

Chapter 15

Software Engineering

So far we have talked in terms of one programmer designing an already specified system. Software development in real life is seldom like this. Software is typically developed by programmers for customers who do not know precisely what they want. Programmers must understand the customer's needs and then define exactly what the software should do before they can do any actual design. Software development is also typically done by teams of programmers, ranging from a small group of two to four people to a large team involving hundreds of people. Testing and deployment of software is also a major effort, often done by yet another group. Moreover, most of the effort put into a software system, especially a successful one, is spent fixing bugs and adding new features once the system has been "completed," rather than on the system's initial development.

The study of software development, from a crude need on the part of a customer, through design and coding of a software system, and on through the ever-continuing process of fixing and improving the software, is called *software engineering*. This field has as its goal making software development simpler and the resultant software better. This chapter provides a brief overview of software engineering, describing the terminology and basic ideas developed by researchers and practitioners over the past thirty years. We first consider the problems that led to the field in the first place, proceed to describe the various phases of software development, and finally discuss the ways in which these parts can be put together.

THE FOUNDATIONS OF SOFTWARE ENGINEERING

In the early days of computers, software was often ignored. In the 1950s, system development concentrated on the hardware, since software contributed only about 20% of the overall system cost. Software was typically small (as were the machines) and hand-coded in assembly language to make the most efficient use of the machine possible. By the mid-1980s, however, the hardware-software ratio had inverted and software accounted for 80% of the system cost. Today, with inexpensive machines and personal computers already

on people's desks, software is even more dominant, making up 95-100% of the cost of a system.

Software takes an increasing share of the system cost not only because hardware has gotten cheaper, but also because software has gotten larger. A large software system in the past was thousands of lines of code. Today's systems are generally millions of lines long. As software grows, its cost and complexity grow faster. Just as writing a thousand-line program is more than ten times as difficult as writing a hundred-line one, writing a million-line program is much more than ten times as difficult as writing one with one hundred thousand lines. The techniques used in developing small-scale systems do not scale up, and new techniques and methodologies are required.

The Problems of Software

The result of the software explosion has been a set of now-predictable problems. Software has become notorious for these difficulties and whole industries have arisen to alleviate them. Some of the problems, no doubt familiar to the reader, include:

- *Software is late.* Software is often advertised as available on a given date and then is delayed by months or even years. No company is really immune to this, and numerous examples can be found from such well-known manufacturers as Microsoft, Sun, and IBM.
- *Software is expensive.* Companies must estimate how much it will cost to develop a piece of software. These estimates are notoriously low: software always costs more than the original estimate, often by factors of two or more. This comes in part from the personnel costs in software delays, but other factors come into play as well.
- *Software doesn't do what it is supposed to.* Systems are hyped as solving all one's ills and never do. Even well specified, narrowly targeted systems often do not meet the expectations of the potential users.
- *Software is unmaintainable.* Software maintenance entails fixing bugs and evolving the software to meet new user or system demands. Complex systems tend to be extensible up to a certain point, after which either they completely fall apart or programmers refuse to touch them for fear they will do so. The standard solution is to rewrite the ancient software completely rather than attempting to modify it.
- *Software is not understandable.* This should be taken in two ways. We have been emphasizing the importance of making software readable by others because most software written today can be understood only by its author, and then only within a few months of writing it. This, of course, contributes to making software difficult to maintain. Another way of viewing this is that the documentation accompanying the code, which is essential and should be considered part of the software, is

rarely helpful, often unreadable, sometimes nonexistent, and at times just plain wrong. Both design documentation and user documentation must be kept up to date and accurate, and should also be organized so as to present the necessary information in a usable form.

- *Software is unreliable.* Bugs, bugs, bugs. Anyone who has used software is aware of bugs. We are told to save often in the editor, to avoid this or that feature, and we know that our program or machine freezes or crashes periodically. Today's software is so complex and so poorly understood that it can't help but have some problems. Moreover, testing methods are rarely adequate; indeed, many companies resort to letting the user do the testing.
- *Software is inefficient.* Efficiency can be measured in various dimensions — run time, user-interaction time, memory utilization, and disk utilization. Each time we get a new version of a complex piece of software we need to upgrade our machine to run it, since while the old version might have been designed to run on a smaller machine, the new one does not run well enough to be used there.
- *Software is too complex.* This can again be taken at two levels. The inherent complexity of today's systems, requiring coding and maintaining millions of lines of code, leads to many of the problems above. In addition, the interfaces today's systems offer the user have grown in complexity to the point that few people understand all the available features and the extra buttons and widgets only get in the way. (Much the same can be said of C++.)

It is because of these problems that people look for better approaches to producing better software. The studies, experiments, approaches and methodologies developed here and their validation in real-life programming are the heart of software engineering.

Software Engineering

Software engineering has been defined by several authors as the use of engineering principles to obtain high-quality software. Engineering principles include the use of both knowledge and skills in an organized and practical manner. High-quality software has many aspects, most of which involve avoiding the above set of problems.

The first foundation for software engineering is a good working knowledge of computer science theory and practice. The theoretical background involves knowing how and when to use data structures, understanding a broad range of useful algorithms, knowing how to develop new algorithms where necessary, and understanding what problems can be solved and what can't. The practical knowledge includes a good understanding of the workings of the hardware as well as thorough knowledge of the available programming lan-

guages and tools. It also involves understanding a multitude of approaches to design such as the design patterns discussed in Chapter 13.

This knowledge is typically accumulated over time through course work, readings, and practical experience in the design, development and maintenance of systems. Good programmers are like good teachers, always striving to broaden their horizons and learn more. The best programmers and designers have a good background and can remember and apply to a new situation everything they have seen previously.

In addition to knowledge, good software engineering (and good engineering in general) requires a number of skills the programmer should develop, including inventiveness, judgment, communication, and objectivity. Inventiveness is needed to put together solutions to new problems as they arise. It can be overdone, but programming is problem solving to a large extent, and understanding how to solve problems has an important role. Good judgment comes into play in deciding among competing solutions, in tracking down bugs, in determining what features can and cannot be added to a system, and even in figuring out what system to build in the first place. It comes with experience and knowledge, but must also be exercised and kept fresh.

Communication skills, both oral and written, are often underemphasized in training programmers. Programmers today rarely work alone. Most work in groups, and senior programmers typically find themselves leading teams of programmers. As teams grow larger, more time is typically spent in meetings and in interacting with the other programmers than in doing the actual coding. In all these circumstances, the ability to get one's point across and to understand what others are saying is an essential part of the job. Written communication is equally important: one's design descriptions must be readable so that others can understand them later, and documentation on how the system can and should be used is equally essential.

The final thing a programmer needs is a questioning attitude. Early on we indicated that design is an evolving process, that one should always ask oneself how a design can be improved. The same is true of the code, the user interface, and most other facets of the system. A questioning attitude is also essential in reviewing your own and others' code and designs. A technique that has evolved for improving software is reviewing the design and code as early in the development process as possible. During such reviews, one questions if the design will work under all circumstances and attempts to figure out how to break it. A questioning attitude also comes into play in testing and debugging. Testing involves finding ways to make the program fail and asks how this can be done. Debugging involves asking the related question: if the program failed in this way, what was the cause?

