chapter 1

Software Engineering

Goals

Measurable goals for this chapter include that you should be able to

- describe software life cycle activities
- describe the goals for "quality" software
- explain the following terms: software requirements, software specifications, algorithm, information hiding, abstraction, stepwise refinement
- describe four variations of stepwise refinement
- explain the fundamental ideas of object-oriented design
- explain the relationships among classes, objects, and inheritance and show how they are implemented in Java
- explain how CRC cards are used to help with software design
- interpret a basic UML state diagram
- identify sources of software errors
- describe strategies to avoid software errors
- specify the preconditions and postconditions of a program segment or method
- show how deskchecking, code walk-throughs, and design and code inspections can improve software quality and reduce effort
- explain the following terms: acceptance tests, regression testing, verification, validation, functional domain, black box testing, white box testing
- state several testing goals and indicate when each would be appropriate
- describe several integration-testing strategies and indicate when each would be appropriate
- explain how program verification techniques can be applied throughout the software development process
- create a Java test driver program to test a simple class

At this point you have completed at least one semester of computer science course work. You can take a problem of medium complexity, design a set of objects that work together to solve the problem, code the method algorithms needed to make the objects work, and demonstrate the correctness of your solution.

In this chapter, we review the software process, object-oriented design, and the verification of software correctness.

1.1 The Software Process

When we consider computer programming, we immediately think of writing code in some computer language. As a beginning student of computer science, you wrote programs that solved relatively simple problems. Much of your effort went into learning the syntax of a programming language such as Java or C++: the language's reserved words, its data types, its constructs for selection and looping, and its input/output mechanisms.

You learned a programming methodology that takes you from a problem description all the way through to the delivery of a software solution. There are many design techniques, coding standards, and testing methods that programmers use to develop high-quality software. Why bother with all that methodology? Why not just sit down at a computer and enter code? Aren't we wasting a lot of time and effort, when we could just get started on the "real" job?

If the degree of our programming sophistication never had to rise above the level of trivial programs (like summing a list of prices or averaging grades), we might get away with such a code-first technique (or, rather, a *lack* of technique). Some new programmers work this way, hacking away at the code until the program works more or less correctly—usually less!

As your programs grow larger and more complex, you must pay attention to other software issues in addition to coding. If you become a software professional, you may work as part of a team that develops a system containing tens of thousands, or even millions, of lines of code. The activities involved in such a software project's whole "life cycle" clearly go beyond just sitting down at a computer and writing programs. These activities include:

- *Problem analysis* Understanding the nature of the problem to be solved
- *Requirements elicitation* Determining exactly what the program must do
- *Software specification* Specifying what the program must do (the functional requirements) and the constraints on the solution approach (nonfunctional requirements, such as what language to use)
- *High-* and *low-level design* Recording how the program meets the requirements, from the "big picture" overview to the detailed design
- *Implementation of the design* Coding a program in a computer language
- *Testing and verification* Detecting and fixing errors and demonstrating the correctness of the program
- *Delivery* Turning over the tested program to the customer or user (or instructor)

- *Operation* Actually using the program
- *Maintenance* Making changes to fix operational errors and to add or modify the function of the program

Software development is not simply a matter of going through these steps sequentially. Many activities take place concurrently. We may be coding one part of the solution while we're designing another part, or defining requirements for a new version of a program while we're still testing the current version. Often a number of people work on different parts of the same program simultaneously. Keeping track of all these activities requires planning.

We use the term software engineering to refer to the discipline concerned with all aspects of the development of high-quality software systems. It encompasses *all* variations of techniques used during the software life cycle plus supporting activities such as documentation and teamwork. A software process is a specific set of inter-related software engineering techniques used by a person or organization to create a system.

Software engineering The discipline devoted to the design, production, and maintenance of computer programs that are developed on time and within cost estimates, using tools that help to manage the size and complexity of the resulting software products

Software process A standard, integrated set of software engineering tools and techniques used on a project or by an organization

What makes our jobs as programmers or software engineers challenging is the tendency of software to grow in size and complexity and to change at every stage of its development. Part of a good software process is the use of tools to manage this size and complexity. Usually a programmer has several toolboxes, each containing tools that help to build and shape a software product.

Hardware
One toolbox contains the hardware itself: the computers and their peripheral devices (such as monitors, terminals, storage devices, and printers), on which and for which we develop software.

Software
A second toolbox contains various software tools: operating systems, editors, compilers, interpreters, debugging programs, test-data generators, and so on. You've used some of these tools already.

Ideaware
A third toolbox is filled with the knowledge that software engineers have collected over time. This box contains the algorithms that we use to solve common programming problems, as well as data structures for modeling the information processed by our programs. Recall that an algorithm is a step-by-step description of the solution to a problem.

Algorithm A logical sequence of discrete steps that describes a complete solution to a given problem computable in a finite amount of time and space

Ideaware contains programming methodologies, such as object-oriented design, and software concepts, including information hiding, data encapsulation, and abstraction. It includes aids for creating designs such as CRC (Classes, Responsibilities, and Collaborations) cards and methods for describing designs such as the UML (Unified Modeling Language). It also contains tools for measuring, evaluating, and proving the correctness of our programs. We devote most of this book to exploring the contents of this third toolbox.

Some might argue that using these tools takes the creativity out of programming, but we don't believe that to be true. Artists and composers are creative, yet their innovations are grounded in the basic principles of their crafts. Similarly, the most creative programmers build high-quality software through the disciplined use of basic programming tools.

Goals of Quality Software

Quality software is much more than a program that accomplishes its task. A good program achieves the following goals:

1. It works.
2. It can be modified without excessive time and effort.
3. It is reusable.
4. It is completed on time and within budget.

It's not easy to meet these goals, but they are all important.

Goal 1: Quality Software Works

A program must accomplish its task, and it must do it correctly and completely. Thus, the first step is to determine exactly what the program is required to do. You need to have a definition of the program's requirements. For students, the requirements often are included in the instructor's problem description. For programmers on a government contract, the requirements document may be hundreds of pages long.

Requirements A statement of what is to be provided by a computer system or software product

Software specification A detailed description of the function, inputs, processing, outputs, and special requirements of a software product. It provides the information needed to design and implement the program.

We develop programs that meet the requirements by fulfilling software specifications. The specifications indicate the format of the input and output, details about processing, performance measures (how fast? how big? how accurate?), what to do in case of errors, and so on. The specifications tell *what* the program does, but not *how* it is done. Sometimes your instructor provides detailed specifications; other times you have to write them yourself, based on a problem description, conversations with your instructor, or intuition.

How do you know when the program is right? A program has to be

- *complete:* it should "do everything" specified
- *correct:* it should "do it right"
- *usable:* its user interface should be easy to work with
- *efficient:* at least as efficient as "it needs to be"

For example, if a desktop-publishing program cannot update the screen as rapidly as the user can type, the program is not as efficient as it needs to be. If the software isn't efficient enough, it doesn't meet its requirements, and thus, according to our definition, it doesn't work correctly.

Goal 2: Quality Software Can Be Modified

When does software need to be modified? Changes occur in every phase of its existence.

Software is changed in the design phase. When your instructor or employer gives you a programming assignment, you begin to think of how to solve the problem. The next time you meet, however, you may be notified of a change in the problem description.

Software is changed in the coding phase. You make changes in your program because of compilation errors. Sometimes you see a better solution to a part of the problem after the program has been coded, so you make changes.

Software is changed in the testing phase. If the program crashes or yields wrong results, you must make corrections.

In an academic environment, the life of the software typically ends when a program is turned in for grading. When software is developed for actual use, however, many changes can be required during the maintenance phase. Someone may discover an error that wasn't uncovered in testing, someone else may want to include additional functionality, a third party may want to change the input format, and a fourth party may want to run the program on another system.

The point is that software changes often and in all phases of its life cycle. Knowing this, software engineers try to develop programs that are easy to modify. Modifications to programs often are not even made by the original authors but by subsequent maintenance programmers. Someday you may be the one making the modifications to someone else's program.

What makes a program easy to modify? First, it should be readable and understandable to humans. Before it can be changed, it must be understood. A well-designed, clearly written, well-documented program is certainly easier for human readers to understand. The number of pages of documentation required for "real-world" programs usually exceeds the number of pages of code. Almost every organization has its own policy for documentation.

Second, it should be able to withstand small changes easily. The key idea is to partition your programs into manageable pieces that work together to solve the problem, yet are relatively independent. The design methodologies reviewed later in this chapter should help you write programs that meet this goal.

Goal 3: Quality Software Is Reusable

It takes time and effort to create quality software. Therefore, it is important to receive as much value from the software as possible.

One way to save time and effort when building a software solution is to reuse programs, classes, methods, and so on from previous projects. By using previously designed and tested code, you arrive at your solution sooner and with less effort. Alternatively, when you create software to solve a problem, it is sometimes possible to structure that software so it can help solve future, related problems. By doing this, you are gaining more value from the software created.

Creating reusable software does not happen automatically. It requires extra effort during the specification and design of the software. Reusable software is well documented and easy to read, so that it is easy to tell if it can be used for a new project. It usually has a simple interface so that it can easily be plugged into another system. It is modifiable (Goal 2), in case a small change is needed to adapt it to the new system.

When creating software to fulfill a narrow, specific function, you can sometimes make the software more generally useable with a minimal amount of extra effort. Therefore, you increase the chances that you will reuse the software later. For example, if you are creating a routine that sorts a list of integers into increasing order, you might generalize the routine so that it can also sort other types of data. Furthermore, you could design the routine to accept the desired sort order, increasing or decreasing, as a parameter.

One of the main reasons for the rise in popularity of object-oriented approaches is that they lend themselves to reuse. Previous reuse approaches were hindered by inappropriate units of reuse. If the unit of reuse is too small, then the work saved is not worth the effort. If the unit of reuse is too large, then it is difficult to combine it with other system elements. Object-oriented classes, when designed properly, can be very appropriate units of reuse. Furthermore, object-oriented approaches simplify reuse through class inheritance, which is described later in this chapter.

Goal 4: Quality Software Is Completed on Time and within Budget

You know what happens in school when you turn your program in late. You probably have grieved over an otherwise perfect program that received only half credit—or no credit at all—because you turned it in one day late. "But the network was down for five hours last night!" you protest.

Although the consequences of tardiness may seem arbitrary in the academic world, they are significant in the business world. The software for controlling a space launch must be developed and tested before the launch can take place. A patient database system for a new hospital must be installed before the hospital can open. In such cases, the program doesn't meet its requirements if it isn't ready when needed.

"Time is money" may sound trite but failure to meet deadlines is *expensive*. A company generally budgets a certain amount of time and money for the development of a piece of software. If part of a project is only 80% complete when the deadline arrives, the company must pay extra to finish the work. If the program is part of a contract with a customer, there may be monetary penalties for missed deadlines. If it is being developed for commercial sales, the company may be beaten to the market by a competitor and be forced out of business.

Once you know what your goals are, what can you do to meet them? Where should you start? There are many tools and techniques that software engineers use. In the next few sections of this chapter, we focus on a review of techniques to help you understand, design, and code programs.

Specification: Understanding the Problem

No matter what programming design technique you use, the first steps are the same. Imagine the following situation. On the third day of class, you are given a 12-page description of Programming Assignment 1, which must be running perfectly and turned in by noon, a week from yesterday. You read the assignment and realize that this program is three times larger than any program you have ever written. Now, what is your first step?

