Some software developers prefer a rapid prototyping approach, whereby a small portion of the software is initially developed and evaluated through use. The software is gradually made robust through incremental improvements to the specification, design, and implementation. In contrast, with the life cycle approach, software is fully specified, fully designed, then fully implemented.

Other authors have made various arguments for and against the merits of rapid prototyping. These arguments do not concern us here, since the OMT methodology applies equally well in either case. The notion of objects provides a solid basis for specifying, designing, and implementing software in one pass, or gradually building software through multiple passes. Object-oriented software development is anchored to real-world objects; software objects can easily assume additional behavior associated with their real-world counterparts. In contrast, rapid prototyping is more difficult with function-based software development, since a functional breakdown suitable for a prototype may not be suitable for a full implementation.

7.2 THE OMT METHODOLOGY

The OMT methodology consists of several phases:

Analysis (Chapter 8) is concerned with understanding and modeling the application and the domain within which it operates. The initial input to the analysis phase is a problem statement which describes the problem to be solved and provides a conceptual overview of the proposed system. Subsequent dialogue with the customer and real-world background knowledge are additional inputs to analysis. The output from analysis is a formal model that captures the three essential aspects of the system: the objects and their relationships, the dynamic flow of control, and the functional transformation of data subject to constraints.

The overall architecture of the system is determined during *System Design* (Chapter 9). Using the object model as a guide, the system is organized into subsystems. Concurrency is organized by grouping objects into concurrent tasks. Overall decisions are made about interprocess communication, data storage, and implementation of the dynamic model. Priorities are established for making design trade-offs.

During the *Object Design* phase (Chapter 10), the analysis models are elaborated, refined, and then optimized to produce a practical design. During object design there is a shift in emphasis from application concepts towards computer concepts. First the basic algorithms are chosen to implement each major function of the system. Based on these algorithms, the structure of the object model is then optimized for efficient implementation. The design must also account for concurrency and dynamic control flow as determined during system design. The implementation of each association and attribute is determined. Finally, the subsystems are packaged into modules.

Chapter 11 summarizes the OMT methodology presented in Chapters 8, 9, and 10. Chapter 12 compares the OMT methodology with other software engineering methodologies.

7.3 IMPACT OF AN OBJECT-ORIENTED APPROACH

The OMT methodology is an object-oriented software construction approach which differs from traditional software development approaches. These differences affect the process of software development, and ultimately the software product, itself.

Shifting of development effort into analysis. An object-oriented approach moves much of the software development effort up to the analysis phase of the life cycle. It is sometimes disconcerting to spend more time during analysis and design, but this extra effort is more than compensated by faster and simpler implementation. Because the resulting design is cleaner and more adaptable, future changes are much easier.

Emphasis on data structure before function. An object-oriented approach focuses attention on data structure instead of the functions to be performed. This change of emphasis gives the development process a more stable base and allows the use of a single unifying software concept throughout the process: the concept of an object. All other concepts, such as functions, relationships, and events, are organized around objects so that information recorded during analysis is not lost or transformed when design and implementation take place.

The data structures of an application and the relationships between them are much less vulnerable to changing requirements than the operations performed on the data. Organizing a system around objects rather than around functions gives the development process a stability that is lacking in function-oriented approaches. Encapsulated objects, with public interfaces that hide their private internal implementation, are further protected from the effects of change.

Seamless development process. Because an object-oriented approach defines a set of problem-oriented objects early in the project and continues to use and extend these objects throughout the development cycle, the separation of life cycle phases is much less distinct. In the Object Modeling Technique, the object model developed during analysis is used for design and implementation, and work is channeled into refining the model at progressively more detailed levels rather than converting from one representation into another. The process is seamless because there are no discontinuities in which a notation at one phase is replaced by a different notation at another phase.

Iterative rather than sequential. Although the description of the Object Modeling Technique is of necessity linear, the actual development process is iterative. The seamlessness of object-oriented development makes it easier to repeat the development steps at progressively finer levels of detail. Each iteration adds or clarifies features rather than modifies work that has already been done, so there is less chance of introducing inconsistencies and errors.

7.4 CHAPTER SUMMARY

A software engineering methodology consists of a process for organized development based on a set of coordinated techniques. The OMT methodology is based on the development of a three-part model of the system, which is then refined and optimized to constitute a design. The object model captures the objects in the system and their relationships. The dynamic

9

System Design

After you have analyzed a problem, you must decide how to approach the design. *System design* is the high-level strategy for solving the problem and building a solution. System design includes decisions about the organization of the system into subsystems, the allocation of subsystems to hardware and software components, and major conceptual and policy decisions that form the framework for detailed design.

The overall organization of a system is called the *system architecture*. There are a number of common architectural styles, each of which is suitable for certain kinds of applications. One way to characterize an application is by the relative importance of its object, dynamic, and functional models. Different architectures place differing emphasis on the three models.

In this chapter you will learn about the many aspects of an application problem that you should consider when formulating a system design. We also present several common architectural styles that you can use as a starting point for your designs. This list of architectural styles is not meant to be complete; new architectures can always be invented or adapted as needed. The treatment of system design in this chapter is intended for small to medium software development efforts; large complex systems, involving more than about ten developers, are limited by human communication issues and require a much greater emphasis on design logistics. Most of the suggestions in this chapter are suitable for non-object-oriented as well as object-oriented systems.

9.1 OVERVIEW OF SYSTEM DESIGN

During analysis, the focus is on *what* needs to be done, independent of *how* it is done. During design, decisions are made about how the problem will be solved, first at a high level, then at increasingly detailed levels.

System design is the first design stage in which the basic approach to solving the problem is selected. During system design, the overall structure and style are decided. The *system architecture* is the overall organization of the system into components called *subsystems*.

The architecture provides the context in which more detailed decisions are made in later design stages. By making high-level decisions that apply to the entire system, the system designer partitions the problem into subsystems so that further work can be done by several designers working independently on different subsystems.

The system designer must make the following decisions:

- Organize the system into subsystems [9.2]
- Identify concurrency inherent in the problem [9.3]
- Allocate subsystems to processors and tasks [9.4]
- Choose an approach for management of data stores [9.5]
- Handle access to global resources [9.6]
- Choose the implementation of control in software [9.7]
- Handle boundary conditions [9.8]
- Set trade-off priorities [9.9]

Often the overall architecture of a system can be chosen based on its similarity to previous systems. Certain kinds of system architecture are useful for solving several broad classes of problems. Section 9.10 surveys several common architectures and describes the kinds of problems for which they are useful. Not all problems can be solved by one of these architectures but many can. Many other architectures can be constructed by combining these forms.

9.2 BREAKING A SYSTEM INTO SUBSYSTEMS

For all but the smallest applications, the first step in system design is to divide the system into a small number of components. Each major component of a system is called a *subsystem*. Each subsystem encompasses aspects of the system that share some common property—similar functionality, the same physical location, or execution on the same kind of hardware. For example, a spaceship computer might include subsystems for life support, navigation, engine control, and running scientific experiments.

A subsystem is not an object nor a function but a package of classes, associations, operations, events, and constraints that are interrelated and that have a reasonably well-defined and (hopefully) small interface with other subsystems. A subsystem is usually identified by the *services* it provides. A *service* is a group of related functions that share some common purpose, such as I/O processing, drawing pictures, or performing arithmetic. A subsystem defines a coherent way of looking at one aspect of the problem. For example, the file system within an operating system is a subsystem; it comprises a set of related abstractions that are largely, but not entirely, independent of abstractions in other subsystems, such as the memory management subsystem or the process control subsystem.

Each subsystem has a well-defined interface to the rest of the system. The interface specifies the form of all interactions and the information flow across subsystem boundaries but does not specify how the subsystem is implemented internally. Each subsystem can then be designed independently without affecting the others.

Subsystems should be defined so that most interactions are within subsystems, rather than across subsystem boundaries, in order to reduce the dependencies among the subsystems. A system should be divided into a small number of subsystems; 20 is probably too many. Each subsystem may in turn be decomposed into smaller subsystems of its own. The lowest level subsystems are called *modules*, as discussed in Chapter 3.

The relationship between two subsystems can be *client-supplier* or *peer-to-peer*. In a client-supplier relationship, the client calls on the supplier, which performs some service and replies with a result. The client must know the interface of the supplier, but the supplier does not have to know the interfaces of its clients because all the interactions are initiated by clients using the supplier's interface. In a peer-to-peer relationship, each of the subsystems may call on the others. A communication from one subsystem to another is not necessarily followed by an immediate response. Peer-to-peer interactions are more complicated because the subsystems must know each other's interfaces. Communications cycles can exist that are hard to understand and liable to subtle design errors. Look for supplier-client decompositions whenever possible because a one-way interaction is much easier to build, understand, and change than a two-way interaction.

The decomposition of systems into subsystems may be organized as a sequence of horizontal *layers* or vertical *partitions*.

9.2.1 Layers

A layered system is an ordered set of virtual worlds, each built in terms of the ones below it and providing the basis of implementation for the ones above it. The objects in each layer can be independent, although there is often some correspondence between objects in different layers. Knowledge is one-way only: A subsystem knows about the layers below it, but has no knowledge of the layers above it. A supplier-client relationship exists between lower layers (providers of services) and upper layers (users of services).

In an interactive graphics system, for example, windows are made from screen operations, which are implemented using pixel operations, which execute as device I/O operations. Each layer may have its own set of classes and operations. Each layer is implemented in terms of the classes and operations of lower layers.

Layered architectures come in two forms: closed and open. In a *closed architecture*, each layer is built only in terms of the immediate lower layer. This reduces the dependencies between layers and allows changes to be made most easily because a layer's interface only affects the next layer. In an *open architecture*, a layer can use features of any lower layer to any depth. This reduces the need to redefine operations at each level, which can result in a more efficient and compact code. However, an open architecture does not observe the principle of information hiding. Changes to a subsystem can affect any higher subsystem, so an open architecture is less robust than a closed architecture. Both kinds of architectures are useful; the designer must weigh the relative value of efficiency and modularity.

Usually only the top and bottom layers are specified by the problem statement: The top is the desired system, the bottom is the available resources (hardware, operating system, existing libraries). If the disparity between the two is too great (as it often is), then the system

designer must introduce intermediate layers to reduce the conceptual gap between adjoining layers.

A system constructed in layers can be ported to other hardware/software platforms by rewriting one layer. It is a good practice to introduce at least one layer of abstraction between the application and any services provided by the operating system or hardware. Define a layer of interface classes providing logical services and map them onto the concrete services that are system-dependent.

9.2.2 Partitions

Partitions vertically divide a system into several independent or weakly-coupled subsystems, each providing one kind of service. For example, a computer operating system includes a file system, process control, virtual memory management, and device control. The subsystems may have some knowledge of each other, but this knowledge is not deep, so major design dependencies are not created.

A system can be successively decomposed into subsystems using both layers and partitions in various possible combinations: Layers can be partitioned and partitions can be layered. Figure 9.1 shows a block diagram of a typical application, which involves simulation of the application and interactive graphics. Most large systems require a mixture of layers and partitions.

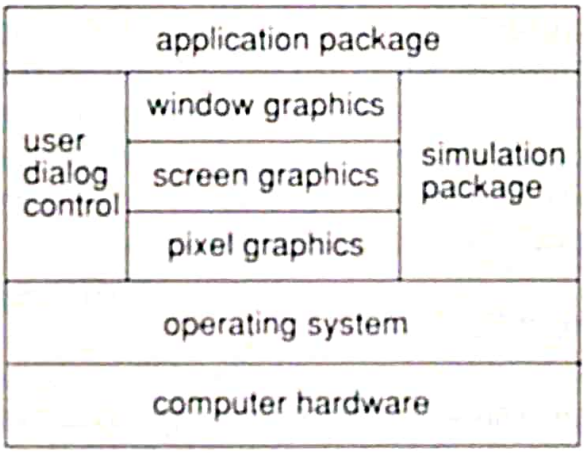


Figure 9.1 Block diagram of a typical application

9.2.3 System Topology

When the top-level subsystems are identified, the designer should show the information flow among subsystems with a data flow diagram (see the examples of architectural styles in Section 9.10). Sometimes, all subsystems interact with all other subsystems, but often the flow is simpler. For example, many computations have the form of a pipeline; a compiler is an example. Other systems are arranged as a star, in which a master subsystem controls all interactions with other subsystems. Use simple topologies when they are suitable for a problem to reduce the number of interactions among subsystems.

9.3 IDENTIFYING CONCURRENCY

In the analysis model, as in the real world and in hardware, all objects are concurrent. In an implementation, however, not all software objects are concurrent because one processor may support many objects. In practice, many objects can be implemented on a single processor if the objects cannot be active together. One important goal of system design is to identify which objects must be active concurrently and which objects have activity that is mutually exclusive. The latter objects can be folded together in a single thread of control, or task.

9.3.1 Identifying Inherent Concurrency

The dynamic model is the guide to identifying concurrency. Two objects are inherently concurrent if they can receive events at the same time without interacting. If the events are unsynchronized, the objects cannot be folded onto a single thread of control. For example, the engine and the wing controls on an airplane must operate concurrently (if not completely independently). Independent subsystems are desirable because they can be assigned to different hardware units without any communication cost.