Asking questions in the programming process is all the more difficult because programmers are put in the situation of criticizing themselves. Looking for a better design involves assuming that your own design is not the best; finding problems in code requires assuming that your code may be wrong, that you may have made a mistake. What programmers are told to strive for here

is the ability to question oneself without being critical. Such an egoless attitude takes a lot of practice and self-confidence, but it does tend to make better programmers.

Software Quality

The quality of a piece of software, what software engineering is designed to achieve, is assessed in a variety of ways. Unfortunately, all these assessments are imprecise and the best one can do is to estimate the quality of the software from a variety of hard-to-measure values.

The most obvious measure of quality is whether the software does what it is supposed to. Here one would like to take the definition of the problem the software was meant to address and see if it actually solves that problem. This is called determining the *correctness* of the software; the process of doing this is called *verification* because one is attempting to verify or prove the software does what it should. Verification, however, is generally impossible since the statement of what the software is supposed to do is usually imprecise and does not cover all the cases. Moreover, proving anything about something as complex as a piece of software is impractical at best and generally intractable.

A better measure of the utility of the software is whether it meets the user's needs. This is called *validation*. Validation is typically the result of a long testing process in which the testers and the eventual users of a system try it out to see how well it meets their actual needs. Validation is a much more tractable problem than verification. However, it does not necessarily prove that the software actually works or that it works for all cases or under all circumstances. Like testing, it is an incomplete process.

The problem with validation is that it can overlook aspects of the *reliability* of the software. Reliability is a measure of the number of bugs remaining in the software. It is probably the best quantitative measure of software quality; it is the only such measure proven to have a strong correlation with the software's overall perceived quality. Moreover, since bugs are generally the aspect of the software most irksome to the user, their elimination can only help to make the software better.

Software bugs can be mere nuisances or major problems. The *robustness* of a software system is a measure of how it reacts in the presence of errors, whether the user's, its own, or the underlying machine's. A robust system recovers gracefully from errors, generally displaying a message and indicating how to undo any damage or to accomplish what was actually meant. A system that is not robust will crash, lose the user's work, or display some other undesirable behavior in the face of an error.

Another measure of quality is the *performance* of the system. Performance can be viewed along a variety of dimensions. Some of these involve the utilization of machine resources such as processing time or memory and disk space; others involve the performance of the user interface and whether it meets user expectations (see Chapter 11). In measuring system performance, one must

take into account what the system is supposed to do and measure performance relative to the minimum required for these tasks.

User perception of a program affects quality in ways other than performance. An increasingly important criterion for software quality is the quality of the user interface, generally measured by how easy it is to use. A *user-friendly* interface is easy to use both for novices and experts and provides the necessary information in the best manner possible. Assessing the quality of a user interface involves all the human-factors criteria discussed in Chapter 11.

Two other measures of software quality concern the system's evolution over time. The first, *portability*, has to do with how easy it is to move the system from one platform to another. This could be as extreme as moving from UNIX to Windows or as simple as moving from one release of an operating system to another. Portable software is generally more adaptable and easier to evolve. The second measure, *extensibility*, is more difficult to pin down; it concerns how easy it is to add new features to the software or to adapt it to different applications. While this is often difficult to determine, it does become one of the most important criteria for a long-lived system.

Software Management

As software has gotten larger and teams of programmers must be coordinated to build a system, software engineering has come to include many of the tasks typically associated with management. These include organizing people, devising an overall process for software development, and estimating costs.

When people started to work on software in teams, they soon discovered that team programming offered diminishing returns. The amount of code generated by one programmer decreased as more programmers were added to a team. Some studies showed that with a team of five, each programmer's productivity went down by 20%, while a team of ten yielded a 40% decrease in productivity. My own experience with student teams suggests that once a team reaches a critical size of around five, little further work is accomplished by adding more programmers.

This phenomenon was called the "mythical man-month" by Fred Brooks.¹ The cost of software is typically measured in terms of the amount of work required for its development. In most industries, software included, this is calculated by multiplying the number of people working on a project by the time, in months or years, that each person works. Thus, if four people work on a given project for six months each, the project is said to require twenty-four person-months (or man-months in the standard terminology). The fact that programmers become decreasingly productive as teams get larger makes this

1. See Frederick P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, 1995, for more information. (This is the second edition, the first edition actually dates back to 1975.)

measure inappropriate for software: two people working on a project for ten months generally produce more and better code than ten people working on the same project for two months.

The productivity loss with increasing numbers of programmers is due primarily to the increased communication costs. One way to minimize these costs and thereby maximize productivity is to concentrate on organizational methods for software teams. These techniques look for an organizational structure for the team that minimizes the number of people each programmer needs to communicate with, generally by employing some sort of managerial hierarchy. This goal is often undercut by the independent nature of many programmers and the complex interdependencies in most software systems.

Another approach to increasing productivity is to design the application to be easy for multiple programmers to work on. This involves breaking the program into isolated components and developing well-defined interfaces between them. If components are independent except for well-defined interfaces, then in principle each component can be written and tested independently by different programmers and the pieces will fit together quite readily. In practice, this is often not the case, since interfaces change and are not always well-understood even if well-defined. However, a good design goes a long way toward simplifying the work of a programmer team.

This initial breakdown into separate interfaces is one place where object-oriented design really shines. An object-based design allows the definition of sophisticated yet simple-to-understand interfaces. The power of objects lets the interfaces be organized and presented in a way that is natural for the given problem. Object methods generally require fewer parameters and are easier to understand than a corresponding procedural design. Also, because objects are generally self-contained, each object definition is a natural basis for dividing the problem into independent subproblems. We address issues in designing larger systems more fully in the next chapter.

Other management issues arise in developing a programming process that determines what is done where and by whom. This includes the documentation required at each stage, how testing is to be done, coding conventions, as well as design and code reviews. Many companies doing software development have a book describing their programming process (even if it's not always followed) that they expect incoming programmers to adopt. We address some of the process issues further in discussing the various programming stages later in this chapter.

A final management issue addressed by software engineering is estimating the cost of a software project. Cost here can be the monetary cost or just the amount of time necessary to write a program. Cost estimation is important at all levels. Companies need it to determine whether or not to build a given system and what resources should be devoted to its construction. Professors need it to determine how long a given assignment should take or how complex an assignment can be for a given amount of time. Students need it to determine when to start their assignments.

There are a variety of techniques for cost estimation. The more complex involve building a detailed model of the software system that includes information on the complexity and interactions of each component. This approach requires both a good understanding of the system to be written and accurate modeling parameters. It is sometimes useful in industry but not that useful to students. The most widely used approach to cost estimation remains expert judgment, i.e. asking someone who has built a lot of systems (preferably similar ones) to evaluate the requirements, complexity, environment, etc., and make an educated guess at the amount of time involved. My own personal approach is to make such an estimate and then multiply it by four.

Other approaches to cost estimation exist. Some involve using a top-down or bottom-up breakdown of the software to divide the cost estimate into smaller, more tractable pieces, and then putting the result back together. Others, including most student projects, use Parkinson's Law; i.e. the time needed to complete a software project increases up to the maximum time available. In industry, another approach is cost-to-win. Here, in bidding on a contract to write a software system, one determines the cost level necessary to win the contract and estimates that, regardless of what the actual cost may be. All told, however, as one gets more experience with writing systems, one tends to get better at estimating the amount of effort involved. Our advice here, especially to students, is always to start software development as soon as possible since it always takes longer than you expect.