The responses listed here are typical of those given by a class of students in such a situation:

1. Panic and do nothing	39%
2. Panic and drop the course	30%
3. Sit down at the computer and begin typing	27%
4. Stop and think	4%

Response 1 is a predictable reaction from students who have not learned good programming techniques. Students who adopt Response 2 find their education progressing rather slowly. Response 3 may seem to be a good idea, especially considering the deadline looming. Resist the temptation, though, to immediately begin coding; the first step is to *think*. Before you can come up with a program solution, you must understand the problem. Read the assignment, and then read it again. Ask questions of your instructor to clarify the assignment. Starting early affords you many opportunities to ask questions; starting the night before the program is due leaves you no opportunity at all.

One problem with coding first and thinking later is that it tends to lock you into the first solution you think of, which may not be the best approach. We have a natural tendency to believe that once we've put something in writing, we have invested too much in the idea to toss it out and start over.

Writing Detailed Specifications

Many writers experience a moment of terror when faced with a blank piece of paper—where to begin? As a programmer, however, you should always have a place to start. Using the assignment description, first write a complete definition of the problem, including the details of the expected inputs and outputs, the processing and error handling, and all the assumptions about the problem. When you finish this task, you have a *specification*—a definition of the problem that tells you what the program should do. In addition, the process of writing the specification brings to light any holes in the requirements. For instance, are embedded blanks in the input significant or can they be ignored? Do you need to check for errors in the input? On what computer system(s) is your program to run? If you get the answers to these questions at this stage, you can design and code your program correctly from the start.

Many software engineers make use of operational *scenarios* to understand requirements. A scenario is a sequence of events for *one* execution of the program. Here, for example, is a scenario that a designer might consider when developing software for a bank's automated teller machine (ATM).

1. The customer inserts a bankcard.
2. The ATM reads the account number on the card.
3. The ATM requests a PIN (personal identification number) from the customer.
4. The customer enters 5683.
5. The ATM successfully verifies the account number and PIN combination.

6. The ATM asks the customer to select a transaction type (deposit, show balance, withdrawal, or quit).
7. The customer selects show balance.
8. The ATM obtains the current account balance ($1,204.35) and displays it.
9. The ATM asks the customer to select a transaction type (deposit, show balance, withdrawal, or quit).
10. The customer selects quit.
11. The ATM returns the customer's bankcard.

Scenarios allow us to get a feel for the behavior expected from the system. A single scenario cannot show all possible behaviors, however, so software engineers typically prepare many different scenarios to gain a full understanding of the requirements.

Sometimes details that are not explicitly stated in the requirements may be handled according to the programmer's preference. In some cases you have only a vague description of a problem, and it is up to you to define the entire software specification; these projects are sometimes called *open problems*. In any case, you should always document assumptions that you make about unstated or ambiguous details.

The specification clarifies the problem to be solved. However, it also serves as an important piece of program documentation. Sometimes it acts as a contract between a customer and a programmer. There are many ways in which specifications may be expressed and a number of different sections that may be included. Our recommended program specification includes the following sections:

- processing requirements
- sample inputs with expected outputs
- assumptions

If special processing is needed for unusual or error conditions, it too should be specified. Sometimes it is helpful to include a section containing definitions of terms used. It is also useful to list any testing requirements so that verifying the program is considered early in the development process. In fact, a test plan can be an important part of a specification; test plans are discussed later in this chapter in the section on verification of software correctness.

1.2 Program Design

Remember, the specification of the program tells *what* the program must do, but not *how* it does it. Once you have clarified the goals of the program, you can begin the design phase of the software life cycle. In this section, we review some ideaware tools that are used for software design and present a review of object-oriented design constructs and methods.

Tools

Abstraction

The universe is filled with complex systems. We learn about such systems through *models*. A model may be mathematical, like equations describing the motion of satellites around the earth. A physical object such as a model airplane used in wind-tunnel tests is another form of model. Only the characteristics of the system that are essential to the problem being studied are modeled; minor or irrelevant details are ignored. For example, although the earth is an oblate ellipsoid, globes (models of the earth) are spheres. The small difference in shape is not important to us in studying the political divisions and physical landmarks on the earth. Similarly, in-flight movies are not included in the model airplanes used to study aerodynamics.

An abstraction is a model of a complex system that includes only the essential details. Abstractions are the fundamental way that we manage complexity. Different viewers use different abstractions of a particular system. Thus, while we see a car as a means of transportation, the automotive engineer may see it as a large mass with a small contact area between it and the road (Figure 1.1).

Abstraction A model of a complex system that includes only the details essential to the perspective of the viewer of the system

What does abstraction have to do with software development? The programs we write are abstractions. A spreadsheet program used by an accountant models the books used to record debits and credits. An educational computer game about wildlife models an ecosystem. Writing software is difficult because both the systems we model and the processes we use to develop the software are complex. One of our major goals is to convince you to use abstractions to manage the complexity of developing software. In nearly every chapter, we make use of abstractions to simplify our work.

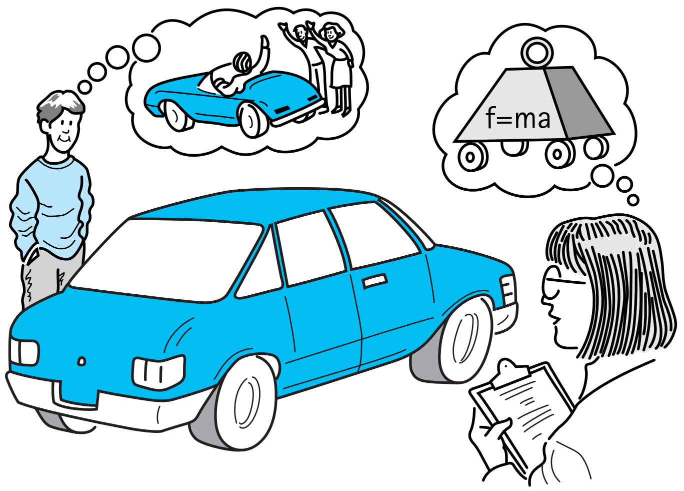


Figure 1.1 *An abstraction includes the essential details relative to the perspective of the viewer*

Information Hiding

Many design methods are based on decomposing a problem's solution into modules. By "module" we mean a cohesive system subunit that performs a share of the work. In Java, the primary module mechanism is the *class*. Decomposing a system into modules helps us manage complexity. Additionally, the modules can form the basis of assignments for different programming teams working separately on a large system.

Modules act as an abstraction tool. The complexity of their internal structure can be hidden from the rest of the system. This means that the details involved in implementing a module are isolated from the details of the rest of the system. Why is hiding the details desirable? Shouldn't the programmer know everything? *No!* Information hiding helps manage the complexity of a system since a programmer can concentrate on one module at a time.

Information hiding The practice of hiding the details of a module with the goal of controlling access to the details from the rest of the system

Of course, a program's modules are interrelated, since they work together to solve the problem. Modules provide services to each other through a carefully defined interface. The interface in Java is usually provided by the public methods of a class. Programmers of one module do not need to know the internal details of the modules it interacts with, but they do need to know the interfaces. Consider a driving analogy—you can start a car without knowing how many cylinders are in the engine. You don't need to know these lower-level details of the car's power subsystem in order to start it. You just have to understand the interface; that is, you only need to know how to turn the key.

Similarly, you don't have to know the details of other modules as you design a specific module. Such a requirement would introduce a greater risk of confusion and error throughout the whole system. For example, imagine what it would be like if every time we wanted to start our car, we had to think, "The key makes a connection in the ignition switch that, when the transmission safety interlock is in "park," engages the starter motor and powers up the electronic ignition system, which adjusts the spark and the fuel-to-air ratio of the injectors to compensate for…".

Besides helping us manage the complexity of a large system, abstraction and information hiding support our quality goals of modifiability and reusability. In a well-designed system, most modifications can be localized to just a few modules. Such changes are much easier to make than changes that permeate the entire system. Additionally, a good system design results in the creation of generic modules that can be used in other systems.

To achieve these goals, modules should be good abstractions with strong *cohesion*; that is, each module should have a single purpose or identity and the module should stick together well. A cohesive module can usually be described by a simple sentence. If you have to use several sentences or one very convoluted sentence to describe your module, it is probably *not* cohesive. Each module should also exhibit information hiding so that changes within it do not result in changes in the modules that use it. This independent quality of modules is known as *loose coupling*. If your module depends on the internal details of other modules, it is *not* loosely coupled.

But what should these modules be and how do we identify them? That question is addressed in the subsection on object-oriented design later in this chapter.

Stepwise Refinement

In addition to concepts such as abstraction and information hiding, software developers need practical approaches to conquer complexity. Stepwise refinement is a widely applicable approach. It has many variations such as top-down, bottom-up, functional decomposition and even "round-trip gestalt design." Undoubtedly, you have learned a variation of stepwise refinement in your studies, since it is a standard method for organizing and writing essays, term papers, and books. For example, to write a book an author first determines the main theme and the major subthemes. Next, the chapter topics can be identified, followed by section and subsection topics. Outlines can be produced and further refined for each subsection. At some point the author is ready to add detail—to actually begin writing sentences.

In general, with stepwise refinement, a problem is approached in stages. Similar steps are followed during each stage, with the only difference being the level of detail involved. The completion of each stage brings us closer to solving our problem. Let's look at some variations of stepwise refinement:

- Top-down: First the problem is broken into several large parts. Each of these parts is in turn divided into sections, then the sections are subdivided, and so on. The important feature is that *details are deferred as long as possible* as we move from a general to a specific solution. The outline approach to writing a book is a form of top-down stepwise refinement.
- Bottom-up: As you might guess, with this approach the details come first. It is the opposite of the top-down approach. After the detailed components are identified and designed, they are brought together into increasingly higher-level components. This could be used, for example, by the author of a cookbook who first writes all the recipes and then decides how to organize them into sections and chapters.
- Functional decomposition: This is a program design approach that encourages programming in logical action units, called functions. The main module of the design becomes the main program (also called the main function), and subsections develop into functions. This *hierarchy of tasks* forms the basis for functional decomposition, with the main program or function controlling the processing. Functional decomposition is not used for overall system design in the object-oriented world. However, it can be used to design the algorithms that implement object methods. The general function of the method is continually divided into sub-functions until the level of detail is fine enough to code. Functional decomposition is top-down stepwise refinement with an emphasis on functionality.
- Round-trip gestalt design: This confusing term is used to define the stepwise refinement approach to object-oriented design suggested by Grady Booch,[1] one of the leaders of the object movement. First, the tangible items and events in the problem domain are identified and assigned to candidate classes and objects.

[1]Grady Booch, *Object Oriented Design with Applications* (Redwood City, CA: Benjamin Cummings, 1991).

Next the external properties and relationships of these classes and objects are defined. Finally, the internal details are addressed, and unless these are trivial, the designer must return to the first step for another round of design. This approach is top-down stepwise refinement with an emphasis on objects and data.

Good designers typically use a combination of the stepwise refinement techniques described here.

