[CHAPTER] 1

OVERVIEW OF Modern Software Development

1.1 Introduction

This is a text on modern topics in software design and development, appropriate for use in a second or third course in computer science. Therefore, the first questions we should answer are: What does the phrase *modern software development* mean? What are the appropriate topics to include in its study?

In your first computer science course, you probably spent a good deal of time learning basic algorithmic concepts and the syntax and semantics of a high-level programming language—perhaps C++, Python, Scheme, or Java. You wrote a number of small programs, struggled to get them working correctly, and in the process, learned about such fundamental language concepts as variables, scope, conditionals, iteration, recursion, functions, and parameters. You also gained valuable experience with algorithms to solve important problems such as searching, sorting, and pattern matching.

This emphasis (some might say overemphasis) on programming in the first course often leads to the mistaken impression that coding and debugging are the most important activities in software development and the most challenging part of the software design process. Nothing could be further from the truth. In fact, coding and debugging may occupy as little as 15 to 20 percent of the time required to develop production software. Why is there such a huge discrepancy between what occurs in the classroom and what happens in the real world?

The answer can be summarized in a single word—*size*. The programs written in a first computer science course usually contain 50 to 500 lines of code. When writing programs of this limited size, you can keep all necessary implementation details in your head. You can remember what has been finished, what is in progress, and what still needs to be done. With small programs, you do not need a technical strategy or a management plan any more than children need a business plan to run a neighborhood lemonade stand. Instead, you dive directly into the coding, making decisions about how to address problems as they occur and debugging on the fly.

The problem, however, is that software packages that solve important tasks are not a few hundred lines long, and the strategy used for writing small programs does not work as you move to jobs on a larger scale. Figure 1-1 categorizes the size of software packages, lists the typical number of programmers for each category, and gives the typical duration from initial specification to delivery of the finished product. Although these numbers are only approximations, they do give you a good idea of the enormous size of most modern software packages.

CATEGORY	PROGRAMMERS	DURATION	PRODUCT SIZE (Lines of Code)
Trivial	1	1–4 weeks	< 1000 lines
Small	1–2	1–6 months	1000–10,000 lines
Medium	2–5	6 months to 2 years	10,000–100,000 lines
Large	5–20	2–3 years	100,000–500,000 lines
Very large	20–200	3–5 years	500,000–2,000,000 lines
Extremely large	> 200	> 5 years	> 2,000,000 lines

[FIGURE 1-1] Size categories of software products

As you can see from Figure 1-1, the overwhelming majority of programs developed in a first course would be categorized as trivial, although a student who has just spent all night in a computer lab struggling to get a program working might strongly disagree. Even by the end of a student's four-year computer science program, it would be unusual to develop a program beyond the category of small. However, virtually all real-world software falls at least into the medium category, and many are much larger. Such software could contain hundreds of thousands or millions of lines of code and occupy dozens of programmers for months or years. This includes such well-known packages as operating systems, productivity software, database programs, and e-commerce applications.

How do we approach programs of such enormous magnitude? How can we implement correct, efficient, user-friendly software, containing thousands or millions of lines, and make it easy to modify when errors are found or changes are proposed? The answers to these difficult but important questions will occupy us for the remainder of the book. Although we will not ask you to develop software that contains tens of thousands of lines of code (and we assume that neither will your instructor), the ideas that we will present, unlike the ones in your first course, *will* allow you to manage the massive projects that you will certainly encounter in your future activities.

NOW THAT'S BIG!

Figure 1-1 listed categories of software size, but only abstractly, and without referring to any actual packages. How large are some of the software packages in common use?

When describing program size, the most widely used metric is SLOC (source lines of code), a count of the number of nonblank, noncomment lines in the source program. Sometimes this value can be hard to determine because the application is proprietary and its source code is not available to the public. However, you can obtain some interesting SLOC values that indicate the magnitude of some well-known software.

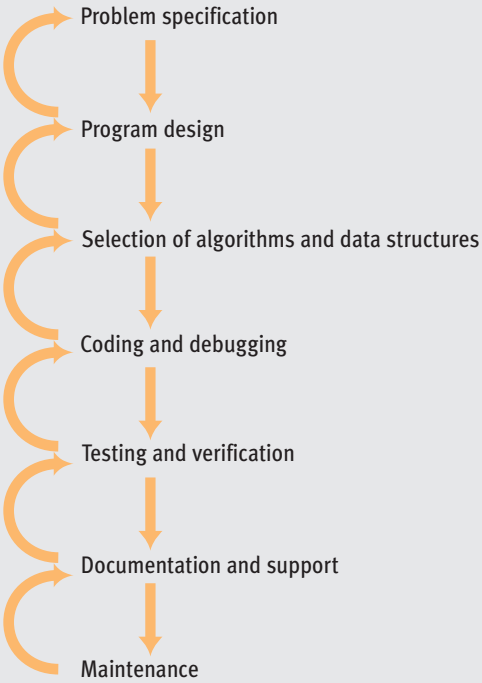
For example, the Mozilla Web browser, popular with Apple Macintosh users, contains about 300,000 SLOC, putting it in the Large category listed in Figure 1-1. The flight control software for the Boeing 777-200 jetliner has about 5 million lines of code. The popular Windows 3.1 operating system, released in 1990, had about 2 million SLOC, but future versions grew rapidly—4 million SLOC in Windows NT (1995), 18 million in Windows 98 (1998), and 40 million in Windows XP (2002). Red Hat Linux, Version 7.1 (2001) contained 30 million lines and took about 8000 person-years to develop. Finally, Debian GNU/Linux Version 3.1, released in 2005, contains 213 million SLOC. To give you an idea of how big that is, if the program listing were printed at 60 lines per page and stacked up, the pile would almost reach the top of the Empire State Building! Now that's big.

1.2 The Software Life Cycle

The first important point about software development is that a great deal of preparation is required before we even think about writing code. This preparatory work involves specifying the problem, designing the overall structure of the proposed solution, and selecting and analyzing the algorithms and data structures to use in the solution. This work is like the task of building a new house. It would be foolhardy to immediately pick up a hammer, nails, and lumber and start putting up walls. Instead, a great deal of planning, designing, and budgeting must be done first, culminating in a set of architectural blueprints that lay out the project specifications.

Similarly, a good deal of work must be done *after* the software has been completed, including such activities as testing, documentation, support, and maintenance. Again, using the housing comparison, these follow-up steps are equivalent to inspecting the house to ensure there are no construction flaws and making additions or modifications to the original home as your family size increases.

There is no universal agreement on the exact sequence of steps involved in software development, but there is general consensus on which activities are essential for producing correct, efficient software. This set of operations is called the **software life cycle**, and it is diagrammed in Figure 1-2. The following sections provide overviews of each of the phases shown in the figure and identify the later chapters that treat each topic in more detail.



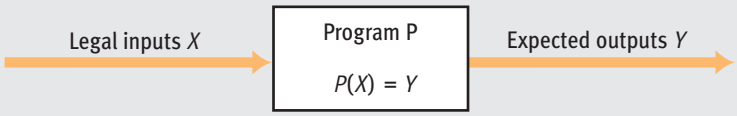
[FIGURE 1-2] Software life cycle

1.2.1 Problem Specifications

The **problem specification phase** involves the creation of a complete, accurate, and unambiguous statement of the exact problem to be solved. There is an old saying in computer science that it is not enough to solve a problem *correctly*; you must also solve the *correct problem*. Before starting design work, you must know exactly what you are to do, and the user and developer must agree about exactly what will be produced. It doesn't help anyone to build a correct, working program if it does not provide the desired results.

To specify a problem and clearly describe what tasks must be done, we must list the *inputs* provided to the program and, for each input, list exactly what *outputs* the program

will produce. Thus, a **problem specification document** is really an *input/output* document that says, "If you give me *X*, I will give you *Y*," as diagrammed in Figure 1-3. Notice that the specification document says nothing about how *X* is transformed into *Y*.



[FIGURE 1-3] Basic structure of a problem specification document

In addition to input/output specifications, the document usually contains information on the delivery date, budget constraints, performance requirements, and acceptance criteria. In an academic environment, you should be quite familiar with the concept of a specification document, although you are more likely to call it *homework*. A typical classroom project, such as Figure 1-3, describes its inputs, asks you to write a program that produces specific outputs from these inputs, and requires that you do it by a given date. Thus, in a classroom setting, the first phase of software development is not done by students but by the instructor who specifies the problem to be solved. In real life, this is not the case—you must flesh out a complete problem statement through meetings and interviews with potential users.

