14

System Design

After you have analyzed a problem, you must decide how to approach the design. During system design you devise the high-level strategy—the ***system architecture***—for solving the problem and building a solution. You make decisions about the organization of the system into subsystems, the allocation of subsystems to hardware and software, and major policy decisions that form the basis for class design.

In this chapter you will learn about the many aspects that you should consider when formulating a system design. We also list several common architectural styles that you can use as a starting point. This list is not meant to be complete; new architectures can always be invented. The treatment 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 logistics. Most of the suggestions in this chapter are suitable for non-OO as well as OO systems.

14.1 Overview of System Design

During analysis, the focus is on *what* needs to be done, independent of *how* it is done. During design, developers make decisions about how the problem will be solved, first at a high level and then with more detail.

System design is the first design stage for devising the basic approach to solving the problem. During system design, developers decide the overall structure and style. The system architecture determines the organization of the system into subsystems. In addition, the architecture provides the context for the detailed decisions that are made in later stages. You must make the following decisions.

- Estimate system performance. [14.2]
- Make a reuse plan. [14.3]

- Organize the system into subsystems. [14.4]
- Identify concurrency inherent in the problem. [14.5]
- Allocate subsystems to hardware. [14.6]
- Manage data stores. [14.7]
- Handle global resources. [14.8]
- Choose a software control strategy. [14.9]
- Handle boundary conditions. [14.10]
- Set trade-off priorities. [14.11]
- Select an architectural style. [14.12]

You can often choose the architecture of a system by analogy to previous systems. Certain kinds of architecture pertain to broad classes of problems. Section 14.12 surveys several common architectures and describes their corresponding problems. Not all problems can be solved by one of these architectures, but many can. You can construct additional architectures by combining these forms.

14.2 Estimating Performance

Early in the planning for a new system you should prepare a rough performance estimate. Engineers call this a "back of the envelope" calculation. The purpose is not to achieve high accuracy, but merely to determine if the system is feasible. Getting within a factor of two is usually sufficient, although what you can achieve depends on the problem. The calculation should be fast and involve common sense. You will have to make simplifying assumptions. Don't worry about details—just approximate, estimate, and guess, if necessary.

ATM example. Suppose we are planning an ATM network for a bank. We might proceed as follows. The bank has 40 branches. Suppose there are an equal number of terminals in supermarkets and other stores. Suppose on a busy day half the terminals are busy at once. (We could assume all of the terminals are busy without changing the results much. The point is to establish reasonable performance limits.) Suppose that each customer takes one minute to perform a session, and that most transactions involve a single deposit or withdrawal. So we estimate a peak requirement of about 40 transactions a minute, or about one per second. This may not be precise, but it shows that we do not require unusually fast computer hardware. The situation would be much different if we were estimating for an online bookseller or stockbroker, in which case the computer hardware would become a big issue.

You can perform similar estimates for data storage. Count the number of customers, estimate the amount of data for each one, and multiply. In the case of a bank, the requirements for data storage are more severe than for ATM computing power, but they are hardly enormous. Again, the situation would be different for a satellite-based ground imaging system, in which both data storage and access bandwidth would be key architectural issues.

14.3 Making a Reuse Plan

Reuse is often cited as an advantage of OO technology, but reuse does not happen automatically. There are two very different aspects of reuse—using existing things and creating reusable new things. It is much easier to reuse existing things than to design new things for uncertain uses to come. Of course, someone must have designed things in the past in order for us to reuse them now. The point is that most developers reuse existing things, and only a small fraction of developers create new things. Don't feel that you should start with OO technology by building reusable things—that takes a great deal of experience.

Reusable things include models, libraries, frameworks, and patterns. Reuse of models is often the most practical form of reuse. The logic in a model can apply to multiple problems.

14.3.1 *Libraries*

A ***library*** is a collection of classes that are useful in many contexts. The collection of classes must be carefully organized, so that users can find them. Good organization takes a lot of work, and it can be difficult to decide where to place everything. Online searching can help, but is no substitute for careful organization. In addition, the classes must have accurate and thorough descriptions to help users determine their relevance. [Korson-92] notes several qualities of "good" class libraries.

- **Coherence**. A class library should be organized about a few, well-focused themes.
- **Completeness**. A class library should provide complete behavior for the chosen themes.
- **Consistency**. Polymorphic operations should have consistent names and signatures across classes.
- **Efficiency**. A library should provide alternative implementations of algorithms (such as various sort algorithms) that trade time and space.
- **Extensibility**. The user should be able to define subclasses for library classes.
- **Genericity**. A library should use parameterized class definitions where appropriate.