Visual Aids

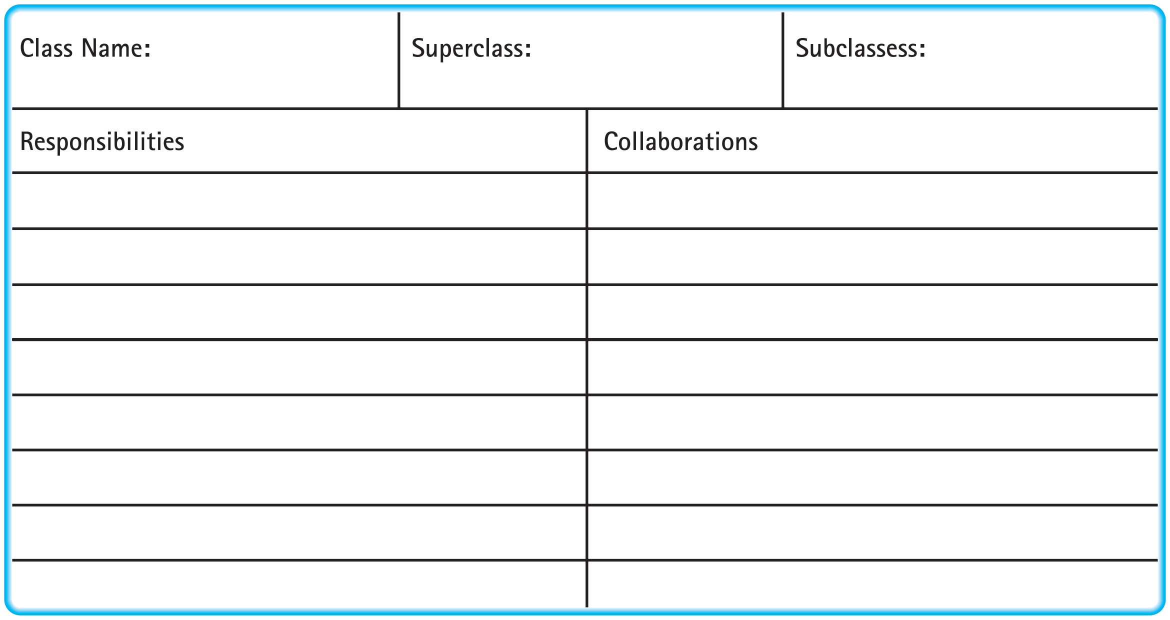
Abstraction, information hiding, and stepwise refinement are inter-related methods for controlling complexity during the design of a system. We will now look at some tools that we can use to help us visualize our designs. Diagrams are used in many professions. For example, architects use blueprints, investors use market trend graphs, and truck drivers use maps.

Software engineers use different types of diagrams and tables. Here, we introduce the *Unified Modeling Language* (*UML*) and *Class*, *Responsibility*, *and Collaboration* (*CRC*) *cards*, both of which are used throughout this text.

The UML is used to specify, visualize, construct, and document the components of a software system. It combines the best practices that have evolved over the past several decades for modeling systems, and is particularly well-suited to modeling object-oriented designs. UML diagrams are another form of abstraction. They hide implementation details and allow us to concentrate only on the major design components. UML includes a large variety of interrelated diagram types, each with its own set of icons and connectors. It is a very powerful development and modeling tool.

Covering all of UML is beyond the scope of this text.[2] We use only one UML diagram type, *detailed class diagrams*, to describe some of our designs. Examples are

[2]The official definition of the UML is maintained by the Object Management Group. Detailed information can be found at `http://www.omg.org/uml/`.



Class Name:	Superclass:	Subclassess:

Responsibilities	Collaborations

Figure 1.2 *A blank CRC card*

shown beginning on pages 16. The notation of the class diagrams is introduced as needed throughout the text.

UML class diagrams are good for modeling our designs after we have developed them. In contrast, CRC cards help us determine our designs in the first place. CRC cards were first described by Beck and Cunningham[3] in 1989 as a means of allowing object-oriented programmers to identify a set of cooperating classes to solve a problem.

A programmer uses a physical 4” × 6” index card to represent each class that has been identified as part of a problem solution. Figure 1.2 shows a blank CRC card. It contains room for the following information about a class:

1. Class name
2. Responsibilities of the class—usually represented by verbs and implemented by public methods
3. Collaborations—other classes/objects that are used in fulfilling the responsibilities

Thus the name CRC card. We have added fields to the original design of the card for the programmer to record superclass and subclass information, and the primary responsibility of the class.

[3]Beck and Cunningham: `http://c2.com/doc/oopsla89/paper.html`.

CRC cards are a great tool for refining an object-oriented design, especially in a team programming environment. They provide a physical manifestation of the building blocks of a system, allowing programmers to walk through user scenarios, identifying and assigning responsibilities and collaborations. The example in the next subsection demonstrates the use of CRC cards for design.

Object-Oriented Design

Review

Before describing approaches to object-oriented design, we present a short review of object-oriented programming. We use Java code to support this review.

The object-oriented paradigm is founded on three inter-related constructs: classes, objects, and inheritance. The inter-relationship among these constructs is so tight that it is nearly impossible to describe them separately. Objects are the basic run-time entities in an object-oriented system. An object is an instantiation of a class; or alternately, a class defines the structure of its objects. Classes are organized in an "is-a" hierarchy defined by inheritance. The definition of an object's behavior often depends on its position within this hierarchy. Let's look more closely at each of these constructs, using Java code to provide a concrete representation of the concepts. Java reserved words (when used as such), user-defined identifiers, class and method names, and so on appear in `this font` throughout the entire textbook.

Classes A class defines the structure of an object or a set of objects. A class definition includes variables (data) and methods (actions) that determine the behavior of an object. The following Java code defines a `Date` class that can be used to manipulate `Date` objects, for example, in a course scheduling system. The `Date` class can be used to create `Date` objects and to learn about the year, month, or day of any particular `Date` object.[4] Within the comments the word "this" is used to represent the current object.

```
public class Date
{
  protected int year;
  protected int month;
  protected int day;
  protected static final int MINYEAR = 1583;

  public Date(int newMonth, int newDay, int newYear)
  // Initializes this Date with the parameter values
```

[4]The Java library includes a `Date` class, `java.util.Date`. However, the familiar properties of dates make them a natural example to use in explaining object-oriented concepts. So we ignore the existence of the library class, as if we must design our own `Date` class.

```
  {
    month = newMonth;
    day = newDay;
    year = newYear;
   }

  public int yearIs()
  // Returns the year value of this Date
  {
    return year;
  }

  public int monthIs()
  // Returns the month value of this Date
  {
    return month;
  }

  public int dayIs()
  // Returns the day value of this Date
  {
    return day;
  }
}
```

The `Date` class demonstrates two kinds of variables: instance variables and class variables. The instance variables of this class are `year`, `month`, and `day`. Their values vary for each different instance of an object of the class. Instance variables represent the `attributes` of an object. `MINYEAR` is a class variable because it is defined to be static. It is associated directly with the `Date` class, instead of with objects of the class. A single copy of a static variable is maintained for all the objects of the class.

Remember that the `final` modifier states that a variable is in its final form and cannot be modified; thus `MINYEAR` is a constant. By convention, we use only capital letters when naming constants. It is standard procedure to declare constants as *static* variables. Since the value of the variable cannot change, there is no need to force every object of a class to carry around its own version of the value. In addition to holding shared constants, static variables can also be used to maintain information that is common to an entire class. For example, a Bank Account class may have a static variable that holds the number of current accounts.

In the above example, the `MINYEAR` constant represents the first full year that the widely used Gregorian calendar was in effect. The idea here is that programmers should not use the class to represent dates that predate that year. We look at ways to enforce this rule in Chapter 2.

The methods of the class are `Date`, `yearIs`, `monthIs`, and `dayIs`. Note that the `Date` method has the same name as the class. Recall that this means it is a special type

Observer A method that returns an observation on the state of an object.

of method, called a class constructor. Constructors are used to create new instances of a class—to instantiate objects of a class. The other three methods are classified as observer methods since they "observe" and return instance variable values. Another name for observer methods is "accessor" methods.

Once a class such as `Date` has been defined, a program can create and use objects of that class. The effect is similar to expanding the language's set of standard types to include a `Date` type—we discuss this idea further in Chapter 2. The UML class diagram for the `Date` class is shown in Figure 1.3. Note that the name of the class appears in the top section of the diagram, the variables appear in the next section, and the methods appear in the final section. The diagram includes information about the nature of the variables and method parameters; for example, we can see at a glance that `year`, `month`, and `day` are all of type `int`. Note that the variable `MINYEAR` is underlined, which indicates that it is a class variable rather than an instance variable. The diagram also indicates the visibility or protection associated with each part of the class (+ is public, # = protected)—we discuss visibility and protection in Chapter 2.

Objects Objects are created from classes at run-time. They can contain and manipulate data. You should view an object-oriented system as a set of objects, working together by sending each other messages to solve a problem.

To create an object in Java we use the *new* operator, along with the class constructor as follows:

```
Date myDate = new Date(6, 24, 1951);
Date yourDate = new Date(10, 11, 1953);
Date ourDate = new Date(6, 15, 1985);
```

We say that the variables `myDate`, `yourDate`, and `ourDate` reference "objects of the class `Date`" or simply "objects of type `Date`." We could also refer to them as "`Date` objects."

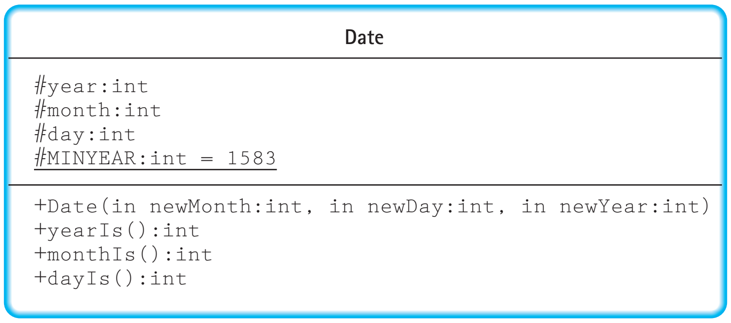


Figure 1.3 *UML class diagram for the* `Date` *class*

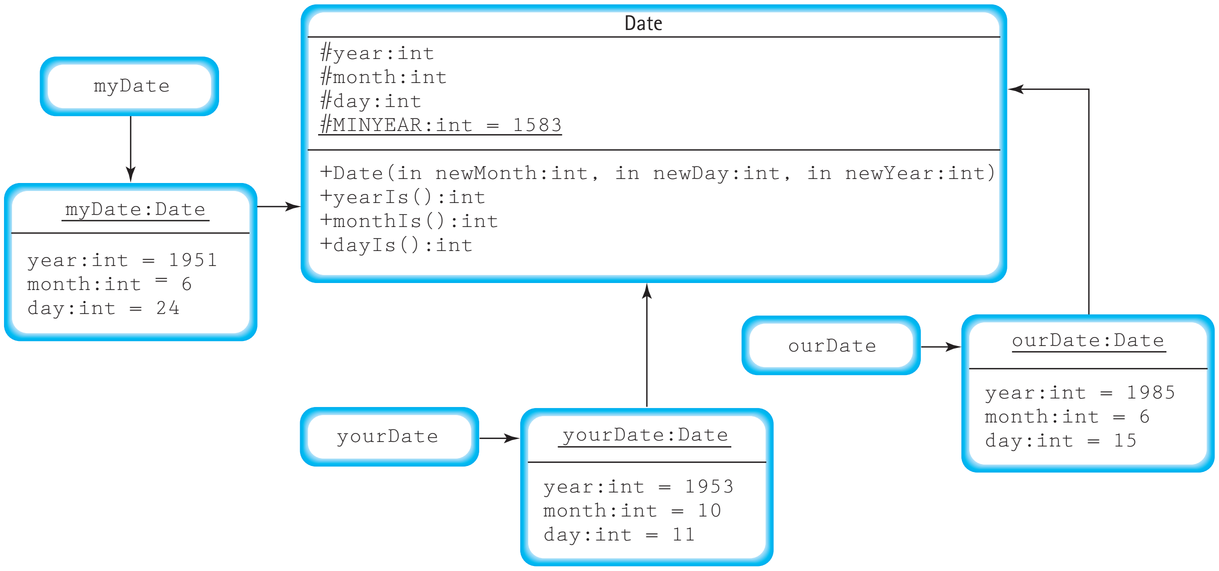


Figure 1.4 *Extended UML class diagram showing `Date` objects*

In Figure 1.4 we have extended the standard UML class diagram to show the relationship between the instantiated `Date` objects and the `Date` class.

