CHAPTER 7

Software Engineering

In this chapter we explore the problems that are encountered during the development of large, complex software systems. The subject is called *software engineering* because software development is an engineering process. The goal of researchers in software engineering is to find principles that guide the software development process and lead to efficient, reliable software products.

Software engineering is the branch of computer science that seeks principles to guide the development of large, complex software systems. The problems faced when developing such systems are more than enlarged versions of those problems faced when writing small programs. For instance, the development of such systems requires the efforts of more than one person over an extended period of time during which the requirements of the proposed system may be altered and the personnel assigned to the project may change. Consequently, software engineering includes topics such as personnel and project management that are more readily associated with business management than computer science. We, however, will focus on topics readily related to computer science.

7.1 The Software Engineering Discipline

To appreciate the problems involved in software engineering, it is helpful to select a large complex device (an automobile, a multistory office building, or perhaps a cathedral) and imagine being asked to design it and then to supervise its construction. How can you estimate the cost in time, money, and other resources to complete the project? How can you divide the project into manageable pieces? How can you ensure that the pieces produced are compatible? How can those working on the various pieces communicate? How can you measure progress? How can you cope with the wide range of detail (the selection of the doorknobs, the design of the gargoyles, the availability of blue glass for the stained glass windows, the strength of the pillars, the design of the duct work for the heating system)? Questions of the same scope must be answered during the development of a large software system.

Because engineering is a well-established field, you might think that there is a wealth of previously developed engineering techniques that can be useful in answering such questions. This reasoning is partially true, but it overlooks fundamental differences between the properties of software and those of other fields of engineering. These distinctions have challenged software engineering projects, leading to cost overruns, late delivery of products, and dissatisfied customers. In turn, identifying these distinctions has proven to be the first step in advancing the software engineering discipline.

One such distinction involves the ability to construct systems from generic prefabricated components. Traditional fields of engineering have long benefited from the ability to use "off-the-shelf" components as building blocks when constructing complex devices. The designer of a new automobile does not have to design a new engine or transmission but instead uses previously designed versions of these components. Software engineering, however, lags in this regard. In the past, previously designed software components were domain specific—that is, their internal design was based on a specific application—and thus their use as generic components was limited. The result is that complex software systems have historically been built from scratch. As we will see in this chapter, significant progress is being made in this regard, although more work remains to be done.

Another distinction between software engineering and other engineering disciplines is the lack of quantitative techniques, called **metrics,** for measuring the properties of software. For example, to project the cost of developing a software system, one would like to estimate the complexity of the proposed product, but methods for measuring the "complexity" of software are evasive. Similarly, evaluating the quality of a software product is challenging. In the case of mechanical devices, an important measure of quality is the mean time between failures,

which is essentially a measurement of how well a device endures wear. Software, in contrast, does not wear out, so this method of measuring quality is not as applicable in software engineering.

The difficulties involved in measuring software properties in a quantitative manner is one of the reasons that software engineering has struggled to find a rigorous footing in the same sense as mechanical and electrical engineering. Whereas these latter subjects are founded on the established science of physics, software engineering continues to search for its roots.

Thus research in software engineering is currently progressing on two levels: Some researchers, sometimes called practitioners, work toward developing techniques for immediate application, whereas others, called theoreticians, search for underlying principles and theories on which more stable techniques can someday be constructed. Being based on a subjective foundation, many methodologies developed and promoted by practitioners in the past have been replaced by other approaches that may themselves become obsolete with time. Meanwhile, progress by theoreticians continues to be slow.

The need for progress by both practitioners and theoreticians is enormous. Our society has become addicted to computer systems and their associated software. Our economy, healthcare, government, law enforcement, transportation, and defense depend on large software systems. Yet there continue to be major problems with the reliability of these systems. Software errors have caused such disasters and near disasters as the rising moon being interpreted as a nuclear attack, a one-day loss of $5 million by the Bank of New York, the loss of space probes, radiation overdoses that have killed and paralyzed, and the simultaneous disruption of telephone communications over large regions.

This is not to say that the situation is all bleak. Much progress is being made in overcoming such problems as the lack of prefabricated components and metrics. Moreover, the application of computer technology to the software development process, resulting in what is called **computer-aided software engineering (CASE),** is continuing to streamline and otherwise simplify the software development process. CASE has led to the development of a variety of computerized systems, known as **CASE tools,** which include project planning systems (to assist in cost estimation, project scheduling, and personnel allocation), project management systems (to assist in monitoring the progress of the development project), documentation tools (to assist in writing and organizing documentation), prototyping and simulation systems (to assist in the development of prototypes), interface

Association for Computing Machinery

The Association for Computing Machinery (ACM) was founded in 1947 as an international scientific and educational organization dedicated to advancing the arts, sciences, and applications of information technology. It is headquartered in New York and includes numerous special interest groups (SIGs) focusing on such topics as computer architecture, artificial intelligence, biomedical computing, computers and society, computer science education, computer graphics, hypertext/hypermedia, operating systems, programming languages, simulation and modeling, and software engineering. The ACM's website is at `http://www.acm.org`. Its Code of Ethics and Professional Conduct can be found at `http://www.acm.org/constitution/code.html`.

design systems (to assist in the development of GUIs), and programming systems (to assist in writing and debugging programs). Some of these tools are little more than the word processors, spreadsheet software, and email communication systems that were originally developed for generic use and adopted by software engineers. Others are quite sophisticated packages designed primarily for the software engineering environment. Indeed, systems known as **integrated development environments (IDEs)** combine tools for developing software (editors, compilers, debugging tools, and so on) into a single, integrated package. Prime examples of such systems are those for developing applications for smartphones. These not only provide the programming tools necessary to write and debug the software but also provide simulators that, by means of graphical displays, allow a programmer to see how the software being developed would actually perform on a phone.

In addition to the efforts of researchers, professional and standardization organizations, including the ISO, the Association for Computing Machinery (ACM), and the Institute of Electrical and Electronics Engineers (IEEE), have joined the battle for improving the state of software engineering. These efforts range from adopting codes of professional conduct and ethics that enhance the professionalism of software developers and counter nonchalant attitudes toward each individual's responsibilities to establishing standards for measuring the quality of software development organizations and providing guidelines to help these organizations improve their standings.

In the remainder of this chapter we discuss some of the fundamental principles of software engineering (such as the software life cycle and modularity), look at some of the directions in which software engineering is moving (such as the identification and application of design patterns and the emergence of reusable software components), and witness the effects that the object-oriented paradigm has had on the field.

Questions & Exercises

1. Why would the number of lines in a program not be a good measure of the complexity of the program?
2. Suggest a metric for measuring software quality. What weaknesses does your metric have?
3. What technique can be used to determine how many errors are in a unit of software?
4. Identify two contexts in which the field of software engineering has been or currently is progressing toward improvements.

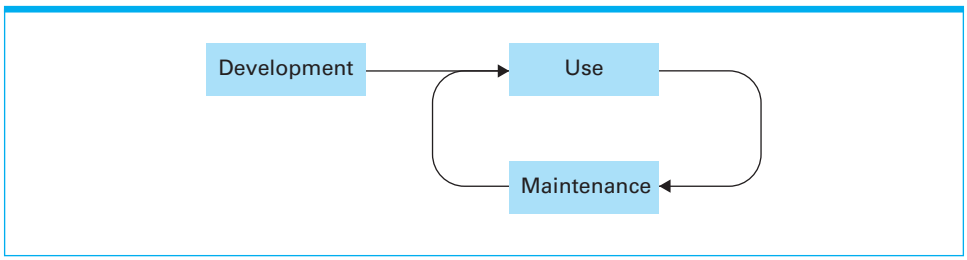
7.2 The Software Life Cycle

The most fundamental concept in software engineering is the software life cycle.

The Cycle as a Whole

The software life cycle is shown in Figure 7.1. This figure represents the fact that once software is developed, it enters a cycle of being used and maintained—a cycle that continues for the rest of the software's life. Such a pattern is common

Figure 7.1 The software life cycle



for many manufactured products as well. The difference is that, in the case of other products, the maintenance phase tends to be a repair process, whereas in the case of software, the maintenance phase tends to consist of correcting or updating. Indeed, software moves into the maintenance phase because errors are discovered, changes in the software's application occur that require corresponding changes in the software, or changes made during a previous modification are found to induce problems elsewhere in the software.

Regardless of why software enters the maintenance phase, the process requires that a person (often not the original author) study the underlying program and its documentation until the program, or at least the pertinent part of the program, is understood. Otherwise, any modification could introduce more problems than it solves. Acquiring this understanding can be a difficult task, even when the software is well-designed and documented. In fact, it is often within this phase that a piece of software is discarded under the pretense (too often true) that it is easier to develop a new system from scratch than to modify the existing package successfully.

Experience has shown that a little effort during the development of software can make a tremendous difference when modifications are required. For example, in our discussion of data description statements in Chapter 6 we saw how the use of constants rather than literals can greatly simplify future adjustments. In turn, most of the research in software engineering focuses on the development stage of the software life cycle, with the goal being to take advantage of this effort-versus-benefit leverage.

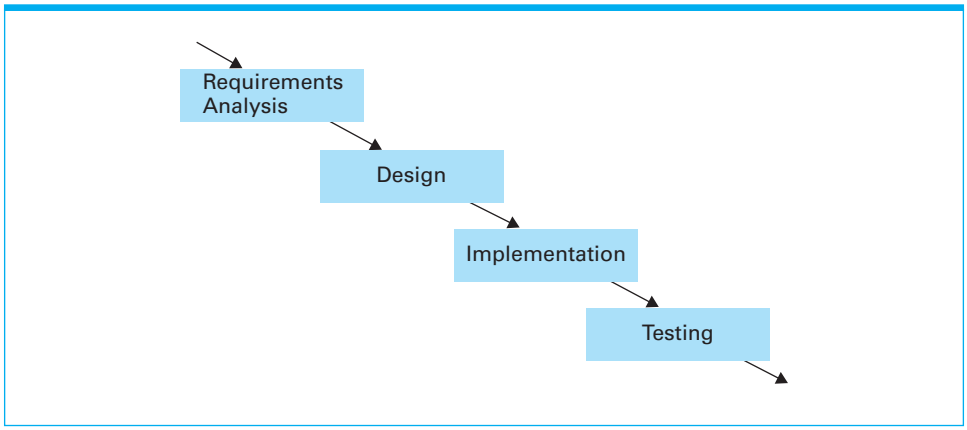
The Traditional Development Phase

The major steps in the traditional software development life cycle are requirements analysis, design, implementation, and testing (Figure 7.2).

Requirements Analysis The software life cycle begins with requirements analysis—the goal of which is to specify what services the proposed system will provide, to identify any conditions (time constraints, security, and so on) on those services, and to define how the outside world will interact with the system.

Requirements analysis involves significant input from the **stakeholders** (future users as well as those with other ties, such as legal or financial interests) of the proposed system. In fact, in cases where the ultimate user is an entity, such as a company or government agency, that intends to hire a software developer for the actual execution of the software project, requirements analysis may start by a feasibility study conducted solely by the user. In other cases, the software

Figure 7.2 The traditional development phase of the software life cycle



developer may be in the business of producing **commercial off-the-shelf (COTS)** software for the mass market, perhaps to be sold in retail stores or downloaded via the Internet. In this setting the user is a less precisely defined entity, and requirements analysis may begin with a market study by the software developer.

In any case, the requirements analysis process consists of compiling and analyzing the needs of the software user; negotiating with the project's stakeholders over trade-offs between wants, needs, costs, and feasibility; and finally developing a set of requirements that identify the features and services that the finished software system must have. These requirements are recorded in a document called a **software requirements specification.** In a sense, this document is a written agreement between all parties concerned, which is intended to guide the software's development and provide a means of resolving disputes that may arise later in the development process. The significance of the software requirements specification is demonstrated by the fact that professional organizations such as IEEE and large software clients such as the U.S. Department of Defense have adopted standards for its composition.

From the software developer's perspective, the software requirements specification should define a firm objective toward which the software's development can proceed. Too often, however, the document fails to provide this stability. Indeed, most practitioners in the software engineering field argue that poor communication and changing requirements are the major causes of cost overruns and late product delivery in the software engineering industry. Few customers would insist on major changes to a building's floor plan once the foundation has been constructed, but instances abound of organizations that have expanded, or otherwise altered, the desired capabilities of a software system well after the software's construction was underway. This may have been because a company decided that the system that was originally being developed for only a subsidiary should instead apply to the entire corporation or that advances in technology supplanted the capabilities available during the initial requirements analysis. In any case, software engineers have found that straightforward and frequent communication with the project's stakeholders is mandatory.

Design Whereas requirements analysis provides a description of the proposed software product, design involves creating a plan for the construction of the proposed

system. In a sense, requirements analysis is about identifying the problem to be solved, while design is about developing a solution to the problem. From a layperson's perspective, requirements analysis is often equated with deciding *what* a software system is to do, whereas design is equated with deciding *how* the system will do it. Although this description is enlightening, many software engineers argue that it is flawed because, in actuality, there is a lot of *how* considered during requirements analysis and a lot of *what* considered during design.