Although it is easy to describe a problem specification document, it can be rather difficult to create a good one. In fact, a significant number of errors occur because of incomplete, inaccurate, and ambiguous specifications. There are at least three reasons for these errors, as described in the following sections.

REASON 1: *The people specifying the problem may not describe it accurately and may frequently change their mind about what they want done.*

Problem specifications are created by talking with users to find out what they want. Some users will be quite certain about their needs and will describe the problem thoroughly and accurately. Unfortunately, most users won't. They will forget essential details, give misleading or incorrect information, and repeatedly change their minds about how the software should function. This makes it difficult to create accurate specifications and leads to errors, omissions, cost overruns, and missed deadlines.

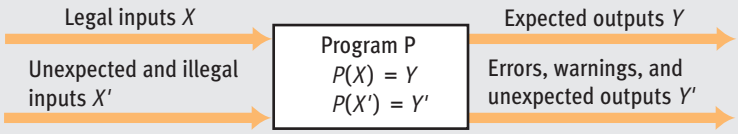
A technique called **rapid prototyping** is often used to help users decide what they really want. The software developer constructs a model, or prototype, of the planned software. The prototype contains a user interface that simulates the look and feel of the proposed software but has little or no functionality. However, seeing an actual interface and a proposed set of operations can help users identify what functions they want, what may be

missing, and what can be safely omitted. These prototypes can usually be built quite quickly and can help produce a more accurate description of program behavior.

The problem specification document, sometimes called a **software requirements document**, is like a contract between the user and developer. The developer agrees to build software that operates as described in the document, and the user agrees to accept it. Changes made after the parties agree to the specifications can be difficult and expensive to fix—much like changes to the design of a new home after the blueprints have been drawn or, even worse, after walls have started going up. Before any design work begins, everyone must be satisfied with the problem description in the specification document.

REASON 2: *Developers must include a description of the program's behavior for every possible input, not just the expected ones.*

When giving directions to a person, we assume a certain level of common sense and omit instructions that anyone would obviously understand. For example, when describing how to drive a car, we usually don't include commands to open the door and get into the driver's seat. Similarly, we do not explain what to do if you lose your keys. However, computer programs do not have common sense, so programmers must describe the behavior of the software for *all* inputs, even the most unexpected ones. For example, if you are developing a program to sort a set of numbers, you need to know what to do if one or more inputs is nonnumeric (such as three point five or XXVII) or if there is no input at all. The majority of a specification document is often devoted to describing appropriate responses to unusual, unexpected, and illegal inputs. This leads to the more realistic diagram of a problem specification document, as shown in Figure 1-4.



[FIGURE 1-4] More realistic diagram of a problem specification document

REASON 3: *Natural language (English, in our case) is a poor notation for writing accurate specifications, but it is the one almost universally used.*

English (or Spanish, Chinese, or Swahili) is a rich language full of contextual nuances, multiple meanings, and differences in interpretation. Such richness is wonderful for poetry or fiction, but not if you are trying to produce accurate specifications. For example, assume that a problem specification document contains an instruction to "Sum up all values in the N-element list L between 1 and 100." Does the word "between" mean that you should

include or exclude the numbers 1 and 100 themselves? What should we do if N is negative, as the concept of *negative list length* is meaningless? The answers are not absolutely clear from the preceding statement, which can lead to faulty software and incorrect results. It would have been more accurate to express this statement as:

If $N \geq 1$ then

$$\text{Sum} = \sum_{i=0}^{N-1} L_i \text{ for all } L_i \text{ such that } [(L_i \geq 1) \wedge (L_i \leq 100)]$$

Else

Sum = 0

The meaning of the problem is clear and unambiguous only if you understand the mathematical notation. In fact, that is the main problem with **formal specification languages**, such as the example shown above. They are more precise than natural languages, but they may be more difficult for people to read and interpret. On the other hand, natural languages are easy to read, but they lack the precision necessary for producing good specifications. In this text, we will use **UML (Unified Modeling Language)** to visually diagram the relationships and dependencies between various sections of programs.

The following example illustrates the difficulties of writing clear and unambiguous specifications. Here is a simple problem statement that you could have seen in a first computer science class:

> *You will be given an* N*-element array of numbers* A *and a key value* X*. If* X *occurs anywhere within* A*, return the position in* A *where* X *occurs. If* X *does not occur within* A*, then return the position of the entry in* A *whose value is closest to* X*.*

This may seem simple enough, but the following uncertainties lurk within this problem statement:

1. What should we do if X occurs *multiple* times within A? For example, assume $X = 8$ and A contains the six values 24, 7, 8, 13, 6, and 8. Should we return a 2 (assuming the first item in A is position 0)? Should we return a 5? Should we return both 2 and 5?
2. In the last sentence, what does the word *closest* mean? Does it mean *lexicographically* closest—that is, the greatest number of identical digits in the same position? In that case, the number 999 is closer to 899 than 1000 because there are two of three matching digits. Or does it mean *numerically* closest—that is, the smallest

value of $|A_i - X|$? In that case, 999 is closer to 1000 than 899 because there is a difference of only 1 rather than 101.

3 What if the list A is empty—that is, $N = 0$? Then nothing in A can be closest to X, regardless of the meaning of the word *closest*.

In this simple three-line statement, we identified at least three uncertainties, each of which could lead to errors in the finished program if not further clarified. (Exercise 3 at the end of the chapter asks you to identify additional ambiguities in the original specification.) Imagine the number of uncertainties that could occur in a real-world specification document containing dozens or hundreds of pages.

Writing good specifications is a difficult but essential first step in software development. All software you are asked to develop must be based on complete and accurate specifications; there should not be uncertainty about what you are supposed to do.

1.2.2 Software Design

When the specification document has been completed, we can move on to the next phase of software development, **software design**. In this phase, we specify an integrated set of software components that will solve the problem described in the specification document. A **component** is a separately compiled program unit. Depending on the language, a component may be called a function, procedure, class, template, package, or interface. However, regardless of what it is called, it serves the same purpose: helping to organize and manage the upcoming coding task. Software design is a **divide-and-conquer** operation in which a single large problem is divided into a number of smaller and simpler subproblems, much like outlining a paper before writing it. This modularization step is essential because it would be virtually impossible to implement a complex piece of software containing thousands of lines of code without some type of "master plan" helping us manage what needs to be done.

For each component in our solution, we specify the following three items:

- **Interface**—How the component is invoked by other units
- **Preconditions**—What conditions must initially be true when the component is invoked by another unit
- **Postconditions**—What conditions will be true when the component finishes, assuming that all preconditions have been met

The collection of these component specifications, along with their relationships, is called a **software design document**.

Just as the problem specification document is like a contract between user and developer, the pre- and postconditions are like a contract between a program unit and any other unit that invokes it. If all preconditions are true, the program unit guarantees that when execution is complete, all postconditions will be true.

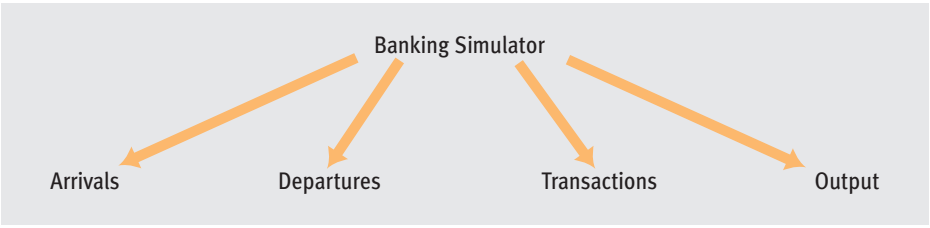
Here is an example of what a typical module specification might look like:

```
/** preconditions:    x is a positive real value > 0.0.
 *                    n is a positive integer >= 1.
 *
 *  postconditions:   positivePower returns the value x^n if
 *                    the preconditions are satisfied.
 *                    Otherwise, it returns -1.0.
 */
public double positivePower(double x, int n);
```

The most interesting question about software design is how to approach the decomposition process. How can we take problem P and subdivide it into pieces A, B, and C that, together, solve the problem? How do we know that this choice of components is better than subdividing P into W, X, Y, and Z instead?

