Chapter 16

Writing Larger Systems

As software systems get larger, the methods and techniques used to organize, develop, and implement them change. The problems surrounding the development of large software systems are inherently different from those of smaller systems. The solutions to these problems, the approaches necessary to ensure success, and the overall philosophy of software development have all evolved to deal with large-scale development.

Larger software systems are inherently more complex than smaller systems. This has several implications:

- Large software systems are more prone to failure than smaller ones. There are more places in the design and code where errors can creep in. Problems are more difficult to find and fix correctly. Testing, because of the sheer number of different options, alternatives, and program states, is much more difficult to do comprehensively. Design and implementation techniques must take into account and mitigate these problems.
- Large software systems require multiple programmers. Even the most productive programmers do not write much more than 50,000 lines of code a year. At this rate, a million-line system would take twenty years to write and would be out of date before it started working. Large systems must be designed and managed to let teams of programmers work on them.
- Large software systems are difficult to understand. The overall gestalt of how the system works is buried under multiple levels of detail and is typically invisible to the individual programmer. No one programmer can be expected to understand all aspects of the system. The system must be designed and developed in a compartmentalized manner on a need-to-know basis that minimizes the amount of detail any one person must learn.

In this chapter we look at some of the issues arising in the development of larger systems. The techniques we emphasize are applicable to at least moderate-sized systems of up to several hundreds of thousands of lines, and work well for smaller systems of tens of thousands of lines. Larger systems, with millions or tens of millions of lines of code, require more emphasis on management issues and additional design and coding techniques.

The chapter is broken into three main parts. We start with a brief overview of larger problems and how to get started, and then cover techniques for developing practical designs for larger systems. This is followed by a discussion of management issues for larger projects.

GETTING STARTED

The first steps in developing a large-scale project are much the same as for a small-scale project. However, the overall success of a large-scale project is much more dependent on doing a good job here than in a small-scale project. Errors in identifying the problem or its potential solution or errors at the top level of design will be magnified by the scale of the problem and can prove ruinous to the overall software effort.

Requirements Analysis and Specifications

The first step in undertaking a software project, then, is thoroughly to understand the problem and its potential solution. The methodologies for doing this were covered in "Requirements Analysis" on page 405 and "Specifications" on page 406. This understanding is gained in two phases. The first involves understanding the problem from the user's perspective. Here the developer must work closely with potential users to understand the needs the software system is planned to meet. This can involve interviewing users, distributing questionnaires, working with users to understand how things are currently done, and analyzing the problems of current techniques or systems.

The second phase in understanding the software system involves looking at the potential solution from the programmer's point of view. Here one attempts to detail exactly what the eventual system will do, describing the various interfaces it will provide as well as particulars on the actions it will perform. A multipronged approach is used to derive this specification. The principal technique is to develop a model of the proposed solution and then describe that model in detail. The model is typically based on data- and control-flow diagrams, as discussed in the previous chapter. The important point here is not how the system works, but what it does. These diagrams are augmented with detailed explanations of the function each box or arc performs.

The second part of the specifications includes a tentative design of the system's various user interfaces. This design can be described either through appropriate sketches or through a user-interface prototype that lets the designer and perhaps even potential users test whether the system will meet its requirements. In principle, the description should provide enough detail that a user manual could be written for the proposed system.

The final part of the specifications puts these two items together, along with any additional information such as system and portability requirements,

to produce a specification document. This document is the foundation for the subsequent design and implementation of the software system. Just as in a building, it is the strength of this foundation that determines whether the system will stand up or collapse. Attempting to build a large system that is poorly defined or understood before one starts is a sure step toward failure.

In Chapter 15 we introduced a simple spacewar problem as an example for large-scale software development. We illustrated there how to develop requirements from the user's point of view and showed part of the result in Figure 15-1. Then we developed a system model, showing the data flow in Figure 15-2, user interfaces in Figure 15-3, and control flow in Figure 15-4. Samples from the overall specification were shown in Figure 15-5. We use this problem and these specifications as the starting point for our discussions here.

Design

An object-oriented design for a large system is developed in much the same way as for a small system. The differences are a much stronger emphasis on hierarchy in design, an emphasis on interfaces and interface definition, and a variety of techniques for selecting and organizing the implementation classes so as to minimize risk and make it easier for multiple programmers to work on the system simultaneously.

The design process starts by looking for candidate classes. This process is much the same as for a smaller program, except that the starting point here is the specifications document that completely describes the target system. The data-flow diagrams in this document provide one set of candidate classes, since each data element here can correspond to a class, as can each of the action boxes. The user-interface diagrams provide a second set of classes, with one top-level class corresponding to each interface and separate classes underneath this for each component of the interface. Another set of candidate classes can be derived from the textual specifications; if this is broken down into subsystems, there should be a class for each subsystem. A part-of-speech analysis of this specification text can produce more candidate classes. A final set of classes can be obtained by understanding the solution and how it works under one or more scenarios. These could be written either as part of the specification (say as the basis for the user-interface diagrams) or by the designer in order to understand the software better.

The difficulty with a large-scale system is that the number of potential classes is so enormous that it is impractical to list them all. The designer has to begin using hierarchy and eliminating classes from consideration even before the design begins. The initial goal here should be to develop 20 to 30 high-level classes from which the overall design of the system can evolve.

Figure 16-1 shows the initial set of candidate classes for the spacewar program. This set has been pruned from the overall set of classes we could have identified. For example, rather than listing all the different types of entities in the game such as spaceships, missiles, stars, and explosions, we include only

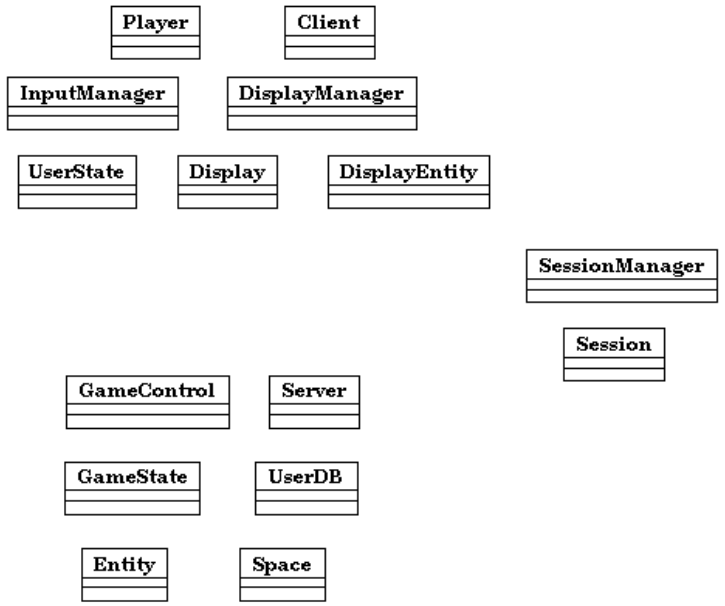

Figure 16-1 Candidate top-level objects for the spacewar program.

the candidate class **Entity**. We also exclude some obviously lower-level classes such as thrusters on the spaceship and gravity for doing computation.

The other thing we have done, as suggested in "Top-Level Design" on page 140, is to cluster the candidate classes. Here we formed three clusters. The top cluster contains information on each individual player. The **Player** class represents the player; the **Client** class represents a connection to the server. The **InputManager** and **DisplayManager** classes handle input from and output to the user respectively, and the **UserState** class represents information about the state of the game as seen by the user. Finally, the **Display** and **DisplayEntity** classes represent the actual display and the items on the display.

The second cluster, at the bottom, represents information needed or used by the server. The **GameControl** class handles the game logic while the **GameState** class stores and manages the current state of the game. The **Server** class represents the connection between the server and each client and the **UserDB** class represents the database of information on how often each user has won a round. The **Entity** and **Space** classes represent information in the game state.

The final cluster has a class representing a session and another class for managing sessions in some way. These are needed to implement some of the connection logic brought out in Chapter 15.

The next step in small-scale object-oriented design would be to select a subset of these classes to represent the top-level design. Here we would select a

set of five to 10 classes encompassing everything else in the design. These classes would then be characterized by defining their obvious data fields and interconnections as well as any top-level methods. We would proceed from there to provide pseudocode for these methods, introducing new methods and secondary classes as needed.

In large-scale design the approach is similar but much more compartmentalized. We first identify the set of top-level classes. In doing this, we look not for the most logical set but instead for the set that breaks the problem up into the most plausible set of independent pieces and maximizes our chances of building a successful system. Next we try to define the interfaces among these pieces without referring to the implementations underlying them. Once proper interfaces are designed, each component can be treated as an independent system on its own and can be designed and implemented using object-oriented techniques. All this is described in more detail in the next sections.

The essential idea in large-scale design is to break the program up into well-defined and independent components. This lets different programmers tackle each component without having to interact excessively with other programmers. Complete interfaces ensure each component is well defined and let the programmers work on their own component without needing to know the details of other components. A high degree of independence ensures that components can be implemented, tested, upgraded, and even replaced without adversely affecting the rest of the system.

Achieving such a design takes practice and understanding. Experienced designers can generally look at a problem, understand it, and then suggest a workable overall design, using their experience with similar systems and their understanding of what types of designs work and which ones don't. They also use methodologies that minimize potential problems and let the design be flexible enough to allow future changes.

DESIGN BY SUBSYSTEM

There are a variety of techniques for selecting and organizing the small set of top-level classes in a large-scale design. These techniques focus on providing independent components with well-defined interfaces and on ensuring that the system built from these components ultimately works.

The simplest technique is that outlined in the previous section. Here the designer identifies a set of candidate classes, clusters them, and then uses the clusters to select a small final set of top-level classes. While this technique is always helpful, it is not in general sufficient for most problems: the set of candidate classes is sometimes incomplete or wrong, the groupings are based on too little information, and the result is often not the best design. This approach is thus often augmented with other techniques.

One of the most powerful such techniques involves adding rather than removing classes. Here we apply a facade design pattern, as described in "Facade" on page 339, to create a new class to be a front end for a set of related classes. The top-level design can then use the facade class and ignore the classes it is a facade for.

This is an instance of *design by subsystem*. The facade class identifies a high-level component or subsystem of the overall program. It replaces a cluster of classes in the initial design with a single class and thereby simplifies the overall design. The actual classes represented by the facade can be defined at a lower level of detail and can be ignored at the top level of the design.