THE PHASES OF SOFTWARE ENGINEERING

Software development has a lot in common with other engineering problems. In the engineering domain, developing a solution to a given problem, whether building a bridge or making an electronic component, involves a sequence of interconnected steps. The steps or phases occur in software development as well.

The first step is formulating the problem. Here the goal is to understand the nature and general requirements of the problem. In building a bridge, this means understanding the load the bridge must carry, the approximate locations where it can be built, the height requirements, and so on. In software development it typically means understanding the problem from the user's perspective and is called *requirements analysis*.

The second step involves defining the problem precisely once it is understood. Here one would specify the site for the bridge, its size, and a general outline of the type of bridge to be built. In software development this step is called *specifications* and involves developing a precise statement of exactly what the program will do, often even to the point of writing user manuals and other such documentation for the prospective system.

The third stage of development is detailing the solution to the problem. In building a bridge this would mean determining the exact configuration, computing the size of the cables and beams, and developing the complete blueprints for the bridge. In software development, this is the design process we have been talking about throughout this text. *Top-level design* involves developing an outline to the problem solution while *detailed design* involves specifying all the details needed so the design can be handed off to a programmer.

The next stage of development, *implementation*, corresponds to the actual building of the bridge or the actual coding of the program. If the design was done correctly and in enough detail, this should be the easiest stage (at least conceptually).

The final stage of development, *maintenance*, starts once the solution has been built. For a bridge it means continually repainting, repaving, and making any other repairs necessary. For software it involves fixing bugs, adding new features, and adapting the software to new architectures as needed.

In the remainder of this section, we consider these phases in more detail from a software point of view. To illustrate these phases, we introduce the problem we discuss here and in the next chapter:

One of the oldest computer games (it was created at MIT in 1962 and I was playing it at Dartmouth in 1970) is Spacewar, where two (or more) spaceships are flying around a 2D universe consisting of one or more suns. The suns have gravity and the ships have thrusters (forward or forward and reverse) and the ability to rotate left and right. They can also shoot unguided, unpropelled missiles that decay after a given time. The object is to shoot the other player(s).

Your job is to program an updated version of this game. Your implementation should handle multiple users playing on separate machines. You might also consider a 3D universe rather than a 2D one. (Note that one current invocation of Spacewar is the network game *netrek*.)

Requirements Analysis

The first step in software development is to understand the problem from the user's perspective. The developer should understand exactly what the program needs to accomplish and how it fits into the user's current or proposed environment.

This step is generally accomplished by interacting with the prospective users by written questionnaires, formal interviews, talking to the potential users, or even working alongside them until the problem and how the solution will fit in are well understood. The important point here is to undertake the process with an open mind, not with a solution already in hand. The idea is to determine what the actual problem is so as to find the proper solution, not to take a solution and fit it to the problem. Note that if you are building a system for yourself, this step simply involves introspection.

Once the problem is understood, the developer's next step is to put together a list of requirements the eventual solution must meet. These requirements

determine the outlines of the “best” solution to the problem. (Note that the best solution in some cases isn’t even a program.) This process is generally done in two stages. The first involves gathering as many requirements as possible. Here the developer uses the interaction with potential users to construct “wish lists” of solution features that should be as broad and inclusive as possible. The second step is to merge and then prioritize this combined list. The outcome here should be a list of the features required in the solution and a second list of optional features, with some indication of their desirability.

Determining the mandatory requirements and assigning priorities to the others is a nontrivial process. Ideally, it should be addressed in conjunction with the eventual users of the system, determining what their actual and perceived needs are and how the proposed system can best meet these needs. This is often done through questionnaires asking potential users to rate various items in order of importance. One should also determine the aspects of the system most crucial to its intended purpose and the aspects needed for its acceptance or commercial success.

The requirements should not be limited to the technical aspects of the solution. They should also include such items as the target architecture and environment, current systems with which the new program should interact, limitations on resources such as memory or disk, display requirements, user interface requirements, and so forth. Figure 15-1 organizes some of the requirements for the Spacewar program as either mandatory or optional. A further refinement might assign actual priorities to the optional requirements.

The result of all this should be a good understanding of the problem without any need to define an actual solution. The developer should understand, after requirements analysis, whether a particular solution will be acceptable to the user and, of two different proposed solutions, which will better suit the user’s needs.

Specifications

The second step in software development is to restate the requirements from the programmer’s point of view. The object of this stage is to produce a document detailing exactly what the eventual system should do. The new document, a specification for the eventual software system, is then a sound basis for actually designing and building the system.

Specifications concentrate on what the program does rather than how it does it. During this development stage, the programmer is attempting to outline the software system so that it can ultimately be designed. This is generally done in four steps: building a model of the system, defining the system inputs and outputs, defining the actions of the system for those inputs and outputs, and finally detailing other information pertinent to the design and eventual coding of the system.

Mandatory requirements:

- 1) Multiple players should be able to connect to or disconnect from the game at the start of each round. Players each have their own display.
- 2) Each player has a unique ship.
- 3) The game is divided into rounds.
 - a) At the start of a round the players are given an initial position and velocity by the system. The position and velocity should be safe (i.e. not inside or directed toward a star).
 - b) Each player gets a fixed number of missiles per round.
 - c) Missiles time out after going about half the size of the screen.
 - d) If an active missile hits a ship, both explode and the player whose ship was hit is out of the round.
 - e) If a missile hits another missile, both explode.
 - f) A round ends when either only one player remains or when all remaining players are out of missiles. The end can come only when no bullets are on the screen.
 - g) A player "wins" a round by being the last player remaining.
- 4) Players should be able to control their ship from either the mouse or the keyboard.
- 5) Ships should have main thrusters as well as rotation thrusters.
 - a) Main thrust is cumulative; i.e. the thrusters accelerate or decelerate the ship, not start it or stop it.
 - b) Rotation is on/off. When a rotation thruster is on, the ship rotates. As soon as it is turned off, the ship stops rotating.
- 6) The program should run on Sun workstations.

Optional requirements:

- 7) The game can be in 2D or 3D.
- 8) Main thrusters can be forward only or forward and reverse.
- 9) Keep track of each person's score, i.e. the number of times he or she has won.
- 10) The system should port to NT.
- 11) The NT and Sun versions should be interoperable.

Figure 15-1 Sample requirements for the Spacewar program.

The first step is to construct a model of what the system will do. This is generally done using a combination of text and diagrams. The most relevant diagrams are data-flow diagrams detailing not how the system will work but rather what data comes into the system, what transforms are applied to that data, and what data comes out of the system.