It is in the design stage that the internal structure of the software system is established. The result of the design phase is a detailed description of the software system's structure that can be converted into programs.

If the project were to construct an office building rather than a software system, the design stage would consist of developing detailed structural plans for a building that meets the specified requirements. For example, such plans would include a collection of blueprints describing the proposed building at various levels of detail. It is from these documents that the actual building would be constructed. Techniques for developing these plans have evolved over many years and include standardized notational systems and numerous modeling and diagramming methodologies.

Likewise, diagramming and modeling play important roles in the design of software. However, the methodologies and notational systems used by software engineers are not as stable as they are in the architectural field. When compared to the well-established discipline of architecture, the practice of software engineering appears very dynamic as researchers struggle to find better approaches to the software development process. We will explore this shifting terrain in Section 7.3 and investigate some of the current notational systems and their associated diagramming/modeling methodologies in Section 7.5.

Implementation Implementation involves the actual writing of programs, creation of data files, and development of databases. It is at the implementation stage that we see the distinction between the tasks of a **software analyst** (sometimes

Institute of Electrical and Electronics Engineers

The Institute of Electrical and Electronics Engineers (IEEE, pronounced "i-triple-e") is an organization of electrical, electronics, and manufacturing engineers that was formed in 1963 as the result of merging the American Institute of Electrical Engineers (founded in 1884 by 25 electrical engineers, including Thomas Edison) and the Institute of Radio Engineers (founded in 1912). Today, IEEE's operation center is located in Piscataway, New Jersey. The Institute includes numerous technical societies such as the Aerospace and Electronic Systems Society, the Lasers and Electro-Optics Society, the Robotics and Automation Society, the Vehicular Technology Society, and the Computer Society. Among its activities, the IEEE is involved in the development of standards. As an example, IEEE's efforts led to the single-precision- floating point and double-precision floating-point standards (introduced in Chapter 1), which are used in most of today's computers.

You will find the IEEE's Web page at `http://www.ieee.org`, the IEEE Computer Society's Web page at `http://www.computer.org`, and the IEEE's Code of Ethics at `http://www.ieee.org/about/whatis/code.html`.

referred to as a system analyst) and a **programmer.** The former is a person involved with the entire development process, perhaps with an emphasis on the requirements analysis and design steps. The latter is a person involved primarily with the implementation step. In its narrowest interpretation, a programmer is charged with writing programs that implement the design produced by a software analyst. Having made this distinction, we should note again that there is no central authority controlling the use of terminology throughout the computing community. Many who carry the title of software analyst are essentially programmers, and many with the title programmer (or perhaps senior programmer) are actually software analysts in the full sense of the term. This blurring of terminology is founded in the fact that today the steps in the software development process are often intermingled, as we will soon see.

Testing In the traditional development phase of the past, testing was essentially equated with the process of debugging programs and confirming that the final software product was compatible with the software requirements specification. Today, however, this vision of testing is considered far too narrow. Programs are not the only artifacts that are tested during the software development process. Indeed, the result of each intermediate step in the entire development process should be "tested" for accuracy. Moreover, as we will see in Section 7.6, testing is now recognized as only one segment in the overall struggle for quality assurance, which is an objective that permeates the entire software life cycle. Thus, many software engineers argue that testing should no longer be viewed as a separate step in software development, but instead it, and its many manifestations, should be incorporated into the other steps, producing a three-step development process whose components might have names such as "requirements analysis *and confirmation,*" "design *and validation,*" and "implementation *and testing.*"

Unfortunately, even with modern quality assurance techniques, large software systems continue to contain errors, even after significant testing. Many of these errors may go undetected for the life of the system, but others may cause major malfunctions. The elimination of such errors is one of the goals of software engineering. The fact that they are still prevalent indicates that a lot of research remains to be done.

Questions & Exercises

1. How does the development stage of the software life cycle affect the maintenance stage?
2. Summarize each of the four stages (requirements analysis, design, implementation, and testing) within the development phase of the software life cycle.
3. What is the role of a software requirements specification?

7.3 Software Engineering Methodologies

Early approaches to software engineering insisted on performing requirements analysis, design, implementation, and testing in a strictly sequential manner. The belief was that too much was at risk during the development of a large software

system to allow for variations. As a result, software engineers insisted that the entire requirements specification of the system be completed before beginning the design and, likewise, that the design be completed before beginning implementation. The result was a development process now referred to as the **waterfall model,** an analogy to the fact that the development process was allowed to flow in only one direction.

In recent years, software engineering techniques have changed to reflect the contradiction between the highly structured environment dictated by the waterfall model and the "free-wheeling," trial-and-error process that is often vital to creative problem solving. This is illustrated by the emergence of the **incremental model** for software development. Following this model, the desired software system is constructed in increments—the first being a simplified version of the final product with limited functionality. Once this version has been tested and perhaps evaluated by the future user, more features are added and tested in an incremental manner until the system is complete. For example, if the system being developed is a patient records system for a hospital, the first increment may incorporate only the ability to view patient records from a small sample of the entire record system. Once that version is operational, additional features, such as the ability to add and update records, would be added in a stepwise manner.

Another model that represents the shift away from strict adherence to the waterfall model is the **iterative model,** which is similar to, and in fact sometimes equated with, the incremental model, although the two are distinct. Whereas the incremental model carries the notion of *extending* each preliminary version of a product into a larger version, the iterative model encompasses the concept of *refining* each version. In reality, the incremental model involves an underlying iterative process, and the iterative model may incrementally add features.

A significant example of iterative techniques is the **rational unified process** (**RUP,** rhymes with "cup") that was created by the Rational Software Corporation, which is now a division of IBM. RUP is essentially a software development paradigm that redefines the steps in the development phase of the software life cycle and provides guidelines for performing those steps. These guidelines, along with CASE tools to support them, are marketed by IBM. Today, RUP is widely applied throughout the software industry. In fact, its popularity has led to the development of a nonproprietary version, called the **unified process,** that is available on a noncommercial basis.

Incremental and iterative models sometimes make use of the trend in software development toward **prototyping** in which incomplete versions of the proposed system, called prototypes, are built and evaluated. In the case of the incremental model these prototypes evolve into the complete, final system—a process known as **evolutionary prototyping.** In a more iterative situation, the prototypes may be discarded in favor of a fresh implementation of the final design. This approach is known as **throwaway prototyping.** An example that normally falls within this throwaway category is **rapid prototyping** in which a simple example of the proposed system is quickly constructed in the early stages of development. Such a prototype may consist of only a few screen images that give an indication of how the system will interact with its users and what capabilities it will have. The goal is not to produce a working version of the product but to obtain a demonstration tool that can be used to clarify communication between the parties involved in the software development process. For example, rapid prototypes have proved advantageous in clarifying system requirements during requirements analysis or as aids during sales presentations to potential clients.

A less formal incarnation of incremental and iterative ideas that has been used for years by computer enthusiasts/hobbyists is known as **open-source development.** This is the means by which much of today's free software is produced. Perhaps the most prominent example is the Linux operating system whose open-source development was originally led by Linus Torvalds. The open-source development of a software package proceeds as follows: A single author writes an initial version of the software (usually to fulfill his or her own needs) and posts the source code and its documentation on the Internet. From there it can be downloaded and used by others without charge. Because these other users have the source code and documentation, they are able to modify or enhance the software to fit their own needs or to correct errors that they find. They report these changes to the original author, who incorporates them into the posted version of the software, making this extended version available for further modifications. In practice, it is possible for a software package to evolve through several extensions in a single week.

Perhaps the most pronounced shift from the waterfall model is represented by the collection of methodologies known as **agile methods,** each of which proposes early and quick implementation on an incremental basis, responsiveness to changing requirements, and a reduced emphasis on rigorous requirements analysis and design. One example of an agile method is **extreme programming (XP).** Following the XP model, software is developed by a team of less than a dozen individuals working in a communal work space where they freely share ideas and assist each other in the development project. The software is developed incrementally by means of repeated daily cycles of informal requirements analysis, designing, implementing, and testing. Thus, new expanded versions of the software package appear on a regular basis, each of which can be evaluated by the project's stakeholders and used to point toward further increments. In summary, agile methods are characterized by flexibility, which is in stark contrast to the waterfall model that conjures the image of managers and programmers working in individual offices while rigidly performing well-defined portions of the overall software development task.

The contrasts depicted by comparing the waterfall model and XP reveal the breadth of methodologies that are being applied to the software development process in the hopes of finding better ways to construct reliable software in an efficient manner. Research in the field is an ongoing process. Progress is being made, but much work remains to be done.

Questions & Exercises

1. Summarize the distinction between the traditional waterfall model of software development and the newer incremental and iterative paradigms.
2. Identify three development paradigms that represent the move away from strict adherence to the waterfall model.
3. What is the distinction between traditional evolutionary prototyping and open-source development?
4. What potential problems do you suspect could arise in terms of ownership rights of software developed via the open-source methodology?

7.4 Modularity

A key point in Section 7.2 is that to modify software one must understand the program or at least the pertinent parts of the program. Gaining such an understanding is often difficult enough in the case of small programs and would be close to impossible when dealing with large software systems if it were not for **modularity**—that is, the division of software into manageable units, generically called **modules,** each of which deals with only a part of the software's overall responsibility.

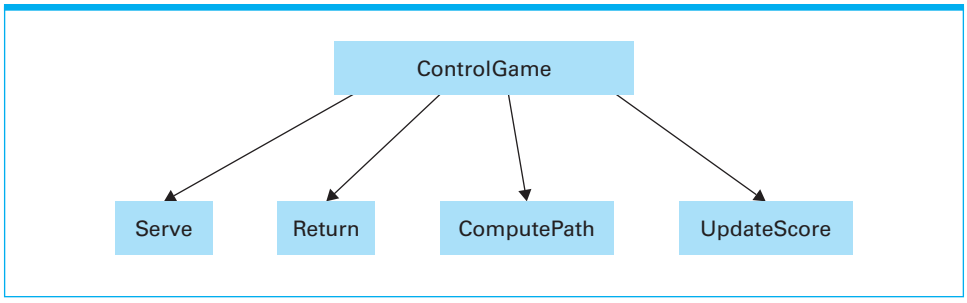
Modular Implementation

Modules come in a variety of forms. We have already seen (Chapters 5 and 6) that in the context of the imperative paradigm, modules appear as functions. In contrast, the object-oriented paradigm uses objects as the basic modular constituents. These distinctions are important because they determine the underlying goal during the initial software design process. Is the goal to represent the overall task as individual, manageable processes or to identify the objects in the system and understand how they interact?

To illustrate, let us consider how the process of developing a simple modular program to simulate a tennis game might progress in the imperative and the object-oriented paradigms. In the imperative paradigm we begin by considering the actions that must take place. Because each volley begins with a player serving the ball, we might start by considering a function named `Serve` that (based on the player's characteristics and perhaps a bit of probability) would compute the initial speed and direction of the ball. Next we would need to determine the path of the ball (Will it hit the net? Where will it bounce?). We might plan on placing these computations in another function named `ComputePath`. The next step might be to determine if the other player is able to return the ball, and if so we must compute the ball's new speed and direction. We might plan on placing these computations in a function named `Return`.

Continuing in this fashion, we might arrive at the modular structure depicted by the **structure chart** shown in Figure 7.3, in which functions are represented by rectangles and function dependencies (implemented by function calls) are represented by arrows. In particular, the chart indicates that the entire game is overseen by a function named `ControlGame`, and to perform its task, `ControlGame` calls on the services of the functions `Serve`, `Return`, `ComputePath`, and `UpdateScore`.

Figure 7.3 A simple structure chart



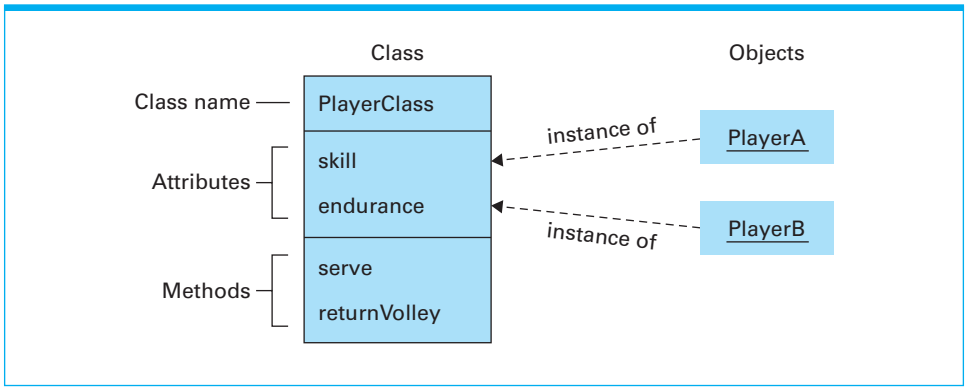
Note that the structure chart does not indicate how each function is to perform its task. Rather, it merely identifies the functions and indicates the dependencies among the functions. In reality, the function `ControlGame` might perform its task by first calling the `Serve` function, then repeatedly calling on the functions `ComputePath` and `Return` until one reports a miss, and finally calling on the services of `UpdateScore` before repeating the whole process by again calling on `Serve`.