Unfortunately, problems can arise when integrating class libraries from multiple sources, as shown below [Berlin-90]. Developers often disperse pragmatic decisions across classes and inheritance hierarchies. Class libraries may adopt policies that are individually sensible, but fundamentally incompatible with those of other class libraries. You cannot fix such pragmatic inconsistencies by specializing a class or adding code. Instead, you must break encapsulation and rework the source code. These problems are so severe that they will effectively limit your ability to reuse code from class libraries.

- **Argument validation**. An application may validate arguments as a collection or individually as entered. Collective validation is appropriate for command interfaces; the user enters all arguments, and only then are they checked. In contrast, responsive user interfaces validate each argument or interdependent group of arguments as it is entered. A combination of class libraries, some that validate by collection and others that validate by individual, would yield an awkward user interface.

- **Error handling.** Class libraries use different error-handling techniques. Methods in one library may return error codes to the calling routine, for example, while methods in another library may directly deal with errors.
- **Control paradigms.** Applications may adopt event-driven or procedure-driven control. With event-driven control the user interface invokes application methods. With procedure-driven control the application calls user interface methods. It is difficult to combine both kinds of user interface within an application.
- **Group operations.** Group operations are often inefficient and incomplete. For example, an object-delete primitive may acquire database locks, make the deletion, and then commit the transaction. If you want to delete a group of objects as a transaction, the class library must have a group-delete function.
- **Garbage collection.** Class libraries use different strategies to manage memory allocation and avoid memory leaks. A library may manage memory for strings by returning a pointer to the actual string, returning a copy of the string, or returning a pointer with read-only access. Garbage collection strategies may also differ: mark and sweep, reference counting, or letting the application handle garbage collection (in C++, for example).
- **Name collisions.** Class names, public attributes, and public methods lie within a global name space, so you must hope they do not collide for different class libraries. Most class libraries add a distinguishing prefix to names to reduce the likelihood of collisions.

14.3.2 Frameworks

A ***framework*** [Johnson-88] is a skeletal structure of a program that must be elaborated to build a complete application. This elaboration often consists of specializing abstract classes with behavior specific to an individual application. A class library may accompany a framework, so that the user can perform much of the specialization by choosing the appropriate subclasses rather than programming subclass behavior from scratch. Frameworks consist of more than just the classes involved and include a paradigm for flow of control and shared invariants. Frameworks tend to be specific to a category of applications; framework class libraries are typically application specific and not suitable for general use.

14.3.3 Patterns

A ***pattern*** is a proven solution to a general problem. Various patterns target different phases of the software development lifecycle. There are patterns for analysis, architecture, design, and implementation. You can achieve reuse by using existing patterns, rather than reinventing solutions from scratch. A pattern comes with guidelines on when to use it, as well as trade-offs on its use.

There are many benefits of patterns. One advantage is that a pattern has been carefully considered by others and has already been applied to past problems. Consequently, a pattern is more likely to be correct and robust than an untested, custom solution. Also when you use patterns, you tap into a language that is familiar to many developers. A body of literature is available that documents patterns, explaining their subtleties and nuances. You can regard patterns as extending a modeling language—you need not think only in terms of primitives; you can also think in terms of recurring combinations. Patterns are prototypical model fragments that distill some of the knowledge of experts.

A pattern is different from a framework. A pattern is typically a small number of classes and relationships. In contrast, a framework is much broader in scope (typically at least an order of magnitude larger) and covers an entire subsystem or application.

ATM example. The notion of a transaction offers some possibility of reuse—transactions are a frequent occurrence in computer systems, and there is commercial software to support them. There may also be an opportunity for reuse with the communications infrastructure that connects the consortium to ATMs and bank computers.

14.4 Breaking a System into Subsystems

For all but the smallest applications, the first step in system design is to divide the system into pieces. Each major piece of a system is called a subsystem. Each subsystem is based on some common theme, such as 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 group of classes, associations, operations, events, and constraints that are interrelated and have a 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 processing I/O, drawing pictures, or performing arithmetic. A subsystem defines a coherent way of looking at part 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 independent of abstractions in other subsystems, such as memory management and process control.

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.

You should define subsystems so that most interactions are internal, rather than across subsystem boundaries. This reduces the dependencies among 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 relationship between two subsystems can be client-server or peer-to-peer. In a ***client-server relationship***, the client calls on the server, which performs some service and replies with a result. The client must know the server's interface, but the server need not know its clients' interfaces because clients initiate all interactions.

In a ***peer-to-peer relationship***, each subsystem 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 occur that are hard to understand and liable to subtle design errors. Look for client-server 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.

14.4.1 Layers

A ***layered system*** is an ordered set of virtual worlds (a set of *tiers*), each built in terms of the ones below it and providing the implementation basis 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 client-server relationship exists between upper layers (users of services) and lower layers (providers 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 affects only 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 the problem statement specifies only the top and bottom layers: The top is the desired system and 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 you must introduce intermediate layers to reduce the conceptual gap between adjoining layers.

You can port a system constructed in layers 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.

