Adabas, Model 204, and so on.

Like SQL, Focus' main strengths lie in the nontechnical characteristics of the language: compactness, locality, linearity, ease of code translation, portability, and availability of CASE tools for documenting analysis and design (see Table 15-1). Occasionally, Focus can be ambiguous in interpreting handling of data across a hierarchy or in multiple joined files.

Focus is a full-function database language. This means that files can be defined, maintained, validated, modified by transaction processing, and queried all in the same environment and the same language regardless of the hardware/software platform. This high level of portability and full-function nature of the processing make Focus a popular 4GL for rapid application development and user query processing.

A reentrant version of Focus is available to support multiuser processing. Application code in Focus is not reentrant. A compiler is available for file modify routines; otherwise, Focus is interpreted. Focus is a language of defaults that does not support user-defined or user-managed resources.

BASIC

BASIC is short for *B*eginner's *A*ll-purpose *S*ymbolic *I*nterchange *C*ode. BASIC is present in this evaluation because of the number of applications written in it regardless of whether it were appropriate or not. BASIC is, well, basic. Nothing fancy is supported in this language, but all rudimentary processing is present (see Table 15-1). BASIC is fairly easy to learn and write, with reasonable levels of uniformity, compactness, and good automated testing aids. The remaining characteristics vary considerably from one version of BASIC to another. In particular, its portability is low-medium since the I/O commands usually must change to suit a particular environment.

BASIC does standard programming operations, supporting a limited, but standard number of data types, with no type checking. There are language constructs for loop, condition, and array processing. Files can be read and written.

BASIC is popular because a whole generation of college graduates was subjected to it as the basis for learning programming. Provided an application does not require any nonstandard processing, BASIC can perform adequately.

COBOL

COBOL stands for *CO*mmon *B*usiness *O*riented *L*anguage. It is the most frequently used language in computer history and continues to maintain that status even though its demise is regularly reported as imminent. COBOL can be likened to a bus. Buses are uncomfortable, take longer than most other modes of transportation, but are suited to many types of trips. Similarly, COBOL is uncomfortable to code, it takes a long time to develop code, but it is suited to many business problems. As an all-purpose language, COBOL does most everything, and it is written in a language that is close to English.

COBOL input/output processing is consistently superior in efficiency and range of data structures supported (see Table 15-1). COBOL is not good for

real-time applications and cannot be used to code reentrant or recursive structures. It is teamed with multiuser software, such as CICS for telecommunication interface processing or IMS DB/DC for telecommunication interface and database manipulation, to build effective interactive, multiuser applications.

In the nontechnical areas, COBOL rates high on availability of CASE tools, code generators, and testing aids. As the most frequently used language, it was first on the list of languages for which automated support was developed. It is a highly portable language and is supported by many efficient compilers. In the other nontechnical areas, COBOL rates less desirable than SQL and Focus, but is comparable to or better than other procedural languages.

Fortran

Shorthand for *FOR*mula *TRAN*slation, **Fortran** gained popularity as a *number-cruncher* language in the 1960s and has maintained a dwindling, but steady, popularity ever since. Fortran's weakness is in the data and file structures it supports (see Table 15-1). It does not interface to DBMS software and is limited to sequential, indexed, and direct files. Also, input/output processing of most Fortran compilers is slow, character operations are awkward and not recommended, and data format control is more limited than other languages.

Fortran's strength is in the efficiency of algorithms generated to perform numeric processing. Fortran's compilers usually are accompanied by a subprogram library that includes many frequently used algorithms for sort, statistical, and mathematical processing. Subroutine and subprogram processing is facilitated through easily defined and accessed global and local variables. The mixed mode data typing in Fortran is an important language feature because numeric processing will have different results depending on the definitions of the fields being processed.

Reusable programs can be developed using Fortran, but no one would use Fortran to develop a complete on-line, interactive system. Rather, Fortran routines for numeric processing might be embedded in a system developed in some other language.

C

C is a *high*-level language developed to perform *low*-level processing.[1] Its generality and lack of constraints coupled with autonomy of data structure definition and a rich set of operators make it an effective language for many tasks, including interactive, reusable, and recursive applications (see Table 15-1). A C program is a series of functions that are invoked by embedding their names in code. Transfer of control is automatic as is return processing. System operators, called *escape sequences*, are embedded in the program and recognized by a preceding backslash '\'.

C is a concise, cryptic language that can be efficient in the hands of an experienced, skilled programmer and can be a mess in the hands of a novice or poor programmer. "The language imposes virtually no rules regarding design or structure of programs and enforces nothing at all. This is not a dummy-proof programming language, and it certainly is not for beginners" [Friedman, 1991, p. 398]. As such, the nontechnical aspects of the language all range from low to high because the rating depends on the skill of the programmer. For expert programmers who understand how to build reusable modules, C language provides the capabilities to build reusable libraries with applications built from them.

Pascal

Pascal is a language designed to be unambiguous for teaching students of computer science.[2] Programs in Pascal are free-format, but the language contains natural structuring syntax that can be indented to make the language easily readable.

ConcurrentPascal provides for real-time control over processing. Other versions of Pascal support development of reusable and recursive programs and

1 C was developed at Bell Labs by Kernighan & Ritchie, 1978.

2 For instance, Cooper & Clancy, 1985, is a frequently used Pascal text.

subprograms (see Table 15-1). However, standard Pascal cannot use subroutine libraries since it assumes all program modules are *instream*, that is, embedded within the code of a single program. There is little control over interrupt processing in the language, so abends cannot be intercepted and redirected. I/O processing is more limited than some languages in not supporting random access files and in very limited string processing.

Pascal is similar to C on the nontechnical characteristics in that the readability, ambiguity, locality, and so forth of the language are dependent on the author using indentation and separation of statements to ensure these characteristics. But, unlike C, the language constructs of Pascal support readability once the indentation is done. Pascal requires less technical knowledge of hardware or operating systems to be efficient.

Because Pascal was developed as a teaching tool, automated programming support environments are available at least in academic settings.[3] These environments require the student to enter the construct desired; the software then displays a template of options for which the student fills in the blanks of the selected subconstructs. There are also many automated testing aids such as visual execution environments available to support Pascal program testing.

PROLOG

PROLOG is short for *PRO*gramming in *LOG*ic. PROLOG is the only strictly artificial intelligence language included in this group. PROLOG was developed at the University of Marseilles in the early 1970s with the most common version in the United States that of David H. D. Warren. PROLOG is a goal-oriented, declarative language with constructs for facts and rules. **PROLOG facts** are pieces of concrete, factual information. A fact might be: "A part of a widget is a *wid*." Another fact might be: "A *wid* weighs 1.25 pounds." **PROLOG rules** define how facts are assembled to make information. An example of a rule might be: "If a widget is overweight, check the weight and tolerance of each component."

PROLOG goals are data that match some selection criteria, for example, the probable cause of a manufacturing problem specified in the query: What could cause finished widgets to be 3.2 pounds overweight? **Subgoals**, which would be subprograms in the terminology of the other languages, are determined from the goal. In the example above, widget components, their weight, weight allowances, and how each is used in widget manufacturing might all be subgoal information to be determined to answer the query. Goals are satisfied/answered by satisfying all subgoals. When a subgoal fails, an alternative for arriving at similar information is found via logical backtracking through the rules. The subgoal might remain unsatisfied, leading to a low level of confidence in the deduced answer.

Although the constructs for PROLOG are similar in many ways to those of declarative, procedural, and object languages, there are many significant differences in both data and program processes (see Table 15-1). Data are facts that are normally stored *in* the program rather than as separate files. This is a limitation in using PROLOG for general purpose business processing.

Program control is maintained through the ordering of clauses for execution and through the use of verbs like *fail*, which initiates backtracking by failing a subgoal, or *cut*, which prevents any more backtracking when a subgoal is fulfilled. Subprograms are simulated via *call/return* processing to clauses. Iteration is performed via recursive processing of rules.

How one rates PROLOG on the nontechnical aspects of the language depends on the size of the problem being automated. For small problems, the language can be compact, local, and linear. For large problems, the language can be highly ambiguous, noncompact, difficult to follow in a linear manner, and without local references to facilitate understanding. Ironically, PROLOG is viewed as a good language for novices with little exposure to procedural

3 Thomas Reps, MIT, developed a Pascal programming environment for Cornell as part of his dissertation [Reps, 1984].

languages. It is easy to learn if one can think in the goal-oriented manner of the language.

Smalltalk

Smalltalk was developed as both operating environment and language during the 1970s at the Xerox Palo Alto Research Center by the Learning Research Group. It is an object-oriented language that treats everything as an object, even for instance, integers. Smalltalk is highly customizable and can, therefore, be used to design efficient applications.

Many important object-oriented concepts are embodied in the language, including abstraction, encapsulation, and some class processing (see Chapters 11 and 12). Abstraction is the definition of identifying characteristics of an object. Encapsulation is the term used to describe the packaging of data and allowable processing on that data together. Objects communicate with each other only by message passing. An individual object is an instance of a class. Classes describe objects that share common data and processes but that also may have data and processes that differ. For instance, the class employee might have subclasses manager, professional, and clerk. All subclasses are also employees and share that data and processing as well as their own. In addition, an individual might be a member of professional and manager classes at the same time.

Smalltalk is a full-function, unconstrained programming language that can literally be used to do anything (see Table 15-1). The major weakness of Smalltalk is that it does not specifically support **persistent objects**, also known as files. But if the file is an object, then it, too, can be processed in Smalltalk.

The strength of Smalltalk is in its use for event-driven processing as in process control, heating system monitoring, or just-in-time notification of manufacturing needs. These types of applications use nonpersistent messages from the external environment to drive the processing done by the application; these applications do not necessarily need files for processing. Similarly, message processing support in Smalltalk assumes point/pick devices, such as a mouse, for interactive, nonpersistent communication with the application user. The only major caveat on Smalltalk use is that object orientation, and therefore object-oriented programming, requires a different kind of thinking than procedural language programming such as COBOL.

Ada

Ada, the official language of the U.S. Department of Defense, with a user population in the hundreds of thousands, has had more thought about its implementation than any other language. Ada was named after Ada, Countess of Lovelace, who originated the idea for stored programs to drive the use of computing devices.

Ada's design by committee has not resulted in a perfect language, but in one that is better than most. Current versions of Ada are object based rather than object oriented. In object-based applications, programs are cooperative collections of objects, each of which represents an instance of some object type. All object types are members of a hierarchy of types which are linked through processing rather than through inheritance relationships. Classes, rather than types, are not formally recognized; there are no persistent objects such as files, and inheritance is not supported (see Table 15-1).

Ada files, as in Smalltalk, are defined as a type within the constructs of the language and all processing is on the type. Also, there is no real message processing in Ada, at least as of 1992. Rather, the system is fooled through function calls and parameter passing to simulate message processing. Like Smalltalk, Ada's strength is its ability to support event-driven processing, like missile guidance in embedded defense-related systems.

Future versions of Ada are expected to adapt multiclass inheritance structures and processing, dynamic binding of objects, real message processing, and persistent objects that provide a variety of data structures. With these extensions, Ada is suitable for virtually any application. The same warning about the difference in object-oriented thinking expressed about Smalltalk is also appropriate here: Object-oriented design and program development is different in kind than procedural development of applications via languages such as COBOL.

PROGRAMMING LANGUAGE EVALUATION

Two ways of matching program languages are considered in this section. The first is to match the programming language to the application type (from Chapter 1). The second is to match the language to the methodology used for developing the application (from Chapters 7–13).

Language Matched to Application Type

Few heuristics have been available to guide programmers in matching a programming language to application type. The lack of heuristics is due mostly to the newness of most languages and their restricted use in academia (e.g., Pascal and PROLOG). Part of the reason for a lack of heuristics is also because most businesses have developed only transaction processing applications until the late 1980s; one or two languages were sufficient for most computing in the organization. With the development of query languages, AI applications and object orientation, more languages have proliferated and heuristics have slowly developed. Keep in mind that as experience with emerging paradigms, such as object orientation and intelligent applications grow, the heuristics will be refined and changed from those presented here. For each application type discussed in Chapter 1, the normally relevant characteristics and language choices are discussed below and summarized in Table 15-2.

Transaction processing applications are divided for classification into batch, on-line, and real-time as the predominant form of processing. For batch applications, COBOL and Focus are best suited (see Table 15-2). For on-line applications, all languages except Fortran and PROLOG might be used. Fortran is excepted because of its poor I/O processing; PROLOG is not recommended because data are usually embedded in the code, precluding most TPS processing. Language actually chosen should be based on the transaction volume, with high volume TPS moving away from the SQL and 4GL languages toward compiled, full-function languages. If there is a DBMS or other special data access software, the choices narrow to Focus or COBOL depending on the specific DBMS.

Some business systems are specialized because they are real-time and have stringent response time requirements in addition to being critical to at least one organization. Examples of real-time TPS include airline reservations, securities transaction processing, manufacturing process control, robotics control, or analog I/O applications. For such systems, the language recommendations are restricted to C, Pascal, Ada, and Smalltalk (see Table 15-2). Any of these languages can be used to develop reentrant, multiuser, real-time applications, although attention to a specific dialect (or vendor version) is required to choose a reentrant version of the language. An alternative is to develop such applications using assembler language as the reentrant base with one or more of the application languages used for individual modules.

Query processing is restricted to SQL, Focus, and PROLOG (see Table 15-2). SQL, Focus, and PROLOG support declarative statements of *what* is desired without having to anticipate the outcome in advance. As such, they are the only three languages of these ten to support query processing. PROLOG has the added feature that it can explain its reasoning process and provide probabilities of accuracy for its data. Both SQL and Focus assume they are working on complete information and there is only one answer to a given query. PROLOG can be programmed to develop confidence estimates in answers as well as to develop all possible answers to a query.

Data analysis applications are those in which statistical routines, trend analysis, or other mathematical manipulation of data is desired. Data analysis applications can be programmed or can use packages combined with programs. For such applications, Focus, Fortran, Pascal, PROLOG, Ada, and Smalltalk might be used (see Table 15-2). COBOL is conspicuously absent from this list because it is not as adept at data analysis as other languages. Focus provides statistical modeling, financial modeling, graphical processing, and query processing all

TABLE 15-2 Application Type Matched to Language

Application Type	SQL	Focus	BASIC	COBOL	Fortran	C	Pascal	PROLOG	Ada	Small-talk
TPS—Batch	X	X	X	X						
TPS—On-Line	X	X	X	X		X	X		X	X
TPS—Real-Time						X			X	X
Query	X	X								
DSS/Data Analysis	X	X			X	X	X	X	X	X
AI/Expert Systems								X		
EIS		X				X			X	X

within its one language. As such, it is the most full-function data analysis tool in this group. The other languages have the individual tools for a programmer to build a data analysis application, but the assumption is that some processing would be done by general purpose modeling languages (e.g., Statistical Analysis System—SAS.[4] If complex simultaneous equations are required, Focus is not the appropriate language. Then, choices are restricted to Fortran, Ada, or Smalltalk. Fortran does not actually provide simultaneous equation solutions, but it can be 'fooled' into performing as if it does. The other languages are better choices for simultaneous equation processing. Some dialects of C (i.e., Concurrent C) and Pascal (i.e., Object Pascal) might also be used for simultaneous equations.

ESS or DSS applications may have changing requirements that are not well understood due to the unstructured nature of the problem domain. For such applications, C, Pascal, PROLOG, Ada, or Smalltalk might be used (see Table 15-2). One or more of these languages might be combined with purchased software packages to provide all the functions of such applications.

GDSS applications almost always use packages to support group decision processes, but might use C, Pascal, PROLOG, Ada, or Smalltalk for part of the processing, depending on the environment (see Table 15-2).

Finally, artificial intelligence applications, specifically expert systems, might use PROLOG (see Table 15-2). Only PROLOG supports inference through logic programming. None of the other languages is appropriate to AI applications.

Language Matched to Methodology

The experience with methodologies is similar to that of languages in that few heuristics are known to guide methodology selection. Rather, at the present time, a company tends to adopt and learn one methodology and it is used for all applications, whether appropriate or not. The position taken here is that the methodology and language should match the application type. In this section, the ten

4 SAS is a registered trademark of the SAS Corporation, Cary, NC.

TABLE 15-3 Application Type Matched to Methodology

Methodology	SQL	Focus	BASIC	COBOL	Fortran	C	Pascal	PROLOG	Ada	Small-talk
Process	X	X	X	X	X	X	X		X	
Data	X	X		X		X			X	
Object						C++		X	X	X

languages are matched to methodologies which were discussed in Chapters 7–13.

Process methodologies which prevailed in business until the mid-1980s are most successfully used with SQL, Focus, BASIC, COBOL, Fortran, C, Pascal, and Ada (see Table 15-3). The other languages require too much attention to data or program design to lead to optimal language use with process methods. Also, the use of process methods should not be used with data-intensive applications because of the lack of specific attention given to data with such methods. The C-language is here because it is process oriented; if C++ were the language, it should only be used with object-oriented (OO) methods. Similarly, Ada *can* be used here but it is best used with OO methods.

Data methodologies balance the design of processes and data evenly and are useful with SQL, Focus, COBOL, C, and Ada applications (see Table 15-3). For interactive applications in which the programmer needs only limited control, SQL and Focus are useful. For more complex applications, COBOL, with a DBMS and telecommunications monitor, provides interactive processing capabilities. The process discussion on C and Ada applies here; both languages *can* be used with data methods but are recommended with OO methods.

Finally, for object methodologies, C++, PROLOG, Ada, and Smalltalk are most likely to lead to successful implementations (see Table 15-3). The languages omitted in the object category do not easily support one or more of the object tenets of polymorphism, message passing, class inheritance, or encapsulation.

AUTOMATED SUPPORT FOR PROGRAM DEVELOPMENT

In the age of the smart machine, the availability of developmental aids, CASE environments, code generators, and testing aids such as debuggers, incremental compilers, windowed execution environments, and so on, all speed development of working code. Any language which has such automated development aids is assumed to lead to increased programmer productivity over languages that do not have such aids (see Table 15-4).

CASE tools frequently have built-in code generators or have interfaces to other vendor's code generators, allowing you to mix and match the development environment and the language generated.

The automated support tools include code generation tools, incremental compilers, and program generation environments. All of these are loosely called *Lower CASE* or *Back-end CASE* tools.

SUMMARY

In this chapter, a number of distinguishing characteristics of languages were defined. These included: data type definitions supported, data type checking, operators supported, type of user processing supported, and processing for loops, conditional statements, arrays, I/O, and subprograms. In addition, nontechnical characteristics included uniformity,

TABLE 15-4 Automated Support Tools for Code Generation

Product	Company	Technique
ADW—Construction Workbench	Knowledgeware, Inc. Atlanta, CA	Builds Pseudocode for modulesthat can be used to Generate Code for MsDOS, MVS
C Development Environment, OOSD/C++	Environments (IDE) San Francisco, CA	Object-oriented C++ code development environment
Developer Assistant for Information Systems (DAISys), Secure user Programming by Refinement/DAISys	S/Cubed Inc. Stamford, CT	Generates COBOL for IBM mainframe, AS/400, OS/2 Generates C Code for MSDOS, OS/2
IEW	Texas Instruments Dallas, TX	Generates COBOL with Embedded SQL Generates C Code for MVS, MsDOS, OS/2 Interfaces to Telon and other Code Generators
NeXTStep 3.0	NeXT Computer Redwood City, CA	Object Oriented DB development environment
ObjectMaker	Mark V Systems	Generates C or C++ Code for MsDOS, VMS, Unix, AIX
Software Through Pictures	Integrated Development	Generates C or C++ Code for Unix, AIX
System Architect	Popkin Software & Systems Inc. New York, NY	Generates C Code for MsDOS, OS/2
Teamwork, Ensemble	Cadre Technologies Providence, RI	Generates C or C++ Code for for Unix, OS/2, AIX
Visible Analyst Workbench	Visible Systems Corp. Newton, MA	Generates C Code for MsDOS

ambiguity, compactness, locality, linearity, ease of code translation, portability, compiler efficiency, and availability of CASE, code generation, and testing tools. Each of ten languages were described according to the characteristics. Then the languages were defined as appropriate for supporting different application requirements and were discussed in terms of their support for development of transaction, query, data analysis, DSS, ESS, and ES applications.

REFERENCES

Ageloff, Roy, and Richard Mojena, *Applied Fortran 77 Featuring Structured Programming*. Belmont, CA: Wadsworth Publishing, 1981.

Alcock, B., *Illustrating Pascal*. New York: Cambridge University Press, 1987.

Barnes, J. G. P., *Programming in Ada*, 3rd ed., Reading, MA: Addison Wesley, 1989.

Barnett, Eugene H., *Programming Time-Shared Computers in Basic*. New York: John Wiley, 1972.

Bjorner, D., and C. B., Jones, *The Vienna Development Method: The Meta-Language*. New York: Springer-Verlag, 1978.

Booch, Grady, *Software Engineering with Ada*, 2nd ed., Menlo Park, CA: The Benjamin/Cummings Publishing Co., Inc., 1987.

Bordillo, Donald A., *Programmer's COBOL Reference Manual*. Englewood Cliffs, NJ: Prentice-Hall, 1978.

Clocksin, William, "A prolog primer," *Byte*, August, 1987, pp. 146–158.

Cooper, Doug, and Michael Clancy, *Oh! Pascal!*, 2nd ed., New York: W. W. Norton & Company, Inc., 1985.

Date, C. J., and Colin While, *A Guide to DB2*, 2nd ed., Reading, MA: Addison-Wesley, 1988.

Friedman, Linda Weiser, *Comparative Programming Languages: Generalizing the Programming Function*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

Gear, C. W., *Programming and Languages*. Chicago: Science Research Associates, 1987.

Goldberg, Adele, *Smalltalk-80: The Interactive Programming Environment*. Reading, MA: Addison-Wesley Publishing Co., 1984.

Higman, B. A., *Comparative Study of Programming Languages*. New York: American Elsevier, 1967.

Information Builders, Inc., *Focus Users Manual*. New York: IBI, Inc., 1984.

Kernighan, Brian W., and Dennis M. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1978.

Martin, J., *Fourth Generation Languages*, Vols. 1–2. Englewood Cliffs, NJ: Prentice-Hall, 1985.

S. Medema, C. H., P. Medema, and M. Boasson, *The Programming Languages: Pascal, Modula, Chill, and Ada*. Englewood Cliffs, NJ: Prentice-Hall, 1983.

Nagrin, Paul, and Henry Ledgard, *Basic with Style: Programming Proverbs*. Rochelle Park, NJ: Hayden Books, Inc., 1978.

Philippakis, A. S., and Leonard J. Kazmier, *Advanced COBOL Programming*, 2nd ed., New York: McGraw-Hill, 1983.

Reps, Thomas W., *Generating Language-Based Environments*. Boston, MA: MIT Press, 1984.

Stroustrup, Bjorn, "Data abstraction in C," *AT&T Bell Labs Technical Journal*, Vol. 63, October 8, 1984, pp. 1701–1732.

Warren, David, H. D., "The SRI model for Or-parallel execution of PROLOG—Abstract design and implementation issues," *Proceeding, 1987 International Symposium on Logic Programming*, August 31–September 4, San Francisco, CA, IEEE, pp. 92–102.

KEY TERMS

Ada
ambiguity
array
array-oriented I/O
automatic type coercion
BASIC
bit data type
Boolean
C
case statement
character string
COBOL
compactness
compiler efficiency
conditional statement
control language constructs
data type
data type checking
date data type
dynamic memory management
ease of code translation
exception handling
exit
Focus
Fortran
global data
input/output (I/O)
integer
language constructs
linearity
list-directed I/O
local data
locality
logical data type
loop
memory management
mixed mode type checking
modularization
object
operator precedence
Pascal
persistent object
physical I/O
pointer
portability
programming
PROLOG
PROLOG facts
PROLOG goals
PROLOG rules
PROLOG subgoals
pseudostrong type checking
reentrant
real number
record-oriented I/O
recursive
reusability
set-oriented I/O
Smalltalk
SQL
static memory management
strong type checking
table
typeless checking
uniformity
user-defined data type

EXERCISES

1. For any (or all) of the cases in the Appendix, define the application concept as batch, on-line, real-time, or a mix of these. For the applications you choose, select an implementation language and develop the reasons why the language you recommend is best. What specific features and characteristics of the language make it your preferred choice?

STUDY QUESTIONS

1. Define the following terms:
   Boolean data type
   dynamic memory management
   local data
   modularization
   operator precedence
   pointer
   reentrant
   set-oriented I/O
   static memory management
   type checking
   user-defined data type
2. Why should we concentrate on language selection rather than on programming?
3. In your opinion, is programming going to disappear as an activity? Justify your response.
4. What is a data type and why is it important in language selection?
5. When is strong type checking important?
6. Why do you think type checking is absent from a language like COBOL?
7. Why is type checking important in object-oriented programs?
8. Define three logic-related language constructs and discuss their differences.
9. What is operator precedence? Why, as a programmer, must you be aware of operator precedence in a language?
10. In an ideal program, how many exits should a module contain? Why?
11. Define the three types of arrays that are commonly supported in languages.
12. For SQL, COBOL, Fortran, Ada, C, and Pascal, define the type of I/O orientation as record-oriented, set-oriented, array-oriented, or list-directed. What difference does the I/O orientation make?
13. What are the differences between local and global data? How do they relate to properties of programs such as reusability, reentrancy, and recursion?
14. Contrast static and dynamic memory management.
15. Why is exception handling desirable in a language? Why don't all languages support exception handling?
16. What level of code sophistication is required to support multiple concurrent users? Why?
17. What is the relationship of recursion, reentrancy, and reusability of programs?
18. List three nontechnical language characteristics and describe why they are important in language selection.
19. Define language portability. Is this property of growing or decreasing interest to businesses, and why?
20. What is COBOL's appeal?
21. Why is C a potentially dangerous language?
22. Describe how and why PROLOG differs so much from the other nine languages in this chapter.
23. How does PROLOG handle databases?
24. What are the object-oriented languages? How do they differ from the other languages?
25. Even though SQL and Focus both use implicit I/O, they are different. What is the main difference in the way they treat data? Which language is 'cleaner' in guaranteeing the results of a query?

★ EXTRA-CREDIT QUESTIONS

1. PROLOG is not the only logic-oriented, artificial intelligence programming language. Lisp is also popular. Investigate the differences between the two programming languages using the characteristics discussed in this chapter.
2. Object orientation and artificial intelligence are two characteristics of applications that are of growing interest to businesses. Can a typical COBOL transaction processing application incorporate object and AI tenets? Will COBOL change or will other languages come to be used? Can other languages be 'grafted on' or interfaced to COBOL gracefully? Be sure to document your arguments.

CHAPTER 16

PURCHASING HARDWARE AND SOFTWARE

INTRODUCTION

When PC software companies first created the end-user market in the early 1980s, the number of PCs in companies was about one per every 4,000 people. By 1986, the number of PCs was about one per every 100 people; companies had settled on standard, supported products for spreadsheets, databases, and word processing. In the intervening years, there was a mad scramble for market share during which vendors' claims were sometimes unfounded, the notion of *vaporware* was created, and major evaluations were done by buying companies. For every new market that develops, a similar set of activities takes place. In the 1990s, object-oriented languages, expert systems, imaging systems, multimedia, CASE products, and distributed databases are the new markets that will have developed recognized leaders by the end of the decade. At best, a company selects a product and vendor that will weather the storms of industry growth and emerge a leader. At worst, they purchase several products before settling on one that works for their company.

The purchasing process tries to minimize the guesswork and provide a rational, objective method of selecting hardware, software, or services. The techniques can be used on products of any type. There are two basic processes, one informal and one formal. There is a great deal of overlap in the activities. The major difference is that the formal process is usually conducted in a more open environment, frequently for legal compliance. All governmental contracting for goods and services, for instance, is subject to a formal procurement process that includes the solicitation of proposals from vendors.

In this chapter, we discuss how to evaluate and choose between alternatives for application use. The trade-off between building the item in-house or purchasing it elsewhere is commonly called a **make-buy decision**. This name is not always accurate, however, because you might be comparing development alternatives, for instance, having a consulting company build a customized application versus purchasing a software package. These alternatives all are considered in the make-buy decision process. RFPs can be used for deciding between vendors that have the same package but are selling turnkey products including all hardware and software in an 'environment,' or for hardware only, software only, services only, or some combination of those three.

In this chapter, we first discuss the formal procurement process, describing the steps performed in the purchasing decision process. The informal process is then described and compared with the formal process. Then, the contents of each RFP section are detailed. Next, we discuss the selection process and criteria that are important to it. Finally, automated support tools for RFP management and eval-

uation are presented. The ABC case is woven throughout the discussion, providing examples of the major points.

REQUEST FOR PROPOSAL PROCESS

A **request for proposal**, or **RFP**, is a formal, written request for bids on some product. In our context, an RFP might relate to hardware, firmware, software, or services such as programming or operations management. Also called **RFQ**, for **request for quotation**, an RFP provides formal requirements, ground rules for responses, and, usually, a standard format for the proposal responses. The basic stages of the request for proposal process, which are discussed in the ensuing sections, include the following:

1. Develop and prioritize requirements
2. Develop schedule and cost
3. Develop requests for proposal
4. Receive proposals
5. Evaluate proposals and select alternative

Develop and Prioritize Requirements

The initial step in all software engineering projects, regardless of whether it is going out for bids or not, is to determine the requirements. When proposals are solicited, the requirements define the problem and the features and functions of the solution that will constitute the work of the bidding companies. In general, the requirements provided in an RFP are identical to those developed during analysis. If a requirements specification is available, it should be appended to the RFP and referenced in the document. If no requirements specification has been developed, at a minimum, the topics summarized below should be provided.

1. General instructions
2. Statement of work
3. Technical specifications
4. Management approach
5. Financial requirements
6. Company information requirements
7. Vendor response guidelines
8. Standard contract terms and conditions

The level of detail and specificity of the requirements varies with the context, situation, and company. Some companies spell out every item in excruciating detail, leaving nothing to the vendors' imaginations. The advantage of such detail is that the proposals can be easily compared to the list of requirements to determine compliance with the basic request. Also, the likelihood of misunderstanding of requirements is lower when more detailed descriptions are used. The disadvantage of detailed requirements is that, in information systems work, the complex engineering nature of the work frequently requires creative design that might be stifled or overshadowed by too specific a requirements list. The creative aspects of systems design also provide for cost differentiation that might not otherwise surface. To overcome this problem, when creativity is desired, it can be specifically identified as a selection criteria in the RFP.

There are four types of requirements: technical, managerial, financial, and company. **Technical requirements** address the specific hardware, software, or services to be provided. **Managerial requirements** identify the level of detail at which schedule, staff plans, and staff management should be discussed in the proposal. **Financial requirements** list the type of bid desired and the expected format for the financial portion of the response. **Company requirements** list the type of vendor information to be supplied to assure the client of vendor ability to complete the work successfully. The details of each section are discussed in the RFP contents section.

Develop Schedule and Cost

The schedule and cost developed during an RFP process are neither as detailed nor as refined as if the item costs were developed in-house. If the in-house estimate is being compared to the vendors'

estimates in a make-buy decision, a detailed schedule and cost should be developed. If the RFP is comparing only external purchase options, less detail and precision are required. In this case, the schedule provides an estimated end-date for the item to be used in comparing the proposals. The expected end-date might be omitted and left as a proposal item, or might be listed as either required or desired in the proposal.

Occasionally, a user manager will mandate the desired completion date for a project. In that case, the in-house estimates are developed to determine the realism of the mandated date. If the date is unlikely because it is very different from the estimate, the vendors can be asked in the proposal requirements how they deal with completion date problems and a tight schedule.

The planning process is the same as that followed in Chapter 6, with the level of precision adjusted to fit the situation. Requirements are converted into a task list. Each task's development time is estimated for the most likely outcome. Sophisticated estimates, including optimistic, average, and pessimistic times, may or may not be developed. During the proposal evaluation process, vendor time estimates are compared to the planned completion date.

A similar activity is done for personnel estimates. A rough estimate of the number of people and their skill levels should be developed, based on the tasks and times for each task. During proposal evaluation, the estimated project team skills are matched against the skills of the people to be assigned to the project by each vendor. The closeness of match indicates several things. First, the closer the match, the more confidence you can have that the vendor understands the problem. Second, the closer the match, the more likely the vendor's reasoning is consistent with your reasoning about the project's needs. Third, the less close the match, the more likely the vendor is staffing the project with people who are learning new skills and who, therefore, will not be fully knowledgeable about the technology or application area of your problem. This third case is not necessarily bad, but it does imply that there will be one, or possibly two, key person(s) on whom the success of the project rests. This places you, as the client, in a somewhat more vulnerable position because you must rely totally on the key person(s), ensuring that they remain on the project until it is operational.

Staffing estimates are used to develop personnel costs for the project. If the proposal includes hardware or software, each item should be priced at the best retail prices available. For instance, *MacWorld* and *PC* magazines include tear-out pages of advertising by discount vendors for both hardware and software. Professional data sources, such as *DataPro*™,[1] provide retail prices which can be used as a basis to which proposed costs might be evaluated.

Develop Request for Proposal

The steps in developing the RFP are first, to determine likely vendors; second, select from the likely vendors the few that best meet your requirements; and third, develop and send the proposal to the vendors.

Determine Likely Vendors

Several stages of information gathering precede the actual bidding process. First, potential vendors are identified. Vendor identification can be from a commercial information service, such as *DataPro*™, or from trade magazine advertisements, for instance, from *PC* Magazine, *Computerworld*, or *Network Week*. This process should identify ten or more vendors.

Narrow the Number of Vendors

When potential vendors are identified, they are contacted and requested to send information. Depending on the company and item, this can be an informal telephone call or can be a formal, written **request for information (RFI)**. Documentation on the products requested is reviewed to narrow the number of alternatives to a manageable few, usually between two and five.

The information review frequently identifies a need for more information to differentiate between products. Either requirements are refined or more information is obtained, or both. Another round of

1 DataPro is a trademarked name of DataPro, Inc., Delran, NJ.

information gathering might then take place. At this point, remaining vendors might be called in to present their product(s) and demonstrate how they work. Specific technical questions to provide missing information are asked.

The decisions after this round of information gathering depend on the nature and use of the product being purchased. If the number of users is small and the product is inexpensive (e.g., under $10,000), a selection might be made. The more users and the more expensive the product, the more extensive the evaluation. Other companies that use the product might be solicited for experience with the company and product, and perhaps, are visited for an on site demonstration. In these cases, when the field of vendors is narrowed to between two and five, an RFP is developed and proposals are requested.

Develop and Send the Proposal to Vendors

The RFP can be developed in parallel with vendor identification. There is some risk that doing so, however, will produce a biased requirements set that favors one particular vendor. The best approach, therefore, is to develop the requirements first, then search for vendors. When the vendor list has been narrowed to between two and five, the RFP is finalized, vendors are notified that they will receive the proposals, and the proposals are sent or delivered to each vendor. From this point, the requesting company begins to manage the proposal process.

Manage Proposal Process

The **proposal process** begins with release of an RFP to vendors and continues until the proposals are delivered and the selection process begins. The proposal process might include one or more formal meetings, informal meetings, inquiry sessions, or other methods of information exchange between the vendors and the requesting organization. The more money involved and the more complex the proposed work product, the more process management is needed to ensure equitable treatment of all vendors. Equitable treatment means ensuring that all vendors receive the same information. Firm compliance with due dates and locations for delivery of proposals is maintained. Late or incorrectly delivered proposals are dropped from further consideration, providing equitable treatment of all vendors.

Assume a proposal is being let by the local police department for development of an application that would deploy computer terminals in each police car for interactive look-up of license plates, arrest warrants, and moving violations. The application requires both hardware and software to be developed for 14,000 police cars in a large metropolitan area with over 3,000,000 inhabitants and covering several jurisdictions. Examples might be Washington, D.C., Los Angeles, New York City, Houston, or Chicago. Hardware cost alone is over $2,000,000. The databases each will have millions of entries with issues to be resolved about how and when information is removed from the files. Interfaces to several other applications for license plate information and access to arrest warrants from multiple local and national databases are desired.

The proposed application has several sources of complexity, the least of which is that vendors probably know little about how a police officer spends his day. When New York City let a similar contract for its police force, they had a formal announcement of the proposal to vendors. Vendors were selected and invited to the presentation by mail based on previous contract work or reputation. Nonsolicited vendors were also welcome in response to announcements of the RFP that ran in the local newspaper for several days.

At the formal presentation, each vendor was invited to spend up to four hours traveling with an officer to view the tasks firsthand, for which the application would be built. A specific officer was identified as the liaison for these tours.

In addition, the liaison officer was available for questions at any time until proposals were submitted. If questions were asked by a vendor, the question and response were recorded and a list of all such queries was sent to all vendors attending the proposal announcement meeting. The purpose of providing all queries and responses to all vendors was to ensure that information inadvertently left out of the RFP that might alter the decision process could not be used by one vendor to the detriment of the

others. By giving everyone all responses, every vendor had the same information.

Halfway through the two-month proposal process, another meeting was held for vendors to come ask more questions and to clarify the requirements from the document. That meeting was well attended but contained no real information. When one person was asked why he bothered attending, he replied, "To see what the competition asked."

Each vendor presented his or her proposal on the due date and left the written copy for NYC review. Each vendor, then, heard the other vendors' proposals and had some sense of the differences between them. Ironically, the company with the best solution lost because the company was too small. One shortcoming of the RFP was that it had not identified company size as a selection criterion; if it had, the vendor would not have wasted his time bidding.

Evaluate Proposals and Select Alternative

The sections of the proposal responses are each evaluated separately, then summarized together. The technical evaluation reviews that requirements are met and scores the proposal based on the priority criteria developed during the preparation of the RFP. A **benchmark**, or comparison test, might be used to identify differences between hardware or software packages.

The management approach is evaluated for the type, quality, and nature of staff and vendor company resources proposed for the work. A financial evaluation is developed to show the present value of the proposed amount(s). Other analysis, such as payback period, or average cost per vendor employee, might be developed for comparison purposes. Next, the vendor's prior experience with the firm, similar applications, and business reputation are ranked to evaluate the vendor's capability to do the proposed work. Finally, each section is weighted again for comparative section importance, creating a summary of the ratings and final weighted score for each vendor. Objectively, the vendor with the highest, overall weighted score is selected for the work. Each type evaluation is discussed in the evaluation sections. After selection, a contract is negotiated and work begins.

INFORMAL PROCUREMENT

Most of the same information required for the RFP is required for the informal procurement process. The major difference is in the approach. In the informal process, few, if any, written documents are used for vendor-client communications. Rather, telephone calls, meetings, and document reviews are the major sources of information. The process of selection is similar to that of the RFP process, including trials and benchmarks for acceptance of the item being procured.

Negotiation is verbal and may go back and forth between the principals for several weeks. Vendors signify agreement with the negotiated terms via a memo. A memo proposal summarizes the main points of agreement, then lawyers are called in, as with an RFP, to add the legal terms.

CONTENTS OF RFP

RFP contents include a summary, information on the technical, managerial, company and financial aspects of the bid, a schedule of the process, selection criteria, vendor response requirements, and any standard contract terms (e.g., for EEO or OSHA compliance). Each RFP section is detailed below to identify optional and required information.

Vendor Summary

The Vendor Summary section provides a short, one-page summary of the work to be done (see Table 16-1). General terms and conditions of the proposal process are usually first to allow vendors to quickly decide whether or not they are interested in the engagement. The contents of the general instructions sections should include proposal instructions, location and date for proposal delivery, dates for bidders' conferences, and contacts for status reporting and inquiries.

TABLE 16-1 Detailed RFP Outline

1.0 General instructions

2.0 Statement of work
- 2.1 Description of work to be performed
- 2.2 Project milestones and deliverable products
- 2.3 Criteria for vendor qualification

3.0 Technical specifications
Technical outlines are in Tables 16-4, 16-5, and 16-7 for hardware, network, or operating system, and customer software or package, respectively.

4.0 Management approach
- 4.1 Schedule and staffing
- 4.2 Support requirements of vendor
- 4.3 Reporting
- 4.4 Staff reporting structure and problem management

5.0 Financial requirements

6.0 Company information

7.0 Vendor response guidelines

8.0 Standard contract terms and conditions

Required Information

The requirements list details the requirements of the work as described in the sections on hardware and software. The section can refer to an attached document that might have been developed in-house for functional requirements of the application, hardware, or software. In any case, requirements should be listed and identified as mandatory or optional. A set of prioritized weights for the requirements should also be developed for use in scoring, but weights should not be published in the RFP. There are four general classes of requirements: technical, management, corporate, and financial.

Technical Requirements

GENERAL REQUIREMENTS. The requirements should place the company and problem in a context for the vendors. First, a brief overview of the industry, company, and work domain is appropriate. Then, a summary of the problem being automated is presented. The major complexity, such as geographic dispersion across 16 states, should be identified. Then, the details of work to be provided are described.

DETAILED REQUIREMENTS. The work might include hardware, software, programming services, or other IS services. The criteria for each item should be detailed as much as possible. In general, regardless of the type of procurement, the features and functions of the equipment should be described in sufficient detail to enable the vendor to design a solution. Functional requirements—*what*—the item is expected to do are described in detail. Volume of data, throughput, response times, and growth requirements are identified. The type, contents, timing, and format of interfaces also are provided. A hardware interface might list, for instance, a network interface connection to a fractional T-1 (cable) service for internetwork communication. A software interface might list, for instance, a DBMS interface connection to a SQL server. An application interface might list, for instance, electronic messages to be sent to an Accounts Receivable Application.

For services, the work description varies depending on the work. The two most common service RFPs request proposals for software development and for outsourcing of operations. For software services, the application requirements are the information provided. For outsourcing operations, the business functions included and any existing job descriptions relating to those functions should be provided to the vendors.

Diagrams, tables, and lists should be supplemented by text to provide clarification of incomplete, misleading, or ambiguous diagrams. For instance, a data flow diagram cannot describe timing of processes or process interrelationships that might be important. They also do not include constraints, need for simultaneous processes, and so on. Requirements for these items would be described in text as required.

AUDIT AND APPLICATION CONTROL REQUIREMENTS. Recall from Chapter 10 that audit controls are frequently needed to prove processing. The audit and control section of the RFP identifies the minimum acceptable level of auditability required. If audit controls are in

compliance with laws or other professional guidelines, the requisite laws and guidelines should be referenced.

Vendors' designs might assume human interventions to ensure accurate application processing. A requirement should be developed to surface such assumptions. For instance, controls might include data integrity, data and process access, exception management, and print control of prenumbered documents (e.g., checks). These examples usually require manual interventions supplemented with interactive processing to recover from failures or to fix hardware problems. For instance, a check might jam in a printer after it is printed. Both software and human procedures are required to reprint the check and to account for the damaged check. (See Chapter 10 for types of failures that should be planned.) Vendors should be required to identify and detail all such interventions as part of their proposal.

PERFORMANCE REQUIREMENTS. Performance requirements include manual, hardware, and software performance. For instance, hardware performance might define acceptable limits for downtime, precision for mathematical computation, or cycle time.

CONVERSION REQUIREMENTS. Recall that conversion requirements define the required changes from the current environment to the new automated environment (see Chapter 14 to review this discussion). The RFP typically identifies data for conversion, including the current format, current volume, and growth required. Conversion timing constraints should be identified if any exist. The vendors' designs should describe the target database for the data and a migration path for conversion. The vendors' conversion plans also should estimate conversion impacts on users, computer operations, and project staffing.

TRAINING. Training to be provided as part of the contract should be listed as a required topic of the vendors. Training options can be left open to vendor proposal or be specified as requirements. Training might be provided for users, software maintenance staff, operations staff, or user support staff. The type of training can be one-on-one, programmed, individually self-paced, classroom, computer-based training, or some variation of these. Training information provided might include the type, number of sessions, location, and audience for training. The qualifications of expected trainers should also be requested.

ACCEPTANCE. **Acceptance criteria**, specifying the contents and timing of the acceptance test, should be identified so the vendor knows how work will be judged. Acceptance criteria might include type and amount of test data, length of time for parallel and pilot runs, phased cutover approach and speed desired (e.g., five locations per month for five months), and performance criteria for success (e.g., five consecutive days with all accounts in balance at the end of daily processing).

Hardware and software packages are usually benchmarked to verify that they perform as advertised. A benchmark is a comparison test between two or more configurations. The contents of the test are a suite of application programs that are representative of the expected work load of the production system. A benchmark test provides you the ability to compare throughput performance with the representative work load. In addition to the benchmark which precedes installation, hardware and software packages might also be run through a trial period similar to that described above for acceptance.

Management Approach

SCHEDULE AND STAFFING. Vendors should be required to develop a schedule for the proposed work. Pert, critical path (CPM), Gantt charts, or other graphical schedules might be required. Milestones for the project and deliverable work products should be identified as specific requirements. The discussion of work should be required to include number, timing, and skills of the expected employees. For contract software development, vendors frequently attach resumes of the intended project manager(s) and project team members for client information. If the client wants the right of refusal on all employees, a representative set of contractor staff resumes should be provided for client review.

PROJECT MANAGEMENT. Project management is an important issue in an RFP because it frequently identifies the one or two people the client will work with most closely. The requirements can include reporting structures, management of work, and problem resolution policies of the vendor firm. In general, vendors should identify an on-site manager and a more senior, vendor manager to oversee and guarantee the quality and quantity of vendor work. The resumes of one or both of those contacts should be required in the response to allow assessment of the qualifications of the managers for the proposed work.

PROJECT REPORTING. Status reporting form, content, and timing should be requested of vendors. This can be left to the vendor to describe, or can be stated as a requirement for compliance by the vendor. Normally, status meetings are held as required or weekly, whichever is more often. A written status report should be required to identify work completed, progress against the schedule, problems needing resolution for project completion, and work assignments for the next period.

VENDOR ASSUMPTIONS. Special vendor requirements should be identified. The idea behind this section is that there should be no surprises because of erroneous assumptions by a vendor after a selection is made. The vendor's assumptions are stated in the response to ensure that the client also shares the same assumptions. Any hardware, configuration, purchased software, or facilities alterations assumed by the vendor to be available for their use are solicited. For instance, when vendors build custom software, they normally assume that their employees work at the client site, use client computing equipment and software, and follow the client's employment practices.

The vendor's expectations and type of support required from the client should be identified. For instance, copying, clerical, and secretarial support might be expected. In addition, access to the users should be identified with estimates of the number and expected participants for data gathering meetings.

Further assumptions about how application information will be entered into the computer (e.g., keyboard entry by clerks, keyboard entry by programmers), the availability of computer resources for testing, and the frequency of tests for each vendor staff member should be identified.

Company Information

Information in this section should qualify the vendor as viable to perform the work. Standard company information required in an RFP includes the company history, ownership, growth, current size, previous contracts with the requesting company, and references for similar work. If the company performs a specialized service, such as LAN installation and service, it can be highlighted in this section.

Financial Information

The last of the requirements is for a cost estimate of the work. Cost estimates vary depending on whether the work is for hardware or software or services (see summary in Table 16-2). In general, the vendors' responses provide the opening for negotiation of the financial aspect of a procurement. Except for fixed-price bids defined below, the price quoted is rarely nonnegotiable.

For purchased hardware, the options are to lease, to lease with an option to buy, or to purchase the equipment. Under a **lease option**, equipment is on loan from the vendor and is paid under a monthly leasing arrangement. In general, the more equipment leased, the more flexible the vendor for lease negotiation.

The **lease with option to buy** provides a basic leasing arrangement with some percentage of the lease payment applied to purchase of the equipment. At the end of the lease period, the lessor has the option of returning the equipment to the vendor or of paying the vendor the **residual price** (i.e., the remaining value of the equipment) and purchasing the items.

A **purchase option** identifies the total current cost of the equipment and the payment terms that can be offered to entice a purchase. Frequently the purchase of very expensive equipment (e.g.,

TABLE 16-2 Financial Options

Item Being Acquired	Financial Options
Hardware	Lease Lease with Option to Buy Purchase
Software	Base License Fee Plus Monthly/Annual Maintenance Fees
Services	Time & Materials (T&M) Fixed Price T&M with Ceiling

$250,000+) can span several years and the variations between proposals can be great.

Software **package purchases** usually include a one-time **license fee** with a monthly maintenance fee, both of which are negotiable. The more sites and higher number of users, the lower the per copy price of software. As the number of sites and users increases, the average incremental cost per user decreases. The goal, then, of the licensing options is to have the vendor define available options from which a negotiation begins.

For services, the financial options are a fixed price, time and materials (T&M) estimate, or T&M with a ceiling (i.e., semifixed). A **fixed price bid** means that the work is contracted for a set price and neither side is expected to renegotiate the terms unless a major change in the contractual arrangements occurs. Fixed-price bids can be analyzed with no change.