As you can see, the objects are concrete instantiations of the class. Notice that the `myDate`, `yourDate`, and `ourDate` variables are not objects, but actually hold references to the objects. The references are shown by the pointers from the variable boxes to the objects. In reality, references are memory addresses. The memory address of the instantiated object is stored in the memory location assigned to the variable. If no object has been instantiated for a particular variable, then its memory location holds a `null` reference.

Object methods are invoked through the object upon which they are to act. For example, to assign the value of the `year` variable of `ourDate` to the integer variable `theYear`, a programmer would code

```
theYear = ourDate.yearIs();
```

Inheritance The object-oriented paradigm provides a powerful reuse tool called *inheritance*, which allows programmers to create a new class that is a specialization of an existing class. In this case, the new class is called a subclass of the existing class, which in turn is the superclass of the new class.

A subclass "inherits" features from its superclass. It adds new features, as needed, related to its specialization. It can also redefine inherited features as necessary. Contrary to the intuitive meaning of super and sub, a subclass usually has more variables and methods than its superclass. Super and sub refer to the relative positions of the classes in a hierarchy. A subclass is below its superclass, and a superclass is above its subclasses.

Suppose we already have a `Date` class as defined above, and we are creating a new application to manipulate `Date` objects. Suppose also that in the new application we are often required to "increment" a `Date` variable—to change a `Date` variable so that it represents the next day. For example, if the `Date` object represents 7/31/2001, it would represent 8/1/2001 after being incremented. The algorithm for incrementing the date is not trivial, especially when you consider leap-year rules. But in addition to developing the algorithm, we must address another question: where to implement the algorithm. There are several options:

- Implement the algorithm within the new application. The code would need to obtain the month, day, and year from the `Date` object using the observer methods, calculate the new month, day, and year, instantiate a new `Date` object to hold the updated month, day, and year, and assign it to the same variable. This might appear to be a good approach, since it is the new application that requires the new functionality. However, if future applications also need this functionality, their programmers have to reimplement the solution for themselves. This approach does not support our goal of reusability.
- Add a new method, called `increment`, to the `Date` class. The code would use the incrementing algorithm to update the `month`, `year`, and `day` values of the current object. This approach is better than the previous approach because it allows any future programs that use the `Date` class to use the new functionality. However, this also means that *every* application that uses the `Date` class can use this method. In some cases, a programmer may have chosen to use the `Date` class because of its built-in protection against changes to the object variables. Such objects are said to be immutable. Adding an `increment` method to the `Date` class undermines this protection, since it allows the variables to be changed.
- Use inheritance. Create a new class, called `IncDate`, that inherits all the features of the current `Date` class, but that also provides the `increment` method. This approach resolves the drawbacks of the previous two approaches. We now look at how to implement this third approach.

We often call the inheritance relationship an *is a* relationship. In this case we would say that an object of the class `IncDate` is also a `Date` object, since it can do anything that a `Date` object can do—and more. This idea can be clarified by remembering that inheritance typically means specialization. `IncDate` is a special case of `Date`, but not the other way around.

To create `IncDate` in Java we would code:

```
public class IncDate extends Date
{
  public IncDate(int newMonth, int newDay, int newYear)
  // Initializes this IncDate with the parameter values
```

```
    {
      super(newMonth, newDay, newYear);
     }

    public void increment()
    // Increments this IncDate to represent the next day, i.e.,
    // this = (day after this)
    // For example if this = 6/30/2003 then this becomes 7/1/2003
    {
      // Increment algorithm goes here
    }
}
```

Note: sometimes in code listings we emphasize the sections of code most pertinent to the current discussion by underlining them.

Inheritance is indicated by the keyword `extends`, which shows that `IncDate` inherits from `Date`. It is not possible in Java to inherit constructors, so `IncDate` must supply its own. In this case, the `IncDate` constructor simply takes the month, day, and year parameters and passes them to the constructor of its superclass; it passes them to the `Date` class constructor using the `super` reserved word.

The other part of the `IncDate` class is the new `increment` method, which is classified as a transformer method, because it changes the internal state of the object. `increment` changes the object's `day` and possibly the `month` and `year` values. The `increment` transformer method is invoked through the object that it is to transform. For example, the statement

Transformer A method that changes the internal state of an object

```
ourDate.increment();
```

transforms the `ourDate` object.

Note that we have left out the details of the `increment` method since they are not crucial to our current discussion.

A program with access to both of the date classes can now declare and use both `Date` and `IncDate` objects. Consider the following program segment. (Assume `output` is one of Java's `PrintWriter` file objects.)

```
Date myDate = new Date(6, 24, 1951);
IncDate aDate = new IncDate(1, 11, 2001);

output.println("mydate day is:    " + myDate.dayIs());
output.println("aDate day is:     " + aDate.dayIs());

aDate.increment();
output.println("the day after is: " + aDate.dayIs());
```

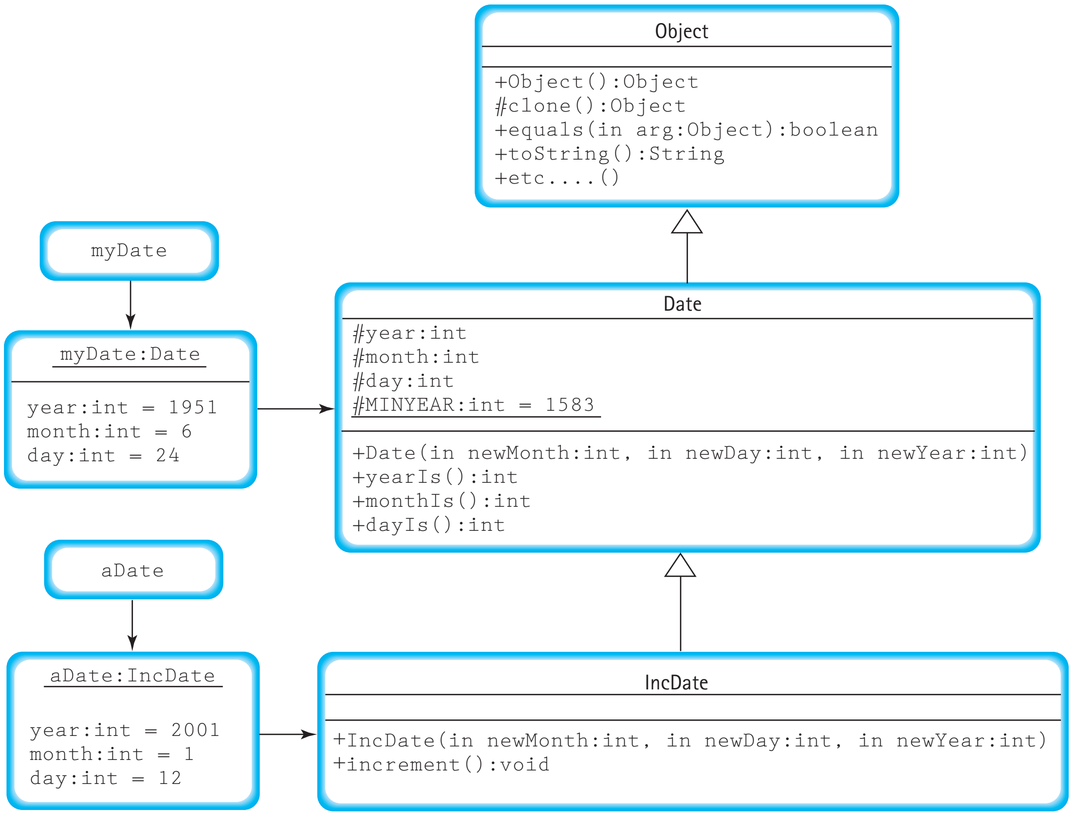


Figure 1.5 *Extended UML class diagram showing inheritance*

This program segment instantiates and initializes `myDate` and `aDate`, outputs the values of their days, increments `aDate` and finally outputs the new day value of `aDate`. You might ask, "How does the system resolve the use of the `dayIs` method by an `IncDate` object when `dayIs` is defined in the `Date` class?" Understanding how inheritance is supported by Java provides the answer to this question. The extended UML diagram in Figure 1.5 shows the inheritance relationships and captures the state of the system after the `aDate` object has been incremented. This figure helps us investigate the situation.

The compiler has available to it all the declaration information captured in the extended UML diagram. Consider the `dayIs` method call in the statement:

```
output.println("aDate day is:    " + aDate.dayIs());
```

To resolve this method call, the compiler follows the reference from the `aDate` variable to the `IncDate` class. Since it does not find a definition for a `dayIs` method in the `IncDate` class, it follows the inheritance link to the superclass `Date`, where it finds, and links to, the `dayIs` method. In this case, the `dayIs` method returns an `int` value that represents the `day` value of the `aDate` object. During execution, the system changes the `int` value to a `String`, concatenates it to the string "aDate day is: " and prints it to `output`.

Note that because of the way method calls are resolved, by searching *up* the inheritance tree, only objects of the class `IncDate` can use the `increment` method. If you tried to use the `increment` method on an object of the class `Date`, such as the `myDate` object, there would be no definition available in either the `Date` class or any of the classes above `Date` in the inheritance tree. The compiler would report a syntax error in this situation.

Notice the `Object` class in the diagram. Where did it come from? In Java, any class that does not explicitly extend another class implicitly extends the predefined `Object` class. Since `Date` does not explicitly extend any other class, it inherits directly from `Object`. The `Date` class is a subclass of `Object`. The solid arrows with the hollow arrowheads indicate inheritance in a UML diagram.

All Java classes can trace their roots back to the `Object` class, which is so general that it does almost nothing; objects of the class `Object` are nearly useless by themselves. But `Object` does define several basic methods: comparison for equality (`equals`), conversion to a string (`toString`), and so on. Therefore, for example, any object in any Java program supports the method `toString`, since it is inherited from the `Object` class.

Just as Java automatically changes an integer value to a string in a statement like

```
output.println("aDate day is:    " + aDate.dayIs());
```

it automatically changes an object to a string in a statement like

```
output.println("tomorrow: " + aDate);
```

If you use an object as a string anywhere in a Java program, then the Java compiler automatically looks for a `toString` method for that object. In this case, the `toString` method is not found in the `IncDate` class, nor is it found in its superclass, the `Date` class. However, the compiler continues looking up the inheritance hierarchy, and finds the `toString` method in the `Object` class. Since all classes trace their roots back to `Object`, the compiler is always guaranteed to find a `toString` method eventually.