Thus design by subsystem offers advantages in terms of both simplicity and information hiding. It also lets the designer concentrate on the external interfaces needed by the set of classes the facade represents without having to worry about the interfaces among the implementations of these classes. Since these external interfaces both are the important details at the top level and are essential for getting the remainder of the system working, this generally leads to a better and easier-to-understand overall design.

Care must be taken in defining subsystems using facades: if the subsystem is not correctly identified, the overall design is generally more complex and weaker, not simpler and stronger. The essential element is choosing the right set of initial classes as the underlying subsystem. These classes should have a common purpose so that the facade class is cohesive: it should be possible to state the function of the facade class in a simple, non-compound clause. The grouped classes should also exhibit some coupling: most of the communication these classes do should be among themselves, not with other components of the system. This ensures that the facade is a gateway and not a hindrance to the eventual implementation. Finally, the classes should be chosen so that classes outside of the identified subsystem do not need detailed access to individual elements within the facade.

Subsystems that can be defined naturally will greatly simplify the design and generally lead to a better system. Thus we recommend:

Design a large system with subsystems in mind.

RISK-BASED DESIGN

Another technique useful in selecting a good set of top-level classes is to minimize the risk that the design may be bad or the system may fail. A large system has no "correct" design. Instead, there is a wide range of acceptable designs, some better than others. But while a number of designs will work, there is also a broad range of designs that will yield a marginal or nonfunctional system. A first step in the design process is to ensure that a system based on the design one is developing will work.

One way to do this is to identify the critical issues of the problem and design either for or around these. A novice designer has trouble identifying which aspects of the design are fundamental. A good starting point, however, is to determine the portions of a potential solution that seem to be the most difficult.

There are several ways of identifying the difficult issues in a system. One can start with the specifications as they are presented. While the specification model describes what the system does, it is not a big jump for the designer to attempt to determine how each of the actions described could be undertaken. Any actions whose potential implementation is not clear should be identified as potential problems. The designer should list the potential problems and then order them by difficulty.

Building a Design Model

While this is a starting point, it is unlikely to be sufficient to identify all the potential difficulties or even the most crucial ones. Many of the thorniest design problems arise because the implementations of several different actions are inconsistent with one another. Locating these problems is difficult. The best approach is actually to build a model of a potential solution using the set of candidate classes and go through the specifications to ensure that all actions can be done within the model. Where different actions conflict, one can either change the model and try again or just note the potential problem.

An initial design model can be most easily derived by starting with the data-flow diagram contained in the specifications, which shows the basic components and data structures. In an object-oriented implementation, both the components and data structures will be viewed as classes, so the diagram can readily be mapped into an object-oriented framework. Of course, the diagram should be augmented with any additional assumptions and should attempt to show object communication rather than simple data flow.

Figure 16-2, an initial design model for the spacewar program, shows boxes representing classes and arcs representing connections among them. Note that the connections here indicate that one class needs to interact with another — they are not the more constrained associations typical of a static structure diagram. If we ultimately accept this design, the relationships will be refined in a future step. This diagram breaks the design into three components. The first, represented by the five classes in the upper left, is the code for each user. The second, represented by the five classes in the upper right, is the server. The remaining component, the session manager, contains the two classes at the bottom.

An essential component of such a model is a description of how each component works. The **Player** class is the manager for the user code. It uses the **Client** class for all communication with the server or with the session manager at start-up. It creates a model of what should be displayed and the state of the game in the **UserState** class. The **InputManager** class uses this to determine

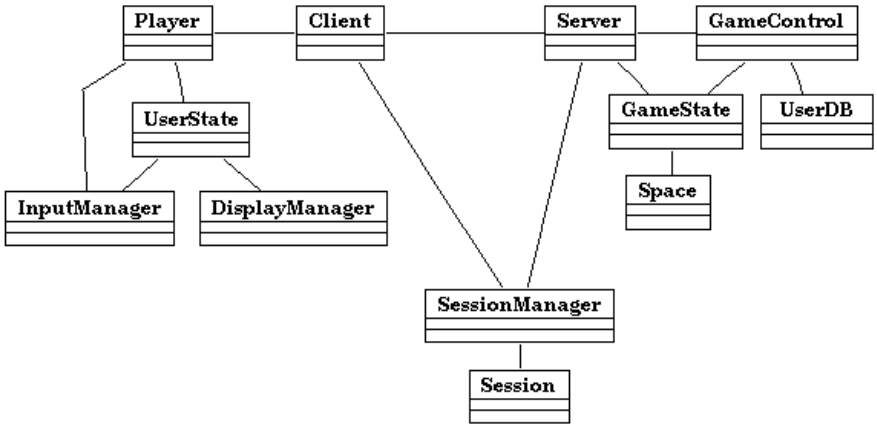

Figure 16-2 Design model for the spacewar program.

what inputs are valid before passing them onto the **Player** object to send to the server. The **DisplayManager** class reads the game state and continually puts up the corresponding display.

The server component is centered around the **GameControl** and **GameState** classes. The former manages the overall server, determining when to connect, who is playing, and when to start a game. The latter manages each actual game, keeping track of all the objects and their positions and using the **Space** class to store the objects. The **UserDB** class manages the set of users and their cumulative scores. Finally, the **Server** class manages all communication with the users and with the session manager.

The class design of the session manager itself is fairly simple. The **SessionManager** class keeps track of currently active sessions and starts a new session when appropriate. The **Session** class represents a single session and the information about that session.

Identifying Difficult Problems

Once a design model is developed, the designer should ensure that each action in the specifications can be accomplished within the framework the model provides. Again, any actions difficult to describe within the framework should be added to the list of potential problems. Finally, the set of potential problems should be sorted by order of difficulty.

Several potential problems in the spacewar program can be identified by this process:

- How to convey the game state from the server to the client in a timely fashion. The client display must update at least 10 times per second for smooth animation. Can the server send the game state to the client this

fast, can the server send several frames at once, or should the client handle some object movement?

- How to identify the active sessions when a new user wants to connect to the system. The session manager needs to know what is currently running; moreover, the client and server both need to talk to the session manager.
- How to achieve smooth animation on the client display. Should we use xor, as in the bouncing balls example? Should we use a fixed background display and just update the moving objects? Should we do double buffering? Or are overlay planes available for moving objects?
- When to update the display. Should the display be updated automatically to achieve a constant frame rate, or only when new information comes to the client from the server? In the former case, what assumptions can be made about moving objects? In the latter, should we assume a fixed delay between frames or should we update immediately?

Solving the Difficult Problems

Once the difficult problems are identified, the next step is to focus the design to solve or isolate those problems. For each problem, the designer should determine whether a solution in the current framework is possible. This can be done either analytically or, where necessary, by implementing a prototype to test possible alternatives.

Consider the first problem above. This is crucial to the overall program. If the server can't provide the information fast enough to the client, then we must restructure the solution completely so that the game is actually played independently at each client, with the server simply coordinating input and ensuring that the various clients remain consistent. This is a much more difficult design to implement, but it can be done. Here we could run a few experiments to determine the number of messages per second the server can send to the client to determine if a problem will arise. It turns out that hundreds of messages per second can be sent without putting too great a load on either the server or the client. Since this is much greater than the ten-message-per-second target, we should be okay. However, we will have to take care in designing the server to ensure that messages are sent frequently, and in designing the client to ensure that messages are read frequently.

Next consider the second problem. Determining the active sessions is not as easy as it sounds. Ideally we want to put the session manager in the client. If we use files to store the different server sockets' address, then we can go through whatever directory these are stored in and identify the different sessions, since each server corresponds to a session. However, if a server crashes or doesn't remove its address file, this solution will make spurious sessions be reported. This could be alleviated by having the server lock the file using system-wide locking and having the session manager in the client check to see if

the file is locked. A little experimentation here shows that this almost works, but locks are not reliable and can still exist for an extra five or ten minutes if a server crashes.

The preferred alternative solution is to view the session manager as a separate process in the design. Both the server and the client connect to the session manager when they start up. The session manager keeps track of the servers currently connected and reports that information to the client when asked. It also detects when a server disappears through the loss of the connection and can update its database accordingly. Here, then, the solution has a strong effect on the eventual system design.

The third problem involves the best way to handle the display. Here we know that different solutions are possible and that at least one of the solutions will work. We may want to experiment with the different solutions to see what works best, but this is really an implementation detail and need not be considered at the top level of the design. What we want to do for problems like this is to ensure that the top-level design isolates the problem within a class. The display manager class should be the only class affected by the technique eventually used here. Moreover, the interface to this class should not reflect the solution but should be independent of it. Here we are using information hiding to isolate the potential problem from the rest of the design. Doing this lets us change the implementation of the display manager as needed without affecting the rest of the system.

The last problem again involves the display. The mechanism used to trigger updates will depend to some extent on how animation is done. It will also depend on how fast the server is and on aesthetics, which are difficult to determine without actually using the system. Our design should again isolate this decision within the display class so that the rest of the system is not affected. The one difficult aspect of this is giving a way to guess an interim position for an entity if no new information is obtained from the server. To avoid this, we assume that the server provides updates at least as fast as the frame rate, possibly faster. Otherwise, we would have to do linear interpolation within the display component itself, with each display entity keeping track of its current velocity (as the change of position between frames) and possibly its current acceleration (as the change of velocity between frames). Note that here we are changing the specifications by putting an additional constraint on one of the other components to ensure that the problem can be solved.