At this stage we have obtained only a very simplistic outline of the desired program, but our point has already been made. In accordance with the imperative paradigm, we have been designing the program by considering the activities that must be performed and are therefore obtaining a design in which the modules are functions.

Let us now reconsider the program's design—this time in the context of the object-oriented paradigm. Our first thought might be that there are two players that we should represent by two objects: `PlayerA` and `PlayerB`. These objects will have the same functionality but different characteristics. (Both should be able to serve and return volleys but may do so with different skill and strength.) Thus, these objects will be instances of the same class. (Recall that in Chapter 6 we introduced the concept of a class: a template that defines the functions (called methods) and attributes (called instance variables) that are to be associated with each object.) This class, which we will call `PlayerClass`, will contain the methods `serve` and `return` that simulate the corresponding actions of the player. It will also contain attributes (such as `skill` and `endurance`) whose values reflect the player's characteristics. Our design so far is represented by the diagram in Figure 7.4. There we see that `PlayerA` and `PlayerB` are instances of the class `PlayerClass` and that this class contains the attributes `skill` and `endurance` as well as the methods `serve` and `returnVolley`. (Note that in Figure 7.4 we have underlined the names of objects to distinguish them from names of classes.)

Next we need an object to play the role of the official who determines whether the actions performed by the players are legal. For example, did the serve clear the net and land in the appropriate area of the court? For this purpose we might establish an object called `Judge` that contains the methods `evaluateServe` and `evaluateReturn`. If the `Judge` object determines a serve or return to be acceptable, play continues. Otherwise, the `Judge` sends a message to another object named `Score` to record the results accordingly.

Figure 7.4 The structure of PlayerClass and its instances



At this point the design for our tennis program consists of four objects: `PlayerA`, `PlayerB`, `Judge`, and `Score`. To clarify our design, consider the sequences of events that may occur during a volley as depicted in Figure 7.5 where we have represented the objects involved as rectangles. The figure is intended to present the communication between these objects as the result of calling the `serve` method within the object `PlayerA`. Events appear chronologically as we move down the figure. As depicted by the first horizontal arrow, `PlayerA` reports its serve to the object `Judge` by calling the method `evaluateServe`. The `Judge` then determines that the serve is good and asks `PlayerB` to return it by calling `PlayerB`'s `returnVolley` method. The volley terminates when the `Judge` determines that `PlayerA` erred and asks the object `Score` to record the results.

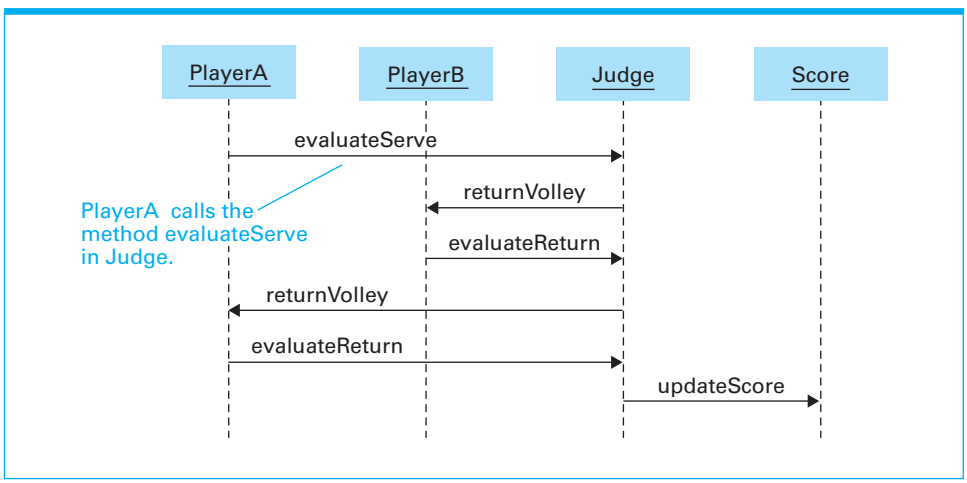
As in the case of our imperative example, our object-oriented program is very simplistic at this stage. However, we have progressed enough to see how the object-oriented paradigm leads to a modular design in which fundamental components are objects.

Coupling

We have introduced modularity as a way of producing manageable software. The idea is that any future modification will likely apply to only a few of the modules, allowing the person making the modification to concentrate on that portion of the system rather than struggling with the entire package. This, of course, depends on the assumption that changes in one module will not unknowingly affect other modules in the system. Consequently, a goal when designing a modular system should be to maximize independence among modules or, in other words, to minimize the linkage between modules (known as intermodule **coupling**). Indeed, one metric that has been used to measure the complexity of a software system (and thus obtain a means of estimating the expense of maintaining the software) is to measure its intermodule coupling.

Intermodule coupling occurs in several forms. One is **control coupling,** which occurs when a module passes control of execution to another, as in a function call. The structure chart in Figure 7.3 represents the control coupling that exists between functions. In particular, the arrow from the module `ControlGame`

Figure 7.5 The interaction between objects resulting from PlayerA's serve



to Serve indicates that the former passes control to the latter. It is also control coupling that is represented in Figure 7.5, where the arrows trace the path of control as it is passed from object to object.

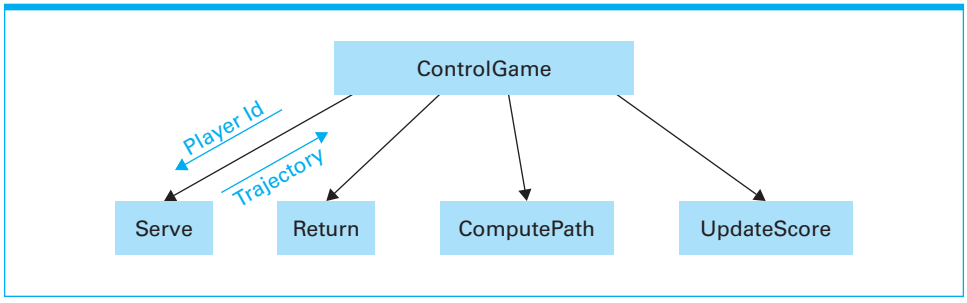
Another form of intermodule coupling is **data coupling,** which refers to the sharing of data between modules. If two modules interact with the same item of data, then modifications made to one module may affect the other, and modifications to the format of the data itself could have repercussions in both modules.

Data coupling between functions can occur in two forms. One is by explicitly passing data from one function to another in the form of parameters. Such coupling is represented in a structure chart by an arrow between the functions that is labeled to indicate the data being passed. The direction of the arrow indicates the direction in which the item is transferred. For example, Figure 7.6 is an extended version of Figure 7.3 in which we have indicated that the function ControlGame will tell the function Serve which player's characteristics are to be simulated when it calls Serve and that the function Serve will report the ball trajectory to ControlGame when Serve has completed its task.

Similar data coupling occurs between objects in an object-oriented design. For example, when PlayerA asks the object Judge to evaluate its serve (see Figure 7.5), it must pass the trajectory information to Judge. On the other hand, one of the benefits of the object-oriented paradigm is that it inherently tends to reduce data coupling between objects to a minimum. This is because the methods within an object tend to include all those functions that manipulate the object's internal data. For example, the object PlayerA will contain information regarding that player's characteristics as well as all the methods that require that information. In turn, there is no need to pass that information to other objects and thus interobject data coupling is minimized.

In contrast to passing data explicitly as parameters, data can be shared among modules implicitly in the form of **global data,** which are data items that are automatically available to all modules throughout the system, as opposed to local data items that are accessible only within a particular module unless explicitly passed to another. Most high-level languages provide ways of implementing both global and local data, but the use of global data should be employed with caution. The problem is that a person trying to modify a module that is dependent on global data may find it difficult to identify how the module in question interacts with other modules. In short, the use of global data can degrade the module's usefulness as an abstract tool.

Figure 7.6 A structure chart including data coupling



Cohesion

Just as important as minimizing the coupling between modules is maximizing the internal binding within each module. The term **cohesion** refers to this internal binding or, in other words, the degree of relatedness of a module's internal parts. To appreciate the importance of cohesion, we must look beyond the initial development of a system and consider the entire software life cycle. If it becomes necessary to make changes in a module, the existence of a variety of activities within it can confuse what would otherwise be a simple process. Thus, in addition to seeking low intermodule coupling, software designers strive for high intramodule cohesion.

A weak form of cohesion is known as **logical cohesion.** This is the cohesion within a module induced by the fact that its internal elements perform activities logically similar in nature. For example, consider a module that performs all of a system's communication with the outside world. The "glue" that holds such a module together is that all the activities within the module deal with communication. However, the topics of the communication can vary greatly. Some may deal with obtaining data, whereas others deal with reporting results.

A stronger form of cohesion is known as **functional cohesion,** which means that all the parts of the module are focused on the performance of a single activity. In an imperative design, functional cohesion can often be increased by isolating subtasks in other modules and then using these modules as abstract tools. This is demonstrated in our tennis simulation example (see again Figure 7.3) where the module `ControlGame` uses the other modules as abstract tools so that it can concentrate on overseeing the game rather than being distracted by the details of serving, returning, and maintaining the score.

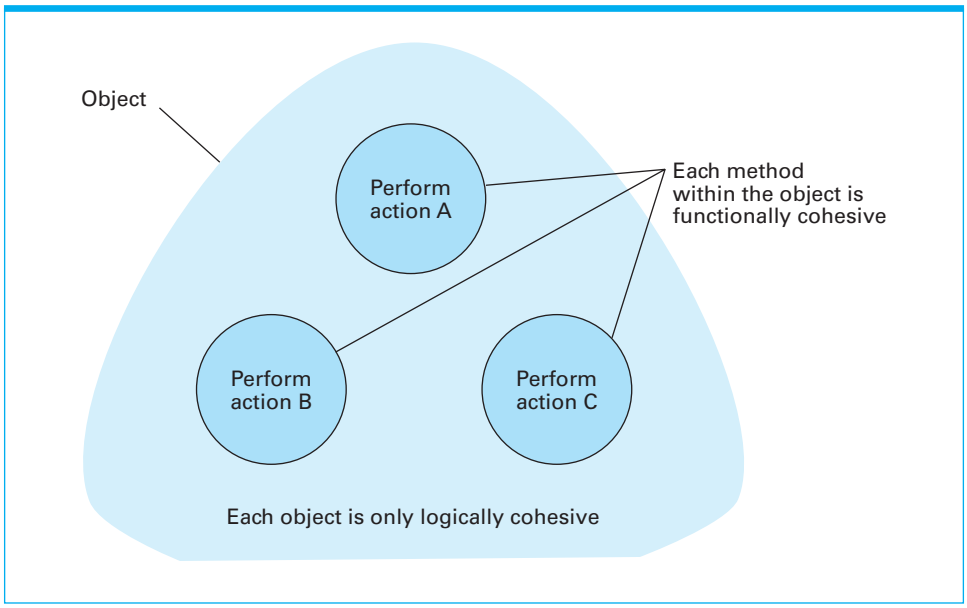
In object-oriented designs, entire objects are usually only logically cohesive because the methods within an object often perform loosely related activities—the only common bond being that they are activities performed by the same object. For example, in our tennis simulation example, each player object contains methods for serving as well as returning the ball, which are significantly different activities. Such an object would therefore be only a logically cohesive module. However, software designers should strive to make each individual method within an object functionally cohesive. That is, even though the object in its entirety is only logically cohesive, each method within an object should perform only one functionally cohesive task (Figure 7.7).

Information Hiding

One of the cornerstones of good modular design is captured in the concept of **information hiding,** which refers to the restriction of information to a specific portion of a software system. Here the term *information* should be interpreted in a broad sense, including any knowledge about the structure and contents of a program unit. As such, it includes data, the type of data structures used, encoding systems, the internal compositional structure of a module, the logical structure of a procedural unit, and any other factors regarding the internal properties of a module.

The point of information hiding is to keep the actions of modules from having unnecessary dependencies or effects on other modules. Otherwise, the validity of a module may be compromised, perhaps by errors in the development of other modules or by misguided efforts during software maintenance. If, for example, a module does not restrict the use of its internal data from other modules, then that data may become corrupted by other modules. Or, if one module is designed

Figure 7.7 Logical and functional cohesion within an object



to take advantage of another's internal structure, it could malfunction later if that internal structure is altered.

It is important to note that information hiding has two incarnations—one as a design goal, the other as an implementation goal. A module should be designed so that other modules do not need access to its internal information, and a module should be implemented in a manner that reinforces its boundaries. Examples of the former are maximizing cohesion and minimizing coupling. Examples of the latter involve the use of local variables, applying encapsulation, and using well-defined control structures.