Time and materials bids (T&M) sum the total cost of personnel time plus the cost of paper, secretarial, copy, computer, and any other vendor-supplied support in doing the work. T&M bids are frequently presented as a range of times and costs. Optimistic, realistic and pessimistic estimates might be provided. The formula for determining the cost to be used in the financial analysis develops a weighted average cost which favors realistic estimates (see Figure 16-1).

Bids that are **T&M with a ceiling** purport to provide the best risk sharing between vendor and client. Fixed-price bids put the risk of not completing the work as scheduled on the vendor who loses any moneys above the bid price. T&M bids place the risk on the client who pays for all work until it is done whether it is on schedule or not. T&M with a ceiling tries to share the risk by allowing T&M, assuming the work is on budget, or a ceiling with vendor risk if the schedule is not kept. Whether this notion works or not in reality is subject to debate. In any case, the ceiling price is most commonly used in comparisons since most projects tend to end up at that price anyway.

Schedule of RFP Process

The schedule provides important dates throughout the RFP process, including all dates and periods of time for interaction between the vendors and requesting company, and the due date for the RFP. In addition, it should include the important dates for requesting company action, especially the decision date for the winning proposal.

Description of Selection Processes

The more vendors know about the selection process, the better able they are to determine if it is worth their effort to prepare a proposal. Specific require-

Weighted Average Cost = ((Optimistic + (2 * Realistic) + Pessimistic) / 4)

Example:

Total Optimistic Time	=	4.2 Person Years
Total Realistic Time	=	6.0 Person Years
Total Pessimistic Time	=	10 Person Years

WAC = ((4.2 + (2 * 6) + 10) / 4) = 26.2 / 4
= 6.55 Person Years

FIGURE 16-1 Weighted Average Cost Formula

ments that might alter a vendor's interest in bidding are especially important. For instance, if only companies for which the proposed work is less than 10% of net income will be considered, this should be made known.

Other information that might be provided is a brief description of the selection process in terms of required and mandatory functions, and the relative importance of the four major areas of information: technical, management, financial, and company. The individual weights should *never* be identified or the responses will be written to get a good score rather than to address the design issues.

Vendor Response Requirements

An optional part of an RFP is the format for the vendor's response. The argument for requiring a set response is that the comparison of multiple proposals is simplified when the responses all use the same format. The disadvantage of a fixed response is that some important area of consideration that might have been overlooked in the RFP or in the outline for the response, might never surface until after a vendor is selected. Some companies opt for requiring the financial data to be identical but allow discretion for the remainder of the vendors' responses.

The major advantage of a standardized vendor response is an easier comparison of responses. If there is no standard format, you must first find the answer to each piece of desired information in each proposal, then create a cross-reference document, or otherwise identify each item for easy reference. Having no standard format also requires you to be much more careful in reading every document to ensure that you have identified the requisite information.

The response outline should be tailored to fit the specific proposal and to ease response evaluation. In general, the sections follow the requirements above: Technical Response, Management Approach, Corporate Information, and Cost/Price. A full outline for a vendor response is provided in Table 16-3.

One of the most important standard response formats is for financial information. Each vendor usually provides his own information in his own format in such a way that his proposal is favorably presented. The risk is that information from several vendors will not contain information in such a way that it can be compared. The solution to this problem is to tell the vendors exactly how to present financial information. A simple format that summarizes all costs for easy use and analysis is best. Figure 16-2 provides a sample financial summary format that includes all information that you, as the requesting organization, might need to compare the proposals. In customizing the form for a

TABLE 16-3 Vendor Response Outline

1.0 Technical response
- 1.1 Overview of the system
- 1.2 Diagram(s) of processes and data
- 1.3 Configuration diagram(s)
- 1.4 Performance data
- 1.5 Detailed explanation of the system features and functions
- 1.6 Compatibility (with other client equipment, applications, etc.)
- 1.7 Degree of risk in using proposed hardware, software, or application
- 1.8 Maintenance estimates
- 1.9 Reliability
- 1.10 Quality assurance and control
- 1.11 Training
- 1.12 Deliverable products

2.0 Management approach
- 2.1 Organization
- 2.2 Personnel and manpower controls
- 2.3 Vendor/subcontractor and client relationship
- 2.4 Delivery schedules and project plans
- 2.5 Proposed staffing
- 2.6 Consultants to the vendor (e.g., subcontractors)
  - 2.6.1 Subcontractor identification
  - 2.6.2 Subcontractor relationship and management structure
- 2.7 Status reporting schedule and approach

3.0 Corporate information
- 3.1 Company background—Ownership, size, age, experience, capabilities, products, services
- 3.2 Vendor previous experience with the client company
- 3.3 Vendor experience with similar projects

PRICE PROPOSAL FOR VENDOR _____ _____

Application Name _____ Number _____

_____ Development _____ _____ Operating Expenses _____

Item:	Analysis	Design	Implement	Test	Parallel	Year1	Year2	Year3	Year4	Year5
I. Hardware										
1.										
2.										
3.										
4.										
TOTAL Hardware										
II. Software										
1.										
2.										
3.										
TOTAL Software										
III. Labor (by Type) Rates: Base and Overtime										
1.										
2.										
3.										
4.										
TOTAL Labor										
IV. Other										
1. Documentation										
2. Testing/Assurance										
3. Site Preparation										
4. Supplies										
5. Travel										
6. Maintenance										
7.										
8.										
9.										
10.										
TOTAL Other										

TOTAL Project Price (Operating + Development) $ __,___,___.__

FIGURE 16-2 Sample Financial Summary Form for Vendor Response

specific procurement, you omit sections not required. So, for software-only procurement, omit all hardware sections.

Standard Contract Terms

Finally, the RFP should include any standard contract terms and conditions that might be desired. For instance, penalties for nonperformance against the contract or conditions under which payments will be made or withheld are terms of an agreement for which companies frequently require standard contract clauses.

HARDWARE

Hardware might be described in detail, or more commonly, is described by its functionality. In this section we discuss the hardware criteria that are usually in an RFP. The main categories of hardware information in this section are functionality, operational environment, and performance. Training and acceptance criteria were discussed above and also apply to hardware.

Functionality

A technical requirements section outline for hardware is shown as Table 16-4. Keep in mind that the outline would be customized to fit the specific components desired. If hardware requirements are known, they should be specified in sufficient detail to exactly identify needs. In this case, the vendors are not selecting a solution to your problem; they are bidding on a hardware configuration. For this type of specification, if the configuration does not meet all of your needs, the vendor is not liable because you detailed specific hardware.

More often, you will not know the specific hardware, but you do know the functions to be performed by the hardware. In this case, the information provided is to detail the work environment closely enough that the solution will work. For hardware, this means that all work to be performed and the constraints for the work should be identified. The number of users, work profile for each user, timing of work, volume of inputs and printed outputs, types, volume and contents of files, and software are minimal requirements to specify.

More detailed requirements, such as CPU cycle time or response time to a type of query, are provided for any critical requirements. Critical requirements, here, mean those monitoring or relating to human life (e.g., EKG monitor or space rocket

TABLE 16-4 Technical Requirements Outline for Hardware

3.0 Technical Hardware Requirements
- 3.1 CPU cycle time
- 3.2 Number of processors
- 3.3 Memory cycle time and processing
- 3.4 Number and type of registers
- 3.5 Number, type, and priority structure for interrupts
- 3.6 Memory organization, maximum addressable memory
- 3.7 Parallel operation capabilities
- 3.8 Math/graphics co-processors
- 3.9 Number, type, and transfer rate for data channels
- 3.10 Channel control unit—type, maximum device assignment, effect on CPU
- 3.11 Storage devices—number and type
- 3.12 Tape drives—density, speed, transfer rate, tracks, size
- 3.13 Disk—access time (seek + search), rotational delay, transfer rate, tracks and cylinders, capacity per unit
- 3.14 Communications control capabilities—maximum remotes, lines, interfaces
- 3.15 Ability to connect to I/O peripherals—bar code, microform, imaging, graphics, multimedia, and other special purpose equipment
- 3.16 Expandability
- 3.17 MTBF—mean time between failures
- 3.18 MTTR—mean time to repair
- 3.19 Support
- 3.20 Software compatibility for operating systems and specific packages desired
- 3.21 Site requirements—air-conditioning, electrical, heating, cooling, etc.
- 3.22 Budget limitations
- 3.23 Throughput requirements
- 3.24 Delivery requirements

lift-off). Error processing, error recovery, security, types of human intervention, and so on are all specified if they are important considerations in the decision process.

Equipment *not* desired should also be identified. For instance, most configurations can alternate between minicomputers or local area networks. If minicomputers, for instance, are not desired as a solution, the requirements should be stated as: "This proposal is for a local area networking solution to support . . ." followed by a list of major application functions. ABC's RFP might finish that sentence: ". . . a rent/return processing application for a video store."

Operational Environment

The operational environment includes the geographic, building, and room specifications for the equipment. The extent of vendor responsibility and information should be defined as explicitly as possible. If the vendor is responsible for the installation, that information should also be included. If a location is known, but the site is not improved for the equipment yet, the vendor should be required to specify flooring, air, ventilation, electrical, plumbing, and other environmental requirements of the installation. If a site is already prepared for equipment, it should be described in sufficient detail for the vendor to know whether his configuration will tolerate the environment or if further alteration is required. Schematics of desk configurations should be provided for multiunit RFPs.

Performance

Performance requirements identify the tolerance limits for different types of problems. Hardware performance requirements might include any of the following:

- Acceptable limits of downtime
- Inquiry response time
- File update response time
- Maximum percentage of communication errors
- Recovery times for hardware failures
- Maintenance and reliability requirements
- Peak and average transaction time requirements
- Geographic or other environmental constraints on equipment

Frequently, in the absence of specific hardware design requirements, performance requirements are the basis for the RFP. A sample outline for operating system or network performance requirements is shown as Table 16-5. In the table, the list includes support for all desired functions of the environment. The implication is that hardware is less important than the functional support to be provided by the operating system. Table 16-6 shows an example of performance requirements that might be used to specify a local area network to support diverse work.

Vendor responses to hardware requests should be required to include all operating system, programming language, software, and interface requirements, growth capabilities, and limitations.

Software

The basic categories of software criteria are needs, resources, performance, flexibility, and operating

TABLE 16-5 Technical Requirements Outline for Operating System or Network Performance

3.0 Generic Operating System Requirements
3.1 Instruction set and types of numeric processing
3.2 Cache memory
3.3 Hardware compatibility
3.4 Software compatibility
3.5 Virtual and real memory requirements
3.6 Job, task, data management structure and function
3.7 Multiprocessing capabilities
3.8 Fixed system overhead
3.9 Variable system overhead
3.10 Control language
3.11 Compilers supported
3.12 Packaged software supported
3.13 File access methods supported

TABLE 16-6 Example of LAN Performance Requirements

3.1 Software Requirements
- Must support and run DB2 or SQL Server software.
- Must support and run Quattro or Lotus spreadsheet.
- Must support and run ADW or IEF PC-based CASE tools.
- Must support and run Word Perfect.
- Must provide multiuser support for simultaneous users of all software with lockout at the record or data item level. File level locking should be a user-selectable option.
- Must allow levels of security including at least three levels for department, group, and user; security assignment by software package, directory, or minidisk; and by function (e.g., read, write, or both).
- Must support transaction logging either in the operating system or by other packages, e.g., the database server and CASE packages.
- Must support roll-back processing or permit it by the software in the environment.

3.2 Printing
- Must support direct access to printers by all users.
- Must provide printing of at least 100 pages per minute (ppm).
- Printer(s) must accommodate:
  - One-ply, preprinted forms,
  - 8½ × 11, 16-lb or 20-lb paper

3.2 Printing, *continued*
- 8½ × 14, 16-lb or 20-lb paper
- 4 × 9 envelopes
- 7 × 10 envelopes
- Transparencies for overheads
- Graphics
- Address labels

3.3 Processing
- Interactive processing for up to 64 PCs at a time.
- Growth to 120 PCs at a time in five years.
- Able to accommodate Word Perfect—36 users, SQL processing (see above)—12 users, spreadsheet (see above)—10 users, and CASE tool processing—six users simultaneously.
- Able to accommodate doubling of users in all categories within five years.

3.4 Benchmark Evaluation Criteria
- Current and past workload data.
- Current size and planned growth.
- Future fiscal policies that affect how computers are charged to users.
- Current and future manpower for operations support.

characteristics. These categories and the major requirements specified in an RFP are summarized in Table 16-7 and discussed below.

Needs

Needs identify context-specific requirements such as file processing, maximum number of simultaneous users, number of buffers, size of files and records that can be handled, or precision of numbers for mathematical computation.

In addition, environmental factors are extremely important. The operating system, programming language, and interfaces with other software determine whether the package can exist in your operational environment, even if your requirements are met.

Resources

Package resources identify the hardware configuration requirements for the software. The **working set** is the minimal, real memory usage when the software is running. All software is designed to have two components, a real memory component and a virtual memory component. The virtual component is swapped in and out of memory as it is accessed. When it is out of memory it is stored on some peripheral device, usually a disk. The real memory component is that minimal core of the software that maintains the beginning and ending addresses for buffers, queues, lists, arrays, and other memory the software manages, and the task management software to control the software's execution.

TABLE 16-7 Technical Requirements Outline for Customer Software or Package (Section 3.0 of Detailed RFP Outline Table 16-1.)

3.0 Technical Specifications
- 3.1 Concept and overview
  - 3.1.1 Diagrams of processes, entities, configuration, etc. as appropriate
  - 3.1.2 Functional requirements classified as mandatory or optional
  - 3.1.3 Dictionary defining all terms, items, processes, entities, and relationships in the diagrams and above sections in detail sufficient to provide complete understanding of the nature of the work expected
- 3.2 Audit and data integrity requirements
- 3.3 Security and recoverability requirements
- 3.4 Performance requirements
- 3.5 Conversion requirements
- 3.6 Interface requirements
- 3.7 Special requirements, e.g., facilities alterations
- 3.8 Training
- 3.9 Acceptance criteria

In addition to the working set, the peak usage real memory component is important. When the maximum number of users are present on the system, maximum sizes of the working set and virtual memory requirements should be identified. If there is no change to the memory requirements, but there is an optimum size for real memory for efficient processing, that should be required information.

The third type of resource information required is the amount of disk space required to store the software on disk, and the average storage requirements for several standard sizes of files. An example of a standard file is a 10,000 record file with 50 fields of 8 characters each, and three multifield indexes.

Performance

The performance requirements should identify both total throughput and individual transaction response requirements if there are any.

Flexibility

Flexibility issues are of two types. First, the packages' interconnectivity to other packages might be of interest. Second, the potential for client modification and customization might also be of interest. For either type of flexibility, the ability of the package to change, expand functionality, and loosen restrictions (e.g., move from ten open files to 256 open files at once) are a good indicator of how fast the package can change to accommodate different business needs.

To assess the vendor's capabilities, you can require a list of packages with which the package under review is compatible. Also, you can require information about the type and frequency of new releases of the software. If the new releases only fix old bugs, the software is more risky than if the new releases enhance package functionality.

To assess the extent to which you might be able to customize the software, the requirements should specify this as necessary. In general, many software companies will not honor warranties on software if any code is changed. Several mainframe software vendors, for instance D&B-MSA™,[2] specifically design their software for client customization.

Operating Characteristics

The operating characteristics requirements should require identification of the hardware, operating systems, and compatible configurations of networks on which the software can run. If interplatform connections are desired, such compatibility should be identified as a requirement.

A second operating characteristic is the form of package code. Vendors typically supply a load module form of package that cannot be examined for errors. If package does not function, the vendor is the only recourse for help. In this case, clients usually request source code and have a contract clause that specifies that source code should be held in escrow and made available at such time that the vendor

2 D&B-MSA™ is a wholly owned subsidiary of Dun and Bradstreet, New Jersey.

company goes out of business or the software ceases to function. The same argument and requirements should be written for access to data stored under proprietary methods in a software package.

The type of installation work required and vendor assistance available are the other operating characteristics of interest. Some installations require little or no planning, with the installer simply running a program that actually does the installation. Other software installations require weeks of planning, including both logical and physical design, and decisions about how internal queues, buffers, and so on will be stored. The amount of work is important for operations planning, but the type and amount of vendor support during the installation are also important.

RFP Evaluation

General Evaluation Guidelines

In general, the evaluation proceeds as follows. In each section, the required items are scored using a rating system previously decided. Vendor recommendations beyond the requirements are identified and evaluated. Through discussion and professional judgment, the contribution of enhancements to the quality of the finished work product is determined and scored. The weighted scores for the individual items are summed by vendor to yield a section score. For the financial section, formulae are applied to the bids and like numbers across proposals are compared and scored. The weighted scores for each section are summarized and summed to yield a single score for each vendor. The vendor with the highest overall score is selected.

If there is a tie or several vendors are very similar, the proposals are reevaluated for scoring method, weights, and relative importance of each type of information to eventually select a winner. The reevaluation is a form of sensitivity analysis that determines where the scoring method is most sensitive to between-vendor differences and if the method should be changed to remove the sensitivity, or if one vendor is clearly superior in some area.

Scoring Methods

The scoring for the technical evaluation requires assessment of the extent to which the vendor complies with requirements and addresses the required features and functions in the proposal. An implicit ranking of quality of proposed solution is included in the technical assessment.

There is no one right way to score a proposal. Rather, three methods are most common. One scoring method ranks each item according to its relative merit compared to the other vendors. If there are three vendors the items are all ranked on a scale of one to three. The second common method is to use a fixed scale, say zero to ten, and all items are evaluated and placed on that scale regardless of the number of proposals. The third scoring method is to simply list requirements of the application and simply score a *zero* if the requirement is not met and a *one* if the requirement is met. The chosen scale is then used for scoring technical requirements, management approach, and company history.

Figure 16-3 shows the effect of the three methods on the same set of requirements. The second method is most sensitive to qualitative differences but is also the most subjective. The binary scoring method is most objective but the least sensitive to qualitative differences. The first method is both objective and able to distinguish differences. Therefore, the first method is recommended for your use when you are given a choice.

In the remaining example, we use the ranking method for the four vendors evaluated. Each item is scored and entered on the list. When the list is complete, the item scores are multiplied by the weights to develop the weighted scores. The weighted scores are summed to give a section evaluation score for each vendor.

The flaws of the methods are seen in the example in Figure 16-3. According to the summary, we would select vendor 3 using the first two scoring methods and Vendor 2 with the binary method. However, let's say we disqualify Vendor 3 for noncompliance with the second requirement. Then, the first two methods give us different answers. This example highlights the need to do sensitivity analysis on

	Vendor 1 Ranking					
Requirements	Method 1 (Rank 1–3)		Method 2 (Rank 1–10)		Method 3 (Binary)	
Provide for at least 10 relational files to be open simultaneously	Weight 1	Rank 1	Weight .5	Rank 1	Weight .5	Rank 0
Provide at least three indexes per relation	Weight 2	Rank 3	Weight 1.5	Rank 10	Weight 1.5	Rank 1
Provide user views that join up to six relations	Weight 3	Rank 2	Weight 3	Rank 3	Weight 3	Rank 1
Summary Σ (Weight * Rank)	13		24.5		4.5	

	Vendor 2 Ranking					
Requirements	Method 1 (Rank 1–3)		Method 2 (Rank 1–10)		Method 3 (Binary)	
Provide for at least 10 relational files to be open simultaneously	Weight 1	Rank 2	Weight .5	Rank 9	Weight .5	Rank 1
Provide at least three indexes per relation	Weight 2	Rank 2	Weight 1.5	Rank 10	Weight 1.5	Rank 1
Provide user views that join up to six relations	Weight 3	Rank 1	Weight 3	Rank 4	Weight 3	Rank 1
Summary Σ (Weight * Rank)	9		31.5		5	

FIGURE 16-3 Example of Requirements Scoring Methods

the scale used in scoring to ensure balancing of weights and scores.

Technical Evaluation

In ABC's scoring example, the technical section evaluation shows one page of technical requirements that describe characteristics of a software environment without specifying the software (see Figure 16-4). This list might be an additional 8–10 pages longer when complete. To get the ranks for each item, the vendor responses are reviewed and the quality and completeness of each response is rated. The ranks are multiplied by the item weight and weighted scores are summed. The weighted score can then be normalized to account for between-section differences in the number of items ranked by dividing the weighted score by the number of items, in the example, eleven.

In both scoring systems, raw and normalized, Vendor 4's solution meets more criteria with a higher quality rating than the other vendors. Vendor 4 did not get the highest marks on all items, however. Vendors 1 and 2 have low scores for the technical section with more bottom ratings than the other vendors. These vendors would probably not be chosen and could be deleted from the remaining analysis if it were extensive.

None of the proposed language solutions has all required items with the highest rating. This means two things. First, the solution, whichever one is selected, should be reevaluated before a final deci-

	Vendor 3 Ranking					
Requirements	Method 1 (Rank 1–3)		Method 2 (Rank 1–10)		Method 3 (Binary)	
Provide for at least 10 relational files to be open simultaneously	Weight 1	Rank 3	Weight .5	Rank 10	Weight .5	Rank 1
Provide at least three indexes per relation	Weight 2	Rank 0	Weight 1.5	Rank 0	Weight 1.5	Rank 0
Provide user views that join up to six relations	Weight 3	Rank 3	Weight 3	Rank 10	Weight 3	Rank 1
Summary Σ (Weight * Rank)	12		35		3.5	

Method	Vendor 1	Vendor 2	Vendor 3
1	**13**	9	12
2	24.5	31.5	**35**
3	4.5	**5**	3.5

Selected Vendor is shown in bold for each method

FIGURE 16-3 Example of Requirements Scoring Methods (*Continued*)

sion to ensure that the application can be done without too many design compromises because of flaws in the proposed language. Second, more evaluation of possible languages for implementation can be done and a language recommendation might be made to the selected vendor. In other words, the language for implementation becomes a negotiating point for lowering the cost or for changing the proposed solution.

Management Approach Evaluation

The section of the proposal on the management approach includes all the information about how the vendor will manage the staff and the process to the satisfactory completion of the client. The schedule, staffing, management reporting, and problem resolution should be included. In addition, the vendor discusses expected resources of the client company for the work engagement.

There is no one right way to evaluate the management approach. Rather, this section is reviewed to determine the fit with client expectations and the realism of the approach. For instance, if there are more than one vendor staff, one of the staff should be designated the 'senior' person in charge of the work products and problems of the other person(s). Any personnel problems of vendor staff should not be dealt with by the client; the senior person has this responsibility. Also, a person from the vendor's management staff should be designated as responsible for guaranteeing the quality of work product by the company's staff. This person is usually the management contact for the client and is the ultimate manager for the vendor staff even though there may also be an on-site, working manager.

Proposals should be assessed in a manner consistent with that of the technical requirements. That is, either a zero/one grading system, or a ranking system, is used, depending on which is used for the technical requirements. The management techniques requested in the RFP for the management approach should have been previously prioritized and weighted. The items are listed, scored, and

Technical Requirements	Weight	Vendor 1 Rank	Vendor 2 Rank	Vendor 3 Rank	Vendor 4 Rank
3 files with 2,000—200 character records, 40 fields	5	1	2	3	4
2 files with 50,000—40 character records, 10 fields	5	2	1	4	3
Process up to six simultaneous transactions	5	2	1	4	3
Up to six attributes in compound key	5	1	2	3	4
Max text field length of 300 characters	3	2	1	3	4
Max integer length 15 digits	3	2	1	3	4
Max decimal number 9.2	3	1	2	3	4
Provide for at least 10 relational files to be open simultaneously	4	3	1	2	4
Provide at least three indexes per relation	5	3	2	1	4
Provide user views that join up to six relations	5	2	1	3	4
Supports bar code reader	4	2	4	1	3
Summary Σ (Weight * Rank) for items shown	–	90	77	129	174
Normalized score	–	8.2	7	11.7	15.8

FIGURE 16-4 ABC Technical Scoring

weighted. Finally, a weighted average score for management approach is computed for each vendor and added to the financial summary sheet.

In the example shown in Figure 16-5, Vendor 1 omitted resumes of the proposed staff and lost several rating points as a result. Vendor 2 assumed many more client resources than the company was willing to commit and lost points as a result. Only Vendors 3 and 4 provided information that was complete. Their scores reflect their proposals' assessed quality differences.

Financial Evaluation

The next analysis evaluates the financial aspects of the proposals. The financial evaluation is independent from the technical evaluation and assumes that all required features are present. In fact, the technical and financial evaluations might be done by different people in different departments. The project manager and SEs usually perform the technical evaluation, while the project manager and/or a financial support group might do the financial evaluation.

Management Requirements	Weight	Vendor 1 Rank	Vendor 2 Rank	Vendor 3 Rank	Vendor 4 Rank
Schedule and staffing	5	1	2	3	4
Project management	5	2	1	4	3
Status reporting	5	2	1	4	3
Problem management	5	1	2	3	4
Summary					
Σ (Weight * Rank)	–	30	30	70	70
Normalized score*	–	7.5	7.5	17.5	17.5

*Normalized Score = Σ (Weight * Rank) / # Items

FIGURE 16-5 ABC Management Approach Scoring

Recall that for services such as custom software development, there are three types of financial proposals: fixed, time/materials (T&M), and T&M with a ceiling (semifixed). For hardware, there are three types of proposals: lease, lease with option to buy, and purchase. And for software packages, there is a basic license fee plus a maintenance fee. The first step in the financial analysis is to determine what set of numbers to compare. The proposal should specify the type(s) of financial bids solicited, but, if not, the three types need to be equated for proper comparison.

After deciding on which numbers to compare, a simple net present value (NPV) analysis may be developed (see Chapter 6). Recall that NPV computes the present value of multitime period expenditures, assuming a specific interest rate on money (see Figure 16-6). If all vendors' proposed expenditures are in the same time period, NPV is not necessary and a simple comparison is used. The final value of each project is entered on the summary evaluation sheet (see Figure 16-7). Other analyses might include payback period, cost per vendor employee, and so on, depending on company convention.

In Figure 16-7, the present value of all hardware options is listed and separated from the cost of labor. Then, rankings for hardware and software are applied based on the low cost. Both average and nor-

$$NPV = \sum_{t=0}^{n} \frac{B_t - C_t}{(1+d)^t}$$

Where: d = Discount interest rate

n = Life of project in years

B_t = Value of benefits in period t

C_t = Value of costs in period t

For instance, assume the life of the project is five years, and the per period cost is $1,000,000 at .075 interest. The benefits of the project for the five years are zero, $100,000, $250,000, $450,000, and $2,000,000. The NPV is:

$$NPV = (-1{,}000{,}000)/1.075 + (100{,}000 - 1{,}000{,}000)/1.075^2 + (250{,}000 - 1{,}000{,}000)/1.075^3 + (700{,}000 - 1{,}000{,}000)/1.075^4 + (2{,}000{,}000 - 1{,}000{,}000)/1.075^5$$

$$= -930{,}232 - 576{,}923 - 443{,}548 - 223{,}880 + 689{,}655$$

$$= -\$1{,}494{,}928$$

FIGURE 16-6 Sample Net Present Value Computation

Financial Summary	Vendor 1	Vendor 2	Vendor 3	Vendor 4
Net Present Value Hardware Lease	$22,000	$30,000	$27,250	$22,300
NPV Hardware Lease with Option to Purchase	$30,000	$32,000	$31,750	$24,600
NPV Hardware Purchase	$45,000	$37,800	$37,500	$32,500
Total Cost T&M Labor	$17,500	$22,600	$28,400	$27,500

	Weight	Vendor 1 Rank	Vendor 2 Rank	Vendor 3 Rank	Vendor 4 Rank
Hardware Rank	4	4	1	2	3
Software Rank	6	4	1	2	3
Summary Σ (Weight * Rank)		40	10	20	30

FIGURE 16-7 ABC Financial Evaluation Summary

malized scores can be generated as we have shown. The normalized scores are used in the comparison with the scores of the other sections. In the rankings, the higher the score, the lower the cost. In this case, Vendor 1 receives the highest cost scores for both hardware and software. Vendor 4, the highest ranked vendor in the technical section, was third on both items, with a weighted score of 3. If hardware and software are equally important, we could have averaged the scores for each vendor. Using weights which are somewhat higher than the weights for the other score categories increases the importance of the financial evaluation relative to the technical and other evaluations.

Company Evaluation

One risk any client takes when contracting work to others is that the vendor might not meet the terms of the contract for some reason. The company evaluation is one way to define such risks and assign a score to each vendor company.

In general, the longer the company has been in business and the larger the size, the less likely the company is to go out of business. Similarly, the smaller a percentage of the total company's work this particular contract is, the less likely schedule problems, for fixed price work, for instance, are to severely hamper the vendor's ability to do business. The first score assessed, then, is one of risk that the vendor can do this work without straining his or her own organization.

The second type of evaluation gives credit for past work with the client company. The idea is to favor a company with successful past experience because their personnel are likely to know the client's way of doing business and need less introductory time than vendors without that experience. Other project managers who know the vendor should be asked about the quality and quantity of work of vendor employees, satisfaction with the vendor, and compliance with contract terms. A high ranking should be given for successful past projects and a low ranking (e.g., negative) for unsuccessful past work. Obviously, no ranking can be given for a company with no history at the client site. If this item is the decision criteria for a proposal, then the technical evaluation should be reevaluated to ensure that the best proposal is being selected.

A similar evaluation gives vendors who have developed similar work products credit for that knowledge. Vendors who are already familiar with the problem domain, and who need less start-up time to learn the domain, are favored over those without that knowledge.

Vendor Company Criteria	Weight	Vendor 1 Rank	Vendor 2 Rank	Vendor 3 Rank	Vendor 4 Rank
Age/Size	5	3	1	4	2
Similar work	5	3	1	2	4
Work with ABC	5	0	0	0	0
Reputation	5	3	1	4	2
Summary					
Σ (Weight * Rank)	–	45	15	50	40
Normalized score		9	3.7	12.5	10

FIGURE 16-8 ABC Vendor Company Rating

Vendors who claim similar experience should be checked by discussing the experience with reference clients. Unless another firm gives a positive recommendation, no credit for domain experience should be given. A neutral or negative recommendation does not necessarily mean a negative rating. Rather, a negative rating raises a flag that not all projects are perfect but should not cause a vendor to be disqualified. If all recommendations are negative, then the vendor might be given a negative rating.

The scores for company evaluation are entered in the summary sheet which is completed. The final scores in all areas are multiplied by the weight for the section and summed to develop a final weighted assessment. The company with the highest overall score is selected unless some extenuating circumstance is present.

In Figure 16-8, all four vendors scored zero in prior work with ABC, meaning that none have worked with ABC before. Vendor 4, the preferred vendor based on technical scores, ranks low in age/size and low in reputation. These numbers can be interpreted as identifying a small, fairly new company that has had some successes and some failures. This interpretation implies some risk in using Vendor 4.

If we look at Vendor 3, who was ranked second on the technical list, the age/size and reputation are the best of the four vendors. The problem here is that Vendor 3 has little experience with similar work, thus, also identifying a source of risk.

The summary form (see Figure 16-9) shows the proposals summarized on one page for a management overview. The weighted and normalized weighted scores should both be shown as an indicator of the sensitivity of the weighting system in selecting the proposal winner.

AUTOMATED SUPPORT TOOLS FOR VENDOR EVALUATION

There are no automated tools that are advertised as specifically for RFP use. Rather, there is general purpose software that can be used for different parts of the work. For instance, spreadsheets can be used for the financial analysis and for maintaining and monitoring the scores easily. Word processing and CASE tools might be used in the preparation of the RFP document, but are of less use in evaluating the vendor proposals. Table 16-8 shows the most popular spreadsheets on the market at this time. Any of these packages can be used in the RFP analysis.

SUMMARY

This chapter discusses the procurement of hardware, software, or services for an IS organization. The formal RFP process includes the development of work

Criteria	Weight	Vendor 1 Rank	Vendor 2 Rank	Vendor 3 Rank	Vendor 4 Rank
Technical	.4	90	77	129	174
Management approach	.2	30	30	70	70
Company	.1	45	15	50	40
Financial average rank	.3	40	10	20	30
Cost of cheapest alternative		$42,500	$42,600	$55,650	$49,800
		Vendor 1	Vendor 2	Vendor 3	Vendor 4
Total weighted score		58.5	41.3	76.6	96.6
Total weighted normalized score		14.6	10.3	19.2	24.2

FIGURE 16-9 ABC Vendor Summary Ratings

requirements, identification of vendors, development of a Request for Proposal (RFP), management of the RFP process, evaluations of the proposals, and selection of a vendor.

An RFP consists of a management summary, statement of requirements, proposal process description with important dates, and standard contract terms. Optional sections of the RFP include a definition of the vendor response.

Ranking RFP responses requires the definition of the ranking scheme and of weights signifying the relative importance of the ranked items. The least subjective, most informative of the three common ranking schemes is one that uses the number of vendors as the number of ranks. Other options are a binary system and a subjective ranking based on an arbitrary number of ranks, such as 10.

All response areas—technical, managerial, company, and financial—are ranked taking care to compare like things across the vendors. When complete, the weighted ranks are summed by section and for the whole RFP, and the vendor with the highest

TABLE 16-8 Automated Tools for Vendor Evaluation

Product	Company	Technique
COMNET, LANNET	CACI San Diego, CA	Simulation of network performance for different network operating systems
Excel, Multiplan	MicroSoft Redmond, WA	Spreadsheet for financial evaluation
Lotus 1-2-3	Lotus Development Corp.	Spreadsheet for financial evaluation
Quattro Pro	Borland Corp.	Spreadsheet for financial evaluation

score is selected. Do sensitivity analysis of the ranking scheme to minimize obvious bias.

REFERENCES

Joslin, Edward O., *Computer Selection*. Reading, MA: Addison-Wesley Publishing Co., Inc., 1968.

King, John L., and Edward L. Schrems, "Cost-benefit analysis in information systems development and operation," in *Computing Surveys*, Vol. 10, #1, March 1978, p. 25.

Lucas, Henry C., Jr., *The Analysis, Design and Implementation of Information Systems*, 4th ed., New York, Mitchell McGraw-Hill, 1992.

Stamper, David, *Business Data Communications*, 3rd ed., Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1991.

KEY TERMS

acceptance criteria
benchmark
company requirements
data communications network
financial requirements
fixed price bid
lease option bid
lease with option to buy bid
license fee
make-buy decision
management approach
management requirements
package purchase
proposal process
purchase bid
request for information (RFI)
request for proposal (RFP)
request for quotation (RFQ)
residual price
T&M with ceiling bid
technical requirements
time and materials bid (T&M)
working set

EXERCISES

1. Using the information provided for the ABC Case in Chapter 16, develop a different way of scoring the vendor responses that is plausible. Defend the use of your method.
2. Develop an analysis of two well-known packages, such as Lotus and Quattro spreadsheets. What features are the same? Which are different? How would you choose between them?
3. Develop a list of issues for deciding what PC software packages should be the standards for a company. Are the criteria software features organizational in nature? Why?

STUDY QUESTIONS

1. Define the following terms:

benchmark	T&M
fixed price bid	management approach
make-buy decision	technical requirements
RFI	working set

2. Is the process of selecting a product through the RFP completely objective? Why or why not?
3. What is the purpose of an RFP?
4. How does the RFP process differ from the informal procurement process?
5. List and describe the seven types of financial proposals.
6. Why are there so many types of financial proposals?
7. List five criteria to be provided in a hardware RFP.
8. List five criteria to be provided in a software package RFP.
9. List five criteria to be provided in a software development RFP.
10. What RFP criteria are provided for a network?
11. Describe the three types of scoring systems for vendor proposals, identifying the pros and cons of each.
12. Why is a standard for vendor responses a good idea?
13. What formula is applied to develop the financial analysis? When is it *not* needed?
14. What is the purpose of a benchmark? For which type(s) of procurement is benchmarking used?
15. How can you determine a company's reputation?

★ EXTRA-CREDIT QUESTION

1. Take some application that might be used at ABC Video—accounts payable, general ledger, payroll, rental order processing—and perform a software evaluation for two competing products. Try to be objective in the criteria and weights you assign. What are the deciding factors in the evaluation?

CHAPTER 17

TESTING AND QUALITY ASSURANCE

INTRODUCTION

Testing is the process (and art) of finding errors; it is the ultimate review of specifications, design, and coding. The *purpose* of testing is to guarantee that all elements of an application mesh properly, function as expected, and meet performance criteria.

Testing is a difficult activity to accept mentally because we are deliberately analyzing our own work or that of our peers to find fault. Thus, after working in groups and becoming productive teams, we seek to find fault and uncover mistakes through testing. When the person conducting the test is not on the project as, for instance, acceptance testers, they are viewed as adversaries.

Testing is a difficult activity for management to accept because it is costly, time consuming, and rarely finds all errors. Frequently, resources are difficult to obtain and risks of not testing are inadequately analyzed. The result is that most applications are not tested enough and are delivered with 'bugs.'

Research studies show that software errors tend to cluster in modules. As errors are found in a tested unit, the probability that more errors are present increases. Because of this phenomenon, as severe errors are found, the lower the confidence in the overall quality and reliability of the tested unit should be.

In this chapter, we discuss useful strategies for testing and the strategies which are most applicable to each level of testing. Then, we discuss each level of testing and develop test plan examples for the ABC rental system. Finally, automated test support within CASE tools and independent test support tools are defined and examples listed. The next section defines testing terminology.

TESTING TERMINOLOGY

As above, **testing** is the process (and art) of finding errors. A good test has a high probability of finding undiscovered errors. A successful test is one that finds new errors; a poor test is one that never finds errors.

There are two types of errors in applications. A **Type 1 error** defines code that does not do what it is supposed to do; these are errors of omission. A **Type 2 error** defines code that does something it is not supposed to do; these are errors of commission. Type 1 errors are most prevalent in newly developed applications. Type 2 errors predominate in maintenance applications which have code 'turned off' rather than removed. Good tests identify both types of errors.

Testing takes place at different levels and is conducted by different individuals during the application development. In this chapter we discuss the testing performed by the project team and testing performed by outside agents for application acceptance. Project team tests are termed **developmental tests**. Developmental tests include unit, subsystem, integration, and system tests. Tests by outside agents are called quality assurance (QA) and acceptance tests. The relationship between testing levels and project life-cycle phases are summarized in Figure 17-1.

A **unit test** is performed for each of the smallest units of code. **Subsystem, integration** tests verify the logic and processing for suites of modules that perform some activity, verifying communications between them. **System tests** verify that the functional specifications are met, that the human interface operates as desired, and that the application works in the intended operational environment, within its constraints. During maintenance, testers use a technique called regression testing in addition to other types of tests. **Regression tests** are customized to test that changes to an application have not caused it to *regress* to some state of unacceptable quality.

Finally, outside agents perform **quality assurance (QA) tests of acceptance** for the application. The outside agent is either the user or a user representative. The goal is to perform an objective, unbiased assessment of the application, and an outside agent is considered more objective than a team member. QA tests are similar to system tests in their makeup and objectives, but they differ in that they are beyond the control of the project team. QA test reports usually are sent to IS and user management in addition to the project manager. The QA tester plans his own strategy and conducts his own test to ensure that the application meets all functional requirements. QA testing is the last testing done before an application is placed into production status.

Each test level requires the definition of a strategy for testing. Strategies are either white box or black box, and either top-down or bottom-up. **Black-box strategies** use a 'toaster mentality': You plug it in,

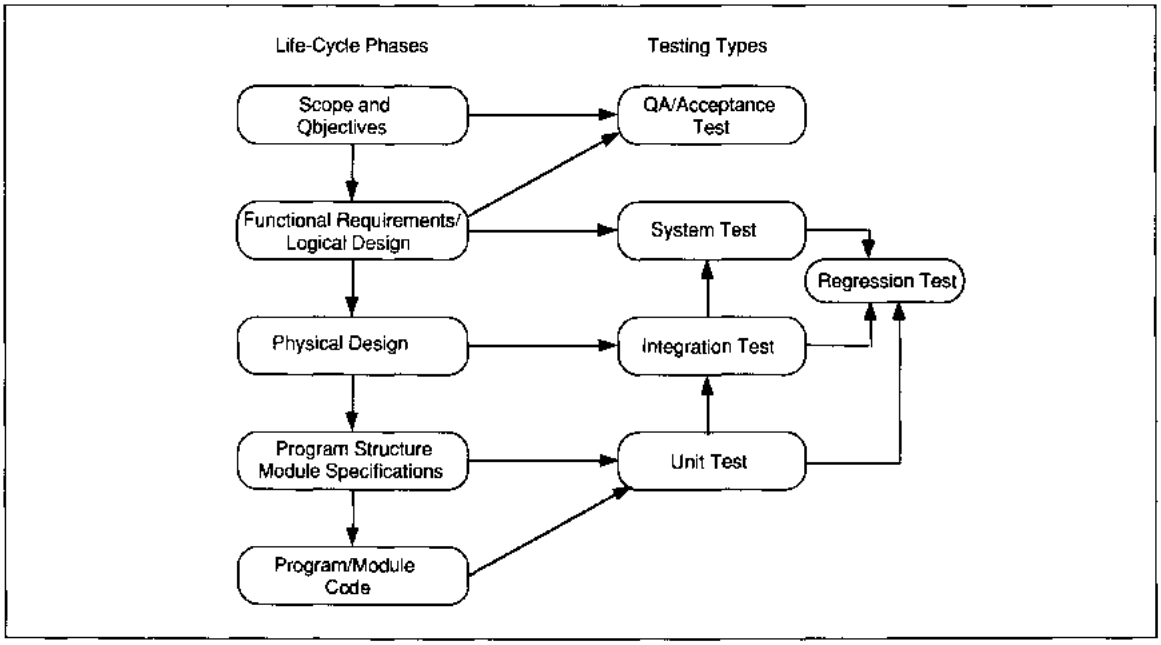


FIGURE 17-1 Correspondence between Project Life-Cycle Phases and Testing

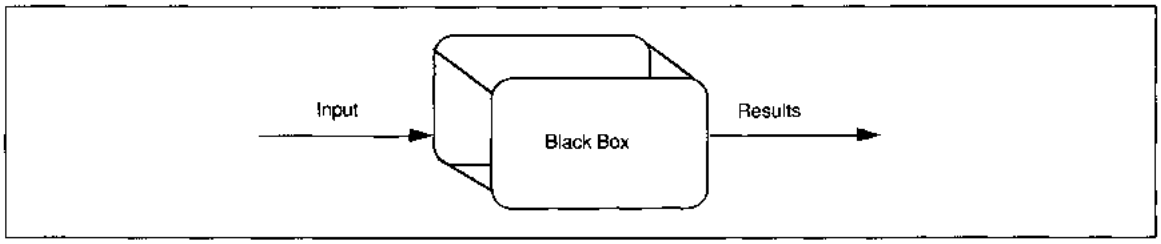


FIGURE 17-2 Black Box Data Testing Strategy

it is supposed to work (see Figure 17-2). Created input data is designed to generate variations of outputs without regard to how the logic actually functions. The results are predicted and compared to the actual results to determine the success of the test.

White-box strategies open up the 'box' and look at specific logic of the application to verify *how* it works (see Figure 17-3). Tests use logic specifications to generate variations of processing and to predict the resulting outputs. Intermediate and final output results can be predicted and validated using white-box tests.

The second type of testing strategy defines how the test and code development will proceed. **Top-down testing** assumes that critical control code and functions will be developed and tested first (see Figure 17-4). These are followed by secondary functions and supporting functions. The theory is that the more often critical modules are exercised, the higher the confidence in their reliability can be.

Bottom-up testing assumes that the lower the number of incremental changes in modules, the lower the error rate. Complete modules are coded and unit tested (see Figure 17-5). Then the tested module is placed into integration testing.

The test strategies are not mutually exclusive; any of them can be used individually and collectively. The test strategy chosen constrains the type of errors that can be found, sometimes necessitating the use of more than one. Ideally, the test for the application combines several strategies to uncover the broadest range of errors.

After a strategy is defined, it is applied to the level of test to develop actual test cases. **Test cases** are individual transactions or data records that cause logic to be tested. For every test case, all results of processing are predicted. For on-line and real-time applications, **test scripts** document the interactive dialogue that takes place between user and application and the changes that result from the dialogue. A **test plan** documents the strategy, type, cases, and scripts for testing some component of an application. All the plans together comprise the test plan for the application.

Testing is iterative until no errors, or some acceptable number of errors, are found. In the first step

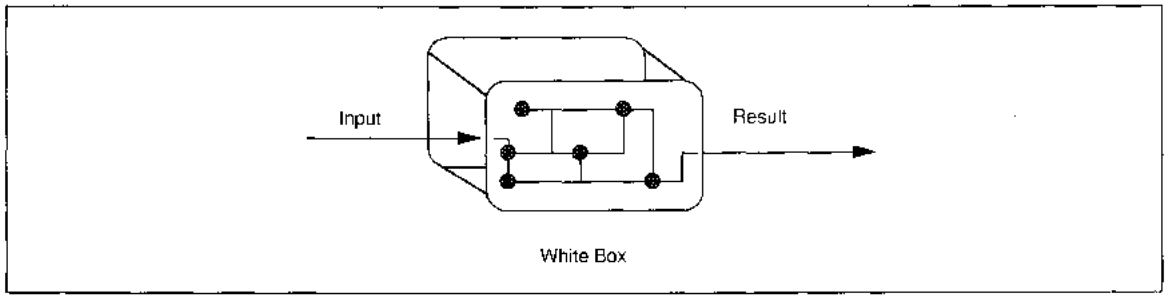


FIGURE 17-3 White Box Logic Testing Strategy

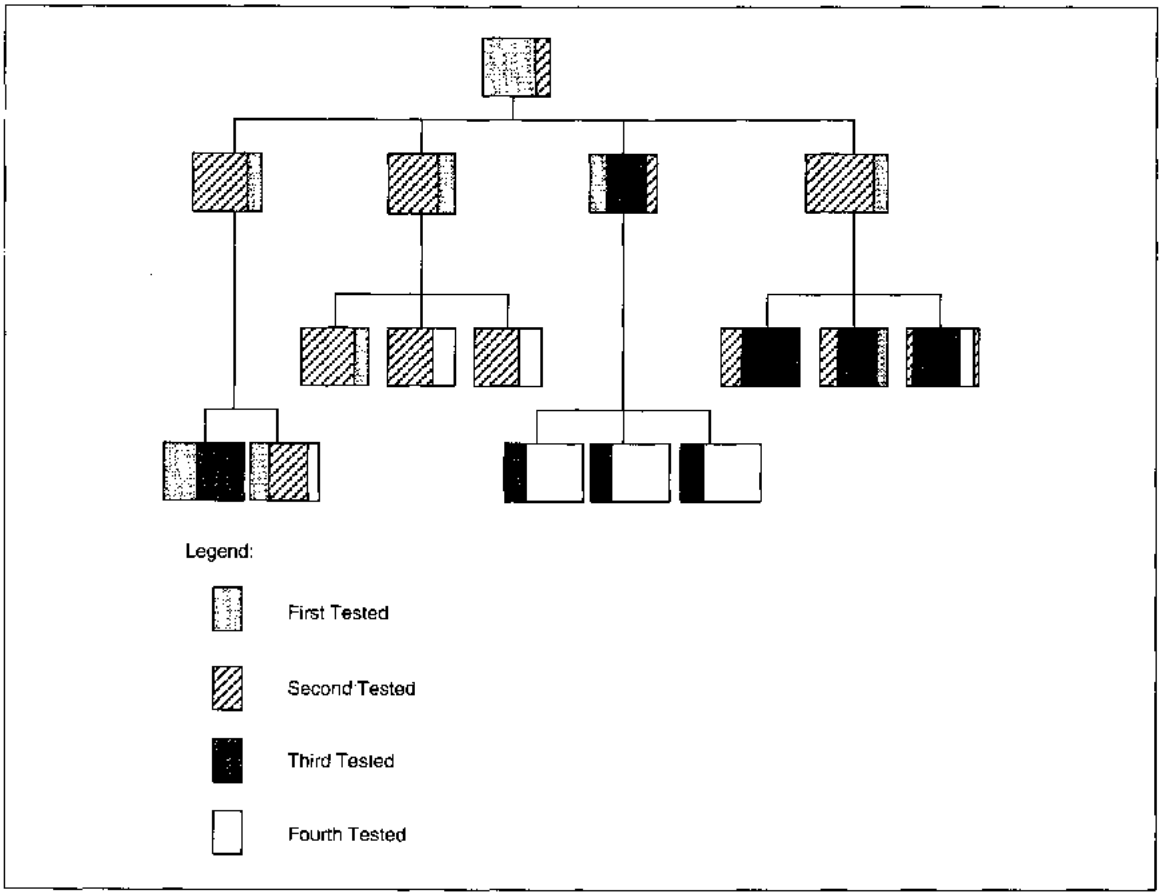


FIGURE 17-4 Top-Down Testing Strategy

of the testing process, test inputs, configuration, and application code are required to conduct the actual test. The second step is to compare the results of the test to predicted results and evaluate differences to find errors. The next step is to remove errors, or 'debug' the code. When recoding is complete, testing of changes ensures that each module works. The revised modules are then reentered into the testing cycle until a decision to end testing is made. This cycle of testing is depicted in Figure 17-6 for a top-down strategy.