This approach of identifying and either solving or isolating the difficult problems attempts to reduce the risks in the solution. It provides a good starting point for the design, offering points to focus on and identifying classes and interface requirements. The spiral model of software development, introduced in “Prototypes and the Spiral Model” on page 419, formalizes this and incorporates it into industrial practice. For most moderate-sized systems, however, just identifying the potential problems and using them as the focus for the

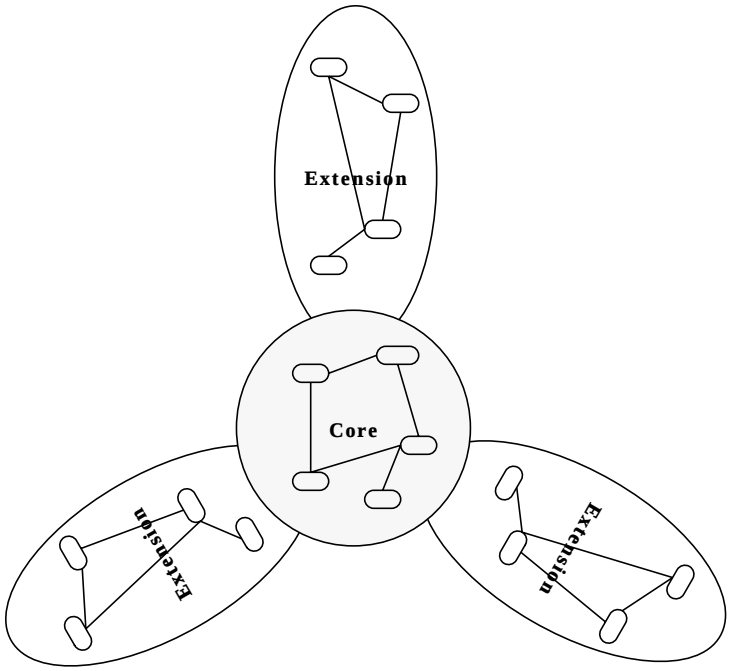

Figure 16-3 Core-plus-extensions design.

design model and the eventual design should be sufficient. Thus we recommend:

> **Design a system with the difficult problems and their solutions in mind.**

CORE-PLUS-EXTENSIONS DESIGN

Another technique for large-scale design focuses on organizing the system so that it consists of a small core and various extensions that plug into the core, as shown in Figure 16-3. This is called a *core-plus-extensions* design.

The core of a core-plus-extensions design is a small set of *interface classes*, classes that other components can see. They are publicly available, generally by being defined in public header files. There should be no more than 10 such classes. The core also includes whatever support classes are needed to implement these public classes, and represents the heart of the system, providing the basic functionality needed by all the other components.

The extensions are designed to plug into the core and interact with the rest of the system only through the interface classes of the core; they do not inter-

act with one another. Extensions can give other extensions such basic services as input and output, but they do so through methods provided by the core. However, most extensions provide optional functionality that expands the basic system to meet the specifications better. Such functionalities are sometimes viewed as *features*, so that the core-plus-extensions approach boils down to a core system onto which any number of features can be grafted.

The primary objective in a core-plus-extensions design is to keep the core as small as possible, using the independent extensions for most of the system's actual functionality. All programmers on a project need to understand thoroughly the interface to the core, since all extensions must fit neatly and precisely into the framework the core provides. The core must be designed and implemented before any of the extensions.

A logically simple core consisting of a small set of classes is easy for programmers to understand completely. It provides more flexibility and a better interface for attaching the extensions, and it should be easier to design, implement, test, and debug so that it should be ready earlier. Putting most of the system's functionality into extensions makes the system inherently more flexible and easier to evolve. It also gives the various programmers more independence, leading to fewer misunderstandings and higher productivity.

Pros and Cons

The core-plus-extensions design methodology has several advantages:

- A basic working system is available early in the development cycle. This is good for the programmers psychologically since they have something tangible to show for their efforts. It is also good for the system since it can help programmers understand the strengths and weaknesses of the design and correct any flaws early on.
- Communications among programmers are minimized. Once the core of the system is written, programmers working on separate extensions need only minimal communication with other programmers, since extensions do not interact directly. This should enhance productivity in a multiple-person project. Also, the overall system is less dependent on any one programmer. If an extension is incomplete because a programmer was ill or incompetent, the other extensions and the rest of the system can still be developed and tested.
- The framework provides a good basis for testing. The core, as the most important part of the system, is generally tested first using drivers. It is then tested again and again as new extensions are added, thereby ensuring that the heart of the system is the part most tested. Extensions can be tested one at a time as they are added to the core.
- The overall system should be easy to extend and evolve. If the core is specified correctly, it should be simple to extend the system by adding new functionality, whether by replacing existing extensions or adding

new ones. Porting to a new operating system or to new hardware can be done by upgrading the core.

These benefits, however, do not come without cost: the designer here must be aware of the problems and pitfalls associated with a core-plus-extensions design. These include:

- The success of the overall system is highly dependent on a well-designed and well-implemented core: this approach can only amplify problems in the design of the core.
- Changes to the core, especially its interfaces, can necessitate changes in a large part of the system. This can make evolving the core itself very difficult and means it is very important to get the design of the core right the first time.
- Attempting to bundle the core functionality of a system inside a small set of classes can be artificial: the resultant classes may not be cohesive and the methods they provide may be quite complex.
- The resultant implementation may not be as efficient or direct as with another design. Extensions may very well need to interact with one another. A core-plus-extensions design forces these interactions to go through the classes of the core, thus adding several unnecessary levels of indirection.
- Because extensions interact with each other, the system is also prone to the *feature interaction problem*: while each extension or feature is considered independent, they actually interact with one another through the core in weird and wondrous ways whose results are not always predictable. This problem has been particularly vexing in the digital telephone-switching systems in which new features interact to cause problems with existing features.

In general, however, the benefits of a core-plus-extensions approach outweigh the drawbacks if a core can be naturally defined. When a system is large enough to require a sizeable team of programmers, it is worthwhile to try to cast the design into this framework by identifying core classes and then forcing all the interactions among the non-core classes to go through the core.

Defining the Core

The difficult part in a core-plus-extensions approach is determining what is in the core. This requires a careful analysis of the communications among the proposed classes. It requires modifying the interaction paths to minimize the interaction among potential extensions. It also requires some analysis of how the system might evolve in the future so that the design of the initial core can take these changes into account.

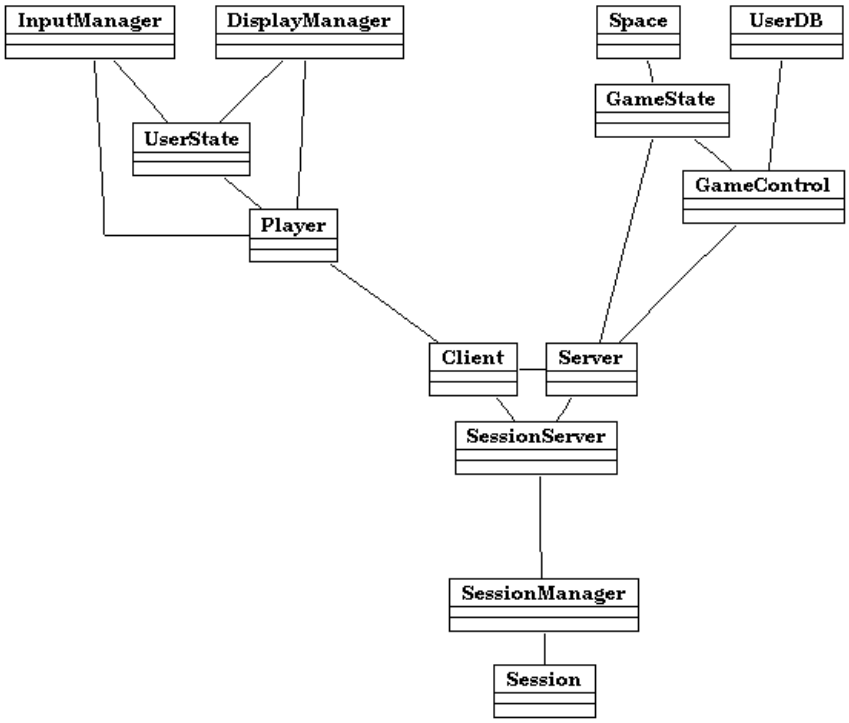

Figure 16-4 Core-plus-extensions overview of the spacewar program.

In designing a complex system, it is best to do a top-level system design with all the extensions and features one can possibly identify. This large design can then be pruned down to include only those features needed in the initial implementation. This approach tends to make the resultant system easier to modify and evolve, especially for previously identified functionality. This is helpful in a core-plus-extensions design where it is important to minimize the changes needed in the core as the system evolves.

While the spacewar program is a bit too simple to benefit greatly from a full core-plus-extensions approach, it can still serve as an example. At a trivial level, it already exhibits some of the characteristics of this approach. Because we have tacitly assumed that the client, server, and session manager are distinct entities, we can recast the design to use the communications classes as the core, as in Figure 16-4. Here the core is the three classes **Client**, **Server**, and **SessionServer**; the three extensions are the client program, the server program, and the session manager program. All interactions among these components must go through the core classes.

The design of the server for the game provides a more instructive example. Figure 16-5 shows a more detailed design for the spacewar server. Here we

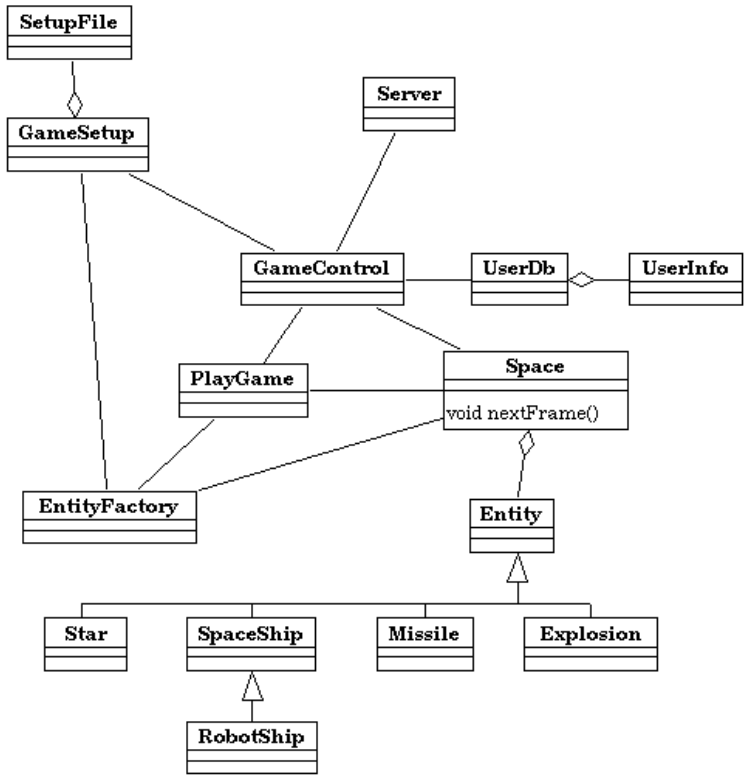

Figure 16-5 Detailed design of the spacewar server.

have added the entity hierarchy; a factory class for creating entities; a class for setting up a new game, possibly based on one or more setup files; a class to actually play the game; and classes containing the per-user information in the user database.