But, wait a minute. What does it mean to "change an object to a string"? Well, that depends on the definition of the `toString` method that is associated with the object. The `toString` method of the `Object` class returns a string representing some of the internal system implementation details about the object. This information is somewhat cryptic and generally not useful to us. This is an example of where it is useful to redefine an inherited method. We generally override the default `toString` method when creating our own classes, to return a more relevant string. For example, the following `toString` method could be added to the definition of the `Date` class:

```
public String toString()
{
  return(month + "/" + day + "/" + year);
}
```

Now, when the compiler needs a `toString` method for a `Date` object (or an `IncDate` object), it finds the method in the `Date` class and returns a more useful string. Figure 1.6 shows the output from the following program segment.

```
Date myDate = new Date(6, 24, 1951);
IncDate currDate = new IncDate(1, 11, 2001);

output.println("mydate:   " + myDate);
output.println("today:    " + currDate);

currDate.increment();
output.println("tomorrow: " + currDate);
```

The results on the left show the output generated if the `toString` method of the `Object` class is used by default; and on the right if the `toString` method above is added to the `Date` class:

Object class *toString* Used

```
mydate:    Date@256a7c
today:     IncDate@720eeb
tomorrow:  IncDate@720eeb
```

Date class *toString* Used

```
mydate:    6/24/1951
today:     1/11/2001
tomorrow:  1/12/2001
```

Figure 1.6 *Output from program segment*

One last note: Remember that subclasses are assignment compatible with the superclasses above them in the inheritance hierarchy. Therefore, in our example, the statement

```
myDate = currDate;
```

would be legal, but the statement

```
currDate = myDate;
```

would cause an "incompatible type" syntax error.

Design

The *object-oriented design* (OOD) methodology originated with the development of programs to simulate physical objects and processes in the real world. For example, to simulate an electronic circuit, you could develop a class for simulating each kind of component in the circuit and then "wire-up" the simulation by having the modules pass information among themselves in the same pattern that wires connect the electronic components.

Identifying Classes The key task in designing object-oriented systems is identification of classes. Successful class identification and organization draws upon many of the tools that we discussed earlier in this chapter. Top-down stepwise refinement encourages us to start by identifying the major classes and gradually refine our system definition to identify all the classes we need. We should use abstraction and practice information hiding by keeping the interfaces to our classes narrow and hiding important design decisions and requirements likely to change within our classes. CRC cards can help us identify the responsibilities and collaborations of our classes, and expose holes in our design. UML diagrams let us record our designs in a form that is easy to understand.

When possible, we should organize our classes in an inheritance hierarchy, to benefit from reuse. Another form of reuse is to find prewritten classes, possibly in the standard Java library, that can be used in a solution.

There is no foolproof technique for identifying classes; we just have to start brainstorming ideas and see where they lead us. A large program is typically written by a team of programmers, so the brainstorming process often occurs in a team setting. Team members identify whatever objects they see in the problem and then propose classes to represent them. The proposed classes are all written on a board. None of the ideas for classes are discussed or rejected in this first stage.

After the brainstorming, the team goes through a process of filtering the classes. First they eliminate duplicates. Then they discuss whether each class really represents an object in the problem. (It's easy to get carried away and include classes, such as "the user," that are beyond the scope of the problem.) The team then looks for classes that seem to be related. Perhaps they aren't duplicates, but they have much in common, and so they are grouped together on the board. At the same time, the discussion may reveal some classes that were overlooked.

Usually it is not difficult to identify an initial set of classes. In most large problems we naturally find entities that we wish to represent as classes. For example, in designing a program that manages a checking account, we might identify checks, deposits, an account balance, and account statements as entities. These entities interact with each other through messages. For example, a check could send a message to the balance entity that tells it to deduct an amount from itself. We didn't list the amount in our initial set of objects, but it may be another entity that we need to represent.

Our example illustrates a common approach to OOD. We begin by identifying a set of objects that we think are important in a problem. Then we consider some scenarios in which the objects interact to accomplish a task. In the process of envisioning how a scenario plays out, we identify additional objects and messages. We keep trying new scenarios until we find that our set of objects and messages is sufficient to accomplish any task that the problem requires. CRC cards help us enact such scenarios.

A standard technique for identifying classes and their methods is to look for objects and operations in the problem statement. Objects are usually nouns and operations are usually verbs. For example, suppose the problem statement includes the sentence: "The student grades must be sorted from best to worst before being output." Potential objects are "student" and "grade," and potential operations are "sort" and "output." We propose that on a printed copy of your requirements you circle the nouns and underline the verbs. The set of nouns are your candidate objects, and the verbs are your candidate methods. Of course, you have to filter this list, but at least it provides a good starting point for design.

Recall that in our discussion of abstraction and information hiding we stated that program modules should display strong cohesion. A good way to validate the cohesiveness of an identified class is to try to describe its main responsibility in a single coherent phrase. If you cannot do this, then you should reconsider your design. Some examples of cohesive responsibilities are:

- maintain a list of integers
- handle file interaction
- provide a date type

Some examples of "poor" responsibilities are:

- maintain a list of integers and provide special integer output routines
- handle file interaction and draw graphs on the screen

In summation, we have discussed the following approaches to identifying classes:

1. Start with the major classes and refine the design.
2. Hide important design decisions and requirements likely to change within a class.
3. Brainstorm with a group of programmers.
4. Make sure each class has one main responsibility.
5. Use CRC cards to organize classes and identify holes in the design.
6. Walk through user scenarios.
7. Look for nouns and verbs in the problem description.

Design Choices When working on design, keep in mind that there are many different correct solutions to most problems. The techniques we use may seem imprecise, especially in contrast with the precision that is demanded by the computer. But the computer merely demands that we express (code) a particular solution precisely. The process of deciding which particular solution to use is far less precise. It is our human ability to make choices without having complete information that enables us to solve problems. Different choices naturally lead to different solutions to a problem.

For example, in developing a simulation of an air traffic control system, we might decide that airplanes and control towers are objects that communicate with each other. Or we might decide that pilots and controllers are the objects that communicate. This choice affects how we subsequently view the problem, and the responsibilities that we assign to the objects. Either choice can lead to a working application. We may simply prefer the one with which we are most familiar.

Some of our choices lead to designs that are more or less efficient than others. For example, keeping a list of names in alphabetical rather than random order makes it possible for the computer to find a particular name much faster. However, choosing to leave the list randomly ordered still produces a valid (but slower) solution, and may even be the best solution if you do not need to search the list very often.

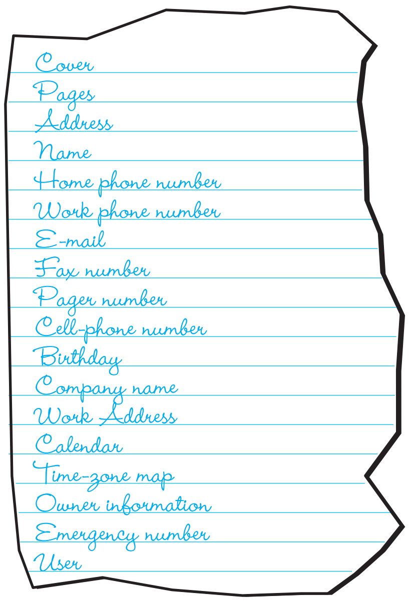
Other choices affect the amount of work that is required to develop the remainder of a problem solution. In creating a program for choreographing ballet movements, we might begin by recognizing a dancer as the important object and then create a class for each dancer. But in doing so, we discover that all of the dancers have certain common responsibilities. Rather than repeat the definition of those responsibilities for each class of dancer, we can change our initial choice and define a class for a generic dancer that includes all the common responsibilities and then develop subclasses that add responsibilities specific to each individual.

The point is, don't hesitate to begin solving a problem because you are waiting for some flash of genius that leads you to the perfect solution. There is no such thing. It is better to jump in and try something, step back, and see if you like the result, and then either proceed or make changes. In the example below we show how the CRC card technique helps you explore different design choices and keep track of them.

Design Example

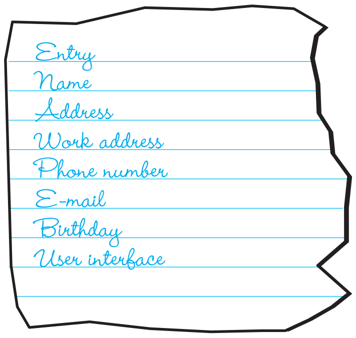
In this subsection we present a sample object-oriented design process that might be followed if we were on a small team of software engineers. Our purposes are to show the classes that might be identified for an object-oriented system, and to demonstrate the utility of CRC cards. We assume that our team of engineers has been given the task of automating an address book. A user should be able to enter and retrieve information from the address book. We have been given a sample physical address book on which to base their product.

First our team studies the problem, inspects the physical address book, and brainstorms that the application has the following potential objects:



Then we enter the filtering stage. Our application doesn't need to represent the physical parts of an address book, so we can delete Cover and Pages. However, we need something analogous to a page that holds all the same sort of information. Let's call it an Entry. The different telephone numbers can all be represented by the same kind of object. So we can combine Home, Work, Fax, Pager, and Cell-phone into a Phone number class. In consultation with the customer, we find that the electronic address book doesn't need the special pages that are often found in a printed address book, so we delete Calendar, Time-zone map, Owner information, and Emergency number.

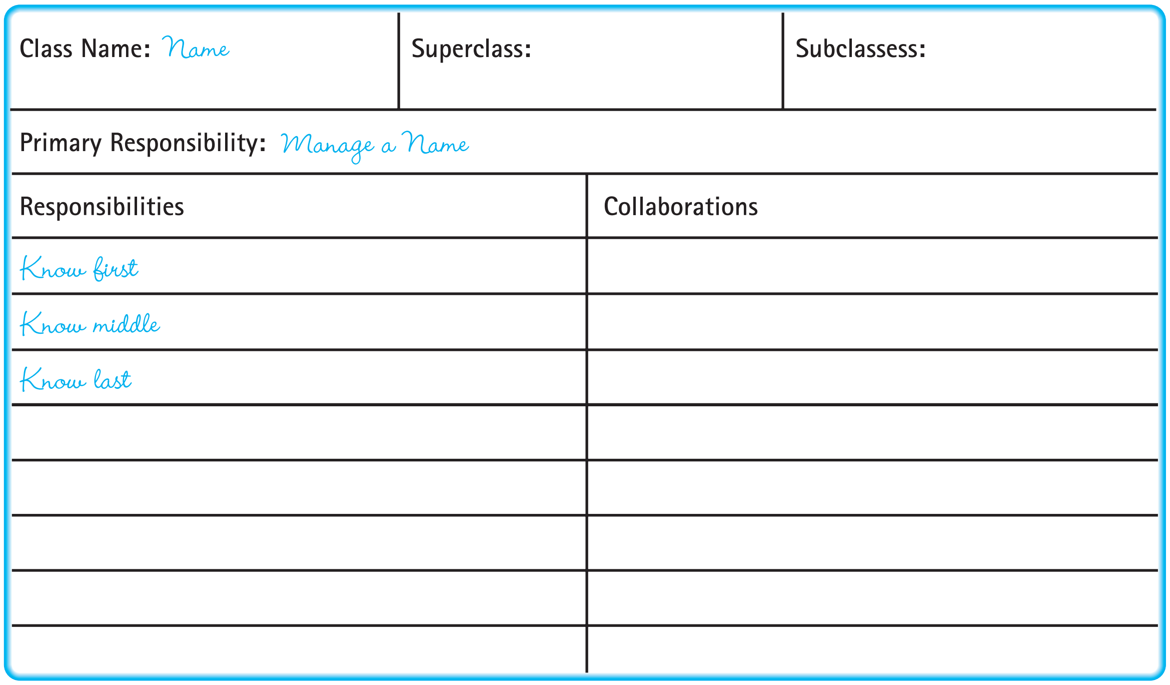
Further thought reveals that the User isn't part of the application, although this does point to the need for a User interface that we did not originally list. A Work Address is a specific kind of address that has additional information, so we can make it a subclass of Address. Company names are just Strings, so there is no need to distinguish them, but Names have a first, last, and middle part. Our filtered list of classes now looks like this.