Consider the Spacewar program. The system must read and interpret user input and translate this input into internal commands. These commands are sent to a central controller that maintains the state of the game and sends commands back to a display package to display this state. In addition, the cen-

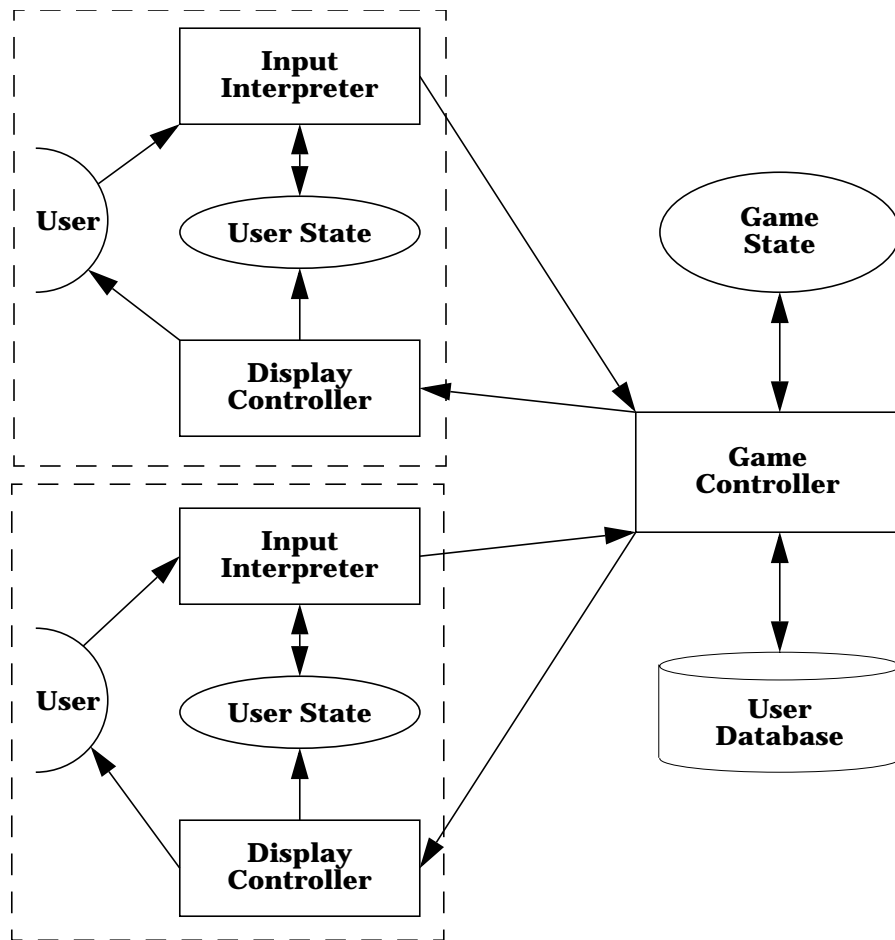


Figure 15-2 Data-flow diagram for a simple Spacewar program.

tral controller maintains a database of users to keep track of their wins. This information can later be provided to the display controller when a round is over. Finally, the display controller needs to interact with the input processor so that only currently relevant inputs are processed.

This simple structure is reflected in the data-flow diagram of Figure 15-2. There are several different types of nodes here. The semicircles on the left represent the users of the program; these are actors that can generate inputs and outputs as needed. The rectangular boxes represent actions; these take data in, process it, and then generate other data out, and are typically labeled with a description of what they do. Arrows between the users and the boxes or between different boxes represent the flow of data, typically commands or messages, between the actions. The elliptical boxes represent local data repos-

itories, which are data structures maintained only while the program is running. The action box connected to such a data node can either store or retrieve data from the data structure. The cylindrical box at the bottom right is a more permanent data repository that is typically saved between system runs and can even be a full-fledged database system shared among multiple users. Finally, the dotted lines group components together to form units.

The diagram shows that commands flow from the user to the input interpreter, which uses information from the local or user state to determine what is legal and eventually to send the next move or other information to the game controller. The game controller processes these requests from multiple users, using the temporary repository reflecting the state of the game and the permanent repository of user information. The result of this processing is sent back to each user's display controller, which in turn updates the user's display and modifies the local state information. The dotted boxes indicate the grouping of components for each user in case additional users are allowed.

A data-flow diagram like this, annotated to describe what each of the nodes and arrows is responsible for, is a starting point for a more complete specification. It contains the outlines of a model describing what the eventual software will do. This model must be fleshed out with details of the system's inputs and outputs and the processing the system will perform.

There are no standard forms for data-flow diagrams. (Actually, there are quite a few different ones, but many standards do not make a "standard.") The format we use here represents a simplified consensus approach and illustrates the different types of notes and relationships typically included within such a diagram. Programmers should generally adopt a particular convention and use it consistently.

An alternative approach here is to create an object-oriented specification in which one uses objects to describe the problem rather than a data-flow diagram. The two approaches are closely related, since the elements of the data-flow diagram are often essentially objects. However, there are differences in emphasis that are important for inexperienced designers. Specifications should define the problem, not its solution. When using objects, one is immediately tempted to think in terms of the solution, especially if one will be using an object-oriented design. This essentially bypasses the specifications stage and generally yields less well-thought out and hence poorer designs. Using a separate notation here helps distinguish the two phases and lets the designer concentrate on the problem rather than the solution.

Once the programmer has completed a diagrammatic model of the system, the next step is to detail the inputs and outputs. The idea here is to describe the user interface so that it can be easily implemented. If the user interface is graphical, this should include the interface sketches discussed in Chapter 10. If the interface is textual, it should include a grammar describing all the commands. In any case, all buttons and commands available to the user should be noted along with their appropriate meaning and corresponding actions. The

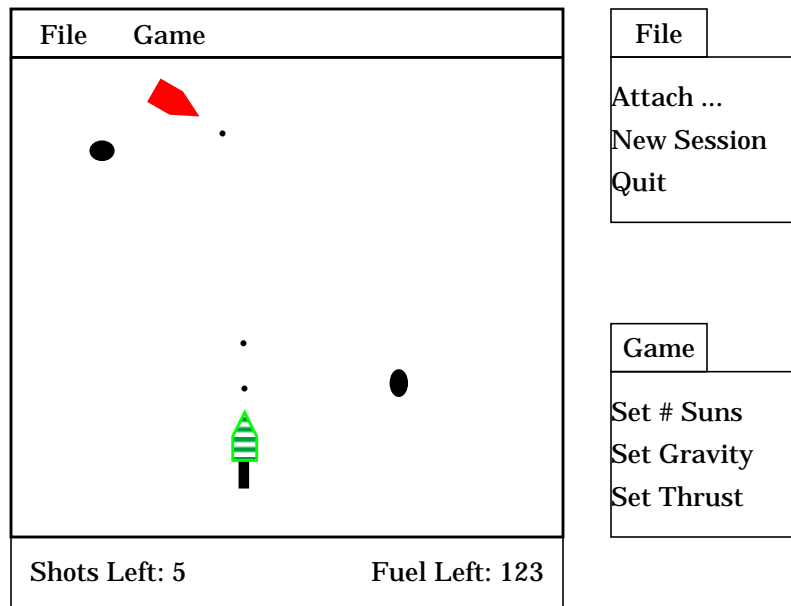


Figure 15-3 User-interface sketches for the Spacewar program.

output views, both standard and resulting from the different inputs, should also be detailed.

For the Spacewar program, the user-interface specification might show sketches of the interface, as in Figure 15-3, accompanied by a description of how users control their spaceships showing which keys are used and how the mouse buttons work. It would also include sketches of other screens and dialog boxes. In Spacewar, dialog boxes are needed for creating a new session (to name the session so others can attach to it), for attaching to a session, and for displaying the current scores list at the end of a game.