We can do a core-plus-extensions version of this design by focusing on the communications paths and identifying the central system components. A first approximation to such a design focuses on the classes **GameControl**, **Space**, and **EntityFactory** as the core. These form five extensions, one for the entities, one for the user database, one for playing the game, one for setting up the game, and one for the server. While this is a viable design, it has the drawback that the entity factory probably should be associated with the entity extension, not with the core. To get around this, we can add a method to the **Space** class to serve as an entry to the factory. We thus recast this design in the core-plus-extensions form shown in Figure 16-6. Here the core is the two classes **GameControl** and **Space** and the diagram shows that none of the five extensions communicates with anything other than the core.

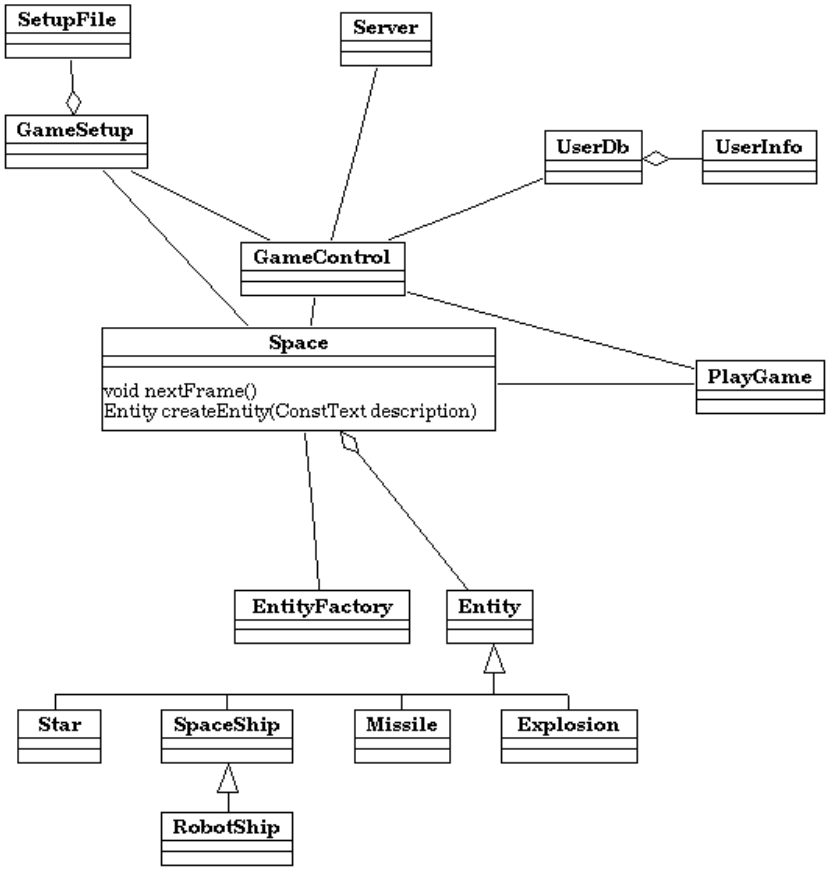

Figure 16-6 Core-plus-extensions version of spacewar server design.

One difficulty with this approach in this case is that the core must call the extensions; in particular, the game control module needs to invoke methods in the **GameSetup**, **Server**, **UserDb** and **PlayGame** classes. The best way to accomplish this is to view the calls from the core to the extensions as callbacks. The core should include a set of callback classes defining its interface to the extensions. These classes are designed to be inherited (as noted in "Inheritance for Callbacks" on page 123) and the callback methods are redefined by the extensions. The core can then be self-contained, making calls on the abstract callback objects as needed. Extensions can be plugged into the core by simply defining their implementation of the callback class.

Exactly how callbacks are used here depends on the situation. The easiest but least extensible approach is to define a separate callback class for each extension needing to be invoked from the core, so that there will be a single, well-defined object the core needs to use for each particular set of callbacks.

This simplifies both the core and the particular extension. It has the disadvantage, however, of forcing one to understand the extension fairly completely while developing the core: it is difficult to change the extension's interface or functionality or later to divide the extension into separate extensions. This approach should be used where an extension can be well defined early in the design process and the interface to that extension from the core is specialized and well understood.

An alternative is to use some sort of observer design pattern (see "Observer" on page 353). Here the core defines a generic callback interface for multiple extensions. For example, if the core maintains a data structure, it might provide a callback to inform arbitrary extensions whenever a portion of the structure changes. Alternatively, it might provide an interface to let extensions specify portions of the data structure they are interested in and then provide callbacks only when those portions change. This approach can support a wide range of different extensions using a single interface to the core. If it can be achieved, it can make developing both the core and the extensions easier and can yield a more robust and flexible system.

This alternative does, however, have its drawbacks. It is often difficult at best to identify the "right" generic callbacks in a given situation. A general mechanism like this can also be inefficient in having to do too much checking or making too many unnecessary callbacks, and can also lead to extensions that are over-complex or have unnatural designs. Still, where a generic callback facility can be identified, it is generally better to implement it in the core than to provide a set of more targeted interfaces.

INTERFACE DEFINITION

Once an object structure has been determined for the system, the next tasks are to separate the system into components and to define the interfaces between these components as precisely as possible. The number of components should be kept relatively small and the interfaces between them should be kept as simple as possible.

If a core-plus-extensions approach is used, the first relevant classes are the public classes of the core, including any abstract or callback classes representing functionality to be invoked in extensions from the core. In a more traditional approach, the relevant classes are those included in the top-level design. Note that the design of Figure 16-6 can be simplified into the traditional high-level design seen in Figure 16-7. Here we have changed the names for consistency with what follows: the standard prefix Space indicates the program; the class previously named **Space** is now **SpaceArena**, since "Space-Space" didn't sound good, and some of the other classes have shortened names as well.

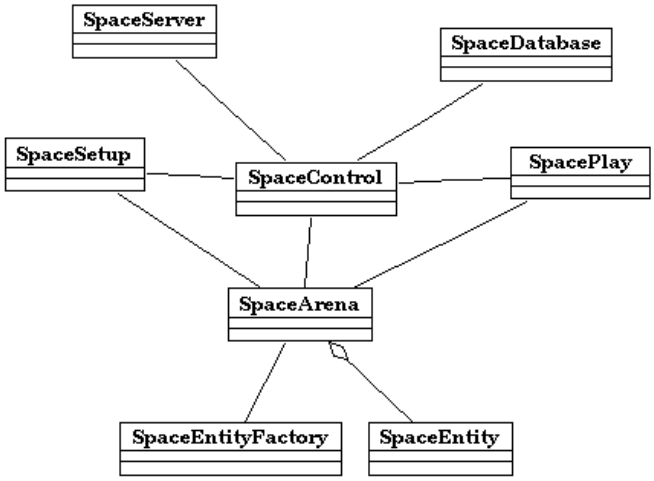

Figure 16-7 Top-level design of spacewar server.

Finding Initial Methods

In our previous designs we arrived at the set of methods by starting with a known method and adding methods as needed. This involved writing pseudocode for each method in turn to determine what information or functionality it needed from its or some other class. This approach is not feasible in a large-scale system, since it is important to determine the interface methods for the top-level design without having to fully specify how the components of this design actually work.

What is used here is an iterative approach. We first derive an initial set of interfaces by using high-level algorithms for the various components and intuition about what might be needed. Then, with this set of interfaces, the different components are designed in more detail. As these designs proceed, the interface is forced to change. Some components require additional information or functionality from the interface. Other components find it difficult to provide the desired functionality and propose instead to provide something close but not exact. Some of the interface functionality will be found to be superfluous. Negotiations among those responsible for the various components, moderated by those overseeing the whole project, then evolve the candidate interface into a stable and workable solution.

The first step in this process is to provide a good set of candidate interfaces. Here one looks at each class in turn and provides a detailed description of the class with respect to the overall design. From this description one can list the methods the class should provide as part of its interface. For example, the **SpaceControl** class could be described as:

> The SpaceControl class is responsible for the overall control of the game. It takes control when the server starts and waits until a user is ready to play. It manages the set of currently active users, adding or removing from this set as needed. It also manages when to play a round. It uses the SpaceSetup class to set up a new round and then uses the SpacePlay class to play the round. When a round is over, it sets up for a new round. It also handles requests from the clients to display the high scores.

This description implies aspects of the **SpacePlay**, **SpaceSetup**, and **SpaceDatabase** classes that are discussed below. It also indicates that we must be more specific about the interface between the client and the server, probably indicating the possible messages in terms of methods of the **SpaceServer** object. Finally, since it indicates that this class is invoked when the server starts, we should add a method to it:

```
void play();
```

that is called by the main program and sets everything up.

The communication between the client and the server is embodied in the **SpaceServer** class. The calls from the client to the server, obtained from a detailed understanding of the client and the workings of the overall system, include:

```
void newClient(SpaceClient id,ConstText username);
void removeClient(SpaceClient id);
void readyToPlay(SpaceClient id);
```

These should correspond to methods in the **SpaceControl** class that are invoked by the **SpaceServer** object. Note that clients are identified by an item of type **SpaceClient** that might be an integer or a structure (we can defer this decision until the actual implementation). The server should also provide communication back to the client. Here the messages include:

```
void highScores(...);
void objectPositions(...);
```

where the first returns the set of high scores for possible display by the client and the second returns the current game state as a set of objects and their positions. These we place as methods of **SpaceServer**.

We can continue with the **SpacePlay** class, whose description might be:

> The SpacePlay class is responsible for conducting a round of Spacewar. It assumes that the game has already been set up and that the SpaceArena class holds the proper entities at the proper locations. It cycles continually, possibly with some time delay (to cause the cycles to be at a constant time interval). In each cycle it uses the SpaceArena class methods to update the position of each object. It then must check for collisions among the objects, again using methods in the SpaceArena class. The SpacePlay class should check after each cycle whether the game is over (i.e. whether there is at most one ship left and no missiles). If an end of game is detected, it should set a timer for some number of seconds. When this time period elapses, it should determine if any ships are

> left in the system and, if so, should indicate that the corresponding player has won the round.

From this description we can define the one method needed in the abstract **SpacePlay** class:

```
virtual SpaceUser playRound(SpaceArena,SpaceServer);
```

This method returns the user who won the round (or `NULL` if there is no winner). Rather than assuming that the object knows of the arena object, we pass in the current **SpaceArena** object as a parameter. Similarly, we pass in the server object for it to call the `objectPositions` method as needed.