The process of test development begins during design. The test coordinator assigned should be a capable programmer-analyst who understands the requirements of the application and knows how to conduct testing. The larger and more complex the application, the more senior and skilled the test coordinator should be. A test team may also be assigned to work with the coordinator on large, complex projects. The testing team uses the functional requirements from the analysis phase and the design and program specifications from the design phase as input to begin developing a strategy for testing the system. As a strategy evolves, walk-throughs are held to verify the strategy and communicate it to the entire test team. Duties for all levels of testing are assigned. Time estimates for test development and completion are developed. The test team works independently in parallel with the development team to do their work. They work with the DBA in developing a test database that can support all levels of testing. For unit testing, the test team verifies results

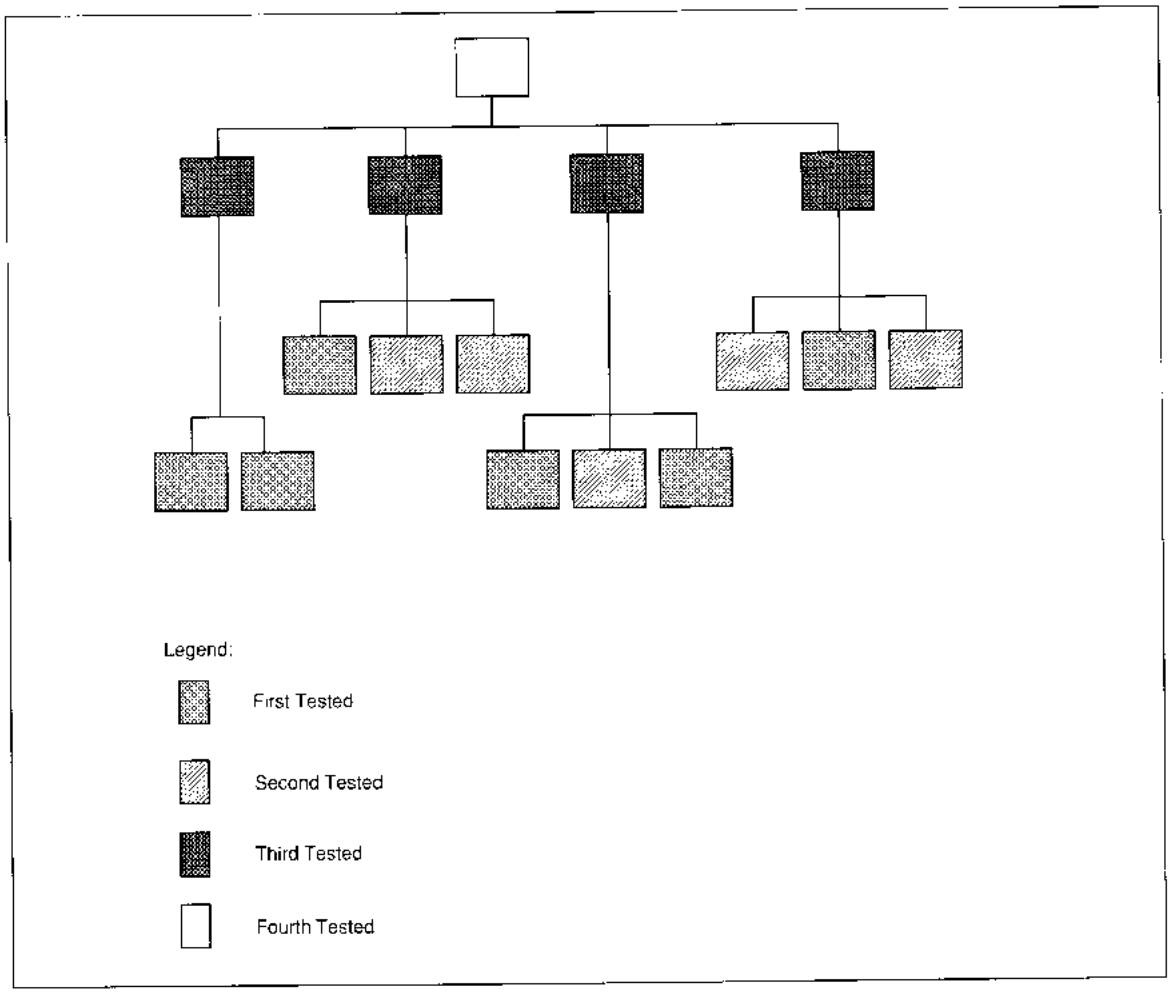


FIGURE 17-5 Bottom-Up Testing Strategy

and accepts modules and programs for integration testing. The test team conducts and evaluates integration and system tests.

TESTING STRATEGIES

There are two kinds of testing strategies. The first type of strategy relates to how logic is tested in the application. Logic testing strategies are either black-box or white-box. Black-box testing strategies assume that module (or program or system) testing is concerned only that what goes in comes out correctly. The details of logic are hidden and not specifically analyzed. Black-box strategies are data-driven, which means that all test cases are based on analysis of data requirements for the test item.[1]

White-box approaches to testing assume that specific logic is important and to be tested. White-box tests evaluate some or all of the logic of a test item to verify correct functioning. White-box strategies are

1 *Test item* is the term used through the remainder of the discussion to identify some *thing* being tested. A test item might be a module, group of modules, or the whole application.

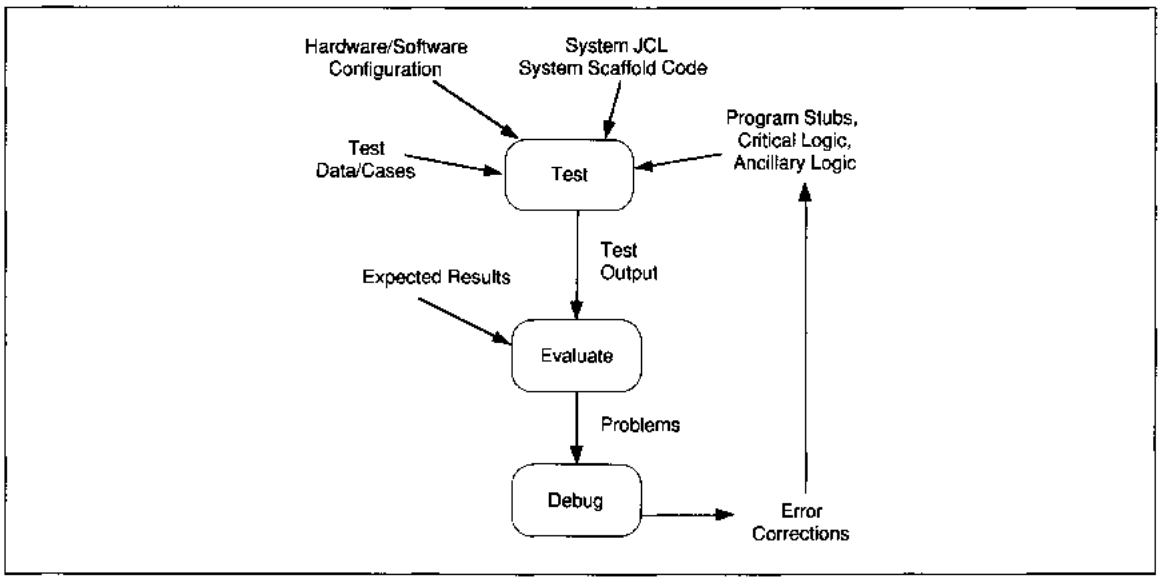


FIGURE 17-6 Testing Information Flow

logic-driven, which means that all test cases are based on analysis of expected functions of the test item.

The second type of testing strategy relates to how testing is conducted, regardless of logic testing strategy. These conduct, or process, strategies are top-down and bottom-up. Both top-down and bottom-up testing fit the project life-cycle phases in Figure 17-1; the difference is in the general approach. Top-down is incremental; bottom-up is 'all or nothing.'

Top-down testing assumes the main application logic is most important. Therefore, the main logic should be developed and tested first and continuously throughout development. Continuous successful testing raises confidence levels about code reliability. **Program stubs** that contain minimal functional logic are tested first with additional logic added as it is unit tested. Top-down testing frequently requires extra code, known as **scaffolding**, to support the stubs, partial modules, and other pieces of the application.

Bottom-up testing assumes that individual programs and modules are fully developed as standalone processes. These are tested individually, then combined for integration testing. Bottom-up testing treats test phases as somewhat discrete. Unit testing leads to integration testing which leads to system testing. The next section discusses variations of black- and white-box testing strategies.

Black-Box Testing

Black-box testing attends to process results as evidenced by data. The test item is treated as a *black box* whose logic is unknown. The approach is effective for single function modules and for high-level system testing. Three commonly used methods of black box testing are:

- equivalence partitioning
- boundary value analysis
- error guessing

A fourth method that is less common in business, cause-effect graphing, is also used. Each of these methods are described in this section.

Equivalence Partitioning

The goals for equivalence partitioning are to minimize the number of test cases over other methods and design test cases to be representative of sets of data. For the given level of test, the test item data

inputs are divided into **equivalent partitions** each representing some set of data. Then, test cases are designed using data from each representative, equivalent set. The theory is that by exhaustively testing one item from each set, we can assume that all other *equivalent* items are also exhaustively tested.

For instance, at the module level, field values identify equivalent sets. If the field domain is a range of values, then one set is allowable values and the other set is disallowed values. The analysis to define equivalent domain sets continues for each data item in the input.

Equivalence partitioning gains power when used at more abstract levels than fields, however. For instance, interactive programs, for integration tests, can be defined as equivalent sets at the screen, menu selection, or process levels. At the system test level, equivalence can be defined at the transaction, process, or activity level (from Information Engineering).

Test scripts for on-line applications can be black-box equivalence partitioning tools. A test script is an entry-by-entry description of interactive processing. A script identifies what the user enters, what the system displays in response, and what the user response to the system should be. *How* any of these entries, actions, and displays takes place is not tested.

Boundary Value Analysis

Boundary value analysis is a stricter form of equivalence partitioning that uses *boundary* values rather than *any* value in an equivalent set. A boundary value is *at the margin*. For example, the domain for a month of the year ranges from one to 12. The boundary values are one and 12 for valid values, and zero and 13 for the invalid values. All four boundary values should be used in test cases. Boundary value analysis is most often used at the module level to define specific data items for testing.

Error Guessing

Contrary to its name, error guessing is not a random guessing activity. Based on intuition and experience, it is easy for experts to test for many **error** conditions by **guessing** which are most likely to occur. For instance, dividing by zero, unless handled properly, causes abnormal ending of modules. If a module contains division, use a test that includes a zero divisor. Since it is based on intuition, error guessing is usually not effective in finding all errors, only the most common ones. If error guessing is used, it should always be used with some other strategy.

Cause-Effect Graphing

One shortcoming of equivalence and boundary testing is that compound field interactions are not identified. Cause-effect analysis compensates for this shortcoming. A **cause-effect graph** depicts specific transformations and outputs as effects and identifies the input data causing those effects. The graphical notation identifies iteration, selection, Boolean, and equality conditions (see Figure 17-7). A diagram of the effects works backward to determine and graph all causes. Each circle on the diagram represents a sequence of instructions with no decision or control points. Each line on the diagram represents an equivalent class of data and the condition of its usage. When the graph is done, at least one valid and one invalid value for each equivalent set of data on the graph is translated into test case data. This is considered a black-box approach because it is concerned not with logic, but with testing data value differences and their effect on processing. An example cause-effect graph for Customer Create processing is shown in Figure 17-8.

Cause-effect graphing is a systematic way to create efficient tests. The trade-off is in time to develop the set of graphs for an application versus the time consumed executing large numbers of less efficient, possibly less inclusive test cases. The technique is used more in aerospace than in general business.

Cause-effect graphs are more readily created from DFDs, PDFDs, and state-transition diagrams than from Booch diagrams even though it is particularly useful for real-time and embedded systems. Both types of systems use state-transition diagrams to show the causes and effects of processing. A cause-effect graph can be superimposed on a state-transition diagram or easily developed from the state-transition diagram. Cause-effect graphing can be used in place of white-box approaches whenever

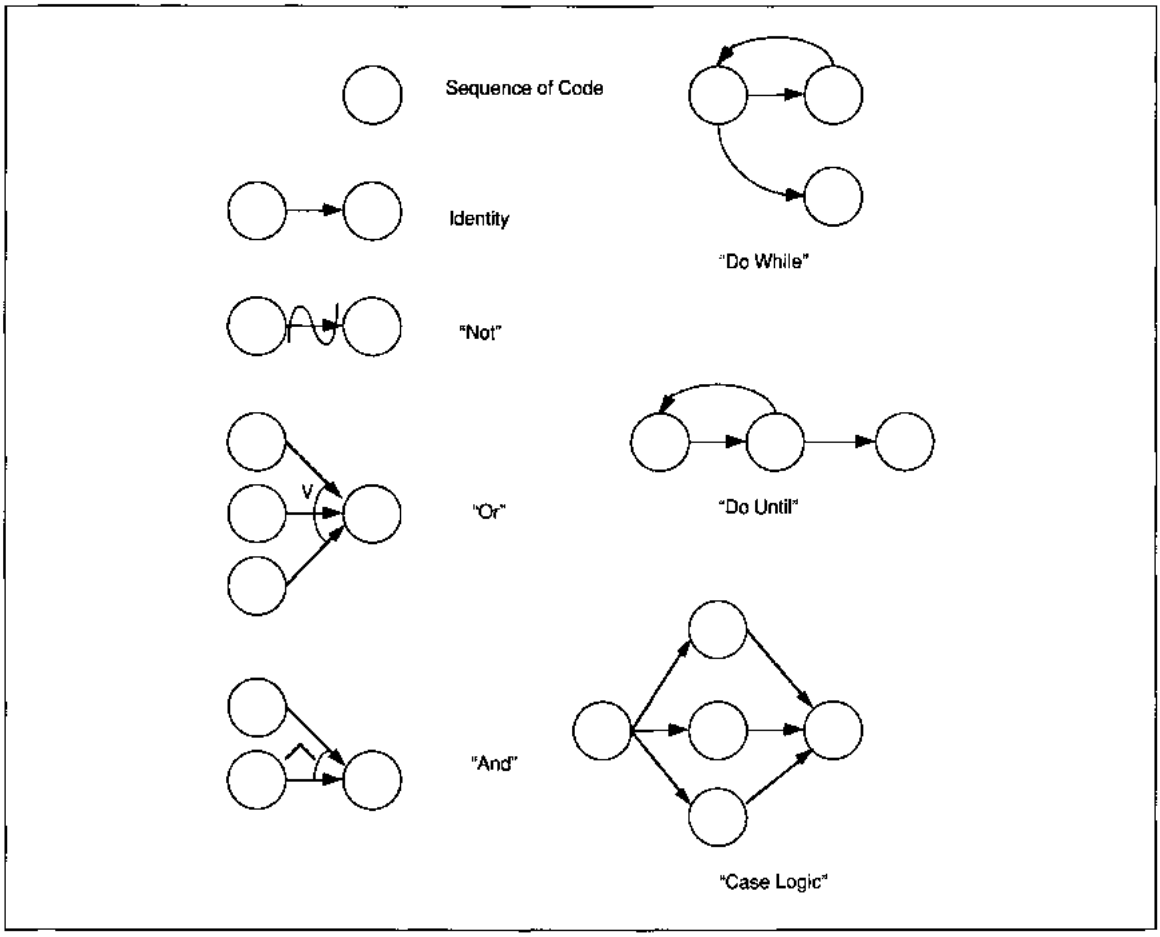


FIGURE 17-7 Cause-Effect Graphical Notation

specific logic cannot be realistically tested because of combinatorial effects of multiple logic conditions.

White-Box Testing

White-box testing evaluates specific execute item logic to guarantee its proper functioning. Three types of white-box techniques are discussed here: logic tests, mathematical proofs, and cleanroom testing. Logic coverage can be at the level of statements, decisions, conditions, or multiple conditions. In addition, for mathematically specified programs, such as predicate logic used in artificial intelligence applications, theorem proof tests can be conducted. The newest development in white-box strategies is the 'clean room' approach developed by IBM.

Logic Tests

Logic tests can be detailed to the statement level. While execution of every statement is a laudable goal, it may not test all conditions through a program. For instance, an *if* statement tested once tests either success or failure of the *if*. At least two tests are required to test both conditions. Trying to test all conditions of all statements is simply not practical. In a small module with 10 iterations through a four-path loop, about 5.5 million test cases would

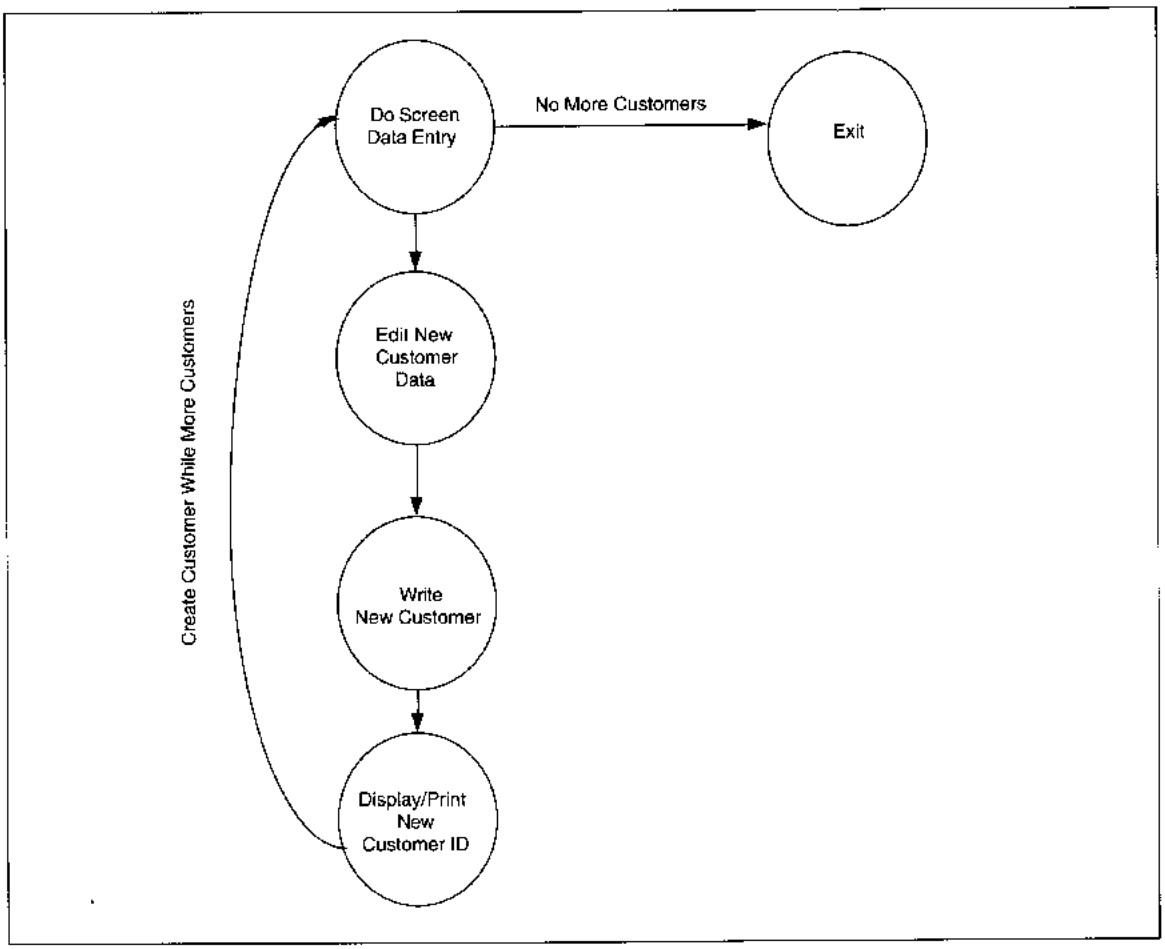


FIGURE 17-8 Cause-Effect Graph for Customer Create

be needed to try all possible combinations of paths (i.e., $4^{10} + 4^9 + 4^8 \ldots + 4^1$). Obviously, some other method of deciding test cases is needed. The other white-box logic testing strategies offer some alternatives.

Decision logic tests look at each decision in a module and generate test data to create all possible outcomes. The problem is that decisions are not always discrete and providing for compound decisions requires a different strategy. A problem with logic tests at this level is that they do not test module conformance to specifications. If the test is developed based on the specification, but the specification is interpreted differently by the programmer (for better or worse), the test is sure to fail. The solution to this issue is to require program specifications to detail all logic. While this may be practical for first- and second-generation languages (i.e., machine and assembler languages), it defeats the purpose of higher level, declarative languages.

Condition logic tests are designed such that each condition that can result from a decision is exercised at least once. In addition, multiple entry conditions are tested. Thus, condition tests are more inclusive than decision logic tests. They still suffer from the problem of ignoring compound decision logic.

Multicondition tests generate each outcome of multiple decision criteria and multiple entry points

to a module. These tests require analysis to define multicriteria decision boundaries. If the boundaries are incorrectly defined, the test is ineffective. When designed properly, multicondition logic tests can minimize the number of test cases while examining a large number of conditions in the case. The use of this technique requires practice and skill but can be mentally stimulating and even fun.

Mathematical Proof Tests

When evaluating logic, the goal is zero defects. One method of approaching zero defects is to apply mathematical reasoning to the logic requirements, proving the correctness of the program. This method requires specifications to be stated in a formal language such as the **Vienna Development Method (VDM)**. Formal languages require both mathematical and logic skills that are beyond the average business SE's ability at the present time. An example of a general process overview for a payroll system as specified in VDM is shown as Figure 17-9. While a detailed discussion of these methods is beyond the scope of this text, they deserve mention because they are the only known way for attaining zero defects and knowing it.

Cleanroom Tests

Cleanroom testing is an extension of mathematical proof that deserves some comment. Cleanroom testing is a manual verification technique used as a part of **cleanroom development**. The theory of cleanroom development is that by preventing errors from ever entering the process, costs are reduced, software reliability is increased, and the zero defect goal is attained. The process was introduced in IBM in the early 1980s by Mills, Dyer, and Linger, and applies hardware engineering techniques to software. Formal specifications are incrementally developed and *manually* verified by walk-through and inspections teams. Any program that is not easily read is rewritten. All program development is on paper until all verification is complete.

Cleanroom testing techniques are walk-throughs and formal mathematical verification. The goal is to decompose every module into functions and their linkages. Functional verification uses mathematical techniques, and linkage verification uses set theory whenever possible to prove the application design and code.

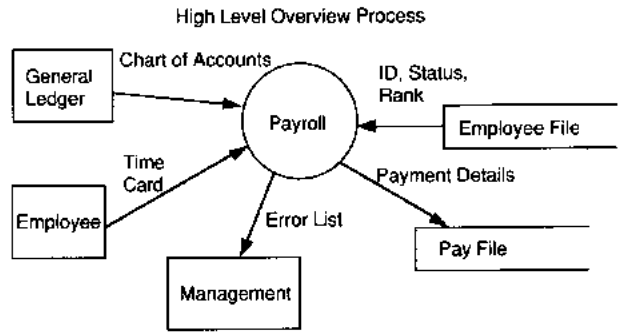
After verification, an independent testing team compiles and executes the code. Test data is compiled by analysis of the functional specification and is designed to represent statistical proportions of data expected to be processed by the live system. In addition to normal data, every type of catastrophic error is produced to test that the application does degrade gracefully.

The success of cleanroom development and testing is such that more than 80% of reported projects have an average failure time of less than once every 500 software years. Software years are counted by number of sites times number of years of operation. For example, 100 sites for one year is 100 software years. This is an impressive statistic that, coupled with the 80-20 rule, can guide redevelopment of error-prone modules. The 80-20 rule says that 80% of errors are in 20% of modules. If those modules can be identified, they should be redesigned and rewritten. Modules for redevelopment are more easily identified using cleanroom techniques than other techniques. The disadvantages of cleanroom development are similar to those of mathematical proof. Skills required are beyond those of the average business SE, including math, statistics, logic, and formal specification language. We will say more about the 80-20 rule later.

Top-Down Testing

Top-down testing is driven by the principle that the main logic of an application needs more testing and verification than supporting logic. Top-down approaches allow comparison of the application to functional requirements earlier than a bottom-up approach. This means that serious design flaws should surface earlier in the implementation process than with bottom-up testing.

The major drawback to top-down testing is the need for extra code, known as scaffolding, to support the stubs, partial modules, and other pieces of the application for testing. The scaffolding usually begins with job control language and the main logic of the application. The main logic is scaffolded



VDM Overview Process

PAYROLL

ext rd tl:TcLst, rd db:EmpDb, rd al:ActLst, wr el:ErrLst, wr pl:PcLst
pre db /= <> and al /= <> and pl = <> and el = <>
post len tl = len pl + len el and for all i belonging to inds tl.
paid(tl(i),al,db,pl) or error (tl(i),al,db,el)

LEGEND:

VDM Name	Diagram Name
ActLst	Chart of Accounts
EmpDb	Employee File
ErrLst	Error List
PcLst	Payment Details
TcLst	Time Card

Adapted from Conger, Fraser, Galiano, Kumar, McLean, Owen, and Vaishnavi, 1990.

FIGURE 17-9 Vienna Development Method (VDM) Formal Specification Language Example

and tested hierarchically. First, only the critical procedures and control logic are tested.

For example, Figure 17-10 shows the mainline logic for *Customer Maintenance*. The mainline of logic is important because it will be executed every time a maintenance request is performed. Since customer creation is the most probable maintenance activity, it should be guaranteed as much as possible. Further, if creation works, it is easily modified to provide update and delete capabilities which are a subset of creation functionality. Figure 17-11 is an example of COBOL stub logic for *Create Customer*. These two modules would be tested first, before any other logic is verified.

When critical procedures are working, the control language and main line code for less critical procedures are added. In the example above, the stubs for updating, deleting, and querying customers would be tested second. These are tested and retested throughout development as proof that the mainline of logic for all modules works.

After stubs, the most critical logic is coded, unit tested, and placed into integration testing upon completion. In our example, the code for *Create Cus-*

```
Procedure Division.
    Main-Line.
        Display Cust-Maint-menu.
        Accept Cust-Maint-Selection.
        If Cust-Maint-Selection =("A" or F6)
            Call Create-Customer

        else
        If Cust-Maint-Selection =("U" or F7)
            Call Update-Customer

        else
        If Cust-Maint-Selection =("D" or F8)
            Call Delete-Customer


        else
        If Cust-Maint-Selection =("R" or F9)
            Call Query-Customer


        else
        If Cust-Maint-Selection =("E" or F3)
            Go to Cust-Maint-Exit


        else
        Display Selection-Err
        Go To Main-Line.

    Cust-Maint-Exit. Exit.
```

FIGURE 17-10 Mainline Logic for Customer Maintenance

tomer would be tested next. The 'critical' code includes screen data entry and writing to the file. Finally, ancillary logic, such as editing input fields, is completed and placed into testing. In our example, the less critical code is the actual edit and validation processing with error messages. Thus, in top-down testing, the entire application is developed in a skeletal form and tested. As pieces of the skeleton are *fleshed out*, they are added to the test application.

In theory, top-down testing should find critical design errors earlier in the testing process than other approaches. Also, in theory, top-down testing should result in significantly improved quality of delivered software because of the iterative nature of the tests. Unit and integration testing are continuous. There is no discrete integration testing, per se. When the unit/integrated test is complete, further system tests are conducted for volume and constraint tests.

```
Identification Division.
  Program-ID.
       CreateCust.
Environment Division.
Configuration Section.
Source-Computer. IBM.
Object-Computer. IBM.
File Section.
FD   Customer-Screen
     ...
01   Customer-Screen-Record.
     ... screen description
FD   Customer-File
     ...
01   Customer-File-Record.
     ... customer record description
Data Division.
Working-Storage Section.
01   Cust-Screen.
     ...
01   Customer-relation.
     ...
Procedure Division.
Main-Line.
  Perform Display-Cust-Screen.
  Perform Accept-Values.
  Perform Edit-Validate.
  Perform Write-Customer.
  Display Continue-Msg.
  Accept Cust-Response.
  If Cust-Resp = 'y'
       go to main-line
  else
       go to create-customer-exit.
Display-Cust-Screen.
  Write Cust-Screen from Customer-Screen-Record.
DCS-exit. Exit.

Accept-Values.
AV-Exit. Exit.

Edit-Validate.
EV-Exit. Exit.

Write-Customer.
  Write Customer-Relation from Customer-File-Record
    on error perform Cust-Backout-Err.
WC-Exit. Exit.
Create-Customers-Exit. Exit.
```

FIGURE 17-11 COBOL Stub Program for Customer Create

Top-down easily supports testing of screen designs and human interface. In interactive applications, the first logic tested is usually screen navigation. This serves two purposes. First, the logic for interactive processing is exhaustively exercised by the time all code and testing is complete. Second, users can see, at an early stage, how the final

application will look and feel. The users can test the navigation through screens and verify that it matches their work.

Top-down testing can also be used easily with prototyping and iterative development. Prototyping is iterative and follows the same logic for adding code as top-down testing. Presenting prototyping, iterative development, and top-down testing together for user concurrence helps ensure that prototypes actually get completed.

Bottom-Up Testing

Bottom-up testing takes an opposite approach based on the principle that any change to a module can affect its functioning. In bottom-up testing, the entire module should be the unit of test evaluation. All modules are coded and tested individually. A fourth level of testing is frequently added after unit testing to test the functioning of execute units. Then, execute units are turned over to the testing team for integration and systems testing.

The next section discusses the development of test cases to match whatever strategy is defined. Then, each level of testing is discussed in detail with ABC Video test examples to show how to design each test.

Test Cases

Test cases are input data created to demonstrate that both components and the total system satisfy all design requirements. Created data rather than 'live,' production data, is used for the following reasons:

1. Specially developed test data can incorporate all operational situations. This implies that each processing path may be tested at the appropriate level of testing (e.g., unit, integration, etc.).
2. Predetermined test case output should be predicted from created input. Predicting results is easier with created data because it is more orderly and usually has fewer cases.
3. Test case input and output are expanded to form a model database. The database should statistically reflect the users' data in the amount and types of records processed while incorporating as many operational processing paths as possible. The database is then the basis for a regression test database in addition to its use for system testing. Production data is real, so finding statistically representative cases is more difficult than creating them.

Each test case should be developed to verify that specific design requirements, functional design, or code are satisfied. Test cases contain, in addition to test case input data, a forecast of test case output. Real or 'live' data should be used to reality test the modules after the tests using created data are successful.

Each component of an application (e.g., module, subroutine, program, utility, etc.) must be tested with *at least* two test cases: one that works and one that fails. All modules should be deliberately failed at least once to verify their 'graceful degradation.' For instance, if a database update fails, the application should give the user a message, roll back the processing to leave the database as it was before the transaction, and continue processing. If the application were to abend, or worse, continue processing with a corrupt database, the test would have caught an error.

Test cases can be used to test multiple design requirements. For example, a requirement might be that all screens are directly accessible from all other screens; a second requirement might be that each screen contain a standard format; a third requirement might be that all menus be pull-down selections from a menu bar. These three requirements can all be easily verified by a test case for navigation that also attends to format and menu selection method.

The development of test case input may be facilitated by the use of test data generators such as IEBDG (an IBM utility) or the test data generators within some case tools. The analysis and verification of processing may be facilitated by the use of language-specific or environment-specific testing supports (see Figure 17-12). These supports are discussed more completely in the section on automated supports.

COBOL Language Supports:
- Display
- Exhibit
- Ready Trace
- Interactive Trace
- Snap Dump

Focus Language Supports:
- Variable Display
- Transaction Counts
- Online Error Messages
- Online Help

FIGURE 17-12 Examples of Language Testing Supports

To insure that test cases are as comprehensive as possible, a methodical approach to the identification of logic paths or system components is indicated. Matrices, which relate system operation to the functional requirements of the system, are used for this purpose. For example, the matrix approach may be used in

- unit testing to identify the logic paths, logic conditions, data partitions or data boundaries to be tested based on the program specification.
- integration testing to identify the relationships and data requirements among interacting modules.
- system testing to identify the system and user requirements from functional requirements and acceptance criteria.

An example of the matrix approach for an integration test is illustrated as Figure 17-13. The example shows a matrix of program requirements to be met by a suite of modules for *Read Customer File* processing. The test verifies that each module functions independently, that communications between the modules (i.e., the message format, timing, and content) are correct, and verifies that all input and output are processed correctly and within any constraints.

The functional requirements of the *Read Customer File* module are related to test cases in the

	Good Cust-ID	Bad Cust-ID	Missing ID	Retrieve by Name (Good)	Retrieve by Name (Bad)	Good Credit	Bad Credit	Good Data	Bad Data	Call from GetValid Customer (Good)	Call from GetValid Customer (Bad)
1.	X	X	X	X	X					X	X
2.	X					X	X			X	X
3.	X	X	X					X	X	X	X
4.	X	X	X	X	X	X	X	X		X	X

Legend:
1. Read Customer
2. Check Credit
3. Create Customer
4. Display Customer

FIGURE 17-13 Read Customer File Requirements and Test Cases

matrix in Figure 17-13. The 11 requirements can be fully tested in *at most* seven test cases for the four functions.

Matching the Test Level to the Strategy

The goal of the testers is to find a balance between strategies that allows them to prove their application works while minimizing human and computer resource usage for the testing process. No one testing strategy is sufficient to test an application. To use only one testing strategy is dangerous. If only white-box testing is used, testing will consume many human and computer resources and may not identify data sensitive conditions or major logic flaws that transcend individual modules (see Table 17-1). If only black-box testing is used, specific logic problems may remain uncovered even when all specifications are tested; type 2 errors are difficult to uncover. Top-down testing by itself takes somewhat longer than a combined top-down, bottom-up approach. Bottom-up testing by itself does not find strategic errors until too late in the process to fix them without major delays.

In reality, we frequently combine all four strategies in testing an application. White-box testing is used most often for low-level tests—module, routine, subroutine, and program testing. Black-box testing is used most often for high-level tests—integration and system level testing. White-box tests find specific logic errors in code, while black-box tests find errors in the implementation of the functional business specifications. Similarly, top-down tests are conducted for the application with whole tested modules plugged into the control structure as they are ready, that is, after bottom-up development. Once modules are unit tested, they can be integration tested and, sometimes, even system tested with the same test cases.

Table 17-2 summarizes the uses of the box and live-data testing strategies for each level of test. Frequently black- and white-box techniques are combined at the unit level to uncover both data and logic errors. Black-box testing predominates as the level of test is more inclusive. Testing with created data at all levels can be supplemented by testing with live data. Operational, live-data tests ensure that the application can work in the real environment. Next, we examine the ABC rental application to design a strategy and each level of test.

TABLE 17-1 Test Strategy Objectives and Problems

Test Strategy	Method	Goal	Shortcomings
White-Box	Logic	Prove processing.	Functional flaws, data sensitive conditions, and errors across modules are all difficult to test with white-box methods.
Black-Box	Data	Prove results.	Type 2 errors and logic problems difficult to find.
Top-Down	Incremental	Exercise critical code extensively to improve confidence in reliability.	Scaffolding takes time and may be discarded. Constant change may introduce new errors in every test.
Bottom-up	All or nothing	Perfect parts. If parts work, whole should work.	Functional flaws found late and cause delays. Errors across modules may be difficult to trace and find.

TABLE 17-2 Test Level and Test Strategy

Level	General Strategy	Specific Strategy	Comments on Use
Unit	Black-Box	Equivalence Partitioning	Equivalence is difficult to estimate.
		Boundary Value Analysis	Should always be used in edit-validate modules.
		Cause-Effect Graphing	A formal method of boundary analysis that includes tests of compound logic conditions. Can be superimposed on already available graphics, such as state-transition or PDFD.
		Error Guessing	Not a great strategy, but can be useful in anticipating problems.
	Math Proof, Cleanroom	Logic and/or mathematical proof	The best strategies for life-sustaining, embedded, reusable, or other critical modules, but beyond most business SE skills.
	White-Box	Statement Logic	Exhaustive tests of individual statements. Not desirable unless life-sustaining or threatening consequences are possible, or if for reusable module. Useful for 'guessed' error testing that is specific to the operational environment.
		Decision Logic Test	A good alternative to statement logic. May be too detailed for many programs.
		Condition Logic	A good alternative providing all conditions can be documented.
		Multiple Condition Logic	*Desired alternative* for program testing when human resources can be expended.
	Live-Data	Reality Test	Can be useful for timing, performance, and other reality testing *after* other unit tests are successful.
Integration	Black-Box	Equivalence Partitioning	Useful for partitioning by module.
		Boundary Value Analysis	Useful for partitioning by module.
		Cause-Effect Graphing	Useful for application interfaces and partitioning by module.
		Error Guessing	Not the best strategy at this level.

(Table continues on next page)

TABLE 17-2 Test Level and Test Strategy (*Continued*)

Level	General Strategy	Specific Strategy	Comments on Use
Integration	Live-Data	Reality Test	Useful for interface and black-box tests *after* other integration tests are successful.
System/QA-Application Functional Requirements Test	Black-Box	Equivalence Partitioning	Most productive approach to system function testing.
		Boundary Value Analysis	Too detailed to be required at this level. May be used to test correct file usage, checkpoint/restart, or other data-related error recovery.
		Cause-Effect Graphing	Can be useful for intermodule testing and when combined with equivalence partitioning.
	White-Box	Statement Logic	Not a useful system test strategy.
		Decision Logic Test	May be used for critical logic.
		Condition Logic	May be used for critical logic.
		Multiple Condition Logic	May be used for critical logic.
System/QA-Human Interface	Black-Box	Equivalence Partitioning	Useful at the level for screen and associated process and for screen navigation.
		Boundary Value Analysis	Useful at screen level for associated process and screen navigation.
			Useful for QA testing.

TEST PLAN FOR ABC VIDEO ORDER PROCESSING

Test Strategy

Developing a Test Strategy

There are *no rules* for developing a test strategy. Rather, loose heuristics are provided. Testing, like everything else in software engineering, is a skill that comes with practice. Good testers are among the most highly skilled workers on a development team. A career can revolve around testing because skilled testers are in short supply.

As with all other testing projects, the strategy should be designed to prove the application works and that it is stable in its operational environment. While scheduling and time allotted are not most important, when the strategy is devised, one subgoal is to minimize the amount of time and resources (both human and computer) that are devoted to testing.

The first decision is whether and what to test top-down and bottom-up. There are no rules, or even

TABLE 17-2 Test Level and Test Strategy (*Continued*)

Level	General Strategy	Specific Strategy	Comments on Use
System/QA-Human Interface	White-Box	Condition Logic	May be used for critical logic.
		Multiple Condition Logic	May be used for critical logic.
System/QA-Constraints	Black-Box	Equivalence Partitioning	May be useful at the execute unit level.
		Boundary Value Analysis	Should not be required at this level but could be used.
		Cause-Effect Graphing	Might be useful for defining how to measure constraint compliance.
	White-Box	Multiple Condition Logic	Could be used but generally is too detailed at this level of test.
	Live-Data	Reality Test	Useful for black-box type tests of constraints *after* created data tests are successful.
System/QA-Peak Requirements	White-Box	Multiple Condition Logic	May be used for critical logic, but generally too detailed for this level of testing.
	Live-Data	Reality Test	Most useful for peak testing.

heuristics, for this decision. Commitment to top-down testing is as much cultural and philosophical as it is technical. To provide some heuristics, in general, the more critical, the larger, and the more complex an application, the more top-down benefits outweigh bottom-up benefits.

The heuristics of testing are dependent on the language, timing and operational environment of the application. *Significantly* different testing strategies are needed for third (e.g., COBOL, PI/1), fourth (e.g., Focus, SQL), and semantic (e.g., Lisp, PROLOG) languages. Application timing (see Chapter 1) is either batch, on-line, or real-time. Operational environment includes hardware, software, and other co-resident applications. Heuristics for each of these are summarized in Table 17-3.

Package testing differs significantly from self-developed code. More often, when you purchase package software, you are not given the source code or the specifications. You are given user documentation and an executable code. By definition, you have to treat the software as a black box. Further, top-down testing does not make sense because you are presented with a complete, supposedly working, application. Testing should be at the system level only, including functional, volume, intermodular communications, and data-related black-box tests. Next, we consider the ABC test strategy.

ABC Video Test Strategy

The ABC application will be developed using some SQL-based language. SQL is a fourth-generation language which simplifies the testing process and suggests certain testing strategies. The design from Chapter 10, Data-Oriented Design, is used as the basis for testing, although the arguments are the same for the other methodologies.

First, we need to decide the major questions: Who? What? Where? When? How?

TABLE 17-3 Test Strategy Design Heuristics

Condition										
Critical	Y	Y	–	–	N	N	N	N	N	N
Large	Y	–	Y	–	N	N	N	N	N	N
Complex	Y	–	–	Y	N	N	N	N	N	N
Timing	–	–	–	–	BS	BE	BS	BE	BE	–
Language Generation	–	–	–	–	2	2	3/4	3	4	Rule
Test Strategy										
Top-Down/ Bottom-Up, Both, or Either	Both	Both	Either	Either	Either	Either	Cont T Mod B	Either	Both	Cont T Mod B
Black/White/ Both/Either	Both	Both	Cont W Mod Bl	Cont W Mod Bl	Both	Either or Both	Either or Both	Both	Cont W Mod Bl	Bl

Legend:

Y	=	Yes
N	=	No
BS	=	Batch—stand-alone
BE	=	Batch—execute unit
Cont	=	Control Structure
Mod	=	Modules
T	=	Top-down
B	=	Bottom-up
W	=	White
Bl	=	Black

Who? The test coordinator should be a member of the team. Assume it is yourself. Put yourself into this role and think about the remaining questions and how *you* would answer them if you were testing this application.

What? All application functions, constraints, user acceptance criteria, human interface, peak performance, recoverability, and other possible tests must be performed to exercise the system and prove its functioning.

Where? The ABC application should be tested in its operational environment to also test the environment. This means that *all* hardware and software of the operational environment should be installed and tested. If Vic, or the responsible project team member, has not yet installed and tested the equipment,

they are now delaying the conduct of application testing.

When? Since a 4GL is being used, we can begin testing as soon as code is ready. An iterative, top-down approach will be used. This approach allows Vic and his staff early access to familiarize themselves with the application. Testing at the system level needs to include the scaffold code to support top-down testing. The schedule for module coding should identify and schedule all critical modules for early coding. The tasks identified so far are:

1. Build scaffold code and test it.
2. Identify critical modules.
3. Schedule coding of critical modules first.
4. Test and validate modules as developed using the strategy developed.

How? Since a top-down strategy is being used, we should identify critical modules first. Since the application is completely on-line, the screen controls and navigation modules must be developed before anything else can be tested. Also, since the application is being developed specifically to perform rental/return processing, rental/return processing should be the second priority. Rental/return cannot be performed without a customer file and a video file, both of which try to access the respective *create* modules. Therefore, the creation modules for the two files have a high priority.

The priority definition of create and rental/return modules provides a prioritized list for development. The scaffolding should include the test screens, navigation, and stubs for all other processing. The last item, backup and recovery testing, can be parallel to the others.

Next, we want to separate the activities into parallel equivalent chunks for testing. By having parallel testing streams, we can work through the system tests for each parallel stream simultaneously, speeding the testing process. For ABC, *Customer Maintenance*, *Video Maintenance*, *Rental/Return*, and *Periodic* processing can all be treated as stand-alone processes. Notice that in Information Engineering (IE), this independence of processes is at the *activity* level. If we were testing object-oriented design (Chapters 11 and 12), we would look at *processes* from the *Booch diagram* as the independent and parallel test units. If we were testing process design (Chapters 7 and 8), we would use the structure charts to decide parallel sets of processes.

Of the ABC processes, *Rental/Return* is the most complex and is discussed in detail. *Rental/Return* assumes that all files are present, so the DBA must have files defined and populated with data before *Rental/Return* can be tested. Note that even though files must be present, it is neither important nor required that the file maintenance processes be present. For the two create processes that are called, program stubs that return only a new *Customer ID*, or *Video ID /Copy ID*, are sufficient for testing.

In addition to parallel streams of testing, we might also want to further divide *Rental/Return* into several streams of testing by level of complexity, by transaction type, or by equivalent processes to further subdivide the code generation and testing processes. We choose such a division so that the same person can write all of the code but testing can proceed without all variations completed. For example, we will divide *Rental/Return* by transaction type as we did in IE. The four transaction types are rentals with and without returns, and returns with and without rentals. This particular work breakdown allows us to test all major variations of all inputs and outputs, and allows us to proceed from simple to complex as well. In the next sections, we will discuss from bottom-up how testing at each level is designed and conducted using *Rental/Return* as the ABC example.

Next, we define the integration test strategy. The IE design resulted in small modules that are called for execution, some of which are used in more than one process. At the integration level, we define inputs and predict outputs of each module, using a black-box approach. Because SQL calls do not pass data, predicting SQL set output is more important than creating input. An important consideration with the number of modules is that intermodular errors that are created in one module but not evidenced until they are used in another module. The top-down approach should help focus attention on critical modules for this problem.

Because SQL is a declarative language, black-box testing at the unit level is also appropriate. The SQL code that provides the control structure is logic

and becomes an important test item. White-box tests are most appropriate to testing the control logic. Therefore, a mix of black- and white-box testing will be done at the unit level.

To summarize, the top-down strategy for testing the application includes:

1. Test screen design and navigation, including validation of security and access controls.
2. Test the call structure for all modules.
3. Test rental/return processing.
4. Test create processing for customers and videos.
5. Test remaining individual processes and file contents as parallel streams.
6. Test multiple processes and file manipulations together, including validation of response time and peak system performance. The test will use many users doing the same and different processes, simultaneously.
7. Test backup and recovery strategies.

Now, we develop and try a unit test to test the strategy. If a small test of the strategy works, we implement the strategy.

Unit Testing

Guidelines for Developing a Unit Test

Unit tests verify that a specific program, module, or routine (all referred to as 'module' in the remaining discussion) fulfills its requirements as stated in related program and design specifications. The two primary goals of unit testing are conformance to specifications and processing accuracy.

For conformance, unit tests determine the extent to which processing logic satisfies the functions assigned to the module. The logical and operational requirements of each module are taken from the program specifications. Test cases are designed to verify that the module meets the requirements. *The test is designed from the specification, not the code.*

Processing accuracy has three components: input, process, and output. First, each module must process all allowable types of input data in a stable, predictable, and accurate manner. Second, all possible errors should be found and treated according to the specifications. Third, all output should be consistent with results predicted from the specification. Outputs might include hard copy, terminal displays, electronic transmissions, or file contents; all are tested.

There is no one strategy for unit testing. For input/output bound applications, black-box strategies are normally used. For process logic, either or both strategies can be used. In general, the more critical to the organization or the more damaging the possible errors, the more detailed and extensively white-box testing is used. For example, organizationally critical processes might be defined as any process that affects the financial books of the organization, meets legal requirements, or deals with client relationships. Examples of application damage might include life-threatening situations such as in nuclear power plant support systems, life-support systems in hospitals, or test systems for car or plane parts.

Since most business applications combine approaches, an example combining black- and white-box strategies is described here. Using a white-box approach, each program specification is analyzed to identify the distinct logic paths which serve as the basis for unit test design. This analysis is simplified by the use of tables, lists, matrices, diagrams, or decision tables to document the logic paths of the program. Then, the logic paths most critical in performing the functions are selected for white-box testing. Next, to verify that all logic paths not white-box tested are functioning at an acceptable level of accuracy, black-box testing of input and output is designed. This is a common approach that we will apply to ABC Video.

When top-down unit testing is used, control structure logic paths are tested first. When each path is successfully tested, combinations of paths may be tested in increasingly complex relationships until all possible processing combinations are satisfactorily tested. This process of simple-to-complex testing ensures that all logic paths in a module are performing both individually and collectively as intended.

Similarly, unit testing of multiuser applications also uses the simple-to-complex approach. Each program is tested first for single users. Then multiuser tests of the single functions follow. Finally, multiuser tests of multiple functions are performed.

Unit tests of relatively large, complex programs may be facilitated by reducing them to smaller, more manageable equivalent components such as

- transaction type:
  e.g., Debit/Credit, Edit/Update/Report/Error
- functional component activity
  e.g., Preparing, Sending, Receiving, Processing
- decision option
  e.g., If true . . . If false . . .

When the general process of reduction is accomplished, both black-box and white-box approaches are applied to the process of actually defining test cases and their corresponding output. The black-box approach should provide both good and bad data inputs and examine the outputs for correctness of processing. In addition, at least one white-box strategy should be used to test specific critical logic of the tested item.