Finally we should note that information hiding is central to the theme of abstraction and the use of abstract tools. Indeed, the concept of an abstract tool is that of a "black box" whose interior features can be ignored by its user, allowing the user to concentrate on the larger application at hand. In this sense then, information hiding corresponds to the concept of sealing the abstract tool in much the same way as a tamperproof enclosure can be used to safeguard complex and potentially dangerous electronic equipment. Both protect their users from the dangers inside as well as protect their interiors from intrusion from their users.

Components

We have already mentioned that one obstacle in the field of software engineering is the lack of prefabricated "off-the-shelf" building blocks from which large software systems can be constructed. The modular approach to software development promises hope in this regard. In particular, the object-oriented programming paradigm is proving especially useful because objects form complete, self-contained units that have clearly defined interfaces with their environments. Once an object, or more correctly a class, has been designed to fulfill a certain role, it can be used to fulfill that role in any program requiring that service. Moreover, inheritance provides a means of refining prefabricated object definitions

in those cases in which the definitions must be customized to conform to the needs of a specific application. It is not surprising, then, that the object-oriented programming languages C++, Java, and C# are accompanied by collections of prefabricated "templates" from which programmers can easily implement objects for performing certain roles. In particular, C++ is associated with the C++ Standard Template Library, the Java programming environment is accompanied by the Java Application Programmer Interface (API), and C# programmers have access to the .NET Framework Class Library.

The fact that objects and classes have the potential of providing prefabricated building blocks for software design does not mean that they are ideal. One problem is that they provide relatively small blocks from which to build. Thus, an object is actually a special case of the more general concept of a **component,** which is, by definition, a reusable unit of software. In practice, most components are based on the object-oriented paradigm and take the form of a collection of one or more objects that function as a self-contained unit.

Research in the development and use of components has led to the emerging field known as **component architecture** (also known as component-based software engineering) in which the traditional role of a programmer is replaced by a **component assembler** who constructs software systems from prefabricated components that, in many development environments, are displayed as icons in a graphical interface. Rather than be involved with the internal programming of the components, the methodology of a component assembler is to select pertinent components from collections of predefined components and then connect them, with minimal customization, to obtain the desired functionality. Indeed, a property of a well-designed component is that it can be extended to encompass features of a particular application without internal modifications.

An area where component architectures have found fertile ground is in smartphone systems. Due to the resource constraints of these devices, applications are actually a set of collaborating components, each of which provides some discrete

Software Engineering in the Real World

The following scenario is typical of the problems encountered by real-world software engineers. Company XYZ hires a software-engineering firm to develop and install a company-wide integrated software system to handle the company's data processing needs. As a part of the system produced by Company XYZ, a network of PCs is used to provide employees access to the company-wide system. Thus each employee finds a PC on his or her desk. Soon these PCs are used not only to access the new data management system but also as customizable tools with which each employee increases his or her productivity. For example, one employee may develop a spreadsheet program that streamlines that employee's tasks. Unfortunately, such customized applications may not be well designed or thoroughly tested and may involve features that are not completely understood by the employee. As the years go by, the use of these ad hoc applications becomes integrated into the company's internal business procedures. Moreover, the employees who developed these applications may be promoted, transferred, or quit the company, leaving others behind using a program they do not understand. The result is that what started out as a well-designed, coherent system can become dependent on a patchwork of poorly designed, undocumented, and error-prone applications.

function for the application. For example, each display screen within an application is usually a separate component. Behind the scenes, there may exist other service components to store and access information on a memory card, perform some continuous function (such as playing music), or access information over the Internet. Each of these components is individually started and stopped as needed to service the user efficiently; however, the application appears as a seamless series of displays and actions.

Aside from the motivation to limit the use of system resources, the component architecture of smartphones pays dividends in integration between applications. For example, Facebook (a well-known social networking system) when executed on a smartphone may use the components of the contacts application to add all Facebook friends as contacts. Furthermore, the telephony application (the one that handles the functions of the phone), may also access the contacts' components to lookup the caller of an incoming call. Thus, upon receiving a call from a Facebook friend, the friend's picture can be displayed on the phone's screen (along with his or her last Facebook post).

Questions & Exercises

1. How does a novel differ from an encyclopedia in terms of the degree of coupling between its units such as chapters, sections, or entries? What about cohesion?
2. A sporting event is often divided into units. For example, a baseball game is divided into innings and a tennis match is divided into sets. Analyze the coupling between such "modules." In what sense are such units cohesive?
3. Is the goal of maximizing cohesion compatible with minimizing coupling? That is, as cohesion increases, does coupling naturally tend to decrease?
4. Define coupling, cohesion, and information hiding.
5. Extend the structure chart in Figure 7.3 to include the data coupling between the modules `ControlGame` and `UpdateScore`.
6. Draw a diagram similar to that of Figure 7.5 to represent the sequence that would occur if `PlayerA`'s serve is ruled invalid.
7. What is the difference between a traditional programmer and a component assembler?
8. Assuming most smartphones have a number of personal organization applications (calendars, contacts, clocks, social networking, email systems, maps, etc.), what combinations of component functions would you find useful and interesting?

7.5 Tools of the Trade

In this section we investigate some of the modeling techniques and notational systems used during the analysis and design stages of software development. Several of these were developed during the years that the imperative paradigm dominated the software engineering discipline. Of these, some have found useful roles in the context of the object-oriented paradigm whereas others, such as the

structure chart (see again Figure 7.3), are specific to the imperative paradigm. We begin by considering some of the techniques that have survived from their imperative roots and then move on to explore newer object-oriented tools as well as the expanding role of design patterns.

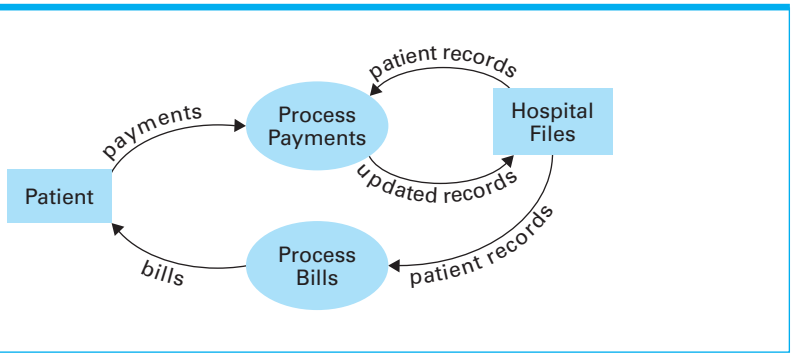
Some Old Friends

Although the imperative paradigm seeks to build software in terms of procedures or functions, a way of identifying those functions is to consider the data to be manipulated rather than the functions themselves. The theory is that by studying how data moves through a system, one identifies the points at which either data formats are altered or data paths merge and split. In turn, these are the locations at which processing occurs, and thus dataflow analysis leads to the identification of functions. A **dataflow diagram** is a means of representing the information gained from such dataflow studies. In a dataflow diagram, arrows represent data paths, ovals represent points at which data manipulation occurs, and rectangles represent data sources and stores. As an example, Figure 7.8 displays an elementary dataflow diagram representing a hospital's patient billing system. Note that the diagram shows that `Payments` (flowing from patients) and `PatientRecords` (flowing from the hospital's files) merge at the oval `ProcessPayments` from which `UpdatedRecords` flow back to the hospital's files.

Dataflow diagrams not only assist in identifying procedures during the design stage of software development, but they are also useful when trying to gain an understanding of the proposed system during the analysis stage. Indeed, constructing dataflow diagrams can serve as a means of improving communication between clients and software engineers (as the software engineer struggles to understand what the client wants and the client struggles to describe his or her expectations), and thus these diagrams continue to find applications even though the imperative paradigm has faded in popularity.

Another tool that has been used for years by software engineers is the **data dictionary,** which is a central repository of information about the data items appearing throughout a software system. This information includes the identifier used to reference each item, what constitutes valid entries in each item (Will the item always be numeric or perhaps always alphabetic? What will be the range of values that might be assigned to this item?), where the item is stored (Will the item be stored in a file or a database and, if so, which one?), and where the item is referenced in the software (Which modules will require the item's information?).

Figure 7.8 A simple dataflow diagram



One goal of constructing a data dictionary is to improve communication between the stakeholders of a software system and the software engineer charged with the task of converting all of the stakeholder needs into a requirements specification. In this context the construction of a data dictionary helps ensure that the fact that part numbers are not really numeric will be revealed during the analysis stage rather than being discovered late in the design or implementation stages. Another goal associated with the data dictionary is to establish uniformity throughout the system. It is usually by means of constructing the dictionary that redundancies and contradictions surface. For example, the item referred to as `PartNumber` in the inventory records may be the same as the `PartId` in the sales records. Moreover, the personnel department may use the item `Name` to refer to an employee while inventory records may contain the term `Name` in reference to a part.

Unified Modeling Language

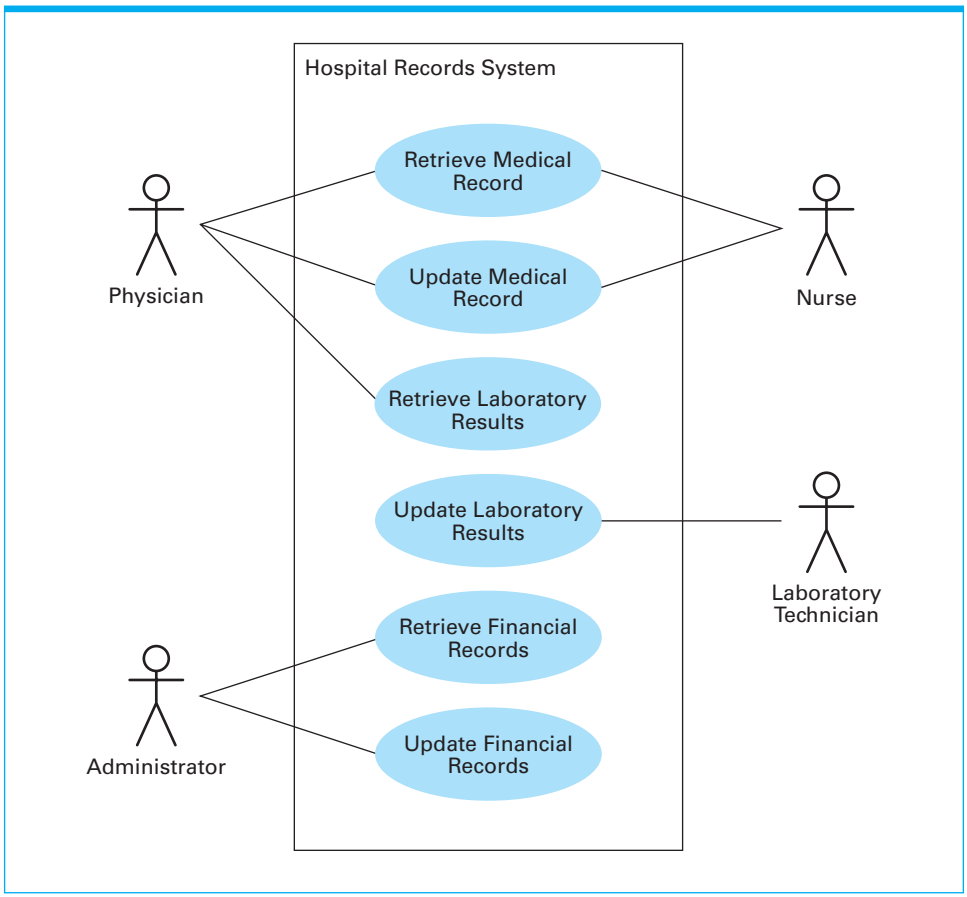
Dataflow diagrams and data dictionaries were tools in the software engineering arsenal well before the emergence of the object-oriented paradigm and have continued to find useful roles even though the imperative paradigm, for which they were originally developed, has faded in popularity. We turn now to the more modern collection of tools known as **Unified Modeling Language (UML)** that has been developed with the object-oriented paradigm in mind. The first tool that we consider within this collection, however, is useful regardless of the underlying paradigm because it attempts merely to capture the image of the proposed system from the user's point of view. This tool is the **use case diagram**—an example of which appears in Figure 7.9.

A use case diagram depicts the proposed system as a large rectangle in which interactions (called **use cases**) between the system and its users are represented as ovals and users of the system (called **actors**) are represented as stick figures (even though an actor may not be a person). Thus, the diagram in Figure 7.9 indicates that the proposed `Hospital Records System` will be used by both `Physicians` and `Nurses` to `Retrieve Medical Records`.

Whereas use case diagrams view a proposed software system from the outside, UML offers a variety of tools for representing the internal object-oriented design of a system. One of these is the **class diagram,** which is a notational system for representing the structure of classes and relationships between classes (called **associations** in UML vernacular). As an example, consider the relationships between physicians, patients, and hospital rooms. We assume that objects representing these entities are constructed from the classes `Physician`, `Patient`, and `Room`, respectively.