In addition, this description suggests the following methods for **SpaceArena**:

```
virtual void moveEntities();
virtual void handleCollisions();
virtual void reportPositions(SpaceServer);
virtual Boolean checkEndOfGame(SpaceUser&);
```

The last one here checks if there is at most one spaceship and no bullets currently active. If so, it returns `TRUE`; otherwise it returns `FALSE`. The parameter here is used to return the owner of the remaining ship.

A description of the **SpaceSetup** class might be:

> The SpaceSetup class is responsible for setting up the game board for a new game. It does this by first clearing the SpaceArena and then using the factory method this class provides to create the necessary entities, including ships of varying kinds (one for each user) and possibly robot ships and stars. It can get the game information from a file or from a static game or can generate it randomly.

This implies an abstract interface with the method:

```
void setupRound(SpaceArena,Integer numuser,SpaceUser * users);
```

where the arena to set up is passed in as well as the number and names of the users. The description also requires at least two new methods for **SpaceArena**:

```
void clear();
SpaceEntity createEntity(ConstText);
```

The latter method is actually the sole responsibility of the **SpaceEntityFactory** class and should also be a method in that class.

A brief description of the **SpaceDatabase** class might include:

> The SpaceDatabase class is responsible for maintaining the scores of all users of the system. When a round is over, the class should be told of all the participants and the winner. The class should also provide information about the people with the top scores so far.

This could be implemented by an interface offering the three methods:

```
SpaceUser findUser(ConstText name);
void recordRound(SpaceUser winner,Integer num,SpaceUser *);
Integer topScores(Integer max,SpaceUser *);
```

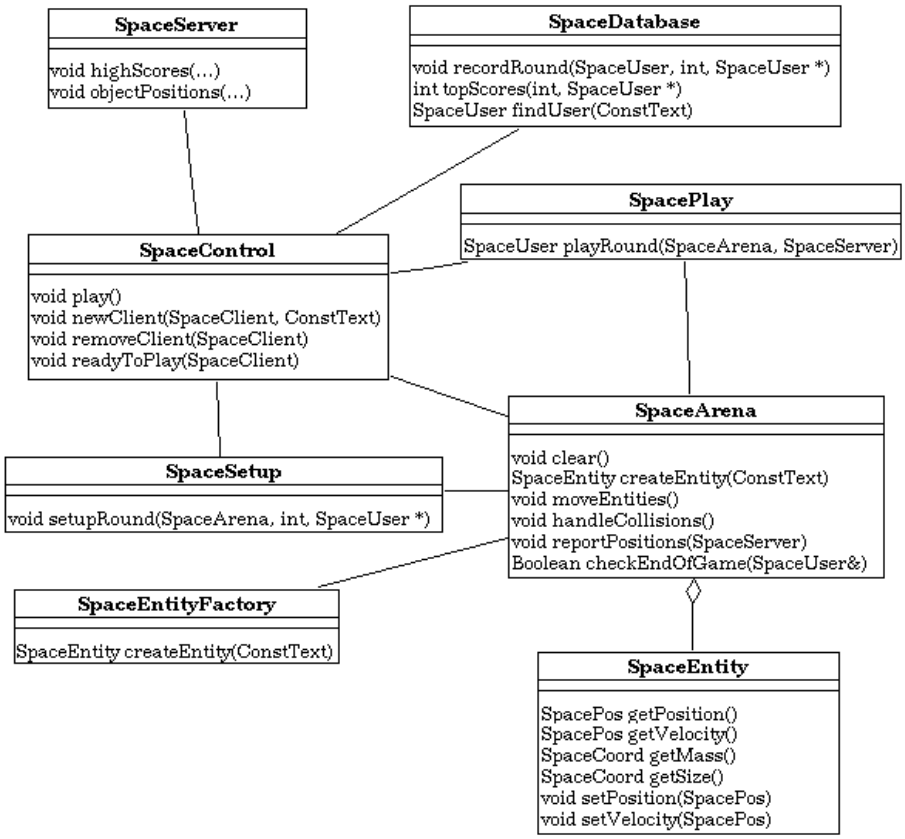

Figure 16-8 Interface definitions for the spacewar program.

where the scoring information is retained and thus returned within the **SpaceUser** class.

Finally, we consider the **SpaceArena** class. We know it must maintain the set of entities. It also must move the entities and check for collisions. This means it must be able to get the position, velocity, size, and mass of each of the entities; remove and add new entities (explosions); and change the position and velocity of the entities. We thus add the corresponding methods to the **SpaceEntity** class.

The result of all this can be seen in the initial interfaces in the static structure diagram in Figure 16-8. This interface is by no means complete: we still need to determine what the classes **SpaceUser** and **SpaceClient** contain and how they are accessed. We need to determine how the **SpaceControl** object gets handles to the other objects. We need to determine the contents of the `highScores` and `objectPositions` messages sent to the clients. Most importantly, we must evaluate the design to ensure it is both high-quality and workable.

Evolving the Interface

The next step in the design process is to take each class in turn and begin to design its implementation. This will help us resolve some of the design issues. For example, the design of the **SpaceServer** and **SpaceControl** classes will help us resolve what the **SpaceClient** object should consist of; the design of the **SpaceDatabase** class will help refine what is expected from a **SpaceUser** object, although the **SpaceServer** class may need to understand this as well to communicate information back to a client.

Furthermore, we will undoubtedly find additional methods needed in the interface. For example, the above description assumes that we can tell who the user is for a given spaceship, but that information is not currently available. Depending on how the **SpaceEntityFactory** and **SpaceEntity** classes are implemented, we may want to augment the `createEntity` method to take an optional **SpaceUser** and then augment the **SpaceEntity** class to return the current user. The **SpaceArena** class may also want to handle different types of entities separately. For example, it may want to compute gravity only between **Sun** objects and movable objects, and this information can be encoded in the mass or be available separately. The choices will depend on the design of the **SpaceArena** class.

The various classes are designed with the given interface in mind. They each define their own subclasses and their methods and data. Whatever changes they require in the interface must be resolved by the overall project team. While some changes, such as adding a data field to the entity to hold the corresponding user, are simple, others can be quite complex. Suppose, for example, that the designer of the **SpaceArena** class decides that each entity should be responsible for computing its own next position. Not only does this require additional methods in the interface, it also shifts a possibly significant part of the application burden to the **SpaceEntity** class, which might not be appreciated by the designers in charge of that class. Resolution of these conflicts requires both introspection by the designers and some arbitrators who have a good overall view of the project.

Finally, during the creation of the initial set of interfaces one must continually evaluate the design and be prepared to make it better. At this point, the goals of the design process are to ensure that one can achieve a working system, to minimize the inherent risks, and to make the remainder of design and implementation as simple as possible. To a large extent, this implies developing simple interface descriptions, and a reasonable question here is whether the proposed interfaces can be simplified in a meaningful way.

Because the interfaces drive the rest of the design and thus control the implementation, testing, and maintenance of the whole system, one must put great emphasis on this part of the development process. In essence:

> **A project lives or dies on its interfaces.**

OTHER DESIGN ISSUES

A number of other issues can become important in the design of a large, long-lived system. These issues are best understood and taken care of during system design even if their effects are not apparent until later in the development cycle.

Portability

The first such issue is portability. Portability can have multiple dimensions. Multiple platforms exist today, including Windows, UNIX, and the Macintosh. It might be desirable to have one's system work on all or some of these platforms. The effort involved in moving a system among these platforms can range from a small amount of work done only once for a variety of systems, to a huge amount of work for each system. The effort required is generally determined by whether or not the system was designed to be portable in the first place.

Portability is also an issue within a given platform. Programs written for Windows 3.1 can have difficulty running under Windows 98 or Windows NT. Programs written for SunOS 4.X on the Suns are generally not compatible with Solaris 2.X, even though both run on the same hardware and are considered UNIX. Operating systems and environments tend to change over time. While most strive to provide backward compatibility, this doesn't always happen and old programs often cannot take full advantage of any new features the system might provide.

Portability can also be an issue even with the same operating system on the same architecture, as the hardware options change. The most notable such change involves display architectures, although features such as the availability of a CD-ROM can also make a difference. A program designed solely for a black-and-white monitor is not going to look good on a color system. A program designed to make extensive use of 3D graphics is going to perform poorly on a system lacking the necessary support hardware. A program that can't take advantage of a large window will be frowned upon by users who have spent more to buy a large display.

In each of these cases, the difficulty of modifying the system to accommodate the new software or hardware is strongly dependent on how the system was originally designed. Better designs try to anticipate and then encapsulate potential portability issues. They try to arrange the system so that the changes needed to achieve portability are restricted to a small set of classes that can easily be replaced without affecting the rest of the system. This allows the programmer porting the software simply to rewrite or modify those classes while ignoring the bulk of the system.

For this type of simplification, one must try to anticipate how the system might be ported during its lifetime. Then, any functionality the designer

thinks might change in the future should be encapsulated within a class so as to provide an abstract interface to that functionality.

Consider what one must do to let a program handle multiple operating systems, say UNIX and Windows. The key here is to define a set of classes representing an abstraction of the operating-system functionality needed by the program and to use these classes exclusively throughout. Then these classes can be implemented separately for each operating system. To some extent, C++ has already done this for you, since its standard I/O library provides operating-system-independent input and output calls (although even here filename conventions may have to change from one system to the next). The **SimpleSocket** and **SimpleThread** classes we defined in Chapter 14 are other examples, encapsulating sockets and threads respectively. Other classes can be designed to handle other operating-system issues such as dates and times, shared memory, file names and permissions, and process execution.

A similar approach can be taken to support multiple display environments, whether these are display technologies on the same operating system or different approaches to windows in different operating systems. Here the system should be designed with an abstract interface to the display that has its own notion of windows, menus, dialogs and drawing commands. The remainder of the system is coded to this interface and then the interface itself is implemented separately for each different display environment. Some commercial systems provide this type of functionality in general terms. For many applications in which the display demands are not excessive, this design can even simplify the overall application. For example, a specialized interface can provide a method to draw a box of a certain style or to draw a menu given a simple tabular specification. For both of these, the interface might have to issue dozens of drawing commands to achieve the result, while, with the interface, the application itself has to issue only one method call.