Describing the processing performed by the action boxes of the model is the third step in a specification process. This description should be precise and detailed enough that a software designer can understand the overall system well enough to evaluate alternative designs. It should include information about both normal and error processing. Here information would be given about how the positions of the stars are determined; how gravity, spaceship motion, and missile motion are computed; how close missiles must be to explode, what happens when an explosion occurs; what happens if two ships collide or if a ship collides with a star; and so on.

In saying what the program should do, it is often useful to talk about the different states the system can be in. This can be done graphically with a state-transition diagram such as that in Figure 15-4. Such a diagram should

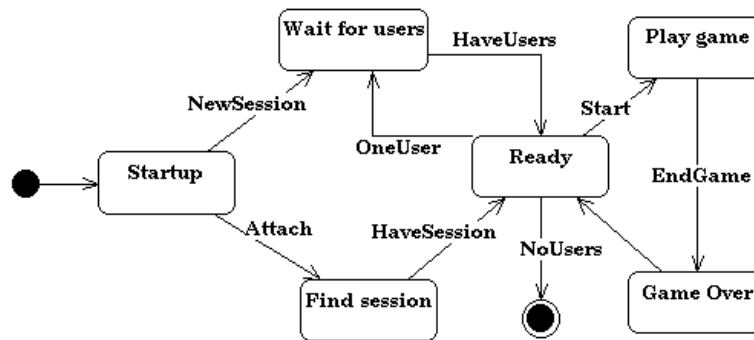


Figure 15-4 State-transition diagram for Spacewar game logic.

again be accompanied by appropriate annotations and a natural-language description to make it comprehensible.

Figure 15-4 shows the top-level logic of the program. The black dot indicates a starting point, the rounded rectangles indicate states and the arcs indicate transitions between the states. The diagram indicates that when the program starts, the user is given the option of starting a new session or attaching to an existing one. If a new session is started, the system waits for another user to attach before starting a game. If attach is selected, the user is prompted to select a session. Once attached, the program can be in one of three states. The ready state exists before the game actually starts; no input is allowed here. Once the game starts, the play state takes over. Here the user can control the spaceship, fire missiles, etc. Finally, once the system detects that a round has ended, the program enters the game-over state in which the winner is declared, high scores are displayed, and the system waits for each user to be ready to go again before reentering the ready state. An additional detail indicated in the diagram is that the system checks whether multiple users are still attached when in the ready state: if only this user remains, the game reenters the wait state, and if no user remains, the game exits. The black dot with a circle around it indicates an exit state.

State-transition diagrams such as this one are part of the Unified Modeling Language (UML) described in Chapter 6. A number of different formats exist for such diagrams and, as long as the programmer is consistent, all of them are acceptable. However, since we are using UML for other design diagrams, it makes sense to use the UML notation for these diagrams as well.

The final step in the specification process is to provide any additional information that might be relevant to the designer. This could include the architectures the system should run on, size limitations, other programs the system should interact with or look like, how the system should handle users connecting or disconnecting, the deadline for the system, and how many programmers will work on it.

The details of the specification can be given in forms ranging from informal comments to formal mathematics. The more mathematical approach, called *formal specifications* or *formal methods*, generally involves using set theory in a programmer-friendly notation to state precisely what the system should do. While such specifications can be helpful for some problems, they are very difficult to do correctly, especially for large interactive systems, and their use is beyond the scope of this text.

The most practical form for larger specifications centers around an outline. The outline details the major components of the system as described in the model and divides each component into major pieces as appropriate. At the lowest level of the outline is a series of short and precise natural-language paragraphs providing the necessary information about some particular aspect of the component. This form can be used as well for simpler specifications to ensure completeness, but in the simpler cases it is often easier just to use diagrams and a natural-language definition, without the formality of an outline. Figure 15-5 shows excerpts from a sample specification for the Spacewar program.

Design

The specifications serve as a prelude to the overall design process. Design, as noted throughout this text, involves analyzing, developing and detailing an appropriate solution to the problem defined by the specification process. It is a process of exploration in which the developer looks at various solutions, determines which solution is best, and then details that solution. It is also a hierarchical process in which the problem is broken down into tractable units.

Designing a moderate-sized software system, especially one involving multiple programmers, is typically more complex than the simple designs discussed so far. However, many of the techniques we have introduced and emphasized were specifically designed to handle the more general case.

The design process works by breaking the system into a small set of components that interact with each other in fixed ways. In a large software system, these components are typically developed by separate programmers or even separate teams of programmers. As such, it is essential that the function and interfaces to each of the components be well defined. This lets the components be developed with minimal interaction with the other components, thereby minimizing communications costs during development. It also provides a basis for testing the individual components. When a component is itself complicated, a good understanding of its function and interface provides the starting point for the second-level design of that component.

The first step in designing a large-scale system, *top-level design*, involves doing such a breakdown and developing the interfaces as the top level. The purpose of this step is solely to understand the function of each component and to specify the interfaces to that functionality. In an object-oriented design, these interfaces are defined as a small set of top-level classes along with their

Game Startup

- 1) When the program starts, the user is asked to choose whether to begin a new session or attach to an existing one.
 - a) An appropriate dialog box should be provided.
 - b) Cancel here should cause the system to exit.
 - c) The dialog should provide a list of active sessions to attach to.
 - d) The dialog should let the user name a new session.
- 2) Sessions exist as long as at least one user is attached. When all users have quit or detached, the session should disappear.
- 3) (Optional) Users should be able to detach from a session and attach to another session without quitting.

Game Control

- 1) Each spaceship maintains a current velocity, position, and orientation.
 - a) When the main thruster is on, it provides a constant acceleration in the direction of the current orientation.
 - b) The rotation thrusters changes the orientation by a fixed amount at each interval. It should take 5-10 seconds to rotate 360 degrees.
 - c) Stars have gravity that provides additional acceleration.
- 2) Missiles are fired from the front of a spaceship.
 - a) Missiles have the spaceship's current velocity plus an additional constant velocity in the direction of the spaceship's orientation.
 - b) Missiles do not accelerate but are affected by gravity.
 - c) A missile should explode if it gets within one missile diameter of another missile, the outside of a spaceship, or a star.
 - d) A missile should last for about half the display when the spaceship is moving at average velocity.
- 3) Stars have a fixed position on the display.
 - a) This can be predetermined or optionally assigned at random for each game.
 - b) The number of stars should be settable by the players.
- 4) The game continues until either at most one player remains or until all players remaining are out of missiles.
 - a) The game does not end if missiles are active.
 - b) Once an end condition has been determined, there should be a delay (about 5 seconds) before the game ends.

Figure 15-5 Sample specifications for the Spacewar program.

public interfaces. In the next chapter we consider different techniques and conventions for creating effective designs for larger software projects.

Interface-centric design is generally repeated until the problem becomes small enough for an individual programmer to understand. It is at this point that implementation details in the form of helper classes, data members, and method pseudocode are typically provided. This is called the *detailed design*. While the top-level design is generally done by a committee of designers or an expert who can understand the overall system, the detailed design is gener-

ally done by the programmer who will eventually implement the corresponding code. The use of objects and interfaces ensures that the rest of the system is isolated from the low-level design decisions made here. Moreover, as we noted, programmers are generally more productive when they work alone, and a compartmentalized solution encourages this.