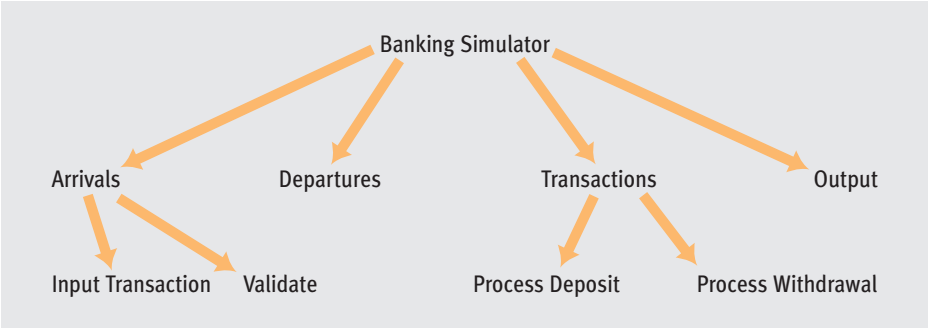
There are a number of ways to carry out this design process. In the early days of programming, the most widely used approach was **top-down design**, based on decomposing the problem *functionally*. We examine the activities taking place and design program units responsible for carrying out a single, coherent, well-defined task. For example, assume that we are designing a program to simulate a banking environment in which customers enter the bank, stand in a waiting line, carry out transactions, and leave. The bank plans to use this program to determine how many tellers to hire in its new branch office to provide the highest level of customer service at the lowest cost.

From a purely functional point of view, we might see the program as composed of four independent functions: customer arrivals, transaction processing, customer departures, and the output of results. Thus, our first-level design might look like Figure 1-5.



[FIGURE 1-5] First-level software design

After specifying the components at this level, we further subdivide the second-level routines based on the functions they perform. For example, the Arrivals routine might input the type and amount of a customer's transaction and validate that it is legal. The Transaction handler might include Process Deposit and Process Withdrawal units to handle these two transaction types. Now our software design will have expanded, as shown in Figure 1-6.



[FIGURE 1-6] Expanded software design

This top-down refinement process continues until we have described the entire problem as a set of small, simple, and easily understood components that do not need to be further decomposed. We are then ready to begin the next phase—coding each of these units.

The top-down design process does work, and for many years it was the most widely used software design methodology. Unfortunately, it suffers from a serious problem: a difficulty in modifying the final program to meet new or unexpected needs. For example, assume that after the simulator was completed, the bank changed the way it coded the transactions—maybe a deposit was no longer expressed as a *1* but as a *D*. Look at the above design document and ask yourself which routines might be affected by this minor change. Certainly, we need to modify Input Transaction and Validate because they both check the transaction type. The Transaction component needs to be aware of the different types of transactions to activate the correct program, so it probably needs some modifications. Finally, the transaction type will probably be included in the final report, so the output may have to be modified as well.

The point is that when you decompose a problem along functional lines, many program units need to know the details of the data structures being manipulated. In this case, many units examined the transaction type field, so all of them needed to see how transactions were represented internally, and the effect of one small change propagated throughout the program. This characteristic makes modifications difficult, and even tiny changes can cause massive headaches.

Since the early 1990s, a new methodology called **object-oriented design** has replaced the top-down approach to become the most important and widely used software design technique. In an object-oriented design, the software is decomposed not according to the functions carried out but along the lines of the *entities* within the problem. That is, we do not look first at what is being done, but at *who* is doing it. We create a component called a **class** that models each distinct type of entity in our problem. Then we instantiate instances

of these classes called **objects**. In the case of our banking simulation, we might choose to decompose our program into the following four classes:

- `Customer`
- `WaitingLine`
- `Teller`
- `Transaction`

Once we have decided on the classes in our design, we describe the behaviors of these classes—that is, the operations that every object of this class can perform. For example, we must be able to add a new customer to the end of every waiting line and remove its first customer. It must also be able to respond to the questions "Are you empty?" and "Are you full?" These operations are called **methods** in object-oriented terminology; selecting the appropriate set of methods for each class is a key part of object-oriented design. Figure 1-7 lists some operations that might be included in an object-oriented design of our banking simulation.

CLASS	`WaitingLine`	
METHODS	`putAtEnd(c)`	Put a new customer `c` at the end of this line.
	`c = getFirstCustomer()`	Remove the first customer; return it in `c`.
	`isEmpty()`	True if this line is empty; false otherwise.
	`isFull()`	True if this line is full; false otherwise.
CLASS	`Teller`	
METHODS	`isBusy()`	True if this teller is busy; false otherwise.
	`serve(c)`	This teller begins serving customer `c`.
CLASS	`Customer`	
METHODS	`depart()`	This customer leaves the bank.
CLASS	`Transaction`	
METHODS	`t = transactionType()`	Return the type of this transaction.
	`a = transactionAmount()`	Return the dollar amount of this transaction.

[FIGURE 1-7] Example of an object-oriented design

We can now begin to understand and appreciate the advantages of an object-based decomposition of a problem. Detailed information about transactions is obtained not by examining the internal structure of the data items themselves but by asking the transaction

object for the information. For example, if we need to print a report that contains the type and amount of a transaction *t*, it might be done like this:

```
type = t.transactionType();       // Ask t for its type
amt = t.transactionAmount();      // Ask t for its amount
printResults(type, amt);          // Output the results
```

The component requesting this information is isolated from any knowledge of how the Transaction object chose to represent transactions internally. Furthermore, if we change that representation, the preceding three lines of code are not affected in any way.

Similarly, if we want to know if waiting line *w* is empty or not, we do not look to see if a linked list pointer is **null** or an array index is 0. These operations would again leave us vulnerable to a change in implementation of the waiting line *w*. Instead, we simply ask the waiting line itself to tell us if it is empty or not:

```
boolean b = w.isEmpty();     // ask w if it is empty
```

As these examples demonstrate, the basic style of programming in an object-oriented environment is to exchange information between objects via method calls. Figure 1-8 shows a typical section of object-oriented code as it might appear using the methods listed in Figure 1-7.

```
Customer c;             // c will refer to a customer
WaitingLine w;          // w is where customers wait for a teller
Teller joe;             // joe will be our bank teller

w = new WaitingLine(); // create a new waiting line
joe = new Teller();    // make a new teller object
c = new Customer();    // c is a newly created customer

// See if the teller joe is serving someone

if ( joe.isBusy() ) {
   // Yes he is, so see if there is room in the line

   if ( w.isFull() ) {
      // No. The line is full, so the customer must leave.
      c.depart();
   }
```

continued

```
    else {
        // The line is not full. The customer can wait.
        // Put the customer at the end of the waiting line.
        w.putAtEnd(c);
        }
    }
// The teller is not busy. The customer can be served.
else {
    joe.serve(c);  // Serve the customer
    c.depart();    // Who then leaves the bank
}
```

[FIGURE 1-8] Example of the object-oriented programming style

Even though you may be unfamiliar with Java, you should still be able to appreciate the clarity of the object-oriented programming style in Figure 1-8.

Each of the objects in the system—`c`, `w`, and `joe`—can send messages to other objects and respond to inquiries about its current state or requests to change state. Only the object itself knows how to carry out these requests; the caller is simply given a response.

Part I of this book, Chapters 2 through 4, takes an in-depth look at the object-oriented design philosophy.

1.2.3 Algorithms and Data Structures

Our task is now laid out before us. The problem has been decomposed into a set of classes, each one representing a distinct entity in our solution. Each class contains methods that objects of the class can execute, and each method is clearly specified in terms of pre- and postconditions.

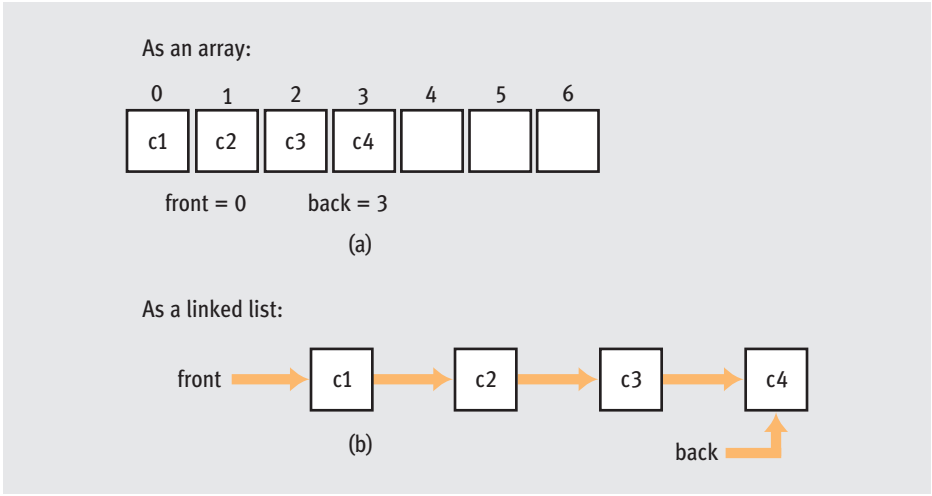
For example, the waiting line class mentioned in Figure 1-7 includes a method called `putAtEnd()` that might be described as follows:

```
/** Preconditions:  The waiting line is not full.
 *  Postconditions: Customer c has been added as the last
 *                  customer in the waiting line. If the
 *                  line is full, the method throws a
 *                  LineFullException.
 */
public void putAtEnd( Customer c ) throws LineFullException;
```

There may be hundreds or thousands of such method specifications in a typical design document, and each one must be implemented and tested for correctness.