Figure 7.10 shows how the relationships among these classes could be represented in a UML class diagram. Classes are represented by rectangles and associations are represented by lines. Association lines may or may not be labeled. If they are labeled, a bold arrowhead can be used to indicate the direction in which the label should be read. For example, in Figure 7.10 the arrowhead following the label `cares for` indicates that a physician `cares for` a patient rather than a patient `cares for` a physician. Sometimes association lines are given two labels to provide terminology for reading the association in either direction. This is exemplified in Figure 7.10 in the association between the classes `Patient` and `Room`.

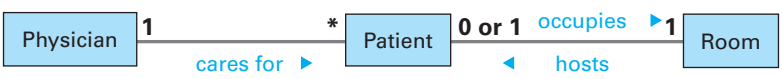
Figure 7.9 A simple use case diagram



In addition to indicating associations between classes, a class diagram can also convey the multiplicities of those associations. That is, it can indicate how many instances of one class may be associated with instances of another. This information is recorded at the ends of the association lines. In particular, Figure 7.10 indicates that each patient can occupy one room and each room can host zero or one patient. (We are assuming that each room is a private room.) An asterisk is used to indicate an arbitrary nonnegative number. Thus, the asterisk in Figure 7.10 indicates that each physician may care for many patients, whereas the 1 at the physician end of the association means that each patient is cared for by only one physician. (Our design considers only the role of primary physicians.)

For the sake of completeness, we should note that association multiplicities occur in three basic forms: one-to-one relationships, one-to-many relationships,

Figure 7.10 A simple class diagram



and many-to-many relationships as summarized in Figure 7.11. A **one-to-one relationship** is exemplified by the association between patients and occupied private rooms in that each patient is associated with only one room and each room is associated with only one patient. A **one-to-many relationship** is exemplified by the association between physicians and patients in that one physician is associated with many patients and each patient is associated with one (primary) physician. A **many-to-many relationship** would occur if we included consulting physicians in the physician–patient relationship. Then each physician could be associated with many patients and each patient could be associated with many physicians.

In an object-oriented design it is often the case that one class represents a more specific version of another. In those situations we say that the latter class is a generalization of the former. UML provides a special notation for representing generalizations. An example is given in Figure 7.12, which depicts the generalizations among the classes `MedicalRecord`, `SurgicalRecord`, and `OfficeVisitRecord`. There the associations between the classes are represented by arrows with hollow arrowheads, which is the UML notation for associations that are generalizations. Note that each class is represented by a rectangle containing the name, attributes, and methods of the class in the format introduced in Figure 7.4. This is UML's way of representing the internal characteristics of a class in a class diagram. The information portrayed in Figure 7.12 is that the class `MedicalRecord` is a generalization of the class `SurgicalRecord` as well as a generalization of `OfficeVisitRecord`. That is, the classes `SurgicalRecord` and `OfficeVisitRecord` contain all the features of the class `MedicalRecord` plus those features explicitly listed inside their appropriate rectangles. Thus, both the `SurgicalRecord` and the `OfficeVisitRecord` classes contain patient, doctor, and date of record, but the `SurgicalRecord` class also contains surgical procedure, hospital, discharge date, and the ability to discharge a patient, whereas the `OfficeVisitRecord` class contains symptoms and diagnosis. All three

Figure 7.11 One-to-one, one-to-many, and many-to-many relationships between entities of types X and Y

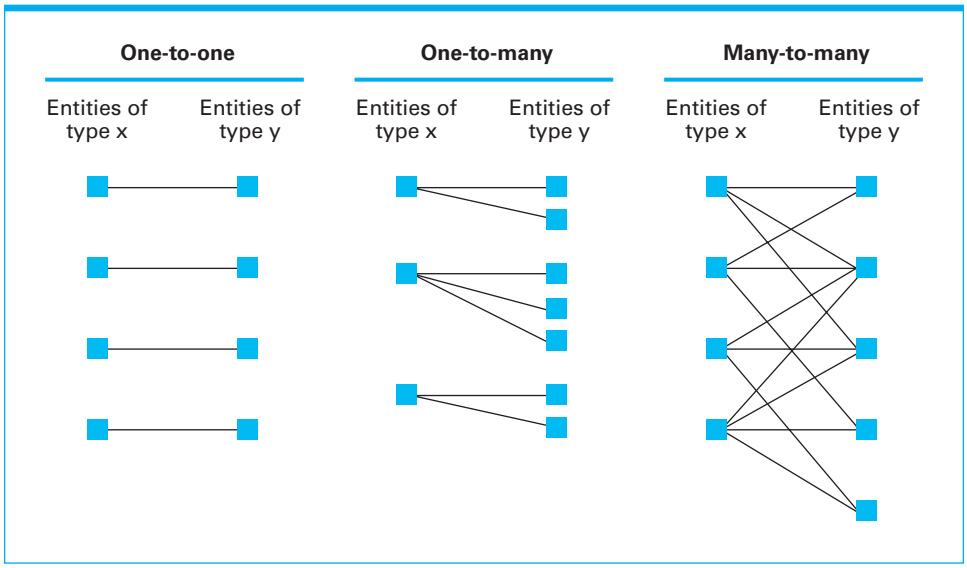
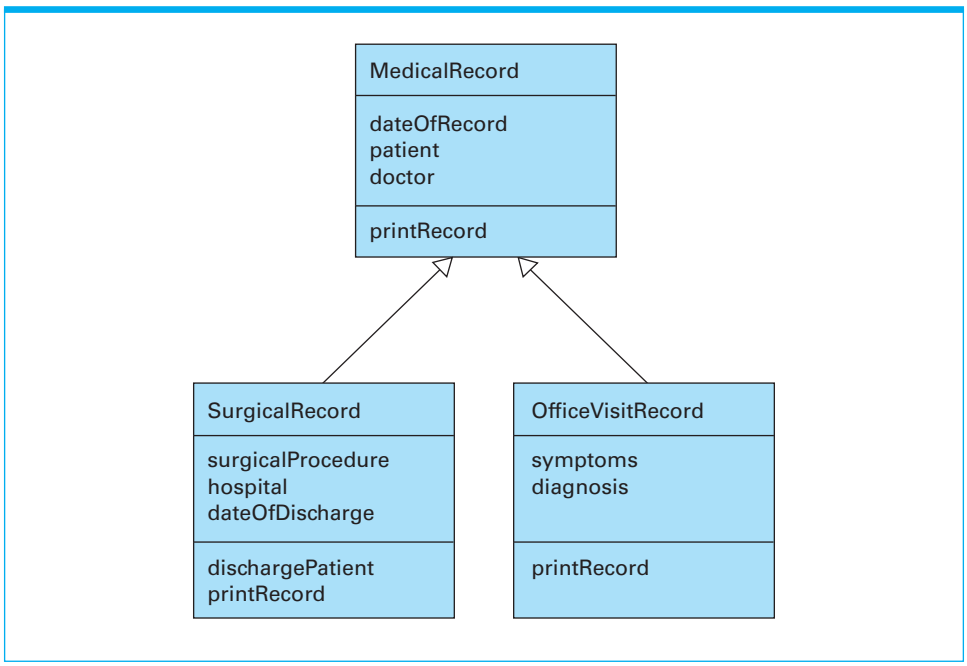


Figure 7.12 A class diagram depicting generalizations



classes have the ability to print the medical record. The `printRecord` method in `SurgicalRecord` and `OfficeVisitRecord` are specializations of the `printRecord` method in `MedicalRecord`, each of which will print the information specific to its class.

Recall from Chapter 6 (Section 6.5) that a natural way of implementing generalizations in an object-oriented programming environment is to use inheritance. However, many software engineers caution that inheritance is not appropriate for all cases of generalization. The reason is that inheritance introduces a strong degree of coupling between the classes—a coupling that may not be desirable later in the software's life cycle. For example, because changes within a class are reflected automatically in all the classes that inherit from it, what may appear to be minor modifications during software maintenance can lead to unforeseen consequences. As an example, suppose a company opened a recreation facility for its employees, meaning that all people with membership in the recreation facility are employees. To develop a membership list for this facility, a programmer could use inheritance to construct a `RecreationMember` class from a previously defined `Employee` class. But, if the company later prospers and decides to open the recreation facility to dependents of employees or perhaps company retirees, then the embedded coupling between the `Employee` class and the `RecreationMember` class would have to be severed. Thus, inheritance should not be used merely for convenience. Instead, it should be restricted to those cases in which the generalization being implemented is immutable.

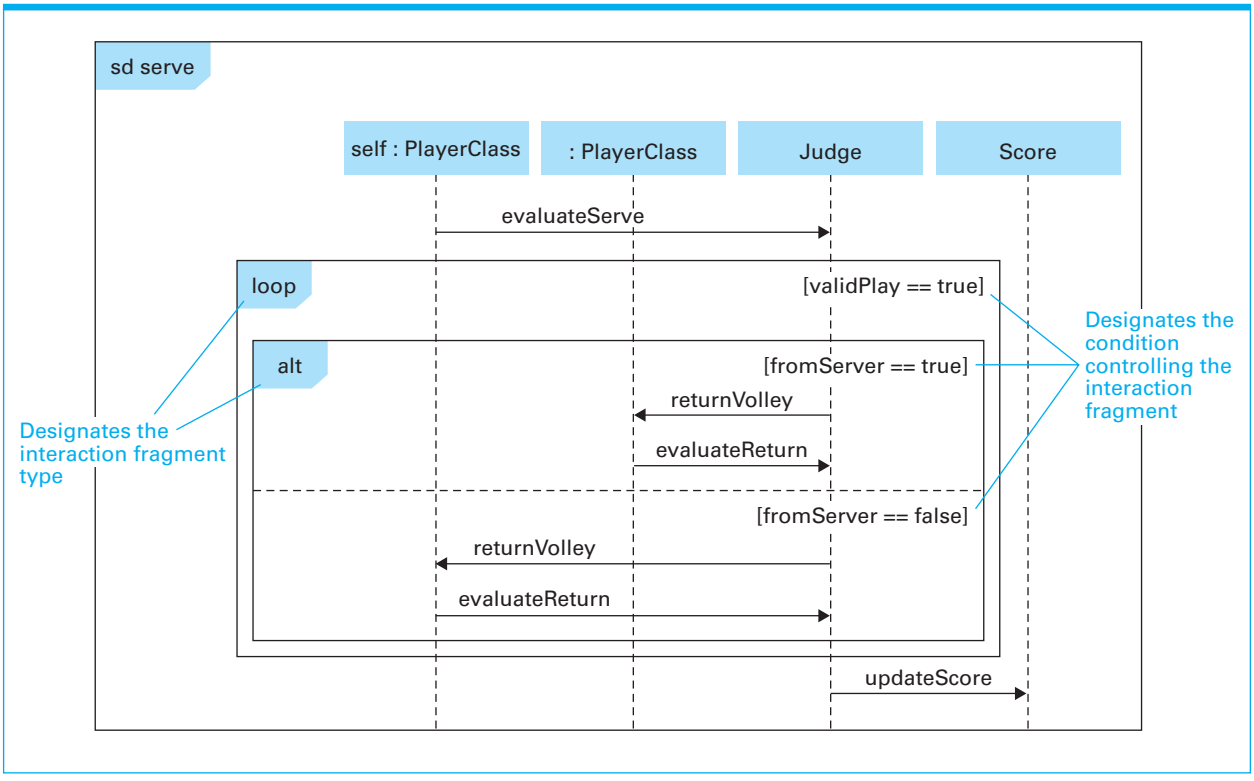
Class diagrams represent static features of a program's design. They do not represent sequences of events that occur during execution. To express such dynamic features, UML provides a variety of diagram types that are collectively known as **interaction diagrams.** One type of interaction diagram is the **sequence diagram** that depicts the communication between the individuals (such as actors,

complete software components, or individual objects) that are involved in performing a task. These diagrams are similar to Figure 7.5 in that they represent the individuals by rectangles with dashed lines extending downward. Each rectangle together with its dashed line is called a **life line.** Communication between the individuals is represented by labeled arrows connecting the appropriate life line, where the label indicates the action being requested. These arrows appear chronologically as the diagram is read from top to bottom. The communication that occurs when an individual completes a requested task and returns control back to the requesting individual, as in the traditional return from a procedure, is represented by an unlabeled arrow pointing back to the original life line.

Thus, Figure 7.5 is essentially a sequence diagram. However, the syntax of Figure 7.5 alone has several shortcomings. One is that it does not allow us to capture the symmetry between the two players. We must draw a separate diagram to represent a volley starting with a serve from `PlayerB`, even though the interaction sequence is very similar to that when `PlayerA` serves. Moreover, whereas Figure 7.5 depicts only a specific volley, a general volley may extend indefinitely. Formal sequence diagrams have techniques for capturing these variations in a single diagram, and although we do not need to study these in detail, we should still take a brief look at the formal sequence diagram shown in Figure 7.13, which depicts a general volley based on our tennis game design.