Extensibility

This approach to portability can also be applied to other aspects of the design. Another issue arising in design is extensibility, the need eventually to add new features to a system. In general, any system that is used extensively will find new applications and users who want additional functionality. Moreover, in today's world of shrink-wrapped software, such new functionality must be added merely to remain competitive.

The best way to handle extensions is to try to anticipate in the initial design what or at least where changes might be needed in the future. When planning a system, one should plan it with all the possible bells and whistles imaginable. One can even do a top-level design incorporating all these extensions. The initial implementation, however, should then be simplified to include only what is necessary for the first version of the system. Designing a system larger than one needs to build makes any planned extensions much easier to add. If the extension is represented as a class with little or no initial

functionality, adding it to the overall system can entail merely adding to or replacing the designated class.

Design can also help with unplanned extensions. Designing a larger system forces the designer to use more abstraction and to make the system more generic, and this makes it more amenable to extension. A good core-plus-extensions design, one where the core is correctly identified and makes it easy to plug in additional functionality, can also make adding unplanned extensions easier. In general, however, extensions anticipated during the design phase are much easier to add than ones that were not, and we recommend:

Design a system with all possible extensions in mind.

This approach is taken to its logical conclusion by systems with a generic external interface to accept a wide range of plug-in extensions. Adobe *Photoshop*, for example, achieves much of its power and popularity by providing a standard interface to support arbitrary plug-ins. It is often difficult to identify such an all-purpose interface and even harder to define it to accommodate all possible extensions. However, if this approach can be used, it generally produces a much more powerful and flexible design that is easy to evolve.

Existing Frameworks

A third issue that can be handled by encapsulation in the design phase and can profoundly influence system design is the use of existing frameworks. Existing frameworks can arise in a variety of ways that must be anticipated throughout the design process.

Consider the design of a user interface. In developing a program for Windows, one wants its user interface to conform to the conventions and standards used by other Windows applications. Similarly, in developing for the Macintosh, one wants the application to look and act like a standard Macintosh application. This use of existing conventions affects the design and implementation of the user interface. It constrains the designer's options in setting up window formats and menus and might also make the designer add new options and buttons. Windows, for example, assumes a view and document model in which all applications can save and load their views from a file, in which files are typed, and in which the File menu offers the set of recently used files as options. The designer must take all this into account in developing both the user interface and the overall structure of the application.

Library packages are another example of the use of existing frameworks. The current C++ standard includes the template library we have described as well as many functions from the standard C library. These libraries, however, cover only some of the necessary functionality. A large application might want to use a library for 3D graphics, libraries for statistical computations, libraries

for graph drawing, etc. The choice of such libraries can significantly affect the design of the system.

Consider, for example, an application that needs 3D graphics. Various libraries are available to accomplish this, each with its own strengths and weaknesses. OpenGL is a C-based library now available on a broad range of different platforms. OpenInventor, a C++ library built on top of OpenGL, offers much additional functionality and is generally easier to use; however, it is a bit slower, more geared to modeling than interactive applications, and is not freely available on all platforms. DirectX is a high-performance library directed toward game applications available only on the Windows platform. XGL is Sun's low-level library. The choice among these alternatives will affect the design and implementation of the application. For example, if OpenInventor is used, the application might be developed in part by subclassing from the OpenInventor classes. If the application will use OpenGL and have any type of animation, it must implement its own main drawing loop to refresh the screen.

If the application is not too complex, it might be possible to write a wrapper class embodying the 3D functionality the application uses, code the rest of the application so that it uses only this class, and then implement this class using any of the target packages. A more sophisticated use of 3D graphics, however, would probably make this difficult or impossible, since the definition and drawing would pervade the system. Here one must evaluate the different libraries early on, choose one, and then take the conventions and functionality of this library into account throughout the design.

Libraries are one type of functionality available to an application. Other functionality is available through separate systems and interfaces. For example, a database server is generally available on most platforms. This is a separate application to manage relational (or other) databases. Generally, the server provides an interface through which the application can send SQL-based commands to access or modify the database. (SQL is a standard high-level database query language.) If such a server is available, one can simplify the design and implementation of the overall system by using it rather than implementing one's own database system.

Just as with libraries, however, care must be taken in using an external system. Different systems have different interfaces, different functionalities, and different availabilities. Designers can commit to a single system and design for its features. Alternatively, they can attempt to find a common subset of features and restrict the design to using these, so that it would be possible to move from one external system to another. A third choice would be to use encapsulation to hide the external system with an interface class.

Finally, a new system is generally designed not in isolation, but to cooperate with existing tools and systems. This can be as simple as sharing files or as complex as actively communicating with existing systems. In the past few years a number of somewhat standard frameworks have evolved to support such communications, including DCOM and ActiveX under Windows and

CORBA and the Common Desktop Environment under UNIX. If a system must send commands to other systems or handle requests from other systems dynamically using one of these frameworks, then it must be designed to make this possible. This interoperation requirement will have an impact throughout the design process and must be taken into account early in design.

MANAGING A SOFTWARE PROJECT

As noted in the previous chapter, a significant part of software engineering concerns techniques and skills for managing a software project. While most of this work has been geared to large projects with tens or hundreds of software developers attempting to work together, some of it is applicable to smaller projects with ten or fewer programmers. In this section we briefly cover some of the issues arising in these smaller projects, with an emphasis on management techniques for student projects.

Personnel Management

The primary reason that programmers working as a team are not as productive as programmers working alone is the need for communication. Programmers all want to have input on the design, and they all need to convey their interfaces to those who need them. As the system is coded and the interfaces change, the programmers must understand and negotiate on the changes. As bugs are found, they need to be attributed to one or another programmer's components so that they can be found and fixed. Misunderstandings among programmers regarding functionality or interfaces must be resolved, and ideas for extensions or changes must be discussed.

The time spent on communication increases with the number of programmers involved. Moreover, once there are more than around four developers, design and organization are effectively being managed by committee, since most groups attempt to reach a consensus and spend a lot of time doing so. As the team size reaches ten or so, however, design by committee no longer yields a useful result, and the time spent in meetings and waiting for other programmers significantly reduces everyone's productivity. This is reflected in the number of potential communication paths among programmers. In a four-person project, there are only six combinations; in a ten-person project, there are forty-five paths, as illustrated at the top of Figure 16-9.

A common solution here is to create a personnel hierarchy. An organized *chief programmer team* or *programmer team* generally consists of a project manager or chief programmer, a project librarian, and a core of programmers. The communications paths here are only between the project manager and librarian and the rest of the programmers, as shown at the bottom of

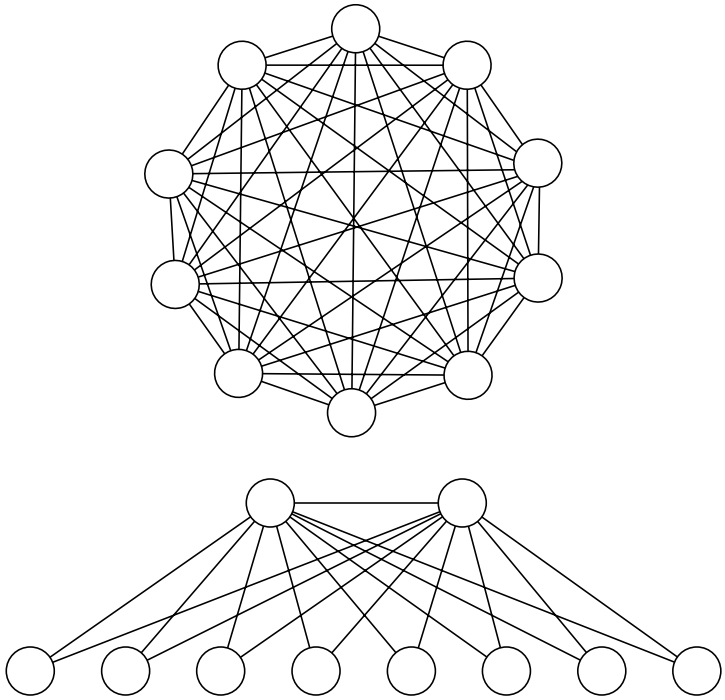

Figure 16-9 Communication paths in a ten-person project.

Figure 16-9. The programmers do not, in theory at least, need to communicate among themselves.

The programmer team is run as a benevolent dictatorship by the project manager. This provides a single focal point for design decisions and arbitrating among the different programmers when changes are required. It also lets one person define the overall program schedule, assign different programmers to work on the parts for which they are best suited, and ensure a better project.

The most experienced or most advanced programmer in the team is usually chosen as project manager. She is the one who must have a good overall understanding of the whole system. She must be able to assess the complexity of the different components so as to assign the proper personnel to that section and determine how long each section should take to get working. She is the one who assumes overall responsibility for the whole project. Whether the project manager does any actual coding depends on the size and complexity of the project. In a student project, the manager typically gets involved with some of the more important aspects of the system but has less code responsibility than the other programmers.

The project librarian acts as the repository of all the information about the system under development. He is in charge of maintaining system documentation and of ensuring that changes made to the design by one person are communicated to all the others. Any changes needed in an interface must be cleared by the project librarian: he ensures that the change is adequately documented, that appropriate changes are made to the design and specifications documents, and that the change is communicated to the remaining programmers. The librarian should be able to answer any questions on any aspect of the system by using the accumulated and maintained documentation. The librarian may or may not also do programming, again depending on the size and complexity of the project. If he has programming and design tasks, they are typically smaller to allow him time for his other duties.

The remainder of the programmer team, the programmers, are each given responsibility for one or more compartmented pieces of the system. They should design and code their portions so that they conform to the agreed-upon interface specifications. They should test their portions as much as possible before attempting to integrate them into the system. They should also be available to make prompt fixes to problems in their code as others start using it in the integrated system.

Good team programmers know how to fit into a group. They understand the costs involved in changing interfaces and the need for conformity to group standards and procedures. They provide input on proposed interfaces as needed, understand their component of the system, and attempt to make their interfaces both fair and functional. Most importantly, they code accurately and defensively and get their parts of the system done on time.

Design Management

The design of a larger software system is generally broken into two parts. The top-level design, where an overview and component breakdown of the system is achieved, is typically done by a few designers (possibly only one) who have a good overall understanding of the problem at hand and can evaluate potential solutions. The detailed design of each of the components is typically done by the programmer who is to implement that component. Along the way, both designs must be checked and be given room to evolve.