An important decision you need to make before coding is selecting the data structures to represent data objects and choosing the algorithms to access and modify these structures. As you will learn in Chapter 5, software efficiency is most strongly influenced by the data structures and algorithms selected to manipulate the data, not by the choice of programming language, how well the code is written, or by a machine's computing speed. An efficient algorithm remains reasonably efficient no matter how inelegantly it is written, and no amount of programming cleverness or machine speed can turn an inherently inefficient method into an efficient one. Therefore, this book will spend a good deal of time on data structures and the different techniques for storing and representing information.

For example, referring to the previous specification of `putAtEnd()`, we could choose to implement the waiting line as an array, as shown in Figure 1-9, or we might opt to implement it as a linked list. Both diagrams show a waiting line that contain four customers, `c1`, `c2`, `c3`, and `c4`, in that order.



[FIGURE 1-9] Different ways of implementing a waiting line

If we choose to represent our waiting line as an *array*, then the `putAtEnd()` operation can be implemented in just two steps, if there is room:

```
back = (back + 1) % arraySize;  // add 1 to back
                                // modulo array size
a[back] = c;                    // put c at end of the line
```

However, if the array is full, then we have made a poor choice. An array is a static data structure that cannot be dynamically enlarged beyond the number of spaces allocated when

it is declared. Instead, we need to create a new array and copy all values from the old array into the new one—a potentially time-consuming task.

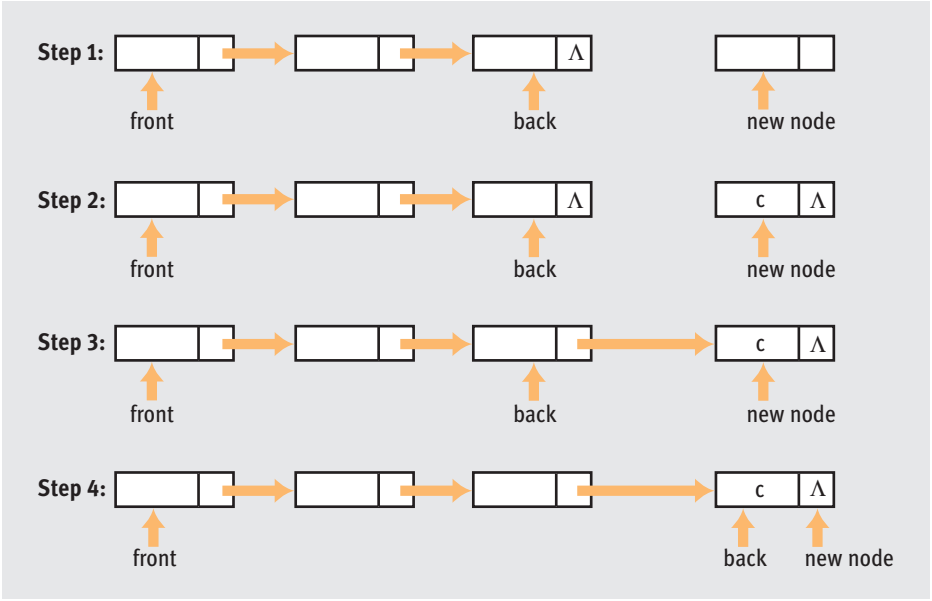
If the maximum size of the waiting line is unknown, then we might be better off selecting a *linked list* representation, which does not place an upper bound on the maximum number of customers, as long as memory is available. A linked list is a collection of units called nodes. Each node contains a data item and a next field that explicitly points to the next node in the list. Now the `putAtEnd()` algorithm would behave as described in the following four steps and diagrammed in Figure 1-10, assuming that the list is not empty:

Step 1: Get a new node to hold this customer.

Step 2: Put the customer object `c` into the data field of the new node and a `null` (Λ) into the next field, where `null` means this node is not pointing at anything.

Step 3: Reset the next field of the last node in the list (called `back` in Figure 1-10) so that it points to the new node.

Step 4: Reset `back` so the new node just added becomes the last node in the list.



[FIGURE 1-10] Implementing `putAtEnd()` using a linked list

This is an efficient operation that takes only four steps, but notice how much more memory is required to represent the waiting line. For every customer in the list, we need a next field that points to the next customer in line. This field is not needed when we use an array. If only a few customers are in line, the amount of memory is trivial and not a problem. However, if the line contains tens or hundreds of thousands of customers (for example,

people waiting to access a popular Web site), then this choice could require a great deal of extra memory.

Is there another way to represent a waiting line that does not suffer from the problem of fixed size or extra memory but still allows the `putAtEnd()` operation to be implemented efficiently? These are exactly the kind of questions we will study in Part II of the book, Chapters 5 through 9.

THE BILLION DOLLAR BUG

On June 4, 1996, the European Space Agency (ESA) launched the first of its new Ariane 5 rockets, which took 10 years to develop at a cost of more than $7 billion. Unfortunately, the rocket self-destructed just after takeoff due to a malfunction in its flight control software. Extensive examination revealed that the crash was caused by a software bug that converted a 64-bit floating-point number into a 16-bit signed integer. The floating-point number was too large to be represented in 16 bits, and it generated a conversion error. A coding oversight caused this conversion problem to be incorrectly handled, leading first to instability and then destruction of the entire vehicle. It would be 17 months (October 30, 1997), before the next Ariane test flight would be scheduled.

The overall cost of this software error was ultimately $1 billion—certainly one of the most expensive bugs in computing history. Remember this example the next time there is a "small and rather insignificant error" in your program that you overlooked during debugging.

1.2.4 Coding and Debugging

Coding is the software development phase you undoubtedly know best, as it is covered extensively in virtually every first course in computer science. This book uses Java, which may not be the programming language you initially studied. If it is, you will have no problem understanding our code. If your first language was not Java, consult the appendix of this book before continuing. The appendix explains basic Java syntax and covers basic procedural concepts such as declarations, assignment, input/output, conditionals, iteration, functions, and parameters.

Java was developed in 1991 by James Gosling of Sun Microsystems Inc. It was not originally conceived as a programming vehicle for traditional applications. Instead, Sun

wanted to create a platform-independent environment to control **embedded systems**—microprocessors inside consumer devices such as televisions, microwaves, and coffeemakers. Unfortunately, the consumer electronics market of the early 1990s was not as lucrative as Sun had expected, and the Java language project was in danger of being cancelled. At about the same time, however, another application exploded onto the marketplace—the World Wide Web.

Sun quickly realized that Java's architecturally neutral design made it a perfect language for getting dynamic content into Web pages. Using Java, you can write a small program called an **applet** and embed it into a Web page. When someone accesses this page, the applet is transmitted with the HTML commands and is executed on the client's computer, regardless of its make or model. Sun released Java in May 1995, and it generated enormous interest because of its commercial possibilities. Since then, it has become one of the most popular programming languages in both the commercial and academic marketplaces.

If Java was only useful for building applets and "sexy" Web pages, the language would hold little interest in academic computer science. However, Java has become an important classroom tool for three other reasons:

Java is a true object-oriented language designed from the start to support the concepts of object-oriented design.

Object-oriented design became popular in the early to mid-1980s, well after the appearance of popular languages from the 1950s, 1960s, and 1970s such as Fortran, COBOL, BASIC, Pascal, and C. If programmers in these older languages wanted to use object-oriented design principles, they had to discard their current language and learn a new one. (There was one object-oriented language available at that time, called Smalltalk.) Most programmers did not want to learn a totally new language, so they took existing languages and created object-oriented versions by tacking on new features. This was the reasoning behind the design of C++, an object-oriented extension of C. However, it was usually obvious that these new languages were quick-and-dirty patch jobs that grafted object-oriented features onto existing languages.