Test cases should be both exhaustive and minimal. This means that test cases should test every condition or data domain possible but that no extra tests are necessary. For example, the most common errors in data inputs are for edit/validate criteria. Boundary conditions of fields should be tested. Using equivalence partitioning of the sets of allowable values for each field we develop the test for a date formatted YYYYMMDD (that is, 4-digit year, 2-digit month, and 2-digit day). A good year test will test last year, this year, next year, change of century, all zeros, and all nines. A good month test will test zeros, 1, 2, 4 (representative of months with 30 days), 12, and 13. Only 1 and 12 are required for the boundary month test, but the other months are required to test day boundaries. A good day test will test zeros, 1, 28, 29, 30, 31, and 32, depending on the final day of each month. Only one test for zero and one are required, based on the assumption that if one month processes correctly, all months will. Leap year and nonleap years should also be tested. An example of test cases for these date criteria is presented. Figure 17-14 shows the equivalent sets of data for each domain. Table 17-4 lists exhaustive test

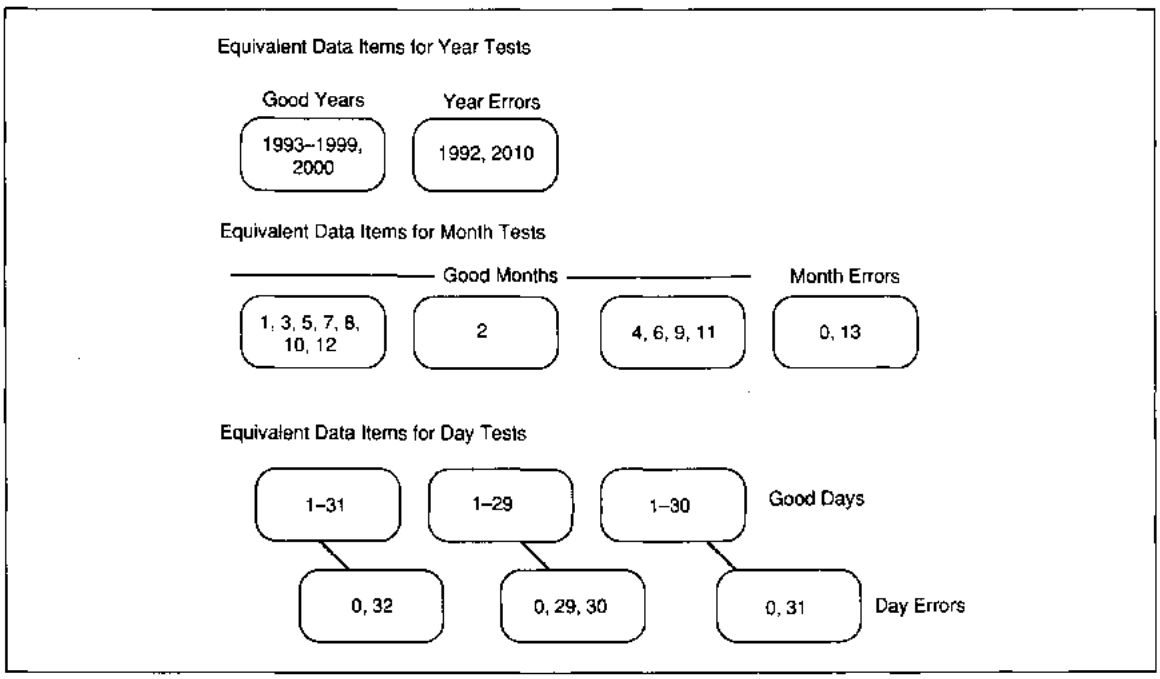


FIGURE 17-14 Unit Test Equivalent Sets for a Date

TABLE 17-4 Exhaustive Set of Unit Test Cases for a Date

Test Case	YYYY	MM	DD	Comments
1	aaaa	aa	aa	Tests actions against garbage input
2	1992*	0	0	Tests all incorrect lower bounds
3	2010	13	32	Tests all incorrect upper bounds
4	1993	1	31	Tests correct upper day bound
4a	1994	12	31	Not required . . . could be optional test of upper month/day bound. Assumption is that if month = 1 works, all valid, equivalent months will work.
5	1995	1	1	Tests correct lower day bound
6	1996	12	1	Not required . . . could be optional test of upper month/lower day bound. Assumption is that if month = 1 works, all valid, equivalent months will work.
7	1997	1	32	Tests upper day bound error
8	1998	12	32	Not required . . . could be optional test of upper month/upper day bound error. Assumption is that if month = 1 works, all valid, equivalent months will work.
9	1999	12	0	Retests lower bound day error with otherwise valid data . . . Not strictly necessary but could be used.
10	2000	2	1	Tests lower bound . . . not strictly necessary
11	2000	2	29	Tests leap year upper bound
12	2000	2	30	Tests leap year upper bound error
13	1999	2	28	Tests nonleap year upper bound
14	1999	2	29	Tests nonleap year upper bound error
15	1999	2	0	Tests lower bound error . . . not strictly necessary
16	2001	4	30	Tests upper bound
17	2001	4	31	Tests upper bound error
18	2002	4	1	Tests lower bound . . . not strictly necessary
19	2003	4	0	Tests lower bound error . . . not strictly necessary

*Valid dates are between 1/1/93 and 12/31/2009.

TABLE 17-5 Minimal Set of Unit Test Cases for a Date

Test Case	YYYY	MM	DD	Comments
1	aaaa	aa	aa	Tests actions against garbage input
2	1992	0	0	Tests all incorrect lower bounds
3	2010	13	32	Tests all incorrect upper bounds
4	1993	1	31	Tests correct upper day bound
5	1995	1	1	Tests correct lower day bound
6	1997	1	32	Tests upper day bound error
7(9)	2000	2	29	Tests leap year upper bound
8(10)	2000	2	30	Tests leap year upper bound error
9(11)	1999	2	28	Tests nonleap year upper bound
10(12)	1999	2	29	Tests nonleap year upper bound error
11(14)	2001	4	30	Tests upper bound
12(15)	2001	4	31	Tests upper bound error

cases for each set in the figure. Table 17-5 lists the reduced set after extra tests are removed.

Other frequently executed tests are for character, field, batch, and control field checks. Table 17-6 lists a sampling of errors found during unit tests. Character checks include tests for blanks, signs, length, and data types (e.g., numeric, alpha, or other). Field checks include sequence, reasonableness, consistency, range of values, or specific contents. Control fields are most common in batch applications and are used to verify that the file being used is the correct one and that all records have been processed. Usually the control field includes the last execution date and file name which are both checked for accuracy. Record counts are only necessary when not using a declarative language.

Once all test cases are defined, tests are run and results are compared to the predictions. Any result that does not *exactly match* the prediction must be reconciled. The only possible choices are that the tested item is in error or the prediction is in error. If the tested item is in error, it is fixed and retested. Retests should follow the approach used in the first tests. If the prediction is in error, the prediction is researched and corrected so that specifications are accurate and documentation shows the correct predictions.

Unit tests are conducted and reviewed by the author of the code item being tested, with final test results approved by the project test coordinator.

How do you know when to stop unit testing? While there is no simple answer to this question, there are practical guidelines. When testing, each tester should keep track of the number of errors found (and resolved) in each test. The errors should be plotted by test shot to show the pattern. A typical module test curve is skewed left with a decreasing number of errors found in each test (see Figure 17-15). When the number of errors found approaches zero, or when the slope is negative and approaching zero, the module can be moved forward to the next level of testing. If the number of errors found stays constant or increases, you should seek help either in interpreting the specifications or in testing the program.

ABC Video Unit Test

Above, we said we would use a combination of black- and white-box testing for ABC unit tests. The

TABLE 17-6 Sample Unit Test Errors

Edit/Validate	
	Transaction rejected when valid
	Error accepted as valid
	Incorrect validation criteria applied
Screen	
	Navigation faulty
	Faulty screen layout
	Spelling errors on screen
	Inability to call screen
Data Integrity	
	Transaction processed when inconsistent with other information
	Interfile matching not correct
	File sequence checking not correct
File Processing	
	File, segment, relation of field not correctly processed
	Read/write data format error
	Syntax incorrect but processed by interpreter
Report	
	Format not correct
	Totals do not add/crossfoot
	Wrong field(s) printed
	Wrong heading, footing or other cosmetic error
	Data processing incorrect

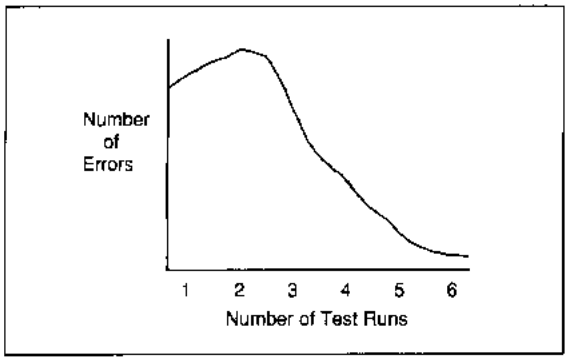


FIGURE 17-15 Unit Test Errors Found Over Test Shots

application is being implemented using a SQL software package. Therefore, all code is assumed to be in SQL. The control logic and non*SELECT* code is subject to white-box tests, while the *SELECT* modules will be subject to black-box tests.

In Chapter 10, we defined *Rent/Return* processing as an execute unit with many independent code units. Figure 17-16 shows partial SQL code from two *Rent/Return* modules. Notice that most of the code is defining data and establishing screen addressability. As soon as two or three modules that have such strikingly similar characteristics are built, the need to further consolidate the design to accommodate the implementation language should be obvious. With the current design, more code is spent in overhead tasks than in application tasks. Overhead code means that users will have long wait times while the system changes modules. The current design also means that debugging the individual modules would require considerable work to verify that the modules performs collectively as expected. Memory locations would need to be printed many times in such testing.

To restructure the code, we examine what all of the *Rent/Return* modules have in common—*Open Rentals* data. We can redefine the data in terms of *Open Rentals* with a single-user view used for all *Rent/Return* processing. This simplifies the data part of the processing but increases the vulnerability of the data to integrity problems. Problems might increase because the global view of data violates the principle of information hiding. The risk must be taken, however, to accommodate reasonable user response time.

The common format of the restructured SQL code is shown in Figure 17-17. In the restructured version, data is defined once at the beginning of *Rent/Return* processing. The cursor name is declared once and the data is retrieved into memory based on the data entered through the *Get Request* module. The remaining *Rent/Return* modules are called in sequence. The modules have a similar structure for handling memory addressing. The problems with many prints of memory are reduced because once the data is brought into memory, no more retrievals are necessary until updates take place at the end of the transaction. Processing is simplified by unifying the application's view of the data.

```
UPDATE OPEN RENTAL FILE  (BOLDFACE CODE IS REDUNDANT)
        DCL     INPUT_VIDEO_ID          CHAR(8);
        DCL     INPUT_COPY_ID           CHAR(2);
        DCL     INPUT_CUST_ID           CHAR (9);
        DCL     AMT_PAID                DECIMAL (4,2);
        DCL     CUST_ID                 CHAR(9);
        ...
          CONTINUE UNTIL ALL FIELDS USED ON THE SCREEN OR USED TO
        CONTROL SCREEN PROCESSING ARE DECLARED ...
        DCL     TOTAL_AMT_DUE           DECIMAL(5,2);
        DCL     CHANGE                  DECIMAL(4,2);
        DCL     MORE_OPEN_RENTALS       BIT(1);
        DCL     MORE_NEW_RENTALS        BIT(1);
        EXEC SQL INCLUDE SQLCA; /*COMMUNICATION AREA*/
        EXEC SQL DECLARE CUSTOMER TABLE
                (FIELD DEFINITIONS FOR CUSTOMER RELATION);
        EXEC SQL DECLARE VIDEO TABLE
                (FIELD DEFINITIONS FOR VIDEO RELATION);
        EXEC SQL DECLARE COPY TABLE
                (FIELD DEFINITIONS FOR COPY RELATION);
        EXEC SQL DECLARE OPENRENTAL TABLE
                (FIELD DEFINITIONS FOR OPENRENTAL RELATION);
        EXEC SQL DECLARE SCREEN_CURSOR CURSOR FOR UPDATE OF
                ORVIDEOID
                ORCOPYID
                ORCUSTID
                ORRENTALDATE;
        ORDER BY ORCUSTID, ORVIDEOID, ORCOPYID;
        EXEC SQL OPEN SCREEN, CURSOR;
        GOTOLABEL
        EXEC SQL FETCH SCREEN CURSOR INTO TARGET
                :CUSTID
                :VIDEOID
                :COPYID
                :RENTALDATE
        IF SQLCODE = 100  GOTO GOTOEXIT;
        EXEC SQL UPDATE OPENRENTAL
                SET ORCUSTID = CUSTID
                SET ORVIDEOID = VIDEOID
                SET ORCOPYID = COPYID
                SET ORRENTALDATE = TODAYSDATE
        WHERE CURRENT OF SCREEN  CURSOR;
        GOTO GOTOLABEL;
        GOTOEXIT;
        EXEC SQL CLOSE SCREEN_CURSOR;
```

(Figure continues on next page)

FIGURE 17-16 Two Modules Sample Code

```
ADD RETURN DATE (Boldface code is redundant)
        DCL     INPUT_VIDEO_ID          CHAR(8);
        DCL     INPUT_COPY_ID           CHAR(2);
        DCL     INPUT_CUST_ID           CHAR (9);
        DCL     AMT_PAID                DECIMAL (4,2);
        DCL     CUST_ID                 CHAR(9);
        ...
          CONTINUE UNTIL ALL FIELDS USED ON THE SCREEN OR USED TO
        CONTROL SCREEN PROCESSING ARE DECLARED ...
        DCL     TOTAL_AMT_DUE           DECIMAL(5,2);
        DCL     CHANGE                  DECIMAL(4,2);
        DCL     MORE_OPEN_RENTALS       BIT(1);
        DCL     MORE_NEW_RENTALS        BIT(1);
        EXEC SQL INCLUDE SQLCA: /*COMMUNICATION AREA*/
        EXEC SQL DECLARE CUSTOMER TABLE
                (FIELD DEFINITIONS FOR CUSTOMER RELATION);
        EXEC SQL DECLARE VIDEO TABLE
                (FIELD DEFINITIONS FOR VIDEO RELATION);
        EXEC SQL DECLARE COPY TABLE
                (FIELD DEFINITIONS FOR COPY RELATION);
        EXEC SQL DECLARE OPENRENTAL TABLE
                (FIELD DEFINITIONS FOR OPENRENTAL RELATION);
        EXEC SQL DECLARE SCREEN_CURSOR CURSOR FOR
                SELECT * FROM OPEN_RENTAL
                        WHERE VIDEOID = ORVIDEOID
                        AND COPYID = ORCOPYID;
        EXEC SQL OPEN SCREEN_CURSOR
        GOTOLABEL
        EXEC SQL FETCH SCREEN CURSOR INTO TARGET
                :CUSTID
                :VIDEOID
                :COPYID
                :RENTALDATE
        IF SQLCODE = 100  GOTO GOTOEXIT;
        EXEC SQL SET :RETURNDATE = TODAYS_DATE
                WHERE CURRENT OF SCREEN_CURSOR;
        EXEC SQL UPDATE OPEN_RENTAL
                SET ORRETURNDATE = TODAYS_DATE
                WHERE CURRENT OF SCREEN_CURSOR;
        GOTO GOTOLABEL;
        GOTOEXIT;
        EXEC SQL CLOSE SCREEN_CURSOR;
```

FIGURE 17-16 Two Modules Sample Code (*Continued*)

The restructuring now requires a change to the testing strategy for *Rent/Return*. A strictly top-down approach cannot work because the *Rent/Return* modules are no longer independent. Rather, a combined top-down and bottom-up approach is warranted. A sequential bottom-up approach is more effective for the functional *Rent/Return* processing. Top-down, black-box tests of the *SELECT* code are done before being embedded in the execute unit. Black-box testing for the *SELECT* is used because SQL controls all data input and output. Complete *SELECT* statements are the test unit.

```
DCL     INPUT_VIDEO_ID          CHAR(8);
DCL     INPUT_COPY_ID           CHAR(2);
DCL     INPUT_CUST_ID           CHAR (9);
DCL     AMT_PAID                DECIMAL (4,2);
DCL     CUST_ID                 CHAR(9);
 ...
  continue until all fields used on the screen or used to control screen processing are
declared ...


DCL     TOTAL_AMT_DUE           DECIMAL(5,2);
DCL     CHANGE                  DECIMAL(4,2);
DCL     MORE_OPEN_RENTALS       BIT(1);
DCL     MORE_NEW_RENTALS        BIT(1);
EXEC SQL INCLUDE SQLCA: /*COMMUNICATION AREA*/
EXEC SQL DECLARE  RENTRETURN TABLE
(field definitions for user view including all fields from customer, video, copy,
open rental, and customer history relations);


EXEC SQL DECLARE SCREEN_CURSOR CURSOR FOR
        SELECT * from rentreturn
        where (:videoid = orvideo_id  and :copyid = orcopyid)
        or :custid = orcustid)
EXEC SQL OPEN SCREEN_CURSOR
EXEC SQL FETCH SCREEN_CURSOR INTO TARGET
        :Request
If :request eq "C?" set :custid = :request
else    set :videoid = :request1
        set :copyid = :request2;


        (At this point the memory contains the related relation data
        and the remaining rent/return processing can be done.)

All the other modules are called and contain the following common format:

GOTOLABEL
EXEC SQL FETCH SCREEN_CURSOR INTO TARGET
:screen fields


IF SQLCODE = 0  next step;   (return code of zero means no errors)
IF SQLCODE = 100 (not found condition) CREATE DATA or CALL END PROCESS;
IF SQLCODE < 0 CALL ERROR_PROCESS, ERROR-TYPE;
Set screen variables (which displays new data)
Prompt next action


GOTO GOTOLABEL;
GOTOEXIT;
EXEC SQL CLOSE SCREEN_CURSOR;
```

FIGURE 17-17 Restructured SQL Code—Common Format

Test	Type
1. Test SQL SELECT statement	Black Box
2. Verify SQL cursor and data addressibility	White Box
3. Test *Get Request*	White Box
4. Test *Get Valid Customer, Get Open Rentals*	Black Box for embedded SELECT statement, White Box for other logic
5. Test *Get Valid Video*	White Box for logic, Black Box for embedded SELECT statement
6. Test *Process Payment and Make Change*	White Box
7. Test *Update Open Rental*	Black Box for Update, White Box for other logic
8. Test *Create Open Rental*	Black Box for Update, White Box for other logic
9. Test *Update Item*	Black Box for Update, White Box for other logic
10. Test *Update/Create Customer History*	Black Box for Update, White Box for other logic
11. Test *Print Receipt*	Black Box for Update, White Box for other logic

FIGURE 17-18 Unit Test Strategy

The screen interaction and module logic can be tested as either white box or black box. At the unit level, white-box testing will be used to test inter-module control logic. A combination of white-box and black-box testing should be used to test intra-module control and process logic.

The strategy for unit testing, then, is to test data retrievals first, to verify screen processing, including SQL cursor and data addressability second, and to sequentially test all remaining code last (see Figure 17-18).

Because all processing in the ABC application is on-line, an interactive dialogue *test script* is developed. All file interactions predict data retrieved and written, as appropriate. The individual unit test scripts begin processing at the execute unit boundary. This means that menus are not necessarily tested. A test script has three columns of information developed. The first column shows the computer messages or prompts displayed on the screen. The second column shows data entered by the user. The third column shows comments or explanations of the interactions taking place.

A partial test script for Rent/Return processing is shown in Figure 17-19. The example shows the script for a return with rental transaction. Notice that the test begins at the *Rent/Return* screen and that both error and correct data are entered for each field. After all errors are detected and dispatched properly, only correct data is required. This script shows one of the four types of transactions. It shows only one return and one rental, however, and should be expanded in another transaction to do several rentals and several returns; returns should include on-time and late videos and should not include all tapes checked out. This type of transaction represents the requisite variety to test returns with rentals. Of course, other test scripts for the other three types of transactions should also be developed. This is left as an extra-credit activity.

Subsystem or Integration Testing

Guidelines for Integration Testing

The purpose of integration testing is to verify that groups of interacting modules that comprise an execute unit perform in a stable, predictable, and accu-

System Prompt	User Action	Explanation
Menu	Press mouse, move to Rent/Return, and release	Select Rent/Return from menu
Rent/Return screen, cursor at request field	Scan customer bar code 1234567	Dummy bar code
Error Message 1: Illegal Customer or Video Code, Type Request	Enter: 1234567 <cr>	Dummy bar code
Customer Data Entry Screen with message: Illegal Customer ID, enter new customer	<cr>	Carriage return entered to end Create Customer process
Rent/Return screen, cursor at request field	Scan customer bar code 2221234	Legal customer ID. System should return customer and rental information for M. A. Jones, Video 12312312, Copy 3, Terminator 2, Rental date 1/23/94, not returned.
Cursor at request field	Scan 123123123	Cursor moves to rented video line
Cursor at return date field	Enter yesterday's date	Error message: Return date must be today's date.
Cursor at return date field	Enter today's date	Late fee computed and displayed . . . should be $4.00.
Cursor at request field	Scan new tape ID—123412345	New tape entered and displayed. Video #12341234, Copy 5, Mary Poppins, Rental date 1/25/94, Charge $2.00.
Cursor at request field	Press <cr>	System computes and displays *Total Amount Due* . . . should be $6.00.
Cursor at Total Amount Paid field	Enter <cr>	Error Message: Amount paid must be numeric and equal or greater than *Total Amount Due*.
Cursor at Total Amount Paid field	Enter 10 <cr>	System computes and displays *Change Due* . . . should be $4.00. Cash drawer should open.
Cursor at Request field	Enter <cr>	Error Message: You must enter P or F5 to request print.
Cursor at Request field	Enter P <cr>	System prints transaction

Go to SQL Query and verify Open Rental and Copy contents

Open Rental tuple for Video 123123123 contents should be:

22212341231231230123940200012594040000000000000

Open Rental tuple for Video 123412345 should be:

222123412341234501259402000000000000000000000000

Copy tuple for Video 12312312, Copy 3 should be:

12312312311019200103

Copy tuple for Video 12341234, Copy 5 should be:

12341234511319010000

Verify the contents of the receipt.

FIGURE 17-19 ABC Video Unit Test Example—Rent/Return

rate manner that is consistent with all related program and systems design specifications.

Integration tests are considered distinct from unit tests. That is, as unit tests are successful, integration testing for the tested units can begin. The two primary goals of integration testing are compatibility and intermodule processing accuracy.

Compatibility relates to calling of modules an operational environment. The test verifies first that all modules are called correctly, and, even with errors, do not cause abends. Intermodule tests check that data transfers between modules operate as intended within constraints of CPU time, memory, and response time. Data transfers tested include sorted and extracted data provided by utility programs, as well as data provided by other application modules.

Test cases developed for integration testing should be sufficiently exhaustive to test all possible interactions and may include a subset of unit test cases as well as special test cases used only in this test. The integration test does *not* test logic paths within the modules as the unit test does. Instead, it tests interactions between modules only. Thus, a black-box strategy works well in integration testing.

If modules are called in a sequence, checking of inputs and outputs to each module simplifies the identification of computational and data transfer errors. Special care must be taken to identify the source of errors, not just the location of bad data. Frequently, in complex applications, errors may not be apparent until several modules have touched the data and the true source of problems can be difficult to locate. Representative integration test errors are listed in Table 17-7.

TABLE 17-7 Sample Integration Test Errors

Intermodule communication
Called module cannot be invoked
Calling module does not invoke all expected modules
Message passed to module contains extraneous information
Message passed to module does not contain correct information
Message passed contains wrong (or inconsistent) data type
Return of processing from called module is to the wrong place
Module has no return
Multiple entry points in a single module
Multiple exit points in a single module
Process errors
Input errors not properly disposed
Abend on bad data instead of graceful degradation
Output does not match predicted results
Processing of called module produces unexpected results does not match prediction
Time constrained process is over the limit
Module causes time-out in some other part of the application

Integration testing can begin as soon as two or more modules are successfully unit tested. When to end integration tests is more subjective. When exceptions are detected, the results of all other test processing become suspect. Depending on the severity and criticality of the errors to overall process integrity, all previous levels of testing might be reexecuted to reverify processing. Changes in one module may cause tests of other modules to become invalid. Therefore, integration tests should be considered successful only when the entire group of modules in an execute unit are run *individually and collectively* without error. Integration test curves usually start low, increase and peak, then decrease (see Figure 17-20). If there is pressure to terminate integration testing before all errors are found, the rule of thumb is to continue testing until fewer errors are found on several successive test runs.

ABC Video Integration Test

Because of the redesign of execute units for more efficient SQL processing, integration testing can be concurrent with unit code and test work, and should

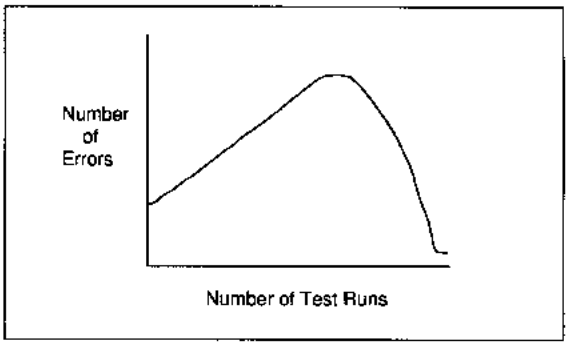


FIGURE 17-20 Integration Test Errors Found Over Test Shots

integrate and test the unit functions as they are complete. The application control structure for screen processing and for calling modules is the focus of the test.

Black-box, top-down testing is used for the integration test. Because SQL does not pass data as input, we predict the sets that SQL will generate during *SELECT* processing. The output sets are then passed to the control code and used for screen processing, both of which have been unit tested and should work. To verify the unit tests at the integration level, we should:

1. Ensure that the screen control structure works and that execute units are invoked as intended.
2. Ensure that screens contain expected data from *SELECT* processing.
3. Ensure that files contain all updates and created records as expected.
4. Ensure that printed output contains expected information in the correct format.

First, we want to define equivalent sets of processes and the sets' equivalent sets of data inputs. For instance, the high level processes from IE analysis constitute approximately equivalent sets. These were translated into modules during design and, with the exception of integrating data access and use across modules, have not changed. These processes include *Rent/Return*, *Customer Maintenance*, *Video Maintenance*, and *Other* processing. If the personnel are available, four people could be assigned to develop one script each for these equivalent sets of processing. Since we named *Rent/Return* as the highest priority for development, its test should be developed first. The others can follow in any order, although the start-up and shutdown scripts should be developed soon after *Rent/Return* to allow many tests of the entire interface.

First, we test screen process control, then individual screens. Since security and access control are embedded in the screen access structure, this test should be white box and test every possible access path, including invalid ones. Each type of access rights and screen processing should be tested. For the individual screens, spelling, positioning, color, highlighting, message placement, consistency of design, and accuracy of information are all validated (see Figure 17-21).

The integration test example in Figure 17-22 is the script for testing the start-up procedure and security access control for the application. This script would be repeated for each valid and invalid user including the other clerks and accountant. The start-up should only work for Vic, the temporary test account, and the chief clerk. The account numbers that work should *not* be documented in the test

1. Define equivalent sets of processes and data inputs.
2. Define the priorities of equivalent sets for testing.
3. Develop test scrips for *Rent/Return*, *Other* processing, *Customer Maintenance*, *Video Maintenance*.
4. For each of the above scripts, the testing will proceed as follows:
   a. Test screen control, including security of access to the *Rent/Return* application.
   b. Evaluate accuracy of spelling, format, and consistency of each individual screen.
   c. Test access rights and screen access controls.
   d. Test information retrieval and display.
   e. For each transaction, test processing sequence, dialogue, error messages, and error processing.
   f. Review all reports and file contents for accuracy of processing, consistency, format, and spelling.

FIGURE 17-21 ABC Integration Test Plan

Test Startup Security

System Prompt	User Action	Explanation
C:>	StRent<cr>	StRent is Exec to startup the Rental/Return Processing application
Enter password	<cr>	Error
Password must be alphanumeric and six characters.		
Enter Password	123456<cr>	Error—illegal password
Password illegal, try again.		
Enter Password.	Abcdefg	Error—illegal password
Three illegal attempts at password. System shutdown		
C:>	StRent<cr>	Error—3 illegal attempts requires special start-up.
Illegal start-up attempt		
System begins to beep continuously until stopped by system administrator. No further prompts.		

Single User Sign-on

C:>	StRent<cr>	StRent is Exec to startup the Rental/Return Processing application
Enter Password:	<cr>	Error
Password illegal, try again.		
Enter Password:	VAC5283	Temporary legal entry
User Sign-on menu		
Enter Initials:	<cr>	Error
You must enter your initials.		
Enter Initials:	VAV	Error
Initials not authorized, try again.		
Enter Initials:	VAC	Legal entry (VAC is Vic)
Main Menu with all Options	Begin Main Menu Test.	

FIGURE 17-22 ABC Video Integration Test Script

script. Rather, a note should refer the reader to the person responsible for maintaining passwords.

In the integration portion of the test, multiuser processing might take place, but it is not necessarily fully tested at this point. File contents are verified after each transaction is entered to ensure that file updates and additions are correct. If the integration test is approached as iteratively adding modules for testing, the final run-through of the test script should include all functions of the application, including

start-up, shutdown, generation and printing of all reports, queries on all files, all file maintenance, and all transaction types. At least several days and one monthly cycle of processing should be simulated for ABC's test to ensure that end-of-day and end-of-month processing work.

Next, we discuss system testing and continue the example from ABC with a functional test that is equally appropriate at the integration, system, or QA levels.

System and Quality Assurance Testing

Guidelines for Developing System and Quality Assurance Tests

The system test is used to demonstrate an application's ability to operate satisfactorily in a simulated production environment using its intended hardware and software configuration. The quality assurance test (QA) is both a system test and a documentation test. Both tests also verify that all of the system's interacting modules do the following:

1. Fulfill the user's functional requirements as contained in the business system design specifications and as translated into design requirements in the design spec and any documents controlling interfaces to other systems.
2. The human interface works as intended. Screen design, navigation, and work interruptability are the test objects for human interface testing. All words on screens should be spelled properly. All screens should share a common format that is presented consistently throughout the application. This format includes the assignment of program function keys as well as the physical screen format. Navigation is the movement between screens. All menu selections should bring up the correct next screen. All screens should return to a location designated somewhere on the screen. If direct navigation from one screen to any other is provided, the syntax for that movement should be consistent and correct. If transactions are to be interruptible, the manner of saving partial transactions and calling them back should be the same for all screens. System level testing should test all of these capabilities.
3. All processing is within constraints. General constraints can relate to prerequisites, postrequisites, time, structure, control and inferences (see Chapter 1). Constraints can be internally controlled by the application or can be externally determined with the application simply meeting the constraint. Internally controlled constraints are tested through test cases specifically designed for that purpose. For instance, if response time limits have been stated, the longest possible transaction with the most possible errors or other delays should be designed to test response. If response time for a certain number of users is limited, then the test must have all users doing the most complex of actions to prove the response time constraint is met. Externally controlled constraints are those that the application either meets or does not. If the constraints are not met, then some redesign is probably required.
4. All modules are compatible and, in event of failures, degrade gracefully. System tests of compatibility prove that all system components are capable of operating together as designed. System components include programs, modules, utilities, hardware, database, network, and other specialized software.
5. Has sufficient procedures and code to provide disaster, restart, and application error recovery in both the designed and host software (e.g., DB2)
6. All operations procedures for the system are useful and complete. Operations procedures include start-up, shutdown, normal processing, exception processing, special operator interventions, periodic processing, system specific errors, and the three types of recovery.

In addition, the QA test evaluates the accuracy, consistency, format, and content of application

documentation, including technical, user, on-line, and operations documentation. Ideally, the individual performing the QA test does not work on the project team but can deal with them effectively in the adversarial role of QA. Quality assurance in some companies is called the acceptance test and is performed by the user. In other companies, QA is performed within the IS department and precedes the user acceptance test.

The system test is the final developmental test under the control of the project team and is considered distinct from integration tests. That is, the successful completion of integration testing of successively larger groups of programs eventually leads to a test of the entire system. The system test is conducted by the project team and is analogous to the quality assurance acceptance test which is conducted by the user (or an agent of the user). Sample system test errors are shown in Table 17-8.

Test cases used in both QA and system testing should include as many normal operating conditions as possible. System test cases may include subsets of all previous test cases created for unit and integration tests as well as global test cases for system level requirements. The combined effect of test data used should be to verify all major logic paths (for both normal and exception processing), protection mechanisms, and audit trails.

QA tests are developed completely from analysis and design documentation. The goal of the test is to verify that the system does what the documentation describes and that all documents, screens, and processing are consistent. Therefore, QA tests go beyond system testing by specifically evaluating application information consistency across environments in addition to testing functional software accuracy. QA tests find a broader range of errors than system tests; a sampling of QA errors is in Table 17-9.

System testing affords the first opportunity to observe the system's hardware components operating as they would in a production mode. This enables the project's test coordinator to verify that response time and performance requirements are satisfied.

Since system testing is used to check the entire system, any errors detected and corrected may

TABLE 17-8 Sample System Test Errors

Functional
Application does not perform a function in the functional specification
Application does not meet all functional acceptance criteria
Human Interface
Screen format, spelling, content errors
Navigation does not meet user requirements
Interruption of transaction processing does not meet user requirements
Constraints
Prerequisites treated as sequential and should be parallel . . . must all be checked by (x) module
Prerequisite not checked
Response Time/Peak Performance
Response time not within requirements for file updates, start-up, shutdown, query, etc.
Volume of transactions expected cannot be processed within the specified run-time intervals
Batch processing cannot be completed in the time allotted
Expected number of peak users cannot be accommodated
Restart/Recovery
Program—Interrupted printout fails to restart at the point of failure (necessary for check processing and some confidential/financial reporting)
Software—Checkpoint/restart routine is not called properly
Hardware—Printer cannot be accessed from main terminal
Switches incorrectly set
System re-IPL called for in procedures cannot be done without impacting other users not of this application
Expected hardware configuration has incompatible components

TABLE 17-9 Sample QA/Acceptance Test Errors

Documentation
Two or more documents inconsistent
Document does not accurately reflect system feature
Edit/Validate
Invalid transaction accepted Valid transaction rejected
Screen
Navigation, format, content, processing inconsistent with functional specification
Data Integrity
Multifile, multitransaction, multimatches are incorrect
File
File create, update, delete, query not present or not working
Sequence, data, or other criteria for processing not checked
Report specification
Navigation, format, content, processing inconsistent with functional
Recovery
Printer, storage, memory, software, or application recovery not correct
Performance
Process, response, user, peak, or other performance criteria not met
User Procedures
Do not match processing
Incomplete, inconsistent, incomprehensible
On-line help differs from paper documents
Operations Procedures
Do not match processing
Incomplete, inconsistent, incomprehensible

require retesting of previously tested items. The system test, therefore, is considered successful only when the entire system runs without error for all test types.

The test design should include all possible legal and illegal transactions, good and bad data in transactions, and enough volume to measure response time and peak transaction processing performance. As the test proceeds, each person notes on the test script whether an item worked or not. If a tested interaction had unexpected results, the result obtained is marked in the margin and noted for review.

The first step is to list all actions, functions, and transactions to be tested. The information for this list is developed from the analysis document for all required functions in the application and from the design document for security, audit, backup, and interface designs.

The second step is to design transactions to test all actions, functions and transactions. Third, the transactions are developed into a test script for a single user as a general test of system functioning. This test proves that the system works for one user and all transactions. Fourth, the transactions are interleaved across the participating number of users for multiuser testing. In general, the required transactions are only a subset of the total transactions included in the multiuser test. Required transactions test the variations of processing and should be specifically designed to provide for exhaustive transaction coverage. The other transactions can be a mix of simple and complex transactions at the designer's discretion. If wanted, the same transaction with variations to allow multiple use can be used. Fifth, test scripts for each user are then developed. Last, the test is conducted. These steps in developing system/QA tests are summarized as follows:

1. List all actions, functions, and transactions to be tested.
2. Design transactions to test all actions, functions, and transactions.
3. Develop a single-user test script for above.
4. Interleave the tests across the users participating in the test to fully test multiuser functioning of the application.
5. Develop test scripts for each user.

6. Conduct the test.
7. Review test results and reconcile anomalous findings.

Designing multiuser test scripts is a tedious and lengthy process. Doing multiuser tests is equally time-consuming. Batch test simulator (BTS) software is an on-line test aid available in mainframe environments. BTSs generate data transactions based on designer-specified attribute domain characteristics. Some BTSs can read data dictionaries and can directly generate transactions. The simulation portion of the software executes the interactive programs using the automatically generated transactions and can, in seconds, perform a test that might take people several hours. BTSs are not generally available on PCs or LANs yet, but they should be in the future.

Finally, after the system and QA tests are successful, the minimal set of transactions to test the application are compiled into test scripts for a regression test package. A **regression test package** is a set of tests that is executed every time a change is made to the application. The purpose of the regression test is to ensure that the changes do not cause the application to *regress* to a nonfunctional state, that is, that the changes do not introduce errors into the processing.

Deciding when to stop system testing is as subjective as the same decision for other tests. Unlike module and integration tests, system tests might have several peaks in the number of errors found over time (see Figure 17-23). Each peak might represent new modules or subsystems introduced for testing or might demonstrate application regression due to fixes of old errors that cause new errors. Because of this multipeak phenomenon, system testing is the most difficult to decide to end. If a decreasing number of errors have not begun to be found, that is, the curve is still rising, do not stop testing. If all modules have been through the system test at least once, and the curve is moving toward zero, then testing can be stopped if the absolute number of errors is acceptable. Testing should continue with a high number of errors regardless of the slope of the line. What constitutes an acceptable number of errors, however, is decided by the project manager, user, and IS managers; there is no right number.

QA testing is considered complete when the errors do not interfere with application functioning. A complete list of errors to be fixed is developed and given to the project manager and his or her manager to track. In addition, a QA test report is developed to summarize the severity and types of errors found over the testing cycle. Errors that are corrected before the QA test completes are noted as such in the report.

The QA report is useful for several purposes. The report gives feedback to the project manager about the efficacy of the team-testing effort and can identify weaknesses that need correcting. The reports are useful for management to gain confidence (or lose it) in project managers and testing groups. Projects that reach the QA stage and are then stalled for several months because of errors identify training needs that might not otherwise surface.

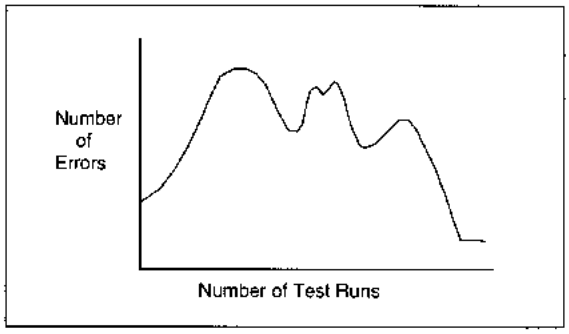


FIGURE 17-23 System Test Errors Found Over Test Shots

ABC Video System Test

Because ABC's application is completely on-line, the system test is essentially a repeat of the integration test for much of the functional testing. The system test, in addition, evaluates response time, audit, recovery, security, and multiuser processing. The functional tests do not duplicate the integration test exactly, however. The first user might use the integration test scripts. Other user(s) dialogues are designed to try to corrupt processing of the first user data and processes and to do other

Trans #	Rents	Returns	Late Fees	Payment	Receipt
T111	2	0	–	Exact	Automatic
T112	1	0	–	Over	Automatic
T113	1	1 (Total)	No	Over	Automatic
T121	10	0	–	Over	Automatic
T122	0	2 (From T121)	No	–	No
T141	0	2 (From T121)	2, 4 days	Over	Automatic
T151	4	2 (From T121)	2, 5 days	Over	Automatic
T211	1	1 (Total)	1 day	Exact	Automatic
T212	0	1 (Total)	No	–	No
T213	0	1 (Total)	No	–	Requested
T214	0	1 (Total)	2 days	Under, then exact	Automatic
T221	2	0	–	Under—abort	No
T222—Wait required	0	2 (From T121)	No	–	Requested
T311	0	1 (Total)	10 days	Over	Automatic
T312	1 (with other open rentals)	0	–	Over	Automatic
T313	6 (with other open rentals), error then rent 5	1	0	Exact	Automatic
T411=T311 Err	0	1 (Total)	10 days	Over	Automatic
T412=T312 Err	1 (with other open rentals)	0	–	Over	Automatic
T413=T313 Err	6 (with other open rentals), error then rent 5	1	0	Exact	Automatic
T331	0	2 (From T121)	2, 2 days	Exact	Automatic
T332	2	0	–	Under—abort	No
T511	5 (with other open rentals)	2	1 tape, 3 days	Over	Automatic

NOTE: Txyz Transaction ID: x = User, x = Day, z = Transaction number

FIGURE 17-24 ABC Video System Test Overview—Rent/Return Transactions

independent processing. If the total number of expected system users is six people simultaneously, then the system test should be designed for six simultaneous users.

The first step is to list all actions, functions, and transactions to be tested. For example, Figure 17-24 lists required transactions to test multiple days and all transaction types for each major file and process-

User 1	User 2	User 3	User 4	User 5	User 6
Start-up—success	Start-up—Err	Start-up—Err	Password—Err	Logon—Err	
Logon	Logon	Logon	Logon	Logon	Logon
Rent—T111 Errs + Good data	Rent—T211 Errs + Good data	Cust Add Errs + Good data	Cust Change—Err, Abort	Video Add Errs + Good data	Shutdown—Err
Rent—T112	Rent—T111 Err, abort	Rent—T311	Cust—Change	Copy Change—Errs + Good data	Try to crash system with bad trans
Rent—T113	Rent—T212—Err	Rent—T312	Rent—T411	Rent—T511	Delete Cust—Errs + Good data
Rent—T14	Rent—T213	Rent—T313	Rent—T412	Rent—any trans	Delete Video Errs
Rent—any trans	Rent—any trans	Rent—any trans	Rent—any trans	Rent—any trans	Delete Copy—Errs + Good data
END OF DAY, SHUT-DOWN, and STARTUP					
Rent—T121	Rent—T221	Rent—any trans	Rent—any trans	Rent—any trans	Rent—any trans
Rent—T122	Rent—T111	Rent—any trans	Rent—any trans	Rent—any trans	Rent—any trans
END OF DAY, SHUT-DOWN, and STARTUP					
Cust Add Errs + Good data	Cust Change—Err, Abort	Rent—T331	Copy Change—Errs + Good data	Try to crash system with bad trans	Rent—any trans
Delete Cust—Errs + Good data	Delete Video Errs	Rent—T332	Cust—Change	Video Add	Rent—any trans
END OF DAY, SHUT-DOWN, and STARTUP					
END OF MONTH					

NOTE: Txyz Transaction ID: x = User, x = Day, z = Transaction number

FIGURE 17-25 ABC Video System Test Overview—Test Schedule

ing activity for *Rent/Return*. These transactions would be developed into a test script for a single user test of the application.

Then, the transactions are interleaved with other erroneous and legal transactions for the other ABC processes as planned in Figure 17-25. Notice that the

required transactions are only a subset of the total transactions included in the test. The required transactions provide for exhaustive transaction coverage. The other transactions in Figure 17-25 are a mix of simple and complex transactions. Test scripts to follow the plan for each user are then developed; this is left as a student exercise.

Last, the test is conducted. During each shutdown procedure, the end-of-day reports are generated and reset. The data may or may not be checked after the first day to verify that they are correct. If errors are suspected, the files and report should be checked to verify accuracy. When one whole day is run through without errors, the entire set of test scripts can be executed. After an entire execution of each test script completes, the test team convenes and reviews all test scripts together to discuss unexpected results. All data from the files are verified for their predicted *final* contents. That is, unless a problem is suspected, intermediate intraday results are not verified during system testing. Errors that are found are reconciled and fixed as required. The test scripts are run through repeatedly until no errors are generated. Then, the test team should take real transactions for several days of activity and do the same type of test all over again. These transactions should also have file and report contents predicted. This 'live-data' test should be successful if system testing has been successful. If it is not, the errors found should be corrected and transactions to cause the same errors should be added to the system test. After the test is complete, the regression test package is developed for use during application maintenance.

AUTOMATED SUPPORT TOOLS FOR TESTING

Many CASE tools now support the automatic generation of test data for the specifications in their design products. There are also hundreds of different types of automated testing support tools that are not related to CASE. Some of the functions of these tools include

- static code analyzers
- dynamic code analyzers
- assertion generators and processors
- test data generators
- test driver
- output comparators

In Table 17-10, several examples of CASE testing tools are presented. Many other types of testing support tools are available for use outside of a CASE environment. The most common test support tools are summarized below and sample products are listed in Table 17-11.

A code analyzer can range from simple to complex. In general, **static code analyzers** evaluate the syntax and executability of code without ever executing the code. They cross-reference all references to a line of code. Analyzers can determine code that is never executed, infinite loops, files that are only read once, data type errors, global, common, or parameter errors, and other common problems. Another output of some static analyzers is a cross-reference of all variables and the lines of code on which they are referenced. They are a useful tool, but they cannot determine the worth or reliability of the code which are desired functions.

A special type of code analyzer *audits* code for compliance to standards and structured programming (or other) guidelines. Auditors can be customized by each using company to check their conventions for code structure.

A more complex type of code analyzer is a dynamic tool. **Dynamic code analyzers** run while the program is executing, hence the term dynamic. They can determine one or more of: coverage, tracing, tuning, timing, resource use, symbolic execution, and assertion checking. **Coverage analysis** of test data determines how much of the program is exercised by the set of test data. **Tracing** shows the execution path by statement of code. Some tools list values of key variables identified by the programmer. Languages on PCs usually have dynamic tracers as an execute option. **Tuning analyzers** identify the parts of the program executed most frequently, thus identifying code for tuning should a timing problem occur. **Timing analysis** reports CPU

TABLE 17-10 CASE Test Tools

Tool Name	Vendor	Features and Functions
Teamwork	Cadre Technologies, Inc. Providence, RI	Testing Software
Telon and other products	Pansophic Systems, Inc. Lisle, IL	Code Generation, Test Management

time used by a module or program. **Resource usage** software reports physical I/Os, CPU time, number of database transactions, and other hardware and software utilization. **Symbolic executors** run with symbolic, rather than real data, to identify the logic paths and computations for programmer-specified levels of coverage.

An **assertion** is a statement of fact about the state of some entity. An **assertion generator** makes facts about the state the data in a program should be in, based on test data supplied by the programmer. If the assertions fail based on program performance, an error is generated. Assertion generators are useful testing tools for artificial intelligence programs and any program language with which a generator can work. **Assertion checkers** evaluate the truth of programmer-coded assertions within code. For instance, the statement *'Assert make-buy = 0.'*, might be evaluated as true or false.

A **test data generator (TDG)** is a program that can generate any volume of data records based on programmer specifications. There are four kinds of

TABLE 17-11 Other Testing Support Tools

Tool Name	Vendor	Features and Functions
Assist		Coverage analysis, logic flow tracing, tracing, symbolic execution
Attest	University of Massachusetts Amherst, MA	Coverage analysis, test data generation, data flow analysis, automatic path selection, constraint analysis
Automatic Test Data Generator (ATDG)	TRW Systems, Inc. Redondo Beach, CA	Test data generation, path analysis, anomaly detection, variable analysis, constraint evaluation
Autoretest	TRW, Defense Systems Dept. Redondo Beach, CA	Comparator, test driver, test data management, automated comparison of test parameters
C/Spot/Run	Procase Corp. Santa Clara, CA	Syntax analysis, dependency analysis, source code filtering, source code navigation, graphical representation of function calls, error filtering

TABLE 17-11 Other Testing Support Tools (*Continued*)

Tool Name	Vendor	Features and Functions
COBOL Optimizer Instrumentor	Softool Corp. Goleta, CA	COBOL testing, path flow tracing, tracing, tuning
Cotune		Coverage analysis, timing
Datamacs	Management & Computer Services, Inc. Valley Forge, PA	Test file generation, I/O specification analysis, file structure testing
DAVE	Leon Osterweil University of Colorado Boulder, CO	Static analyzer, diagnostics, data flow analysis, interface analysis, cross-reference, standards enforcer, documentation aid
DIFF	Software Consulting Services Allentown, PA	File comparison
FACOM and Fadebug	Fujitsu, Ltd.	Output comparator, anomaly detector
Fortran Optimizer Instrumentor	Softool Corp. Goleta, CA	Coverage analysis Fortran testing, path flow tracing, tracing, tuning
McCabe Tools	M. McCabe & Associates Columbia, MD	Specification analysis, visual path testing generates conditions for untested paths computes metrics
MicroFocus Cobol Workbench	MicroFocus Palo Alto, CA	Source navigation, interactive dynamic debugging, structure analysis, regression testing, tuning
Softool 80	Softool Corp. Goleta, CA	Coverage analysis, tuning, timing, tracing
UX-Metric	Quality Tools for Software Craftsmen Mulino, OR	Static analyzer, syntax checking, path analysis, tuning, volume testing, cyclic tests

test data generators: static, pathwise, data specification, and random. A **static TDG** requires programmer specification for the type, number, and data contents of each field. A simple static TDG, the IEBDG utility from IBM, generates letters or numbers in any number of fields with some specified number of records output. It is useful for generating volumes of test data for timing tests as long as the records contain mostly zeros and ones. Unless the test data generator is easy to use, it quickly becomes more cumbersome than self-made test data.