Two subsystems that are inherently concurrent need not necessarily be implemented as separate hardware units. The purpose of hardware interrupts, operating systems, and tasking mechanisms is to simulate logical concurrency in a uniprocessor. Physically-concurrent input must of course be processed by separate sensors, but if there are no timing constraints on response then a multitasking operating system can handle the computation.

Often the problem statement specifies that objects must be implemented as distinct hardware units. For example, if the ATM statement from Chapter 8 contained the requirement that each machine should continue to operate locally in the event of a central system failure (perhaps with reduced limits on transactions), then we would have no choice but to include a CPU in each ATM machine with a full control program.

9.3.2 Defining Concurrent Tasks

Although all objects are conceptually concurrent, in practice many objects in a system are interdependent. By examining the state diagrams of individual objects and the exchange of events among them, many objects can often be folded together onto a single thread of control. A *thread of control* is a path through a set of state diagrams on which only a single object at a time is active. A thread remains within a state diagram until an object sends an event to another object and waits for another event. The thread passes to the receiver of the event until it eventually returns to the original object. The thread splits if the object sends an event and continues executing.

On each thread of control, only a single object at a time is active. Threads of control are implemented as *tasks* in computer systems. For example, while the bank is verifying an account or processing a bank transaction, the ATM machine is idle. If the ATM is controlled directly by a central computer, then the ATM object can be folded together with the bank transaction object as a single task.

9.4 ALLOCATING SUBSYSTEMS TO PROCESSORS AND TASKS

Each concurrent subsystem must be allocated to a hardware unit, either a general purpose processor or a specialized functional unit. The system designer must:

- Estimate performance needs and the resources needed to satisfy them.
- Choose hardware or software implementation for subsystems.
- Allocate software subsystems to processors to satisfy performance needs and minimize interprocessor communication.
- Determine the connectivity of the physical units that implement the subsystems.

9.4.1 Estimating Hardware Resource Requirements

The decision to use multiple processors or hardware functional units is based on a need for higher performance than a single CPU can provide. The number of processors required depends on the volume of computations and the speed of the machine. For example, a military radar system generates too much data in too short a time to handle in a single CPU, even a very large one. The data must be digested by many parallel machines before the final analysis about a threat can be performed.

The system designer must estimate the required CPU processing power by computing the steady state load as the product of the number of transactions per second and the time required to process a transaction. The estimate will usually be imprecise. Often some experimentation is useful. The estimate should then be increased to allow for transient effects, due to both random variations in load as well as synchronized bursts of activity. The amount of excess capacity needed depends on the acceptable rate of failure due to insufficient resources.

9.4.2 Hardware-Software Trade-offs

Hardware can be regarded as a rigid but highly optimized form of software. The object-oriented view is a good way of thinking about hardware. Each device is an object that operates concurrently with other objects (other devices or software). The system designer must decide which subsystems will be implemented in hardware and which in software. Subsystems are implemented in hardware for two main reasons:

- Existing hardware provides exactly the functionality required. Today it is easier to buy a floating point chip than to implement floating point in software. Sensors and actuators must be hardware, of course.
- Higher performance is required than a general purpose CPU can provide, and more efficient hardware is available. For example, chips that perform the Fast Fourier Transform (FFT) are widely used in signal processing applications.

Much of the difficulty of designing a system comes from meeting externally-imposed hardware and software constraints. Object-oriented design provides no magic solution, but the

external packages can be modeled nicely as objects. You must consider compatibility, cost, and performance issues. You should also think about flexibility for future changes, both design changes and future product enhancements. Providing flexibility costs something; the architect must decide how much it is worth.

9.4.3 Allocating Tasks to Processors

The tasks for the various software subsystems must be allocated to processors. Tasks are assigned to processors because:

- Certain tasks are required at specific physical locations, to control hardware or to permit independent or concurrent operation. For example, an engineering workstation needs its own operating system to permit operation when the interprocessor network is down.
- The response time or information flow rate exceeds the available communication bandwidth between a task and a piece of hardware. For example, high performance graphics devices require tightly-coupled controllers because of their high internal data generation rates.
- Computation rates are too great for a single processor, so tasks must be spread among several processors. Those subsystems that interact the most should be assigned to the same processor to minimize communication costs. Independent subsystems should be assigned to separate processors.

9.4.4 Determining Physical Connectivity

After determining the kinds and relative numbers of physical units, the system designer must choose the arrangement and form of the connections among the physical units. The following decisions must be made:

- Choose the topology of connecting the physical units. Associations in the object model often correspond to physical connections. Client-supplier relationships in the functional model also correspond to physical connections. Some connections may be indirect, of course, but the designer must attempt to minimize the connection cost of important relationships.
- Choose the topology of repeated units. If several copies of a particular kind of unit or group of units are included for performance reasons, their topology must be specified. The object model and functional model are not useful guides because the use of multiple units is primarily a design optimization not required by analysis. The topology of repeated units usually has a regular pattern, such as a linear sequence, a matrix, a tree, or a star. The designer must consider the expected arrival patterns of data and the proposed parallel algorithm for processing it.
- Choose the form of the connection channels and the communication protocols. The system design phase may be too soon to specify the exact interfaces among units, but the general interaction mechanisms and protocols must usually be chosen. For example, in-

teractions may be asynchronous, synchronous, or blocking. The bandwidth and latency of the communication channels must be estimated and the correct kind of connection channels chosen.

Even when the connections are logical and not physical, the connections must be considered. For example, the units may be tasks within a single operating system connected by interprocess communication calls. On most operating systems, such IPC calls are much slower than subroutine calls within the same program and may be impractical for certain time-critical connections. In that case, the tightly-linked tasks must be combined into a single task and the connections made by simple subroutine calls.

9.5 MANAGEMENT OF DATA STORES

The internal and external data stores in a system provide clean separation points between subsystems with well-defined interfaces. In general each data store may combine data structures, files, and databases implemented in memory or on secondary storage devices. For example, a personal computer application may use memory data structures, a RAM disk, and a hard disk. An accounting system may use a database and files to connect subsystems. Different kinds of data stores provide various trade-offs between cost, access time, capacity, and reliability.

Files are a cheap, simple, and permanent form of data store. However, file operations are low level and applications must include additional code to provide a suitable level of abstraction. File implementations vary for different computer systems, so portable applications must carefully isolate file system dependencies. Implementations for sequential files are mostly standard, but commands and storage formats for random access files and indexed files vary widely.

Databases, managed by database management systems (DBMS), are another kind of data store. Various types of DBMS are available from vendors: hierarchical, network, relational, object-oriented, and logic. DBMS attempt to cache frequently accessed data in memory in order to achieve the best combination of cost and performance from memory and disk storage. Databases are powerful and make applications easier to port to different hardware and operating system platforms, since the vendor ports the DBMS code. One disadvantage is that DBMS have a complex interface. Many database languages integrate awkwardly with programming languages.

The following guidelines characterize the kind of data that belongs in a formal database:

- Data that requires access at fine levels of detail by multiple users
- Data that can be efficiently managed with DBMS commands
- Data that must port across many hardware and operating system platforms
- Data that must be accessible by more than one application program

The following guidelines characterize the kind of data that belongs in a file and not in a relational database:

- Data that is voluminous in quantity but difficult to structure within the confines of DBMS (such as a graphics bit map)
- Data that is voluminous in quantity and of low information density (such as archival files, debugging dumps, or historical records)
- "Raw" data that is summarized in the database
- Volatile data that is kept a short time and then discarded

9.5.1 Advantages of Using a Database

There are many advantages to using a DBMS instead of simple files:

- *Many infrastructure features*, such as crash recovery, sharing between multiple users, sharing between multiple applications, data distribution, integrity, extensibility, and transaction support have already been programmed by the DBMS vendor.
- *Common interface for all applications*. Each application accesses the subset of the information it needs and ignores the rest.
- *A standard access language*. The SQL language is supported by most commercial relational database management systems.

9.5.2 Disadvantages of Using a Database

DBMS also have disadvantages that complicate and sometimes prevent their use on real problems. DBMS provide a general purpose engine for flexible management of data. But at times, DBMS functionality is not powerful enough or the performance overhead from providing general services too high. Some limitations of current DBMS, particularly relational DBMS, are:

- *Performance overhead*. Few relational DBMS can exceed 50 simple transactions per second on a computer such as a VAX 11/785. A simple transaction updates one row of a relational DBMS table. For demanding applications, system designers must work with the DBMS vendor to wring out extra performance or develop a custom solution.
- *Insufficient functionality for advanced applications*. Relational DBMS were developed for business applications that have large quantities of data with simple structure. Relational DBMS are difficult to use for applications that require richer data types or nonstandard operations.
- *Awkward interface with programming languages*. Relational DBMS support set-oriented operations that are expressed through a nonprocedural language. Most programming languages are procedural in nature and can only access relational DBMS tables a row at a time. [Premerlani-90] discusses a solution to integrating object-oriented languages with relational DBMS.

Some of these disadvantages may disappear as efficient object-oriented DBMS are implemented.

9.6 HANDLING GLOBAL RESOURCES

The system designer must identify global resources and determine mechanisms for controlling access to them. Global resources include: physical units, such as processors, tape drives, and communication satellites; space, such as disk space, a workstation screen, and the buttons on a mouse; logical names, such as object IDs, filenames, and class names; and access to shared data, such as databases.

If the resource is a physical object, then it can control itself by establishing a protocol for obtaining access within a concurrent system. If the resource is a logical entity, such as an object ID or a database, then there is danger of conflicting access in a shared environment. Independent tasks could simultaneously use the same object ID, for example. Each global resource must be owned by a "guardian object" that controls access to it. One guardian object can control several resources. All access to the resource must pass through the guardian object. For example, most database managers are free-standing tasks that other tasks can call to obtain data from the database. Allocating each shared global resource to a single object is a recognition that the resource has identity.

A logical resource can also be partitioned logically, such that subsets are assigned to different guardian objects for independent control. For example, one strategy for object ID generation in a parallel distributed environment is to preallocate a range of possible IDs to each processor in a network; each processor allocates the IDs within its preallocated range without the need for global synchronization.

In a time-critical application, the cost of passing all access to a resource through a guardian object is sometimes too high, and clients must access the resource directly. In this case, locks can be placed on subsets of the resource. A *lock* is a logical object associated with some defined subset of a resource that gives the lock holder the right to access the resource directly. A guardian object must still exist to allocate the locks, but after one interaction with the guardian to obtain a lock the user of the resource can access the resource directly. This approach is more dangerous because each resource user must be trusted to behave itself in its access to the resource. The use of direct access to shared resources should be discouraged in an object-oriented design unless absolutely necessary.

9.7 CHOOSING SOFTWARE CONTROL IMPLEMENTATION

During analysis, all interactions are shown as events between objects. Hardware control closely matches the analysis model, but the system designer must choose among several ways to implement control in software. Although there is no logical necessity that all subsystems use the same implementation, usually the designer chooses a single control style for the whole system. There are two kinds of control flows in a software system: external control and internal control.

External control is the flow of externally-visible events among the objects in the system. There are three kinds of control for external events: procedure-driven sequential, event-driven sequential, and concurrent. The control style adopted depends on the resources available

(language, operating system) and on the pattern of interactions in the application. External control is discussed in this section.

Internal control is the flow of control within a process. It exists only in the implementation and therefore is not inherently concurrent or sequential. The designer may choose to decompose a process into several tasks for logical clarity or for performance (if multiple processors are available). Unlike external events, internal transfers of control, such as procedure calls or inter-task calls, are under the direction of the program and can be structured for convenience. Three kinds of control flow are common: procedure calls, quasi-concurrent inter-task calls, and concurrent inter-task calls. Quasi-concurrent inter-task calls, such as coroutines or lightweight processes, are programming conveniences in which multiple address spaces or call stacks exist but in which only a single thread of control can be active at once.

9.7.1 Procedure-driven Systems

In a procedure-driven sequential system, control resides within the program code. Procedures issue requests for external input and then wait for it; when input arrives, control resumes within the procedure that made the call. The location of the program counter and the stack of procedure calls and local variables define the system state.

The major advantage of procedure-driven control is that it is easy to implement with conventional languages; the disadvantage is that it requires the concurrency inherent in objects to be mapped into a sequential flow of control. The designer must convert events into operations between objects. A typical operation corresponds to a pair of events: an output event that performs output and requests input and an input event that delivers the new values. Asynchronous input cannot be easily accommodated with this paradigm because the program must explicitly request input. The procedure-driven paradigm is suitable only if the state model shows a regular alternation of input and output events. Flexible user interfaces and control systems are hard to build using this style.

Note that all major object-oriented languages, such as Smalltalk, C++, and CLOS, are procedural languages. Do not be fooled by the Smalltalk phrase *message passing*. A message *is* a procedure call with a built-in case statement that depends on the class of the target object. A major drawback of conventional object-oriented languages is that they fail to support the concurrency inherent in objects. Some concurrent object-oriented languages have been designed, but they are not yet widely used.

9.7.2 Event-driven Systems

In an event-driven sequential system, control resides within a dispatcher or monitor provided by the language, subsystem, or operating system. Application procedures are attached to events and are called by the dispatcher when the corresponding events occur ("callback"). Procedure calls to the dispatcher send output or enable input but do not wait for it in-line. All procedures return control to the dispatcher, rather than retaining control until input arrives. Events are handled directly by the dispatcher. Program state cannot be preserved using

the program counter and stack because procedures return control to the dispatcher. Procedures must use global variables to maintain state or the dispatcher must maintain local state for them. Event-driven control is more difficult to implement with standard languages than procedure-driven control but is often worth the extra effort.