Java, however, is a true object-oriented language, not a hastily crafted extension, and the class concept is central to the design of the language—in other words, every entity (with a few minor exceptions) is an object that is a member of some class. With Java, you will learn a modern programming language created to fully incorporate the capabilities and techniques of object-oriented design. Java is not the only modern object-oriented programming language, but it is one of the most widely used.

Java includes an enormous number of standard classes that provide a range of important and useful services.

One issue studied by computer scientists is **software productivity**—how to increase the amount of correct, working code a programmer produces in a given time period. In the early days of computing, from 1950 through 1980, the operational principle of software design was: "Machines are costly, programmers are cheap, so optimize for the computer." In this environment, it was appropriate to assign dozens or hundreds of programmers to a task and let them take as much time as necessary, as long as the final program fully and efficiently used the multimillion-dollar computer on which it was run. That situation has totally reversed. Computers cost as little as $500 or $1000, but developers' salaries may exceed $100,000 per year. In this environment, our new goal is to maximize the productivity of these highly trained and very expensive programmers.

One way to increase a programmer's productivity is through **software reuse**—making use of software that already exists rather than developing it from scratch. It is certainly faster and more efficient to take a finished program out of a library than to write it yourself. Unfortunately, while software reuse is a highly desirable goal, it has not yet made much of an impact. Complete programs are not always easily sharable "chunks" of code because they may incorporate application-specific material that you can't transfer to other situations. You might be able to develop a totally new program in less time than it takes to modify existing software to fit your needs.

However, classes are an ideal unit of sharing, as they do not address the entire program but only a small component that typically occurs repeatedly in computing applications. For example, if you already developed a waiting line class in the earlier bank simulation example and placed it in a library, you could reuse the class in any future application that included a waiting line data structure, greatly simplifying the task and reducing development time.

Java includes a huge library of classes that address common aspects of programming, such as error handling, graphics, user interface design, file input/output, networking, security, multitasking, and cryptography. Java 1.0, the first public version of the language, contained 212 distinct classes, and Java 1.1 raised that number to 504. Java 1.2 had 1520 classes in 59 different packages, while Java 1.4 increased those numbers to 2757 classes in 135 packages. That seems like an enormous number—certainly more than we could ever use. However, the most recent version, Java 1.5, which we use in this text, has increased the size of its standard libraries to 166 packages and 3279 classes—a truly monumental amount of freely available software. We will make use of many built-in classes throughout this text.

Although there is no way in a one semester text to discuss even a fraction of these classes, we will highlight some useful services provided by the standard Java libraries. Our goal is to make you realize that, in modern software development, your first thought during coding should not be "How do I write it?" but "Where can I find it?" If you can use code that is already written, debugged, and tested, your productivity will increase significantly, and the cost of software development will be dramatically reduced.

Java contains a wide range of language features that are vital in modern software development.

Because Java was created in the 1990s, rather than the 1960s, 1970s, or 1980s, its designers could think about what features developers would need in the 21st century and include them in the language and its environment. As a result, Java contains features not found in many other languages, which are becoming increasingly important in the design of correct, cost-efficient, maintainable code. We will examine these features in Part III of this book, "Contemporary Programming Techniques," found in Chapters 10–13. Some of these important new ideas are discussed in the following pages.

1.2.4.1 Exceptions

You know how frustrated you become when a program crashes with an unfathomable error message or the infamous spinning "beach ball." Computing is critical to the proper functioning of society, and software failures are no longer just frustrating; they can lead to financial ruin or catastrophic disasters. (See "The Billion Dollar Bug" earlier in this chapter.) Today, software must deal with all types of faults in a planned, orderly way. It is no longer acceptable to write the following code:

```
if (code < 0)                // transaction code is illegal
     errorFlag = true;
else {
     errorFlag = false;
     activateHandler(code);  // code is legal
}
```

and then hope and pray that the person who invokes your program remembers to check the value of `errorFlag` before assuming that the transaction was correctly processed.

Exception handling is a powerful way to deal with errors, failures, and other unexpected circumstances that occur during program execution. Java includes a number of built-in exceptions to handle common error conditions; Java also enables users to define and handle their own exception conditions. Exceptions are discussed in detail in Chapter 10.

1.2.4.2 Streams

As commercial and business applications become more information centered, programming languages increasingly must deal with different types of data streams transmitted via a network or an external file. Java contains a powerful set of classes for handling network and file input/output in many formats. It lets the programmer view information as either binary or

character streams, either buffered or unbuffered. Java also provides classes to deal with such issues as data compression, type conversion, and data encryption. We take an in-depth look at these topics in Chapter 10.

1.2.4.3 Threads

From the earliest days of computing, most languages have been sequential in nature, and their programs execute serially, one statement at a time from beginning to end. However, this is changing, and modern high-speed computers can carry out multiple tasks at once. You exploit this capability whenever you start a print job and then, instead of waiting for it to complete, immediately begin reading your e-mail or surfing the Web.

A **thread** is a program unit in execution, so the capability of executing more than one thread at the same time is called **multithreading**. If your system has more than one processor, then each thread can execute in **parallel**, each one running on a separate processor. If your computer has only a single processor, you can still do multithreading, but the threads must take turns. Each thread runs for a while and then suspends, allowing another thread to execute. This round-robin mode of execution is called **concurrent processing**, which we see whenever a print server or Web server deals with multiple requests at the same time.

In the days of computing prior to the 1990s, statements for supporting concurrency and parallelism were rarely included in a programming language. The only way to implement these services was by accessing complex operating system primitives, which were understood by only the most advanced programmers. It was certainly not a topic for first-year students. However, modern software makes extensive use of concurrency or parallelism, and threads now need to be introduced in first-year classes so students can see the concepts and feel comfortable when they are discussed more fully in advanced classes.

Java includes an extensive set of routines for creating, scheduling, executing, suspending, and terminating threads. These operations are located in the `java.lang.Thread` class, and they will be discussed in Chapter 11. Our goal is to introduce you to the concepts of parallelism and concurrency. These ideas will be discussed at greater length in advanced classes on operating systems, distributed systems, and parallel processing.

1.2.4.4 Graphical User Interfaces

Most applications developed in the past few years use a **graphical user interface** (GUI, pronounced *goo-eey*) to interact with users. No programming environment can claim to be modern and up to date if it does not include an extensive array of routines for building high-quality visual interfaces like the one in Figure 1-11. These routines allow users to create visual features such as windows, buttons, text fields, labels, choice boxes, check boxes, and lists. These routines also allow you to control the placement, size, and color of components to customize the look and feel of your interface.

[FIGURE 1-11] Example of a modern graphical user interface

Java includes a powerful collection of classes called the **Abstract Windowing Toolkit**, located in the package `java.awt`. This package allows users to create the type of interfaces shown in Figure 1-11. Java later added a second package called **Swing** with an even more extensive set of visual components. Both of these GUI toolkits will be discussed in Chapter 12.

1.2.4.5 Networking

Most programs today are "net-centric," which means they work in tandem with a computer network such as an Ethernet LAN or the Internet. We are all familiar with e-mail, file transfer, and Web browsers. More recently, there has been an explosion of new network-based applications such as distributed databases, dynamic Web pages, chat rooms, Internet TV and telephone, remote sensing, and a host of other client/server applications. In addition, e-commerce and e-banking, which would not exist without networks, are multibillion-dollar industries. Today, software rarely functions in a stand-alone fashion without addressing data communications.

As with threads, statements to support and implement networking and data communications were generally not part of older programming languages. You had to access the operating system to obtain these services. Therefore, it was another topic considered appropriate only for advanced students, not the beginning curriculum. However, Java includes a package

called `java.net` that allows users to develop client/server network applications using the TCP/IP and UDP network protocols.

The classes in `java.net` are easy to explain and understand, and are totally appropriate to include in the first year of study. As before, our goal is not to make you a networking expert but to introduce important topics in modern software development—sockets, connections, packets, datagrams, clients, servers, and protocols. This will help make you comfortable with the ideas when they reappear in advanced classes on networks, data communications, and distributed computing. Chapter 13 discusses the `java.net` package and briefly introduces the concept of **security**, an essential component of any modern networking package.