Pathwise TDGs use input domain definitions to exercise specific paths in a program. These TDGs read the program code, create a representation of the

control flow, select domain data to create representative input for a programmer-specified type of test, and execute the test. The possible programmer choices for test type include all feasible paths, statement coverage, or branch coverage. Since these are white-box techniques, unless a programmer is careful, a test can run for excessively long times.

Test drivers are software that simulate the execution of module tests. The tester writes code in the test driver language to provide for other module stubs, test data input, input/output parameters, files, messages, and global variable areas. The driver uses the test data input to execute the module. The other tester-defined items are used during the test to execute pieces of code without needing physical interfaces to any of the items. The major benefits of test drivers are the ease of developing regression test packages from the individual tests, and the forced standardization of test cases. The main problem with drivers is the need to learn another language to use the driver software.

On-line test drivers are of several types. **Batch simulators** generate transactions in batch-mode processing to simulate multi-user, on-line processing. **Transaction simulators** copy a test script as entered in single-user mode for later re-execution with other copied test scripts to simulate multi-user interactions.

Output comparators compare two files and identify differences. This makes checking of databases and large files less time-consuming than it would otherwise be.

SUMMARY

Testing is the process of finding errors in an application's code and documentation. Testing is a difficult activity because it is a high-cost, time-consuming activity for which the returns diminish upon success. As such, it is frequently difficult for managers to understand the importance of testing in application development.

The levels of developmental testing include unit, integration, and system. In addition, an agent, who is not a project team member, performs quality assurance testing to validate the documentation and processing for the user. Code tests are on subroutines, modules, and programs to verify that individual code units work as expected. Integration tests verify the logic and processing for suites of modules, verifying intermodular communications. Systems tests verify that the application operates in its intended environment and meets requirements for constraints, response time, peak processing, backup and recovery, and security, access, and audit controls.

Strategies of testing are either white-box, black-box, top-down, or bottom-up. White-box tests verify that specific logic of the application works as intended. White-box strategies include logic tests, mathematical proof tests, and cleanroom tests. Black-box strategies include equivalence partitioning, boundary value analysis, and error guessing. Heuristics for matching the test level to the strategy were provided.

REFERENCES

Currit, P. A., M. Dyer, and H. D. Mills, "Certifying the reliability of software," *IEEE Transactions of Software Engineering*, Vol. SE-12, 1986, pp. 3–11.

Dunn, Robert H., *Software Quality: Concepts and Plans*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1990.

Mills, H. D., M. Dyer, and R. Linger, "Cleanroom software engineering," *Software*, Vol. 4, #5, 1987, pp. 19–25.

Musa, J. D., and A. F. Ackerman, "Quantifying software validation: When to stop testing?" *Software*, Vol. 6, #3, May, 1989, pp. 19–27.

Myers, Glenford J., *The Art of Software Testing*. NY: John Wiley & Sons, 1979.

Selby, R. W., V. R. Basili, and F. T. Baker, "Cleanroom software development: An empirical evaluation," *IEEE Transactions of Software Engineering*, Vol. SE-13, 1987 pp. 1027–1037.

BIBLIOGRAPHY

De Millo, Richard A., W. Michael McCracken, R. J. Martin, and John F. Passafiume, *Software Testing and Evaluation*. Reading MA.: Benjamin Cummings Publishing Co., 1987.

This text describes testing and evaluation for military contracts and compliance with Department of Defense standards such as 2167a which describes the phases

and documents required of all government sponsored software development projects. It includes a rich description of different types of testing, in particular formal verification.

Dyer, M., *Cleanroom Software Development Method*. IBM Federal Systems Division, Bethesda, MD, October 14, 1982.
This monograph is a detailed description of the cleanroom development method.

Mills, H. D., M. Dyer, and R. Linger, "Cleanroom software engineering," *Software*, Vol. 4, #5, 1987, pp. 19–25.
This is a brief description of the methodology of cleanroom development which includes a description of testing.

Musa, John D., Anthony Iannino, and Kazuhira Okumoto, *Software Reliability: Measurement, Prediction, and Application*. NY: McGraw-Hill Book Co., 1987.
This text takes a quantitative approach to proving program correctness.

Musa, J. D., and A. F. Ackerman, "Quantifying software validation: When to stop testing?" *Software*, Vol. 6, #3, May, 1989, pp. 19–27.
Musa and Ackerman's article discusses the trajectory of error finding over test shots and when the risk of stopping begins to diminish.

Myers, Glenford J., *The Art of Software Testing*. NY: John Wiley & Sons, 1979.
This is the best book I have ever read on testing. It is short, clear, and easy to follow. The only drawback is that real-time systems were not prevalent enough to have been included in the book.

Selby, R. W., V. R. Basili, and F. T. Baker, "Cleanroom software development: An empirical evaluation," *IEEE Transactions of Software Engineering*, Vol. SE-13, 1987 pp. 1027–1037.
This article reviews cleanroom projects and develops statistics about reliability of the software.

KEY TERMS

acceptance test
assertion
assertion checker
assertion generator
batch simulator
black-box strategy
bottom-up testing
boundary value analysis
cause-effect graphing
cleanroom development
coverage analysis
developmental test
dynamic code analyzer
equivalence partitioning
error guessing
integration test
output comparator
pathwise test data generator
program stub
quality assurance (QA) test
regression test
regression test package
resource usage
scaffolding
static code analyzer
static test data generator
subsystem test
symbolic executor
system test
test case
test data generator (TDG)
test driver
test plan
test script
test strategy
testing
timing analyzer
top-down testing
tracing
transaction simulator
tuning analyzer
type 1 error
type 2 errors
unit test
Vienna Development Method (VDM)
white-box strategy

EXERCISES

1. Describe the process of test development for an application. What are the roles, activities, documentation, and procedures followed by participants to testing?
2. Develop a test script for user 1 for the system test.

STUDY QUESTIONS

1. Define the following terms:
   scaffolding — test strategy
   white box testing — test case
   black box testing — test plan
   integration test
2. What is testing and why is it important?
3. How do you know when a test result is right?
4. Why do managers shorten the time allotted to testing?
5. Why do SEs and programmers sometimes resent testing?
6. What is the purpose of predicting results? Do the results have to be exact or can they be approximated? Why?
7. What is the purpose of a unit test? How is it met through test strategy selection?
8. When is it appropriate *not* to test all program logic? How do you decide what *to* test?

9. What are the different test strategies? Define each and discuss how they differ.
10. How many test cases does a program need?
11. What is the purpose of an integration test? What test strategy(s) are usually used in integration testing? Why?
12. What is the purpose of a systems test and how does it differ from the other test types: unit and integration?
13. Why is top-down testing *by itself* not a good idea at the system level?
14. Why is top-down testing a good idea at the system level?
15. At which test levels are bottom-up and top-down most appropriate? How do the top-down and bottom-up pieces get integrated?
16. At which level of testing does the human interface get tested?
17. How does prototyping fit with testing? Does prototyping also require a testing strategy? Why or why not?
18. What is the role of users during testing? Can users conduct the systems test? the integration tests? the unit tests? For each, why or why not?
19. For each level of testing, when can you end testing?

★ EXTRA-CREDIT QUESTIONS

1. Develop the test plan for Customer Maintenance in the ABC rental application.
2. Develop test scripts to unit test the other three transaction types for ABC Video. Use the screen design from Chapter 14 to help you visualize the data and processing requirements. The three transaction types are rentals without returns, returns without rentals, rentals with returns (i.e., *Customer ID* is entered first rather than *Video ID*).
3. Develop a test strategy for testing the entire application for a case in the appendix. Keep in mind that testing that involves users should minimize their time commitment while obtaining essential information from their involvement. Specifically define roles, responsibilities, timing, and test strategy for each level of testing.
4. Develop a presentation to senior user and IS managers to justify the time and resources required to do application testing. Present the discussion to your class.

CHAPTER 18

CHANGE MANAGEMENT

INTRODUCTION

Nothing is rarer in information systems development than an application without changes. Users forget requirements and remember them late in the design. The business changes. Bugs get fixed and require documentation. Change occurs in all phases and all levels of application development. Procedures to manage change, therefore, are necessary to maintain sanity and order on the project team.

The three major types of change in an application's life cycle—requirements, software, and documentation—are discussed in this chapter. For each, the importance of the change management techniques is discussed. Then, for each, techniques for managing changes are developed. At the end of the chapter, automated tools are identified for collaborative work, documentation, reverse engineering, and code management. First, we discuss the importance of designing for maintenance, regardless of the environment, architecture, or item being developed.

DESIGNING FOR MAINTENANCE

Applications are usually in production for an average of eight years. Many applications are much older, having been patched and modified regularly for 10 or even 20 years. Applications that are flexible enough to withstand years of modification are designed with change in mind. That is, regardless of the methodology, independent modules with local effects are developed.

Programs with 10,000 lines of, for instance, COBOL procedure code, rarely are modified easily. Usually, they are such spaghetti, that if they ever work, it is due to good luck. Frequently, change is precarious and likely to cause problems in untouched parts of the program.

In this section, we discuss the techniques used in designing for maintenance. The first, reusable libraries, have been used widely in the aerospace industry. Because cost savings can now be demonstrated from reusable libraries, they are moving into other industry segments. Reusable modules are complete programs that perform some complete function. The next section relates methodology to maintenance effort and discusses how each methodology attempts to provide for maintenance. Finally, CASE tools are related to maintenance and change.

Reusability

Reusability is a property of a code module such that the module can be used, as is, by several applications. In designing for reuse, the goal is to identify modules for potential reuse. The two most popular

methods of implementing code reuse are program templates and reusable modules.

Program templates consist of standard code that performs a simple function. For instance, there are three basic types of business programs: report, edit/validate, and file update. For a report, there are standard sections for reading file data, formatting the data, and writing the report (see Figure 18-1). Reading and writing can be standardized regardless of the data definition for input. The formatting of data must be customized. In writing the report, there are sections of code for beginning-of-page, body-of-page, and end-of-page. There may be sections for beginning-of-report and end-of-report, too. The report program might or might not have an internal sort routine that changes the sequence of the input file.

Templates can be developed to describe the 12 or so most common variants of the three basic types of programs. For instance, a report program is developed with and without sorts. COBOL or some other procedural language is used to define the standard versions and the only items left to the application programmer are procedures specific to the application.

The templates are stored as **read only** modules in a library. When a new use is defined, the module to be used is copied and given a new name. The newly named module is then modified and customized for its current use.

The advantage of a template is that a finite number of variations are developed and then are modified as needed for a specific use. There is little or no maintenance on the templates once they are developed, and only a few new templates per year would ever be developed. The number of support staff could be close to zero.

A template is a partial program that is completed for a particular application. A **reusable module** is a small, single function, well-defined, and standardized program module that can be used as a called routine, or as a *copy book* in COBOL. For instance, a date edit routine might be developed as a reusable module (see Figure 18-2).

When a reusable module is desired, a **library of reusable modules** is studied to determine which ones fit the application's needs. For reusable modules that do fit an application, the individual module code is examined to verify that it performs as required. Then the module is called at the appropriate place in the application's processing.

Each application team determines which modules it might have that could be reused in its own or in other applications. Then the modules are singled out for special development as independent routines. The finished module is quality assurance tested by the librarians to ensure that it performs as documented. **The librarian** is an expert in reusable standards, quality assurance testing, and code management techniques. Eventually, the code is stored in a reusable library whose contents are published for application developers' use.

Publication of reusable library contents can be awkward. Paper might be too voluminous to be useful or cost-effective. Electronic publication requires indices to assist users in identifying potential modules for their use. The indices might include keywords to describe function, language, date of development, type of input, and so on. If indices are not coded to capture the essential characteristics of the modules, they are useless.

The amount of organizational support required to maintain reusable libraries has been the major impediment to reusable library adoption in most industries. Librarians test, store, and maintain references to the modules in the reusable library. A large number of modules, for instance over 1,000, makes maintenance of the library integrity and accuracy a major task. Locating modules for use is also a major task. Librarians become specialized in performing these functions. Without proper organizational support, reusable libraries soon become unused and useless.

The arguments for reuse are substantial. As much as 75% of all code on a typical business application is redundant, and therefore, a candidate for reuse. Database descriptions, program procedure templates, and individual modules are all candidates for reuse that can save companies time and money in application development. The more reused code, the less extensive the custom code developed, the less extensive the testing required, and the less the cost of the application.

```
Identification Division.
Program-ID.  ABCVIDADD.
Environment Division.
Configuration Section.
Source-Computer.  IBM-3080.
Object-Computer.  IBM-3080.
File Section.
  Select Input-File from UR-D0001 as RPTIN.
  Select Report-File from UR-P001 as RPTOUT.
File Division.
Input Section.
FD          Input-File
  Block contains 100 records.
  Record contains 400 characters.
01          Input-File-Record                Pic x(400).
FD          Report-File
  Block contains 1 record.
  Record contains 132 characters.
01          Report-File-Record               Pic x(132).
Working-Storage Division.
01          Miscellaneous-counters.
            05    Page-Count                 Pic 99          value zero.
            05    Line-Count                 Pic 99          value zero.
            05    Input-record-count         Pic 9(7)        value zero.
            05    Output-record-count        Pic 9(7)        value zero.
            05    End-of-file-marker         Pic 9           value zero.
                  88    End-of-file                          value 1.
                  88    Not-end-of-file                      value 0.
********
01          Copy Input-File-Description statement goes here.
********
01          Report-Headers.
            05    Header-01.
                  10    Filler             pic x(45)       Value spaces.
                  10    H1                 pic x(23)       value
                        'Company Standard Header'.
                  10    Filler             pic x(15)       value spaces.
                  10    Date               pic x(8)        value spaces.
            05    Header-2.
                  10    Filler             pic x(45)       Value spaces.
                  10    H1                 pic x(23)       value
                        'Report Standard Header'.
                  10    Filler             pic x(15)       value spaces.
                  10    Time
                        15    Hour         pic xx          value spaces.
                        15    Filler       pic x           value ':'.
                        15    Hour         pic xx          value spaces.
                        15    Filler       pic x           value ':'.
                        15    Hour         pic xx          value spaces.
```

FIGURE 18-1 Partial COBOL Program Template for a Report

```
Linkage Section.
01          In-Date.
            05      In-Date-Month                   pic xx.
            05      In-Date-Day                     pic xx.
            05      In-Date-Year                    pic xx.
01          Errors.
            05      Err-table   occurs x times.
                    10 Err                          pic 9 comp.
Procedure Division.
Link.
            Enter linkage.
            Entry Link-date-edit using in-date, errors
            Enter COBOL.
Initialize.
            Move zeros to Errs.
Check-Numerics.
            If In-Date-Mo       not numeric move 1 to err(1).
            If In-Date-Day      not numeric move 1 to err(2).
            If In-Date-Year     not numeric move 1 to err(3).
            If err(1) = 1 or err(2) = 1 or err (3) = 1 go to End-Test.
Check-values.
            If In-Date-Day > 0
                    continue
            else
                    move 1 to err(4).
            If In-Date-Year > 1992
                    and In-Date-Year < 2015
                    continue
```

FIGURE 18-2 Reusable COBOL Module for Date Edit

Methodology Design Effects

In this section, we discuss the suitability of reusable libraries and program templates to the three classes of methodologies. Because of the encapsulation of data and function in object orientation, object methods are best suited to the large scale development of reusable modules. The other methodologies, process and data, can use program templates and reusable modules, but such modules are not identified as naturally as with objects.

Object methods are best suited to reusable components because the design method results in small, single function modules automatically. The method assumes that only data needed for a function will be available to it when it is called. Thus, the entire method assumes and strives for modules that are potentially reusable. When a module is identified in object analysis as being invoked from multiple calling objects, it is automatically targeted as potentially reusable. Further analysis determines if the functionality is identical for all users. If the functionality is the same, the module becomes locally reusable.

The step from local reuse to organizational reuse is small, with the criteria being the number of other applications needing the function. Here too, object methods are more amenable to identifying reusable functionality at the analysis stage than the other methodologies. Think back to Chapter 11, in which we developed the table of actions (or functions) and the objects to which they were attached (see Table 18-1). It is at this stage that reuse is identified. When an action has more than one object attached, they are examined to determine whether the same action is performed for each. If both objects use the action identically, they are labeled potentially reusable.

```
                else
                        move 1 to err(5).
                If In-Date-Month = 2
                        If In-Date-Year = (1992 or 1996 or 2000 or 2004 or 2008 or 2012)
                                If In-Date-Day < 30
                                        go to End-Test
                                else move 1 to err(6)
                        else
                        If In-Date-Day < 29
                                go to End-Test
                        else move 1 to err(7)
                else
                If In-Date-Month = (4 or 6 or 9 or 11)
                        If In-Date-Day < 31
                                go to End-Test
                        else move 1 to err(8)
                else
                If In-Date-Month = (1 or 3 or 5 or 7 or 10 or 12)
                        If In-Date-Day < 32
                                go to End-Test
                        else move 1 to err(9)
                else
                        move 1 to err(10).
End-Test.
                Enter linkage.
                Return.
                Enter COBOL.
```

FIGURE 18-2 Reusable COBOL Module for Date Edit (*Continued*)

Then, the potentially reusable actions are used to search the reusable library to see if similar actions in reusable form already exist. When a match is found, the reusable module code is examined to determine its fit to the current need. Based on the closeness of fit, the designers choose to design their own module or use the reusable module. The reusable module can be used as it exists or can be customized to exactly fit the application. The point is that the analysis action is matched to a reusable action at the *logical* level. Only when the logical actions match, the physical implementation is then examined for its appropriateness. When many such logical level matches are found, the time savings in analysis, design, and implementation can be considerable.

It has long been held that structured and modular design reduces maintenance effort by facilitating the definition of understandable *chunks* of analysis and designs. Modular design, in turn, is then applied to program modules. The designer uses his or her experience, applying the principles of information hiding, minimal coupling and maximal cohesion, to develop single function modules. In this manner, the nonobject methodologies are more *brute force* methods of developing modules with object-like properties. While the nonobject methodologies rely on personal designer knowledge, such knowledge also is more important in object methods than is commonly recognized at present. The results in nonobject methodologies, though, are less uniform and less likely to cause ready recognition of reusable components than object methods. Therefore, reusable component libraries are most likely to be effective and widely used in object-oriented environments.

TABLE 18-1 Sample Actions with Related Objects

Verb from Paragraph	Space	Process Name	Objects–Action*
is entered	S	EnterCustPhone	Customer, Data entry (DE)
to create	S	CreateOrder	Order (R)
are displayed	S	DisplayOrderVOO	Order, VOO (D)
are entered	S	EnterBarCode	VOO (DE)
are retrieved	S	RetrieveInventory	VideoInventory (R)
are displayed	S	DisplayInventory	VideoInventory (D)
computes	S	ComputeOrderTotal	Order (Process)
is entered	S	EnterPayAmt	Order (DE)
is computed	S	ComputeChange	Order (P)

*Actions are (R)ead, (W)rite, Data Entry (DE), (D)isplay (P)rocess in memory, (PR)int

The opposite situation is true of program templates. The nonobject methods, because they are used mostly for COBOL applications, can take advantage of program template libraries easily and effectively. As much as 60–80% of all COBOL code is **boilerplate**, that is, code which does not vary from one program to another. The boilerplate can be standardized and provided as program templates.

With object methods, the boilerplate in an object package is minimal but still can be standardized. The remaining code is either reused or customized. The types of COBOL template programs, for instance, a report with a sort, do not exist in the same form as objects. There might be a report object and there might be a sort object, and both might be reusable, but the code for *using* either object is most likely provided by custom developed code.

Role of CASE

Computer Aided Software Engineering (CASE) tools are critical to maintaining applications at the functional level rather than at the code level. The argument for CASE runs something like this. The 40-20-40 rule applies to software engineering application development. The rule states that 40% of the work is performed during feasibility, analysis, and design; 20% is during coding; and the remaining 40% is during testing (see Figure 18-3).

The 80-20 rule also applies (see Figure 18-3). According to this rule, 20% of the development work is performed during the original application development. The other 80% is performed during maintenance. This ratio holds because maintenance is a much longer period of an application's life.

Putting these two rules together, to gain substantive productivity increases we need to reduce time spent on coding, testing, and maintenance *more* than we need to reduce the time spent on analysis and design. CASE that covers analysis and design only reduces the time spent on documentation and maintenance of documents. CASE that includes database schema generation and code generation further reduces the coding, testing, and maintenance activities. Fully integrated CASE tools, I-CASE (see Chapter 3 and Automated Tools section of this chapter), that interface with code generators, support all of these productivity improvements. With I-CASE tools, maintenance changes are reflected in the requirements for an application. The requirements are, in turn, used to regenerate the database schemas and code for the application. Thus, the changes take place at the logical level and are *automatically generated* by the CASE tool at the physical level. The capability to do all application maintenance in this way is not here yet but should be before the new century.

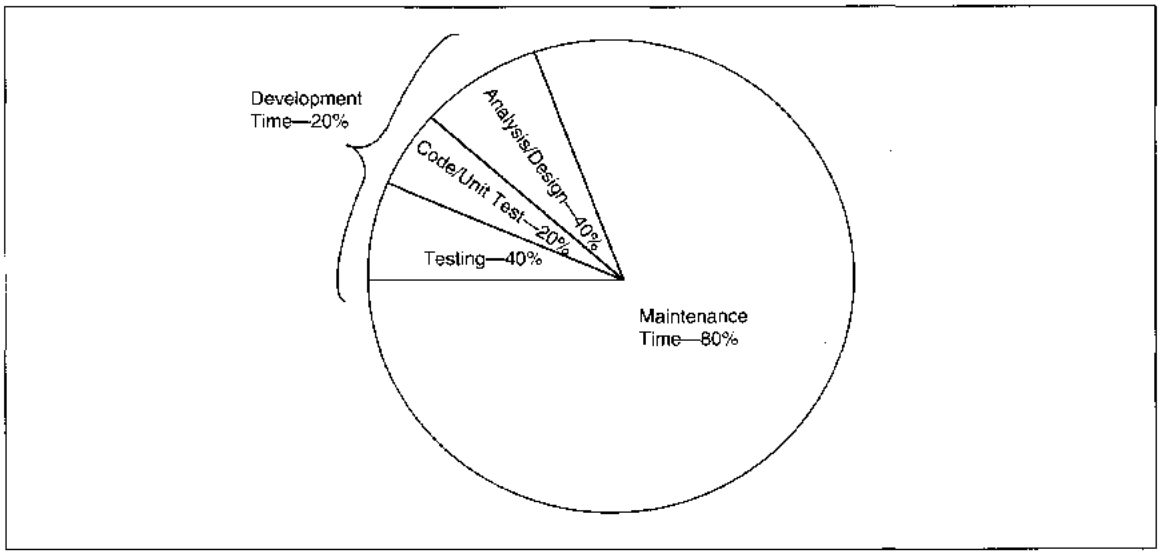


FIGURE 18-3 Application Life Cycle Time Distribution

A more futuristic feature of CASE tools will be the ability of the tool to recognize *reusable analysis and design fragments*, rather than relying on humans to recognize reusable *code fragments*. Purchasable options of the CASE tools will include intelligent options to detect feature and function similarities across applications. The fragments would then be imported from the original library to the using application library (or repository). Very intelligent CASE will be able to recognize a design fragment, logically link to the base definition of the reused item, and use already operational code modules. This level of intelligent CASE that could manage the use of reusable code may surface in our lifetimes, but not soon.

APPLICATION CHANGE MANAGEMENT

Importance

Applications frequently undergo redesign. Three typical conditions for redesign are assignment of a new management team, a project that is chronically over budget, late, and full of bugs, and the loss of the user-owner confidence that the SEs understand their needs. Even without drastic redesign, reviews (e.g., for user agreement or quality assurance) frequently turn up items that were originally compromised or rethought several times before final version agreement. The history of decisions and the reasoning about decisions is rarely kept as part of project notes. But, any project manager and SE can tell you that they frequently rehash the same arguments and reasonings over and over, even reaching the same conclusions.

In a paper-based work environment, keeping track of the history of decisions is not practical; so much paper would be generated that finding anything becomes impossible. In a CASE environment, or in an imaging environment, maintaining the history of application decisions electronically becomes a manageable, and sometimes desirable, activity. The ability to recall reasoning through a decision, whether it is logical or political, can save time and provide continuity between managers.

Finally, changes in the business, legal requirements, or stakeholders in the application can all necessitate legitimate changes to application designs. Knowing the history of decisions sometimes makes them more palatable and easier to convey to

staff. For instance, being able to relate a change of design to a developing business situation helps those who must cope with the change appreciate the business of the application. If the change is to keep a valued customer or increase competitiveness in a new area, the systems developers are more likely to be enthusiastic about shifting design.

Changes can be to requirements, designs, programs, interfaces, hardware, or purchased software. Most changes are initiated from within the organization developing the application, but might be motivated by some outside event, such as a change in laws. Using change controls protects the development team from user whims while allowing for action on legitimate requests. The idea that a specification is **frozen**, meaning not changeable after it is accepted as complete, motivates users to be as complete in their thinking as possible.

Designs do not stay frozen forever. Usually, once an application begins coding, no changes are implemented until the application becomes operational. Then the project manager, SE, and user review the backlog of requests to develop priorities and plan the changes. Some changes may be so critical that the design is unfrozen to add the crucial functionality, regardless of the phase of development.

Change Management Procedures

Change control management is in effect from the time the work product is accepted as complete until the project is retired. First, baseline work products that are to be managed are identified. A **baseline** work product is a product that is considered complete and that is the basis for other, current work by the project development team. A baseline document would be, for instance, the functional requirements specification after it is accepted by the user.

A history of change request file actions for a functional specification are listed here as an example.

1. Create Open Request
2. File Impact Statement
3. File Approval of Schedule and Cost signed by User/Owner
4. Complete Project Manager's Check List for the Change
5. File Documentation relating to changes. If documentation or programs changed, identify date and items updates completed. If procedures or training changed, identify dates at which revisions were operationalized.
6. File Close Request Form Approved by User/Owner
7. Summarize Dates, Durations, and Costs

First, the baseline document is frozen, then change requests are added, but no action is taken. The fourth request, for example, might be urgent and receive immediate attention. When the functional specification is updated to accommodate the change, it is again frozen and the work continues. The three previous requests might have been added to the application if they did not significantly alter it. They may just as likely be ignored until after the application is implemented.

Changes can be classified in several ways. First, they can be classified by type as eliminating defects, improving performance, or changing functionality. Second, changes can be classified as required or optional. Third, change can be classified by priority as emergency, mandatory with a required end date, mandatory with an open end date, or low priority. Usually, eliminating defects is a required emergency, while changing functionality is required mandatory maintenance, and improving performance is optional and might have any priority.

Knowing the change request classification determines whether it is subject to change control or not. Emergency changes usually circumvent the change control procedures in that the activities might all be followed but they are documented after the change is complete. All other change types should be required to comply with change controls.

For example, changes to functional requirements can occur at any time, but once the functional requirements specification is approved, it is frozen until the application is operational. Changes are subject to change control: they are added to a change request list for future consideration unless given an emergency designation.

Project # _____

Project Name _____

CHANGE CONTROL REQUEST

Initiator _____ Date _____

Department _____ Request # _____

Reason for Request

Description of Change

Documents Affected:		Category of Change	
Func. Spec.	_____	A. Reqts.	_____
Interface	_____	B. Design	_____
Design	_____	C. Code	_____
Mod. Spec.	_____	D. Interface	_____
Code	_____	E. Hardware	_____
Operations	_____	F. Other	_____
User Doc.	_____		

Class of Change

Emergency	_____
Mandated	_____
Enhancement	_____
Other	_____

Initiator	Date		
Owner	Date	Prroject Manager	Date

FIGURE 18-4 Sample Change Request Form

A procedure for change control (listed below) requires that a formal request for a change is submitted by the user to the project manager (PM).

1. User sends the project manager and owner (if different person) a Change Request form (see Figure 18-4).
2. Project manager and SE develop an impact statement. At this time, the project manager's Check List is used to identify all work actions and changes relating to the request.
3. The Change Request is discussed with the User/Owner to establish priority, schedule, and cost changes.

4. Agreement is formalized and User/Owner approval of schedule and cost changes is obtained.
5. Using the impact statement, application and all related documentation are changed. Implement the change. As tasks are complete, check off the task on the project manager's Check List.
6. User/Owner approval to close the request is obtained and the request is closed.

The PM and SE define the schedule and cost impacts of the change (see Figure 18-5). The changes are then discussed with the user. Based on the negotiation with the user, the change is assigned a priority for action, and the cost and schedule are changed.

The request, expected date of action, schedule change, and cost increments are added to a project history file. The changes may be monitored by a **Change Control Clerk**, a person charged with maintaining project history and change control records, and with issuing a monthly change control report. A **Change Control File** contains all requests, correspondence, and documentation about changes. An **Open Change Request** is created when the request is made and a change number is assigned. The open change request stays on file until the request is completed, closed, and reported.

As the change is made, affected items are updated, including the appropriate documentation, code, training, and so forth (see Figure 18-6). A project manager's check list is used to check off required actions. The new documentation is filed with the Change Control Clerk who distributes it to all interested parties.

The completion date for the change is entered in the Change Control File. The change is identified as closed in the next status report and the open request is removed from the Change Control File.

Depending on the organization, the IS executive might want to track change requests for projects to identify success in meeting requests. Overall costs of changes for a year are used as one indicator that an application is a candidate for either retirement or reengineering. In such cases, both costs and volumes of change requests are tracked through the change control process. Summary reports by project of the changes over a given period, or comparing periods (e.g., a current period compared to the same period last year) can be developed. Three such reports are shown as Figures 18-7 through 18-9 for total cost by type, cost and schedule impacts, and change requests, respectively.

Historical Decision Logging

At the beginning of the project, the project manager and SE decide to use tools to store the decision process. This means that either electronic group meetings are used or that a written version of meetings and decisions is maintained and stored in word processed form. With electronic meetings, the electronic transcripts are maintained. With manual recording, the old version is updated and renamed when a document changes. For instance, functional specifications for ABC might be named *ABCFS-mmddyy*, where *ABC* is the company, *FS* abbreviates Functional Specification, and *mmddyy* is the date. The date portion of the name would change for every major change of the document. The change management procedure in the next section would be followed.

Documentation Change Management

Documentation changes should be identified by a change table of contents at the beginning of each document. The change table of contents includes the effective date, affected sections of the document, and a summary of the change (see Figure 18-10). The purpose of the change table of contents is to summarize all changes for the reader.

Changes should be redlined in the text to identify the changed portion. If the old information is important, it may be moved to a footnote, dated, and labeled as a previous version. An example of this type of documentation change is shown in Figure 18-11. Keep in mind that you also keep the old version of the document for history.

(Text continues on page 749)

Project # _____

Project Name _____

CHANGE CONTROL IMPACT ASSESSMENT

Date _____

Request # _____

Impact of Change Request:

Impact

Type	Cost	Person Days	Business Days
A.	_____	_____	_____
B.	_____	_____	_____
C.	_____	_____	_____
D.	_____	_____	_____
E.	_____	_____	_____
F.	_____	_____	_____
Total	_____	_____	_____

Budget Control

Initiation Date	_____
Request #	_____
Amount	_____
Approval Date	_____

STATUS	Scheduled Completion	Actual Completion
Initiated Date	_____	_____
Analysis Date	_____	_____
Development Date	_____	_____
Testing Date	_____	_____
Implementation Date	_____	_____

Comments:

Initiator Date

Owner Date Prroject Manager Date

FIGURE 18-5 Sample Change Request Impact Form

Project # _____ Date _____

Project Name _____ Request # _____

PROJECT MANAGER CHANGE CONTROL CHECK LIST

DEVELOPMENT

	Required	Completion Date
1. QA/Documentation Review		
2. Update Source Document(s)		
3. Update Baseline Document(s)		
4. Update Program Specifications		
5. Revise Code		
6. Update User Documentation		
7. Update Operations Documentation		
8. Other: _____		

IMPLEMENTATION

	Required	Completion Date
1. Baseline Documents Update		
2. Requirement Change		
3. Design Changes		
4. Programming Changes		
Pgm #'s _____, _____, _____		
_____, _____, _____		
5. Unit Testing		
6. System/Regression Testing		
7. Interface Changes		
8. Operations Changes		
9. Other: _____		

Comments:

Initiator Date

Owner Date Prroject Manager Date

FIGURE 18-6 Project Manager's Change Check List

CHANGE CONTROL ANALYSIS BY TYPE — Month of: May, 1994

PROJECT-TO-DATE

Number and Cost of Change by Type

Application Name:	#	A Cost	#	B Cost	#	C Cost	#	D Cost	#	E Cost	#	F Cost	#	G Cost	Total #	Cost
1. Branch Pilot	60	$45.6	1	$ 2.6	–	–	–	–	3	$40.7	–	–	–	–	64	$ 88.9
2. Securities Transfer	17	–	–	–	2	–	–	–	–	–	–	–	–	–	19	–
3. Settlements	16	36.0	11	18.6	–	–	–	–	–	–	2	.5	–	–	29	55.1
4. Float Allocation	–	–	3	6.0	16	$11.0	–	–	3	.4	1	$10.0	–	–	23	27.4
Total	93	$81.6	15	$27.2	18	$11.0	–	–	6	$41.1	3	$10.5	–	–	135	$171.4

Change Type Legend

A. Requirements/Design
B. Application Programs/Testing
C. Documentation
D. Hardware
E. Purchased Software
F. Interfaces
G. Application Support

Notes: Costs in thousands
Changes with no cost are planned maintenance.

FIGURE 18-7 Summary Report of Change Costs

CHANGE CONTROL COST/SCHEDULE IMPACT* Month of May, 1994

Application	Current Month Cost	Current Month Schedule	Year-to-Date Cost	Year-to-Date Schedule	Project-to-Date Cost	Project-to-Date Schedule
1. Branch Pilot	–	–	$ 48.8	24	$ 88.9	39
2. Securities Transfer	$ 15.0	8	15.0	8	25.0	14
3. Settlements	111.0	64	111.0	64	225.0	140
Total	$126.0	72	$174.8	96	$338.9	193

*All data based on change Submission Date

Cost in thousands

Schedule in business days

FIGURE 18-8 Summary Report of Cost and Schedule Impacts

CHANGE CONTROL ACTIVITY Month of May, 1994

Project Name	Current Month S	P	A	D	O	C	Year-to-Date S	P	A	D	O	C	Project-to-Date S	P	A	D	O	C
1. Branch Pilot	–	–	–	–	–	–	6	1	4	1	1	3	64	1	51	12	25	26
2. Securities Transfer	3	1	2	–	1	1	22	1	18	3	6	12	22	1	18	3	6	12
3. Settlements	16	9	7	–	3	4	16	9	7	–	3	4	16	9	7	–	3	4
Total	19	10	9	–	4	5	42	11	29	4	10	19	102	11	76	15	34	42

LEGEND:

S Submitted
P Pending
A Approved
D Disapproved/Cancelled
O Open
C Completed

FIGURE 18-9 Summary of Change Requests

CHANGE PAGE

Page No.	Reason for/description of change	Date
1–48	Original issue	11/4/93
All	Audit review and Revisions	1/3/94
1–2, 22	Corrections and revisions to reflect organization changes	2/6/94
6, 9, 37–44	Describe imaging interface	6/3/94

FIGURE 18-10 Sample Document Change Table of Contents

SOFTWARE MANAGEMENT

Introduction

Two of the roles of the SE in software management are to recommend what type of maintenance should be performed and to select code maintenance software. These are discussed in this section.

Types of Maintenance

The types of maintenance are minor modifications, restructuring, reengineering, or rebuilding.[1] **Minor modifications** are changes to existing code and can be any of the project manager classifications discussed above. **Restructuring** is the redevelopment of a portion of an application with a bridge to the old application. **Reengineering** is the reverse analysis of an old application to conform to a new methodology, usually Information Engineering or object orientation. Reengineering is also known as **reverse engineering**. **Rebuilding** is the retirement and redevelopment of an application.

To select the appropriate type of maintenance, several questions are asked (see Figure 18-12). First, ask if the software works. If the answer is no, you retire the old application. Then you reengineer and rebuild it using a methodology. If the answer is yes, you continue to the next question: Does the

1 This discussion is based on Martin, 1990.

Functional Specification Settlements 1/15/94
Page 22

...

The settlements system uses relational database design techniques and fully normalized data entities.[1] The database design is fully documented in Figure 2-1. The diagram shows the 17 entities used in settlements processing and the relationships between them. Each entity and its descriptive attributes are fully described in the data dictionary attached as Appendix 1; they are also available on-line through both IEF, the CASE tool being used for the application, and Project-Notes, the on-line help tool.

...

1 Prior to January, 1994, a nonnormalized, relational approach to the data was used. This resulted in a loss of data integrity that necessitated strict enforcement of relational theory to comply with audit requirements for the application.

FIGURE 18-11 Sample Documentation Change with Old Contents

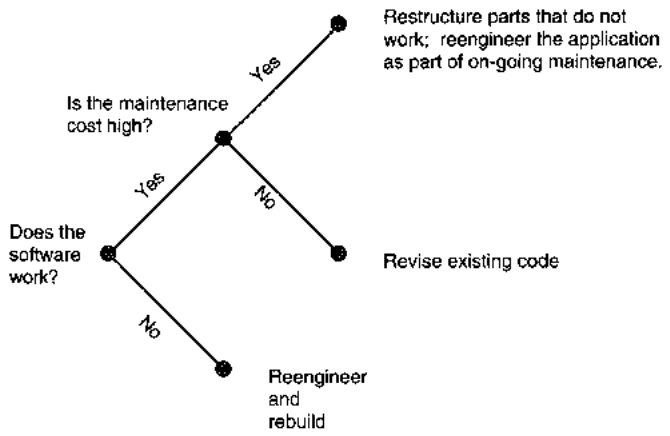


FIGURE 18-12 Decision Tree for Selecting the Maintenance Type

application have a high maintenance cost? If the maintenance cost is low, the answer is no; then do a simple revision. If the answer is yes, immediately restructure the parts that do not work, and reengineer the entire application as part of on-going work.

Reengineering

Reengineering is the analysis and design of an existing application to bring it into conformance with a methodology. When the application conforms to a methodology, it is rebuilt. To reengineer program code, the code first must be structured. Code restructuring can be done by automated tools. The restructured code from all programs in an application is entered into a CASE tool with reverse engineering capabilities.

Code restructuring also can be done manually. If no CASE products are used, the code is analyzed and the underlying data and process structures are mapped into a methodology. If Information Engineering is used, for instance, an entity relationship diagram (ERD) and a process data flow diagram (PDFD) are first developed for each program. Then, the diagrams are consolidated across programs to develop application ERDs and PDFDs. A data dictionary to document diagram contents is developed. The ERD is normalized and compared to the automated data to determine the extent of deviation from the normalized state. If the denormalized state was for performance purposes (this is an example of the importance of a historical file of design decisions), then problems with data integrity resulting from the denormalization should be noted for correction. Finally, the detailed process diagrams are used to develop a process hierarchy diagram. The hierarchy diagram is matched to the real organizational functions to determine the extent of application function redesign required.

If the methodology is object-oriented, the code modules are classified by object type and function. If multiple objects call a function, it is classified as reusable and set aside for further analysis. After module classification, the extent to which the code matches a true object design is determined. Reusable modules are evaluated to ensure that they perform single functions, hide information, and use minimal coupling techniques. For minor deviation from the object method, individual modules or object types are reengineered to bring them into conformance with object tenets. For major deviation, the application is reengineered and redeveloped using object techniques.

CONFIGURATION MANAGEMENT

Introduction

In the mainframe world, one disk storage device can hold 10,000 or more different data files; large projects develop hundreds of program modules every year; and programmers may manage several different versions of code modules at one time. To support multiple users across different platforms might require multiple operational versions and variations of code modules, and they all have to be maintained. **Configuration management** is the identification, organization, and control of modifications to software built by a programming team. **Code library** management software provides a means to identify and manage the baseline for program code modules. The **baseline** is the *official* version of a code module that is in production use at any time. Two types of code libraries and the application types they support are discussed in this section. Derivations, which identify each module's history, are included in the discussion.

Configuration management addresses problems originally present in large COBOL applications but are equally useful for the more complex environments of object and distributed software. A programmer might keep several copies of a program and personally track which is in production at any one time. The problem with individual programmers maintaining their own copies is that eventually their multiple copies will diverge and knowing which is the most current can be a problem. Trusting individuals to be good librarians is asking for errors.

Assume next that one *official* version of programs exists. If several people are performing maintenance tasks on the one version of a program, a high probability exists that the changes of one person will

interfere with the changes of the other person. Either the changes of one will be cancelled by being overwritten by the other, or one person will have to wait while the other makes the changes. Both situations lead to delays and are error prone.

In the complex world of distributed systems and multiple hardware/software platforms, different versions of the same software might be present. The only differences might be to accommodate platform idiosyncrasies, but such differences imply multiple versions of software that can cause maintenance problems. When a general change is made, somehow it must be verified as being made to all versions for all platforms. Specific changes for each platform must also be accommodated to allow fixing of bugs or changes that only affect one type of hardware.

Configuration management that consists primarily of code library management software plus manual procedures supports both single and multiple versions of programs to control for different platforms, evolving functionality, and debugging of software changes.

Types of Code Management

The most common code management procedure is the creation of derivations. The two code management types are for versions and variations. They can all be supported in the same software library or can be in separate libraries. Each type serves a different purpose.

Derivation

A **derivation** is a list that identifies the specific versions of multiple modules that were linked to create a load module or joint memory resident work unit. The purpose of a derivation is to allow tracing of errors that might be due to vendor software. All software used to create a load unit are specifically identified with vendor, version, and last installation date. The sample shown in Figure 18-13 identifies specific platform, operating system, compiler, for creation of a work unit, and the dates of the creation of each stage. If a problem were found, for example, a rounding error occurs in computing interest, the error is traced backward through the development software to find the problem. The program is checked first, then the compiler, then the operating system, and so on. Let's say, for instance, that a new version of the compiler was installed one week before this module's creation, and that, upon inspection, the rounding algorithm used only allowed four decimal places to real numbers. If more than four places are needed, a new compiler would be required.

The difference between a load module and joint memory resident work unit is in the dynamism of the processes. A **load module** is a compiled version of one or more source code modules that have been compiled and link-edited together, forming the load module. **Compilation** translates a module from source code to object (assembler) code. **Linkage editing** resolves references to other modules by replacing *Call* references with relative memory addresses, thus joining related modules for processing as a single work unit (see Figure 18-14).

A joint **memory resident work unit** is a series of load modules that work together in a dynamic, real-time environment. Linkage editing creates static modules that are fixed until the next linkage edit process. In real-time application environments, one goal of the procedures is to relieve the need to freeze specific module references until they are needed in operation. This liberates programmers from the linkage editing process but can create chaos when an error occurs and must be traced. Both situations require maintenance of derivations.

Recording of derivations requires precise identification of the software, option, code inputs, responsible person, and date that a load module was created (see Figure 18-15). The level of detail for derivations should match each process a module undergoes from source code to load unit. This means that if the translation is from source code to load unit, there are two derivations. If the translations are from source to object to load unit, there are three derivations. *All* software used in creating the derivation is recorded, including the compiler, linkage-editor, and so on, and their versions. Derivation maintenance provides an audit trail for software and is the only way that errors can be guaranteed to be traceable.

Work Unit Name: _____
Creation Date: _____

Date	Time	Software	Options	Code Module	Person
2/1/93	2:53a	Cob 88, 2.1	Defaults	STL1001	A. Bryon
2/1/93	2:54a	Cob 88, 2.1	Defaults	STL1002	A. Bryon
2/1/93	2:56a	Cob 88, 2.1	Defaults	STL1003	A. Bryon
2/1/93	2:58a	Cob 88, 2.1	Defaults	STL1004	A. Bryon
2/1/93	2:59a	Cob 88, 2.1	Defaults	STL1005	A. Bryon
2/1/93	3:00a	LinkEdit 88, 3.7	Defaults	STL1001, STL1002, STL1003, STL1004, STL1005 object	A. Bryon

Comments:

FIGURE 18-13 Sample Derivation

Delta Version

Delta means *difference*. A **delta file** is a file of *differences* between versions of a program. **Versions** are multiple copies of a single program that represent incremental changes.

When a delta version is kept, the main program logic is maintained once. Then, the delta version is applied to the main logic, with specific lines of code being replaced to derive the delta (see Figure 18-16). The advantage of using a delta strategy is that changes in functionality affect only the original

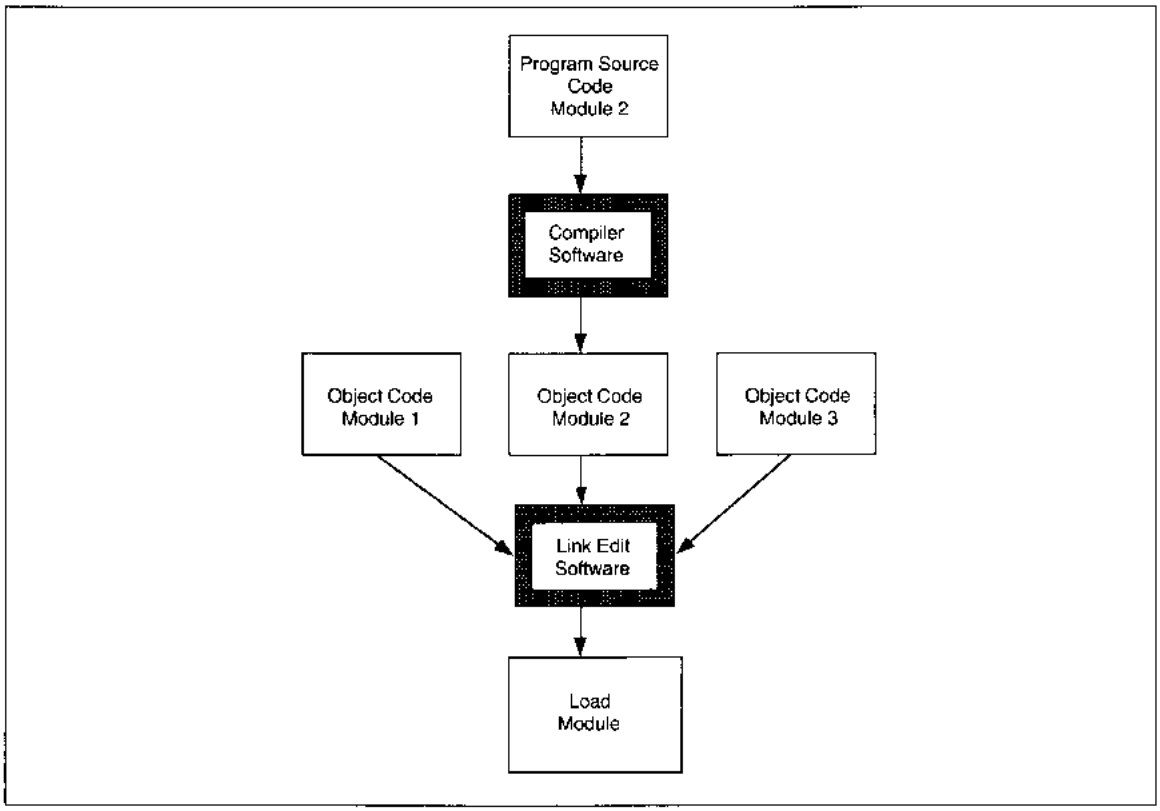


FIGURE 18-14 Compile and Link Edit

code. The disadvantages are that loss or corruption of the original also affects all deltas, and that delta references based on code line numbers can lead to errors when the original changes.

Many software librarians and operating system editors work on the delta version principle. For instance, the Unix editor maintains delta versions of changes to text files, which includes program code. Using line numbers as the reference point, the original is stored. As changes are made, changed lines are kept plus new line numbers are appended in a delta file. When the file is referenced, the original is loaded into memory, then the deltas are applied until the memory version reflects all changes.