14.4.2 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 and avoids major design dependencies.

One difference between layers and partitions is that layers vary in their level of abstraction. In contrast, partitions merely divide a system into pieces, all of which have a similar level of abstraction. Another difference is that layers ultimately depend on each other, usually in a client-server relationship through an open or closed architecture. In contrast, partitions are peers that are independent or mutually dependent (peer-to-peer relationship).

14.4.3 Combining Layers and Partitions

You can decompose a system into subsystems by combining layers and partitions. Layers can be partitioned, and partitions can be layered. Figure 14.1 shows a block diagram of a typical application, which involves simulation and interactive graphics. Most large systems require a mixture of layers and partitions.

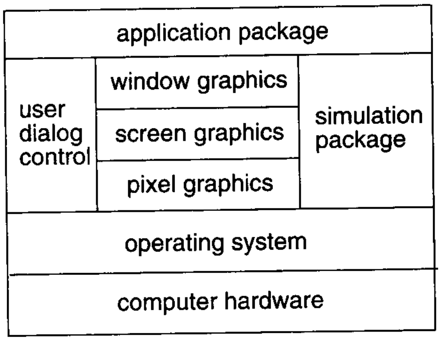

Figure 14.1 Block diagram of a typical application. Most large systems mix layers and partitions.

Once you have identified the top-level subsystems, you should show their information flow. 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 possible to reduce the number of interactions among subsystems.

ATM example. Figure 14.2 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 (comm links). The architecture uses the station code and the bank code to distinguish the phone lines to the consortium computer.

14.5 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

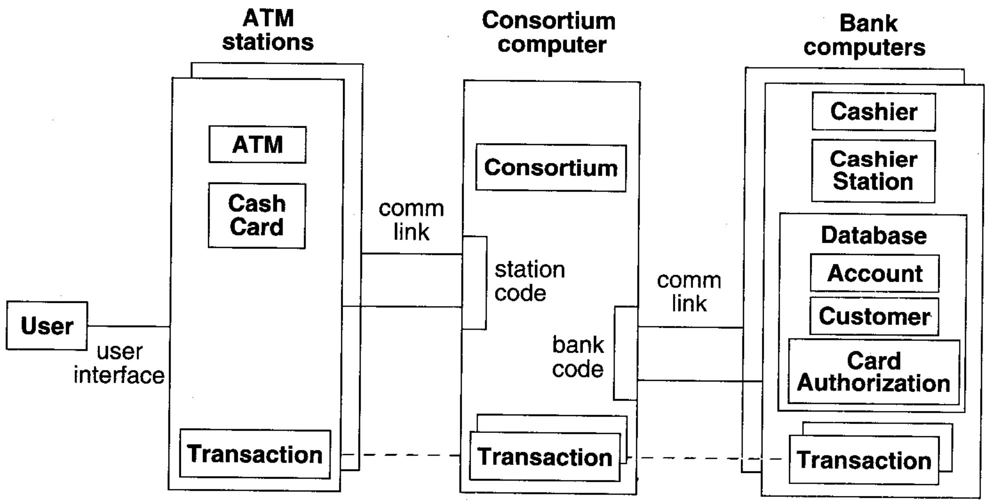

Figure 14.2 Architecture of ATM system. It is often helpful to make an informal diagram showing the organization of a system into subsystems.

may support many objects. In practice, you can implement many objects on a single processor if the objects cannot be active together. One important goal of system design is to identify the objects that must be active concurrently and the objects that have mutually exclusive activity. You can fold the latter objects onto a single thread of control, or task.

14.5.1 Identifying Inherent Concurrency

The state 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, you cannot fold the objects 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 you can assign them to different hardware units without any communication cost.

You need not implement two subsystems that are inherently concurrent as separate hardware units. The purpose of hardware interrupts, operating systems, and tasking mechanisms is to simulate logical concurrency in a uniprocessor. Separate sensors must, of course, process physically concurrent input, but if there are no timing constraints on response, then a multitasking operating system can handle the computation. Often the problem statement specifies that distinct hardware units must implement the objects.

ATM example. If the ATM statement from Chapter 11 contained the requirement that each machine should continue to operate locally in the event of a central system failure (perhaps with reduced transaction limits), then we would have no choice but to include a CPU in each ATM machine with a full control program.

14.5.2 *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, you can often fold many objects 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. You can implement threads of control as tasks in computer systems.

ATM example. While the bank is verifying an account or processing a bank transaction, the ATM machine is idle. If a central computer directly controls the ATM, we can combine the ATM object with the bank transaction object as a single task.

14.6 Allocation of Subsystems

You must allocate each concurrent subsystem to a hardware unit, either a general-purpose processor or a specialized functional unit as follows.

- 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.

14.6.1 *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. Many parallel machines must digest the data before analyzing a threat.