Event-driven systems permit more flexible patterns of control than procedure-driven systems. Event-driven systems simulate cooperating processes within a single multi-threaded task; an errant procedure can block the entire application, so care must be taken. Event-driven user interface subsystems are particularly useful; some commercial examples include SunView and X-Windows.

Use an event-driven system for external control in preference to a procedure-driven system whenever possible because the mapping from events to program constructs is much simpler and more powerful. Event-driven systems are also more modular and can handle error conditions better than procedure-driven systems.

9.7.3 Concurrent Systems

In a concurrent system, control resides concurrently in several independent objects, each a separate task. Events are implemented directly as one-way messages (*not* Smalltalk "messages") between objects. A task can wait for input, but other tasks continue execution. The operating system usually supplies a queuing mechanism for events so that events are not lost if a task is executing when they arrive. The operating system resolves scheduling conflicts among tasks. Examples of concurrent systems include tasks on an operating system and Ada tasks. If there are multiple CPUs, then different tasks can actually execute concurrently.

Ada supports concurrent tasks within the language. Programs in other languages, such as Fortran, C, or C++, can of course be run as tasks within a standard operating system, but inter-task communication usually requires costly operating system calls that are not portable to other operating systems. None of the current major object-oriented languages directly support tasking. Research is currently underway to develop concurrent object-oriented languages, some of which are in limited use.

9.7.4 Internal Control

During the design process, operations on objects are expanded into lower-level operations on the same or other objects. Internal interactions between objects can be viewed similarly to external interactions among objects, as events passed between objects, and the same implementation mechanisms can be used. There is an important difference: External interactions inherently involve waiting for events because different objects are independent and cannot force other objects to respond; internal operations are generated by objects as part of the implementation algorithm, so their response patterns are predictable. Most internal operations can therefore be thought of as procedure calls, in which the caller issues a request and waits for the response. There are algorithms in which concurrency can be used profitably, if parallel processing is available, but many computations are well represented sequentially and can easily be folded onto a single thread of control.

9.7.5 Other Paradigms

We assume that the reader is primarily interested in procedural programming, but other paradigms are possible, such as rule-based systems, logic programming systems, and other forms of nonprocedural programs. These constitute another control style in which explicit control is replaced by declarative specification with implicit evaluation rules, possibly nondeterministic or highly convoluted. Such languages are currently used in limited areas, such as artificial intelligence and knowledge-based programming, but we expect their use to grow in the future. Because these languages are totally different from procedural languages (including object-oriented languages), the remainder of this book has little to say about their use.

9.8 HANDLING BOUNDARY CONDITIONS

Although most of the design effort in many systems is concerned with the steady-state behavior, the system designer must consider boundary conditions as well: initialization, termination, and failure. The following kinds of issues must be addressed:

Initialization. The system must be brought from a quiescent initial state to a sustainable steady state condition. Things to be initialized include constant data, parameters, global variables, tasks, guardian objects, and possibly the class hierarchy itself. During initialization only a subset of the functionality of the system is usually available. Initializing a system containing concurrent tasks is most difficult because independent objects must not get either too far ahead or too far behind other independent objects during initialization.

Termination. Termination is usually simpler than initialization because many internal objects can simply be abandoned. The task must release any external resources that it had reserved. In a concurrent system, one task must notify other tasks of its termination.

Failure. Failure is the unplanned termination of a system. Failure can arise from user errors, from the exhaustion of system resources, or from an external breakdown. The good system designer plans for orderly failure. Failure can also arise from bugs in the system and is often detected as an "impossible" inconsistency. In a perfect design, such errors would never happen, but the good designer plans for a graceful exit on fatal bugs by leaving the remaining environment as clean as possible and recording or printing as much information about the failure as possible before terminating.

9.9 SETTING TRADE-OFF PRIORITIES

The system designer must set priorities that will be used to guide trade-offs during the rest of design. The designer is often required to choose among desirable but incompatible goals. For example, a system can often be made faster by using extra memory. Design trade-offs must be made regarding not only the software itself but also regarding the process of developing it. Sometimes it is necessary to sacrifice complete functionality to get a piece of soft-

ware into use (or into the marketplace) earlier. Sometimes the problem statement specifies which goals are of highest priority, but often the burden falls on the designer to reconcile the incompatible desires of the client and decide how the trade-offs should be made.

The system designer must determine the relative importance of the various criteria as a guide to making trade-off decisions during design. All the trade-offs are not *made* during system design, but the priorities for making them are established. For example, the first video games ran on processors with limited memory. Conserving memory was the highest priority, followed by fast execution. Designers had to use every programming trick in the book, at the expense of maintainability, portability, and understandability. As another example, there are several mathematical subroutine packages available on a wide range of machines. Well-conditioned numerical behavior is crucial to such packages, as well as portability and understandability. These cannot be sacrificed for fast development. In another case, a user interface is hard to evaluate without actually using it. The designer often uses *rapid prototyping*, which is a "quick and dirty" implementation of part of the system to be evaluated, while ignoring or simulating the rest of the system. Rapid prototyping minimizes initial design time by sacrificing completeness of functionality, efficiency, and robustness. Once the prototype is evaluated, the interface can be reimplemented using different design trade-offs, and the remainder of the system can be implemented.

The entire character of a system is affected by the trade-off decisions made by the designer. The success or failure of the final product may depend on whether its goals are well-chosen. Even worse, if no system-wide priorities are established, then the various parts of the system may optimize opposing goals ("suboptimization"), resulting in a system that wastes resources. Even on small projects, programmers often forget the real goals and become obsessed with "efficiency" when it really is not important.

Setting trade-off priorities is at best vague. You cannot expect numerical accuracy ("speed 53%, memory 31%, portability 15%, cost 1%"). Priorities are rarely absolute; for example, trading memory for speed does not mean that any increase in speed, no matter how small, is worth any increase in memory, no matter how large. We cannot even give a full list of design criteria that might be subject to trade-offs. Instead, the priorities are a statement of design philosophy by the system designer to guide the design process. Judgment and interpretation are required when trade-offs are actually made during the rest of the design.

9.10 COMMON ARCHITECTURAL FRAMEWORKS

There are several prototypical architectural frameworks that are common in existing systems. Each of these is well-suited to a certain kind of system. If you have an application with similar characteristics, you can save effort by using the corresponding architecture, or at least using it as a starting point for your design. The kinds of systems include:

- Batch transformation—a data transformation executed once on an entire input set.
- Continuous transformation—a data transformation performed continuously as inputs change.
- Interactive interface—a system dominated by external interactions.

- Dynamic simulation—a system that simulates evolving real-world objects.
- Real-time system—a system dominated by strict timing constraints.
- Transaction manager—a system concerned with storing and updating data, often including concurrent access from different physical locations.

This is not meant to be a complete list of known systems and architectures but a list of common forms. Some problems require a new kind of architecture, but most can use an existing framework or at least a variation on it. Many problems combine aspects of these architectures.

9.10.1 Batch Transformation

A *batch transformation* is a sequential input-to-output transformation, in which inputs are supplied at the start, and the goal is to compute an answer; there is no ongoing interaction with the outside world. Examples include standard computational problems: compiler, payroll program, VLSI automatic layout, stress analysis of a bridge, and many others.

The state model is trivial or nonexistent for batch transformation problems. The object model may be simple or complex. The most important aspect of a batch transformation is the functional model which specifies how input values are transformed into output values. This is probably the area best addressed by current methodologies emphasizing data flow diagrams and functional decomposition. However, the use of object models for data structures improves on previous methods of representing data for problems that have complex, often polymorphic data. A compiler is an example of a batch transformation with complex data structures. Figure 9.2 shows the data flow diagram for a compiler.

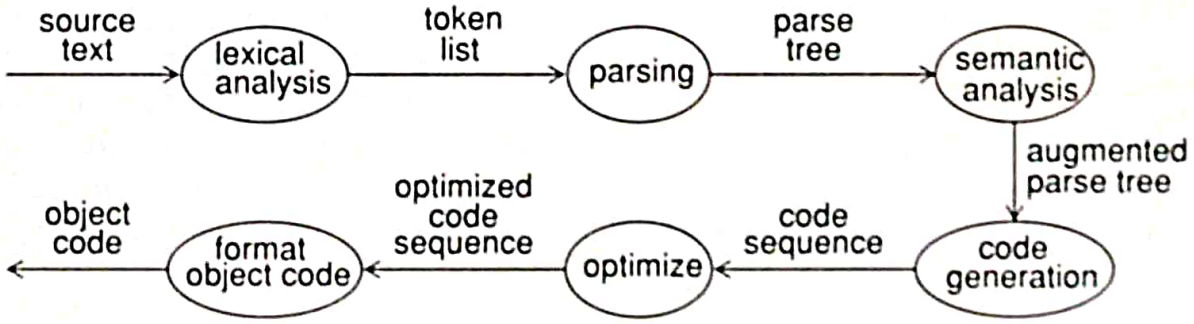


Figure 9.2 Data flow diagram of compiler

The steps in designing a batch transformation are:

- Break the overall transformation into stages, each stage performing one part of the transformation. The system diagram is a data flow diagram. This can usually be taken straight from the functional model, although additional detail may be added during system design.
- Define intermediate object classes for the data flows between each pair of successive stages. Each stage knows only about the objects on either side of it, its own inputs and

outputs. Each set of classes forms a coherent object model in the design, loosely coupled to the object models from neighboring stages.

- Expand each stage in turn until the operations are straightforward to implement.
- Restructure the final pipeline for optimization.

9.10.2 Continuous Transformation

A *continuous transformation* is a system in which the outputs actively depend on changing inputs and must be periodically updated. Unlike a batch transformation, in which the outputs are computed only once, the outputs in an active pipeline must be updated frequently (in theory continuously, although in practice they are computed discretely at a fine time scale). Because of severe time constraints, the entire set of outputs cannot usually be recomputed each time an input changes (otherwise the application would be a batch transformation). Instead, the new output values must be computed incrementally. Typical applications include signal processing, windowing systems, incremental compilers, and process monitoring systems.

The functional model together with the object model defines the values being computed. The dynamic model is less important because most of the structure of the application is due to the steady flow of data and not due to discrete interactions.

Because a complete recomputation is impossible for every input value change, an architecture for a continuous transformation must facilitate incremental computation. The transformation can be implemented as a pipeline of functions. The effect of each incremental change in an input value must be propagated through the pipeline. To make incremental computation possible, intermediate objects may be defined to hold intermediate values. Redundant values may be introduced for performance reasons.

Synchronization of values within the pipeline may be important for high-performance systems, such as signal processing applications. In such cases, operations are performed at well-defined times and the flow path of operations must be carefully balanced so that values arrive at the right place at the right time without bottlenecks.

The steps in designing a pipeline for a continuous transformation are:

- Draw a data flow diagram for the system. The input and output actors correspond to data structures whose values change continuously. Data stores within the pipeline show parameters that affect the input-to-output mapping. Figure 9.3 shows a graphics application in three stages: First geometric figures in user-defined coordinates are mapped into window coordinates; then the figures are clipped to fit the window bounds; finally each figure is offset by the position of its window to yield a position on the screen.

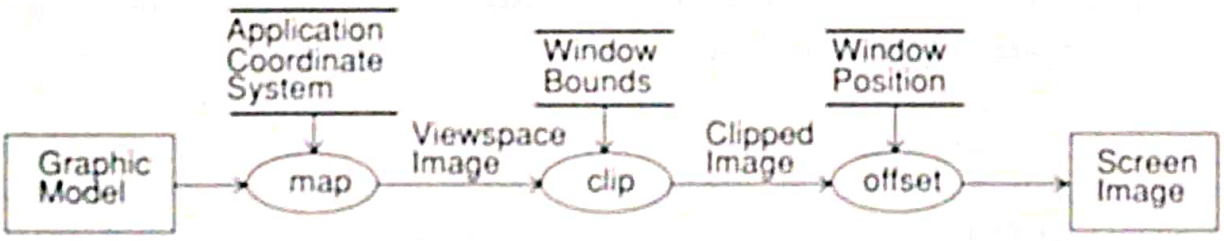


Figure 9.3 Data flow diagram for a graphics application

- Define intermediate objects between each pair of successive stages, as in the batch transformation. For example, each geometric figure from the graphic model has a mapped version of itself at each stage in the computation.
- Differentiate each operation to obtain incremental changes to each stage. That is, propagate the incremental effects of each change to an input object through the pipeline as a series of incremental updates. For example, a change in the position of a geometric figure requires its old image to be erased, its new position computed, and its new image displayed; the images of other figures are unchanged and need not be recomputed.
- Add additional intermediate objects for optimization.

9.10.3 Interactive Interface

An *interactive interface* is a system that is dominated by interactions between the system and external agents, such as humans, devices, or other programs. The external agents are independent of the system, so their inputs cannot be controlled, although the system may solicit responses from them. An interactive interface usually includes only part of an entire application, one that can often be handled independently from the computational part of the application. The major concerns of an interactive interface are the communications protocol between the system and the external agents, the syntax of possible interactions, the presentation of output (the appearance on the screen, for instance), the flow of control within the system, the ease of understanding and user interface, performance, and error handling. Examples of interactive systems include a forms-based query interface, a workstation windowing system, the command language for an operating system, and the control panel for a simulation.