When using a delta version, then, it is important to create a new file periodically to save storage and processing time for delta overlays. This minimizes the extent to which you are making changes to changes. To create the new file, you save the old file with a new name. Renaming modules is necessary to create a permanent version of the program with deltas incorporated into the saved version. Maintaining many renamed versions can cause errors in remembering the most current version, too.

Variation Storage

Variations are alternative, interchangeable program modules created for multiple environments or purposes. For instance, you might create an IBM PS/2 version of a program and a Novell Netware 386 version of a program. The functionality is the same, but specific modules are different to support the specific hardware/software platform.

Variations in a COBOL environment, for instance, might have a different interface for users in

Item	Definition
Date	Date when the derivation was created
Time	Time of day when the derivation was created
Software	Specific software used to create the derivation
Options	Software options selected or defaults
Code Module	Name of input module(s)
Person	Person executing the derivation create
Hardware	Machine ID if there are multiple machines
Installation	Location or other installation ID when there are multiples

FIGURE 18-15 List of Requirements for Recording Derivations

the United States and users in South America. Variations in an Ada environment, as another example, might be for performing the same process using integers or using real numbers.

Variations are named rather than numbered because there is no meaningful relationship between variations (see Figure 18-17). The name of each variation should reflect what makes it different. For instance, the names *PS2SORT* (for PS/2 sort routine) and *N386SORT* (for Netware 386 sort routine), would be good variation names because they identify both the platform and the function of the variation.

Configuration Management Procedures

Strict configuration management requires that one person (or group) on each development and maintenance project be assigned as the project librarian. The **project librarian** is the only person authorized to write into the baseline library for the project. The procedure is summarized below.

1. File baseline code module.
2. Allow checkout for read-only purposes to individuals needing access. For instance, test team needs access for testing.
3. Allow chargeout for update to authorized programmers.
4. Monitor that chargeout items are returned.
5. Notify testers of chargein items for testing.
6. Verify that the text preamble to code identifies the change, date, programmer, and lines of code affected.
7. Chargein the item, refiling the module.
8. If derivations are used, file the derivation with project documentation.

When a project is in the code and unit test stage, the project librarian establishes an application library. As each module is unit tested and moves into subsystem and integration testing, the programmer's final version is given to the project librarian for addition to the library.

Error fixes, changes during testing, and maintenance changes are all managed the same way. The programmer tells the librarian she or he is checking the module out for update, and the librarian keeps track of this fact. The code is copied out of the library and into the programmer's own workspace. The changes are made and unit tested. Upon completion of the unit test, the programmer gives the module and documentation to the librarian for reentry to the library.

The librarian checks that no other changes have been made during the time the programmer has the module out for update. If not, the module is rewritten into the library.

Depending on the library software used, additional features allow the librarian to issue a charge-out against a module. A **charge-out** causes a lock to be placed on the module such that no other chargeouts for update may be performed until the lock is removed. When the changed version of the code module is reentered into the library, a **charge-in** occurs. A charge-in is the updating of a charge-out module to remove the lock. The more intelligent the software, the more actions taken during charge-in. For instance, some library software packages initiate a regression test when a chargein action is taken.

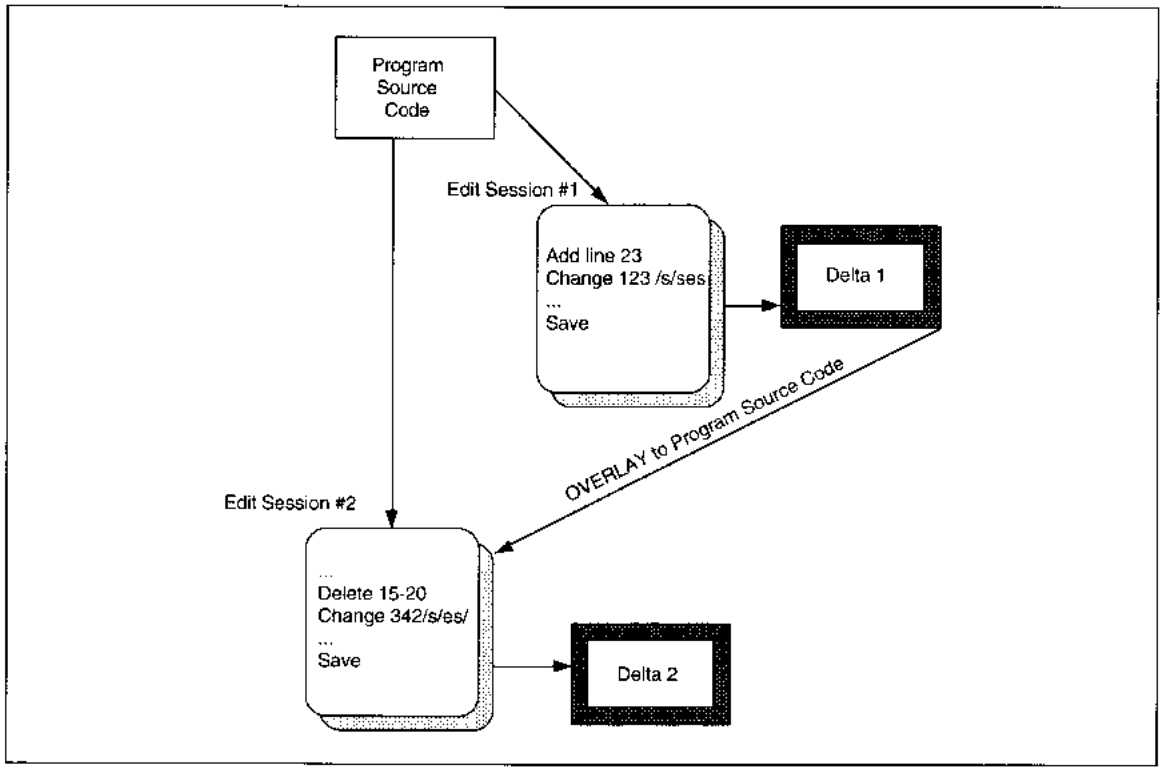


FIGURE 18-16 Delta Version Development

The disadvantage to having a formal project librarian is that the librarian becomes indispensable. The risk is that the librarian might become a bottleneck to updating the production library. For instance, if one person is the librarian, he or she might be called for jury duty and be out of work for several weeks. During that time, unless another librarian is named, no updates can be performed.

Automated Tools for Change Management

There are different classes of automated tools for each type of change management. Each class of tools is discussed separately in this section.

Collaborative Work Tools

Collaborative work tools support group decision making and facilitate the development and historical maintenance of project decisions. Collaborative tools have developed out of research programs in group decision making at the Universities of Arizona and Minnesota in collaboration with IBM. Relatively primitive software of the 1980s for facilitating meetings has blossomed into a new industry for facilitating group work. Xerox Palo Alto Research Center (PARC) is a major contributor of new technology to this industry.

The specific technologies involved range from the relatively familiar, like electronic mail, or e-mail, to the exotic, for instance, *media space* clear boards that change our concepts of *being there* (see Table 18-2). Many of the technologies are emerging, but the emergence is at such a rapid

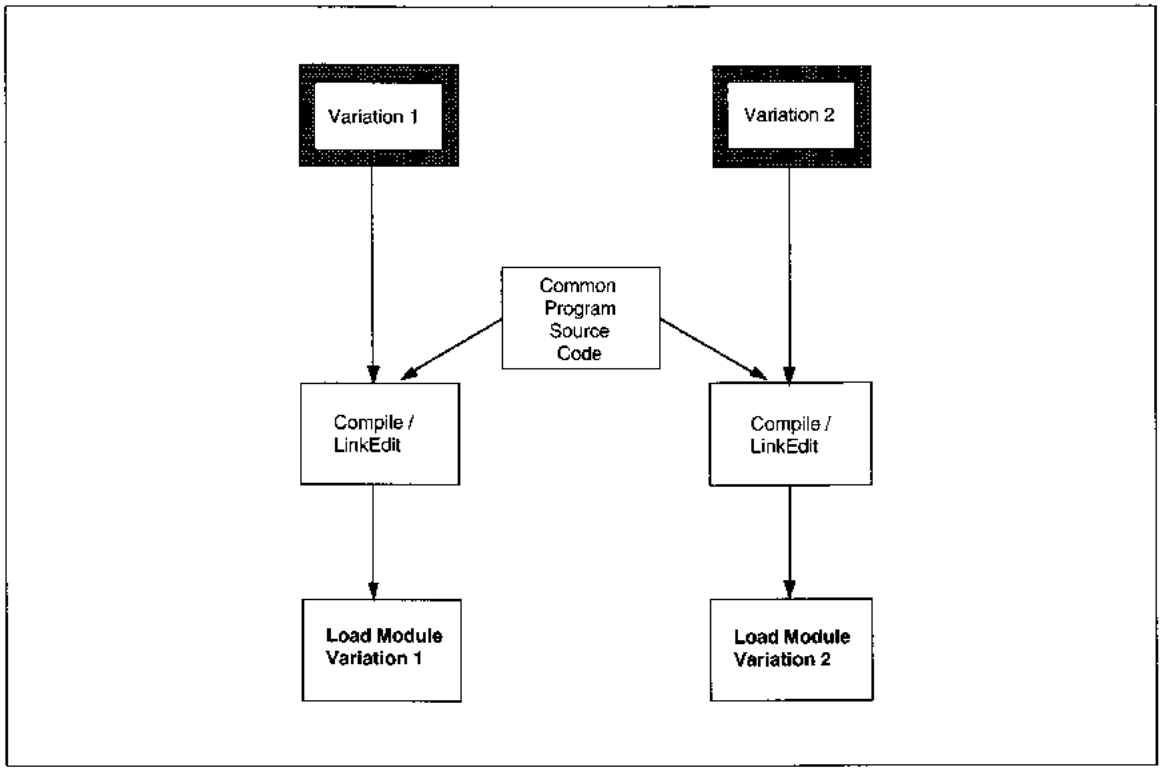


FIGURE 18-17 Variation Development

rate that by the new century we will routinely use many of these technologies at work, if not at our homes.

Media space technology allows several participants to sit on opposite sides of a *clear* glass *board* display that has electronics imbedded in it. The board can display computer images, text, and graphics as well as reflect hand-drawn notes and graphics of the meeting participants. The most effective use at the moment is between two people who both have clear access to the board. Clear boards allow people to see both the work and the co-worker, minimizing attention shift time. At the moment, the technology requires the people to be co-located, that is, in the same room; but the intention is to provide video conferencing capabilities using clear boards that are mirror images, thus *simulating* the face-to-face experience with the added electronic board interface. Thus, the user sees both the face of the other participant(s) and the contents of the board simultaneously. By removing the limitations of both time and geography our concept of *being there* is altered. By removing these limitations, clear board technology facilitates group work. This technology was developed, in this country, at Xerox PARC.

A different type of product provides a text-based communication environment that supports group passing of messages with storage of reader comments. Such a product, Notes,[2] provides an e-mail feature with the capability of user-built discussion *forums* and other data-sharing features. These products allow the development of decisions, history of the process, and easy sharing of information within and between work groups.

2 Notes® is a product of Lotus Development Corp.

TABLE 18-2 Collaborative Work Tools

Tool	Vendor	Functions
Cruiser®™	Bellcore Morristown, NJ	A video windowing system that allows the user to *cruise* offices visually and, perhaps, initiate a visit. Uses telephone and video technologies.
Greyboard	NeXT Computer Mountain View, CA	Multiuser drawing program
Groupkit	Dept. of Computer Science University of Calgary Calgary, Alberta, Canada	Real-Time Conferencing Toolkit; requires Interviews Software, Unix running X-Windows
Notes	Lotus Development Corp. MA	E-mail, group bulletin board, data sharing
Oracle Mail, Alert, Toolkit, and Glue	Oracle Corp. Redwood City, CA	E-mail, application development, and application programming interfaces for LANs
Timbuktu™	Farallon Computing, Inc. Berkeley, CA	Sharing of single-user software among several users
Video Whiteboard	ACM SIGCHI Proceedings '91, pp. 315–322	Wall-mounted whiteboard that portrays shadow of the other user
VideoDraw	ACM SIGCHI Proceedings '90, pp. 313–320	Multiuser drawing program
Windows for Workgroups	Microsoft, Inc. Belleview, WA	LAN-based windows sharing

Documentation Tools

Word processing tools, such as WordPerfect, are rapidly being replaced with more sophisticated and intelligent products for document development and maintenance (see Table 18-3).

In the old days of the 1980s, word processors became sophisticated enough to support such functions as **redlining**, the identification of changes in documents by means of a vertical line drawn in the margin of the change area. Typical word processors that merely automate the document preparation, such as redlining, still require significant text manipulation and creation of multiple documents with redundant information. Newer tools are beginning to emerge in the workplace that will eventually become as important as word processing has been.

One drawback of serial, word-processed text is that ideas that interrelate to many different topics either have to be replicated or cross-referenced in some way. **Hypertext** software eliminates that need by allowing any number of associative relationships to be defined for a given text item. **Hypermedia** extend hypertext to support audio, video, image, graphics, text, and data. In hypermedia, these multiple technologies may all be interrelated and coresident in one environment. In addition, because these tools do not restrict the number of connections an item may have, and because they use mainstream computer technology, application documentation remains on-line and interactively available to all users. Of course, interactive availability also implies a need for hyperlibrary management to control changes to library contents.

TABLE 18-3 Documentation Maintenance Tools

Tool	Vendor	Functions
Folio Views	Folio Provo, UT	Works with Word Perfect to provide multimedia support, highlighting and post-it type document annotation.
Hypertext™	Apple Computer Cupertino, CA	Associative management of text and graphics
MS/Word	Microsoft, Inc. Belleview, WA	Word processing
Word Perfect and Word Perfect Mac with Grammatik	Word Perfect Corp. Orem, UT	Word processing plus grammar checking
Words and Beyond	Lundeen and Associates Alameda, CA	Documentation production including text and graphics

Tools for Reverse Engineering of Software

Reverse engineering tools are rapidly becoming sophisticated enough that the needs for human intervention and extensive training to understand them are diminishing. Several CASE products support reverse engineering through the analysis of code to determine data and process structures that underlie the code (see Table 18-4). Individual programs are analyzed at this point. By the next century, whole applications will be able to be analyzed with intelligent functions pointing out inconsistencies and errors across the old 'spaghetti' code. All tools represented in this section are available in the market and are rated as usefully working products.

Tools for Configuration Management

Configuration management tools, commonly called software libraries or code libraries, have been around since the early 1970s (see Table 18-5). The more sophisticated, newer models make version and variation management simpler by supporting complex functions, such as conditional compilation.

SUMMARY

To increase productivity in the application life cycle and reduce time spent in the code, test and maintenance phases are important. To reduce the effort in these phases, applications should use change control, design for maintenance, use reusable libraries, and use code templates. Object methods are best suited to reusable libraries; nonobject methods are best suited to program templates.

I-CASE is critical in reducing coding and testing through automatic code generation. I-CASE is also required to build intelligence to support reusable designs.

If managing application change, change control procedures and management are critical. Requirements, designs, programs, interfaces, hardware, or purchased software are all subject to change. Change management procedures track requests from initiation through implementation and allow management reporting of cost, types, and impacts of changes.

Logging and management of historical decisions can be useful in volatile environments in which applications are subject to redevelopment. A historical decision log keeps track of arguments, reasoning, and rationales for decisions as they are made.

After an application enters operation, documentation is still subject to change to reflect the current

TABLE 18-4 Reverse Engineering Tools

Tool	Vendor	Functions
ADW/Maintenance Workstation	KnowledgeWare, Inc. Atlanta, GA	Reverse engineering for information engineering: Entity-relationship diagrams Process data flow diagrams
Bachman Series	Bachman Information Systems, Inc. Burlington, MA	Reverse engineering of data structures
Design Recovery	Intersolv, Inc.	Reverse engineering of program structure
Ensemble	Cadre Technologies, Inc. Providence, RI	Reverse engineering charts, metrics, and design
Hindsight	Advanced Software Automation, Inc. Santa Clara, CA	Reverse engineering of C-language code: documentation, structure charts, complexity analysis
RE for IE	Texas Instruments, Inc. with Price Waterhouse Dallas, TX	Reverse engineering for information engineering: Entity-relationship diagrams Process data flow diagrams
Smartsystem	Procase Corp. Santa Clara, CA	Reverse engineering of C-language code: function call graphing, syntax and consistency checking
Via/Renaissance	Viasoft, Inc. Phoenix, AZ	Reverse engineering of data structures

TABLE 18-5 Software Configuration Management Tools

Tool	Vendor	Functions
Copylib	IBM Armonk, NY	Software code library for IBM and compatible mainframes
Data Expeditor	Data Administration, Inc.	Data management software—Allows viewing of file definitions from Librarian, Panvalet, and Copylibs, to locate occurrences and variations of data.
Librarian	Pansophic Systems Lisle, IL	Software code library for IBM and compatible mainframes
Panvalet	Pansophic Systems, Inc. Lisle, IL	Software code library for IBM and compatible mainframes

state of the application. A document table of contents summarizes all changes and the parts of the document affected by each change. Similarly, software documentation is kept in derivations to summarize the actual software and steps used to develop a load module or work unit. Configuration management is the use of software code libraries to manage the official, operational code modules of an application. Delta version and variation management are the principle techniques.

REFERENCES

Babich, Wayne A., *Software Configuration Management: Coordination for Team Productivity*. Reading, MA: 1986.

Baecker, Ronald M., ed., *Groupware and Computer-Supported Cooperative Work: Assisting Human-Human Collaboration*. San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1993.

Collofello, James S., and Jeffrey J. Buck, "Software quality assurance for maintenance," *IEEE Software*, September, 1987, pp. 46–51.

Figlar Consulting, Inc., "Automating the reengineering process," presented to New York City Data Administration Management Association (DAMA), May 21, 1992.

Ingram, Ray, "Application reengineering for productivity, performance, and cost effectiveness," Course Handout, Multi-Soft, December 10, 1991.

Lientz, B. P. and E. B. Swanson, *Software Maintenance Management: A Study of Maintenance of Computer Application Software in 487 Data Processing Organizations*. Reading, MA: Addison-Wesley, 1980.

Martin, James, *Information Engineering, Vol. 3: Design and Construction*. Englewood Cliffs, NJ: Prentice-Hall, 1990.

Nash, Kim S., "Whipping worn-out code into new shape," *Computerworld*, August 17, 1992, p. 69.

BIBLIOGRAPHY

Babich, Wayne A., *Software Configuration Management: Coordination for Team Productivity*. Reading, MA: 1986.
Babich is a recognized authority on the use of different types of libraries for configuration management.

Baecker, Ronald M., ed., *Groupware and Computer-Supported Cooperative Work: Assisting Human-Human Collaboration*. San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1993.
This book reprints groupware articles from periodicals, proceedings, and edited texts that might not otherwise be accessible to a reader.

Lientz, B. P. and E. B. Swanson, *Software Maintenance Management: A Study of Maintenance of Computer Application Software in 487 Data Processing Organizations*. Reading, MA: Addison-Wesley, 1980.
Identifies the applicability of the 80-20 rule in the application life cycle with this study of software maintenance in business organizations.

Mantei, Marilyn, and Ronald M. Baecker, eds., *Proceedings of CSCW '92: Sharing Perspectives*. NY: Association for Computing Machinery, 1992.
This annual conference discusses trends and research in computer-supported cooperative work (CSCW). The proceedings of the most recent conference identify many emerging technologies that will alter the way we work.

KEY TERMS

baseline
boilerplate
change control clerk
change control file
changes
charge-in
charge-out
code library
compile
configuration management
delta
delta file
derivation
frozen specification
hypermedia
hypertext
librarian
linkage edit
load module
media space technology
memory resident work unit
minor modifications
open change request
program template
project librarian
read only module
rebuilding
redlining
reengineering
restructuring
reusability
reusable module library
reusable module
reverse engineering
variations
version

EXERCISES

1. Delta Insurance Company has a policyholder subsystem that is causing them fits. Over the

years, the application evolved from using fixed length, multirecord type files to using a hierarchic database to using relational database. The programs did not change much, but the data structures changed radically. Program code was patched to provide for the new data structure. The amount of people-time allocated to policyholder maintenance grew 15% per year over the last five years and is now costing as much per year as it did in 1980 to develop the original application. No one ever considered reevaluating the subsystem for redevelopment, but they would like to now. Upon inspection, the documentation was found to be up-to-date and includes flow charts and data flow diagrams. There are no current diagrams of the data structure. There are also no historical files of decisions or of changes. What should the company do to get this application in order? What type(s) of maintenance should they consider for the next set of changes?

2. Discuss the ethics of group work tools. If a history is kept, does it violate anyone's privacy? What issues are involved in privacy versus open access to information in group work? Is there a *right* solution to these issues?
3. Discuss the implications of group work tools for global organizations. If you consider cultural differences in, for instance, comfortable distance between acquaintances, how might cultural differences impact the use of group tools? How might companies and cultures need to change to avoid misunderstandings with new tools?

STUDY QUESTIONS

1. Define the following terms:

delta	restructuring
derivation	rebuilding
frozen specification	variation
reengineering	version
reverse engineering	

2. Why is designing for maintenance important?
3. Describe how determining reusability of a module works.
4. How can program templates reduce code created?
5. Which methodologies are best suited for reusable libraries and program templates? Why?
6. What is the significance of I-CASE product recognition of design fragments?
7. Discuss the change management procedure recommended for applications undergoing development.
8. Why is it important to have a baseline product? What happens to a baseline when the product changes?
9. Write a job description for a Change Control Clerk.
10. Describe the life cycle of a change request.
11. What types of reports are useful to managers in tracking maintenance requests?
12. What is the purpose of renaming documents when major changes take place?
13. List the four types of maintenance actions that can be taken. Discuss the reasoning process for deciding which action to take.
14. How is reengineering done in a manual environment?
15. What is a code library? What are the variations in how a code library works?
16. When a delta management system is used, why do you periodically need to create a renamed copy of the code?
17. Describe the contents of a derivation. Why is each item necessary?
18. Compare code versions to variations.
19. What is chargeout and why is it important?
20. What is the purpose of collaborative work tools?

★ EXTRA-CREDIT QUESTIONS

1. Research collaborative work tools and develop a 15-minute presentation to the class about tools on the market, or tools that should be available in the next five years.
2. Get a sample demonstration copy of some emerging software that can be used for configu-

ration management, group work, decision history tracking, and so on. Show the demonstration to the class and spend some time brainstorming about how the product might change work practices.

3. Develop the pros and cons of keeping a decision history. What legal or governmental requirements might impact the decision to keep a historical log? What political and organizational issues impact the decision?

CHAPTER 19

SOFTWARE ENGINEERING AS A CAREER

INTRODUCTION

In every student's path lies a career they will pursue. Nowhere are there as many varied opportunities as in information technology related professions. This chapter examines possible career paths for achieving software engineer status, maintaining job skills, and planning for your next job. After you have identified your own job requirements, we show one way to determine the likelihood of your job search success and a way to determine when you need to broaden your job requirements.

EMERGING CAREER PATHS

Software engineering used to be thought of as the province of computer scientists. Over the years, computer scientists tended to migrate into scientific and defense programming, operating systems support, and software package development. In those areas, they applied engineering methods to designing and developing efficient and effective software. In contrast, business organizations used the term *systems analyst* to describe the person who applied computer skills to the development of business transaction processing applications. Computer scientists tended to build one-of, real-time applications while information systems (IS) specialists tended to build batch business transaction applications. As IS moved to on-line applications, the technology gap that somewhat fueled the split between the disciplines got smaller.

Computer science (CS) SEs increasingly study the same topics as IS SEs. The term *systems analyst* is giving way to the term *software engineer* as engineering techniques increasingly are used in business application development. The differences between the two groups are mainly in the emphasis on *technology* for CS and on *application* of technology in *business* for IS. The CS majors still tend to work in the traditional CS industries—defense, scientific organizations, and software development firms. The IS majors still tend to work in finance, manufacturing, government, and retail.

As teaching emphasis moves away from the 'one right way' approach to an ever growing set of theories from which we choose the most appropriate, CS and IS will converge even more. The two groups probably will not be melded completely, however. There is a need for both types of training that will continue to grow throughout the 20th century. The goal of both programs is to teach theories and approaches to problem solving with ways to apply them that prepare you for continuous change in the IS body of knowledge.

For the last decade, the radical changes in applications development coupled with changes in the types of applications businesses build are resulting in a split of duties in the development environment. The first type of career is more technical. This SE will build ever more complex state-of-the-art applications using new technologies. The second type of career is less technical. These SEs work as liaisons to user departments and act as *chauffeurs* for computer usage to assist users who are not inclined to become computer literate themselves. Within a generation, most business people will be computer literate, and these jobs will evolve to developing and managing DSS and EIS for managerial staff.

The issue over whether to get a degree in CS or IS is not too important from an employability perspective. There are careers for both types. Both types are useful and valuable to adding to our store of knowledge about how to build applications. In this chapter, first job levels and types of jobs available are defined. Then, an approach to defining a first job (or next job if you are already employed) is developed. Finally, means to maintaining your competence in the ever-changing world of IS and information technologies are presented.

CAREERS IN INFORMATION SYSTEMS

Job opportunities in information systems can be classified by level and job type. Job levels are generally classified as junior, intermediate, senior, lead, technical specialist, and manager. Each level is defined in terms of how much supervision is provided at the level and how much information and expertise the individual is expected to possess. Job type identifies the nature of the work performed.

Level of Experience

In this section, we discuss the job levels to which you might aspire. The levels are junior, intermediate, senior, lead, technical specialist, and manager. When times in a level and starting years of experience are mentioned in each section, they imply years of different, changing experiences. Many people simply do the same thing over and over; this is not gaining experience.

Junior

A **junior** staff member is directly supervised, but is expected to work on his or her own on some aspects of a job. This is an entry-level position. Juniors are expected to have basic skills and ability to find information to enhance skills. They are in a learning mode most of the time. The time you might expect to perform in a junior-level position is about two years.

Intermediate

An **intermediate** staff member works independently most of the time, requiring direction on some activities. A mid-level person possesses a range of skills and experience but is still in a learning mode much of the time. Starting intermediate people have two to four years of experience. The average time at the intermediate level is from two to five years.

Senior

Seniors work unsupervised most of the time; they possess a wide range of both job and technical experience that is used to train and aid others. Senior-level staff supervise others, depending on the size and complexity of the project. Frequently, senior-level jobs are generally a prerequisite to lead or specialist titles.

A starting senior-level staff member has from five to seven years of experience. Expect to stay at this level at least three years. Many people end their careers at this level and stay on related projects throughout, becoming expert in both a technology and an application type.

Lead

A **lead** person works on his or her own, performing all levels of supervision. A lead person might also be called a *project leader*. Project leaders are a step above seniors and aspire to managerial positions. The lead skill levels are similar to seniors,

but a lead person has more managerial/supervisory responsibility.

A lead person might end a career at this level, becoming totally responsible for small projects but never reaching a managerial level in charge of multiple projects.

Technical Specialist

A **technical specialist** is a person who has extensive experience in a number of different areas. The integration skills needed to develop distributed database networked applications exemplify the expertise of such a person; the skills of an integration specialist include application development, networking, database, and operating systems. The specialist is at the same level as a manager, having many of the same duties and capabilities without the personnel and budget responsibilities of a project manager. Specialists typically have been in IS positions for 10 years or more and might remain at the specialist level until retirement.

Manager

Managers work independently, performing personnel evaluation, budgeting, progress reporting, and managing projects. Managers may or may not be technical in orientation; they have a wide range of job experience and mostly managerial responsibility. For technical managers the distinguishing features of their jobs are the planning, budgeting, monitoring, personnel management, and liaison activities discussed in Chapter 3.

The levels are shown with logical career moves from junior through manager in Figure 19-1. As the figure shows, there is little choice in level movement for junior through senior positions. Once someone is fully knowledgeable about several types of jobs, they can choose to remain technical and become a technical specialist, or to move into management, usually becoming a project leader, then manager. Keep in mind that this career ladder identifies only level of expertise, not area. Movement between job types is possible at all levels and often is required to

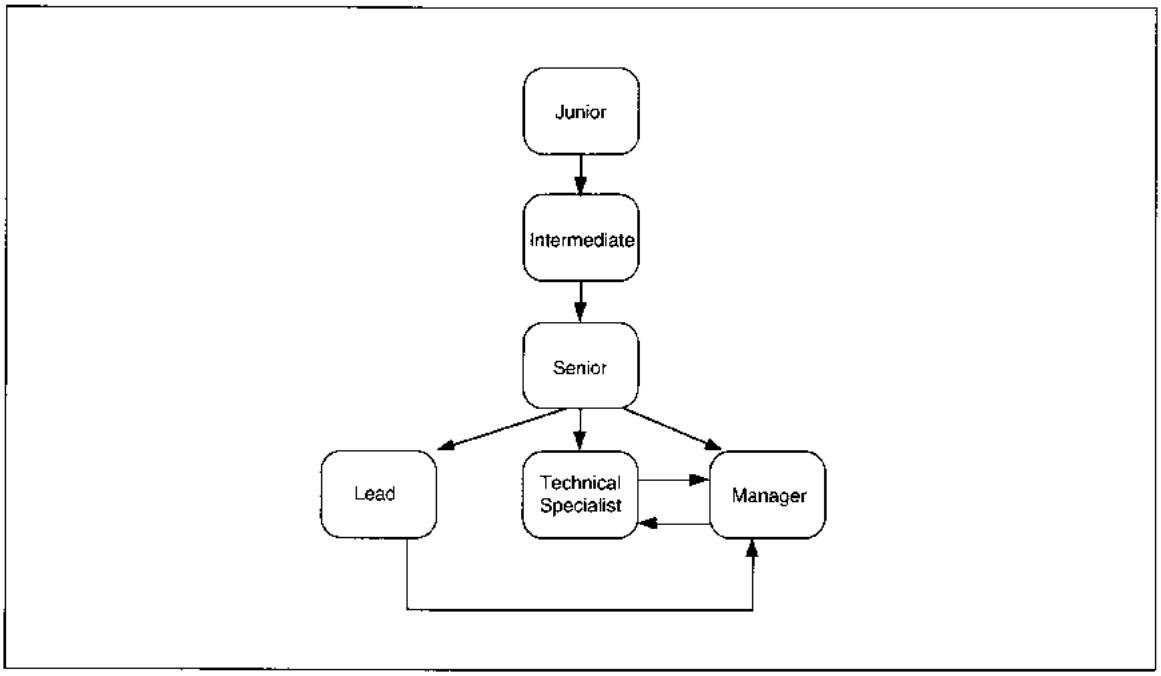


FIGURE 19-1 Career Path for Different Levels of Jobs

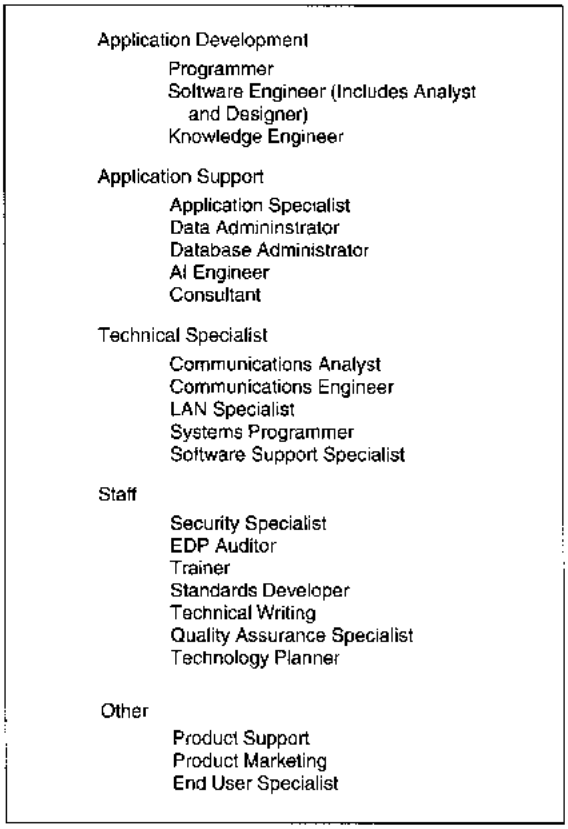


FIGURE 19-2 Summary of IS Jobs

move to specialist and lead positions. Job type definitions are in the next section.

Job Type

Within a given level of experience, job type identifies the job content and nature. Job types are discussed in terms of the areas of specialization: application development, application support, technical specialties, staff positions, and others. The jobs are summarized in Figure 19-2. Keep in mind that these are representative of the specialities in large organizations; the smaller the organization, the more likely multiple skills are required of individual staff members.

Application Development

The main application development jobs are programmer, software engineer, and knowledge engineer. Keep in mind that there are entry-level positions all the way through technical specialist positions in many of these jobs. There is great variety across development jobs depending on the hardware and software environments. Hardware platforms include personal computers, workstations, and mainframes as well as equipment for communications, robotics, process control, office automation, imaging, and microforms. In addition, application environments are increasingly diverse. The software environment might include database, communications, programming language, hypermedia management, computer-aided software engineering (CASE), fourth generation languages, and expert system shells, just to name a few. With this diversity in mind, we discuss application development jobs.

PROGRAMMER. **Programmers** translate design specifications into code modules that they design and unit test themselves. Programmers might rotate duties between development and maintenance applications.

Senior programmers perform other duties besides programming. For instance, they participate in analysis, design, or testing activities for the entire application.

Beginning programmers specialize in one language, while more senior programmers are conversant and experienced in multiple languages. The main generations of languages that apply here include

2GL—Assembler
3GL—COBOL, Fortran, Pascal, Ada, C, C++
4GL—Focus, Lotus, Paradox, dBase, Oracle, SQL
5GL—Lisp, PROLOG.

SOFTWARE ENGINEER. An **SE** performs the functions of analysts, designers, and programmers. **Analysts** define and document functional requirements of applications. Senior analysts also participate in organizational-level IS planning and feasibility studies. **Designers** translate functional

requirements into physical requirements of applications. These traditional titles still exist and frequently are combined in the title *analyst*. Programmers develop and test code modules as discussed above. SEs may do all three—analysis, design, and programming—as well as acting as project leader or project manager, as needed. The differences are in job emphases. A junior SE would spend most of the time programming, while a senior SE would concentrate more on planning, feasibility, analysis, and design.

KNOWLEDGE ENGINEER. **Knowledge engineers** elicit thinking patterns from experts for building expert and artificial intelligence systems. Knowledge engineers are similar in status to SEs, but have specialized skills applying to AI problems. Developing models and programs of knowledge structures requires observation, protocol analysis, in-depth interviewing skills, the ability to abstract in areas that are not areas of personal expertise to make sense of reasoning and information needs, and the capability to develop uncertainty predictions about the information and its accuracy with experts.

Application Support

Application developers require expertise from a number of different specialties in developing even the most routine applications. The jobs that most often support application development include application specialist, data administration, database administrator, artificial intelligence engineer, and consultant. These jobs are not all distinct and may overlap with each other in many organizations; the areas of overlap are most noticeable for consultants who may do all of these specialties. This overlap is ignored for the moment for purposes of defining the essential skills of these support functions.

APPLICATION SPECIALIST. **Application specialists** have the problem domain expertise that allows them to consult to project teams for specific types of applications. For instance, a senior analyst in real-time money transfer might split time between domestic and international money transfer projects, overseeing compliance with all the rules and regulations of the Federal Reserve Bank as well as the various money transfer organizations, (e.g., BankWire, Swift, NYCHA, etc.)

Frequently applications specialists are members of external standards setting organizations. In this capacity, the specialist is a liaison between his or her company and other companies in the industry. Standards are set by consensus development of what should be done and how to do it. The standards get highly detailed, for instance, specifying the number of characters in a header of a bank wire message and the meaning of each character. The major skills needed for this type job are communications-oriented diplomacy, technical application, and problem domain knowledge.

DATA ADMINISTRATION. **Data administrators (DA)** manage information as a corporate resource. In this capacity, data administrators help users define all data used in the company, identifying the data that are critical to the company's functioning. DAs establish and maintain standards and dictionaries for corporate data. These on-line dictionaries, or repositories, are used by on-line 'help' software to provide users with data definitions as they are using a computer.

Once data are defined, a DA works to define and structure subject databases for use by applications. They also track application use of data. For new project development, DAs work with the application developers to locate data that is already automated, and with DBA staff to provide the application group easy access to automated databases.

DATABASE ADMINISTRATOR. **Database administrators** (DBA) manage the physical data environment of an organization. DBAs analyze, design, build, and maintain databases and the software database environment. Working with DA definitions of data, DBAs define physical databases and load actual information into them.

A DBA works with application development teams to provide access to already automated data, and to define the specific database needs for information to be automated.

ARTIFICIAL INTELLIGENCE ENGINEER. **Artificial Intelligence (AI) engineers** work as consultants to project teams to define, design, and implement intelligence in applications. At present, AI is in its infancy and its use in applications is sparse. Most AI work takes place as part of an expert system development. AI engineers work with knowledge engineers to translate and test problem domain data and reasoning information in a specific AI language, such as Lisp. As AI matures and its use increases, this position may move from a support location to application development location in organizations.

AI engineers have attained a higher level of expertise than KEs. As AI experts, they participate in software and hardware surveillance, evaluation, planning, and implementation on a company-wide basis. As experts, they are usually involved in hiring decisions for other AI experts and KEs.

CONSULTANT. **Consultants** are jacks-of-all-trades and practitioners of all. The higher the number of years experience, the greater the knowledge is expected to be. The areas of expertise would likely include several of the job types discussed in this section.

Consultants are hired most often to supplement staff or to provide exotic skills not available in-house. When hired because of exotic skills, they frequently train the in-house staff during the work engagement. Consultants are expected to have specifically identified skills when they are hired, and to apply those skills in performing the consulting engagement.

Consultants are sometimes preferred to permanent hires because they get no benefits and do not require raises from the hiring organization; they already have the desired skills and need no career path planning; they have their own managers and require less personnel-type management. Consultants are easier to hire and fire than full-time staff, too.

Technical Specialists

Other technical specialties are common in organizations but do not always interact with application developers on a regular basis. Some of these specialties include communications analysts and engineers, LAN specialists, systems programmers, and software support specialists.

COMMUNICATIONS ANALYSTS AND ENGINEERS. **Communications analysts and engineers** analyze, design, negotiate, and/or install communications-related equipment and software. They are required to be fully conversant with communications technologies and may work on mainframe or PC-based communication networks.

Integration of voice, data, graphic, and video signals via telecommunications networks is growing in importance to every organization. Certainly, integration of data and voice is commonplace. As the integration levels of information delivery increase, this specialty becomes crucial to organizational success.

To start in communications at an entry-level position, educational background might be in electronics, engineering, applications, computer science, or telecommunications. To transfer into a communications-related position requires intelligent positioning and career planning once you are within the company.

LAN SPECIALISTS. **Local Area Network (LAN) specialists** plan, oversee installation, manage, and maintain local area network capabilities. There is no essential difference between a LAN specialist and a communications specialist except *scale*. A communications specialist works with multiple networks including mainframes; a LAN specialist works on geographically limited networks that are comprised of personal computers (PCs).

The educational background can be in IS or CS with a concentration in telecommunications. In addition, many LAN specialists have certification by a vendor, such as Novell, which certifies its engineers as having basic knowledge as a *Certified Novell Engineer* (CNE).[2]

LAN administrator is an entry-level position in many companies. A LAN administrator creates new

2 Certified Novell Engineer™ is a trademark of the Novell Corporation, Provo, Utah.

users, implements or changes security levels and codes, installs new versions of the LAN operating software, installs new versions of database or other LAN-based software, oversees the resources provided through the LAN, provides backup and recovery capabilities to the LAN, and manages the LAN configuration. Troubleshooting the LAN when problems arise is a valuable skill that frequently qualifies the individual for increasing responsibility beyond an entry level position.

SYSTEMS PROGRAMMER. **Systems programmers** install and maintain operating system and application support software used in mainframe installations. For instance, an IBM 309x class mainframe machine contains several million lines of code in its operating system (OS). At any given time, 50–100 'bugs' might be outstanding and need to be fixed. 'Fixes' are 'patched' into the operating system software until a new level of the operating system is released. If no problems occur in your installation, the fixes are not needed. Evaluating the new features and whether they are necessary at the time is a skill system programmers develop. Monitoring all of the hundreds of applications to determine whether their problems relate to OS problems is a major task. In addition, applying a fix might cause another problem, so the systems programmer needs to be fully conversant with normal operations to determine any ripple effects.

SOFTWARE SUPPORT SPECIALIST. Application software support is a similar, but different, type of system programming. **Software support specialists** install and maintain software packages used by both applications developers and by users. Database, query language, backup and recovery, spreadsheet, disk space management, telecommunications interface, and any other nonoperating system software are in this category.

Application software support programmers and specialists work with application developers and with technology surveillance staff to define the needs of the organization. Then, they work with vendors to obtain and install the product. Finally, they maintain the product on an on-going basis, providing the application development staff with usage support for the product.

System software support (SSS) programmers and specialists work with systems programmers to maintain the software provided as a shared resource for others in the company to use. For instance, in a LAN environment, an SQL Server might be used. The SQL Server software is supported by an SSS person, while the network operating system (NOS) is supported by a systems programmer.

Staff Positions

Most organizations have one or more persons performing these functions, even if they do not have a title to go along with the duties. The tasks that are most often given titles include security specialist, EDP audit, training, standards and technical writing, quality assurance, and technology planning.

SECURITY SPECIALIST. A **security specialist** is responsible for security and for disaster recovery readiness. For security, a specialist establishes standards for data security, assists project teams in determining their security requirements, and establishes standards for data center security. Similarly, for disaster recovery, the security specialist assists managers and project teams in identifying critical data needs of the organization. Then, the specialist assists data centers and project teams in developing and testing disaster recovery plans. This is a valuable specialization that is most often found in large organizations but is needed in all companies.

Research by IBM and others has shown that companies without any backup and recovery plan *will* go out of business in event of a disaster. The studies looked at different geographic areas, different types of disasters, and spanned several years. The result was always the same. If a company could not recreate the data critical to its continuing in business, it could not survive a disaster.

Most disasters are from weather (tornados, hurricanes, and earthquakes), but they can also include acts of terrorism, fires, and other nonweather means of losing a data center. In addition to loss of a data center, security specialists plan for less severe losses,

such as loss of disk drives or malicious tampering with data.

EDP AUDIT. **EDP auditors** perform accountability audits on application designs. Any application that maintains legal obligations, fiduciary responsibilities, or books of the company, *must* be able to recreate any transaction and trace its processing. EDP auditors ensure that company exposure to losses or law suits is minimized through good application design. The design aspects evaluated by auditors are audit trails, recoverability, and security.

TRAINING. A **technical trainer** learns new technologies, vendor products, new language features, and so on, then teaches their use to others in the organization. Training might be done within a company, or in a specialized training company, or as a consultant in a short-term assignment.

Training is often considered a temporary or rotational assignment for people whose career path or job assignments allow them to perform a staff function for some period. The thinking is that training is more easily related to current job assignments in an organization when it is done by someone who is holding, or has recently held, such an assignment. Teaching forces the trainer to organize thoughts, make presentations, answer questions, and develop good communication skills. Therefore, training assignments are one way to allow someone who is a valued employee, but who lacks good communication skills, to develop and practice those skills in a work setting that is not too threatening.

STANDARDS AND TECHNICAL WRITER. **Standards developers** work with managers to define what aspects of work they want to standardize, and to formalize the requirements into standard policies and procedures for the organization. The most important skills for standards developers are verbal and written communications.

Company standards vary in level of detail and breadth of activities covered. Some companies standardize their complete methodology, providing minute detail on all of the steps to developing a project, guidelines on the tasks performed, required signatures and approvals for project work, detailed lists of liaison departments that must be consulted, and so on. Other companies provide loose guidelines with checklists to be consulted to ensure that all needed tasks are considered for inclusion in the project's work plan. Both types require the ability to run meetings, obtain the standards' requirements, negotiate between managers, and write accurate descriptions of desired rules.

Standards development and technical writing are related activities. A **technical writer** takes information about software products, applications, or other information technology products and develops documentation to describe their features, functions, and use. A technical writer needs to have good technical and nontechnical communication skills. The writer uses the technical communication skills in talking with and developing an understanding of the product being documented. He or she uses the nontechnical communication skills in writing about the products for a user audience.

QUALITY ASSURANCE. **Quality assurance** is an IS function that performs quality audits on application feasibility, analyses, designs, programs, test plans, documentation, and implementations. QA is usually functionally separate from the development groups it is auditing; however, in a small company, QA may be an analyst's, or SE's, temporary assignment.

The form of the audit differs by the product being reviewed. A QA analyst is assigned to a development project as it is initiated. He or she has little involvement until the first work products from the development team are available. Then, as documents become available, the QA analyst reviews them for consistency, completeness, accuracy, and feasibility. Any problems found during the review are documented in a memo to the project manager. The problems must be responded to by either explaining why the issue is not a problem or by correcting the erroneous item.

As you can see from the description of this task, QA is a natural adversary to application developers since the QA analyst's job is to find fault with the work of the project team. QA work is usually assigned to senior staff who are respected enough to be listened to and tactful enough not to cause revolts

by the project teams. QA analysts need senior technical, communication, and problem domain skills to perform a quality review. They need experience in all aspects of project development in order to know how it should be done and where problems might arise. At the same time, tact and skill at identifying only critical issues is important. No one likes to be told publicly they have made a mistake, even though they might know intellectually that the project work will benefit from the criticism. The QA analyst needs to be sensitive to both the politics and the problems identified.

TECHNOLOGY PLANNING. **Technology surveillance specialists** monitor technology developments to identify trends, select technologies that are appropriate for experimentation in their organization and, eventually, champion the implementation of new technologies in the organization. These senior staff are liaisons to the outside world and vendor community for the company. Junior-level staff in technology planning might work with a senior person who guides the work, while the junior person does some coordination and technology monitoring.

Other

Numerous other positions relating to ITs and IS development are available for students of IS. Some of these include product support, product marketing, and end-user specialist.

PRODUCT SUPPORT. **Product support staff** work for an end-user group or vendor to provide product-related technical expertise or other "hotline" support. In addition to technical knowledge about the product(s) supported, the individuals in this job require excellent phone skills and must be able to talk nonjargon language to users with problems.

PRODUCT MARKETING. **Marketing support staff** work for vendors to provide technical information to sales representatives in marketing situations. This type of job requires excellent communication and people skills, with some knowledge of marketing tactics, such as narrowing focus of conversation and closing techniques, to effectively work with a sales representative. All software, hardware, and consulting companies have people to perform these functions. Usually, this job is for senior-level people, but if you have a particular area of expertise and support in that area is needed, then you might qualify for such a job without being a senior staff person.

END-USER SPECIALIST. **End-user specialists** translate user requirements into technical language for developers to use. In some companies this is the function of the systems analyst or SE. In other companies, there are end-user liaisons in the user departments to perform this function.

In summary, every company needs many different combinations of job characteristics in all departments of the organization. The challenge to graduates is to decide which aspect of the work fascinates you most. The career is there for the making. To further your chances of a successful entry into the job market, your undergraduate courses should concentrate on *core knowledge* of application development, programming, database, and telecommunications. Then concentrate elective courses in one or more specialties from the above array of jobs.

PLANNING A CAREER

Defining your next job is the first step to determining what to ask for when you talk to personnel recruiters. You must have a goal that is fairly well defined yet realistic for the job market you wish to enter. Once you begin work, you need to know how to plan the next job, and so on. Also, one degree and job in IS does not qualify as a 'career.' Rather, continued growth and development in depth and in breadth of knowledge is required. In this section, we discuss how to plan your first job and extrapolate from that to plan your career. In the next section, we discuss how you keep current to continue to grow as a professional SE. As you read through this section, assess your job wants. The more honest you are about your skills and desires, the more useful you will find this exercise.

Decide on Your Objective

The first activity is to decide on an objective or goal. Where do you want to be in five years? Try to be as specific as possible. Do you want to be making $60,000 a year? Do you want to have a title of Project Manager? Do you want to be a specialist in software engineering? Your objective might be money-related, title-related, or job content-related, or all three, or something else.

Make sure your objective relates to job criteria. For instance, if your objective is to own a house, decide how much you anticipate spending, then translate that into a salary. Once you have identified an objective, use the following sections to determine the company and job characteristics that are most likely to help you meet your goal. Try to translate the money into a position and title, working backward to identify a starting job. If your goal is title- or job-related, use the following sections to identify the most likely tasks, job characteristics, and companies to help you meet your goal.

Define Duties You Like to Perform

Once you have a tentative goal, begin to think about how to reach that goal through one or more jobs during the five-year period. What are likely starting jobs? How do those starting jobs relate to you? In performing this evaluation, you need to do an honest assessment of duties you like to perform. Evaluate the list below, making your own list of tasks and placing a percentage next to each item you are interested in doing in your next job. Make sure that all of the percentages add to 100%.