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. You should increase the estimate to allow for transient effects, due to random variations in load as well as to synchronized bursts of activity. The amount of excess capacity needed depends on the acceptable rate of failure due to insufficient resources. Both the steady-state load and the peak load are important.

ATM example. The ATM machine itself is relatively simple—all it must do is to provide a user interface and, possibly some local processing. At most a single CPU would suffice for each ATM. The consortium computer is essentially just a routing machine—it receives ATM requests and dispatches them to the appropriate bank computer. A large network might need to be partitioned in some way and involve multiple CPUs, so that the consortium computer does not become a bottleneck. The bank computers perform data processing and involve relatively straightforward database applications. The database vendors have single-processor and multiprocessor versions of their products, and the appropriate choice depends on the needed throughput and reliability.

14.6.2 Making Hardware-Software Trade-offs

Object orientation provides a good way for thinking about hardware. Each device is an object that operates concurrently with other objects (other devices or software). You must decide which subsystems will be implemented in hardware and which in software. There are two main reasons for implementing subsystems in hardware.

- **Cost.** 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.
- **Performance.** The system requires a higher performance 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. OO 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.

ATM example. There are no pressing performance issues for the ATM application. Hence general-purpose computers should suffice for the ATMs, consortium, and banks.

14.6.3 Allocating Tasks to Processors

The system design must allocate tasks for the various software subsystems to processors. There are several reasons for assigning tasks to processors.

- **Logistics.** Certain tasks are required at specific physical locations, to control hardware, or to permit independent operation. For example, an engineering workstation needs its own operating system to permit operation when the interprocessor network is down.
- **Communication limits.** The response time or data 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 limits.** Computation rates are too great for a single processor, so several processors must support the tasks. You can minimize communication costs by assigning highly interactive subsystems to the same processor. You should assign independent subsystems to separate processors.

ATM example. The ATM does not have any issues with communication and computation limits. The communication traffic and computation that an ATM user initiates are relatively minor. However, there may be an issue with logistics. If the ATM must have autonomy and operate when the communications network is down, then it must have its own CPU and programming. Otherwise, if the ATM is just a dumb terminal that accesses the network and performs all computation via the network, we can simplify ATM logic.

14.6.4 Determining Physical Connectivity

After determining the kinds and relative numbers of physical units, you must determine the arrangement and form of the connections among the physical units.

- **Connection topology**. Choose the topology for connecting the physical units. Associations in the class model often correspond to physical connections. Client-server relationships also correspond to physical connections. Some connections may be indirect; you should try to minimize the connection cost of important relationships.
- **Repeated units**. Choose the topology of repeated units. If you have boosted performance by including several copies of a particular kind of unit or group of units, you must specify their topology. The class model is not a useful guide, 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. You must consider the expected arrival patterns of data and the proposed parallel algorithm for processing it.
- **Communications**. 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 often it is appropriate to choose the general interaction mechanisms and protocols. For example, interactions may be asynchronous, synchronous, or blocking. You must estimate the bandwidth and latency of the communication channels and choose the correct kind of connection channels.

Even when the connections are logical and not physical, you must consider them. For example, the units may be tasks within a single operating system connected by interprocess communication (IPC) 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, you must combine the tightly linked tasks into a single task and make the connections by simple subroutine calls.

ATM example. Figure 14.2 summarizes physical connectivity. Multiple ATMs connect to the consortium computer and then are routed to the appropriate bank computer. The topology is a star where the consortium computer mediates communication.

14.7 Management of Data Storage

There are several alternatives for data storage that you can use separately or in combination: data structures, files, and databases. Different kinds of data stores provide trade-offs among cost, access time, capacity, and reliability. For example, a personal computer application may use memory data structures and files. An accounting system may use a database to connect subsystems.

Files are cheap, simple, and permanent. 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. Figure 14.3 characterizes the kind of data that belongs in files.

- Data with high volume and low information density (such as archival files or historical records).
- Modest quantities of data with simple structure.
- Data that are accessed sequentially.
- Data that can be fully read into memory.

Figure 14.3 Data suitable for files. Files provide a low-tech solution to data management and should not be overlooked.

Databases, managed by database management systems (DBMSs), are another kind of data store. Various types of DBMSs are available from vendors, including relational and OO. DBMSs cache frequently accessed data in memory in order to achieve the best combination of cost and performance from memory and disk storage. Databases make applications easier to port to different hardware and operating system platforms, since the vendor ports the DBMS code. One disadvantage of DBMSs is their complex interface—many database languages integrate awkwardly with programming languages. Figure 14.4 characterizes the kinds of data that belong in a database.

- Data that require updates at fine levels of detail by multiple users.
- Data that must be accessed by multiple application programs.
- Data that require coordinated updates via transactions.
- Large quantities of data that must be handled efficiently.
- Data that are long-lived and highly valuable to an organization.
- Data that must be secured against unauthorized and malicious access.