Interactive interfaces are dominated by the dynamic model. Objects in the object model represent interaction elements, such as input and output tokens and presentation formats. The functional model describes which application functions are executed in response to input event sequences, but the internal structure of the functions is usually unimportant to the behavior of the interface. An interactive system is concerned with external appearances, not deep semantic structure.

The steps in designing an interactive interface are:

- Isolate the objects that form the interface from the objects that define the semantics of the application.
- Use predefined objects to interact with external agents, if possible. For example, workstation windowing systems such as X-Windows, NeWS, and MacAPP have extensive collections of predefined windows, menus, buttons, forms, and other kinds of objects ready to be adapted to applications.
- Use the dynamic model as the structure of the program. Interactive interfaces are best implemented using concurrent control (multi-tasking) or event-driven control (interrupts or call-backs). Procedure-driven control (writing output and then waiting for input in-line) is awkward for anything but rigid control sequences.

- Isolate physical events from logical events. Often a logical event corresponds to multiple physical events. For example, a graphical interface can take input from a form, from a pop-up menu, from a function button on the keyboard, by typing a command sequence, or from an indirect command file.
- Fully specify the application functions that are invoked by the interface. Make sure that the information to implement them is present.

9.10.4 Dynamic Simulation

A *dynamic simulation* models or tracks objects in the real world. Examples include molecular motion modeling, spacecraft trajectory computation, economic models, and video games. Traditional methodologies built on data flow diagrams are poor at representing these problems because simulations involve many distinct objects that constantly update themselves, rather than a single large transformation. Simulations are perhaps the simplest system to design using an object-oriented approach. The objects and operations come directly from the application. Control can be implemented in two ways: an explicit controller external to the application objects can simulate a state machine, or objects can exchange messages among themselves, similar to the real-world situation.

Unlike an interactive system, the internal objects in a dynamic simulation do correspond to real-world objects, so the object model is usually important and often complex. Like an interactive system, the dynamic model is an important part of simulation systems. Simulators often have a complex functional model as well.

The steps in designing a dynamic simulation are:

- Identify actors, active real-world objects, from the object model. The actors have attributes that are periodically updated.
- Identify discrete events. Discrete events correspond to discrete interactions with the object, such as turning power on or applying the brakes. Discrete events can be implemented as operations on the object.
- Identify continuous dependencies. Real-world attributes may be dependent on other real-world attributes or vary continuously with time, altitude, velocity, or steering wheel position. These attributes must be updated at periodic intervals using numerical approximation techniques to minimize quantization error.
- Generally a simulation is driven by a timing loop at a fine time scale. Discrete events between objects can often be exchanged as part of the timing loop.

The hardest problem with simulations is usually providing adequate performance. In an ideal world, an arbitrary number of parallel processors would execute the simulation in an exact analogy to the real-world situation. In practice, the system designer must estimate the computational cost of each update cycle and provide adequate resources. Continuous processes must be approximated as discrete steps.

9.10.5 Real-time System

A *real-time system* is an interactive system for which time constraints on actions are particularly tight or in which the slightest timing failure cannot be tolerated. For critical actions, the system must be guaranteed to respond within an absolute interval of time. Typical applications include process control, data acquisition, communications devices, device control, and overload relays. To guarantee response time, the worst case scenario has to be determined and provided for. This can simplify analysis because it is usually easier to determine the worst case behavior than the average case behavior.

Real-time design is complex and involves issues such as interrupt handling, prioritization of tasks, and coordinating multiple CPUs. Unfortunately, real-time systems are frequently designed to operate close to their resource limits so that severe, nonlogical restructuring of the design is often needed to achieve the necessary performance. Such contortions come at the cost of portability and maintainability. Real-time design is a specialized topic that we do not cover in detail in this book.

9.10.6 Transaction Manager

A *transaction manager* is a database system whose main function is to store and access information. The information comes from the application domain. Most transaction managers must deal with multiple users and concurrency. A transaction must be handled as a single atomic entity without interference from other transactions. Examples of transaction managers include airline reservation systems, inventory control systems, and database management systems.

The object model is the most important. The functional model in a transaction management system is less important because operations tend to be predefined and to focus on updating and querying information. The dynamic model shows concurrent access of distributed information. Distribution is an inherent part of the real-world problem and must be modeled as part of the analysis. The dynamic model is also important for estimating transaction throughput.

Frequently you can use an existing database management system. In such cases, the main problem is to construct the object model and choose the granularity of transactions that must be considered atomic by the system.

The steps in designing a transaction management system are:

- Map the object model directly into a database. See Chapter 17 for advice on using a relational database.
- Determine the units of concurrency, that is, the resources that inherently or by specification cannot be shared. Introduce new classes as needed.
- Determine the unit of transaction, that is, the set of resources that must be accessed together during a transaction. Typically a transaction succeeds or fails in its entirety.
- Design concurrency control for transactions. Most database management systems incorporate this. The system may need to retry failed transactions several times before giving up.

9.11 ARCHITECTURE OF THE ATM SYSTEM

The ATM system introduced in Chapter 8 is a hybrid of an interactive interface and a transaction management system. The entry stations are interactive interfaces—their purpose is to interact with a human to gather information needed to formulate a transaction. Specifying the entry stations consists of constructing an object model and a dynamic model; the functional model is trivial. The consortium and banks are primarily a distributed transaction management system. Their purpose is to maintain a database of information and to allow it to be updated over a distributed network under controlled conditions. Specifying the transaction management part of the system consists primarily of constructing an object model.

Figure 9.4 shows the architecture of the ATM system. There are three major subsystems: the ATM stations, the consortium computer, and the bank computers. The topology is a simple star; the consortium computer communicates with all the ATM stations and with all the bank computers. Each connection is a dedicated phone line. The station code and the bank code are used to distinguish the phone lines to the consortium computer.

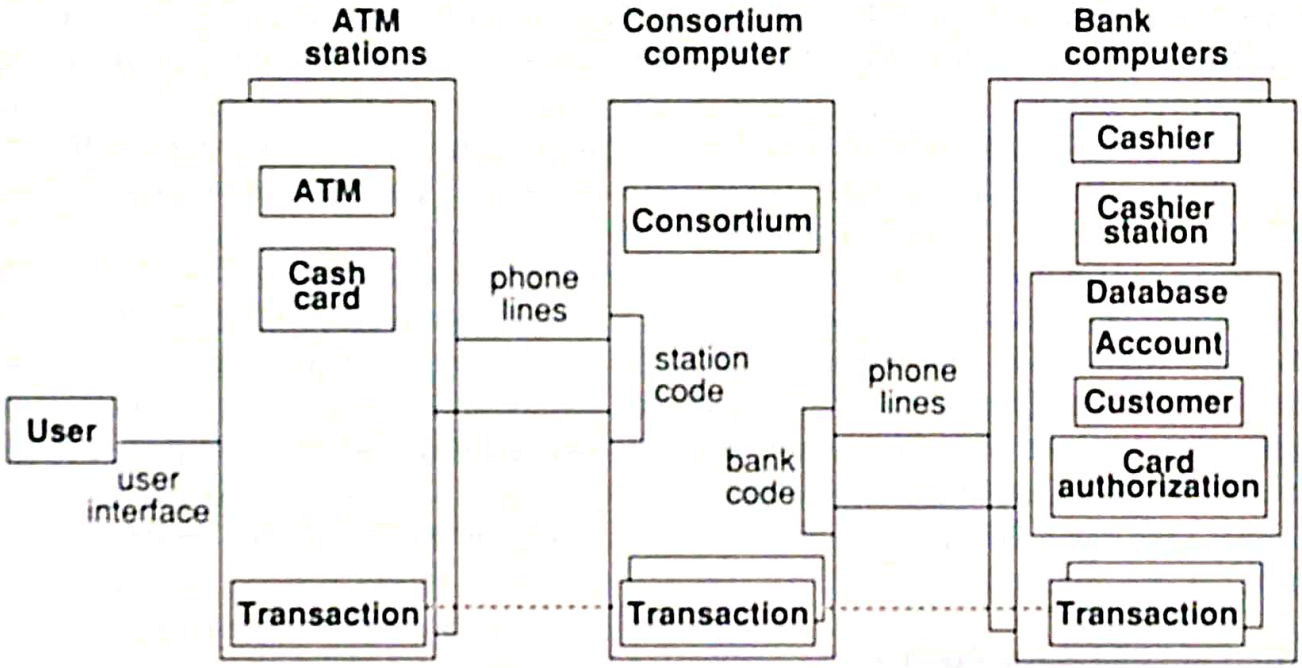


Figure 9.4 Architecture of ATM system

The only permanent data stores are in the bank computers. Because the data must be kept consistent but may be accessed by several concurrent transactions, the data is kept in a database. Each transaction is processed as a single batch operation; an account is locked by a transaction until the transaction is complete.

Concurrency arises because there are many ATM stations, each of which can be active at once. There can be only one transaction per ATM station, but each transaction requires the assistance of the consortium computer and a bank computer. A transaction cuts across physical units; each transaction is shown in the diagram as three connected pieces. During design, each piece will become a separate implementation class. Although there is only one transac-

tion per ATM station, there may be many concurrent transactions per consortium computer or bank computer. This does not pose any special problem because access to any one account is synchronized through the database.

The ATM station is little more than a state diagram. The consortium computer and bank computers will be event driven. Each of them queues input events but processes them one at a time in the order received.

The consortium computer must be large enough to handle the expected maximum number of simultaneous transactions. It may be acceptable to occasionally block a transaction, provided the user receives an appropriate message. The bank computers must also be large enough to handle the expected worst-case load, and they must have enough disk storage to record all transactions.

The system must contain operations to allow ATM stations and bank computers to be added and deleted. Each physical unit must protect itself against the failure or disconnection of the rest of the network. A database will provide protection against loss of data. However, special attention must be paid to failure during a transaction so that neither the user nor the bank lose money. A complicated acknowledgment protocol before committing the transaction may be required. The ATM station should display an appropriate message if the connection is down. Other kinds of failure must be handled as well, such as exhaustion of cash or paper for receipts.

On a financial system such as this, fail-safe transactions are the highest priority. If there is any doubt about the integrity of a transaction, then it must be aborted with an appropriate message to the user.

There is little need for a lower layer of system implementation. The ATM station is just a state machine; its functional model is trivial. The consortium computer simply forwards a message from an ATM station to a bank computer and from a bank computer to an ATM station. It has minimal functionality. The bank computer is the only unit with any non-trivial procedures, but even those are mostly just database updates. The only complexity might come from failure handling.

All in all, the ATM system is a simple architecture, but many applications are similar.

9.12 CHAPTER SUMMARY

After analyzing an application and before beginning the detailed design, the system designer must decide on the basic approach to the solution. The form of the high-level structure of the system, including its breakdown into subsystems, its inherent concurrency, allocation of subsystems to hardware and software, data management, coordination of global resources, software control implementation, boundary conditions, and trade-off priorities, is called the system architecture.

A system can be divided into horizontal layers and vertical partitions. Each layer defines a different abstract world that may differ completely from other layers. Each layer is a client of services of the layer or layers below it and a supplier of services for the layer or layers above it. Systems can also be divided into partitions, each performing a general kind of ser-

Object-oriented design is an iterative process. When you think that the object design is complete at one level of abstraction, add more detail and flesh out the design further at a finer level of detail. You may find that new operations and attributes must be added to classes in the object model, and possibly new classes will be identified. It may even be necessary to revise the relationships between objects (including changes to the inheritance hierarchy). You should not be surprised if you find yourself iterating several times.

10.1.3 Object Modeling Tool Example

Many of the examples in this chapter come from the design of the Object Modeling Tool (OMTool), a program written by one of the authors. OMTool is a graphic editor for constructing object diagrams. With OMTool a person can easily create, load, edit, save, and print object diagrams. A major design goal for OMTool has been to provide simple and natural user interaction.

Before the advent of OMTool, we constructed object diagrams with a general purpose graphic editor. It is tedious to construct object diagrams by drawing lines, boxes, text, and so forth. OMTool permits the user to build object diagrams directly from OMT modeling symbols. For instance, as the user moves a class box, relationship lines stay connected and move with it. OMTool prevents illogical constructs, such as dangling relationship lines. With OMTool, it is easy to quickly sketch out and then clean up an object diagram. We have also developed several backend programs that take acquired OMTool data and generate programming code stubs and relational database schema.

One major architectural decision that we made for OMTool was to store both a logical and graphical model. The graphical model stores the picture that is drawn on the screen: the choice of symbols, position of symbols, length of lines, and so forth. The logical model stores the underlying meaning of the picture, that is, classes, attributes, operations, and their relationships. The graphical model is useful for interacting with the user of OMTool and preparing printouts. The logical model is useful for semantic checking and interacting with the backend programs which need to know what the diagram means but do not care about the precise manner in which it is drawn. The examples in this chapter are taken from both the OMTool graphical and logical models.

10.2 COMBINING THE THREE MODELS

After analysis we have the object, dynamic, and functional models, but the object model is the main framework around which the design is constructed. The object model from analysis may not show operations. The designer must convert the actions and activities of the dynamic model and the processes of the functional model into operations attached to classes in the object model. In making this conversion, we begin the process of mapping the logical structure of the analysis model into a physical organization of a program.