Note also that Figure 7.13 demonstrates that an entire sequence diagram is enclosed in a rectangle (called a **frame**). In the upper left-hand corner of the frame is a pentagon containing the characters *sd* (meaning "sequence diagram")

Figure 7.13 A sequence diagram depicting a generic volley



followed by an identifier. This identifier may be a name identifying the overall sequence or, as in Figure 7.13, the name of the method that is called to initiate the sequence. Note that in contrast to Figure 7.5, the rectangles representing the players in Figure 7.13 do not refer to specific players but merely indicate that they represent objects of the "type" `PlayerClass`. One of these is designated *self,* meaning that it is the one whose serve method is activated to initiate the sequence.

The other point to make regarding Figure 7.13 deals with the two inner rectangles. These are **interaction fragments,** which are used to represent alternative sequences within one diagram. Figure 7.13 contains two interaction fragments. One is labeled "loop," the other is labeled "alt." These are essentially the while and if-else structures that we first encountered in Python in Section 5.2. The "loop" interaction fragment indicates that the events within its boundaries are to be repeated as long as the `Judge` object determines that the value of `validPlay` is true. The "alt" interaction fragment indicates that one of its alternatives is to be performed depending on whether the value of `fromServer` is true or false.

Finally, although they are not a part of UML, it is appropriate at this point to introduce the role of **CRC (class-responsibility-collaboration) cards** because they play an important role in validating object-oriented designs. A CRC card is simply a card, such as an index card, on which the description of an object is written. The methodology of CRC cards is for the software designer to produce a card for each object in a proposed system and then to use the cards to represent the objects in a simulation of the system—perhaps on a desktop or via a "theatrical" experiment in which each member of the design team holds a card and plays the role of the object as described by that card. Such simulations (often called **structured walkthroughs**) have been found useful in identifying flaws in a design prior to the design's implementation.

Design Patterns

An increasingly powerful tool for software engineers is the growing collection of design patterns. A **design pattern** is a predeveloped model for solving a recurring problem in software design. For example, the Adapter pattern provides a solution to a problem that often occurs when constructing software from prefabricated modules. In particular, a prefabricated module may have the functionality needed to solve the problem at hand but may not have an interface that is compatible with the current application. In such cases the Adapter pattern provides a standard approach to "wrapping" that module inside another module that translates between the original module's interface and the outside world, thus allowing the original, prefabricated module to be used in the application.

Another well-established design pattern is the Decorator pattern. It provides a means of designing a system that performs different combinations of the same activities depending on the situation at the time. Such systems can lead to an explosion of options that, without careful design, can result in enormously complex software. However, the Decorator pattern provides a standardized way of implementing such systems that leads to a manageable solution.

The identification of recurring problems as well as the creation and cataloging of design patterns for solving them is an ongoing process in software engineering. The goal, however, is not merely to find solutions to design problems but to find high-quality solutions that provide flexibility later in the software life cycle. Thus,

considerations of good design principles such as minimizing coupling and maximizing cohesion play an important role in the development of design patterns.

The results of progress in design pattern development are reflected in the library of tools provided in today's software development packages such as the Java programming environments provided by Oracle and the .NET Framework provided by Microsoft. Indeed, many of the templates found in these tool kits are essentially design pattern skeletons that lead to ready-made, high-quality solutions to design problems.

In closing, we should mention that the emergence of design patterns in software engineering is an example of how diverse fields can contribute to each other. The origins of design patterns lie in the research of Christopher Alexander in traditional architecture. His goal was to identify features that contribute to high-quality architectural designs for buildings or building complexes and then to develop design patterns that incorporated those features. Today, many of his ideas have been incorporated into software design and his work continues to be an inspiration for many software engineers.

Questions & Exercises

1. Draw a dataflow diagram representing the flow of data that occurs when a patron checks a book out of a library.
2. Draw a use case diagram of a library records system.
3. Draw a class diagram representing the relationship between travelers and the hotels in which they stay.
4. Draw a class diagram representing the fact that a person is a generalization of an employee. Include some attributes that might belong to each.
5. Convert Figure 7.5 into a complete sequence diagram.
6. What role in the software engineering process do design patterns play?

7.6 Quality Assurance

The proliferation of software malfunctions, cost overruns, and missed deadlines demands that methods of software quality control be improved. In this section we consider some of the directions being pursued in this endeavor.

The Scope of Quality Assurance

In the early years of computing, the problem of producing quality software focused on removing programming errors that occurred during implementation. Later in this section we will discuss the progress that has been made in this direction. However, today, the scope of software quality control extends far beyond the debugging process, with branches including the improvement of software engineering procedures, the development of training programs that in many cases lead to certification, and the establishment of standards on which sound software engineering can be based. In this regard, we have already noted

the role of organizations such as ISO, IEEE, and ACM in improving professionalism and establishing standards for assessing quality control within software development companies. A specific example is the ISO 9000 series of standards, which address numerous industrial activities such as design, production, installation, and servicing. Another example is ISO/IEC 15504, which is a set of standards developed jointly by the ISO and the International Electrotechnical Commission (IEC).

Many major software contractors now require that the organizations they hire to develop software meet such standards. As a result, software development companies are establishing **software quality assurance (SQA)** groups, which are charged with overseeing and enforcing the quality control systems adopted by the organization. Thus, in the case of the traditional waterfall model, the SQA group would be charged with the task of approving the software requirements specification before the design stage began or approving the design and its related documents before implementation was initiated.

Several themes underlie today's quality control efforts. One is record keeping. It is paramount that each step in the development process be accurately documented for future reference. However, this goal conflicts with human nature. At issue is the temptation to make decisions or change decisions without updating the related documents. The result is the chance that records will be incorrect and hence their use at future stages will be misleading. Herein lies an important benefit of CASE tools. They make such tasks as redrawing diagrams and updating data dictionaries much easier than with manual methods. Consequently, updates are more likely to be made and the final documentation is more likely to be accurate. (This example is only one of many instances in which software engineering must cope with the faults of human nature. Others include the inevitable personality conflicts, jealousies, and ego clashes that arise when people work together.)

Another quality-oriented theme is the use of reviews in which various parties involved in a software development project meet to consider a specific topic. Reviews occur throughout the software development process, taking the form of requirements reviews, design reviews, and implementation reviews. They may appear as a prototype demonstration in the early stages of requirements analysis,

System Design Tragedies

The need for good design disciplines is exemplified by the problems encountered in the Therac-25, which was a computer-based electron-accelerator radiation-therapy system used by the medical community in the middle 1980s. Flaws in the machine's design contributed to six cases of radiation overdose—three of which resulted in death. The flaws included (1) a poor design for the machine's interface that allowed the operator to begin radiation before the machine had adjusted for the proper dosage, and (2) poor coordination between the design of the hardware and software that resulted in the absence of certain safety features.

In more recent cases, poor design has led to widespread power outages, severance of telephone service, major errors in financial transactions, loss of space probes, and disruption of the Internet. You can learn more about such problems through the Risks Forum at `http://catless.ncl.ac.uk/Risks`.

as a structured walkthrough among members of the software design team, or as coordination among programmers who are implementing related portions of the design. Such reviews, on a recurring basis, provide communication channels through which misunderstandings can be avoided and errors can be corrected before they grow into disasters. The significance of reviews is exemplified by the fact that they are specifically addressed in the IEEE Standard for Software Reviews, known as IEEE 1028.

Some reviews are pivotal in nature. An example is the review between representatives of a project's stakeholders and the software development team at which the final software requirements specification is approved. Indeed, this approval marks the end of the formal requirements analysis phase and is the basis on which the remaining development will progress. However, all reviews are significant, and for the sake of quality control, they should be documented as part of the ongoing record maintenance process.

Software Testing

Whereas software quality assurance is now recognized as a subject that permeates the entire development process, testing and verification of the programs themselves continues to be a topic of research. In Section 5.6 we discussed techniques for verifying the correctness of algorithms in a mathematically rigorous manner but concluded that most software today is "verified" by means of testing. Unfortunately, such testing is inexact at best. We cannot guarantee that a piece of software is correct via testing unless we run enough tests to exhaust all possible scenarios. But, even in simple programs, there may be billions of different paths that could potentially be traversed. Thus, testing all possible paths within a complex program is an impossible task.

On the other hand, software engineers have developed testing methodologies that improve the odds of revealing errors in software with a limited number of tests. One of these is based on the observation that errors in software tend to be clumped. That is, experience has shown that a small number of modules within a large software system tend to be more problematic than the rest. Thus, by identifying these modules and testing them more thoroughly, more of the system's errors can be discovered than if all modules were tested in a uniform, less-thorough manner. This is an instance of the proposition known as the **Pareto principle,** in reference to the economist and sociologist Vilfredo Pareto (1848 – 1923) who observed that a small part of Italy's population controlled most of Italy's wealth. In the field of software engineering, the Pareto principle states that results can often be increased most rapidly by applying efforts in a concentrated area.

Another software testing methodology, called **basis path testing,** is to develop a set of test data that insures that each instruction in the software is executed at least once. Techniques using an area of mathematics known as graph theory have been developed for identifying such sets of test data. Thus, although it may be impossible to ensure that every path through a software system is tested, it is possible to ensure that every statement within the system is executed at least once during the testing process.

Techniques based on the Pareto principle and basis path testing rely on knowledge of the internal composition of the software being tested. They therefore fall within the category called **glass-box testing**—meaning that the software tester is aware of the interior structure of the software and uses this knowledge

when designing the test. In contrast is the category called **black-box testing,** which refers to tests that do not rely on knowledge of the software's interior composition. In short, black-box testing is performed from the user's point of view. In black-box testing, one is not concerned with how the software goes about its task but merely with whether the software performs correctly in terms of accuracy and timeliness.

An example of black-box testing is the technique, called **boundary value analysis,** that consists of identifying ranges of data, called **equivalence classes,** over which the software should perform in a similar manner and then testing the software on data close to the edge of those ranges. For example, if the software is supposed to accept input values within a specified range, then the software would be tested at the lowest and highest values in that range, or if the software is supposed to coordinate multiple activities, then the software would be tested on the largest possible collection of activities. The underlying theory is that by identifying equivalence classes, the number of test cases can be minimized because correct operation for a few examples within an equivalence class tends to validate the software for the entire class. Moreover, the best chance of identifying an error within a class is to use data at the class edges.

Another methodology that falls within the black-box category is **beta testing** in which a preliminary version of the software is given to a segment of the intended audience with the goal of learning how the software performs in real-life situations before the final version of the product is solidified and released to the market. (Similar testing performed at the developer's site is called **alpha testing.**) The advantages of beta testing extend far beyond the traditional discovery of errors. General customer feedback (both positive and negative) is obtained that may assist in refining market strategies. Moreover, early distribution of beta software assists other software developers in designing compatible products. For example, in the case of a new operating system for the PC market, the distribution of a beta version encourages the development of compatible utility software so that the final operating system ultimately appears on store shelves surrounded by companion products. Moreover, the existence of beta testing can generate a feeling of anticipation within the marketplace—an atmosphere that increases publicity and sales.

Questions & Exercises

1. What is the role of the SQA group within a software development organization?
2. In what ways does human nature work against quality assurance?
3. Identify two themes that are applied throughout the development process to enhance quality.
4. When testing software, is a successful test one that does or does not find errors?
5. What techniques would you propose using to identify the modules within a system that should receive more thorough testing than others?
6. What would be a good test to perform on a software package that was designed to sort a list of no more than 100 entries?

7.7 Documentation

A software system is of little use unless people can learn to use and maintain it. Hence, documentation is an important part of a final software package, and its development is, therefore, an important topic in software engineering.

Software documentation serves three purposes, leading to three categories of documentation: user documentation, system documentation, and technical documentation. The purpose of **user documentation** is to explain the features of the software and describe how to use them. It is intended to be read by the user of the software and is therefore expressed in the terminology of the application.

Today, user documentation is recognized as an important marketing tool. Good user documentation combined with a well-designed user interface makes a software package accessible and thus increases its sales. Recognizing this, many software developers hire technical writers to produce this part of their product, or they provide preliminary versions of their products to independent authors so that how-to books are available in book stores when the software is released to the public.

User documentation traditionally takes the form of a physical book or booklet, but in many cases the same information is included as part of the software itself. This allows a user to refer to the documentation while using the software. In this case the information may be broken into small units, sometimes called help packages, that may appear on the display screen automatically if the user dallies too long between commands.

The purpose of **system documentation** is to describe the software's internal composition so that the software can be maintained later in its life cycle. A major component of system documentation is the source version of all the programs in the system. It is important that these programs be presented in a readable format, which is why software engineers support the use of well-designed, high-level programming languages, the use of comment statements for annotating a program, and a modular design that allows each module to be presented as a coherent unit. In fact, most companies that produce software products have adopted conventions for their employees to follow when writing programs. These include indentation conventions for organizing a program on the written page; naming conventions that establish a distinction between names of different program constructs such as variables, constants, objects, and classes; and documentation conventions to ensure that all programs are sufficiently documented. Such conventions establish uniformity throughout a company's software, which ultimately simplifies the software maintenance process.