Figure 14.4 Data suitable for databases. Databases provide heavyweight data management and are used for most important business applications.

OO-DBMSs have not become popular in the mass market. Consequently you should consider them only for specialty applications that have a wide variety of data types or that must access low-level data management primitives. These applications include engineering applications, multimedia applications, knowledge bases, and electronic devices with embedded software. For most applications that need a database, you should use a relational DBMS (RDBMS). RDBMSs dominate the marketplace, and their features are sufficient for most applications. RDBMSs can also provide a very good implementation of an OO model, *if* they are used properly—Chapter 19 presents the details.

ATM example. The typical bank computer would use a relational DBMS—they are fast, readily available, and cost-effective for these kinds of financial applications.

The ATM might also use a database, but the paradigm for that is less obvious. Relational and OO-DBMSs would both be possibilities. Many OO-DBMSs permit access to low-level primitives, and a stripped-down database might enable mass production of ATM software at a low cost. A stripped-down database might also simplify ATM operation. Alternatively, RDBMSs are mature products with many features that might reduce development effort.

14.8 Handling Global Resources

The system designer must identify global resources and determine mechanisms for controlling access to them. There are several kinds of global resources.

- **Physical units**. Examples include processors, tape drives, and communication satellites.
- **Space**. Examples include disk space, a workstation screen, and the buttons on a mouse.
- **Logical names**. Examples include object IDs, filenames, and class names.
- **Access to shared data**. Databases are an example.

If the resource is a physical object, then it can control itself by establishing a protocol for obtaining access. 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.

You can avoid conflict by having a "guardian object" own each global resource and control access to it. One guardian object can control several resources. All access to the resource must pass through the guardian object. Allocating each shared global resource to a single object is a recognition that the resource has identity.

You can also partition a resource logically, assigning subsets 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. Do not use direct access to shared resources unless it is absolutely necessary.

ATM example. Bank codes and account numbers are global resources. Bank codes must be unique within the context of a consortium. Account codes must be unique within the context of a bank.

14.9 Choosing a Software Control Strategy

The analysis model shows interactions as events between objects. Hardware control closely matches the analysis model, but there are several ways for implementing control in software. Although all subsystems need not use the same implementation, it is best to choose 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 concerns 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 appropriate control style depends on the available resources (language, operating system) and on the kind of interactions in the application.

Internal control refers to the flow of control within a process. It exists only in the implementation and therefore is neither inherently concurrent nor 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 intertask 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 intertask calls, and concurrent intertask calls. Quasi-concurrent intertask calls, such as coroutines or lightweight processes, are programming conveniences in which multiple address spaces or call stacks exist but only a single thread of control can be active at once.

14.9.1 Procedure-driven Control

In a procedure-driven sequential system, control resides within the program code. Procedures request 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. This paradigm cannot easily accommodate asynchronous input, 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 with this style.

Note that all major OO languages, such as C++ and Java, are procedural languages. Do not be fooled by the OO 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 OO languages is that they fail to support the concurrency inherent in objects. Some concurrent OO languages have been designed, but they are not yet widely used.

14.9.2 Event-driven Control

In an event-driven sequential system, control resides within a dispatcher or monitor that the language, subsystem, or operating system provides. Developers attach application procedures to events, and the dispatcher calls the procedures 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. Consequently, the program counter and stack cannot preserve state. 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 control than procedure-driven systems. Event-driven systems simulate cooperating processes within a single multithreaded task; an errant procedure can block the entire application, so you must be careful. Event-driven user interface subsystems are particularly useful.

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 simpler and more powerful. Event-driven systems are also more modular and can handle error conditions better than procedure-driven systems.

14.9.3 Concurrent Control

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

14.9.4 Internal Control

During design, the developer expands operations on objects into lower-level operations on the same or other objects. Internal object interactions are similar to external object interactions, because you can use the same implementation mechanisms. However, there is an important difference—external interactions inherently involve waiting for events, because objects are independent and cannot force other objects to respond; objects generate internal operations as part of the implementation algorithm, so their form of response is predictable. Consequently, you can think of most internal operations as procedure calls, in which the caller issues a request and waits for the response. There are algorithms for parallel processing, but many computations are well represented sequentially and can easily be folded onto a single thread of control.

14.9.5 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. Developers currently use such languages 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 OO languages), the remainder of this book has little to say about them.

ATM example. Event-driven control is the appropriate paradigm for the ATM station. The ATM services a single user, so there is little need for concurrent control. The ATM must be responsive in its user interactions, and event-driven control is much better at that than procedure-driven control.

14.10 Handling Boundary Conditions

Although most of system design concerns steady-state behavior, you must consider boundary conditions as well and address the following kinds of issues.