Coding

As we have seen, translating a detailed design into working code should be straightforward. While many little details must be worked out here, the hard part has already been done.

Coding should not be neglected. The emphasis here for a large software system should be on many of the techniques we have stressed throughout the text. Making a large system work and be maintainable over time requires programmers to adhere strictly to a set of programming conventions. This is generally reflected in naming conventions, stylistic conventions about such issues as indentation and the choice of variable names, and coding conventions such as the order of commonly grouped parameters.

Another emphasis should be on defensive programming. The hardest part of coding a large system involves integrating into a single working unit the pieces of the system written by different programmers. One of the main difficulties here is that bugs arising in the combination are difficult to attribute to any one piece. Defensive programming helps to find these bugs early on and isolate problems more quickly.

Testing

Once code is written, it needs to be tested, as discussed in some detail in “Testing” on page 216. Here we noted that testing is the process of finding errors in the code and that a successful test is one that actually finds an error. Programmers taking this negative approach to testing are more likely to find the problems in their code early and fix them while they are easy to fix. In that section we also covered various approaches to testing, emphasizing low-level testing of individual functions or classes.

Testing of a large system generally is done in stages. First each individual component is tested, generally by the programmer who wrote the code. This *module testing* is generally done either one class at a time or with a small set of related classes. The programmer typically needs to define a *test-case driver*, a simple program to make the appropriate calls on the class being tested.

Once programmers have some confidence in their individual components, they can be put together and the result tested. Such *integration testing* continues until all the components are together. Integration testing generally follows the hierarchical breakdown of the overall system, with components at each level being tested before they are integrated. Integration testing is gener-

ally more difficult and time-consuming than module testing, first because the program being tested is more complex and second because it needs to check for mismatches between the components.

When the whole system is put together, *system testing* begins. This involves testing the overall functionality of the system and is generally done at first by the programmers involved. In a larger company, the final stages of system testing may be done by a separate group to avoid the problems arising when programmers test their own code.

The final stage of testing involves handing the program over to the people who requested the system in the first place and asking them to ascertain that the resultant system meets their needs. This *acceptance testing* often determines whether a program written on a contract basis is acceptable and whether the company writing the program will be paid or not. For student programs, it is used to grade the assignment.

Operation and Maintenance

The final stage of software development, and the one most often ignored by students and many programmers, occurs after the software has been accepted and is in operation in a community of users. While one wishes software would just keep going forever once it is released, this never really happens. Software, no matter how well written or well tested, will have bugs. The systems on which the software runs will change with new hardware or new operating systems. The demands made on the software will also evolve over time. As users become more familiar with the software, they will want additional features. Moreover, the problem the software was originally designed to handle is also likely to change and the software must adapt to such changes.

Maintaining a successful software system is generally the most costly and most time-consuming phase of software development. It is also a difficult, often thankless task that is not particularly liked by programmers. In many companies, maintenance is done by programmers other than those who wrote the initial code, often by new or less respected programmers. This only tends to make the situation worse.

Maintenance programming is hard because it involves understanding the whole system and the whole development process. Fixing a bug or adding a new feature to an existing system actually means going back and redesigning some aspect of the system. Typically, however, design information for an existing system is either nonexistent or out of date, and hence useless. The maintenance programmer must understand the complete program with little more than the code as a resource. Modifying or adding to the code in a large system is difficult because changes to one routine can have unintended and unforeseen side effects in some other routine. Moreover, for consistency, the maintenance programmer must adopt the style of the original programmer.

Making maintenance easier and less costly is one of the principal objectives of software engineering. Much of what can be done along these lines involves

design and coding. As we have noted throughout the text, a variety of strategies such as keeping the code and design as simple as possible, defining strict interfaces, maintaining consistent naming and coding conventions, defensive programming, good and accurate commenting, and insuring code readability can all help to make maintenance a more tractable task. It is important, throughout the whole development process, to specify, design, and develop code that can be maintained and can easily meet new or evolving demands.

Testing must also be adapted to the maintenance process. Here one creates a collection of test cases. Each time the system is upgraded, all the test cases in the collection are used to check that the new version of the system does at least what previous versions did. Such *regression testing*, discussed in Chapter 8, is aimed at ensuring that the new system is an advance rather than a regression over the previous one. To make such testing effective, the programmer should attempt continually to add new test cases to the overall collection.

THE SOFTWARE DEVELOPMENT PROCESS

The various phases of software development described above are not done independently. One of the focuses of software engineering has been to determine how to put these different phases together in an overall software development process. The actual connections among the phases and the formalization of a process based on them have evolved as developers explore different alternatives and as software systems become more complex.

The Waterfall Model

The simplest model of the software development process is to view its stages as successors to one another, as shown in Figure 15-6. Here software development proceeds in clearly defined and distinct stages. The first step is to develop a set of requirements for the system to be built, which is written down, evaluated and then approved. Next, assuming these requirements are fixed, complete specifications for the system are developed and are again reviewed and accepted.

Once specifications are complete, the next step is to design the whole system. The top-level design is developed and the problem is broken down. Each of the subcomponents is designed as well and the design is completed for all components of the system down to the stage where their coding is obvious. Once the designs of all components are complete and have been reviewed and checked, coding begins and the whole system is coded according to the design. Once the system is coded, it is then tested, starting at the module level and moving up to system testing. Once testing is complete, the system is distributed and maintenance begins.

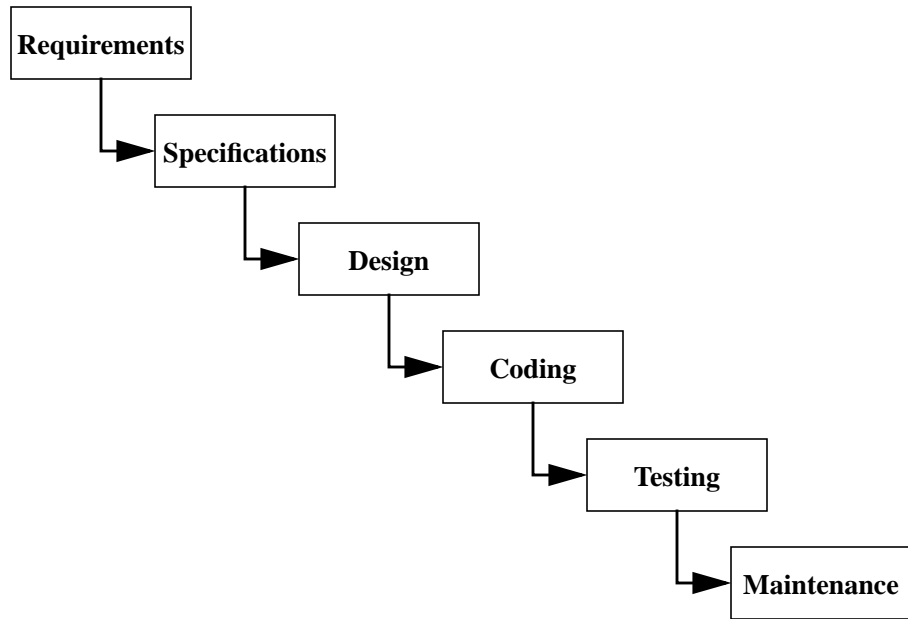


Figure 15-6 Waterfall model of software development.