For each of these classes we create a CRC card. In the case of Work Address, we list Address as its Superclass, and on the Address card we list Work Address in its Subclasses space.

In doing coursework, you may be asked to work individually rather than in a collaborative team. You can still do your own brainstorming and filtering. However, we recommend that you take a break after the brainstorming and do the filtering once you have let your initial ideas rest for a while. An idea that seems brilliant in the middle of brainstorming may lose some of its attraction after a day or even a few hours.

Initial Responsibilities Once you (or your team) have identified the classes and created CRC cards for them, go over each card and write down its primary responsibility and an initial list of resultant responsibilities that are obvious. For example, a Name class manages a "Name" and has a responsibility to know its first name, its middle name, and its last name. We would list these three responsibilities in the left column of its card, as shown in Figure 1.7. In an implementation, they become methods that return the corresponding part of the name. For many classes, the initial responsibilities include knowing some value or set of values.



Class Name: Name	Superclass:	Subclassess:

Primary Responsibility: Manage a Name

Responsibilities	Collaborations
Know first	
Know middle	
Know last	

Figure 1.7 *A CRC card with initial responsibilities*

A First Scenario Walk-Through To further expand the responsibilities of the classes and see how they collaborate, we must pretend to carry out various processing scenarios by hand. This kind of role-playing is known as a *walk-through*. We ask a question such as, "What happens when the user wants to find an address that's in the book?" Then we answer the question by telling how each object is involved in accomplishing this task. In a team setting, the cards are distributed among the team members. When an object of a class is doing something, its card is held in the air to visually signify that it is active.

With this particular question, we might pick up the User Interface card and say, "I have a responsibility to get the person's name from the user." That responsibility gets written down on the card. Once the name is input, the User Interface must collaborate with other objects to look up the name and get the corresponding address. What object should it collaborate with? There is no identified object class that represents the entire set of address book entries.

We've found a hole in our list of classes! The Entry objects should be organized into a Book object. We quickly write out a Book CRC card. The User Interface card-holder then says, "I'm going to collaborate with the Book class to get the address." The collaboration is written in the right column of the card, and it remains in the air. The owner of the Book card holds it up, saying, "I have a responsibility to find an address in the list of Entry objects that I keep, given a name." That responsibility gets written on the

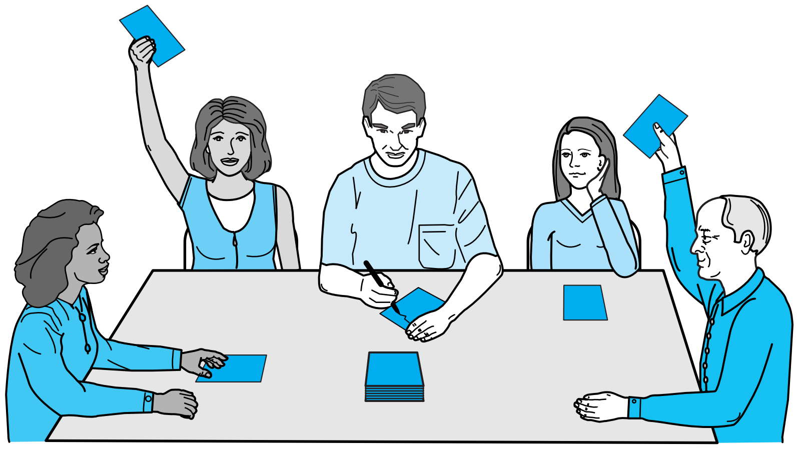


Figure 1.8 *A scenario walk-through in progress*

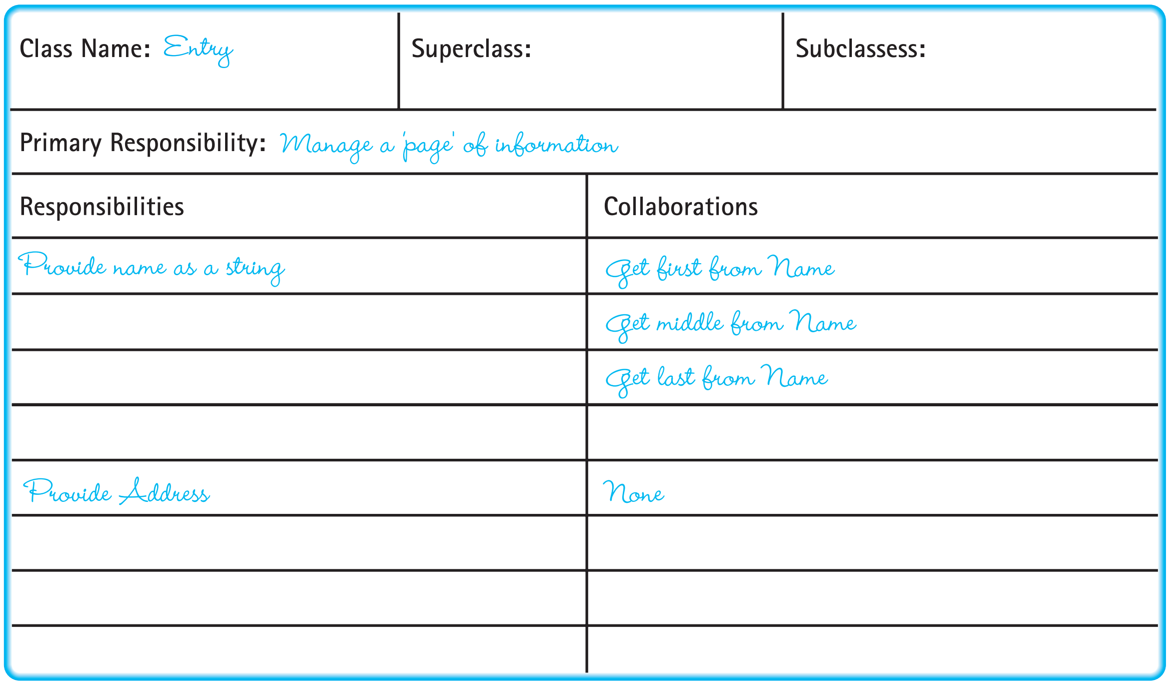
Book Card. Then the owner says, “I have to collaborate with each Entry to compare its name with the name sent to me by the User Interface.” Figure 1.8 shows a team in the middle of a walk-through.

Now comes a decision. What are the responsibilities of Book and Entry for carrying out the comparison? Should Book get the name from Entry and do the comparison, or should it send the name to Entry and receive an answer that indicates whether they are equal? The team decides that Book should do the comparing, so the Entry card is held in the air, and its owner says, “I have a responsibility to provide the full name as a string. To do that I must collaborate with Name.” The responsibility and collaboration are recorded and the Name card is raised.

Name says, “I have the responsibilities to know my first, middle, and last names. These are already on my card, so I’m done.” And the Name card is lowered. Entry says, “I concatenate the three names into a string with spaces between them, and return the result to Book, so I’m done.” The Entry card is lowered.

Book says, “I keep collaborating with Entry until I find the matching name. Then I must collaborate with Entry again to get the address.” This collaboration is placed on its card and the Entry card is held up again, saying “I have a responsibility to provide an address. I’m not going to collaborate with Address, but am just going to return the object to Book.” The Entry card has this responsibility added and then goes back on the table. Its CRC card is shown in Figure 1.9.

The scenario continues until the task of finding an address in the book and reporting it to the user is completed. Reading about the scenario makes it seem longer and more complex than it really is. Once you get used to role playing, the scenarios move quickly and the walk-through becomes more like a game. However, to keep things moving, it is important to avoid becoming bogged-down with implementation details. Book should not be concerned with how the Entry objects are organized on the list. Address doesn’t need to think about whether the zip code is stored as an integer or a String.



Class Name: Entry	Superclass:	Subclassess:

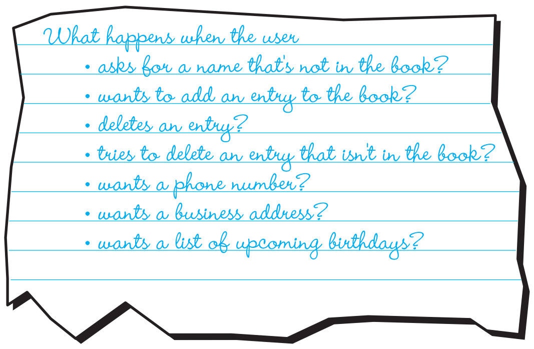
Primary Responsibility: Manage a 'page' of information

Responsibilities	Collaborations
Provide name as a string	Get first from Name
	Get middle from Name
	Get last from Name
Provide Address	None

Figure 1.9 *The CRC card for Entry*

Only explore each responsibility far enough to decide whether a further collaboration is needed, or if it can be solved with the available information.

The next step is to brainstorm some additional questions that produce new scenarios. For example, here is list of some further scenarios.



What happens when the user
- asks for a name that's not in the book?
- wants to add an entry to the book?
- deletes an entry?
- tries to delete an entry that isn't in the book?
- wants a phone number?
- wants a business address?
- wants a list of upcoming birthdays?

We walk through each of the scenarios, adding responsibilities and collaborations to the CRC cards as necessary. After several scenarios have been tried, the number of additions decreases. When one or more scenarios take place without adding to any of the cards, then we brainstorm further to see if we can come up with new scenarios that may not be covered. When all of the scenarios that we can envision seem to be doable with the existing classes, responsibilities, and collaborations, then the design is done.

The next step is to implement the responsibilities for each class. The implementation may reveal details of a collaboration that weren't obvious in the walk-through. But knowing the collaborating classes makes it easy to change their corresponding responsibilities. The implementation phase should also include a search of available class libraries to see if any existing classes can be used. For example, the `java.util.Calendar` class represents a date that can be used directly to implement Birthday.

Enhancing CRC Cards with Additional Information The CRC card design is informal. There are many ways that the card can be enhanced. For example, when a responsibility has obvious steps, we can write them below its name. Each step may have specific collaborations, and we write these beside the steps in the right column. We often recognize that certain data must be sent as part of the message that activates a responsibility, and we can record this in parentheses beside the calling collaboration and the responding responsibility. Figure 1.10 shows a CRC card that includes design information in addition to the basic responsibilities and collaborations.