- **Initialization**. The system must proceed from a quiescent initial state to a sustainable steady state. The system must initialize 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.

14.11 Setting Trade-off Priorities

The system designer must set priorities that will be used to guide trade-offs for the rest of design. These priorities reconcile desirable but incompatible goals. For example, a system can often be made faster by using extra memory, but that increases power consumption and costs more. Design trade-offs involve not only the software itself but also the process of developing it. Sometimes it is necessary to sacrifice complete functionality to get a piece of software into use (or into the marketplace) earlier. Sometimes the problem statement specifies priority, but often the burden falls on the designer to reconcile the incompatible desires of the client and decide how to make trade-offs.

The system designer must determine the relative importance of the various criteria as a guide to making design trade-offs. The system designer does not *make* all the trade-offs, but establishes the priorities for making them. 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, mathematical subroutine packages run 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.

Design trade-offs affect the entire character of a system. The success or failure of the final product may depend on how well its goals are 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 is really unimportant.

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. Subsequent design will still require judgment and interpretation when trade-offs are actually made.

ATM example. The ATM station is a mass-market product. Consequently, the manufacturing cost is a concern, and the resulting product must have a polished user interface. The software must be robust and resilient in the face of failure. Development cost is a lesser concern, since the cost can be amortized across numerous copies.

14.12 Common Architectural Styles

Several prototypical architectural styles 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. Some kinds of systems are listed below.

- **Batch transformation**—a data transformation executed once on an entire input set. [14.12.1]
- **Continuous transformation**—a data transformation performed continuously as inputs change. [14.12.2]
- **Interactive interface**—a system dominated by external interactions. [14.12.3]

- **Dynamic simulation**—a system that simulates evolving real-world objects. [14.12.4]
- **Real-time system**—a system dominated by strict timing constraints. [14.12.5]
- **Transaction manager**—a system concerned with storing and updating data, often including concurrent access from different physical locations. [14.12.6]

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 style or at least a variation on it. Many problems combine aspects of these architectures.

14.12.1 Batch Transformation

A ***batch transformation*** performs sequential computations. The application receives the inputs, and the goal is to compute an answer; there is no ongoing interaction with the outside world. Examples include standard computational problems such as compilers, payroll processing, VLSI automatic layout, stress analysis of a bridge, and many others. The state model is trivial or nonexistent for batch transformation problems. The class model is important—there are class models for the input, output, and the intervening stages. The interaction model documents the computation and couples the class models. The most important aspect of a batch transformation is to define a clean series of steps.

In the past, when we worked at GE R&D, one of our colleagues (Bill Premerlani) built a compiler that received an ASCII file of graphical pictures as input and generated relational database definition code as output. This work preceded the availability of commercial OO modeling tools. Figure 14.5 shows the sequence of steps. The compiler had five class models—one for the input, one for the output, and three for intermediate representations.

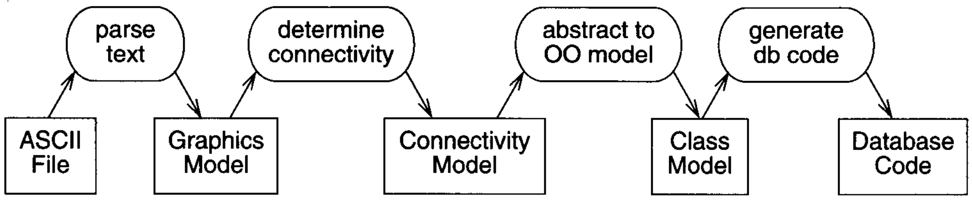

Figure 14.5 Sequence of steps for a compiler. A batch transformation is a sequential input-to-output transformation that does not interact with the outside world.

The steps in designing a batch transformation are as follows.

- Break the overall transformation into stages, with each stage performing one part of the transformation.
- Prepare class models for the input, output, and between each pair of successive stages. Each stage knows only about the models on either side of it.
- Expand each stage in turn until the operations are straightforward to implement.
- Restructure the final pipeline for optimization.

14.12.2 Continuous Transformation

A ***continuous transformation*** is a system in which the outputs actively depend on changing inputs. Unlike a batch transformation that computes the outputs only once, a continuous transformation updates outputs frequently (in theory continuously, although in practice they are computed discretely at a fine time scale). Because of severe time constraints, the system cannot recompute the entire set of outputs each time an input changes (otherwise the application would be a batch transformation). Instead, the system must compute outputs incrementally. Typical applications include signal processing, windowing systems, incremental compilers, and process monitoring systems. The class, state, and interaction models have similar purposes as with the batch transformation.

One way to implement a continuous transformation is with a pipeline of functions. The pipeline propagates the effect of each input change. Developers can define intermediate and redundant objects to improve the performance of the pipeline. Some high-performance systems, such as signal processing, need to synchronize values within the pipeline. Such systems perform operations at well-defined times and carefully balance the flow path of operations so that values arrive at the right place at the right time without bottlenecks.