Each state diagram describes the life history of an object. A transition is a change of state of the object and maps into an operation on the object. We can associate an operation with each event received by an object. In the state diagram, the action performed by a tran-

sition depends on both the event and the state of the object. Therefore the algorithm implementing an operation depends on the state of the object. If the same event can be received by more than one state of an object, then the code implementing the algorithm must contain a case statement dependent on the state. (If the language permits an object to change its class at run time, then the states of the object can be implemented as subclasses of the original class, and the method resolution mechanism eliminates the need for a case statement. State can be considered an example of generalization by restriction, as explained in Section 4.3, but most object-oriented languages do not support dynamic changing of an object's class.) However, in many cases, an event can only be received in a single state, or all transitions on the event result in the same action, so no case statement is necessary.

An event sent by an object may represent an operation on another object. Events often occur in pairs, with the first event triggering an action and the second event returning the result or indicating the completion of that action. In this case, the event pair can be mapped into an operation performing the action and returning control provided that the events are on a single thread of control passing from object to object.

An action or activity initiated by a transition in a state diagram may expand into an entire data flow diagram in the functional model. The network of processes within the data flow diagram represents the body of an operation. The flows in the diagram are intermediate values in the operation. The designer must convert the graph structure of the diagram into a linear sequence of steps in an algorithm. The processes in the data flow diagram constitute suboperations. Some of them, but not necessarily all, may be operations on the original target object or on other objects. Determine the target object of a suboperation as follows:

- If a process extracts a value from an input flow, then the input flow is the target.
- If a process has an input flow and an output flow of the same type, and the output value is substantially an updated version of the input flow, then the input/output flow is the target.
- If a process constructs an output value from several input flows, then the operation is a class operation (constructor) on the output class.
- If a process has an input from or an output to a data store or an actor, then the data store or actor is a target of the process. (In some cases, such a process must be broken into two operations, one for the actor or data store and one for a flow value.)

The original target class is a client of any classes that supply internal operations to one of its operations. The client-supplier relationship defines the structure of the operation calling graph (sometimes called the program structure chart).

10.3 DESIGNING ALGORITHMS

Each operation specified in the functional model must be formulated as an *algorithm*. The analysis specification tells *what* the operation does from the view point of its clients, but the algorithm shows *how* it is done. An algorithm may be subdivided into calls on simpler op-

erations, and so on recursively, until the lowest-level operations are simple enough to implement directly without further refinement.

The algorithm designer must:

- Choose algorithms that minimize the cost of implementing operations
- Select data structures appropriate to the algorithms
- Define new internal classes and operations as necessary
- Assign responsibility for operations to appropriate classes

10.3.1 Choosing Algorithms

Many operations are simple enough that the specification in the functional model already constitutes a satisfactory algorithm because the description of what is done also shows how it is done. Many operations simply traverse paths in the object-link network to retrieve or change attributes or links. For example, Figure 10.1 shows a *Class box* object containing an *Operation list*, which in turn contains a set of *Operation entry* objects. There is no need to write an algorithm to find the class box containing a given operation entry because the value is found by a simple traversal of unique links. Nontrivial algorithms are primarily needed for two reasons: to implement functions for which no procedural specification is given and to optimize functions for which a simple but inefficient algorithm serves as definition.

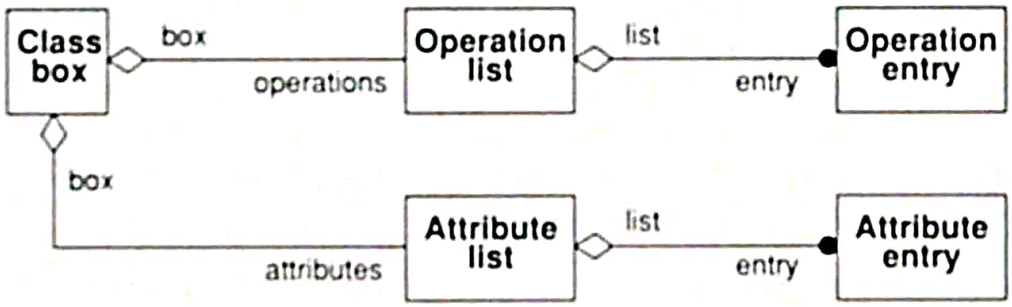


Figure 10.1 Fragment of OMTool model

Some functions are specified as declarative constraints, without any procedural definition. For example, "the circle passing through three noncolinear points" is a nonprocedural specification of a circle. In such cases, you must use your knowledge of the situation (and appropriate reference books) to invent an algorithm. The essence of most geometry problems, such as the example given, is the discovery of appropriate algorithms and the proof that they are correct.

Most functions have simple mathematical or procedural definitions. Often the simple definition is also the best algorithm for computing the function or else is so close to any other algorithm that any loss in efficiency is worth the gain in clarity. For example, a class box from Figure 10.1 is drawn by first drawing its outline and then iteratively drawing its parts, the operation list and the attribute list.

In other cases, the simple definition of an operation would be hopelessly inefficient and must be implemented with a more efficient algorithm. For example, searching for a value in

a set of size n by scanning the set requires an average of $n/2$ operations, whereas a binary search takes $log\ n$ operations and a hash search takes less than 2 operations on average, regardless of the size of the set.

The level of abstraction of the algorithms should not go below the level of granularity of the objects in your object model. For example, in sketching the recursive draw algorithm for the class box in Figure 10.1, it is inappropriate to worry about the low-level graphics calls that will draw the box icon. It is not necessary to write algorithms for trivial operations that are internal to one object, such as setting or accessing the value of an attribute.

Considerations in choosing among alternative algorithms include:

- *Computational complexity.* How does processor time increase as a function of the size of the data structures? Don't worry about small factors in efficiency—avoid "bit pushing." For example, an extra level of indirection is insignificant if it improves clarity. It is essential, however, to think about the complexity of the algorithm, that is, how the execution time (or memory) grows with the number of input values—constant time, linear, quadratic, or exponential—as well as the cost of processing each input value. For example, the infamous "bubble sort" algorithm requires time proportional to n^2, where n is the size of the list, while most alternative sort algorithms require time proportional to $n\ log\ n$.
- *Ease of implementation and understandability.* It is worth giving up some performance on noncritical operations if they can be implemented quickly with a simple algorithm. For example, a pick operation on an OMTool diagram is implemented as a recursive search of top-level diagram elements, such as class boxes and associations, working down toward primitive elements, such as individual attribute entries. This is not the most efficient algorithm theoretically, but it is simple to implement and extensible. Here, speed is not a problem because the operation is only performed when the user pushes a button, and the number of elements on the page is limited.
- *Flexibility.* Most programs will be extended sooner or later. A highly optimized algorithm often sacrifices readability and ease of change. One possibility is to provide two implementations of critical operations—a simple but inefficient algorithm that can be implemented quickly and used to validate the system, and a complicated but efficient algorithm, whose correct implementation can be checked against the simple one. For example, a more complicated algorithm for picking objects in a diagram could be implemented by spatially sorting all the objects in a large, flat data structure. This algorithm would be faster for large diagrams, but it imposes constraints on the form of the objects and requires that the algorithm know details for all objects. In any case, the original simple algorithm could be used as a correctness check.
- *Fine tuning the object model.* If the object model were structured differently, would there be other alternatives? For example, Figure 10.2 shows two designs of the mapping between diagram elements and windows in OMTool. In the original upper design, each diagram element contains a list of windows in which it is visible. This is inefficient because operations on a set of elements must be computed separately for each window. In the lower design, each element belongs to one sheet, which may appear in any number

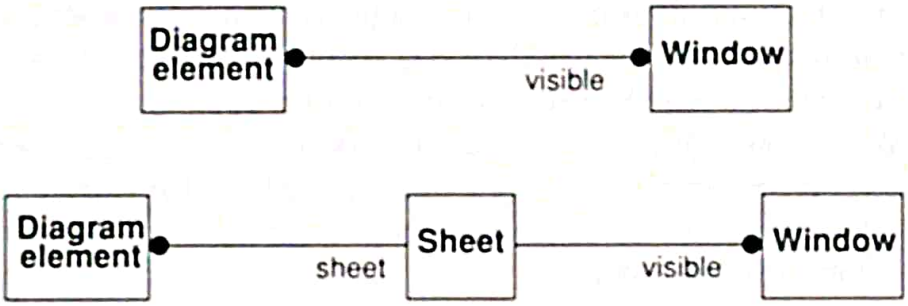


Figure 10.2 Alternative structure

of windows. The image on the sheet can be computed once, and then copied to each window as a bitmap operation. The reduction in repeated operations is worth the extra level of indirection.

10.3.2 Choosing Data Structures

Choosing algorithms involves choosing the data structures they work on. During analysis, we concentrated only on the logical structure of the information in the system, but during object design we must choose the form of the data structures that will permit efficient algorithms. The data structures do not add information to the analysis model, but they organize it in a form convenient for the algorithms that use it. Many implementation data structures are instances of *container classes*. Such data structures include arrays, lists, queues, stacks, sets, bags, dictionaries, associations, trees, and many variations on these, such as priority queues and binary trees. Most object-oriented languages provide an assortment of generic data structures as part of their predefined class libraries.

For example, the diagram elements in a picture must be drawn on the screen in some specific order because the ones drawn last may overlap the ones drawn first. To permit consistent ordering, they are organized into an ordered list.

10.3.3 Defining Internal Classes and Operations

During the expansion of algorithms, new classes of objects may be needed to hold intermediate results. New, low-level operations may be invented during the decomposition of high-level operations.

A complex operation can be defined in terms of lower-level operations on simpler objects. These low-level operations must be defined during object design because most of them are not visible externally. Some of the required lower-level operations may be found among the "shopping-list" operations that were identified during analysis as being potentially useful. But there will usually be a need to add new internal operations as we expand high-level functions. For example, the OMTool erase operation on a diagram element is conceptually simple, but its implementation on a pixel-based screen is more complicated. To erase an object, it must be drawn in the background color, then objects uncovered by the erasure or damaged by the draw must be repaired by being redrawn. The repair operation is purely an internal operation needed because we are working on a pixel-based screen.

When you reach this point during the design phase, you may have to add new classes that were not mentioned directly in the client's description of the problem. These low-level classes are the implementation elements out of which the application classes are built. For example, an OMTool class box image is made of rectangles, lines, and text strings in various fonts. In OMTool, the low-level graphics elements come from the graphics toolkit module, which supplies its services to the rest of the system. Typically low-level implementation classes are placed in a distinct module.

10.3.4 Assigning Responsibility for Operations

Many operations have obvious target objects, but some operations can be performed at several places in an algorithm, by one of several objects, as long as they eventually get done. Such operations are often part of a complex high-level operation with many consequences. Assigning responsibility for such operations can be frustrating, and they are easy to overlook in laying out object classes because they are not an inherent part of any one class.

For example, OMTool has a *drag* operation applicable to diagram elements. Dragging a box moves the box and all the lines attached to it, provided nothing pushes up against an obstacle. The drag operation propagates from object to object along connections between objects. Some drags can fail because of obstacles, and backtracking may be required. Eventually the picture must be redrawn on the screen. When should each object image be redrawn on the screen? After it is dragged by another object? After it drags other objects? After all objects have been dragged? Is each object responsible for redrawing itself, or is the entire picture responsible for redrawing itself? Such questions are hard to answer because the breakdown on a complex externally meaningful operation into internal operations is arbitrary.

When a class is meaningful in the real world, then the operations on it are usually clear. During implementation, however, internal classes are introduced that do not correspond to real-world objects but merely some aspect of them. Since the internal classes are invented for implementation, they are somewhat arbitrary, and their boundaries are more a matter of convenience than of logical necessity.

How do you decide what class owns an operation? When only one object is involved in the operation, the decision is easy: Ask (or tell) that object to perform the operation. The decision is more difficult when more than one object is involved in an operation. You must decide which object plays the lead role in the operation. Ask yourself the following questions:

- Is one object acted on while the other object performs the action? In general, it is best to associate the operation with the *target* of the operation, rather than the *initiator*.
- Is one object modified by the operation, while other objects are only queried for the information they contain? The object that is changed is the target of the operation.
- Looking at the classes and associations that are involved in the operation, which class is the most centrally-located in this subnetwork of the object-model? If the classes and associations form a star about a single central class, it is the target of the operation.

- If the objects were not software, but were the real-world objects being represented internally, what real object would you push, move, activate, or otherwise manipulate to initiate the operation?

Assigning an operation to a class within a generalization hierarchy can sometimes be difficult because the definitions of the subclasses within the hierarchy are often fluid and can be adjusted during design as convenient. It is common to move an operation up and down in the hierarchy during design, as its scope is adjusted.

10.4 DESIGN OPTIMIZATION

The basic design model uses the analysis model as the framework for implementation. The analysis model captures the logical information about the system, while the design model must add details to support efficient information access. The inefficient but semantically-correct analysis model can be optimized to make the implementation more efficient, but an optimized system is more obscure and less likely to be reusable in another context. The designer must strike an appropriate balance between efficiency and clarity.

During design optimization, the designer must:

- Add redundant associations to minimize access cost and maximize convenience
- Rearrange the computation for greater efficiency
- Save derived attributes to avoid recomputation of complicated expressions

10.4.1 Adding Redundant Associations for Efficient Access

During analysis, it is undesirable to have redundancy in the association network because redundant associations do not add any information. During design, however, we evaluate the structure of the object model for an implementation. Is there a specific arrangement of the network that would optimize critical aspects of the completed system? Should the network be restructured by adding new associations? Can existing associations be omitted? The associations that were useful during analysis may not form the most efficient network when the access patterns and relative frequencies of different kinds of access are considered.