To summarize, we have chosen to use Java 1.5 in this text because it is a true object-oriented language, it has class libraries that provide important services, and it has language features that allow you to develop fault-tolerant, visually oriented, multithreaded, net-centric programs. Now that's a modern software development environment!

1.2.5 Testing and Verification

You have written your program, debugged it, and finally gotten it to work correctly on a single data case. But that doesn't mean it is correct. It only means that you are ready to begin the next phase of software development—testing and verification—in which you demonstrate that the program is correct and performs exactly as described in the problem specification document. There are two ways of demonstrating correctness—one is very powerful but difficult and controversial; the other is easier but much less powerful.

Program verification is a technique for formally proving the correctness of a program, much as you prove the truth of a geometric theorem or an algebraic formula. You develop the correctness proof of the program as you are writing the code. You develop a formal argument, which says that given input *I* and the basic axiomatic truths of your program unit (the preconditions), the output *O* of the program unit (the postconditions) are identical to the output given in the specification document. This is a difficult task, but if you can do it, you can state that the program will work according to specifications for *every* possible input, not just the explicit ones that are tested. That is an extremely powerful assertion.

Verification can take a great deal of effort. For example, it is not uncommon for a correctness proof of a one-page code fragment to take pages and pages of formal argument. The proof is as likely to contain an error as the program itself.

Because of this complexity, many people believe that program verification will never become an important software tool. Other computer scientists believe that it will become increasingly useful as we learn more about how to develop proofs and create languages that support and encourage formal verification. (One example might be a functional language such as Scheme.) It is hard to know which of these arguments will ultimately prevail. However, it is fair to say that formal verification is not widely used and has not yet made a

significant impact on the development of real-world software. We will not discuss it further in this book. (However, see the Challenge Work Exercise at the end of this chapter if you are interested in this fascinating topic.)

Instead, most software designers demonstrate correctness through **empirical testing** of a program using a number of carefully crafted test cases. If the program works correctly for all of the cases, then we might argue that the program will work properly for all data sets, even those we did not test. Of course, that statement is not true. There still could be an error that was not detected by the test cases. However, if we are extremely careful in how we choose test data, and if we thoroughly and completely test everything we have written, we can have reasonable confidence that the overwhelming majority of errors were located and corrected.

Empirical testing usually proceeds in three steps. The first step is **unit testing**. As we finish each program unit, we thoroughly test it with a carefully chosen set of test data. The program is not placed into the code library until it has passed all tests with a 100% success rate.

When designing test suites, you must attempt to test every flow path through the program at least once, and preferably multiple times. A **flow path** is a unique execution sequence through a program unit. For example, if B_i represents a Boolean condition and S_i represents a Java statement, then the following code fragment:

```
S1
while (B1) {
      if  (B2)
            S2
      else
            S3
}
```

contains the following seven flow paths:

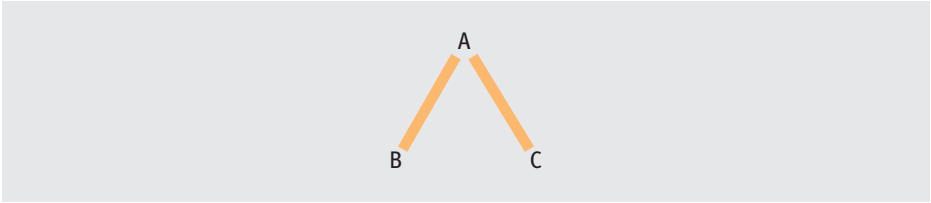
- S_1—The path we take when we do not execute the loop at all.
- S_1, S_2—The path we take when we execute the loop once and execute the true branch of the conditional.
- S_1, S_3—The path we take when we execute the loop once and execute the false branch of the conditional.
- S_1, S_2, S_3 and S_1, S_3, S_2—The two paths we could take if we execute the loop twice and execute a different branch each time.
- S_1, S_2, S_2 and S_1, S_3, S_3—The two paths we could take if we execute the loop twice and execute the same branch each time.

However, as this example demonstrates, there are a virtually infinite number of flow paths because the loop can be executed an arbitrarily large number of times. That is why **exhaustive testing**, the testing of every possible sequence of instructions in a program unit, is usually impossible. Therefore, your goal during unit testing cannot be to test every possible

flow path; instead, it is to make sure that every statement in the program unit is executed at least once (and, ideally, multiple times), and that you have included data sets to test as many of the key flow paths as possible.

For example, if we tested our code fragment with the seven examples shown above, we would have executed every statement in the fragment (`while`, `if`, S_1, S_2, S_3) multiple times, and we would have included test cases in which (a) the loop is not executed, (b) the loop is executed once, (c) the loop is executed more than once, and (d) both the true and false branches of the conditional are executed. These cases exemplify the type of carefully planned selection process you must follow during empirical testing.

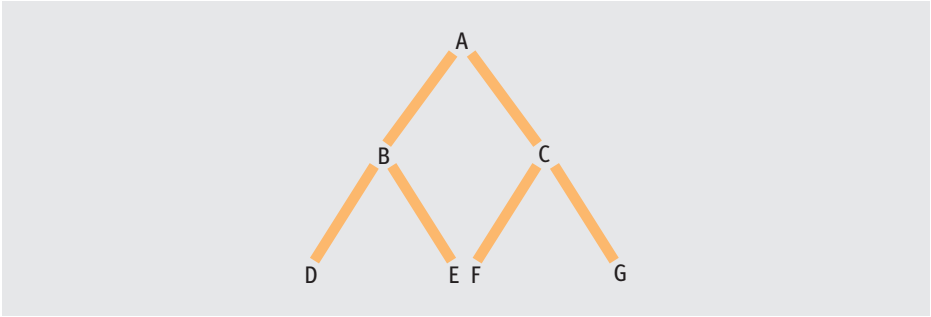
The second level of testing is **integration testing**, in which you test the correctness of a group of routines working together. For example, assume you are given the three program units shown in Figure 1-12.



[FIGURE 1-12] Simple example of integration testing

A, B, and C might work properly by themselves but not when grouped together. For example, when testing A, its builder may assume that he can call B with parameters X and Y by writing B(X,Y). Similarly, the person who coded B may believe that the correct parameter order was B(Y,X). Each test will work exactly as its designer thinks it should. However, when the two units attempt to operate together, an error occurs. It only reveals itself when units A and B are executed as a group.

Integration testing usually proceeds from the bottom up, in ever-increasing group sizes, as shown in Figure 1-13.



[FIGURE 1-13] More complex example of integration testing

After empirical testing of each unit, we might choose to test the BDE cluster and then the CFG cluster. Only when both integration tests have been completed satisfactorily should we proceed to test the entire seven-unit collection.

Both unit testing and integration testing are examples of **clear box testing**, or **alpha testing**, which means we can look inside the code to decide what test cases are needed. We carefully examine the program structure to ensure that we have tested all flow paths.

However, the third and final phase of testing is quite different. It is called **black box testing**, also referred to as **beta testing** or **acceptance testing**. Rather than selecting test data on the basis of the program structure, we place the program into a realistic environment and then run it. The data is related to actual user operations rather than being an artificial stream created to exercise specific sections of code. We see if the program operates correctly in this real environment and measure whether it meets the performance criteria in the specification document. These performance criteria will be statements like the following:

- The program must operate with a mean time between failures of 24 hours.
- The program must process a minimum of 1000 transactions/hour.
- The program must respond to 99.9 percent of customer requests in less than 1 second.

If the beta tests show that the program is operating correctly and meets all performance criteria, then we will have met the requirements in the problem specification document. The software project has been completed, and the finished program can be delivered to the user.

1.2.6 Postproduction Phases

It can be misleading to state that a software project has been completed. In the classroom, once a program has been written, debugged, and tested, it is rarely ever used again. It is handed in to the instructor for grading, and students move on to the next assignment. In the real world, however, programs have a long shelf life. It is not unusual for initial development to last 1 to 3 years, with the program remaining in use for 5 to 15 years. It is expensive to develop good software, so once it has been successfully completed, it is maintained and used as long as possible.

Therefore, the software life cycle includes a number of important steps that continue well after initial completion of the program. Although you will have little opportunity to carry out these steps in an academic setting, we mention them for the sake of completeness.

1.2.6.1 Documentation and Support

Documentation involves producing the information needed both by users and the technical support staff who maintain the program.