This process is called the *waterfall model* of software development because the development stages proceed as a cascade, with the diagram of the phases looking a little like a waterfall. The model does allow some flexibility. For example, it should be possible to begin coding some sections of the program before completing the detailed design of everything else and it should be possible to begin module testing of components as they are coded. However, the general view here is that each phase of development is completed before the next is begun.

This model of software development has proven unrealistic for real systems. Information tends to flow not only from one phase of development to the next, but also from each phase back to its predecessor. When a system is being specified, developers often note new features that might be desirable or want to change the priorities of others in the requirements. When doing design, one often needs to change the specifications either to enhance the system or to handle what would otherwise be conflicting demands. When doing coding, one often finds that the design must be enhanced with additional methods or even at times changed to handle some unforeseen situation. When doing testing, one often must go back and change the code to accommodate bug fixes. Finally, maintenance involves new user requirements that should percolate down into new code.

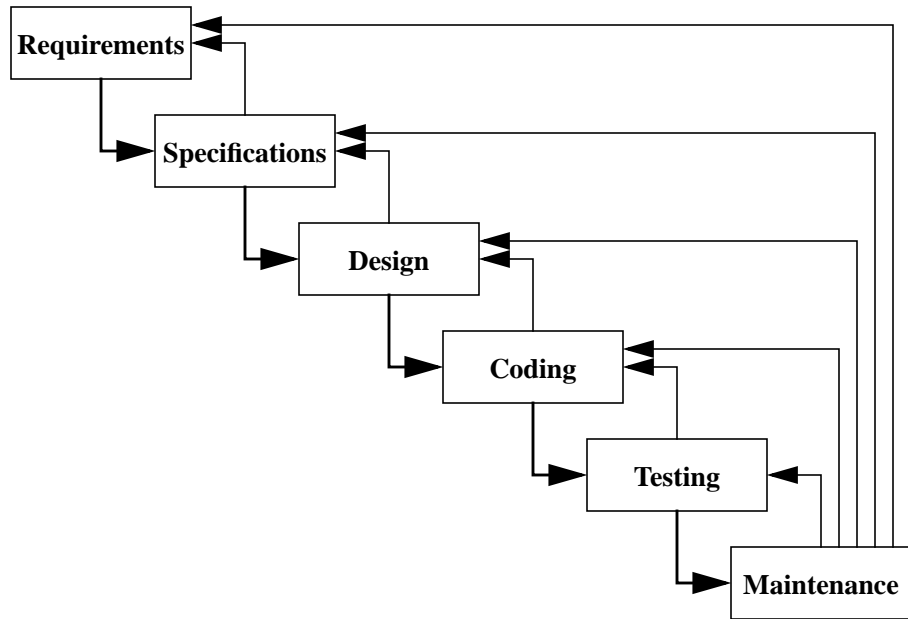


Figure 15-7 The feedback model of software development.

A more realistic model of software development takes these additional information flows into account, typically by augmenting the waterfall model with feedback paths as shown in Figure 15-7. The result is a *feedback model* of software development. Here the principal flow of information is still from one stage to the next. However, unlike the waterfall model, the feedback model lets information flow from any stage back to the previous stage and explicitly notes that maintenance changes can restart development at any stage.

Most software engineering today follows more or less along the lines of the feedback model. Requirements are generally gathered first and then the developers attempt to define the system to be built. As they begin to design the system, feedback from the design causes the specifications to be modified in various ways that are then put into the evolving design. Coding is typically not done until the portions to code have been designed, although coding of some sections and the detailed design of others often proceeds in parallel. Most module testing also occurs as the code is being written, while integration testing, of necessity, requires most of the code to be present. Testing often finds code errors requiring design changes (and sometimes even the specification if something was found to be too difficult or complex) and development then continues.

Prototypes and the Spiral Model

Both the waterfall and feedback models of software development assume that design can be done before coding begins. While this is generally the case, it can sometimes be better to code first and then design. The prime example of this is user-interface design. It is very difficult to design a good user interface on paper without being able to experiment with how the interface can be used, and one often develops a simple program for this purpose. There are other design decisions, too, whose impact or feasibility cannot be decided a priori. In these cases it is often useful to experiment with simple programs illustrating different design alternatives as a means of exploring the design space. This is especially true where a particular design decision can affect the overall success or failure of the eventual system.

The process of writing a simple program to explore the design space is called *prototyping* and the program itself is called a *prototype*. Prototyping is a viable strategy when the choice among different essential design alternatives is not clear, since it lets the developer experiment with the different alternatives and determine which is best. Putting up a prototype user interface gives the designer a sense of both what the user interface looks like and how it feels to interact with it. A user-interface prototype can even be shown to prospective users to get a feeling for how well it meets their needs.

It is generally desirable to build experimental prototypes as quickly as possible so that they can be evaluated more rapidly and take less time away from the overall design process. Because of this, prototypes are often made by using special systems or different programming languages. For example, user-interface prototypes can be developed rapidly by using a user-interface generator that lets the user select components and put them together graphically. Smalltalk and other interactive languages with extensive user-interface libraries are also used for this purpose.

The main drawback of prototypes is that they must be discarded after use. If they are developed by using a prototyping system or very high-level language, they are usually too inefficient for inclusion in a production program. If they were developed as quickly as possible in the target language, the quality of the code and documentation is generally quite poor and the result is not suitable for inclusion in a system that eventually must be fully debugged and maintained. Unless the programmer is willing to throw the prototype away after it has served its purpose (and to recode its functionality using proper design and implementation methods), prototyping is not a worthwhile alternative.

As systems and user interfaces have become more complex, more design issues have become potential targets for prototyping. Moreover, the development of large, complex systems involves a high degree of risk. A poor design decision at a critical point in the development can make the software unfixably slow or useless for its intended purpose. To take this into account and to