To demonstrate the analysis of access paths, consider the design of a company's employee skills database. A portion of the object model from the analysis phase is shown in Figure 10.3. The operation *Company::find-skill* returns a set of persons in the company with a given skill. For example, we might ask for all employees who speak Japanese.

Figure 10.3 Chain of associations

For this example, suppose that the company has 1000 employees each of whom has 10 skills on average. A simple nested loop would traverse *Employs* 1000 times and *Has-skill* 10,000 times. If only 5 employees actually speak Japanese, then the test-to-hit ratio is 2000.

Several improvements are possible. First, *Has-skill* need not be implemented as an unordered list but instead as a hashed set. Hashing can be performed in constant time, so the cost of testing whether a person speaks Japanese is constant, provided *Speaks Japanese* is represented by a unique skill object. This rearrangement reduces the number of tests from 10,000 to 1,000, one per employee.

In cases where the number of hits from a query is low because only a fraction of objects satisfy the test, we can build an *index* to improve access to objects that must be frequently retrieved. For example, we can add a qualified association *Speaks language* from *Company* to *Employee*, where the qualifier is the language spoken (Figure 10.4). This permits us to immediately access all employees who speak a particular language with no wasted accesses. There is a cost of the index: It requires additional memory, and it must be updated whenever the base associations are updated. The designer must decide when it is worthwhile to build indexes. Note that if most queries return all or most of the objects in the search path, then an index really does not save much because the test-to-hit ratio is near 1.

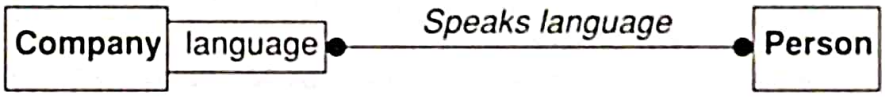


Figure 10.4 Index for personal skills database

Speaks language is a derived association, defined in terms of underlying base associations. The derived association does not add any information to the network but permits the model information to be accessed in a more efficient manner.

Analyze the use of paths in the association network as follows:

- Examine each operation and see what associations it must traverse to obtain its information. Note which associations are traversed in both directions (usually not by a single operation) and which are traversed in a single direction only; the latter can be implemented efficiently with one-way pointers.

For each operation, note the following items:

- How often is the operation called? How costly is it to perform?
- What is the "fan-out" along a path through the network? Estimate the average count of each "many" association encountered along the path. Multiply the individual fan-outs to obtain the fan-out of the entire path, which represents the number of accesses on the last class in the path. Note that "one" links do not increase the fan-out, although they increase the cost of each operation slightly; don't worry about such small effects.
- What is the fraction of "hits" on the final class, that is, objects that meet selection criteria (if any) and are operated on? If most objects are rejected during the traversal for some reason, then a simple nested loop may be inefficient at finding target objects.

Provide indexes for frequent, costly operations with a low hit ratio because such operations are inefficient to implement using nested loops to traverse a path in the network.

10.4.2 Rearranging Execution Order for Efficiency

After adjusting the structure of the object model to optimize frequent traversals, the next thing to optimize is the algorithm itself. Actually, data structures and algorithms are directly related to each other, but we find that usually the data structure should be considered first.

One key to algorithm optimization is to eliminate dead paths as early as possible. For example, suppose we want to find all employees who speak both Japanese and French. Suppose 5 employees speak Japanese and 100 speak French; it is better to test and find the Japanese speakers first, then test if they speak French. In general, it pays to narrow the search as soon as possible. Sometimes the execution order of a loop must be inverted from the original specification in the functional model.

10.4.3 Saving Derived Attributes to Avoid Recomputation

Data that is redundant because it can be derived from other data can be "cached" or stored in its computed form to avoid the overhead of recomputing it. New objects or classes may be defined to retain this information. The class that contains the cached data must be updated if any of the objects that it depends on are changed.

Figure 10.5 shows a use of a derived object and derived attribute in OMTool. Each class box contains an ordered list of attributes and operations, each represented as a text string (left of diagram). Given the location of the class box itself, the location of each attribute can be computed by adding up the size of all the elements in front of it. Since the location of each element is needed frequently, the location of each attribute string is computed and stored. The region containing the entire attribute list is also computed and saved so that input points need not be tested against attribute text elements in other boxes (right of diagram). If a new attribute string is added to the list, then the locations of the ones after it in the list are simply offset by the size of the new element.

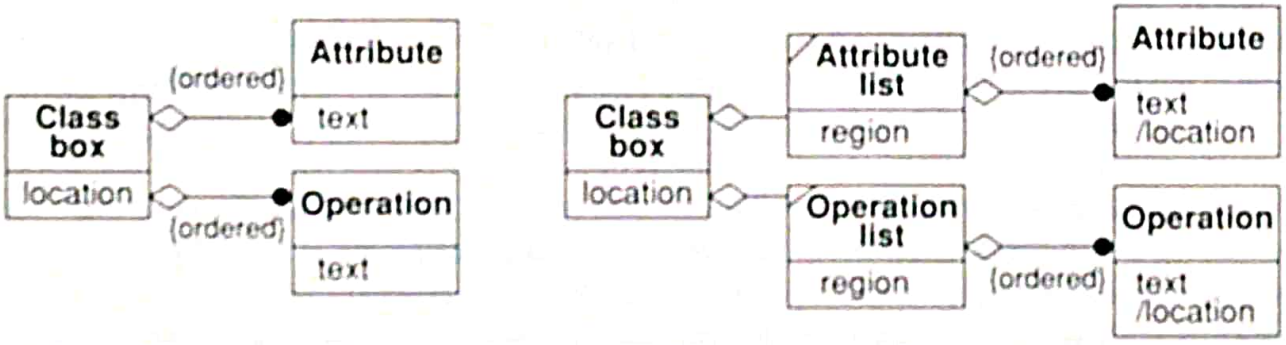


Figure 10.5 Derived attribute to avoid recomputation

The use of an association as a cache is shown in Figure 10.6. A sheet contains a priority list of partially overlapping elements. If an element is moved or deleted, the elements under it must be redrawn. Overlapping elements can be found by scanning all elements in front of the deleted element in the priority list for the sheet and comparing them to the deleted element. If the number of elements is large, this algorithm grows linearly in the number of elements. The *Overlaps* association stores those elements that overlap an object and precede it in the list. This association must be updated when a new element is added, but testing for overlap using the association is more efficient.

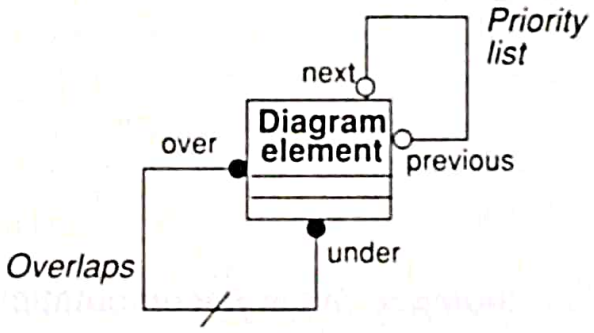


Figure 10.6 Association as a cache

Derived attributes must be updated when base values change. There are three ways to recognize when an update is needed: by explicit code, by periodic recomputation, or by using active values.

Explicit update. Each derived attribute is defined in terms of one or more fundamental base objects. The designer determines which derived attributes are affected by each change to a fundamental attribute and inserts code into the update operation on the base object to explicitly update the derived attributes that depend on it.

Periodic recomputation. Base values are often updated in bunches. Sometimes it is possible to simply recompute all the derived attributes periodically without recomputing derived attributes after each base value is changed. Recomputation of all derived attributes can be more efficient than incremental update because some derived attributes may depend on several base attributes and might be updated more than once by an incremental approach. Also periodic recomputation is simpler than explicit update and less prone to bugs. On the other hand, if the data set changes incrementally a few objects at a time, periodic recomputation is not practical because too many derived attributes must be recomputed when only a few are affected.

Active values. An *active value* is a value that has dependent values. Each dependent value *registers* itself with the active value, which contains a set of dependent values and update operations. An operation to update the base value triggers updates of all the dependent values, but the calling code need not explicitly invoke the updates. Separating the calling code from the dependent object updates provides the same kind of modularity advantage as separating the call of an operation from the methods that it might invoke. Some programming languages implement active values.

10.5 IMPLEMENTATION OF CONTROL

The designer must refine the strategy for implementing the state-event models present in the dynamic model. As part of system design, you will have chosen a basic strategy for realizing the dynamic model (Section 9.7). Now during object design you must flesh out this strategy.

There are three basic approaches to implementing the dynamic model:

- Using the location within the program to hold state (procedure-driven system)
- Direct implementation of a state machine mechanism (event-driven system)
- Using concurrent tasks

10.5.1 State as Location within a Program

This is the traditional approach to representing control within a program. The location of control within a program implicitly defines the program state. Any finite state machine can be implemented as a program (easily using gotos, somewhat harder using nested program structures). Each state transition corresponds to an input statement. After input is read, the program branches depending on the input event received. Each input statement needs to handle any input value that could be received at that point. In highly nested procedural code, low-level procedures must accept inputs that they may know nothing about and pass them up through many levels of procedure calls until some procedure is prepared to handle them. The lack of modularity is the biggest drawback of this approach.

One technique of converting a state diagram to code is as follows:

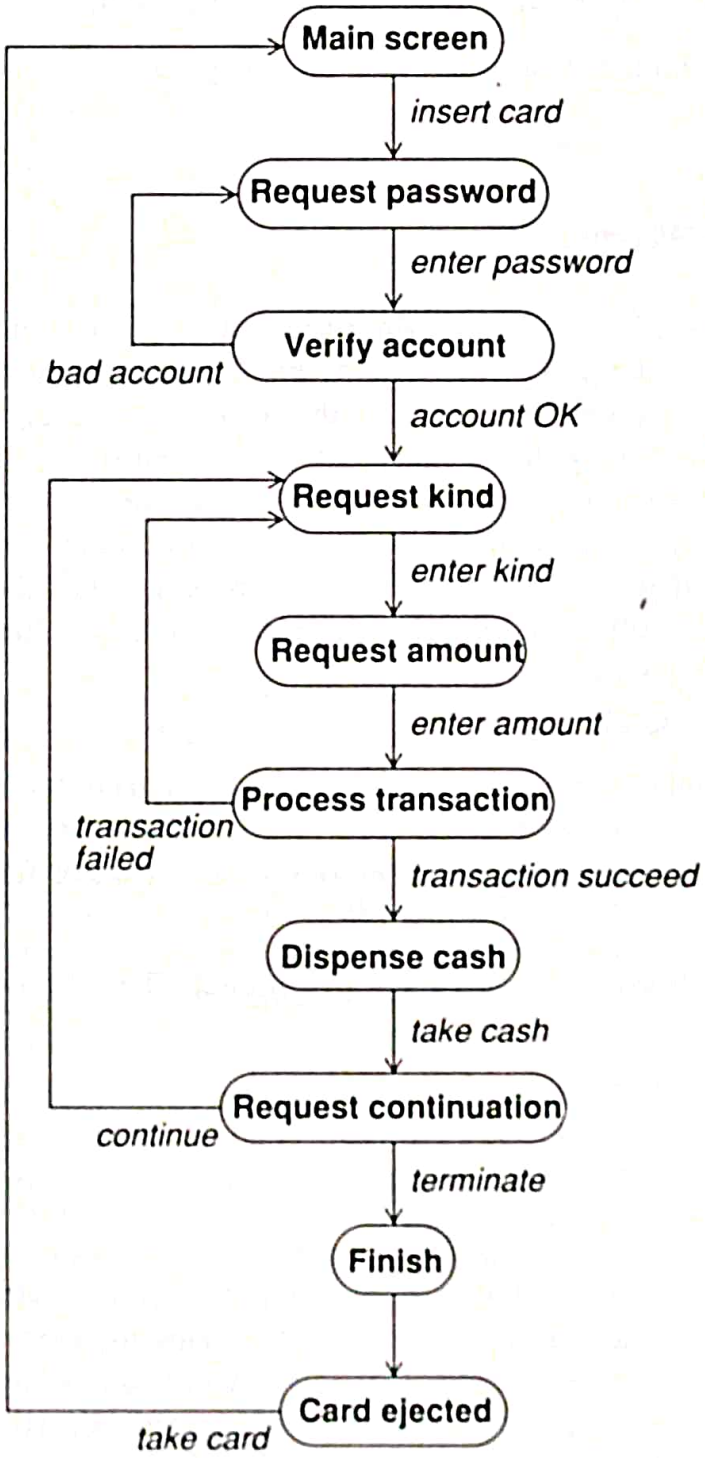
1. Identify the main control path. Beginning with the initial state, identify a path through the diagram that corresponds to the normally expected sequence of events. Write the names of states along this path as a linear sequence. This becomes a sequence of statements in the program.

2. Identify alternate paths that branch off the main path and rejoin it later. These become conditional statements in the program.

3. Identify backward paths that branch off the main loop and rejoin it earlier. These become loops in the program. If there are multiple backward paths that do not cross, they become nested loops in the program. Backward paths that cross do not nest and can be implemented with gotos if all else fails, but these are rare.