Another component of system documentation is a record of the design documents including the software requirements specification and records showing how these specifications were obtained during design. This information is useful during software maintenance because it indicates why the software was implemented as it was—information that reduces the chance that changes made during maintenance will disrupt the integrity of the system.

The purpose of **technical documentation** is to describe how a software system should be installed and serviced (such as adjusting operating parameters, installing updates, and reporting problems back to the software's developer). Technical documentation of software is analogous to the documentation provided to mechanics in the automobile industry. This documentation does not discuss how the car was designed and constructed (analogous to system documentation), nor does it explain how to drive the car and operate its heating/cooling system

(analogous to user documentation). Instead, it describes how to service the car's components—for example, how to replace the transmission or how to track down an intermittent electrical problem.

The distinction between technical documentation and user documentation is blurred in the PC arena because the user is often the person who also installs and services the software. However, in multiuser environments, the distinction is sharper. Therefore, technical documentation is intended for the system administrator who is responsible for servicing all the software under his or her jurisdiction, allowing the users to access the software packages as abstract tools.

Questions & Exercises

1. In what forms can software be documented?
2. At what phase (or phases) in the software life cycle is system documentation prepared?
3. Which is more important, a program or its documentation?

7.8 The Human-Machine Interface

Recall from Section 7.2 that one of the tasks during requirements analysis is to define how the proposed software system will interact with its environment. In this section we consider topics associated with this interaction when it involves communicating with humans—a subject with profound significances. After all, humans should be allowed to use a software system as an abstract tool. This tool should be easy to apply and designed to minimize (ideally eliminate) communication errors between the system and its human users. This means that the system's interface should be designed for the convenience of humans rather than merely the expediency of the software system.

The importance of good interface design is further emphasized by the fact that a system's interface is likely to make a stronger impression on a user than any other system characteristic. After all, a human tends to view a system in terms of its usability, not in terms of how cleverly it performs its internal tasks. From a human's perspective, the choice between two competing systems is likely to be based on the systems' interfaces. Thus, the design of a system's interface can ultimately be the determining factor in the success or failure of a software engineering project.

For these reasons, the human-machine interface has become an important concern in the requirements stage of software development projects and is a growing subfield of software engineering. In fact, some would argue that the study of human-machine interfaces is an entire field in its own right.

A beneficiary of research in this field is the smartphone interface. In order to attain the goal of a convenient pocket-sized device, elements of the traditional human-machine interface (full-sized keyboard, mouse, scroll bars, menus) are being replaced with new approaches; such as gestures performed on a touch screen, voice commands, and virtual keyboards with advanced autocompletion of words and phrases. While these represent significant progress, most smartphone users would argue that there is plenty of room for further innovation.

Research in human-machine interface design draws heavily from the areas of engineering called **ergonomics,** which deals with designing systems that harmonize with the physical abilities of humans, and **cognetics,** which deals with designing systems that harmonize with the mental abilities of humans. Of the two, ergonomics is the better understood, largely because humans have been interacting physically with machines for centuries. Examples are found in ancient tools, weaponry, and transportation systems. Much of this history is self-evident; however, at times the application of ergonomics has been counterintuitive. An often-cited example is the design of the typewriter keyboard (now reincarnated as the computer keyboard) in which the keys were intentionally arranged to reduce a typist's speed so that the mechanical system of levers used in the early machines would not jam.

Mental interaction with machines, in contrast, is a relatively new phenomenon, and thus, it is cognetics that offers the higher potential for fruitful research and enlightening insights. Often the findings are interesting in their subtlety. For example, humans form habits—a trait that, on the surface, is good because it can increase efficiency. But, habits can also lead to errors, even when the design of an interface intentionally addresses the problem. Consider the process of a human asking a typical operating system to delete a file. To avoid unintentional deletions, most interfaces respond to such a request by asking the user to confirm the request—perhaps via a message such as, "Do you really want to delete this file?" At first glance, this confirmation requirement would seem to resolve any problem of unintentional deletions. However, after using the system for an extended period, a human develops the habit of automatically answering the question with "yes." Thus, the task of deleting a file ceases to be a two-step process consisting of a delete command followed by a thoughtful response to a question. Instead, it becomes a one-step "delete-yes" process, meaning that by the time the human realizes that an incorrect delete request has been submitted, the request has already been confirmed and the deletion has occurred.

The formation of habits may also cause problems when a human is required to use several application software packages. The interfaces of such packages may be similar yet different. Similar user actions may result in different system responses or similar system responses may require different user actions. In these cases habits developed in one application may lead to errors in the other applications.

Another human characteristic that concerns researchers in human-machine interface design is the narrowness of a human's attention, which tends to become more focused as the level of concentration increases. As a human becomes more engrossed in the task at hand, breaking that focus becomes more difficult. In 1972, a commercial aircraft crashed because the pilots became so absorbed with a landing gear problem (actually, with the process of changing the landing gear indicator light bulb) that they allowed the plane to fly into the ground, even though warnings were sounding in the cockpit.

Less critical examples appear routinely in PC interfaces. For example, a "Caps Lock" light is provided on most keyboards to indicate that the keyboard is in "Caps Lock" mode (i.e., the "Caps Lock" key has been pressed). However, if the key is accidentally pressed, a human rarely notices the status of the light until strange characters begin to appear on the display screen. Even then, the user often puzzles over the predicament for a while until realizing the cause of the problem. In a sense, this is not surprising—the light on the keyboard is not in the user's field of view. However, users often fail to notice indicators placed directly in their

line of sight. For example, users can become so engaged in a task that they fail to observe changes in the appearance of the cursor on the display screen, even though their task involves watching the cursor.

Still another human characteristic that must be anticipated during interface design is the mind's limited capacity to deal with multiple facts simultaneously. In an article in *Psychological Review* in 1956, George A. Miller reported research indicating that the human mind is capable of dealing with only about seven details at once. Thus, it is important that an interface be designed to present all the relevant information when a decision is required rather than to rely on the human user's memory. In particular, it would be poor design to require that a human remember precise details from previous screen images. Moreover, if an interface requires extensive navigation among screen images, a human can get lost in the maze. Thus, the content and arrangement of screen images becomes an important design issue.

Although applications of ergonomics and cognetics give the field of human-machine interface design a unique flavor, the field also encompasses many of the more traditional topics of software engineering. In particular, the search for metrics is just as important in the field of interface design as it is in the more traditional areas of software engineering. Interface characteristics that have been subjected to measurement include the time required to learn an interface, the time required to perform tasks via the interface, the rate of user-interface errors, the degree to which a user retains proficiency with the interface after periods of nonuse, and even such subjective traits as the degree to which users like the interface.

The **GOMS** (rhymes with "Toms") model, originally introduced in 1954, is representative of the search for metrics in the field of human-machine interface design. The model's underlying methodology is to analyze tasks in terms of user goals (such as delete a word from a text), operators (such as click the mouse button), methods (such as double-click the mouse button and press the delete key), and selection rules (such as choose between two methods of accomplishing the same goal). This, in fact, is the origin of the acronym GOMS—goals, operators, methods, and selection rules. In short, GOMS is a methodology that allows the actions of a human using an interface to be analyzed as sequences of elementary steps (press a key, move the mouse, make a decision). The performance of each elementary step is assigned a precise time period, and thus, by adding the times assigned to the steps in a task, GOMS provides a means of comparing different proposed interfaces in terms of the time each would require when performing similar tasks.

Understanding the technical details of systems such as GOMS is not the purpose of our current study. The point in our case is that GOMS is founded on features of human behavior (moving hands, making decisions, and so on). In fact, the development of GOMS was originally considered a topic in psychology. Thus, GOMS reemphasizes the role that human characteristics play in the field of human-machine interface design, even in the topics that are carryovers from traditional software engineering.

The design of human-machine interfaces promises to be an active field of research in the foreseeable future. Many issues dealing with today's GUIs are yet unresolved, and a multitude of additional problems lurk in the use of three-dimensional interfaces that are now on the horizon. Indeed, because these interfaces promise to combine audio and tactile communication with three-dimensional vision, the scope of potential problems is enormous.

Questions & Exercises

1. a. Identify an application of ergonomics in the field of human-computer interface design.
   b. Identify an application of cognetics in the field of human-computer interface design.
2. A notable difference in the human-computer interface of a smartphone from that of a desktop computer are the techniques used to scroll a portion of the display. On a desktop, scroll is typically achieved by dragging the mouse on scrollbars displayed on the right and bottom sides of the scrolling region, or by using scroll wheels built into the mouse. On the other hand, scroll bars are often not used on a smartphone. (If used, they appear as thin lines to indicate what portion of the underlying display is currently visible.) Scrolling is thus achieved by the gesture of a sliding touch across the display screen.
   a. Based on ergonomics, what arguments can be made in support of this difference?
   b. Based on cognetics, what arguments can be made in support of this difference?
3. What distinguishes the field of human-machine interface design from the more traditional field of software engineering?
4. Identify three human characteristics that should be considered when designing a human-machine interface.

7.9 Software Ownership and Liability

Most would agree that a company or individual should be allowed to recoup, and profit from, the investment needed to develop quality software. Otherwise, it is unlikely that many would be willing to undertake the task of producing the software our society desires. In short, software developers need a level of ownership over the software they produce.

Legal efforts to provide such ownership fall under the category of **intellectual property** law, much of which is based on the well-established principles of copyright and patent law. Indeed, the purpose of a copyright or patent is to allow the developer of a product to release that product (or portions thereof) to intended parties while protecting his or her ownership rights. As such, the developer of a product (whether an individual or a corporation) will assert his or her ownership by including a copyright statement in all produced works; including requirement specifications, design documents, source code, test plans, and in some visible place within the final product. A copyright notice clearly identifies ownership, the personnel authorized to use the work, and other restrictions. Furthermore, the rights of the developer are formally expressed in legal terms in a **software license.**

A software license is a legal agreement between the owner and user of a software product that grants the user certain permissions to use the product without

transferring ownership rights to the intellectual property. These agreements spell out, to a fine level of detail, the rights and obligations of both parties. Thus, it is important to carefully read and understand the terms of the software license before installing and using a software product.

While copyrights and software license agreements provide legal avenues to inhibit outright copying and unauthorized use of software, they are generally insufficient to prevent another party from independently developing a product with a nearly identical function. It is sad that over the years there have been many occasions where the developer of a truly revolutionary software product was unable to capitalize fully on his or her invention (two notable examples are spreadsheets and web browsers). In most of these cases, another company was successful in developing a competitive product that secured a dominant share of the market. A legal path to prevent this intrusion by a competitor is found in patent law.

Patent laws were established to allow an inventor to benefit commercially from an invention. To obtain a patent, the inventor must disclose the details of the invention and demonstrate that it is new, useful, and not obvious to others with similar backgrounds (a requirement that can be quite challenging for software). If a patent is granted, the inventor is given the right to prevent others from making, using, selling, or importing the invention for a limited period of time, which is typically 20 years from the date the patent application was filed.

One drawback to the use of patents is that the process to obtain a patent is expensive and time-consuming, often involving several years. During this time a software product could become obsolete, and until the patent is granted the applicant has only questionable authority to exclude others from appropriating the product.

The importance of recognizing copyrights, software licenses, and patents is paramount in the software engineering process. When developing a software product, software engineers often choose to incorporate software from other products, whether it be an entire product, subset of components, or even portions of source code downloaded over the Internet. However, failure to honor intellectual property rights during this process may lead to huge liabilities and consequences. For example, in 2004, a little-known company, NPT Inc., successfully won a lawsuit against Research In Motion (RIM—the makers of the BlackBerry smartphones) for patent infringement of a few key technologies embedded in RIM's email systems. The judgment included an injunction to suspend email services to all BlackBerry users in the United States! RIM eventually reached an agreement to pay NPT a total of $612.5 million, thereby averting a shutdown.

Finally, we should address the issue of liability. To protect themselves against liability, software developers often include disclaimers in the software licenses that state the limitations of their liability. Such statements as "In no event will Company X be liable for any damages arising out of the use of this software" are common. Courts, however, rarely recognize a disclaimer if the plaintiff can show negligence on the part of the defendant. Thus liability cases tend to focus on whether the defendant used a level of care compatible with the product being produced. A level of care that might be deemed acceptable in the case of developing a word processing system may be considered negligent when developing software to control a nuclear reactor. Consequently, one of the best defenses against software liability claims is to apply sound software engineering principles during the software's development, to use a level of care compatible with the software's application, and to produce and maintain records that validate these endeavors.

Questions & Exercises

1. What is the significance of a copyright notice in requirement specifications, design documents, source code, and the final product?
2. In what ways are copyright and patent laws designed to benefit society?
3. To what extent are disclaimers not recognized by the courts?