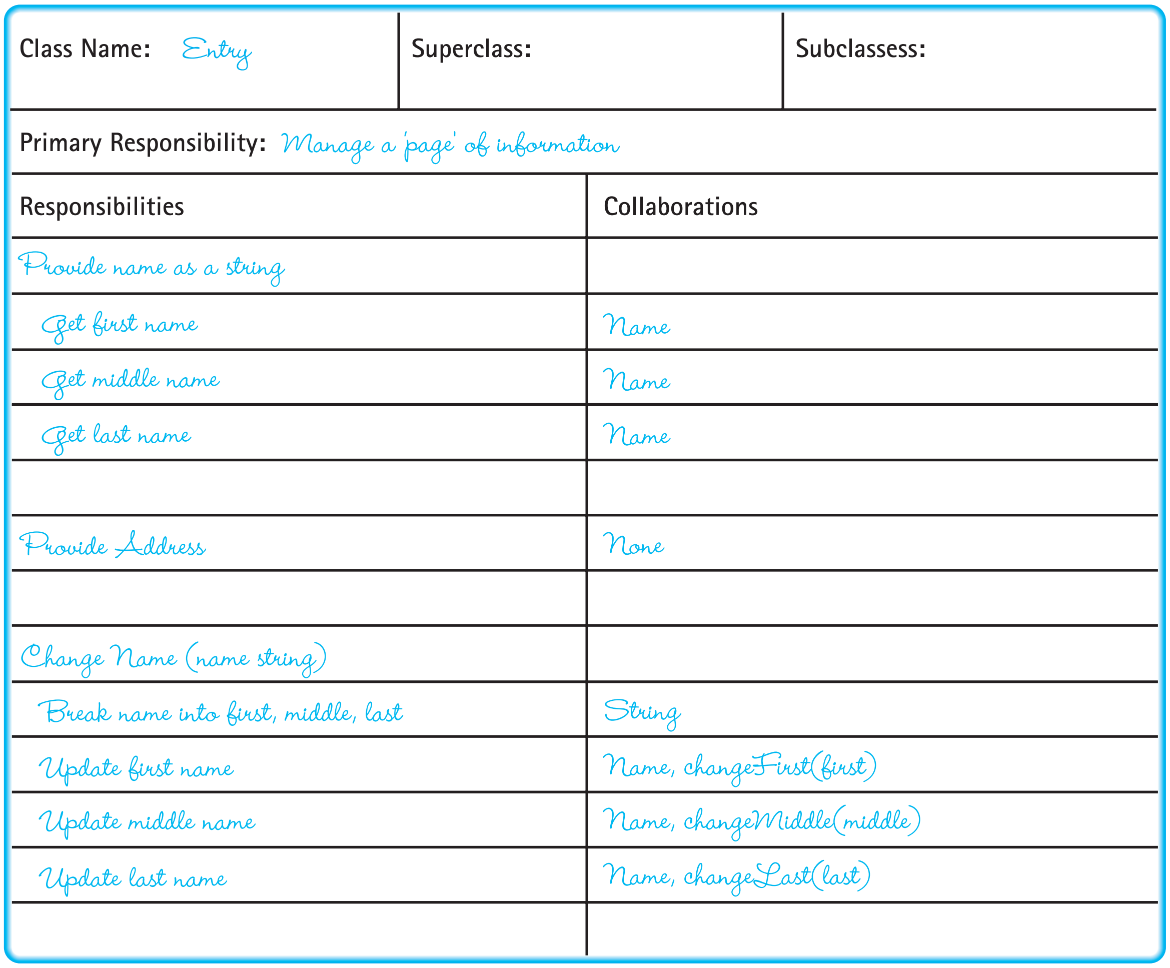
To summarize the CRC Card process, we brainstorm the objects in a problem and abstract them into classes. Then we filter the list of classes to eliminate duplicates. For each class we create a CRC card and list any obvious responsibilities that it should support. We then walk through a common scenario, recording responsibilities and collaborations as they are discovered. After that we walk through additional scenarios, moving from common cases to special and exceptional cases. When it appears that we have all of the scenarios covered, we brainstorm additional scenarios that may need more responsibilities and collaborations. When our ideas for scenarios are exhausted, and all the scenarios are covered by the existing CRC cards, the design is done.

1.3 Verification of Software Correctness

At the beginning of this chapter, we discussed some characteristics of good programs. The first of these was that a good program works—it accomplishes its intended function. How do you know when your program meets that goal? The simple answer is, *test it.*

Testing The process of executing a program with data sets designed to discover errors

Let's look at testing as it relates to the rest of the software development process. As programmers, we first make sure that we understand the requirements, and then we come up with a general solution. Next we design the solution in terms of a system of classes, using good design principles, and finally we implement the solution, using well-structured code, with classes, comments, and so on.



Class Name: Entry	Superclass:	Subclassess:

Primary Responsibility: Manage a 'page' of information

Responsibilities	Collaborations
Provide name as a string	
Get first name	Name
Get middle name	Name
Get last name	Name
Provide Address	None
Change Name (name string)	
Break name into first, middle, last	String
Update first name	Name, changeFirst(first)
Update middle name	Name, changeMiddle(middle)
Update last name	Name, changeLast(last)

Figure 1.10 *A CRC card that is enhanced with additional information*

Once we have the program coded, we compile it repeatedly until the syntax errors are gone. Then we run the program, using carefully selected test data. If the program doesn't work, we say that it has a "bug" in it. We try to pinpoint the error and fix it, a process called debugging.

Debugging The process of removing known errors

Notice the distinction between testing and debugging. Testing is running the program with data sets designed to discover errors; debugging is removing errors once they are discovered.

When the debugging is completed, the software is put into use. Before final delivery, software is sometimes installed on one or more customer sites so that it can be tested in a real environment with real data. After passing this acceptance test phase, the software can be installed at all of the customer sites. Is the verification process now finished? Hardly! More than half of the total life-cycle costs and effort generally occur *after* the program becomes operational, in the maintenance phase. Some changes are made to correct errors in the original program; other changes are introduced to add new capabilities to the software system. In either case, testing must be done after any program modification. This is called regression testing.

Testing is useful for revealing the presence of bugs in a program, but it doesn't prove their absence. We can only say for sure that the program worked correctly for the cases we tested. This approach seems somewhat haphazard. How do we know which tests or how many of them to run? Debugging a whole program at once isn't easy. And fixing the errors found during such testing can sometimes be a messy task. Too bad we couldn't have detected the errors earlier—while we were designing the program, for instance. They would have been much easier to fix then.

Acceptance tests The process of testing the system in its real environment with real data

Regression testing Re-execution of program tests after modifications have been made in order to ensure that the program still works correctly

Program verification The process of determining the degree to which a software product fulfills its specifications

Program validation The process of determining the degree to which software fulfills its intended purpose

We know how program design can be improved by using a good design methodology. Is there something similar that we can do to improve our program verification activities? Yes, there is. Program verification activities don't need to start when the program is completely coded; they can be incorporated into the whole software development process, from the requirements phase on. Program verification is more than just testing.

In addition to program verification—fulfilling the requirement specifications—there is another important task for the software engineer: making sure the specified requirements actually solve the underlying problem. There have been countless times when a programmer finishes a large project and delivers the verified software, only to be told, "Well, that's what I asked for, but it's not what I need."

The process of determining that software accomplishes its intended task is called program validation. Program verification asks, "Are we doing the job right?" Program validation asks, "Are we doing the right job?"[5]

Can we really "debug" a program before it has ever been run—or even before it has been written? In this section, we review a number of topics related to satisfying the criterion "quality software works." The topics include:

- designing for correctness
- performing code and design walk-throughs and inspections
- using debugging methods
- choosing test goals and data
- writing test plans
- structured integration testing

[5]B. W. Boehm, *Software Engineering Economics* (Englewood Cliffs, N.J.: Prentice-Hall, 1981).

Origin of Bugs

When Sherlock Holmes goes off to solve a case, he doesn't start from scratch every time; he knows from experience all kinds of things that help him find solutions. Suppose Holmes finds a victim in a muddy field. He immediately looks for footprints in the mud, for he can tell from a footprint what kind of shoe made it. The first print he finds matches the shoes of the victim, so he keeps looking. Now he finds another, and from his vast knowledge of footprints, he can tell that it was made by a certain type of boot. He deduces that such a boot would be worn by a particular type of laborer, and from the size and depth of the print, he guesses the suspect's height and weight. Now, knowing something about the habits of laborers in this town, he guesses that at 6:30 P.M. the suspect might be found in Clancy's Pub.

In software verification we are often expected to play detective. Given certain clues, we have to find the bugs in programs. If we know what kinds of situations produce program errors, we are more likely to be able to detect and correct problems. We may even be able to step in and prevent many errors entirely, just as Sherlock Holmes sometimes intervenes in time to prevent a crime that is about to take place.

Let's look at some types of software errors that show up at various points in program development and testing and see how they might be avoided.

Specifications and Design Errors

What would happen if, shortly before you were supposed to turn in a major class assignment, you discovered that some details in the professor's program description were incorrect? To make matters worse, you also found out that the corrections were discussed at the beginning of class on the day you got there late, and somehow you never knew about the problem until your tests of the class data set came up with the wrong answers. What do you do now?

Writing a program to the wrong specifications is probably the worst kind of software error. How bad can it be? Most studies indicate that it costs 100 times as much to correct an error discovered after software delivery then it does if it is discovered early in the life cycle. Figure 1.11 shows how fast the costs rise in subsequent phases of software development. The vertical axis represents the relative cost of fixing an error; this cost

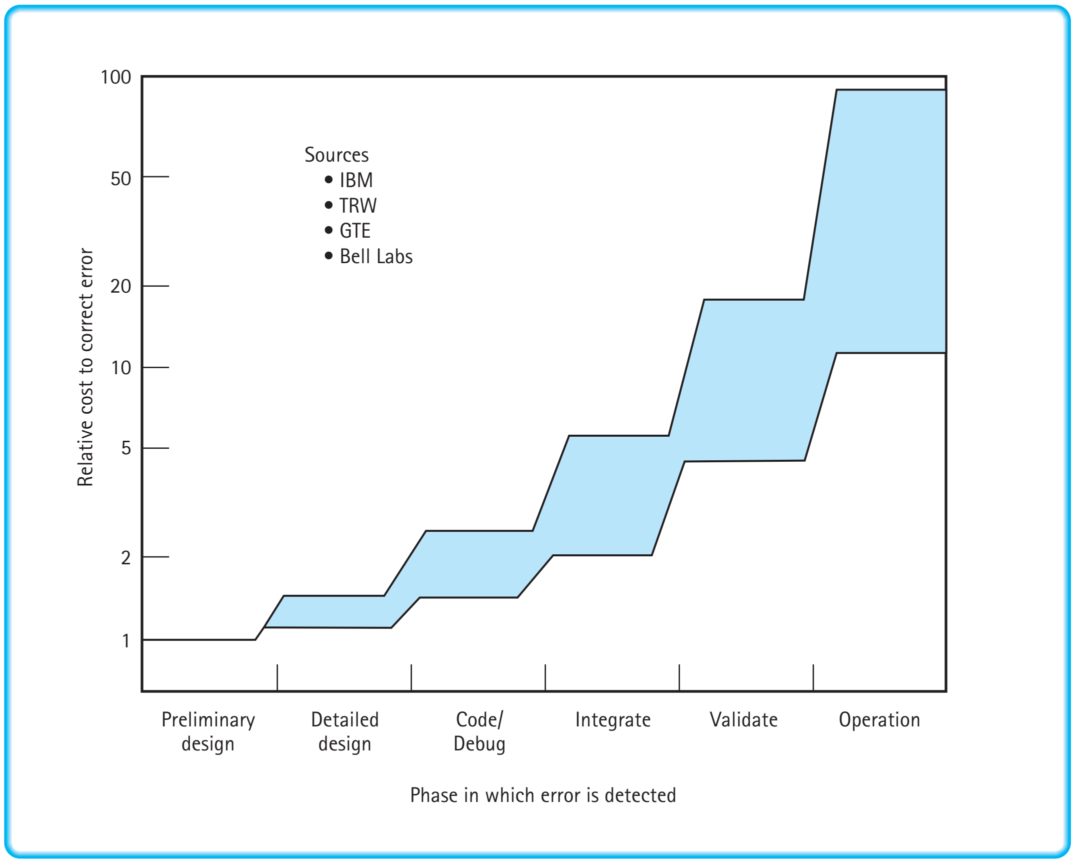


Figure 1.11 *Cost of a specification error based on when it is discovered*

might be in units of hours, or hundreds of dollars, or "programmer months" (the amount of work one programmer can do in a month). The horizontal axis represents the stages in the development of a software product. As you can see, an error that would have taken one unit to fix when you first started designing might take a hundred units to correct when the product is actually in operation!