4. The states and transitions that remain correspond to exception conditions. They can be handled by several techniques including error subroutines, exception handling supported by the language, or setting and testing of status flags. Exception handling is a legitimate use for gotos in a programming language because their use frequently simplifies breaking out of a nested structure, but do not use them unless necessary.

Let's see how the above approach could be applied to the state model for the ATM class introduced in Chapter 8. Figure 10.7 shows the state model and the pseudocode derived from it. First, we identify the main path of control, which corresponds to the reading of a card, querying the user for transaction information, processing the transaction, printing a receipt, and ejecting the card. Alternative flows of control occur if the customer wants to process more than one transaction or if the password is bad and the customer is asked to try again.



Pseudocode

```
do forever
    display main screen
    read card
    repeat
        ask for password
        read password
        verify account
    until account verification is OK
    repeat
        repeat
            ask for kind of transaction
            read kind
            ask for amount
            read amount
            start transaction
            wait for it to complete
        until transaction is OK
        dispense cash
        wait for customer to take it
        ask whether to continue
    until user asks to terminate
    eject card
    wait for customer to take card
```

Figure 10.7 ATM control

Putting these all together, the right portion of Figure 10.7 shows the pseudocode for the ATM control loop. The *cancel* events could be added to the flow of control and implemented as *goto* exception handling code.

Input events within a single-threaded program are coded as blocking I/O reads, that is, I/O statements that wait for input (usually immediately following a write). In a multitasking language, such as Ada, input events can also be coded as wait statements for an inter-task call. The operating system is responsible for catching interrupts and queuing them up for ordinary programs.

10.5.2 State Machine Engine

The most direct approach to implementing control is to have some way of explicitly representing and executing state machines. For example, a general "state machine engine" class could provide the capability to execute a state machine represented by a table of transitions and actions provided by the application. Each object instance would contain its own independent state variables but would call on the state engine to determine the next state and action. (The state machines are objects but not application objects. They are part of the language substrate to support the semantics of application objects.)

This approach allows you to quickly progress from the analysis model to a skeleton prototype of the system by defining classes from the object model, state machines from the dynamic model, and creating "stubs" of the action routines. A stub is the minimal definition of a function or subroutine without any internal code (code to return a precalculated or contrived value may be included). Thus if each stub prints out its name, this technique allows you to execute the skeleton application to verify that the basic flow of control is correct.

A parser, such as Unix *yacc* or *lex*, produces an explicit state machine to implement a user interface. Some application packages, especially in the user interface area, permit state machines to be supplied as tables to be interpreted by the package.

Creation of a state machine mechanism is not particularly difficult using an object-oriented language and should be considered as a practical alternative if you do not have a state machine package already available.

10.5.3 Control as Concurrent Tasks

An object can be implemented as a task in the programming language or operating system. This is the most general approach, as it preserves the inherent concurrency of real objects. Events are implemented as inter-task calls using the facilities of the language or operating system. As in the previous implementation, the task uses its location within the program to keep track of its state.

Some languages, such as Concurrent Pascal or Concurrent C++, support concurrency, but acceptance of such languages in production environments is still limited. Ada supports concurrency, provided an object is equated with an Ada task, although the run-time cost is high. The major object-oriented languages do not yet support concurrency.

10.6 ADJUSTMENT OF INHERITANCE

As object design progresses, the definitions of classes and operations can often be adjusted to increase the amount of inheritance. The designer should:

- Rearrange and adjust classes and operations to increase inheritance
- Abstract common behavior out of groups of classes
- Use delegation to share behavior when inheritance is semantically invalid

10.6.1 Rearranging Classes and Operations

Sometimes the same operation is defined across several classes and can easily be inherited from a common ancestor, but more often operations in different classes are similar but not identical. By slightly modifying the definitions of the operations or the classes, the operations can often be made to match so that they can be covered by a single inherited operation.

Before inheritance can be used, each operation must have the same interface and the same semantics. All operations must have the same signature, that is, the same number and types of arguments and results. If the signatures match, then the operations must be examined to see if they have the same semantics. The following kinds of adjustments can be used to increase the chance of inheritance:

- Some operations may have fewer arguments than others. The missing arguments can be added but ignored. For example, a draw operation on a monochromatic display does not need a color parameter, but the parameter can be accepted and ignored for consistency with color displays.
- Some operations may have fewer arguments because they are special cases of more general arguments. Implement the special operations by calling the general operation with appropriate parameter values. For example, appending an element to a list is a special case of inserting an element into list; the insert point simply follows the last element.
- Similar attributes in different classes may have different names. Give the attributes the same name and move them to a common ancestor class. Then operations that access the attributes will match better. Also watch for similar operations with different names. A consistent naming strategy is important to avoid hiding similarities.
- An operation may be defined on several different classes in a group but be undefined on the other classes. Define it on the common ancestor class and declare it as a no-op on the classes that don't care about it. For example, in OMTool the *begin-edit* operation places some figures, such as class boxes, in a special draw mode to permit rapid resizing while the text in them is being edited. Other figures have no special draw mode, so the *begin-edit* operation on these classes has no effect.

10.6.2 Abstracting Out Common Behavior

Opportunities to use inheritance are not always recognized during the analysis phase of development, so it is worthwhile to reexamine the object model looking for commonality between

classes. In addition, new classes and operations are often added during design. If a set of operations and/or attributes seems to be repeated in two classes, it is possible that the two classes are really specialized variations of the same thing when viewed at a higher level of abstraction.

When common behavior has been recognized, a common superclass can be created that implements the shared features, leaving only the specialized features in the subclasses. This transformation of the object model is called *abstracting out* a common superclass or common behavior. Usually the resulting superclass is abstract, meaning that there are no direct instances of it, but the behavior it defines belongs to all instances of its subclasses. For example, a *draw* operation of a geometric figure on a display screen requires setup and rendering of the geometry. The rendering varies among different figures, such as circles, lines, and splines, but the setup, such as setting the color, line thickness, and other parameters, can be inherited by all figure classes from abstract class *Figure*.

Sometimes it is worthwhile to abstract out a superclass even when there is only one subclass in your project that inherits from it. Although this does not result in any sharing of behavior in the immediate project, the abstract superclass thus created may be reusable in future projects. It may even be a worthwhile addition to your class library. When a project is completed, the potentially reusable classes should be collected, documented, and generalized so that they may be used in future projects.

Abstract superclasses have benefits other than sharing and reuse. The splitting of a class into two classes that separate the specific aspects from the more general aspects is a form of *modularity*. Each class is a separately maintained component with a well documented interface.

The creation of abstract superclasses also improves the *extensibility* of a software product. Imagine that you are developing a temperature-sensing module for a larger computerized control system. There is a specific type of sensor (Model J55) that you must use, with a particular way of reading the temperature, and a formula for converting the raw numeric reading into degrees Celsius. You could implement all this behavior in a single class, with one instance for every sensor in the system. But realizing that the J55 sensor is not the only type available, you create an abstract *Sensor* superclass that defines the general behavior common to all sensors. A particular subclass called *Sensor-J55* implements reading and conversion that is particular to this model.

Now, when your control system converts to use a new model of sensor, all you have to do is implement a new subclass for that model with only the specialized behavior that is different. The common behavior has already been implemented. Perhaps best of all, you will not have to change a single line of code in the large control system that uses these sensors because the interface is the same, as defined by the *sensor* superclass.

There is a subtle but important way that abstract superclasses improve the *configuration management* aspect of software maintenance and distribution. Suppose that your control system software must be distributed to many plants throughout the country, each of which has a different system configuration involving (among other things) a different mix of temperature sensors. Some plants still use the old J55 model, while others have converted to the newer K99 model, and some plants may have a mixture of both types. Generating customized versions of your software to match each different configuration could be tedious.

Instead, you distribute one version of software that contains a subclass for each known model of sensor. When the software starts up, it reads a configuration file provided by the

customer that tells it which model of sensor is used in which location and creates an instance of the particular subclass that handles that type of sensor. All the rest of the code treats the sensors as if they were all the same as defined by the *Sensor* superclass. It is even possible to change from one type of sensor to another *on-the-fly* (while the system is running) if the software is told to create a new object to manage the new type of sensor.

10.6.3 Use Delegation to Share Implementation

Inheritance is a mechanism for implementing generalization, in which the behavior of a superclass is shared by all its subclasses. Sharing of behavior is justifiable only when a true generalization relationship occurs, that is, only when it can be said that the subclass *is* a form of the superclass. Operations of the subclass that override the corresponding operation of the superclass have a responsibility to provide the same services provided by the superclass, and possibly more. When class B *inherits the specification* of class A, we can assume that every instance of class B *is* an instance of class A because it behaves the same.

Sometimes programmers use inheritance as an implementation technique with no intention of guaranteeing the same behavior. It often happens that an existing class already implements some of the behavior that we want to provide in a newly-defined class, although in other respects the two classes are different. The designer is then tempted to inherit from the existing class to achieve part of the implementation of the new class. This can lead to problems if other operations that are inherited provide unwanted behavior. We discourage this *inheritance of implementation* because it can lead to incorrect behavior.

As an example of implementation inheritance, suppose that you are about to implement a *Stack* class, and you already have a *List* class available. You may be tempted to make *Stack* inherit from *List*. Pushing an element onto the stack can be achieved by adding an element to the end of the list and popping an element from a stack corresponds to removing an element from the end of the list. But we are also inheriting unwanted list operations that add or remove elements from arbitrary positions in the list. If these are ever used (by mistake or as a "short-cut") then the *Stack* class will not behave as expected.

Often when you are tempted to use inheritance as an implementation technique, you could achieve the same goal in a safer way by making one class an attribute or associate of the other class. In this way, one object can selectively invoke the desired functions of another class, using *delegation* rather than inheritance. Delegation consists of catching an operation on one object and sending it to another object that is part of or related to the first object. Only meaningful operations are delegated to the second object, so there is no danger of inheriting meaningless operations by accident.

A safer implementation of *Stack* would delegate to the *List* class as shown in Figure 10.8. Every instance of *Stack* contains a private instance of *List*. (The actual implementation of this aggregation is optimized as discussed in Section 10.7, possibly using an embedded object or a pointer attribute.) The *Stack::push* operation delegates to the list by calling its *last* and *add* operations to add an element at the end of the list, and the *pop* operation has a similar implementation using the *last* and *remove* operations. The ability to corrupt the stack by adding or removing arbitrary elements is hidden from the client of the *Stack* class.

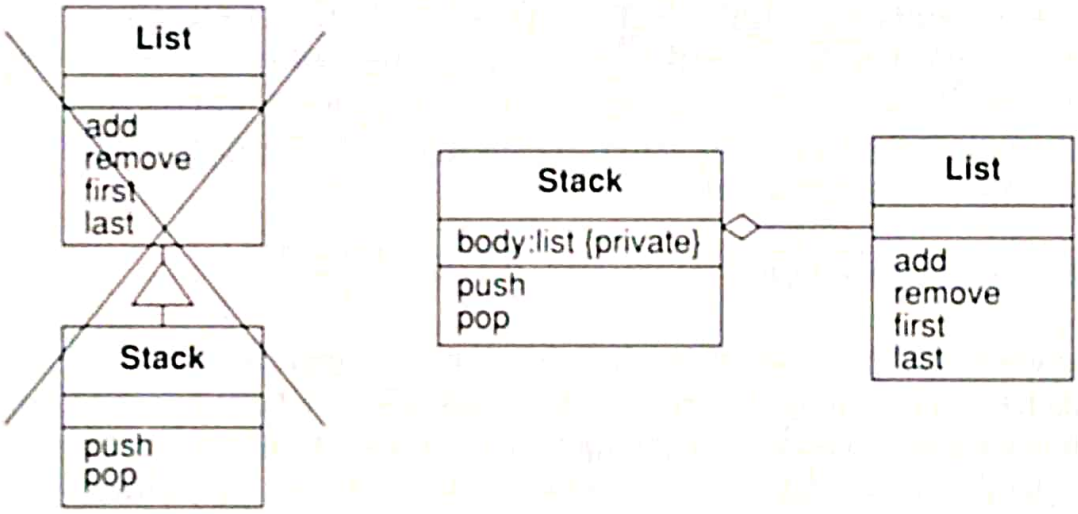


Figure 10.8 Alternative implementations of a Stack using inheritance (left) and delegation (right)

In general, it is best not to use inheritance for strictly implementation reasons. Reserve the use of inheritance for cases where you can say that an instance of one class also *is* an instance of some other class.

Some languages, such as Eiffel and C++, permit a subclass to inherit the form of a superclass but to selectively inherit operations from ancestors and selectively export operations to clients. This is tantamount to the use of delegation, because the subclass *is not* a form of the superclass in all respects and is not confused with it.

10.7 DESIGN OF ASSOCIATIONS

Associations are the "glue" of our object model, providing access paths between objects. Associations are conceptual entities useful for modeling and analysis. During the object design phase we must formulate a strategy for implementing the associations in the object model. We can either choose a global strategy for implementing all associations uniformly, or we can select a particular technique for each association, taking into account the way it will be used in the application. To make intelligent decisions about associations, we first need to analyze the way they are used.

10.7.1 Analyzing Association Traversal

We have assumed until now that associations are inherently bidirectional, which is certainly true in an abstract sense. But if some associations in your application are only traversed in one direction, their implementation can be simplified. Be aware, however, that the requirements on your application may change, and you may add a new operation later that needs to traverse the association in the reverse direction.