- Programming (i.e., new development and maintenance)
- Analysis
- Testing, Quality Assurance
- Technology Surveillance
- Consulting
- User or Technical Training
- User, Help Line, or Product Support
- Standards Development
- Technical Writing
- DBA or other specialized technical position, and so on

Keep in mind that, while this exercise is to find your ideal next job, the work tasks should also be realistic. About 50–70% of newly minted undergraduates begin as programmers. Another 10–15% begin as LAN managers, with an equal percentage beginning as programmer-analysts or SEs. A few begin as technical writers, help line support, and trainers.

Define Features of the Job

After job tasks are identified, evaluate the external features of jobs you prefer. There are two types of job features you should define: technical and non-technical. The technical features are what this text is all about. Choose from the following list those characteristics that appeal to you.

Project type—Maintenance, development, or a mix
Technology type—Mature, or state of the art, or experimental, leading edge
Type position—Project, staff, operations, sales, support, or other
Phases of project work—Planning, feasibility analysis, design, maintenance, programming, or all
Methodology—Process, data, object, semantic
Hardware platform—Mainframe, micro, workstation
Technologies—DBMS, language(s), package(s), CASE tools, LAN

Be as specific as you can in defining each of these job components. This information is used to select target-specific jobs for your job search. Be equally specific about job functions you do *not* want to learn, if there are any. The nice thing about defining the ideal job for yourself is that there is no wrong answer, only ones that fit you better than others.

Next, assess the type of duties you want to perform. Do you want narrowly-defined, specific assignments, or broadly-scoped and less well-defined

assignments? In general, the larger the company, the more esoteric and specific your requirements *can* be, but there is no standard. Also, in general, the smaller the company, the more casual and broader the assignments. This means that a person defined as a programmer might have entirely different time allocations depending on the size of the company. In a large company, a programmer will spend 40–60% of his or her time coding and unit testing program specifications developed by an SE or designer. Remaining time is spent in nonproject work such as reading manuals, attending meetings, learning, and communicating about the work. In a small company, a programmer is likely to spend 20–40% of his or her time developing the specifications with the analyst or SE, 20–40% programming and unit testing, and the remaining time in other activities. Which scenario do you prefer? The larger the company, the more specialized and the narrower the job. Also, the larger the company, the more likely you will be paired with a senior *mentor* who is responsible for monitoring your progress.

Think about how you like to learn new things. Do you like to be given a book and an assignment for completion? Or do you prefer to attend classes and have someone to answer your questions? The first learning approach is one used most by consulting and smaller companies. The classroom approach is used more by large companies.

Next, evaluate nontechnical features of a job, including title, salary, working hours, autonomy, and travel. Title is a more important issue in some industries than others. For instance, in manufacturing institutions, being an 'officer' of the company is significant. But in a bank, about 25% of the staff will be officers. Of this 25%, 60% will be assistant treasurers, or the lowest level officer; 25% will be second vice presidents; 10% will be some level of vice president; and the remaining 5% are executive vice presidents or higher. The titles are more for external prestige and to compensate for low pay than anything else. If title is important, then, financial services and consulting are the most status-conscious of the industries listed. In contrast, a private consulting company might have two to five *principals* and 200–300 *consultants*, and those are the only two titles.

What salary would you like to be making in five years? Target the five-year time frame because your first salary is relatively inelastic if you are not already working in IS. By inelastic, we mean that the salary range for new, inexperienced hires is relatively narrow: $28,000–$34,000 for undergraduate IS degrees, and $30,000–$38,000 for graduate IS degrees, in 1994; and the salaries are relatively invariant across industries.

Take the midpoint of the range that describes your situation and assess the ideal raises you might receive to derive your salary in five years. If you expect to double your salary in five years, you need a compounded growth of about 15% annually to meet that goal. You might get 15% raises in consulting, but it is unlikely anywhere else. Realistically, companies give regular raises that keep a third of all salaries even with inflation. If you are in the top third, you might qualify for merit increases which might be 2–4% over the inflation rate. If you want a six-figure income within five years, then you are either thinking of your own company, or are a genius, or are unrealistic. It is nice to dream, but thinking of salaries requires hard reality.

The next nontechnical issue is the number of hours you want to work. This is an ideal that you might never actually reach, but each industry has different intrinsic demands about hours of work that should be considered. The normal work week is 40 hours in the United States. This time is spent from Monday to Friday with few organizations requiring weekend work.

In addition to the number of hours, *which* hours might also be important. There are two issues here: flextime and shift work. Can you get up and maintain a schedule that requires you to be in an office at a fixed time every day? What if the hours are 7 A.M. to 4 P.M.? How about 9 A.M. to 6 P.M.? If a company has flextime, you choose the time of your arrival, within limits, and work a regular seven- to eight-hour day once you are at work. Most companies in large metropolitan areas use flextime to cope with the vagaries of traffic and transportation problems.

You might consider a job in an industry that works at night. Do you mind shift work? Can you cope with a schedule that requires you to sleep during the day? Keep in mind that you might be a night

owl at college, but all of your friends will probably get day jobs. Will night work shut you off from your social life? How important is that to you?

The last time issue is overtime. Do you mind overtime? How often is overtime acceptable? Could you work for a company that expected a 60-hour week even though the advertised required number of hours is 40? Can you deal with midnight phone calls when you are 'on call' for application problems? Some companies will tell you that you are expected to work until the job is done, and if that means overtime, then you work overtime. Can you live with such an agreement? If not, what are your time requirements for work? If you cannot deal with any overtime, you need to search for a low pressure, staff job or a maintenance job that requires little or no overtime. If you can deal with overtime, then all jobs are open for you. The longest hours are usually in consulting, but most development projects in most companies end up requiring some overtime work.

Next, consider the extent to which you want to work autonomously. As an entry-level person, you most likely will be coupled with a senior person who would be responsible for helping you with problems, bugs, or other issues you are not sure how to deal with. But each company has its own levels of autonomy that its employees are allowed. Do you want leeway in figuring out your own answers or do you want close supervision, at least for a while? In general, the smaller the company, the more autonomy you will be given, and the greater the breadth of the jobs you will be assigned. If you like working alone, then select a smaller company.

Finally, consider the amount of travel you want as part of your job. Be realistic. Travel is demanding, rewarding, and wearing. It requires extreme organization because once the plane leaves you cannot return to the office for that forgotten piece of paper. It also demands family and personal sacrifices because you are frequently on a plane during birthdays and holidays. You may find that you want to travel for awhile and cut back after a few years. After all, someone else *is* paying the bills. That is also an acceptable scenario, just be prepared for the action when it arises. Several industries, especially consulting, require significant travel and frequent temporary relocation for project work. You might need to leave for months at one or two days' notice in this environment. The rewards are commensurate with the sacrifices: The pay in consulting is the highest after successful entrepreneurship.

Define Features of the Organization

Even though this section is for defining features of the organization, you are still assessing your needs in a job. In this section, you assess how 'hard' you want to work, how 'smart' you want to work, and how much ambiguity and stress you can cope with. To some extent you have already answered some of these questions; they have not been phrased in just this way.

When you define how many hours a week, and what type of work you desire, you are, to some extent, answering the 'hard' and 'smart' questions. Several different hierarchies of organizations can be developed for you to position yourself in different industries and different company types. The first hierarchy is based on industry. Based on several different salary surveys over the last 10 years, a hierarchy of industries in average salary order is shown in Figure 19-3. This hierarchy shows that you are most likely to make the most money owning your own company, and are most likely to make the least money working in academia or nonprofit organizations. This hierarchy also translates into a 'hard' work hierarchy. The amount of time and personal sacrifice expected of employees is directly proportional to the amount of money paid. That is, the companies that pay the best expect the most. If you cannot stand stress and long work days, then remove ownership and consulting from your list. If you want the least possible stress and least possible work, target your search in nonprofit, retail, government, and academic organizations.

Keep in mind that these are general rules of thumb at work here. All companies have positions of all types. The generalizations drawn here identify the majority of positions.

A second hierarchy can be developed based on the position of a given company within its industry. Figure 19-4 shows one industry, soft drink

Highest-to-Lowest Salary Industries**	Example
Your own company	
Consulting	
Big 4 Accounting Firm	Ernst & Young, Arthur Anderson
Large IS Consulting	Cap Gemini (CGA)
Internal Consulting in Large Company	
Private Consulting Company	
Vendors	Novell, Microsoft, ATT, Pacific Bell, Bell Labs
Conglomerate Headquarters	Boeing, Mobil
Financial Services and Insurance	American Express, Citibank, Prudential
Government, Transportation, Utilities	U.S. Department of Agriculture, American Airlines, Brooklyn Union Gas
Manufacturing	Whirlpool, Babcock & Wilcox
Retail, Publishing, Medical	Macy's, Any large metropolitan hospital
Nonprofit, Small business of any type	United Way
Education	School Districts, High Schools, Colleges, Universities

**Based on numerous articles in *Computerworld*, *Datamation*, *Wall Street Journal*, *Dallas Morning News* and *The New York Times*.

FIGURE 19-3 Salary-Based Hierarchy of Industries

Largest to Smallest Industry Position:

Coca-Cola
PepsiCo
Dr. Pepper/7-Up
Shasta
Snapple
Others

FIGURE 19-4 Industry Position for Soft Drink Companies

manufacturing, with the major contenders. As the figure shows, Coca-Cola is closely followed by PepsiCo, Dr. Pepper/7-Up, and all others. This industry is fiercely competitive and marketing driven. To be in this industry is to be competitive. Therefore, when you select an industry, try to think of a characterization for the industry and how it fits your personality.

Next, try to match your personality to the company style. Do you want to work for the leading company and be the one to beat? Or do you want to work harder at #2 which is trying to become #1? Or are you more comfortable being at some other level company with less stress? There is nothing wrong with working at any of the levels. The idea is to choose the one that fits you best.

Keep in mind that all of these statements about companies are generalizations. Many companies are not even close to the top of their industries but are in a turnaround position that requires maximum effort from everyone. Such turnaround companies are sometimes the best of places to work and sometimes are the worst of places to work. Similarly, a large, longtime company that is first in its industry might be ready to take a fall. IBM, in 1990–1994 was not a fun place to work.

We identify industry leaders because they are generally more innovative than other companies and have more money to spend (and spend it) on new technologies. Not all is positive for large industry leaders. In some cases, the larger and more leading the company, the slower to promote people to new positions and the more likely to be results-oriented without being people-oriented. Also, not all companies, regardless of industry position, recognize the importance of information technologies to meeting their mission. Ideally, you want to find a company that has a culture that is compatible with your personality, that is as people-oriented as you need, that recognizes the importance of information technolo-

gies, and that will help you reach your personal goals.

Finally, if you have prior experience in some industry, try to leverage that knowledge. Problem domain expertise takes two to four years to learn. If you already have experience and can target IS jobs in your old field, your starting salary should be 5–10% higher than new employees in the same industry.

Define Geographic Location

Next, consider the ideal geographic location for you. You may want to stay near where you are from. That is perfectly reasonable. If you want to live somewhere else because of weather, life-style, or some other criteria, now is the time to choose where you want to live and work.

In the United States, there has been a 30-year migration toward the southern half of the country, but the jobs have not always followed. According to salary surveys covering 1992–1993, the best paying and highest number of jobs are in Alaska. Both New York City and California, traditionally high growth, high-income areas, follow Alaska. Other large, diversified-industry, metropolitan areas also top the list (see Figure 19-5).

The lowest paying and lowest number of positions are in the South and Southeast, particularly Florida.[2] The center of the country has not faired so well either. In 1992, St. Louis and Philadelphia graced the bottom of the salary list.[3]

Define Future-Oriented Job Components

The last job-related components relate to job security, benefits, and speed of advancement. You won't use these until you are interviewing, but it is a good idea to have some goals in mind for these job components when selecting companies and industries. Also, if you are looking for security in a volatile industry, like stock brokerage finance, then you need to reassess your requirements to align more closely with reality.

Highest Salary Locations:

Alaska, New York metro area, California,
Dallas–Fort Worth, Minneapolis–St. Paul
Chicago, Denver
Boston

Lowest Salary Locations:

St. Louis
Last: Southeast and South

Based on Robert Half, International 1992 and 1993 Salary guides and articles in *Datamation*, *Computerworld* and *The New York Times*

FIGURE 19-5 IS Salary by Location in the United States

Security relates to the stability of the industry. For over 50 years, the United States had relative stability in industry, with only companies that had fallen on hard times resorting to layoffs. Many companies (e.g., Chase Manhattan Bank and IBM) used to brag that they had never had a layoff in the company's history. The late 1980s and early 1990s changed all that. The recession during the early 1990s was deeper and longer than many since the Great Depression of 1929, and had the added problem of being worldwide in scope. Virtually every company over $100 million in sales went through some reassessment of company structure and size, laying off and eliminating millions of jobs. As we slowly recover from that period, stability is an issue on which we all share concern.

Financial success is one indicator of likely stability. Companies that have higher percentages of net income and profits compared to competitors are more likely to be stable. But, at the moment, there are no guarantees. If security is very important to you, target companies that are successful relative to their competition, regardless of the industry, and target companies in relatively inflation-proof industries, such as office products.

Benefits include vacation, retirement, medical support, dental support, child support, aging parent

2 Based on Robert Half *1992 Salary Guide*, and *1993 Salary Guide*, San Francisco, CA: Robert Half International, Inc.

3 *Computerworld* publishes an "Industry Snapshot" highlighting hiring trends in a specific industry in each weekly issue.

support, and so on. The average starting benefits include two weeks' vacation after one year, with some medical and dental support. Retirement benefits are in a state of flux. In 1993, most large companies still offer retirement benefits, but the vesting period (that is, the time at which the money becomes legally yours), varies considerably. If you plan to stay with a company a long time, vesting periods are moot. If you foresee some movement between companies in your future, the vesting period becomes important to your consideration of how long you might be tied to a specific company.

The more progressive and larger the company, the more likely they are to also have programs providing some type of support for child or parent care. Decide how important these benefits are to you and keep this information in mind when you are evaluating companies. When you begin interviewing, use your ideal benefits and security needs as one criteria to separate the companies you are interested in from those you are not.

Speed of advancement may be an important factor to you. Do you expect to be promoted every year, assuming that you have exceeded all job requirements? Some companies have average time in grade figures that they might share with you during the interviewing process. In general, consulting companies have the most career mobility; they are also organizations in which you either succeed or you are out. Following this generalization, the industries that pay more, expect more and reward more.

Search for Companies That Fit Your Profile

The next step in targeting companies for jobs is to map the geographic, job, and salary requirements with your intended market area. For the target city or location, map your industry and company characteristics with those of specific organizations in the area. This step requires library searching of business reference guides, *Business Week*, *Forbes*, *Fortune*, *Money* and other business magazines that publish annual reviews of companies by industry.

Look for the geographic region that matches yours, then research the industries in that region. All of this can be done at a global level in an encyclopedia. Next, look at an annual review (e.g., *Fortune*'s '500'), and locate companies in your industry(s) and geographic area. If the headquarters are not in the area, you will need further research. Read company annual reports to locate subsidiaries and their locations. Research companies and industries in each of your target states and metropolitan areas by contacting Better Business Bureaus or Chambers of Commerce. Read reference materials from trade associations and the government to find target companies.

The major warning in this search is to be realistic. If you target, for instance, the chemical and pharmaceutical industries to take advantage of your summer jobs in a small chemical company, the ideal geographic area is the state of New Jersey. Every major pharmaceutical company *in the world* maintains some sort of facility in New Jersey or New York City. At least four major pharmaceutical companies have regional or worldwide headquarters in the area (i.e., Merck, Pfizer, Hoffman-LaRoche, Warner-Lambert). If you target that industry and begin looking in, for instance, Mississippi and Louisiana, you will find only small companies and less than a handful of large ones.

Assess the Reality of Your Ideal Job and Adjust

When you have found the population of companies from which you expect to have a job, evaluate how realistic your chances are. The realism of your probable job is a function of industry turnover and the number of jobs of the type you want in the area in which you want to live. The IS profession has, on average, 15% turnover per year. This means that 15% of the people in IS professions change jobs every year.

In addition, software engineering is the hottest growing job classification in the 1990s.[4] In the same book, Krantz rates computer systems analyst as sec-

4 The growth of software engineering is documented in Les Krantz' *The Jobs Rated Almanac*, 2nd ed., NY: Pharos Publishing, 1992.

ond; computer service technician as fifth; computer programmer as 25th; and technical writer as 147th [Krantz, 1992, p. 218].

If you are choosing an analyst, programmer, or SE position, and you are targeting a geographic area with a large number of target companies, you probably do not need to go through this exercise. If you choose any nonmainstream job, or a limited geographical area, then this exercise might help you assess the reality of your goals. The steps to assessing the reality of your ideal job are:

1. Estimate the number of entry-level jobs available.
2. Estimate the number of people competing for the jobs.
3. Assess the ratio of available jobs to job applicants and adjust your expectations as needed.

Estimate Number of Entry-Level Jobs

First, in assessing the number of potentially available positions, the items of interest are the number of people in IS jobs in an area, the percent of jobs of the type that you want, the average turnover in IS positions, and the percent of entry-level positions. The number of people in IS jobs is one which you must unearth through library and other research. Figure 19-6 shows the major IS job types and estimated percentages of people with that title. Average IS turnover is historically between 15% and 18%. The average number of entry-level positions varies from 2% to 5% per year. When in doubt, use the conservative numbers for your calculations.

The formula for computing the number of likely jobs is as follows:

Number of IS jobs in area
× Percent jobs for your ideal
× Average IS turnover
× Percent of entry level positions
= Number of available jobs

Let's look at an example. If you target the pharmaceutical industry in the New Jersey/New York area, there are approximately 8000 IS jobs. Using the target jobs of programmer or DBA, the total number of likely jobs is 2000 (i.e., (.20 × 8000) + (.05 × 8000)). Multiply this by the 2% to 5% entry-level positions, and you have approximately 40 to 100 programmer and DBA entry-level positions in the pharmaceutical industry in the New Jersey–New York area available in any one year.

Position	Estimated Percentage of Staff
Administration	1% per company
Application Programmer	15–20%
Technical Support, Systems Programming, System Software Support	3–5%
Data Base Administrator	3–5%
Analysts/Designers/ Software Engineers	10–15%
Project Managers	5–10%
Operations	25–35%
EDP Audit	3–5%
Consulting	3–5%
PC/User Support, Help Desk, Information Center	3–5%
Telecommunications	8–10%
Data Administration	3–5%
Other	3–5%

FIGURE 19-6 Estimated Percentage of Major IS Jobs

Estimate the Number of Competitors

Next, evaluate your competition. The competition is all graduating IS majors from local colleges and universities. The number of people moving into and out of the area are not considered here. According to *Computerworld*, the average number of computer-related majors is approximately 2.5% of entering freshman classes.[5] For our purpose, we will use this

5 See *Computerworld*, Vol. 27, #17, April 26, 1993, p. 105.

percentage to extrapolate to graduates. The formula used is:

Total number of graduates from four-year institutions
× Percent of IS graduates
= Number of competitors for IS jobs

For our example, the average number of graduates per year in the New Jersey–New York area is about 16,000. Multiply this by .025 and you find there are about 400 other entry level people against whom you will compete. Since pharmaceuticals employs less than 30% of the IS people in the metropolitan area, your competition should be (400 × .3) or about 120.

Assess Ratio

After computing the number of likely jobs and likely competition, compare the two. If the ratio of jobs to applicants is high, begin your job search. If the ratio of jobs to applicants is low (i.e., less than 1:10), you need to reassess the realism of your goals. In the example, there are 40 to 100 jobs in the industry and job desired. There are about 400 total competitors for all jobs and, on average, about 120 competitors for the same jobs desired. In a growing economy, there is a reasonable likelihood (about 83% probability) of your getting the job you defined. In a weak or falling economy, fewer jobs will be available and the probability of success would be less.

Adjust your Expectations for an Unfavorable Ratio

If you reassess, decide how realistic *this* job is. You might broaden the geographic area or job description you are searching to greatly increase your likelihood of success. If the absolute number of jobs is very low (i.e., under ten per year), then you may need to broaden your view of jobs you are willing to perform. If you want a really specialized job, such as computer game designer, then there might not be many full-time opportunities, but there may be other alternatives and issues to assess. For instance, what is the likelihood of part-time work? What are hiring practices in this industry? Are they different in any way that you can exploit to your advantage? How willing are you to look until you find exactly *this* job?

If there are only a few jobs, but you have your heart set on one of them, plan your job campaign carefully. Why should a company hire you? List the skills and attributes that make you one of the top two candidates out of a field of hundreds. What unique skills or personality characteristics do you possess that you could exploit in this position? Make sure your resume highlights all of your attributes and succinctly summarizes all of your capabilities enough to make a personnel representative want to bring you in for interviews.

Keep in mind that companies are looking for professionals who know how to work, team players who can get along in groups, and self-motivated, domain specialists who know how to find information when they need it.[6] What sells you to a company is your potential and attitude about work. If you present a professional demeanor and appear competent, your probability of success increases.

This section summarizes an approach to locating the ideal job by defining your ideal, then matching it to realistic estimates of the number of likely jobs available in your target geographic area. Keep in mind that the percentages of industry representation for jobs is constantly in a state of change and that you need to do some research to have accurate figures. Fifteen years ago there were no PC-support groups, PC software developers, or LAN managers. Now, those and related jobs are the fastest growing segments of IS professions, just as software engineering is the largest growth position in IS.

MAINTAINING PROFESSIONAL STATUS

Above we mentioned that continuous learning is a requirement for a career in IS. With over 1,000 prod-

6 These traits have been discussed numerous times in *The New York Times*, *Computerworld*, *Datamation*, and other trade periodicals over the last ten years.

uct announcements and introductions a week, the field is everchanging and is changing at an ever-increasing rate. Change is a way of life. You, as a professional SE, must also change and grow to continue to be a valued employee of a company. In this section, we discuss how to develop as a professional through educational, professional, and other types of organizations. Eventually, you need to develop a 'spiral' approach to your knowledge in which you are constantly building on what you have already learned to both reinforce and fix old knowledge more strongly in your mind, and to add nuances and new information that broaden the scope of your knowledge.

Education

As a novice in IS, an undergraduate degree is sufficient for most entry-level positions. If you aspire to managerial or technical specialist positions, however, you should consider obtaining an MS or MBA in either computer science or IS, depending on how technical you wish to be.

The undergraduate degree gives you basic knowledge about the field and a quick survey of theory in developing applications and programs. The emphasis in undergraduate programs is on providing both a skill set to get you a job and a theoretical basis for continued learning in the field. The graduate program emphasizes decision making, problem analysis and solution, and theory of information systems more. The entry-level positions of people with advanced degrees is somewhat higher than that of entry-level undergraduates. The normal masters entry-level position is at an analyst or a first line manager level.

Graduation from a degree program is not sufficient to maintain your growth in the ever-changing field of information systems work. New technologies, new ways of working, new methodologies, and new organizations all demand that you maintain some currency in the field. Many politicians and educators are calling for a *learning-for-life* approach. Using this approach, you take formal degrees and supplement them with continuous education throughout your life. The learning-for-life approach is appropriate to any job in information systems, especially jobs of software engineers. You are the expert in the deployment of new technologies for your company. As the expert, you must learn where and how to find information about any subject required. As the expert, you must try to develop some level of expertise in many fields that are not your specialization. In short, you should try to become a jack-of-all-trades *and* an expert of several.

Professional Organizations

One method to provide you continuous learning experiences while having fun at the same time is to participate in professional organizations. Every organization has conferences or conventions at least annually if not more often. Every specialty has its own organizations or special interest groups (SIGs) as part of a larger, general group. You should seek to be on panels, present papers, or simply participate in at least one conference or convention each year. Many companies pay for their employees to attend such conventions because it is in their interest to have you remain current, too.

Professional organizations are good for a variety of personal goals: keeping current, knowing what other companies are doing, and developing a network of friends for future job possibilities. It is not necessary to belong to every organization; rather, you should pick the one that maps to your goals most closely, provides the literature you most want to keep current with, and is most active in your geographic area. Each organization is discussed in terms of their membership profiles, types of professional activities sponsored, and chances for involvement of industry professionals. Some of these organizations are profiled in this section.

General Technical Organizations

There are many worthy professional organizations in which SEs can participate. Two of the oldest and largest are featured here: ACM and IEEE Computer Society. The addresses for these and other organizations are included in Figure 19-7 for your convenience in contacting them for membership information.

ACM
New York, NY

American Society for Information Science (ASIS)
Washington, DC

Association for Systems Management (ASM)
Cleveland, OH

Computing Professionals for Social Responsibility (CPSR)
Washington, D.C.

Data Processing Management Association (DPMA)
Park Ridge, IL

Graphic Communications Computer Association (GCCA)
Arlington, VA

IEEE
Washington, DC

The Institute for Management Sciences (TIMS)
Providence, RI

Society for Information Management (SIM)
Chicago, IL

Women in Computing (WIC)
New York, NY

FIGURE 19-7 Professional IS Organizations

The Association for Computing Machinery (ACM) is the oldest and largest organization specifically for IS professionals. The ACM was founded in 1947 and has grown to over 81,000 members. The membership ranges from beginning IS students to experienced professionals in industry, education, government, and research. ACM publishes 12 major periodicals with *Communications of the ACM (CACM)* included in the price of membership. *CACM* is generally recognized by academic researchers as *the* premier journal in computing.

Over 30 special interest groups (SIGs) whose specialties span the computing field also have their own newsletters, conferences, and symposia. The SIGs are active organizations that are constantly looking for infusions of new ideas, welcoming new members. Many of the conferences represent both industry and academic members with hundreds of active participants. A representative sample of SIGs includes SIGCHI—computer and human interaction, SIGOIS—office information systems, SIGMOD—management of data, SIGSOFT—software engineering, SIGPLAN—programming languages, SIGGRAPH—graphics, SIGBIT—business information technology, and SIGCAS—computers and society.

Opportunities for involvement include initiating local chapters of SIGs or ACM, participating in one or more of the 30–50 conferences sponsored by ACM each year, participating in any of the SIGs' or ACM's management. Almost all of the work done for ACM is voluntary and requires a time commitment, but the professional recognition and personal benefits are worth the effort.

Another organization, the Institute of Electrical and Electronics Engineers (IEEE), is a 300,000 member organization, of which about one third are members of the Computer Society. The original organization, the American Institute of Electrical Engineers, was founded in 1884 by Thomas A. Edison, Alexander Graham Bell, and Charles P. Steinmetz to foster the development of the engineering profession. Over the years the organization's name changed several times before becoming the IEEE in 1963. In the 1940s, the IEEE established a Committee on Computing Devices that evolved into the Computer Society.

The IEEE is active in all phases of engineering and computing for new and established technologies. Over 30 conferences each year are sponsored by the organization. IEEE Computer Society is known for its quality publications which include tutorials on every major technological development in recent years. The tutorials are compilations of articles exploring the issues, research directions, and likely market outcomes for new technologies and techniques (e.g., object orientation).

IEEE publications are both technically and nontechnically oriented. IEEE *Computer* and *Software* are specifically oriented to professionals working in industry who are trying to maintain current knowledge in the field. Other more technical publications are special interest publications with two of special interest to SEs: *IEEE Transactions on Software Engineering* (*TSE*) and *IEEE Transactions on Knowledge and Data Engineering* (*KDE*). The *TSE*

provides basic research papers on specification, design, development, maintenance, measurement, and documentation of applications. *TSE* is one of the best publications for early discussion of emerging techniques. Its research orientation may make it 'too technical' for some readers. *KDE* is a similar publication aimed at applications' methodologies, storage techniques, AI modeling, and development.

IEEE is subdivided into technical committees (TCs) which participate in industry standards development, conferences, and publications. There are over 20 hardware, software, and interdisciplinary TCs. The software TCs, for instance, include software engineering, computer languages, data engineering, operating systems, real-time systems, and security and privacy.

Conferences are a major TC activity with each group sponsoring one or more major conferences each year. The TC on software engineering coordinates the International Conference on Software Engineering (ICSE), which attracts about 1,500 worldwide participants annually. The major topic areas of ICSE include design, modeling, analysis, and application of software and software systems. The conference usually includes a 'tools fair' which provides vendors an opportunity to feature prototyping languages, CASE environments, language generators, and other software development support tools.

IEEE is more actively involved in standards development than most other organizations. For instance, the 802 committee is the sponsor of many LAN standards in this country. Subcommittees define, for example, the 802.3 ethernet standard. Participants in the technical standards committees are volunteers who are sponsored by their business organizations to participate in the intensive and time-consuming, but personally rewarding, standards definition activities.

Like all of the professional organizations, IEEE strives to involve all of its members in activities. Almost all of the work is voluntary and might include local chapter participation, or participation in national conferences, publications, or organizations.

There are many other equally rewarding organizations listed in Figure 19-7 that are too numerous to detail here. There is significant overlap between the interests of *all* of the organizations, and there is room for you in one or more of them. Keep in mind that it is not necessary to join all of the organizations, but one or two help you maintain current knowledge of IS developments.

User Organizations

In addition to industry organizations, there are many professional user organizations that are sponsored by vendors for their users, or by interested individuals who share common interests.

Hardware User Organizations

Hardware user organizations are vendor-sponsored groups that are convened for users to share their use of the hardware, develop solutions to problems, and to provide guidance and requests to the vendors for future services or capabilities. The organizations are all very active and use volunteers from using organizations whose participation is sponsored by their companies. All major vendors sponsor user groups, including IBM, DEC, Unisys, CDC, Honeywell, AT&T, Sun, Apple, and so on.

IBM, for instance, has two such user organizations: GUIDE and SHARE. GUIDE is an organization of several thousand business and government installations whose use of computers is primarily for business applications, such as transaction processing or decision support applications. SHARE was founded by scientific businesses to support their special needs. Over the years, the missions of the two organizations have come to be similar, but the two organizations remain distinct. Each organization sponsors conferences and workshops several times each year. The conventions are like any professional convention, composed of general sessions in which presentations on topics of interest are made, and working sessions where commitments to work on projects or to present at future meetings are made. The working groups are completely voluntary and first time participants are recommended to attend the meetings of many working groups to get a feel for what they do.

Working group areas include hardware, operating systems, telecommunications, applications, CASE, database, data management, language (e.g., COBOL), security, audit control, and disaster recovery, to name a few.

Software User Organizations

Similar to hardware vendors, major software vendors provide user group support for their users. All user participation is voluntary and at the expense of the user's company. Software vendors include, for instance, Information Builders, Inc. for its 4GL FOCUS, Novell for its network operating system, and all major database vendors, such as Software AG for its Adabas. Each vendor schedules an annual meeting of its user group, providing the facility. Presentations by users center around using the product in their organizations and discussing innovative product use or problems and how they are overcome. The vendors also make presentations at these meetings, including tutorials about using their product and new feature announcements.

Birds-of-a-Feather Groups

Birds-of-a-feather groups are semiformal groups of IS and nonIS professionals who share an interest in some area. The topic matter might be technically specific. For instance, the Data Administration Management Association (DAMA) is a support group for people who are interested in or perform the functions of data administration in business organizations. Similar groups exist for the insurance industry, sponsored through the Life Operations Management Association (LOMA).

For some groups, the topic matter is less specific. For instance, the Boston PC Users Group which number about 15,000 members, is interested in supporting and networking PC users in the Boston metro area. Every metro area has its own user groups that are loosely organized by the type of computer or operating software they own—PC, Macintosh, Pick operating system, Unix operating system, and so on.

Professional Educational Organizations

Another approach to keeping current is to attend seminars that are organized and presented through professional education organizations. There are many noted speakers who reach their audiences in this way, for instance, James Martin, Carma McClure, and Grady Booch, just to name a few. Most such training is company sponsored because of the expense. Expect to pay $600+ per day for these courses.

When attending professionally sponsored training, several important issues should be monitored. Only choose seminars that specifically address your concerns. If you choose, for instance, object-oriented analysis, hoping to hear information about object-oriented languages, you might be disappointed.

Also, beware of the instructor. Review the entire outline of a multiday seminar to make sure that the 'name' person who is to speak actually speaks for a good portion of the time. Review the credentials of all speakers and instructors to ensure that they are qualified to teach the course. If you cannot tell from the brochure, call for more information about the person and the class.

Review outlines of courses for content to ensure that you are attending the course you think you are attending. Sometimes the names and the content are not congruous. Stay away from courses that are programming without any hands-on. If hands-on sessions are planned, assistants (or the instructor) should be present during the session, each student should have a PC, and there should be additional time available at night.

Finally, ask questions about seminar size and maximum number of participants to get a sense of your ability to interact with the instructor. Avoid sessions that have no maximum or minimum or that have maximums over 30 participants except for high-level topic introductions. You know from your own classes that class sizes over 30 are presented differently and have less intimacy between instructor and class. Similarly, less than ten people is not conducive to sharing either. In small groups, it is easier for one individual to monopolize discussion times,

making the instructor's job one of personality management rather than class interaction.

Research and Academic Organizations

The last type of organization in which you might participate focuses on research and teaching of IS-related subjects. Academics have their own conventions that may serve as a forum for debate and presentation of the latest techniques and research on emerging areas of interest. They also provide an outlet for research presentations on a wide variety of topics.

The largest such conference is the International Conference on Information Systems (ICIS) which is held annually in early December. The location of the conference rotates around the world with the majority of conferences currently held in North America. The conference locations for the next several years include Vancouver, British Columbia—1994; The Netherlands—1995; Cleveland, Ohio—1996; and Atlanta, GA—1997.

Topics of interest at recent ICIS conferences include globalization of IS, object orientation, ethics and IS professionals, use of ITs in business organizations, CASE, and computer-supported diversity of organizations. Although about 90% of attendees at ICIS are academics, the remaining 10% of professionals is increasing. Panel sessions frequently include practitioners from industry. Key note addresses are mostly by local CEOs or CIOs who discuss the future of IS from their perspective.

ICIS is not a conference that all practitioners need to attend regularly. Rather, if the theme of the conference matches an interest in your organization, ICIS is a good place to hear about the latest research in the area, and to meet the people doing the research. Occasional attendance at a conference such as ICIS once every three to five years is probably enough to maintain contact with academia.

Accreditation

Professional organizations help you keep current in the field with new developments in new areas and with updates on areas you already know. Accreditation is one method to prove to the world that you indeed are expert in some area. You take an exam which is given once or twice each year, and, if you pass, you obtain a certificate that you know a particular technical area. The major proponents of general IS accreditation are professional organizations, such as DPMA, which sponsors the exams for Certified Data Processor (CDP), Certified Systems Professional (CSP) and others.

A different type of accreditation is managed and provided through vendors to certify the knowledge base of people who support their products. Novell's Certified Netware Engineer (CNE), for instance, requires the passing of an exam that follows completion of a networking and telecommunications course. The courses may be intensive one to two week events that are sponsored by the vendor, or they may be offered through a continuing education program at a local university and span several months of part-time study.

The motivation for accreditation is simple: Many people profess to be IS professionals, few really are. Those few should be rewarded by having the recognition of their knowledge and expertise. Then, when, for instance, consultants advertise their ability to perform a job, the credentials they offer have *some* instant credibility when they include accreditation ratings. The word *some* is emphasized here because passing an exam is still not the same as performing on a job. The point of accreditation is to separate those who have detailed knowledge about the field from those who do not. Having accreditation is no guarantee of work performance.

Read the Literature

Reading *is* fundamental to maintaining currency in methodologies, technologies, and industry with changes that take place as rapidly as in the information systems field. When selecting periodicals, newspapers, and/or books for keeping current, you should have a clear idea of why you are spending your hard-earned money on each purchase. For each type of literature, this section discusses what you should try to keep current on, why you should be current, the general tone and content of articles and/or chapters

for the type of literature, and what you should get from reading this type of writing. The three general types of literature discussed are practitioner journals and newspapers, books, and academic research journals.

Practitioner journals/papers allow you to maintain awareness of the market place and vendors. When reading journals and newspapers, always keep in mind how applicable the products might be to your organization. These periodicals are good for finding out the latest announcements and about products that are already on the market. They provide the following:

- product introductions
- product comparisons
- case studies or descriptions of other organizations' product use

Some periodicals that are in this category include *Computerworld*, *Datamation*, *CIO*, *CASE Trends*, *PC Week*, *PC World*, *MacUser*, *MacWorld*, *InfoWorld*, *Byte*, *PC*, *LAN*, *LAN Week*, and so on.

Books are the next type of reading material you should maintain and read. Books provide summaries of what is currently known on a subject. Read books to increase your knowledge, learn new techniques, find out about a new area, or get ideas to try in your own company. Begin to build a library of reference materials you can use throughout your career. To do this requires careful selection of topics and authors. Seek books that provide information on the following topics as well as others of your interest and *read them!*

- New methodologies (e.g., object orientation such as Peter Coad & Ed Yourdon, *Object Oriented Analysis*, second edition)
- New techniques (e.g., normalization or entity-relationship diagramming such as Peter Chen, *Entity Modeling Techniques*)
- Intellectual development of one person's research (e.g., artificial intelligence such as Roger Schank, *Tell Me a Story*)
- Interesting approaches to solving a problem (e.g., a 37¢ mistake in a Unix LAN billing report led to a spy ring in Germany in Clifford Stoll's, *The Cuckoo's Egg*)
- New ways of combining disparate technologies that will change future ways of computing (e.g., how to combine database, object orientation, and artificial intelligence in Parsaye et al.'s, *Intelligent Database Systems*)
- Well-written and comprehensive text books on all IS topics (e.g., costing, estimating, and CoCoMo use by Barry W. Boehm *Software Engineering Economics*)
- Classics that describe the intellectual growth of IS professions (e.g., Ed Yourdon, *Writings from the Revolution*, or ACM, *Turing Award Lectures 1966–1985*)

Finally, research journals discuss the latest theories about technology use and how it impacts organizations. Many studies are empirical, that is, using a large enough sample to apply statistical techniques in analyzing the theorized behavior. You may not understand all of the statistics in such research, but you should be able to evaluate the quality of the research and assess its applicability to your organization.

Sample journals you might read periodically include *IEEE Transactions on Software Engineering*, *Computer*, *Software*, *Communication of the ACM*, *TOOIS*, *MIS Quarterly*, *Information Systems Research*, and the *IBM Systems Journal*.

Automated Support Tools for Job Search

Two types of automated tools for job search are available and growing in use. First, universities are going on-line in their support of jobs databases that are accessible to students. Gone are the days of leafing through volumes and volumes of randomly organized paper job notices. Instead, the jobs are categorized by seniority, location, salary, job classification, and other demographics. You use a query system to narrow the search and find leads for jobs in which you are interested.

Second, computer bulletin boards for jobs are available in a number of local markets and on the

TABLE 19-1 Automated and Other Support Tools for IS Career Definition

Title	Author/Source	Content
Looking for Work: An Interactive Guide to Marketing Yourself	Frank L. Greenagel, InterDigital Inc. 25 Water St. Lebanon, NJ 08833 (908) 832-2463	Under $30, provides worksheets and tips to finding the right job for you.
No Specific Shareware Title	Software Labs 100 Corporate Point Suite 195 Culver City, CA 90231 (800) 569-7900	Many diskettes available at under $4 each that offer tips on IS jobs.
Bootstrappin' Entrepreneur: The Newsletter for Individuals With Great Ideas and a Little Bit of Cash	Kimberly Stansell Suite B261 8726 S. Sepulveda Blvd. Los Angeles, CA 90045	A free booklet of tips for beginner entrepreneurs.

Internet. Internet is a network of networks that links academic, government, and business organizations worldwide. At last count, there were over one million nodes on the network and many millions of users. Internet and local bulletin boards provide local, almost free access to information about a wide range of subjects. Those relating to job search offer applicants seeking to work in small companies a means to find a company with minor effort. The use of bulletin boards, automated search systems, and other freely available information (e.g., via Internet) will grow considerably in the future.

In addition to automated advertising, tools and booklets are available to help you set your job search course. Several recent publications are listed in Table 19-1.

SUMMARY

In this chapter we discussed emerging career paths for software engineers. Computer science and information systems education are converging due to increasing overlap on areas of emphasis to both groups. While IS SEs will still predominate in business enterprises, and CS SEs will continue to be more technically oriented, both will apply systematic engineering skills and methods to the development of applications.

Next, careers in IS are classified by level and type. The levels of experience are junior, intermediate, senior, lead, technical specialist, and manager. Job types differ depending on area of specialization, including application development, application support, technical specialization, staff positions, and other positions.

Application development includes programmer, software engineer, and knowledge engineer. Application support positions include application specialists, data administration, database administration, artificial intelligence engineering, and consulting. Technical specializations are communications, LANs, systems programming, and software support. Staff positions include security, EDP audit, training, standards and technical writing, quality assurance, and technology planning. The other positions include product support, marketing, and end-user specialists.

Next, one approach to career planning was described. The steps in obtaining your next job are to decide your objective, search companies that fit your profile, assess the likelihood of your attaining the ideal job and, if necessary, adjust your expectations.

Keeping current is important to continued growth as an IS professional. Several methods of maintaining currency were discussed. First, continuous education is important to IS which undergoes continuous change. Professional organization membership and active participation are also useful to maintaining current knowledge of IS developments. Establishing your credentials through accreditation can help you attain credibility with potential employers. Continuous reading of books, periodicals, and research journals can help you continue to grow as a professional software engineer.

REFERENCES

"Computerworld 1992 salary survey," *Computerworld*, Vol. 26, May, 1992.

Kennedy, Joyce Lain, "Getting a fair share: Shareware that can help you find a job," *Dallas Morning News*, Sunday, April 18, 1993, Section D, page 1.

Krantz, Les, *The Jobs Rated Almanac*, 2nd ed. NY: Pharos Publishing, 1992.

Robert Half International, Inc., *1992 Salary Guide*. San Francisco, CA: Robert Half International, Inc., 1991.

Robert Half International, Inc., *1993 Salary Guide*. San Francisco, CA: Robert Half International, Inc., 1992.

KEY TERMS

analyst
application specialist
artificial intelligence (AI) engineer
communications analyst
consultant
data administrator (DA)
database administrator (DBA)
designer
EDP auditor
end-user specialist
junior staff member
intermediate staff member
knowledge engineer
lead staff member
local area network (LAN) specialist
manager
marketing support staff
product support staff
programmers
quality assurance
security specialist
senior staff member
software engineer (SE)
software support specialist
standards developer
system software support specialist
systems programmer
technical specialist
technical trainer
technical writer
technology surveillance specialist

EXERCISES

1. Plan your job search. Identify the type of job, the kind of company, location, and benefits you want. Do research to locate specific companies and to determine your competition. Then, compute the likelihood of getting your ideal job. Discuss your plan with the class or in small groups to assess how realistic your plan is.
2. Research the professional and user organizations that you might join and define a rationale for yourself to choose one or two in which you are interested. Join those organizations.
3. Select one or two periodicals that are of interest to you and further your professional goals. Subscribe to them if you do not already.
4. When you have decided your career goal, go to the library and perform a book search to identify potential books for your personal library. Scan five of the books, then share your information with the class, identifying the one or two of the books you intend to buy. Go buy the books and begin to build your library.
5. Choose four technologies for which you would like to become expert. Map a strategy for jobs, reading, and professional group involvement that will help you become an expert within five to ten years. Discuss your strategy in class or in small groups to assess how realistic it is and to obtain suggestions for other ways to reach your goal.

STUDY QUESTIONS

1. Define the following terms:
   analyst
   DA
   DBA
   programmer
   software engineer
   technology surveillance specialist
2. How do computer science majors and information systems majors differ in the approaches

taken by their academic programs? How do they complement each other?
3. What are the levels of experience generally used in titles to separate different levels of expertise?
4. How do the duties of a lead person differ from those of a manager?
5. How do the duties of a lead person differ from those of a technical specialist?
6. In application development, the job types are programmer, software engineer, and knowledge engineer. Define each job and describe how their job content differs.
7. How do the functions of a DA and DBA differ? How do they complement each other?
8. Why and how do companies use consultants? What are companies' expectations of consultants' knowledge?
9. How does an AI specialist differ from a knowledge engineer?
10. What are the duties of a systems programmer?
11. Why are security specialists needed in organizations?
12. Why is quality assurance in an adversarial role with application development project teams?
13. In what types of companies do product and market support people work?
14. Define the steps to planning a career.
15. Why is it important to have an objective when looking for a job?
16. How do you compute your chances of getting the job you desire in the type of company you want?
17. What are the types of organizations you might join to continue growth as an SE professional? Which type appeals the most to you?
18. Why is continued growth of both knowledge and experience important to a professional SE? What happens if you do not continue to learn?

APPENDIX A

CASES FOR ASSIGNMENTS

ABACUS PRINTING COMPANY

This case describes a currently manual process. Your job is to automate the order processing, scheduling, and customer service functions. Make sure you list any assumption you make during analysis and design.

Abacus Printing Company is a $20-million business owned and operated by three longtime friends. They are automating their order processing for the first time. Abacus Printing is located in Atlanta, Georgia and employs 20 people full-time.

The owners are the sales force. The company is set up so that each owner sells for a different, wholly-owned subsidiary (A Sub, B Sub, and C Sub) to separate commissions and expenses for tax purposes. Below is a description of the work to be automated.

Three clerks do order entry and customer service. An order is given to one of the three clerks to be entered into the order entry part of the system. Orders are batched by subsidiary for processing in the system. There is at least one batch per clerk per day. When a batch is complete, orders are printed. After orders are printed, the system should maintain individual orders for processing (i.e., the integrity of the batch is no longer needed).

Orders are printed and become internal job tickets which are used to schedule and monitor work progress. All order/job tickets go to the scheduler who sorts and prioritizes them to develop a production schedule. Each Monday, he gives the first person in the work chain (there are three possible sequences of processing) the job tickets for completion that week. As the week progresses, he adds to or changes the schedule by altering the order and adding new tickets to the stack of each person beginning a work chain. Each job goes through the same basic steps:

Step 1. Perform requested manufacturing (i.e., the engraving or printing work) according to the job ticket instructions.
Step 2. Verify quality of printed items and count output, that is, actual printed sheets of paper or envelopes. Write the actual count of items to be shipped on the job ticket.
Step 3. Update the order/job ticket with actual shipment information; print shipping papers and invoices which reflect actual shipments.
Step 4. Bundle, wrap, and ship the order.

The updating of the order with actual shipment information may be done by either the shipping clerk or by the same person who entered the order. The

second printing 'closes' the order from any other changes and results in a multipart form being printed. Two of the parts are copies of the invoices, showing all prices and other charges with a total amount due. One invoice copy is sent to the customer; the other is filed for further processing by accounts receivable. The third part of the set of forms is the bill of lading, or shipping papers, that shows all information except money amounts. The fourth part of the form is filed numerically by invoice number in a sequential history file. The fifth part is filed in a customer file which is kept in alphabetic sequence.

The system must allow order numbering by subsidiary company, and must be able to print different subsidiary name headers on the forms. The clerks batch orders so that only orders from one subsidiary are in each batch. Order types include recurring orders, blanket orders (which cover the year with shipments spaced out over the period), and orders with multiple ship-to addresses that differ from the sold-to addresses.

When customers call to change or determine the status of an order, the clerk taking the call first checks the customer file to see if the order is complete. Then, he or she checks with the scheduler to see if the order is in the current day's manufacturing mix. If the order is not complete or scheduled, he or she manually searches current orders to find the paperwork. About 15% of customer calls are answered while the customer is on the phone. About 80% require research and are answered with a call back within 30 minutes. The remaining 5% require tracking, which results in identifying an order taken verbally by a partner and never written down. Customers have been complaining of the lost orders and threatening to go elsewhere with their business.

The current computer system is a smart typewriter and storage facility. The owner wants to provide personal computer access via a local area network for the three partners, three clerks, two shipping staff, and one scheduler. He would like to eliminate the numerical and alphabetical paper filing systems but wants to maintain the information on-line indefinitely for customer service queries.

The managers want ad hoc reporting access to the information at all times. The senior clerk is also the accounting manager and, along with the owner, should be allowed access to an override function to correct errors in the system. The other clerks should be allowed to perform data entry for order processing and actual goods shipped, and to print invoices/shipping papers. The shipping clerk should be allowed to perform order updates with actual goods shipped and to generate shipping papers with a final invoice. The scheduler should be allowed access to all outstanding orders to alter and schedule work for the manufacturing processes. No one else in the company should be allowed access to the system or to the data.

AOS Tracking System

The AOS case is a logical description of a desired application that also includes manual problems to be corrected.

The manager of Administrative Office Services (AOS) wants to develop an automated application to track work through its departments. The departments and services provided include: word processing and proofing, graphic design, copying, and mailing. Work can come into any of the departments, and any number of services might be combined. For instance, word processing and proofing can be the only service. Word processing, proofing, and graphic design might be combined. Another job might include all of the services.