Chapter Review Problems

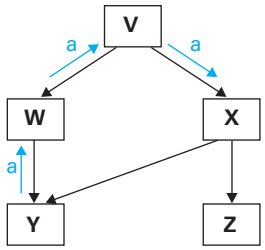
(Asterisked problems are associated with optional sections.)

1. Give an example of how efforts in the development of software can pay dividends later in software maintenance.
2. What is throwaway prototyping?
3. Explain how the lack of metrics for measuring certain software properties affects the software engineering discipline.
4. Would you expect a metric for measuring the complexity of a software system to be cumulative in the sense that the complexity of a complete system would be the sum of the complexities of its parts? Explain your answer.
5. Would you expect a metric for measuring the complexity of a software system to be commutative in the sense that the complexity of a complete system would be the same if it were originally developed with feature X and had feature Y added later or if it were originally developed with feature Y and had feature X added later? Explain your answer.
6. How does software engineering differ from other, more traditional fields of engineering such as electrical and mechanical engineering?
7. a. Identify a disadvantage of the traditional waterfall model for software development.
   b. Identify an advantage of the traditional waterfall model for software development.
8. Is open-source development a top-down or bottom-up methodology? Explain your answer.
9. Describe how the use of constants rather than literals can simplify software maintenance.
10. What is the difference between coupling and cohesion? Which should be minimized and which should be maximized? Why?
11. Select an object from everyday life and analyze its components in terms of functional or logical cohesion.
12. Contrast the coupling between two program units obtained by a simple `goto` statement with the coupling obtained by a function call.
13. In Chapter 6 we learned that parameters can be passed to functions by value or by reference. Which provides the more complex form of data coupling? Explain your answer.
14. What problems could arise during maintenance if a large software system were designed in such a way that all of its data elements were global?
15. In an object-oriented program, what does declaring an instance variable to be public or private indicate about data coupling? What would be the rationale behind a preference toward declaring instance variables as private?

*16. Identify a problem involving data coupling that can occur in the context of parallel processing.

17. Answer the following questions in relation to the accompanying structure chart:
    a. To which module does module Y return control?
    b. To which module does module Z return control?

c. Are modules W and X linked via control coupling?
d. Are modules W and X linked via data coupling?
e. What data is shared by both module W and module Y?
f. In what way are modules W and Z related?



18. Using a structure chart, represent the procedural structure of a simple inventory/accounting system for a small store (perhaps a privately owned curio shop in a resort community). What modules in your system must be modified because of changes in sales tax laws? What modules would need to be changed if the decision is made to maintain a record of past customers so that advertising can be mailed to them?

19. Using a class diagram, design an object-oriented solution for the previous problem.

20. Draw a simple class diagram representing the relationships between magazine publishers, magazines, and subscribers. It is sufficient to depict only the class name within each box representing a class.

21. How is the relationship between the system and the user represented in a use case diagram?

22. Draw a simple use case diagram depicting the ways in which a traveler uses an airline reservation system.

23. Draw a sequence diagram representing the interaction sequence that would ensue when a utility company sends a bill to a customer.

24. Draw a simple dataflow diagram depicting the flow of data that occurs in an automated inventory system when a sale is made.

25. Contrast the information represented in a class diagram with that represented in a sequence diagram.

26. Provide at least two examples of a one-to-one relationship and a many-to-many relationship.

27. Give an example of a one-to-many relationship that is not mentioned in this chapter. Give an example of a many-to-many relationship that is not mentioned in this chapter.

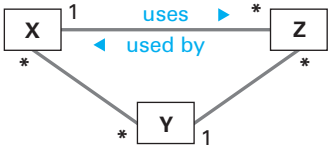
28. Based on the information in Figure 7.10, imagine an interaction sequence that might occur between a physician and a patient during a visit with the patient. Draw a sequence diagram representing that sequence.

29. Draw a class diagram representing the relationships between the servers and customers in a restaurant.

30. Draw a class diagram representing the relationships between magazines, publishers of magazines, and subscribers to magazines. Include a set of instance variables and methods for each class.

31. Extend the sequence diagram in Figure 7.5 to show the interaction sequence that would occur if `PlayerA` successfully returns `PlayerB`'s volley, but `PlayerB` fails to return that volley.

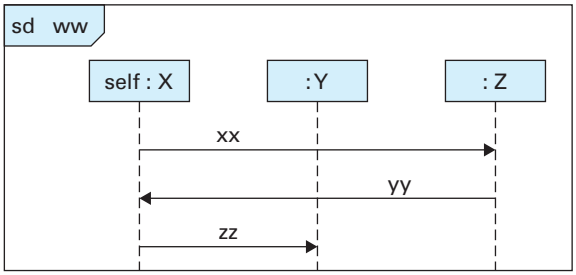
32. Answer the following questions based on the accompanying class diagram that represents the associations between tools, their users, and their manufacturers.



a. Which classes (X, Y, and Z) represent tools, users, and manufacturers? Justify your answer.
b. Can a tool be used by more than one user?
c. Can a tool be manufactured by more than one manufacturer?
d. Does each user use tools manufactured by many manufacturers?

33. In each of the following cases, identify whether the activity relates to a sequence diagram, a use case diagram, or a class diagram.
a. Represents the way in which users will interact with the system
b. Represents the relationship between classes in the system
c. Represents the manner in which objects will interact to accomplish a task

34. Answer the following questions based on the accompanying sequence diagram.



a. What class contains a method named ww?
b. What class contains a method named xx?
c. During the sequence, does the object of type Z ever communicate directly with the object of type Y?

35. Draw a sequence diagram indicating that object A calls the method bb in object B, B performs the requested action and returns control to A, and then A calls the method cc in object B.

36. Extend your solution to the previous problem to indicate that A calls the method bb only if the variable "continue" is true and continues calling bb as long as "continue" remains true after B returns control.

37. Draw a class diagram depicting the fact that the classes Truck and Automobile are generalizations of the class Vehicle.

38. Based on Figure 7.12, what additional instance variables would be contained in an object of type `SurgicalRecord`? Of type `OfficeVisitRecord`?

39. Explain why inheritance is not always the best way to implement class generalizations.

40. What is the significance of using the interaction diagrams provided by UML?

41. Summarize the process of software quality assurance (SQA).

42. To what extent are the control structures in a typical high-level programming language (`if-else`, `while`, and so on) small-scale design patterns?

43. Which of the following involve the Pareto principle? Explain your answers.
a. One obnoxious person can spoil the party for everyone.
b. Each radio station concentrates on a particular format such as hard rock music, classical music, or talk.
c. In an election, candidates are wise to focus their campaigns on the segment of the electorate that has voted in the past.

44. Do software engineers expect large software systems to be homogeneous or heterogeneous in error content? Explain your answer.

45. What is the difference between black-box testing and glass-box testing?

46. Give some analogies of black-box and glass-box testing that occur in fields other than software engineering.

47. How does open-source development differ from beta testing? (Consider glass-box testing versus black-box testing.)

48. Suppose that 100 errors were intentionally placed in a large software system before the system was subjected to final testing. Moreover, suppose that 200 errors were discovered and corrected during this final testing, of which 50 errors were from the group intentionally placed in the system. If the remaining 50 known errors are then corrected, how many unknown errors would you estimate are still in the system? Explain why.

49. What is boundary value analysis?

50. What is alpha testing and beta testing?

51. One difference between the human-computer interface of a smartphone and that of a desktop computer involves the technique used to alter the scale of an image on the display screen to obtain more or less detail (a process called "zooming"). On a desktop, zooming is typically achieved by dragging a slider that is separate from the area being displayed, or by using a menu or toolbar item. On a smartphone, zooming is performed by simultaneously touching the display screen with the thumb and index finger and then modifying the space between both touch points (a process called "double touch— spread" to "zoom in" or "double touch—pinch" to "zoom out").
a. Based on ergonomics, what arguments can be made in support of this difference?
b. Based on cognetics, what arguments can be made in support of this difference?

52. In what way do traditional copyright laws fail to safeguard the investments of software developers?

53. In what ways can a software developer be unsuccessful in obtaining a patent?

Social Issues

The following questions are intended as a guide to the ethical/social/legal issues associated with the field of computing. The goal is not merely to answer these questions. You should also consider why you answered as you did and whether your justifications are consistent from one question to the next.

1. a. Mary Analyst has been assigned the task of implementing a system with which medical records will be stored on a computer that is connected to a large network. In her opinion the design for the system's security is flawed but her concerns have been overruled for financial reasons. She has been told to proceed with the project using the security system that she feels is inadequate. What should she do? Why?
   b. Suppose that Mary Analyst implemented the system as she was told, and now she is aware that the medical records are being observed by unauthorized personnel. What should she do? To what extent is she liable for the breach of security?
   c. Suppose that instead of obeying her employer, Mary Analyst refuses to proceed with the system and blows the whistle by making the flawed design public, resulting in a financial hardship for the company and the loss of many innocent employees' jobs. Were Mary Analyst's actions correct? What if it turns out that, being only a part of the overall team, Mary Analyst was unaware that sincere efforts were being made elsewhere within the company to develop a valid security system that would be applied to the system on which Mary was working? How does this change your judgment of Mary's actions? (Remember, Mary's view of the situation is the same as before.)
2. When large software systems are developed by many people, how should liabilities be assigned? Is there a hierarchy of responsibility? Are there degrees of liability?
3. We have seen that large, complex software systems are often developed by many individuals, few of which may have a complete picture of the entire project. Is it ethically proper for an employee to contribute to a project without full knowledge of its function?
4. To what extent is someone responsible for how his or her accomplishments are ultimately applied by others?
5. In the relationship between a computer professional and a client, is it the professional's responsibility to implement the client's desires or to direct the client's desires? What if the professional foresees that a client's desires could lead to unethical consequences? For example, the client may wish to cut corners for the sake of efficiency, but the professional may foresee a potential source of erroneous data or misuse of the system if those shortcuts are taken. If the client insists, is the professional free of responsibility?

6. What happens if technology begins to advance so rapidly that new inventions are superseded before the inventor has time to profit from the invention? Is profit necessary to motivate inventors? How does the success of open-source development relate to your answer? Is free quality software a sustainable reality?
7. Is the computer revolution contributing to, or helping to solve, the world's energy problems? What about other large-scale problems such as hunger and poverty?
8. Will advances in technology continue indefinitely? What, if anything, would reverse society's dependency on technology? What would be the result of a society that continues to advance technology indefinitely?
9. If you had a time machine, in which period of history would you like to live? Are there current technologies that you would like to take with you? Can one technology be separated from another? Is it realistic to protest against global warming yet accept modern medical treatment?
10. Many applications on a smartphone automatically integrate with services provided by other applications. This integration may share information entered to one application with another. What are the benefits of this integration? Are there any concerns with "too much" integration?

Additional Reading

Alexander, C., S. Ishikawa, and M. Silverstein. *A Pattern Language.* New York: Oxford University Press, 1977.

Beck, K. *Extreme Programming Explained: Embrace Change,* 2nd ed. Boston, MA: Addison-Wesley, 2004.

Bowman, D. A., E. Kruijff, J. J. LaViola, Jr., and I. Poupyrev. *3D User Interfaces Theory and Practice.* Boston, MA: Addison-Wesley, 2005.

Braude, E. *Software Design: From Programming to Architecture.* New York: Wiley, 2004.

Bruegge, B., and A. Dutoit. *Object-Oriented Software Engineering Using UML, Patterns, and Java,* 3rd ed. Boston, MA: Addison-Wesley, 2010.

Cockburn, A. *Agile Software Development: The Cooperative Game,* 2nd ed. Boston, MA: Addison-Wesley, 2006.

Fox, C. *Introduction to Software Engineering Design: Processes, Principles and Patterns with UML2.* Boston, MA: Addison-Wesley, 2007.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Boston, MA: Addison-Wesley, 1995.

Maurer, P. M. *Component-Level Programming.* Upper Saddle River, NJ: Prentice Hall, 2003.

Pfleeger, S. L., and J. M. Atlee. *Software Engineering: Theory and Practice,* 4th ed. Upper Saddle River, NJ: Prentice-Hall, 2010.

Pilone, D., and N. Pitman. *UML 2.0 in a Nutshell.* Cambridge, MA: O'Reilly Media, 2005.

Pressman, R. S., and B. Maxim. *Software Engineering: A Practitioner's Approach,* 8th ed. New York: McGraw-Hill, 2014.

Schach, S. R. *Classical and Object-Oriented Software Engineering,* 8th ed. New York: McGraw-Hill, 2010.

Shalloway, A., and J. R. Trott. *Design Patterns Explained,* 2nd ed. Boston, MA: Addison-Wesley, 2005.

Shneiderman, B., C. Plaisant, M. Cohen, and S. Jacobs. *Designing the User Interface: Strategies for Effective Human-Computer Interaction,* 5th ed. Boston, MA: Addison-Wesley, 2009.

Sommerville, I. *Software Engineering,* 9th ed. Boston, MA: Addison-Wesley, 2010.