For prototype work, we always use bidirectional associations so that we can add new behavior and expand or modify the application rapidly. For production work we optimize some associations. Whichever implementation strategy you choose, you should hide the implementation using access operations to traverse and update the association. This will allow you to change your decision with minimal effort.

10.7.2 One-way Associations

If an association is only traversed in one direction, it may be implemented as a *pointer*—an attribute which contains an object reference. If the multiplicity is "one," as shown in Figure 10.9, then it is a simple pointer; if the multiplicity is "many," then it is a set of pointers. If the "many" end is ordered, then a list can be used instead of a set. A qualified association with multiplicity "one" can be implemented as a dictionary object. (A dictionary is a set of value pairs that maps selector values into target values. Dictionaries are implemented efficiently in most object-oriented languages using hashing.) Qualified associations with multiplicity "many" are rare, but they can be implemented as a dictionary of sets of objects.

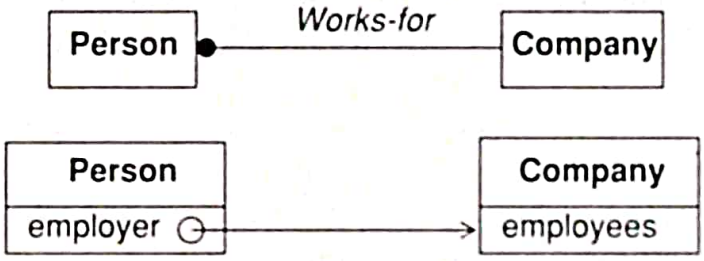


Figure 10.9 Implementation of one-way association using pointers

10.7.3 Two-way Associations

Many associations are traversed in both directions, although not usually with equal frequency. There are three approaches to their implementation:

- Implement as an attribute in one direction only and perform a search when a backward traversal is required. This approach is useful only if there is a great disparity in traversal frequency in the two directions and minimizing both the storage cost and the update cost are important. The rare backward traversal will be expensive.
- Implement as attributes in both directions, using the techniques outlined in the previous section and shown in Figure 10.10. This approach permits fast access, but if either attribute is updated then the other attribute must also be updated to keep the link consistent. This approach is useful if accesses outnumber updates.

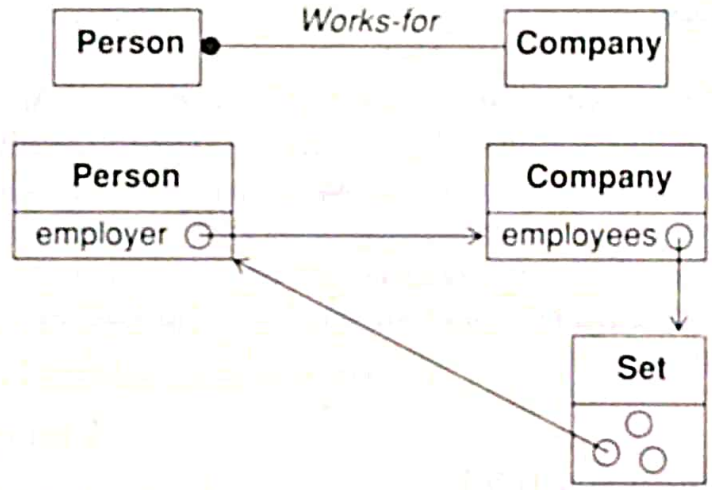


Figure 10.10 Implementation of two-way association using pointers

- Implement as a distinct association object, independent of either class, as shown in Figure 10.11 [Rumbaugh-87]. An association object is a set of pairs of associated objects (triples for qualified associations) stored in a single variable-size object. For efficiency, an association object can be implemented using two dictionary objects, one for the forward direction and one for the backward direction. Access is slightly slower than with attribute pointers, but if hashing is used then access is still constant time. This approach is useful for extending predefined classes from a library which cannot be modified, because the association object can be added without adding any attributes to the original classes. Distinct association objects are also useful for sparse associations, in which most objects of the classes do not participate because space is used only for actual links.

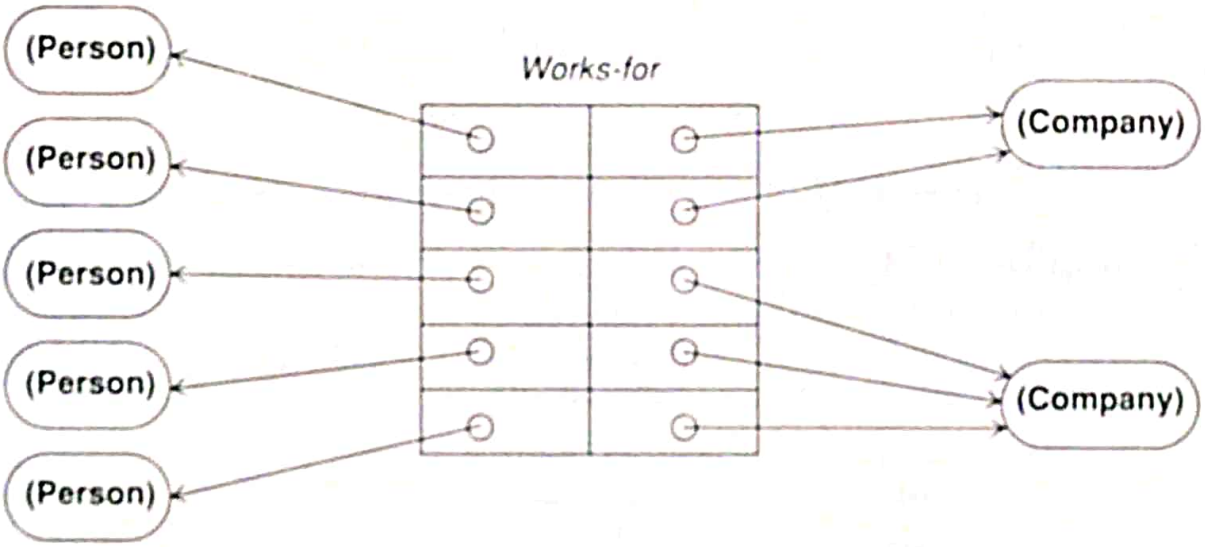


Figure 10.11 Implementation of association as an object

10.7.4 Link Attributes

If an association has link attributes, then its implementation depends on the multiplicity. If the association is one-to-one, the link attributes can be stored as attributes of either object. If the association is many-to-one, the link attributes can be stored as attributes of the "many" object, since each "many" object appears only once in the association. If the association is many-to-many, the link attributes cannot be associated with either object; the best approach is usually to implement the association as a distinct class, in which each instance represents one link and its attributes.

10.8 OBJECT REPRESENTATION

Implementing objects is mostly straightforward, but the designer must choose when to use primitive types in representing objects and when to combine groups of related objects.

Classes can be defined in terms of other classes, but eventually everything must be implemented in terms of built-in primitive data types, such as integers, strings, and enumerated types. For example, consider the implementation of a social security number within an employee object as shown in Figure 10.12. The social security number attribute can be implemented as an integer or a string, or as an association to a social security number object, which itself can contain either an integer or a string. Defining a new class is more flexible but often introduces unnecessary indirection.

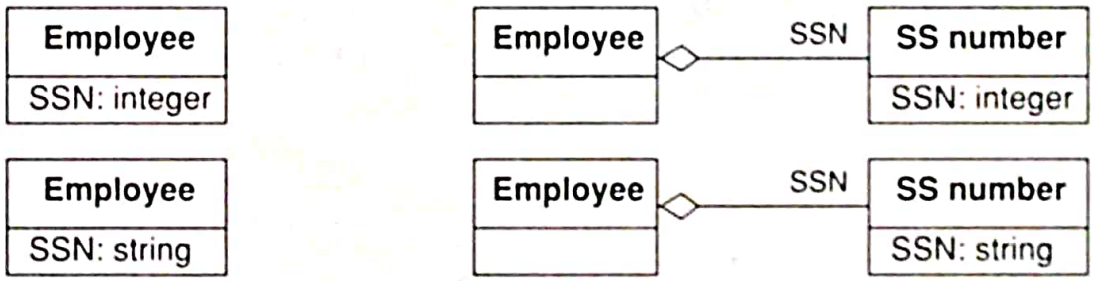


Figure 10.12 Alternative representations for an attribute

In a similar vein, the designer must often choose whether to combine groups of related objects. Figure 10.13 shows two common implementations of 2-dimensional lines, one as a separate class and one embedded as attributes within the *Point* class. Neither representation is inherently superior because both are mathematically correct.

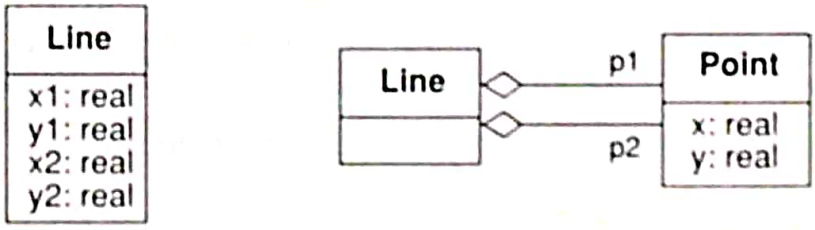


Figure 10.13 Embedded and explicit objects

10.9 PHYSICAL PACKAGING

Programs are made of discrete physical units that can be edited, compiled, imported, or otherwise manipulated. In some languages, such as C and Fortran, the units are source files. In Ada, the *package* is an explicit language construct for modularity. Object-oriented languages have various degrees of packaging. In any large project, careful partitioning of an implementation into packages (of whatever form) is important to permit different persons to cooperatively work on a program. Packaging involves the following issues:

- Hiding internal information from outside view
- Coherence of entities
- Constructing physical modules

10.9.1 Information Hiding

One design goal is to treat classes as "black boxes," whose external interface is public but whose internal details are hidden from view. Hiding internal information permits implementation of a class to be changed without requiring any clients of the class to modify code. Furthermore, additions and changes to the class are surrounded by "fire walls" that limit the effects of any changes so that changes can be understood clearly. There is a trade-off between information hiding and the optimization activities as discussed in Section 10.4. From the packaging viewpoint, we seek to minimize dependencies, while optimization takes advantage of details and may lead to redundant components and associations. The designer must balance these conflicting demands.

During analysis, we were not concerned with information hiding. During design, however, the public interface of each class must be carefully defined. The designer must decide what attributes should be accessible from outside the class. These decisions should be recorded in the object model by adding the annotation *{private}* after attributes that are to be hidden, or by separating the list of attributes into two parts.

Taken to an extreme, a method on a class could traverse all the associations of the object model to locate and access another object in the system. This unconstrained visibility is appropriate during analysis, but methods that know too much about the entire model are fragile because any change in representation invalidates them. During design we try to limit the scope of any one method. We need to define the bounds of visibility that each method requires. Specifying what other classes a method can see defines the dependencies between classes.

Each operation should have a limited knowledge of the entire model, including the structure of classes, associations, and operations. The fewer things that an operation knows about, the less likely it will be affected by any changes. Conversely, the fewer operations know about details of a class, the easier the class can be changed if needed. The following design principles help to limit the scope of knowledge of any operation:

- Allocate to each class the responsibility of performing operations and providing information that pertains to it.

- Call an operation to access attributes belonging to an object of another class.
- Avoid traversing associations that are not connected to the current class.
- Define interfaces at as high a level of abstraction as possible.
- Hide external objects at the system boundary by defining abstract interface classes, that is, classes that mediate between the system and the raw external objects.
- Avoid applying a method to the result of another method, unless the result class is already a supplier of methods to the caller. Instead consider writing a method to combine the two operations.

10.9.2 Coherence of Entities

One important design principle is *coherence* of entities. An entity, such as a class, an operation, or a module, is coherent if it is organized on a consistent plan and all its parts fit together toward a common goal. An entity should have a single major theme; it should not be a collection of unrelated parts.

A method should do one thing well. A single method should not contain both *policy* and *implementation*. *Policy* is the making of context-dependent decisions. *Implementation* is the execution of fully-specified algorithms. Policy involves making decisions, gathering global information, interacting with the outside world, and interpreting special cases. A policy method contains I/O statements, conditionals, and accesses data stores. A policy method does not contain complicated algorithms but instead calls various implementation methods. An implementation method does exactly one operation, without making any decisions, assumptions, defaults, or deviations. All its information is supplied as arguments, so the argument list may be long.

Separating policy and implementation greatly increases the possibility of reusability. The implementation methods do not contain any context dependencies, so they are likely to be reusable. The policy methods must usually be rewritten in a new application, but they are often simple and consist mostly of high-level decisions and calls on low-level methods.

For example, consider an operation to credit interest on a checking account. Interest is compounded daily based on the daily balance, but all interest for a month is lost if the account is closed. The interest crediting should be separated into two parts: an implementation method that computes the interest due between a pair of days, without regard to any forfeitures or other provisions; and a policy method that decides whether and for what interval the implementation method is called. This separation permits either the policy or the implementation to be modified independently and greatly increases the chance of reusing the implementation method, which is likely to be the more complicated. Policy methods are less likely to be reusable, but they are not usually as complicated because they do not contain computational algorithms.

A class should not serve too many purposes at once. If it is too complicated, it can be broken up using either generalization or aggregation. Smaller pieces are more likely to be