The current situation is difficult because each manager has some knowledge of the work in his or her own area, but not where work is once it leaves their area. Overall coordination for completing jobs using multiple services requires the AOS manager to give each department a deadline. Then, the AOS manager must track the jobs to ensure that they are completed and moved along properly.

The basic work in each department is to receive a job, check staff availability based on work load and skills, assign staff, priority, and due date, and update job information (for instance, if the work is reassigned). Jobs are identified by a unique control number that is assigned to each job. Other job information maintained includes: requestor name,

requestor phone, requestor budget code, manner of receipt (either fax, paper, or phone dictation), manner of delivery (either fax, paper, or phone dictation), and dates and times work is received, due, completed, canceled, notified, and returned to requestor.

A job consists of requests for one or more types of service. For each type of service, information must also be kept. Services include word processing and proofing, copying, graphic design, and mailing.

Information kept for word processing and proofing services includes a description of the job, type of request (letter, memo, statistics, legal document, special project, chart, manual, labels, etc.), other services included with this request (i.e., copying, graphic design, mailing), software to be used (WordPerfect, Harvard Graphics, Lotus, Bar Coding, Other), type of paper (logo, plain bond, user provided, envelope, other), color of paper (white, pink, blue, green, buff, yellow, other), paper size (8.5" × 11", 8.5" × 14", other), special characteristics (2-hole punch, 3-hole punch, other), type of envelope (letter, legal, letter window, legal window, bill, kraft 9" × 12", kraft 10" × 13", supplied by requestor, other), number of copies requested, user control number, dates/times required, started, completed, reassigned, proof started, proof completed, revisions started, and revisions completed.

Information kept for copying includes the above except software and dates/times relating to proofing and revisions. In addition, keep requirements for collating, stapling, one-side or two-side, special formats (e.g., reduced 60% and put side-by-side in book format).

Information kept for graphic design and mailing includes that for word processing, except type of envelope. The code schemes for type of request, paper, software, and special characteristics are different from those used for word processing. For instance, paper for graphics refers to type of output media which might actually include slide, transparency, paper, envelope, video still, photograph, moving video, and so on. The type of request must be expanded to include the number of colors, specific color selections, intended usage (intracompany, external, advertising, public relations, other) and level of creativity (i.e., user provides graphic and this department automates the design; user provides concept and this department provides several alternative designs, etc.).

Information kept for mailing includes requested completion date, and the dates and times requests were received, completed, and acknowledged back to requestor as complete. Other information includes whether or not address labels were provided, mailing list to be used (choice of four), number of pieces, method of mailing (e.g., zip+four, carrier route code, bar code, bulk, regular, special delivery, etc.), machinery required (e.g., mail inserter, mail sorter, etc.), and source of mailing (e.g. word processing in AOS, user, other).

As a department's staff gets an incoming job, it should be logged into the system, assigned a log number, and the job information should be entered into the system. In addition, the receiving department completes their service-specific information (e.g., typing) and identifies the sequence of departments which will work on the job. As the individual departments get their task information, they complete the service-specific fields.

Each department manager assigns a person to the task based on skills and availability. First, information matching service requests to staff skills should be done. Then, the staff with required skills should be ordered by their earliest availability date for assignment to the task. The system should allow tracking (and retrieval) of a task by job, department/task, person doing the work, date of receipt, due date, or user.

The manager of AOS would like to receive a monthly listing of all comments received (usually they are complaints) and be able to query details of the job history to determine the need for remedial action. Comments should be linked to a job, service, user, and staff member.

THE CENTER FOR CHILD DEVELOPMENT

This case describes a currently manual process. The analysis and design task is to develop a new work

TABLE 1 Client Card File Information

Last Name
First Name
Middle Initial
Fiscal Year
Medicaid Number
Family Identifier
Line/Person Identifier
Sex
Year of Birth
Diagnosis Code (NA)
Issue Date
Dates of Visits
Fees per Week
Amount Paid
Balance Due (Updated Monthly)

flow and automated system for as much of the Medicaid payment process as possible.

The Center for Child Development (CCD) is a not-for-profit agency that provides psychiatric counseling to children, serving approximately 600 clients per year. Each client has at least one visit to CCD per week when they are in therapy. Most often, the client has multiple visits to the center and to other agencies in one day (e.g., to CCD and, say, to a hospital). Medicaid reimburses expenses for only one such visit per day. This means that multiple appointments at CCD for a given day will have one appointment reimbursed; multiple claims on the same Medicaid number for the same day are paid on a first-in, first-paid basis by Medicaid. The current claims processing takes place monthly; for CCD to remain competitive, Medicaid processing must be done daily. To provide daily Medicaid processing, automation of the process is required. The Medicaid Administration has arranged with personal computer owners to take claims in automated form on diskettes, provided that they conform to the information and format requirements of paper forms.

To develop Medicaid claims, the business office clerk reviews the client card file to obtain Medicaid number and visit information for each client (see Table 1 for Client Card File Information and Table 2 for Visit Card File Information recorded). Based on the card file information, Medicaid forms are completed: one per client with up to four visits listed on each form (see Table 3 for Medicaid information required). Most clients have multiple forms produced because they have more than four visits to the center per month. Each form must be completed in its entirety (i.e., top and bottom) for Medicaid to process them (the forms cannot be batched by client with only variable visit information supplied).

One copy of each form is kept and filed in a Medicaid–Pending Claims File. The other copies of the forms (or disks) are mailed to Medicaid for processing.

About four to six weeks after submission of claims, Medicaid sends an initial determination report on each claim. The response media is either diskette or paper. Reconciliation of all paid amounts is done by manually matching the Medicaid report information with that from the original claim. If automated, report entries are in subscriber (i.e., CCD client) sequence. The paid claims are then filed in a Medicaid–Paid Claims File.

Claims that are disputed by Medicaid (almost 90% are pending on the initial report; of pending claims, 10–20% are ultimately denied) are researched and followed up with more information as required. Electronic reconciliation in other companies reduces the 90%-pending to as few as 10%, thus speeding the reimbursement process. CCD has a contact at Medicaid with whom they work closely to resolve any problems.

TABLE 2 Visit Information

Day
Date
Type Appointment (i.e., Intake, Regular)
Client Name
Time of Appointment
Single/Group Visit
Amount Paid
Amount Owed
Insurance Company
Medicaid (Y/N)
Last Date Seen
Therapist

TABLE 3 Medicaid Claim Form Information

Permanently Assigned Fields	Information Completed by CCD
Company Name (CCD)	Billing Date (must be within 90 days of service)
Invoice Number (Assigned by Medicaid, preprinted on the forms)	Recipient ID Number (Client Medicaid Number)
Group ID Number (Not Applicable, i.e., NA)	Year of Birth
Location Code (03)	Sex
Clinic (827)	Recipient (Client) Name
Category (0160)	Social Worker License Number
Number of Attachments (NA)	Name of Social Worker
Office Number (NA)	Primary/secondary diagnosis (Table look-up, 120 entries) Date of Service
Place of Service (NA)	Procedure Code (This is a two-line entry to identify first the treatment payment on the first line and the treatment code on the second line.)
Social Worker Type (NA)	Procedure Description
Coding Method (6)	Times Performed
Emergency (N, i.e., No)	Amount
Handicapped (N)	Name of person completing the form
Disability (N)	Date
Family Planning (N)	
Accident Code (0)	
Patient Status (0)	
Referral Code (0)	
Abort/Sterile Code (0)	
Prior Approval Number (NA)	
Ignore Dental Insurance (Y)	

(Information in parentheses is the permanent value of that field for CCD)

COURSE REGISTRATION SYSTEM

This case is a logical description of the desired application. Your task is to analyze and design the data and processes to develop an automated application to perform course registration.

A student completes a registration request form and mails or delivers it to the registrar's office. A clerk enters the request into the system. First, the Accounts Receivable subsystem is checked to ensure that no fees are owed from the previous quarter. Next, for each course, the student transcript is checked to ensure that the course prerequisites are completed. Then, class position availability is checked. If all checks are successful, the student's social security number is added to the class list.

The acknowledgment back to the student shows the result of registration processing as follows: If fees are owing, a bill is sent to the student; no registration is done and the acknowledgment contains the amount due. If prerequisites for a course are not

filled, the acknowledgment lists prerequisites not met and that course is not registered. If the class is full, the student acknowledgment is marked with 'course closed.' If a student is accepted into a class, the day, time, and room are printed next to the course number. Total tuition owed is computed and printed on the acknowledgment. Student fee information is interfaced to the Accounts Receivable subsystem.

Course enrollment reports are prepared for the instructors.

DR. PATEL'S DENTAL PRACTICE SYSTEM

The dental practice uses a manual patient and billing system to serve approximately 1,100 patients. The primary components of the manual system are scheduling patient appointments, maintaining patient dental records, and recording financial information. Due to increased competitive pressure, Dr. Patel desires to automate his customer records and billing.

New patients must complete the patient history form. The data elements are listed in Table 1. Then, at the first visit, the dentist evaluates the patient and completes the second half of the patient history information with standard dental codes (there are 2,000 codes) to record recommended treatments. The data elements completed by the dentist are listed as Table 2. The patient history form is filed in a manila folder, with the name of the patient as identification, along with any other documents from subsequent visits.

A calendar of appointments is kept by the secretary, who schedules follow-up visits before the patient leaves the office. The calendar data elements are shown as Table 3. Also, before the patient leaves, any bills, insurance forms, and amounts due are computed. The client may pay at that time, or may opt for a monthly summary bill. The secretary maintains bill, insurance, and payment information with the patient history. Financial data elements are shown in Table 4. Every week, the secretary types mailing labels that are attached to appointment reminder cards and mailed. Once per month, the secretary types and sends bills to clients with outstanding balances.

TABLE 1 Patient History Information

- Patient name
  - Address
  - City
  - State
  - Zip
  - Home telephone
  - Date of birth
  - Sex
  - Parent's name (if under 21) or emergency contact
    - Address
    - City, state, zip
    - Telephone number
  - Known dental problems (room for 1–3)
  - Known physical problems (room for 1–3)
  - Known drug/medication allergies (room for 1–3)
- Place of work name
  - Address
  - City
  - State
  - Zip
  - Telephone number
- Insurance carrier
  - City, state, zip
  - Policy number
- Last dentist name
  - Address
  - City, state, zip
- Physician name
  - City, state, zip

TABLE 2 Dentist Prognosis Information

- Dentist performing evluation
  - Date of evaluation
  - Time of evaluation
  - Recommended treatment (room for 1–10 diagnoses and treatments)
  - Procedure code
  - Date performed (completed when performed)
  - Fee (completed when performed)

TABLE 3 Appointment Calendar

Patient name
Home telephone number
Work telephone number
Date of last service
Date of appointment
Time of appointment
Type of treatment planned

TABLE 4 Patient Financial Information

Patient name
Address
City, state, zip
Home telephone number
Work telephone number
Date of service
Fee
Payment received
Date of payment
Adjustment
Date of adjustment
Outstanding balance
Date bill sent
Date overdue notice sent

THE EAGLE ROCK GOLF LEAGUE

This is a logical description of a desired application. The task is to analyze and design the data and processes required to track golfers and rounds of golf, including computation of match rankings.

The members of the Eagle Rock Golf League regularly compete in matches to determine their comparative ability. A match is played between two golfers; each match either has a winner and a loser, or is declared a tie. Each match consists of a round of 18 holes with a score kept for each hole. The person with the lowest gross score (gross score = sum of all hole scores) is declared the winner. If not a tie, the outcome of a match is used to update the ranking of players in the league: The winner is declared better than the loser and any golfers previously beaten by the loser. Other comparative rankings are left unchanged.

The application should keep the following information about each golfer: name, club ID, address, home phone, work phone, handicap, date of last golf round, date of last golf match, and current match ranking.

Each round of golf should also be tracked including golfer's club ID, name, scores for all 18 holes, total for the round, match indicator (i.e., Yes/No), match opponent ID (if indicator = Y), winner of the match, and date of the match. The application should allow golfers to input their own scores and allow any legal user to query any information in the system. Only the system should be allowed to change rankings. Errors in data entry for winners or losers should be corrected only by a club employee.

GEORGIA BANK AUTOMATED TELLER MACHINE SYSTEM

Georgia Bank describes an application to be developed. The functional requirements are described at a high level of abstraction and the task is to do more detailed analysis or to begin design.

The Georgia Bank is automating an automated teller machine (ATM) network to maintain its competitive position in the market. The bank currently processes all deposit and withdrawal transactions manually and has no capability to give up-to-the-minute balance information. The bank has 200,000 demand-deposit account (DDA, e.g., checking account) customers and 100,000 time deposit (e.g., savings account) customers. All customers have the same account prefix with a two-digit account type identifier as the suffix.

The ATM system should provide for up to three transactions per customer. Transactions may be processed via ATM machines to be installed in each of the 50 branches and via the AVAIL™ network of

Georgia banks. The system should accept an ATM identification card and read the ATM card number. The ATM card number is used to retrieve account information including a personal ID number (PIN) and balances for each DDA and time account. The system should prompt for entry of the PIN and verify its correctness. Then the system should prompt for type of transaction and verify its correctness.

For DDA transactions, the system prompts for amount of money to be withdrawn. The amount is verified as available, and if valid, the system instructs the machine to dispense the proper amount which is deducted from the account balance. If the machine responds that the quantity of money required is not available, the transaction is aborted. A transaction acknowledgment (customer receipt) is created. If the amount is not available or is over the allowable limit of $250 per day per account, an error message is sent back to the machine with instructions to reenter the amount or to cancel the transaction.

For time deposit transactions, the system prompts for amount of money to be deposited and accepts an envelope containing the transaction. The amount is added to the account balance in transit. A transaction acknowledgment is created.

For account balances, the system prompts for type of account—DDA or time—and creates a report of the amount. At the end of all transactions, or at the end of the third transaction, the system prints the transaction acknowledgment at the ATM and creates an entry in a transaction log for all transactions. All other processing of account transactions will remain the same as that used in the current DDA and time deposit systems.

The customer file entries currently include customer ID, name(s), address, social security number, day phone, and for each account: account ID, date opened, current balance, link to transaction file (record of most recent transaction). The transaction file contains: account ID, date, transaction type, amount, source of transaction (i.e., ATM, teller initials) and link to next most recent transaction record. The customer file must be modified to include the ATM ID and password. The transaction log file contains ATM ID, account ID, date, time, location, transaction type, account type, and amount.

SUMMER'S INC. SALES TRACKING SYSTEM

This case describes a manual system for sales tracking. Your design should include work procedures and responsibilities for all affected users.

Summer's Inc. is a family-owned, retail office-product store in Ohio. Recently, the matriarch of the family sold her interest to her youngest son who is automating as much of their processing as possible. Since accounting and inventory management were automated two years ago, the next area of major paper reduction is to automate retail sales to floor processing.

The sales floor has four salespersons who together serve an average of 100 customers per day. There are over 15,000 items for sale, each available from as many as four vendors. The system should keep track of all sales, decrease inventory for each item sold, and provide an interface to the A/R system for credit sales.

A sale proceeds as follows. A customer selects items from those on display and may request ordering of items that are not currently available. For those items currently selected, a sales slip is created containing at least the item name, manufacturer's item number (this is not the same as the vendor's number), retail unit price, number of units, type of units (e.g. each, dozen, gross, ream, etc.), extended price, sales tax (or sales exemption number), and sale total. For credit customers, the customer name, ID number, and purchaser signature are also included. The sales total is entered into a cash register for cash sales and the money is placed into the register. A copy of the sales slip is given to the customer as a receipt, and a copy is kept for Summer's records. For orders or credit sales, the information kept includes customer name, ID number, sale date, salesman initials, and all details of each sales slip. For credit sales, a copy of credit sale information should be in an electronic interface to the accounting system where invoices are created.

In the automated system, both cash and credit sales must be accommodated, including the provision of paper copy receipts for the client and for

Summer's. The inventory database should be updated by subtracting quantity sold from units on hand for that unit type, and the total sales amount for the year-to-date sales of the item should be increased by the amount of the sale. The contents of the inventory database are shown in Table 1.

TABLE 1 Summer's Inc. Inventory Database

General Item Information

Item Name (e.g. Flair Marker, Fine-Point Blue; Flair Marker, Wide-Point Blue, etc.)
Item Manufacturer
Date began carrying item
Units information*
Unit type (e.g., each, dozen, gross, etc.)
Retail unit cost
Units on order
Units on hand
Total units sold in 1993

Vendor-Item Information*
Vendor ID
Vendor item ID
Vendor-units information*
Unit type (e.g., each, dozen, gross, etc.)
Last order date
Discount schedule
Wholesale unit cost

Vendor General Information

Vendor ID
Vendor name
Vendor address
Terms
Ship method
Delivery lead time
Item-Information
Vendor item ID
Unit type (e.g., each, dozen, gross, etc.)
Last order date
Discount schedule
Wholesale unit cost

(Note: Primary keys are underlined; repeating groups are identified with a boldface name and an asterisk.)

TECHNICAL CONTRACTING, INC.

Technical Contracting, Inc. (TCI) describes a manual process to be automated. The data and processes are approximately equally complex; both require some analysis and design before the automated application can be designed. First, decide what information in the problem description is relevant to an automated application for client-contractor matching, then proceed with the assignment.

TCI is a rapidly expanding business that contracts IS personnel to organizations that require specific technical skills in Dallas, TX. Since this business is becoming more competitive, Dave Lopez, the owner, wants to automate the processing of personnel placement and resume maintenance.

The files of applicant resumes and skills are coded according to a predefined set of skills. About 10 new applicant resumes arrive each week. A clerk checks the suitability of the resume for the services TCI provides and returns unsuitable resumes with a letter to the applicant. The applicant is invited to reapply when they have acquired skills that are in high demand, several of which are listed in the letter. High-demand jobs are determined by counting the type of requests that have been received in the last month. Resumes of applicants are added to the file with skills coded from a table. There are currently 200 resumes on file that are updated every six months with address, phone, skills, and project experience for the latest period. Most of the resume information is coded. There is one section per project for a text description. This section is free-form text and allows up to 2,500 characters of description.

Client companies send their requests for specialized personnel to TCI either by mail, phone, or personal delivery. For new clients, one of TCI's clerks records client details such as name, ID, address, phone, and billing information. For each requirement, the details of the job are recorded, including skill requirements (e.g., operating system, language, analysis skills, design skills, knowledge of file structures, knowledge of DBMS, teleprocessing knowledge, etc.), duration of the task, supervisor name,

supervisor level, decision authority name, level of difficulty, level of supervision required, and hourly rate. For established clients, changes are made as required.

Once a day, applicant skills are matched to client requirements. Then Dave reviews the resumes and, based on his knowledge of the personalities involved, selects applicants for interviewing by the client company. When Dave selects an applicant, the resume is printed and sent with a cover letter. Dave follows up the letter with a phone call three days later. If the client decides to interview the applicant(s), Dave first prepares them with a sample interview, then they are interviewed by the client.

Upon acceptance of an applicant, two sets of contracts are drawn up. A contract between TCI and the client company is developed to describe the terms of the engagement. These contracts can be complicated because they might include descriptions of discounts in billings that apply when multiple people are placed on the contract, or might include longevity discounts when contractors are engaged over a negotiated period of time. A contract between TCI and the applicant is developed to describe the terms of participation in the engagement. Basically, the applicant becomes an employee of Dave's organization for the duration of the contract.

TCI keeps information on demand for each type of skill, whether they provide people with the skill or not. Dave also monitors TCI performance in filling requests for each skill and evaluating lost contracts due to nonavailability of applicants (to raise his fees for those services, and to advertise for those skills). TCI advertises for applicants with specific skills when client demand for new skills reaches three requests in any one month, or when demand for skills already on file increases to such an extent that the company is losing more than three jobs per month.

XY University Medical Tracking System

The XY University case is a brief logical description of a simple tracking system with a complex data structure. The key to a good design is to analyze and define the data and services properly.

XY University student medical center serves a student population of 60,000 students and faculty in a large metropolitan area. Over 300 patients receive one or more medical services each day. The university has a new president who wishes to overhaul the existing medical support structure and modernize the facilities to improve the services. In order to plan for these changes, more information on which services are in fact used is required. The university wishes to develop a patient tracking system that traces each patient throughout their stay in school for each visit to the facility.

Students and faculty are identified by their identification numbers. They should be logged into the system (i.e., date, time, and ID) when they enter the facility. They may or may not have appointments. Then, some means of recording and entering information into the computer system must be provided for each of the following: station visited, medical contact person, type of contact (i.e., consultation, treatment, follow-up check, routine checkup, emergency, etc.), length of contact, diagnosis, treatment, medicine prescribed (i.e., name, brand, amount, dosage), and follow-up advised (yes/no). All information must be available for query processing and all queries must be displayed either at terminals or on printers.

GLOSSARY

abstract data type In object orientation, the user-defined data type that encapsulates definitions of object data plus legal processes for that data.

action diagram In information engineering, a graphical representation of procedural structure and processing details suitable for automated code generation.

activity In information engineering, some procedure within a business function that can be identified by its input data and output data which differ.

afferent flows In structured design, the input-oriented processes which read data and prepare it for processing.

affinity Attraction or closeness.

affinity analysis In information engineering, a clustering of business processes by the closeness of their functions on data entities they share in common.

analysis The act of defining what an application will do.

application The set of programs that automate some business task.

application characteristic Descriptive information that is common to all applications and includes data, processes, constraints, and interfaces.

application complexity Fundamental application difficulty which comes from several sources, including management of the number of elements in the application, the degree and types of interactions, support, novelty, and ambiguity.

application type The business orientation of the application as transactional, query, decision, or intelligent.

architecture A snapshot of some aspect of an organization, e.g., data, business processes, technology, or communications network.

associative data relationships Irregular entity relationships, dictated by data content rather than abstractions such as normalization.

atomic process A system process that cannot be further decomposed without losing its system-like qualities.

attribute In object orientation, a named field or property that describes a class/object or a process.

audit control Application design components that prove transaction processing in compliance with legal, fiduciary, or stakeholder responsibilities.

backup The process of making extra copies of data to ensure recoverability.

baseline A product that is considered complete and which is the basis for other current work by the project development team.

batch applications Computer applications in which transactions are processed in groups.

benchmark A comparison test used to identify differences between hardware or software products.

benefit Some improvement in the work product or process that results from a specific alternative.

bid The financial response to an RFP. Bid types for hardware are lease, lease with option to buy, or purchase. For software, bid types are time and materials (T&M), T&M with a ceiling, or fixed price.

binding In object orientation, the process of integrating the code of communicating objects. Binding of objects to operations may be static, pseudo-dynamic, or dynamic.

black box A testing strategy that determines correctness of functioning by creating input data is designed to generate variations of outputs without regard as to how the logic actually functions. Black-box strategies include equivalence partitioning, boundary value analysis, and error guessing.

body of screen The large middle part of a screen containing application-specific variable information.

boilerplate Code that is invariant from one program to another, regardless of program function.

Booch diagram In object orientation, a graphical representation of all objects and their processes in the application, including both service and problem domain objects.

bottom-up testing A testing strategy that tests complete modules, assuming that the lower the number of incremental changes in modules, the lower the error rate.

bracket In information engineering, a graphical structure on an action diagram.

business activity In information engineering, some high level set of procedures within a business function.

business area analysis In information engineering, a tabular clustering of processes which share data creation authority for an entity.

business function In information engineering, a group of activities that accomplish some complete job that is within the mission of the enterprise.

business process Details of an activity, fully defining the steps taken to accomplish the activity.

cardinality The number of an entity relationship; can be one-to-one, one-to-many, or many-to-many.

CASE integration The absence of barriers between one graphical or text form and others.

central transform In structured design, processes having as their major function the change of information from its incoming state to some other state.

champion A manager who actively supports and sells the goals of the application to others in the organization.

change control Project management techniques for dealing with changes to specifications, application functions, documentation, etc.

class In object orientation, like objects that have exactly the same properties, attributes, and processes.

class hierarchy In object orientation, the basic hierarchy of relationships between classes of objects that also accommodates lattice-like network relationships.

class/object In object orientation, a set of items which share the same attributes and processes, and manage the instances of the collection.

client object In object orientation, an object that requests a process from a supplier object.

code The low-level program elements of the software product created from design documentation; procedural computer instructions.

code generator A program that reads specifications and creates code in some target language, such as Cobol or C.

coding The stage of application development during which computer code is generated.

cohesion A measure of internal strength of a module with the notion that maximal or functional cohesion is the goal.

command language High-level programming languages that communicate with software to direct its execution.

composite cost model (CoCoMo) A combination of estimating techniques based on thousands of delivered source instructions.

compromise of requirements A change to application functions to rescope, manipulate, drop, or otherwise change them to fit the environment's limitations.

computer-aided software engineering (CASE) A computer application that automates the development of graphics and documentation of application design. CASE can be intelligent and include verification capabilities to ensure syntactic correctness of information entered.

concurrent processes In object orientation, processes that operate at the same time and can be dependent or independent.

configuration management Management of software code libraries.

constraint Limitations on the behavior and/or processing of entities, including prerequisite, postrequisite, time, structure, control, or inferential.

context A setting or environment.

context diagram A graphic developed during structured analysis to define the interactions of the application with the external world.

contingency planning The identification of tasks designed to prevent risky events and tasks to deal with the events if they should occur.

control point A location (logical or physical) in a procedure (automated or manual) where the possibility of errors exists.

controlled redundancy The deliberate duplication of data for control purposes.

conversion The placing of a computer application into production use; includes direct cutover, functional, geographic methods.

cost The amount of money or other payment for obtaining some benefit.

cost/benefit analysis The comparison of the financial gains and payments that would result from selection of some alternative.

coupling A measure of intermodule connection with minimal coupling of the goal (i.e., less is best).

critical path The sequence of interrelated tasks during application development that takes the most time to develop.

critical success factor Some business activity or function that is crucial to the organization's success.

CRUD matrix *See* entity/process matrix.

cutover A method of conversion such that, on a set day, the old way of work is abandoned and the new way begins to be used.

data The elements in raw material—numbers and letters—that relate to each other to form fields (or attributes) which define entities.

data administration (DA) The management of data to support and foster data sharing across multiple divisions, and to facilitate the development of database applications.

data characteristics Descriptive information about data including ambiguity, completeness, semantics, structure, time-orientation, and volume.

data collection techniques Methods of obtaining information and application requirements, including interviews, meeting, observation, questionnaires, temporary job assignment, document review, and external source review.

data dictionary In structured analysis, a compilation of detailed definitions for each element in a DFD.

data distribution choices In data distribution analysis, possible designs include data centralizing, replicating, vertical partitioning, subset partitioning, or federating.

data flow diagram In structured analysis, a graphic representation of the application's component parts.

data methodology Those development methods that begin defining functional requirements by first evaluating data and their relationships to determine the underlying data architecture.

data model A conceptual description of the major data entities of interest in an organization for reengineering, or in an application for subject area database definition.

data self-sufficiency A property of application target organizations such that 70% (or more) of data used in performing the business functions originates within the subject organizations.

data type A language-fixed definition of data, e.g., integers.

data warehouse The means to store unlimited, continuously growing databases.

data-oriented methodology Approaches to developing applications that assume data are fundamentally more stable than processes and should, therefore, be the focus of activities.

database administration (DBA) An organization created to maintain and monitor DBMS use, including responsibility for physical DB design, disk space allocation, and day-to-day operations support for the actual database.

denormalization The process of designing storage items of data to achieve performance efficiency.

decision support applications (DSS) Applications whose purpose is to seek to identify and solve problems.

depth of hierarchy In structured design, the number of levels in the diagram.

derived field Fields/attributes for which the application is the source, i.e., computed fields.

design The act of defining how the requirements defined during analysis will be implemented in a specific hardware/software environment.

developmental tests Testing conducted by the project development team, including unit, subsystem, integration, and system tests.

dialogue In object orientation and information engineering, interactive communication that takes place between the user and the application, usually via a terminal, to accomplish some work.

dialogue flow diagram In information engineering, a diagram summarizing allowable movement between entries on a menu structure diagram.

direct manipulation Screen interactions during which the user performs an action directly on some display object.

display The screen portion of a computer.

distributed computing A situation in which multiple processors share responsibility for managing pieces of an application.

divide and conquer The principle in structured analysis by which a complex application problem is divided into its parts for individual analysis. A technique to simplify management of application complexity.

document A general analysis and design task that is performed to create useful documents from graphics and supporting text either manually or with computer-based tools.

domain A conceptual area of interest. In organizational reengineering the domains are data, process, network, and technology; in database, a domain is the set of allowable values for an individual attribute.

downsizing The shifting of processing and data from mainframes to some other, less expensive environment, usually to a multiuser midsize machine, such as an IBM AS400, or to a LAN of PCs.

efferent flows In structured design, the output-oriented processes which write, display, and print data.

elaboration A general analysis and design task that is performed to define the details of each thing identified.

elementary process *See* atomic process.

encapsulation In object orientation, a property of programs that describes the complete integration of data with legal processes relating to the data.

entity In information engineering, some person, object, concept, application, or event from the real world about which we want to maintain data; includes attributive, associative, and fundamental entity types.

entity relationship diagram In information engineering, a graphical representation of the normalized data environment and data scope of the application.

entity/process matrix (CRUD) A two-dimensional table of entities and business processes that identifies the functions each process is allowed to perform on data, including create, retrieve, update and delete (e.g., CRUD).

equifinality Many paths lead to the same goal.

estimating Use of expertise to define project work effort, including use of algorithms, models, delphi techniques, expert opinion, function points, top-down, and bottom-up techniques.

ethical dilemma Any situation in which a decision results in unpleasant consequences requiring moral reasoning.

ethics The branch of philosophy that studies moral judgment and reasoning.

exception handling The extent to which programs can be coded to intercept and handle program errors without abending a program.

executable units In structured design for non-real-time languages an execute unit is a link-edited load module. For real-time languages, an execute unit identifies modules that can reside in memory at the same time and are related, usually by mutual communication.

executive information system (EIS) A spinoff from DSS. EIS applications support executive decision making and provide automated environmental scanning capabilities.

expert systems (ES) application Computer applications that automate the knowledge and reasoning capabilities of one or more experts in a specific domain.

external entity In structured analysis, a person, place or thing with which the application interacts.

facilitator A specially trained individual who runs JAD, fast-track, JRP, or walk-through sessions.

factoring In structured design, the process during which net outputs from a DFD are used to determine the initial structure of the structure chart.

fast track A different name for JAD.

feasibility The analysis of risks, costs, and benefits relating to technology, economics, and using organizations.

field format The characteristics of individual fields or values of fields on a screen display, including size, font, style, color, and blink for individual field values, and coding options for field labels.

flash rate Blinking speed for a screen display item.

flicker fusion A physical phenomenon that causes us to see constant light when the flash rate is very high.

footer The lower portion of a screen.

form follows function A principle from architecture which, when applied to structured analysis, defines application functions that transform data as the defining characteristic of applications.

frozen specification A specification that cannot be changed without specific user/sponsor approval with accompanying modification of budget and cost.

function A small program that is self-contained and performs a well-defined, limited procedure.

function key A programmable computer keyboard key used to provide a shortcut command.

function point analysis A method of defining the complexity of an application by systematic definition of global application characteristics.

functional decomposition The division of processes into modules.

functional screen A screen at which the application processes are performed.

generalization class In object orientation, defines a group of similar objects.

global data Data variables and constants that are accessible to any module in the application.

globalization The movement of otherwise local businesses into world markets.

goals of software engineering To build a quality product through a quality process.

group decision support systems (GDSS) A special type of DSS applications. GDSS provide an historical memory of the decision process in support of groups of decision makers who might be geographically dispersed.

hardware installation plan A plan identifying work required, environmental changes (e.g., air conditioning), work responsibilities, timing of materials and labor, and scheduling of tasks as they relate to the installation of computer and other information technology equipment.

hierarchical structure chart In structured design, a graphical input-process-output view of the application that reflects the DFD partitioning.

human interface The means by which an application communicates to its human users.

Humphrey's maturity framework A framework adapted to compare methodologies as having reached initial, repeatable, managed, defined, or optimizing levels of sophistication.

hypermedia Software that allows any number of associative relationships to be defined for a given item; supports audio, video, image, graphics, text and data.

I/O bound In structured design, a structure chart in which the skew is equally balanced between input and output, but processing is a small part of the application.

identification A general analysis and design task that is performed to find the focal things that belong in analysis and how logical requirements will work in the target computer environment in design.

implementation The period of time during which a software product is integrated into its operational environment and is phased into production use. Implementation includes the completion of data conversion, installation, and training.

information engineering (IE) A data-oriented methodology that borrows from both practice and theoretical research to support the development of enterprise level

plans through to individual project developments. IE concentrates on business understanding, assumes user involvement, and covers more phases of the SPLC than most other methodologies.

information hiding A program design principle by which only data needed to perform a function is made available to that function.

information systems architecture framework (ISA) Zachman's method of defining distinct architectures relating business context to application context at progressively more detailed levels.

information systems methodology framework A standard for comparing methodologies based on their representation forms and types of information supported.

information systems plan (ISP) An enterprise level analysis of data, processes, and technology that includes manual or automated work to capture a snapshot of the enterprise in order to define and prioritize applications for development.

inheritance In object orientation, a property that allows the generic description of objects which are then reused by related objects.

input-bound In structured design, a structure chart in which the skew is on the input side.

instance In information engineering, a specific occurrence of an entity, e.g., entity = customer, instance = Sam Jones.

integration test Tests that verify the logic and processing for suites of modules that perform some activity, verifying communications between them.

interdependence A way of describing the interrelationships between organizations; includes pooled, sequential, and reciprocal relationships.

interface Some person, application, or organization with which an application must communicate.

iterative project life cycle A cyclic repetition of analysis, design, and implementation activities.

joint application development/design (JAD) A special form of structured meeting during which user representatives, application developers, and a facilitator meet continuously over several days to define the functional requirements of an application.

language constructs Features of computer languages that determine what and how operations on data are carried out.

learn-as-you-go project life cycle An approach to the development life cycle that assumes every project is so unique that it has no prior precedent upon which to base activities.

legacy data Data used by outdated applications that are required to be maintained for business records.

legacy systems Applications that are in a maintenance phase but are not ready for retirement.

leveled set of DFDs Verified balanced set of entities, data flows and processes within a hierarchic DFD diagram set.

leverage point Some business or application activity from which a competitive advantage can be gained.

librarian A person working with an application development or maintenance team to provide librarian services relating to maintenance of documentation, code objects, reusable modules, etc.

local data Data variables and constants that are used only within a given module.

logical data model An abstract definition of data in an organization that describes the way a user views data

maintenance The changes made to the logic of the system and programs to fix errors (perfective), provide for business changes (adaptive), or make the software more efficient.

make/buy decision The tradeoff between building the item in-house or purchasing it elsewhere.

memory management The ability of a program to allocate more computer random-access memory (RAM) as required.

menu Lists of options on a screen from which a selection is made.

menu structure In information engineering, a diagram translating process alternatives into a hierarchy of menu selection options for an application.

message In object orientation, the unit of communication between two objects.

meta-class In object orientation, classes whose instances are other classes.

meta-data Data about data that gives meaning to data and is information about data, e.g., data type= integer.

meta-meta-data Information about the meta-data that describes its allowable use to the application, e.g., type=hardware.

methodology Procedures, policies, and processes used to direct the activities of each phase of a software life cycle, including process, data, object, semantic, or none.

model A conceptual definition of something, e.g., logical data, physical data, business processes, etc.

modularity The structured design principle that calls for design of small, self-contained units that should lead to maintainability.

module *See* program package.

morphology Form or shape. In structured design, morphology refers to the shape of a structure chart.

multimedia A term that describes the integration of object orientation, data base, and storage technologies in one environment.

multitasking In object orientation, the simultaneous execution of sets of processes.

multitasking objects In object orientation, objects that track and control the execution of multiple threads of control.

multiple inheritance In object orientation, the ability to share attributes and processes from multiple class/objects.

net present value (NPV) A mathematical method of comparing multiperiod projects that equalizes the cost estimates by accounting for the time value of money.

normalization The refinement of data relationships to remove repeating information, partial key dependencies, and nonkey dependencies.

object In object orientation, an instance of the class definition.

object-based A design that is based on object thinking, but is not object-oriented in its implementation.

object-oriented analysis A methodology for analyzing data objects and their allowable processes as encapsulated and having inheritable properties.

object-oriented methodology An approach to system life cycle development that takes a top-down, encapsulated view of data objects, their allowable actions, and the underlying communication requirement to define an application architecture.

off-site storage A location usually 200+ miles away from the main computing site used to store backup copies of databases, software, etc.

on-line application Applications that provide interactive processing to the user with or without immediate file update.

operations The daily processing of a computer application.

option selection The choice for application navigation from among menus, command languages, and windows used to get to a functional screen.

organizational reengineering An evaluation of an organization's data, processes, technologies, and communications needs to ensure that its goals as stated in its mission statement are met.

out-of-the-box thinking Examining a problem or issue without respect to the current context to determine novel approaches to resolving the issue.

output-bound In structured design, a structure chart in which the skew is on the output side.

package specification In object orientation, defines the public interface for both data and processes for each object, and the private implementations and language to be used. Similar to a program specification in non-object methodologies.

packages In object orientation, a set of modules relating to an object which might be modularized for execution.

part class In object orientation, defines a component of a whole class.

partitioning The basic activity of dividing processes into modules.

peer-to-peer networking A computer communications network in which intelligent sharing of resources and data across multiple processors is taking place.

persistent object An object that is maintained over time, a database item.

physical data model The physical definition of data, describing its layout for a particular hardware device.

physical database design The actions required to map a logical database to storage devices in a specific DBMS implementation environment.

physical input and output The movement of data between external computer (e.g., disk) storage and random-access memory (RAM). I/O statements (e.g., read/write) may be record-oriented, set-oriented, or array-oriented.

polymorphism In object orientation, the ability to have the same process take different forms when associated with different objects.

presentation format The method chosen for summarizing information for screen display, including analog, digital, binary graphic, bar chart, column chart, point plot, pattern display, mimic display, text, and text forms.

primary key A unique set of values comprised of one or more attributes identifying an entity, an object, or a database item, depending on the context.

private part (of a class/object) In object orientation, defines local, object-only data and the specific procedures each action takes.

problem space In object orientation, identifies objects/processes that are required to describe the problem, but are not required to describe the solution.

problem-domain objects In object orientation, the class/objects and objects defined during analysis and describing the application functions.

process The sequence of instructions or conjunction of events that operate on data.

process data flow diagram (PDFD) In information engineering, a graphical representation of processes and the data and event triggers that initiate processing. The PDFD is the basis for action diagrams in IE design.

process dependency diagram In information engineer-

ing, a graphical representation of the sequence and types of relationships among processes.

process diagram In object orientation, graphical representation of the hardware environment showing process assignments to hardware.

process model A conceptual description of the business processes of an organization.

process-oriented analysis A method of analyzing application transformation processing as the defining characteristic of applications.

process-oriented methodology Methodologies that take a structured, top-down approach to evaluating problem processes and the data flows with which they are connected.

process/location matrix In data distribution analysis, a table containing processes and, for each location under analysis, the major and minor involvement in performing each process.

program package In structured design, one or more called modules, and functions, and in-line code that will be an execute unit to perform some atomic process. Also called a program unit.

program specification A description of a program's purpose, process requirements, the logical and physical data definitions, input and output formats, screen layouts, constraints, and special processing considerations that might complicate the program.

program template Standard code that performs a simple function.

program unit *See* program package.

programming The process of designing and describing an algorithm to solve a class of problems.

project life cycle The breakdown of work for initiation, development, maintenance, and retirement of an application.

project manager (PM) The person with primary responsibility for organization liaison, project staff management, and project monitoring and control. The PM also performs activities with the SE including project planning, assigning staff to tasks, and selecting from among application approaches.

project plan A summary of the project planning effort that identifies the work breakdown tasks, their interrelationships, and the estimated time to complete each task.

prototyping The building of a subset of an application to assist in requirements definition, to test a proof of concept, or to provide a partial solution to a particular problem.

pseudo-code Specification of processing using the syntax from a programming language in abbreviated form for easy translation.

public part (of a class/object) In object orientation, defines what data are available in the object and the allowable actions of the object.

quality assurance (QA) Any review of an application development work product by a person who is not a member of the project team to determine whether or not the analysis requirements are satisfied.

quality assurance (QA) test A test by an outside agent to determine that functional requirements are satisfied. The outside agent can be a user or a user representative.

query application Another term for data analysis applications.

question Words phrasing an asking sentence that can be open-ended, without a specific answer, or closed-ended and requesting a yes/no or very short specific answer.

reentrant A property of a module that allows it to be shared by several tasks concurrently.

real-time application Applications that process transactions and/or events during the actual time that the related physical (real-world) process takes place.

recovery The process of restoring a previous version of data (or software) from a backup copy to active use following some damage to, or loss of, the previously active copy.

recursive A property of modules such that they call themselves or call another module that, in turn, calls them.

regression test Customized tests to check that changes to an application have not caused it to regress to some state of unacceptable quality.

relationship In entity-relationship diagrams, mutual association between two or more entities. It is shown as a line connecting the entities; includes one-to-one, one-to-many, and many-to-many relationship cardinalities.

repository A data dictionary in a CASE environment that contains not only data, file, process, entity, and data flow definitions, but also contains definitions of all graphical forms, their contents, and allowable definitions (e.g., entity-relationship diagram, process decomposition, etc.)

request for information (RFI) A formal request for information on some product that usually precedes the RFP process.

request for proposal (RFP) A written request for bids on some product, providing formal requirements, ground rules for responses, and, usually, a standard format for the proposal responses.

request for quotation (RFQ) *See* request for proposal.

responsiveness The underlying time orientation of the application as batch, on-line, or real-time.

retirement The period of time in the software life cycle during which support for a software product is terminated.

reusability Also called serial reusability, a property of a module such that many tasks, in sequence, can use the module without its having to be reloaded into memory for each use.

reusable components Programs, functions, or program fragments that are specially designed for use in more than one program.

reusable module A small, single function, well-defined, and standardized program module that can be used as a called routine or as a copy book in COBOL.

reverse engineering *See* software reengineering.

review A general analysis and design task that is to analyze quality of the reviewed product.

risk Events that would prevent the completion of, in this case, an application development alternative in the manner or time desired.

risk assessment A method of determining possible sources of events that might jeopardize completion of the application.

round-trip gestalt In object orientation, an iterative approach to detailed design in which prototypes are built in an incremental development life cycle.

scaffolding Extra code to support the stubs, partial modules, and other pieces of the application, usually created to support top-down testing.

scheduling In object orientation, the process of assigning execution times to a list of processes.

scheduling objects In object orientation, objects that define sequential, concurrent-asynchronous (i.e., independent), or concurrent-synchronous (i.e., dependent) processes.

scope Definition of the boundaries of the project: what is in the project and what is outside of the project.

scope of effect In structured design, the collection of modules that are conditionally processed based on decisions by the module under review.

screen formats The general layout of a screen display including definition of the menu/selection format, the presentation format, and individual field formats.

security plan A plan identifying the physical, data, and application means used to protect corporate information and technology assets.

semantic methodology Methodologies used in the automation of artificial intelligence (AI) applications, including recognizing, reasoning, and learning applications.

sequential development life cycle (SDLC) A subcycle of the SPLC, including phases for analysis, conceptual design, design, implementation, testing, installation and checkout, and ending with delivery of an operational application.

sequential project life cycle (SPLC) The period of time from inception to retirement of a computer application. Phases in SPLC include: initiation, problem definition, feasibility, requirements analysis, conceptual design, design, code/unit test, testing, installation/checkout, operations and maintenance, and retirement.

server object In object orientation, an object that performs a requested process (i.e., client/server processing).

service objects In object orientation, manage application operations, including synchronizing, scheduling or multitasking objects, as required.

skew In structured design, a term to describe the lopsidedness of a program structure chart.

social methodology An approach to SDLC that attends to social and job-related needs of individuals who supply or receive or use data from the application being built.

software engineer Skilled professionals who have a variety of skills that they apply using engineering-like techniques to the definition, design, and implementation of computer applications.

software engineering Systematic development, operation, maintenance, and retirement of software.

software reengineering The reverse analysis of an old application to conform to a new methodology, usually information engineering or object orientation.

solution space In object orientation, identifies objects/processes that are required both to describe the problem, and to develop a solution.

specialization class In object orientation, a subclass that reflects an *is-a* relationship, defining a more detailed description of the gen class.

sponsor A manager who pays for the project and acts as its champion.

stakeholders People and organizations affected by an application.

state In object orientation, a specific configuration of attribute values of an object.

state transition diagram In object orientation, defines allowable changes for data objects.

structure chart In structured design, a hierarchic, input-process-output view of the application that reflects the DFD partitioning.

structured decomposition A technique for coping with

application complexity through the principle of "divide and conquer."

structured design The art of designing system components and the interrelationships among those components in the best possible way to solve some well-specified problem.

structured English Language-independent specification of processing using a restricted subset of English.

structured systems analysis A process-oriented analysis methodology that defines a top-down method of defining and graphically documenting procedural aspects of applications.

subsystem design Subphase of the design phase during which the application is divided into relatively independent chunks for detailed specification.

subsystem test *See* integration test.

subdomain In object orientation, application design is seen as taking place in four distinct domains: human, hardware, software, and data. Encapsulated class/objects (or a subset of them) are assigned to one of the subdomains during design.

subject area data base In information engineering, a database that supports one or more business functions.

supplier object In object orientation, an object that performs a requested process.

synchronizing The coordination of simultaneous events.

synchronizing objects In object orientation, objects that provide a rendezvous for two or more processes to come together after concurrent operations.

synthesis A general analysis and design task that is performed to build a unified view of the application, reconciling any parts that do not fit, and representing requirements in graphic form.

system test A test to verify that the functional specifications are met, that the human interface operates as desired, and that the application works in the intended operational environment within its constraints.

systems theory A theory defining inputs as fed into processes to produce outputs with feedback providing a check on the process.

task profile A description of the job(s) to be performed using a computer application.

technology transfer The large-scale introduction of a new technology to some previously nontechnical environment.

test case Individual transactions or data records that cause logic to be tested.

test plan Documents the strategy, type, cases, and scripts for testing some component of an application. All of the plans together comprise the test plan for the application.

test script Documents the interactive dialogue that takes place between user and application, and the changes that result from the dialogue for on-line and real-time applications.

test strategy The overall approach to testing at some level, used to guide the tester in developing test cases. Test strategies are white-box, black-box, bottom-up or top-down. They are not mutually exclusive and are usually used in combination.

testing A phase of the SDLC during which the application is exercised for the purpose of finding errors.

thread of control In object orientation, a set of potentially concurrent processes. Usually, a single thread of control relates to a single user or a single application-level transaction.

time events In object orientation, the business, system, or application occurrences that cause processes to be activated.

time-event diagram In object orientation, a diagram depicting the relationships among processes that are triggered by related events or have constraints on processing time.

top-down A perspective that begins the activity (e.g., analysis or design) at an abstract level and proceeds to more detailed sublevels.

top-down development A way of thinking about problems that begins at a high level of abstraction and works through successively more detailed levels.

top-down testing A testing strategy that assumes that critical control code and functions will be developed and tested first and followed by secondary functions and supporting functions.

transaction analysis In structured design, a method of analyzing generic activities by transaction type to develop structure charts of processing.

transaction processing application (TPA) Applications that support the day-to-day operations of a business, e.g., order processing.

transaction volume matrix In data distribution analysis, a table summarizing volume of transaction traffic by location.

transform analysis In structured design, a method of identifying the central transform through analysis of afferent and efferent flows.

trigger In information engineering, some data or event that causes a business process to execute.

type 1 error Defines code that does not do what it is supposed to do; errors of omission.

type 2 errors Defines code that does something it is not supposed to do; errors of commission.

type checking The extent to which a language enforces matching of specific data definitions in mathematical and logical operations; includes typeless, mixed-mode, pseudo-strong, and strong.

unit test Tests performed by the author on each of the code units.

user-managed application development The overall management of application development by the user/sponsor of the project to foster a business partner relationship with IS staff and to improve the quality of the finished product.

user profile A description of the user(s) of a computer application.

utility object *See* service object.

validation A review to establish the fitness or quality of a software product for its operational purpose.

vendor response A proposal in response to an RFP.

verification A review to establish the correctness of correspondence between a software product and its specification.

walk-through A formal, structured meeting held to review work products and find problems.

white box A testing strategy that uses logic specifications to generate variations of processing and to predict the resulting outputs. White-box strategies look at specific logic to verify how it works, including various levels of logic tests, mathematical proofs, and cleanroom testing.

whole class In object orientation, defines a composed object type.

windows A form of direct manipulation of the environment that combines full screen, icon symbols, menus, and point-and-pick devices to simplify the human interface by making it represent a metaphorical desk environment.

work around A rethinking of an application design caused by limitations of the language, package, or target environment.

working set The minimal, real random-access memory (RAM) required by software when it is running.

INDEX