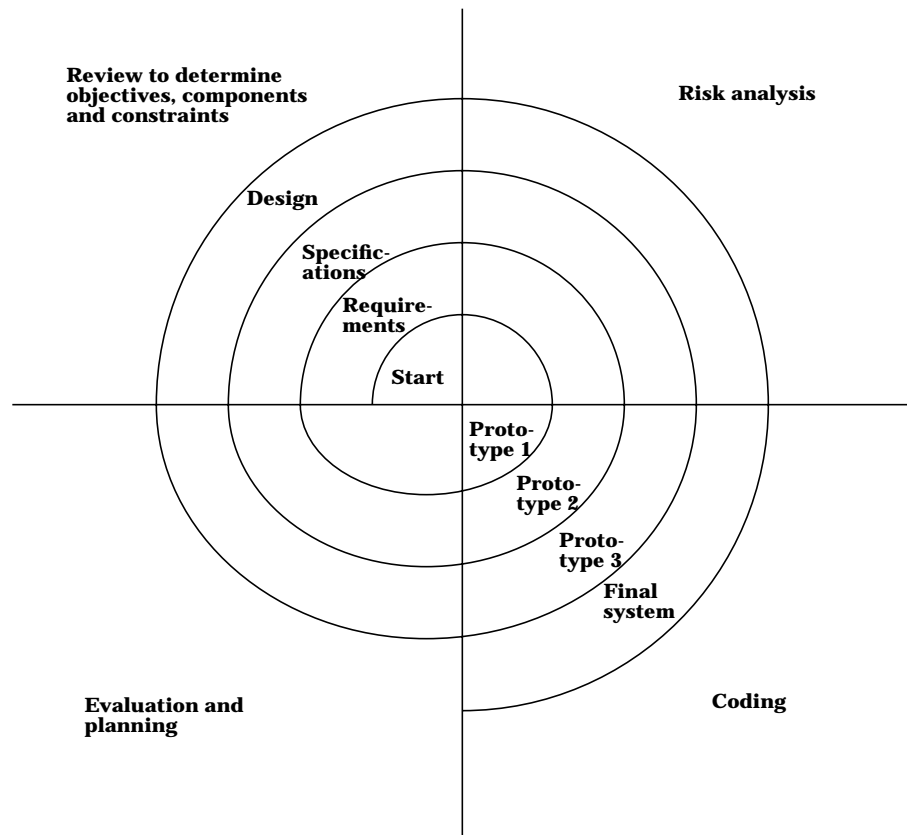


Figure 15-8 Simplified spiral model of software development.

decrease the cost of prototyping, an alternative model of software development has been proposed.

This *spiral model* of software development, shown in simplified form in Figure 15-8, assumes that successive prototypes of the system are built. One starts at the center by reviewing the problem to be solved. Then one does risk analysis to determine the most vulnerable or least understood parts of the proposed system, and these become the subject of the first prototype. The prototype is then tested and reviewed and the results of this analysis are used to derive a set of requirements for the eventual system. Using these desired requirements, a further risk analysis is done again to determine which parts of the system are the least understood or most vulnerable. A second prototype is built to analyze and understand these better. This prototype is again used, tested, and assessed to evaluate the different alternative designs, and from this analysis specifications for the eventual system are developed. These are then tested, after appropriate risk analysis to determine the most vulnerable

portions, in a third prototype. The third prototype is then a fully operational model of the target system. It is again tested to find any weaknesses. From this analysis, a detailed design for the final system is developed and the final system is built using more traditional coding techniques to ensure code quality and maintainability. The last stage involves the normal testing process for this system.

This model can be varied in different ways. If prototypes are unsuccessful or indicate major problems, additional cycles and hence additional prototypes might be required. A third prototype might be used, for example, to experiment with different aspects of the specifications, and a fourth prototype would then represent the fully operational system. The various prototypes, and to some extent even the final system, need not be developed from scratch but can be built on top of their predecessors.

In general a prototype should be as focused as possible. It should provide the minimum functionality necessary to validate or check the design alternative for which it is being developed. This ensures that it can be developed as rapidly as possible and limits the temptation to use it as a production system. In the spiral model, the initial prototypes typically test a single design alternative that must be evaluated in order to understand better how the system should eventually be built. The final prototype is used to demonstrate that the whole system is feasible and actually meets the needs of the target users. In all cases, the prototypes are designed to be discarded, and only their designs and concepts are actually used in the final system.

The proper choice of a development model for a particular system depends on a variety of factors. The more complex and risky the system, the more one tends to use prototyping. If the system is relatively straightforward, for example if it duplicates and extends an existing system, then a design-oriented approach is preferred. If the prospective users aren't sure what they want, prototyping is an inexpensive way to experiment with different alternatives. If the user's requirements are well known in advance, the expense of building and discarding a prototype is probably excessive. In general, prototypes should be used when issues can be resolved only by experimenting with a real system and when the programmer is willing to throw away the prototype and all the work that went into it.

SUMMARY

Software engineering is the use of engineering principles to obtain high-quality software. It has been developed to help with the many problems arising in software development and in software itself. These problems primarily involve the excessive cost of developing and maintaining software and the low quality typical of most software systems.

Software engineering has multiple facets. It involves the proper training and use of programmers. It involves the development of appropriate techniques for defining what to build and how to build it. It involves the development of tools to aid the developer. It involves the use of appropriate management techniques and skills. Throughout all this, it puts an emphasis on developing and maintaining high-quality software.

Software development occurs in phases:

- *Requirements Analysis*: Here the programmer attempts to determine the user's needs and to outline the proposed system. The result is generally a user model and a prioritized list of features to be included in the proposed software.
- *Specifications*: Here the programmer determines precisely what software will be built. This includes a annotated system model, complete user-interface designs, and a prioritized list of the capabilities of the proposed software. The emphasis here is on what is to be built, not how it is to be done.
- *Design*: Here the programmer determines how the software will work. This involves both top-level design, where the overall framework for the system is constructed, and detailed design, where individual packages are specified.
- *Coding*: Here the detailed design is translated into code in the appropriate target language. This should be the easiest phase of software development.
- *Testing*: The code must then be tested. This is done at various levels, starting with module testing to check individual packages or classes, moving up to integration testing as the different packages are put together, and finally ending with system and acceptance testing in which the system is tested as a whole.
- *Operation and Maintenance*: Here the system is used, bugs are fixed, and changes are made to accommodate new user demands or a changing environment. In a successful system, this phase is typically the longest and most costly, often accounting for 80% or more of the overall software cost.

These phases can be organized in various ways. Ideally, they occur one after another, each one finishing before the next one starts; this is the waterfall model of software development. In practice, the knowledge gained in each phase makes one change decisions made in the previous stage. This leads to the more common and practical feedback model. A more recent strategy is to use prototyping to direct software development in order to minimize risk and produce higher-quality software. This is reflected in the spiral model.

EXERCISES

- 15.1** Write complete requirements for the Spacewar program.
- 15.2** Write the requirements for the orrery described in Chapter 1.
- 15.3** Write complete specifications for the Spacewar program.
- 15.4** Suppose you wanted to explore using computers in teaching introductory geology. Interview faculty and students in this area to determine what a suitable system should or could do and then draw up a full set of requirements for the system.
- 15.5** Computer systems are beginning to be equipped with software libraries for speech input and output. Suppose such a library is available. You want to develop a new product that would allow the use of speech in any application currently running on the machine (i.e., it should be able to speak error messages and prompts and simulate keyboard or mouse input based on speech commands). Draw up requirements and specifications for such a system.
- 15.6** Investigate and describe the software development process at some company in your area.
- 15.7** Draw up specifications for a system that allows the user to design and visualize a flower garden. The system should allow the specification of what should be planted when and where in a garden setting. Using a database of the growth characteristics of different plants (height and color based on time from planting), the system should show the user what the garden would look like at any point in time. The system should also provide a user-friendly front end to let the user design the garden.
- 15.8** Develop specifications for a system that simulates a car race. One part of the system should allow a user to design race tracks. The core of the system should support one or more users racing their cars around the predefined tracks.
- 15.9** Develop specifications for a system to do menu planning. As a first step you should do a requirements analysis to determine how people who cook at home might use a computer to help them with planning meals, shopping, adjusting recipes, etc. Then you should draw up complete specifications for a system to meet these requirements. Evaluate the specifications by presenting them to the potential users.