Figure 14.6 shows the example of a graphics application. The application first maps geometric figures in user-defined coordinates to window coordinates. Then it clips the figures to fit the window bounds. Finally it offsets each figure by its window position to yield its screen position.

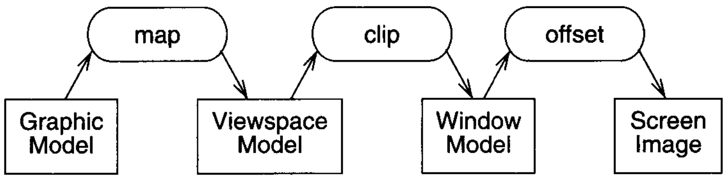

Figure 14.6 Sequence of steps for a graphics application. A continuous transformation repeatedly propagates input changes to the output.

The steps in designing a pipeline for a continuous transformation are as follows.

- Break the overall transformation into stages, with each stage performing one part of the transformation.
- Define input, output, and intermediate models between each pair of successive stages, as for the batch transformation.
- Differentiate each operation to obtain incremental changes to each stage. That is, propagate the incremental effects of each change to an input through the pipeline as a series of incremental updates.
- Add additional intermediate objects for optimization.

14.12.3 Interactive Interface

An ***interactive interface*** is a system that is dominated by interactions between the system and external agents, such as humans or devices. The external agents are independent of the system, so the system cannot control the agents, although it may solicit responses from them. An interactive interface usually includes only part of an entire application, one that can often be handled independently from computations. Examples of interactive systems include a forms-based query interface, a workstation windowing system, and the control panel for a simulation.

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, performance, and error handling. Interactive interfaces are dominated by the state model. The class model represents interaction elements, such as input and output tokens and presentation formats. The interaction model shows how the state diagrams interact.

The steps in designing an interactive interface are as follows.

- Isolate interface classes from the application classes.
- Use predefined classes to interact with external agents, if possible. For example, windowing systems have extensive collections of predefined windows, menus, buttons, forms, and other kinds of classes ready to be adapted to applications.
- Use the state model as the structure of the program. Interactive interfaces are best implemented using concurrent control (multitasking) 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, from a typed-in 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.

14.12.4 Dynamic Simulation

A ***dynamic simulation*** models or tracks real-world objects. Examples include molecular motion modeling, spacecraft trajectory computation, economic models, and video games. Simulations are perhaps the simplest system to design using an OO approach. The objects and operations come directly from the application. There are two ways for implementing control: 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 class model is usually important and often complex. Like an interactive system, the state and interaction models are also important.

The steps in designing a dynamic simulation are as follows.

- Identify active real-world objects from the class model. These objects 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, for example. 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.

Usually, the hardest problem with simulations is 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. Discrete steps must approximate continuous processes.

14.12.5 Real-time System

A ***real-time system*** is an interactive system with tight time constraints on actions. Hard real-time software involves critical applications that require a guaranteed response within the time constraints. In contrast, soft real-time software must also be highly reliable, but can occasionally violate time constraints. Typical real-time applications include process control, data acquisition, communications devices, device control, and overload relays.

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 this book.

14.12.6 Transaction Manager

A ***transaction manager*** is a system whose main function is to store and retrieve data. Most transaction managers deal with multiple users who read and write data at the same time. They also must secure their data to protect it from unauthorized access as well as accidental loss. Transaction managers are often built on top of a database management system (DBMS)—this is a form of reuse. A DBMS has generic functionality for managing data that you can reuse and need not implement. Examples of transaction managers include airline reservations, inventory control, and order fulfillment.

The class model is dominant. The state model is occasionally important, especially for specifying the evolution of an object as well as constraints and methods that apply at different points in time. The interaction model is seldom significant.

The steps in designing an information system are as follows.

- Map the class model to database structures. See Chapter 19 for advice.
- 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. A transaction succeeds or fails in its entirety.
- Design concurrency control for transactions. Most database management systems provide this. The system may need to retry failed transactions several times before giving up.

14.13 Architecture of the ATM System

The ATM system 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 a class model and a state model. The consortium and banks are primarily a distributed transaction management system. Their purpose is to maintain data and allow it to be updated over a distributed network under controlled conditions. Specifying the transaction management part of the system consists primarily of constructing a class model. Figure 14.2 shows the architecture of the ATM system.

The only permanent data stores are in the bank computers. A database ensures that data is consistent and available for concurrent access. The ATM system processes each transaction as a single batch operation, locking an account until the transaction is complete.