User documentation usually means a complete and well-written user's manual. In most modern software, it also may include online documentation, such as a searchable database of helpful information. In addition to both printed and online documentation, users today expect an extensive support environment. This might include phone and e-mail support, a Web site, downloads of error corrections and new versions of the software, and chat rooms where users can exchange ideas and get help. For example, the Web site *http://java.sun.com* contains a Java tutorial, chat rooms, frequently asked questions (FAQs), and other useful features for new students of the language.

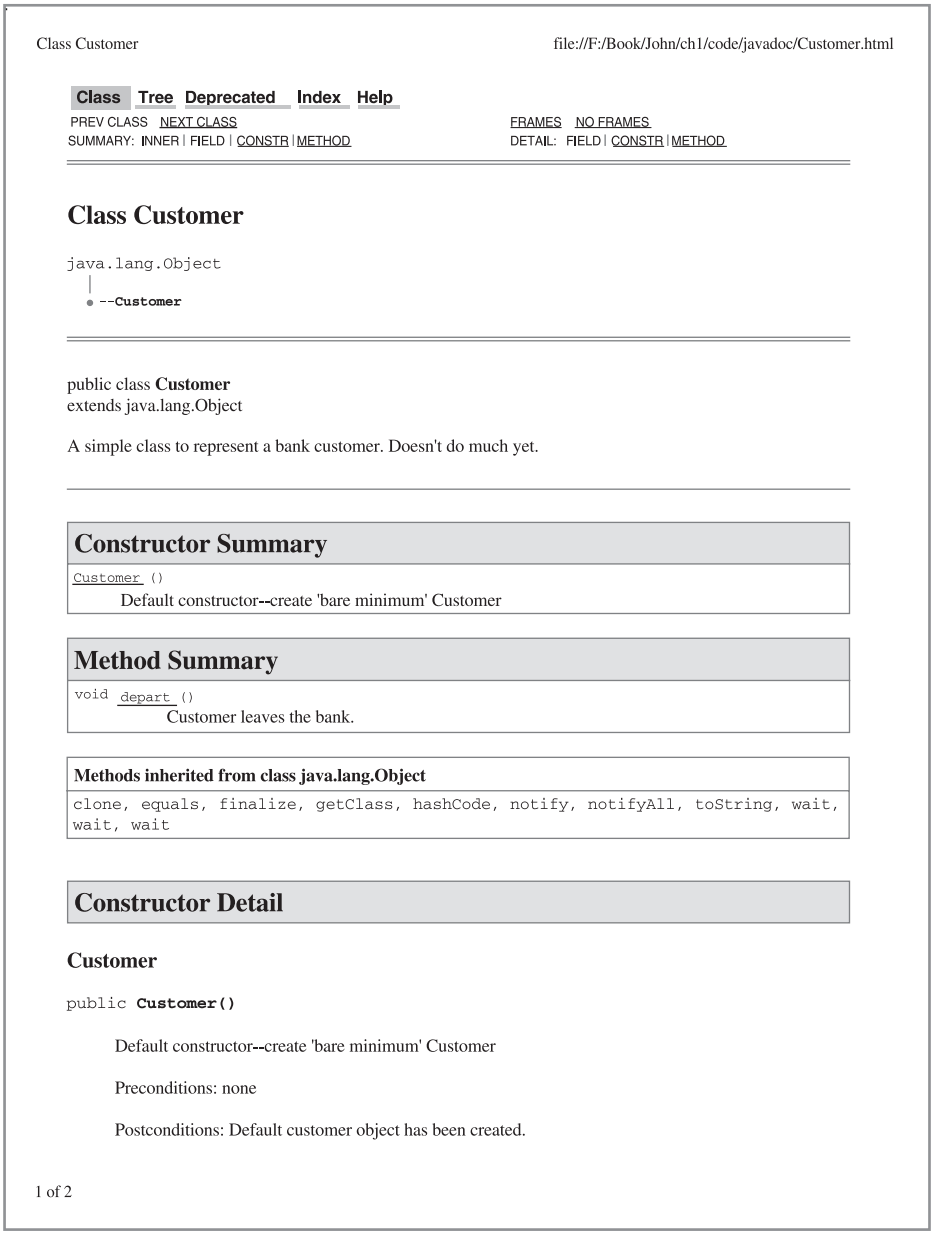
Technical documentation is a collection of all the written materials produced during the software life cycle. This includes the documents and descriptions described earlier in this chapter, including the problem specification document, the program design document, a description of the algorithms and data structures, a listing of the program along with inline comments, and a description of the testing and acceptance procedures to which the program was subjected.

The designers of Java realized the importance of high-quality technical documentation, so they developed a special tool to assist in its preparation. This special type of inline comment, called a **Javadoc comment**, is placed within the symbols `/**` and `*/` in the program code itself. The comments contain special markers, called **tags**, that provide information about a program unit. The tags are preceded by the special symbol `@`, and they can be extracted by a special Javadoc processor and displayed in HTML format. There are more than a dozen types of tags, but four of the more important ones are:

- `@author` *name* — Identifies the author of the code
- `@version` *text* — Specifies the version number of the code
- `@param` *name description* — Gives information about all of the parameters
- `@return` *description* — Describes the value returned by the method

The `@author` and `@version` tags provide important historical information about a program unit. The `@param` and `@return` tags behave much like the pre- and postcondition comments described in Section 1.2.2.

For example, if we included the Javadoc tags given in the `Customer` class of Figure 1-7, we could produce documentation that would look like the output in Figure 1-14.



Class Customer file://F:/Book/John/ch1/code/javadoc/Customer.html

Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: INNER | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class Customer

```
java.lang.Object
  |
  +--Customer
```

public class **Customer**
extends java.lang.Object

A simple class to represent a bank customer. Doesn't do much yet.

Constructor Summary

`Customer ()`
Default constructor--create 'bare minimum' Customer

Method Summary

`void depart ()`
Customer leaves the bank.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

Customer

`public Customer()`

Default constructor--create 'bare minimum' Customer

Preconditions: none

Postconditions: Default customer object has been created.

1 of 2

[FIGURE 1-14] Example of Javadoc documentation

Not only is this documentation both highly readable and in a standardized format, it is also in HTML format, which means it is easy to place on the Web for all developers to access.

1.2.6.2 Program Maintenance

The last step in software development is program maintenance. This is not really a separate phase but a realization that completed programs are not static entities. As mentioned earlier, some programs are in use for 5, 10, or even 15 years after initial development. During that time, new errors will be uncovered, new hardware will be purchased, and user needs and marketplace whims will fluctuate. Therefore, the original program needs to be modified to meet changing conditions.

Program maintenance is the process of adapting software to maintain correctness and keep it current with changing specifications and new equipment. In a sense, program maintenance implies that the software life cycle must be repeated more than once. For example, as new errors are uncovered, we need to rewrite code, test it, update documentation, and distribute the modifications. When newer, faster hardware becomes available, we may want to modify the program design and code to exploit the new equipment. If the user has new and unexpected needs or the marketplace demands new capabilities, we may decide to redo the specification document and implement a totally new version of the software.

If our software package has been accurately specified, organized using the object-oriented design philosophy, coded in Java using good object-oriented programming techniques, and thoroughly tested and documented, then program maintenance will be a much less onerous task. Given the modular nature of object-oriented software, program maintenance should, at least theoretically, be similar to modern TV repair. When there is a problem with a TV, a repairperson checks the components one by one, locates the defective part, removes it, and snaps in a new one. To fix or update software, we should be able to locate the class that needs to be adapted, remove the current code, plug in new code, and have a correct working version.

Thus, following the software life-cycle guidelines in this section will help to make the initial program work correctly, as well as ensure that future maintenance will not be difficult and expensive.

1.3 Summary

There is much more to programming than coding. Recent studies on the time spent by software developers in various stages of initial program development show that relatively little time is spent on implementation (coding and debugging), but a much larger amount is spent

on necessary preparatory work such as specification, design, and algorithmic planning. Figure 1-15 lists the approximate percentage of time spent on each major phase of initial software development.

PHASE	APPROXIMATE PERCENTAGE OF TIME
Specification, design, and planning	30–40
Coding and debugging	15–20
Testing, reviewing, and fixing	30–40
Documentation and support	10–15

[FIGURE 1-15] Approximate percentage of time spent on phases of software development

Note that Figure 1-15 describes only the time allocated for *initial* development. Over the 5- to 15-year life of a software package, a great deal of time and money will be spent maintaining the software after it has been released for general use.