The best way to check a design is to present it to a team of critical reviewers whose job is to find the potential problems. This is called a *design review*. A typical design review starts when the designer hands out a complete design document to the reviewers in advance, so they can get a good overall sense of the design. At the actual review, the designer presents the design, emphasizing why critical design decisions were made, and then illustrates the design by showing how it will work under various circumstances that represent both normal operation and unusual conditions.

A successful design review identifies actual or potential problems with the design at an early stage, when they can be corrected easily and cheaply. The

job of the panel of reviewers is thus to question the design, attempting to find such problems. They should identify possible alternatives and ask the presenter to justify the design presented in their light. They should attempt to bring up scenarios for which the design may fail. When the presenter gives a scenario, the reviewers should be thinking of "what if" questions to identify circumstances under which the design may fail. Finally, the reviewers should be attempting to determine if the design proposed is as simple as it can be.

The design review process is also a constructive one. As design problems are diagnosed, it is the task of both the presenter and the reviewers to identify potential solutions to these problems. Where the design is overly complex, simpler alternatives should be proposed and evaluated.

Design reviews are one critical component in managing the design aspects of a larger system. The other component is managing the interface specifications among the different programmers. This is typically the job of the project manager, often in conjunction with the project librarian, but can also be discussed in group meetings. As designs of the individual components are completed and reviewed, the interfaces these components provide or the interfaces they use often must be changed. When this happens, everyone involved with those interfaces must be notified of the change.

Managing the evolving interfaces is difficult because of the care necessary to maintain the overall integrity of the design. While one minor change probably will not hurt the design, a large number of such changes actually might. Changes tend to make the design more complex and to create additional dependencies among the components, and they also tend to create components that are no longer balanced. This can make future modifications and evolution of the system more difficult. It is the task of all the designers, and the project manager in particular, to evaluate the proposed interface changes in the light of the overall design and to propose alternatives where appropriate.

Code Management

Once the design has been completed, coding can begin. Managing the coding of a large project is probably more complex than managing the design. Code management requires developing an implementation plan, evaluating or reviewing code, ensuring that code meets project standards, and providing appropriate strategies for integration and system testing.

The first task here is to develop an implementation plan. This plan should designate who is in charge of each component and the deadline for its completion. The resultant schedule should take into account dependencies among the components so that a component can be tested and used when it is completed. For example, in a core-plus-extensions design, it is a good idea to complete the core before the extensions. This may involve putting the faster programmers on the core or splitting the core into subcomponents so that it can be developed by multiple programmers before they go off and work on the extension components for which they are responsible.

The best way of achieving a reasonable implementation plan is to identify *milestones*, dates by which a particular piece of functionality of the overall system must be achieved. For example, one might state: "By the last Thursday of this month, components A, B, and C will be tested individually and ready for integration into a base system." Such milestones, especially ones involving the whole or a substantial part of the team, provide incentives (in the form of achievements and peer pressure) for the different team members to get things done on time. They let the different programmers work at their own pace, taking their other commitments into account as necessary, and knowing when a task needs to be accomplished. They also provide the project manager with a whip of sorts to keep the team members in line and to identify how far behind the project might be. Because of this we recommend:

Use milestones to guide a project's implementation.

The overall implementation plan should also take into account the unforeseen. Software development typically takes longer than one anticipates. This is especially true in a multiple-person project in which the actions of the different team members are somewhat independent. People will get sick, have outside commitments, or just sleep through meetings. Some parts of the system turn out to be more difficult than anticipated and some turn out to be easier (but try to get the programmers in charge of the latter to admit it). The operating system will crash or the hardware will be unavailable on a critical date. A reasonable schedule will try to take some of these factors into account by introducing some slack into the schedule and revising it as needed.

The second task in code management for a larger software system is to ensure consistency among the different designs and implementations. To some extent this involves developing and enforcing a common set of guidelines that are contained in comprehensive coding standards to be followed by all members of the team. The coding standards should cover code presentation including formatting and inline documentation conventions, naming conventions, as well as guidelines on parameter order, use of inheritance, callback methods, use of libraries, etc. A simple such set is shown in the course standards shown in Appendix A. The more the code developed by the different programmers can be made to look and feel the same, the easier it will be for them to read and understand others' code to facilitate debugging and testing, and the fewer problems there will be in integrating the different aspects.

One way to check this and to test code without a fully integrated system is to undertake selective *code reviews*. A code review is the implementation analog of a design review. Here the project manager selects one or more portions of the code that are complex enough or interrelated enough that problems are likely in the code or in how people use it. The programmers responsible for these portions then produce a set of handouts, generally the code listing along with the design information describing what the code should be doing, and

distribute them to a panel of reviewers before the code review. In the actual code review, the programmer goes over the code in front of the reviewers, justifying why it is written as it is and demonstrating that it works. The reviewers' job, again as in a design review, is to find faults in the code so that they can be corrected early in the development process.

Code reviews can take two forms. The programmer can either simply go through the code line by line to ensure that all the reviewers understand the purpose of each line, or can take one or more scenarios and run the code on paper through those scenarios. In either case, it is the reviewers' job to identify other scenarios under which the code might fail and to indicate any potential problems with the code and ways in which it might be simplified. The review should also check that the programmer followed the coding standards adopted in the project and that other components needing to use or be used by the code integrate correctly.

Configuration Management

Another aspect of code management is managing the files of the project. This is the task of *configuration management*. There are two aspects to this task. *Version management* involves organizing, controlling, and sharing the source files among the programmers. *System modeling* entails directing how the source files are used to build the resultant system.

The first step in configuration management is to organize the project. This is generally done by identifying the various components of the overall project and putting them in a hierarchical structure. This is often represented first as a directory structure, with the top-level directory representing the project and its subdirectories representing the various components. Each subdirectory can be further subdivided either to separate source and binary files or to identify different subprojects. Note that for projects of the size we are addressing, one level of hierarchy is generally sufficient. The different directories provide a context for the corresponding component, supporting both version management and system modeling for that component.

Version management is supported by a variety of tools. The older tools, such as *sccs* and *rcs* on UNIX, assume that the single copy of the source is to be shared among the different programmers. Here a programmer *checks out* a source file, i.e. requests the right to be the current owner of that file so as to make changes to it. While this programmer has it checked out, no other user can modify it. When he or she is finished with the modifications, the file is *checked in* and other programmers can again access it. This works in small projects in which each component is almost the exclusive domain of a single programmer.

Newer version-management systems take a different approach in which each programmer has a full copy of the system to work on. Programmers can make changes to whatever files they want. When they go to commit or check in their changes, the system attempts to resolve any conflicts or asks the pro-

grammer to do so. This approach allows more cooperation among programmers, but also requires more coordination to avoid conflicts where two programmers make incompatible changes to the same piece of code.

Either form of version management accomplishes the primary purpose: letting multiple programmers work on a common set of files without stepping on each other's toes too often. Version management offers the developer other benefits as well. Version systems keep track of all past versions of the software, letting programmers easily go back to a working version if a set of changes they are trying does not work out. The systems support experimental development by letting multiple versions be developed in parallel. Finally, the systems support maintenance by letting the developer maintain a copy of the software as it was provided to the users so that user problems can be debugged.

Many tools support system modeling as well. In the UNIX environment, the common tool is a form of the *make* utility originally developed at Bell Laboratories. Here the system model is given as a text file listing the dependencies between the generated and source files and providing the necessary rules and commands for creating the generated files. *Make* lets the programmer define a variety of different targets in a single directory. It both supports the construction of arbitrary types of targets and lets the programmer define new commands such as "`make clean`" to remove all object files or "`make print`" to get an ordered listing of all the source files.

Modern environments try to integrate *make*-like facilities into the environment. Microsoft *Visual C++*, for example, uses the notion of a project that is targeted at building a single binary or library. The user merely tells the system (via a series of dialog boxes) what source files and libraries compose the project, and the environment then defines the appropriate system model for building the project.

System-modeling tools are very convenient because they provide a simple way of defining all the options and conventions that go into constructing a complex piece of software. They let the programmer easily change the compiler flags or libraries for a system. They also have the intelligence to recompile only those portions of the system that have been modified, allowing quicker turnaround on small changes.

Testing Management

The most complex and time-consuming part of a multiple person project is integrating the components written by separate programmers into a single working system. No matter how much work has been put into development, the individual pieces of the system will not fit together perfectly and bug-free. Finding and fixing the problems can become a frustrating and costly team effort: the cause of the problems is often difficult to pin down to one particular component, and (for a large project at least) the person whose code actually contains a particular bug is unlikely to be present to fix it when it arises.

The best way to deal with these problems is to avoid them as much as possible. This is where solid, well-understood interfaces, design reviews, code reviews, module testing, and all the other topics cited throughout this text come in. One of the more important aspects of group software, however, is defensive programming. The individual's goal during integration testing should be to prove that his or her code is not at fault when a problem arises. Programmers want to demonstrate that a problem, even if exhibited in their code, was actually caused by someone else. The easiest way to do this is to put lots of defensive code in those portions of the program dealing with the interfaces to other components. This was covered in depth in "Defensive Programming" on page 191, and has been emphasized throughout the text.

Defensive code should generally remain in the system even after it is completed. Such code will be very useful in tracking down errors that occur while the system is being used and in identifying problems as the system evolves. Using assertions is okay if the error is fatal and would cause a crash anyway, but the error message generated by an assertion is generally meaningless to the user. Print statements are usually worse: their output tends either to be a annoyance to the user or to be ignored. However, for an error condition that can easily be recovered from, they might be the preferred alternative. In general, however, exceptions give the best protection. They can return the system to a known state and possibly recover from whatever error caused the problem. At worst, they can trigger an automatic save and a clean abort so the user doesn't lose any work. Exceptions can also be easily changed to have different behaviors while the system is being tested (where they abort the system) than when the system is being used in production (where they provide error recovery and possibly send information about the error back to the developers).

Another helpful technique here is for programmers to maintain three versions of their components. The first version is the *working* version. This is the code the programmer is currently editing; it can change frequently and no other programmer is immediately dependent on it. The second version is the *experimental* version. When the programmer feels the working version is stable, i.e. is a clean build that incorporates the latest set of changes and has passed module testing, it can be upgraded to experimental. The experimental version is meant to be used by other programmers who need the component and contains code that should be better than before but has not been fully tested. As other components try to use it, it will become better tested and more stable.