Concurrency arises because there are many ATM stations, each of which can be active at any time. There can be only one transaction per ATM station, but each transaction requires the assistance of the consortium computer and a bank computer. As Figure 14.2 shows, a transaction cuts across physical units; the diagram shows each transaction as three connected pieces. During design, each piece will become a separate implementation class. Although there is only one transaction per ATM station, there may be many concurrent transactions per consortium computer or bank computer. This does not pose any special problem, because the database synchronizes access to any one account.

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 has minimal functionality. It simply forwards a message from an ATM station to a bank computer and from a bank computer to an ATM station. The consortium computer must be large enough to handle the transaction load. It may be acceptable to block an occasional transaction, provided the user receives an appropriate message.

The bank computer is the only unit with any nontrivial procedures, but even those are mostly just database updates. The only complexity might come from failure handling. The bank computers must have capacity to handle the expected worst-case load, and they must have enough disk storage to record all transactions.

The system must contain operations for adding and deleting ATM stations and bank computers. Each physical unit must protect itself against the failure or disconnection from the rest of the network. A database protects against loss of data. However, special attention must be paid to failure during a transaction so that neither the user nor the bank loses money—this may require a complicated acknowledgment protocol before committing the transaction. The ATM station should display an appropriate message if the connection is down. The ATM must handle other kinds of failure 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 the ATM must abort the transaction with an appropriate message to the user.

14.14 Chapter Summary

After analyzing an application and before beginning the class design, the system designer must decide on the basic approach to the solution. The form of the high-level strategy for building the system is called the system architecture.

Early in the planning for a new system you should estimate the performance. The intention is to have a rough idea of what to expect. You want to make sure that it is reasonable and that there are no big surprises as development proceeds.

Next, prepare a reuse plan. Reuse is often cited as a benefit of OO technology, but it does not happen automatically. There are two different aspects of reuse. Most developers should focus on reusing existing models, libraries, frameworks, and patterns that are relevant to their applications. In addition, elite developers can create artifacts for reuse by others.

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 server of services for the layer or layers above it. Systems can also have partitions, each performing a general kind of service. Simple system topologies, such as pipelines or stars, reduce complexity. Most systems are a mixture of layers and partitions.

Inherently concurrent objects execute in parallel, and a single thread of control cannot combine them; they require separate hardware devices or separate tasks in a processor. You can combine nonconcurrent objects onto a single thread of control and implement them as a single task.

A system must have enough processors and special-purpose hardware units to meet performance goals. You should assign objects to hardware so that hardware use is balanced and meets concurrency constraints. You can do this by estimating computational throughput and allowing for queuing effects in configuring the hardware. You may want to use special-purpose hardware for compute-intensive computations. One goal in partitioning a hardware network is to minimize communications traffic between physically distinct modules.

Data stores can cleanly separate subsystems within an architecture and give application data some degree of permanence. In general, memory data structures, files, and databases can implement data stores. Files are simple, cheap, and permanent but may provide too low a level of abstraction for an application and necessitate much additional programming. Databases provide a higher level of abstraction than files, but they too involve compromises in terms of overhead costs and complexity.

The system designer must identify global resources and determine mechanisms for controlling access to them. Some common mechanisms are: establishing a "guardian" object that serializes all access, partitioning global resources into disjoint subsets which are managed at a lower level, and locking.

Hardware control is inherently concurrent, but software control can be procedure driven, event driven, and concurrent. Control for a procedure-driven system resides within the program code; the location of the program counter and the stack of procedure calls and local variables define the system state. In an event-driven system control resides within a dispatcher or monitor; application procedures are attached to events and are called by the dispatcher when the corresponding events occur. In a concurrent system, control resides concurrently in multiple independent objects. Event-driven and concurrent implementations are much more flexible than procedure-driven control.

Most of system design is concerned with steady-state behavior, but boundary conditions (initialization, termination, and failure) are also important.

An essential aspect of system architecture is making trade-offs between time and space, hardware and software, simplicity and generality, and efficiency and maintainability. These trade-offs depend on the goals of the application. The system designer must state the priorities, so that trade-off decisions during subsequent design will be consistent.

Several kinds of systems are frequently encountered for which standard architectural styles exist. These include two kinds of functional transformations: batch computation and continuous transformation; three kinds of time-dependent systems: interactive interface, dynamic simulation, and real-time; and a database system: transaction manager. Most application systems are usually a hybrid of several forms, possibly one for each major subsystem. Other kinds of architecture are possible.

architecture	hardware requirements	service
client-server	inherent concurrency	subsystem
concurrency	layer	system design
data management	partition	system topology
event-driven system	peer-to-peer	thread of control
framework	reuse plan	trade-off priorities

Figure 14.7 Key concepts for Chapter 14

Bibliographic Notes

Simple software applications do not require much systems engineering, but complex systems must be decomposed and the parts assigned to the appropriate specialists. [Clements-02] presents a process for evaluating software architectures. Essentially a group of stakeholders meet and prioritize criteria that the architecture should satisfy; they quantify the criteria with