Software development is an extremely complex task made up of many phases, each of which is important and contributes to the overall solution. Do not equate the topics of *software development* and *coding*. Remember that coding must be preceded by a good deal of preparatory work clarifying, specifying, and designing the solution, and it must be followed by testing, verifying, measuring, and documenting the finished program to guarantee its quality. The term **software engineering** is often used to refer to the systematic and disciplined approach to the overall design, implementation, and maintenance of computer software.

DR. FREDERICK P. BROOKS, JR.,
The Father of Software Engineering

Fred Brooks is a computer scientist and was a software engineer before the meaning of the term was well understood. He received his PhD from Harvard University in 1956 and then went to work at IBM, where he became project manager for the development of the new OS/360 operating system—one of the largest software projects ever attempted. At the time, the concepts of planning, designing, and organizing software development projects had not been well established, and it was widely believed that any program, regardless of size, could be completed on time by adding more programmers until it was on schedule. Well, the OS/360 project ran late—so late that IBM had to ship the 360 computer without

continued

its software—quite an embarrassment for the corporate giant. Years slipped by and the program was still not finished, even though the number of people working on it grew to more than 2000 in 15 laboratories in six countries. OS/360 was eventually finished, but only after enormous cost overruns and a huge number of errors, faults, and omissions.

In 1975, based on his unpleasant experiences with the OS/360 project, Brooks wrote *The Mythical Man-Month*, which became one of the most important and influential books in computer science. It described a number of principles and guidelines for software development that were revolutionary at the time. For example, "Brooks' Law" states that adding manpower to a late software project will only make it later. The "Second System Effect" holds that the second system an engineer designs is the most dangerous, because it will likely be horribly overdesigned and bloated beyond recognition. That was one of the problems with OS/360. Its immediate predecessor, the IBM 7000 operating system, was small, elegant, and completed on time. IBM mistakenly assumed that the OS/360 project would be similar.

There were many other insightful observations, and many of Brooks' ideas went on to become the bedrock of the emerging field of software engineering. In 1986 he authored another seminal essay, *No Silver Bullet*, which had an equally profound effect on people's view of software quality and productivity.

Even though *The Mythical Man-Month* is more than 30 years old, it is still a best seller and a must-read for developers. It is in the top 0.5% of all books listed on Amazon.com, an amazing statistic in a field where anything 18 months old is viewed as ancient!

Brooks is the Kenan Professor of Computer Science at the University of North Carolina (he founded the department in 1965), a member of the National Academy of Science and the National Academy of Engineering, and a 1999 winner of the ACM Turing Award, the highest award in computer science, for fundamental contributions to software engineering.

1.3.1 Using this Book

This book follows the outline of the software life cycle diagrammed in Figure 1-2. Part I focuses on the design phase as it reviews the object-oriented design (OOD) philosophy. Chapter 2 investigates fundamental OOD concepts such as classes, objects, behavior, state, and inheritance. Chapter 3 looks at object-oriented programming (OOP)—specifically, Java features that support the concepts presented in Chapter 2. Chapter 4 presents a case study that uses OOD and OOP techniques to demonstrate how a complex software project can be organized and managed using a modern design strategy.

Part II looks at the algorithm and data structure selection phase of the software life cycle. Chapter 5 presents the mathematical tools needed to analyze the efficiency of algorithms, and then introduces the four fundamental categories of data representation. This taxonomy forms the basis for the discussion of linear structures in Chapter 6, hierarchical structures in Chapter 7, and set and graph structures in Chapter 8. Chapter 9 introduces a set of important Java classes called the **Java Collection Framework**, which automatically provides many of these data structures, such as lists, sets, maps, and tables. This framework allows software developers to focus not on how to *build* the best data structure to solve a problem, but how to *select* the best one.

Part III looks at modern programming techniques supported in Java. These include exceptions and streams (Chapter 10), threads (Chapter 11), graphical user interfaces (Chapter 12), and networking and security (Chapter 13).

Our goal in this text is to help a person who knows coding to become a "modern software developer," someone who understands and appreciates the full range of skills required to build efficient, maintainable, high-quality software. Although this process will take far more than a single class and textbook, we hope that the ideas in the following pages will start you down this interesting path.

EXERCISES

1. Find out (possibly from your computing center staff) the total number of lines of code in some of the popular software packages used on campus, including word processors, e-mail, Web browsers, or operating systems. Using Figure 1-1, what category does each package fit into? How does this compare to the size of the programs you wrote in your first computer science class?

2. Here is a simple problem specification statement: You are given *N* numbers as input. Sort these numbers into order and print them out.

   a. Identify what information is missing from this problem statement and how it could lead to errors in the final program. These omissions would include information required to ensure that the program runs correctly in all circumstances, not just the expected ones.

   b. Make reasonable assumptions about the omissions identified in Exercise 2a and write a more thorough, complete, and unambiguous set of problem specifications.

3. Look over the problem statement in Section 1.2.1, "Problem Specifications." Can you identify additional problems with this statement besides the ones described in the text?

4. Would either of the following changes affect the code fragment shown in Figure 1-8?

   a. The implementation of the waiting line *w* is changed from an array to a linked list.

   b. A Boolean value (true, false) is changed to an integer value (0, 1) for tracking whether a teller was busy.

   What do your responses to these changes say about the maintainability of object-oriented code?

5. Propose some additional methods that you think might be appropriate for the `Teller` and `WaitingLine` classes discussed in Section 1.2.2 and listed in Figure 1-7.

6. Describe some situations in everyday life (other than software development) in which you use the idea of divide and conquer to solve a large, complex problem.

7. Assume that you are working on a spell-checking program. One of its most important data structures is the dictionary. The program looks up words to see if they are in the dictionary. If so, you assume that the word is correctly spelled; if not, it is misspelled. In addition, you must allow users to add new words to the dictionary to customize it for their needs.

We have many choices for how to store the words in the dictionary and how to search it. These choices will significantly affect the performance of your spell-checking software. Discuss what you think are the main advantages and disadvantages of each of the following choices:

a A sorted array of words

b An unsorted array of words

c An unsorted, linked list of words

8 Browse the set of packages that are available in the Java 1.5 class library. You can find them at *http://java.sun.com/j2se/1.5.0/docs/api/index.html*.

Look at the operations performed by some of these packages and then write a brief summary of how they could be helpful in creating applications in such areas as:

a Games

b Client/server network applications

c E-commerce

9 Do you agree or disagree with the following statements? Explain your answers.

a Empirical testing is an adequate technique because all you need to do is exhaustively test all possible input sequences.

b Empirical testing allows you to determine both the presence and absence of errors in a program unit.

c Empirical testing can prove the presence of errors but not their absence.

10 Assume you are given the following sequence of statements, where the B_i are any Boolean expressions and the S_i are any Java statements:

```
while (B1)      {
   S1;
   if (B2)
         S2;
   else
         if (B3)
               while (B4)    S3;
         else
               S4;
}
```

Explain exactly how you would empirically test this code fragment to ensure that it is correct. Describe the set of test data you would select and your reasons for

choosing each member of the set. What does your answer say about testing large, complex program units that contain dozens or hundreds of lines of code?

11 Interview one or two professional software developers (possibly from your computing center) to find out what percentage of their time is spent on each of the software development phases listed in Figure 1-2. Compare their answers with an estimate of how much time you spent in each phase when you were writing programs for your first class in computer science.

CHALLENGE WORK EXERCISE

Read some introductory material about the concepts of formal program verification and how to prove the correctness of a code fragment. Familiarize yourself with such issues as loop invariants, pre- and postconditions, and weakest preconditions, and then try to develop a correctness proof of the following six-line code fragment, whose purpose is to sum up the N elements of array A:

```
/*  preconditions: N ≥ 1 and A[1], ..., A[N] are defined
    postcondition: sum = A[1] + ... + A[N]     */
j = 1;
sum = A[1];
while (j < N)           {
      j = j + 1;
      sum = sum + A[j];
}
```

After you have developed a proof of correctness, write a report comparing the difficulty of developing a formal proof with that of empirically testing this same fragment. Using this example, express your opinion about the use of program verification as a viable alternative to empirical testing.

For more information on program verification, try the following helpful references:

- Steven R. Rakitin, *Software Verification and Validation: A Practitioners Guide*, Artech House Publishers, 1997.
- Michael R. A. Huth and Mark D. Ryan, *Logic in Computer Science: Modeling and Reasoning about Systems*, Cambridge University Press, 2004.