When all the interested components have verified that an experimental version works for them, the version is upgraded to a *stable* version. The stable version is a non-changing implementation of the component. Other components are free to use it with the knowledge that the functionality there is stable and working, and that any bugs or features there will remain. This isolates them from the local changes to the module and from experimental changes that have not been completely tested.

This approach should be augmented with communications whereby all programmers are notified when new versions of components become available. It can also be synchronized by using milestones. For instance, a particular milestone might ensure that all components in experimental form have been bound together and passed some set of tests. When this occurs, all the experimental versions are converted into stable versions at one time and further development proceeds with the construction of new experimental versions.

Another approach to aid in system testing is to attempt to develop mini-versions of the system as milestones. Here one creates a very simple version of the system with only limited functionality early in the development cycle — a shell containing only the user interfaces, for example. This version of the system is then slowly augmented with additional functionality one step at a time. At each step, if the old system worked and the new one fails, the cause of the failure lies in the newly added code, either directly or indirectly (through using some feature of the previous code that hadn't been used before). Note that this approach not only aids testing but also gives the developers a sense of accomplishment in actually having a running system throughout most of the development process. This tends to increase programmer motivation and get the overall system completed more quickly.

While individual programmers are responsible for testing their own components, responsibility for system testing must lie with the whole team. To organize this, however, one of the team members, generally the project librarian, should be in charge of collecting and documenting system test cases. A set of standard test cases should check both the normal operations and error checking of the system, and these should be applied whenever a new version of the system is built. This is *regression testing*, as discussed in Chapter 8. When possible, it is often helpful to designate one person in the group as the "tester" whose job is to find successful test cases, i.e. those causing the system to fail. As noted, it is much easier to test other people's code or at least code in whose correctness one doesn't have a stake.

Documentation

Keeping a large project on time and organized depends, to a large extent, on documentation. Moreover, if the project is going to continue to be used and evolve, documentation will play a key role. Writing a software system involves not just writing the code but also writing all the documentation that goes along with the code. We thus note:

Without documentation, a system is incomplete and useless.

System documentation serves a variety of purposes. During development, its most important use is as a reference for the programmers. Programmers should be able to look up details of an interface they need to use, browse

through the available system libraries to find the method or classes required, understand what exactly is expected from their component, see why a certain feature is designed or implemented as it is from a specifications point of view, understand what the user wants out of the system, and know the current state of development and the current schedule. Once development is complete, documentation is even more important. Maintenance programmers need to know all the above information. In addition, since the maintainers are often a completely different set of programmers, they need information on the motivations and reasoning that went into the actual system design by the original developers.

In order to achieve these goals, the documentation must be accurate, complete, and up to date. If these criteria are not met, the documentation will be useless. If the documentation is not accurate, the programmers and developers learn it cannot be relied upon and resort to reading the appropriate code and header files instead. If the documentation is not complete, programmers either become frustrated with the missing information or simply do not use the undocumented features. If the documentation is not up to date, it will be considered inaccurate and again won't be used. Worse, it might be relied upon and cause misunderstandings and errors to be inserted into the system.

It is important therefore to maintain an up-to-date and complete repository of information about the system. This can be organized in a variety of ways, either electronically or in notebooks, and should contain at least the following information:

- *Requirements documents* stating what the users originally wanted. As additional input is obtained from the prospective users, these documents should be updated accordingly.
- *Specifications documents* describing the user interfaces and commands and what the system will do. As the requirements change or as the design imposes additional constraints or provides additional functionality, these should be updated.
- *User-interface documentation* defining the interfaces, how they look, and what they do. This can be derived originally from the specifications, but should be replaced with the design diagrams showing the user interface. This documentation should be updated as the user interface evolves, and should eventually be replaced by one or more user manuals for it.
- *Design documentation* describing the top-level design and providing a breakdown into components. This should include the top-level static structure diagram and its explanations as well as a detailed description of the purpose and function of each component. It should be kept current, with both the diagram and the descriptions changing as the system evolves.
- *Interface specifications* that precisely define the top-level interfaces provided by the components. These should provide both the syntax (the call-

ing sequence with parameters and return types) and the semantics of both normal and error operation of all the public methods the various top-level components provide. Such documentation should be easy to access and search and must be kept up to date as the interfaces evolve.

- *Design documentation* for each component, showing the detailed design of that component. This includes static structure diagrams, descriptions, message-trace diagrams, and any other information generated by the individual programmers as they develop the various components. The individual programmers should maintain this documentation and keep it up to date.
- *Code documentation* contained in the source files detailing what the code actually does. Comments should be used wherever a reader might not immediately understand the function or workings of a particular piece of code. Assertions and suitably noted defensive checks should also be thought of as code documentation and should be broadly used. These are particularly valuable since they become part of the code and cannot get out of date.
- *Test-case documentation* describing the various system tests a new version of the system must pass before it can be accepted. This should include a description of each test case, how to run it, and what the expected output is. If the program can be tested mechanically (this is difficult for a program with a graphics interface), the test cases should be automated with an appropriate shell script or testing program. Otherwise, the documentation should be sufficient for someone to sit down with the system, run it according to directions, and check off the success or failure of each test case.
- *Current status and plans* describing the overall project. These should include all the project milestones, the expected start and end date for each top-level component, and a schedule showing when and how the various components will be integrated to form a working system. Ideally this should show a series of system versions, starting with a user-interface shell and ending with the completed system.

Another sort of information that can be helpful to both the individual programmers and the project manager is the time spent on the project by each programmer. This can be kept as time logs in which programmers log how much time they spend on what task each time they work on the system. These time logs help programmers see how to better use their time and let the manager more evenly distribute the workload of the overall development effort.

The overall responsibility for maintaining project documentation lies with the project librarian in conjunction with the rest of the project team. The librarian provides the means for organizing and accessing the documentation, but the actual documentation must be written and maintained by those most closely involved with the particular designs, interfaces, and code. Before

accepting code to integrate into the rest of the system or letting a programmer install a new stable or experimental library, the librarian (with the assistance of the project manager) should insist on revised documentation to match the changes. Before code is written, the librarian should receive detailed design documentation from the programmer. Before an interface can be changed, the documentation for that change must be submitted and circulated to the affected programmers.

Finally, we emphasize once more that the code itself is a form of documentation. It is the source of last resort and the only thing guaranteed accurately to describe the actual workings of the system. Even when the documentation is complete and accurate, the code is still used to address fine details and must be understood on a line-by-line basis when making changes or debugging. As such, it is essential (as we have noted throughout this text) to:

Write your code to be read by others.

CONCLUSION

The problems involved in designing large systems and the benefits of the various solutions can only be truly appreciated through experience. Similarly, the complexity and challenge of software design can only be understood by actually doing many designs for many different systems.

We have attempted to show how to go about building a software system. While we have concentrated on design, we have also covered both high-level issues of software engineering and a lot of low-level issues involving how the actual code gets written. However, even the best text is no substitute for experience. Many of the methods and strategies discussed here can be appreciated only after you have suffered through the consequences of not using them. Many of the statements about what works and what doesn't will not be compelling until you try different approaches and experience the results yourself.

Building a large software system is generally a valuable and rewarding experience. There is a thrill in demonstrating a system you or your team created that now performs some useful purpose and that other people might actually want to use. While the work can be great, the rewards are even greater. Thus:

Design and build a software system.

SUMMARY

Software complexity tends to grow much faster than software size. Thus large software systems are much more difficult to design, implement, test, debug, and maintain than smaller systems.

The first step in building a large software system is to understand thoroughly what it is you want to build. This is where requirements and specifications, discussed primarily in the previous chapter, come into play. Once this is accomplished, design can proceed. In addition to the standard design techniques covered previously, we introduced four techniques aimed specifically at large-scale software development. These were:

- *Clustering*: Here the designer identifies a rich set of candidate classes and then forms clusters of these classes on the basis of the relationships among them. The clustering lets the designer remove classes from the candidate set and eventually settle on five to 10 top-level classes.
- *Design by subsystem*: Here the designer groups related classes together into subsystems, creating a single new class (a facade) as the interface to that subsystem.
- *Risk-based design*: Here the designer attacks a problem by identifying the difficulties that are potential risks to a successful system. Each of these problems is then either solved or isolated, leading to a less risky design that is more likely to work.
- *Core-plus-extensions design*: This design scheme organizes a large system so communication paths are minimized, it is easier for multiple programmers to interact, and the system is easier to change over time. A core set of classes is identified as the heart of the system. All other classes and subsystems are designed as extensions that communicate only with the core.

Other factors come into play in designing a large-scale system, including portability, extensibility, and interaction with existing systems and frameworks. These need to be understood as part of the specifications and taken into account throughout the design process.

These design techniques are generally applied to develop a top-level design for the proposed system. This design is then reflected in a set of interfaces defining how the various classes interact with one another. The key to a successful large-scale design is that its interfaces are well thought out, well developed, simple, and complete. A project lives or dies on its interfaces.

Finally, while a good design is necessary for a project, appropriate management techniques are required to convert the design successfully into an actual system. These include developing an appropriate personnel hierarchy; using design and code reviews to insure a workable and high-quality system; developing a project timetable with appropriate milestones; using appropriate tools

for configuration management, coding, testing, and debugging; and, perhaps most important of all, documenting what is being done as it is done.

EXERCISES

16.1 Finish the top-level design for the spacewar program.

16.2 Suppose you have a four-person team to develop the spacewar program. Devise a timetable with milestones that assigns these people to components and specifies the order in which the components should be built. If you had six people, how would your plans change?

16.3 One of the best ways of learning about something is trying to develop a system to implement it. Suppose you are developing a program to facilitate defining, organizing, and accessing program documentation.
a) Draw up a list of requirements for such a system.
b) Draw up specifications for the requirements.
c) Draw up a top-level design meeting the specifications.

16.4 Design an XML browser that can read and display XML documents and follow links to other documents. Look into what libraries are available to build such a browser as part of the design process.

16.5 Implement the spacewar program designed in this chapter. What changes in the design were required during the implementation and testing of your system?

16.6 Fully design and implement the solar system example described in Chapter 1. Once the system is working, augment it with a graphical user interface. Be sure that the original system design can accommodate such an interface.

16.7 Design and implement the garden designer of Exercise 15.7.

16.8 Design and implement the race-car system of Exercise 15.8.

16.9 Design and implement the menu-planning system of Exercise 15.9.