Many specification errors can be prevented by good communication between the programmers (you) and the party who originated the problem (the professor, manager, or customer). In general, it pays to ask questions when you don't understand something in the program specifications. And the earlier you ask, the better.

A number of questions should come to mind as you first read a programming assignment. What error checking is necessary? What algorithm or data structure is supposed to be used in the solution? What assumptions are reasonable? If you obtain answers to these questions when you first begin working on an assignment, you can incorporate them into your design and implementation of the program. Later in the program's development, unexpected answers to these questions can cost you time and effort. In short, in order to write a program that is correct, you must understand precisely what it is that your program is supposed to do.

Compile-Time Errors

In the process of learning your first programming language, you probably made a number of syntax errors. These resulted in error messages (for example, "TYPE MISMATCH," "ILLEGAL ASSIGNMENT," "SEMICOLON EXPECTED," and so on) when you tried to compile the program. Now that you are more familiar with the programming language, you can save your debugging skills for tracking down important logical errors. *Try to get the syntax right the first time.* Having your program compile cleanly on the first attempt is a reasonable goal. A syntax error wastes computing time and money, as well as programmer time, and it is preventable.

As you progress in your college career or move into a professional computing job, learning a new programming language is often the easiest part of a new software assignment. This does not mean, however, that the language is the least important part. In this book we discuss data structures and algorithms that we believe are language independent. This means that they can be implemented in almost any general-purpose programming language. The success of the implementation, however, depends on a thorough understanding of the features of the programming language. What is considered acceptable programming practice in one language may be inadequate in another, and similar syntactic constructs may be just different enough to cause serious trouble.

It is, therefore, worthwhile to develop an expert knowledge of both the control and data constructs and the syntax of the language in which you are programming. In general, if you have a good knowledge of your programming language—and are careful—you can avoid syntax errors. The ones you might miss are relatively easy to locate and correct. Once you have a "clean" compilation, you can execute your program.

Run-Time Errors

Errors that occur during the execution of a program are usually harder to detect than syntax errors. Some run-time errors stop execution of the program. When this happens, we say that the program "crashed" or "abnormally terminated."

Run-time errors often occur when the programmer makes too many assumptions. For instance,

```
result = dividend / divisor;
```

is a legitimate assignment statement, if we can assume that `divisor` is never zero. If `divisor` is zero, however, a run-time error results.

Run-time errors also occur because of unanticipated user errors. If a user enters the wrong data type in response to a prompt, or supplies an invalid filename to a routine, most simple programs report a runtime error and halt; in other words, they crash.

Robustness The ability of a program to recover following an error; the ability of a program to continue to operate within its environment

Well-written programs should not crash. They should catch such errors and stay in control until the user is ready to quit.

The ability of a program to recover when an error occurs is called robustness. If a commercial program is not robust, people do not buy it. Who wants a word processor that crashes if the user says "SAVE" when there is no disk in the drive? We want the program to tell us, "Put your disk in the drive, and press Enter." For some types of software, robustness is a critical requirement. An airplane's automatic pilot system or an intensive care unit's patient-monitoring program just cannot afford to crash. In such situations, a defensive posture produces good results.

In general, you should actively check for error-creating conditions rather than let them abort your program. For instance, it is generally unwise to make too many assumptions about the correctness of input, especially interactive input from a keyboard. A better approach is to check explicitly for the correct type and bounds of such input. The programmer can then decide how an error should be handled (request new input, print a message, or go on to the next data) rather than leave the decision to the system. Even the decision to quit should be made by a program that is in control of its own execution. If worse comes to worst, let your program die gracefully.

This does not mean that everything that the program inputs must be checked for errors. Sometimes inputs are known to be correct—for instance, input from a file that has been verified. The decision to include error checking must be based upon the requirements of the program.

Some run-time errors do not stop execution but produce the wrong results. You may have incorrectly implemented an algorithm or initialized a variable to an incorrect value. You may have inadvertently swapped two parameters of the same type on a method call or used a less-than sign instead of a greater-than sign. These logical errors are often the hardest to prevent and locate. Later we talk about debugging techniques to help pinpoint run-time errors. We also discuss structured testing methods that isolate the part of the program being tested. But knowing that the earlier we find an error the easier it is to fix, we turn now to ways of catching run-time errors before run time.

Designing for Correctness

It would be nice if there were some tool that would locate the errors in our design or code without our even having to run the program. That sounds unlikely, but consider an analogy from geometry. We wouldn't try to prove the Pythagorean theorem by proving that it worked on every triangle; that would only demonstrate that the theorem works for every triangle we tried. We prove theorems in geometry mathematically. Why can't we do the same for computer programs?

The verification of program correctness, independent of data testing, is an important area of theoretical computer science research. The goal of this research is to establish a method for proving programs that is analogous to the method for proving theorems in geometry. The necessary techniques exist, but the proofs are often more complicated than the programs themselves. Therefore, a major focus of verification research is to attempt to build automated program provers—verifiable programs that verify other programs. In the meantime, the formal verification techniques can be carried out by hand.[6]

Preconditions and Postconditions

Suppose we want to design a module (a logical chunk of the program) to perform a specific operation. To ensure that this module fits into the program as a whole, we must clarify what happens at its boundaries—what must be true when we enter the module and what is true when we exit.

To make the task more concrete, picture the design module as it is usually coded, as a method that is exported from a class. To be able to invoke the method, we must know its exact interface: the name and the parameter list, which indicates its inputs and outputs. But this isn't enough: We must also know any assumptions that must be true for the operation to function correctly.

We call the assumptions that must be true when invoking the method preconditions. The preconditions are like a product disclaimer:

Preconditions Assumptions that must be true on entry into an operation or method for the postconditions to be guaranteed

WARNING
If you try to execute this operation when the preconditions are not true, the results are not guaranteed.

For example, the `increment` method of the `IncDate` class, described in the previous section, might have preconditions related to legal date values and the start of the Gregorian calendar. The preconditions should be listed with the method declaration:

```
public void increment()
// Preconditions: Values of day, month, and year represent a valid date
//                The represented date is not before minYear
```

Previously we discussed the quality of program robustness, the ability of a program to catch and recover from errors. While creating robust programs is an important goal,

[6]We do not go into this subject in detail here. If you are interested in this topic, you might start with David Gries' classic, *The Science of Programming* (NewYork: Springer-Verlag, (1981)).

it is sometimes necessary to decide at what level errors are caught and handled. Using preconditions for a method is similar to a contract between the programmer who creates the method and the programmers who use the method. The contract says that the programmer who creates the method is not going to try to catch the error conditions described by the preconditions, but as long as the preconditions are met, the method works correctly. It is up to the programmers who use the method to ensure that the method is never called without meeting the preconditions. In other words, the robustness of the system in terms of the method's preconditions is the responsibility of the programmers who use the class, and not the programmer who creates the class. This approach is sometimes called "programming by contract." It can save work because trapping the same error conditions at multiple levels of a hierarchical system is redundant and unnecessary.

Postconditions Statements that describe what results are to be expected at the exit of an operation or method, assuming that the preconditions are true

We must also know what conditions are true when the operation is complete. The postconditions are statements that describe the results of the operation. The postconditions do not tell us how these results are accomplished; they merely tell us what the results should be.

Let's consider what the preconditions and postconditions might be for another simple operation: a method that deletes the last element from a list. (We are using "list" in an intuitive sense; we formally define it in Chapter 3.) Assuming the method is defined within a class with the responsibility of maintaining a list, the specification for `RemoveLast` is as follows:

void RemoveLast()

Effect:	Removes the last element in this list.
Precondition:	This list is not empty.
Postcondition:	The last element has been removed from this list.

What do these preconditions and postconditions have to do with program verification? By making explicit statements about what is expected at the interfaces between modules, we can avoid making logical errors based on misunderstandings. For instance, from the precondition we know that we must check outside of this operation for the empty condition; this module *assumes* that there is at least one element.

Experienced software developers know that misunderstandings about interfaces to someone else's modules are one of the main sources of program problems. We use preconditions and postconditions at the method level in this book, because the information they provide helps us to design programs in a truly modular fashion. We can then use the classes we've designed in our programs, confident that we are not introducing errors by making mistakes about assumptions and about what the classes actually do.

Design Review Activities

When an individual programmer is designing and implementing a program, he or she can find many software errors with pencil and paper. Deskchecking the design solution is a very common method of manually verifying a program. The programmer writes down essential data (variables, input values, parameters, and so on) and walks through the design, marking changes in the data on the paper. Known trouble spots in the design or code should be double-checked. A checklist of typical errors (such as loops that do not terminate, variables that are used before they are initialized, and incorrect order of parameters on method calls) can be used to make the deskcheck more effective. A sample checklist for deskchecking a Java program appears in Figure 1.12. A few minutes spent deskchecking your designs can save lots of

Deskchecking Tracing an execution of a design or program on paper

The Design

1. Does each class in the design have a clear function or purpose?
2. Can large classes be broken down into smaller pieces?
3. Do multiple classes share common code? Is it possible to write more general classes to encapsulate the commonalities and then have the individual classes inherit from that general class?
4. Are all the assumptions valid? Are they well documented?
5. Are the preconditions and postconditions accurate assertions about what should be happening in the method they specify?
6. Is the design correct and complete as measured against the program specification? Are there any missing cases? Is there faulty logic?
7. Is the program designed well for understandability and maintainability?

The Code

1. Has the design been clearly and correctly implemented in the programming language? Are features of the programming language used appropriately?
2. Are methods coded to be consistent with the interfaces shown in the design?
3. Are the actual parameters on method calls consistent with the parameters declared in the method definition?
4. Is each data object to be initialized set correctly at the proper time? Is each data object set correctly before its value is used?
5. Do all loops terminate?
6. Is the design free of "magic" values? (A magic value is one whose meaning is not immediately evident to the reader. You should use constants in place of such values.)
7. Does each constant, class, variable, and method have a meaningful name? Are comments included with the declarations to clarify the use of the data objects?

Figure 1.12 *Checklist for deskchecking programs*

time and eliminate difficult problems that would otherwise surface later in the life cycle (or even worse, would not surface until after delivery).

Have you ever been really stuck trying to debug a program and showed it to a classmate or colleague who detected the bug right away? It is generally acknowledged that someone else can detect errors in a program better than the original author can. In an extension of deskchecking, two programmers can trade code listings and check each other's programs. Universities, however, frequently discourage students from examining each other's programs for fear that this exchange leads to cheating. Thus, many students become experienced in writing programs but don't have much opportunity to practice reading them.

Most sizable computer programs are developed by *teams* of programmers. Two extensions of deskchecking that are effectively used by programming teams are design or code walk-throughs and inspections. These are formal team activities, the intention of which is to move the responsibility for uncovering bugs from the individual programmer to the group. Because testing is time-consuming and errors cost more the later they are discovered, the goal is to identify errors before testing begins.

Walk-through A verification method in which a *team* performs a manual simulation of the program or design

Inspection A verification method in which one member of a team reads the program or design line by line and the others point out errors

In a *walk-through,* the team performs a manual simulation of the design or program with sample test inputs, keeping track of the program's data by hand on paper or a blackboard. Unlike thorough program testing, the walk-through is not intended to simulate all possible test cases. Instead, its purpose is to stimulate discussion about the way the programmer chose to design or implement the program's requirements.

At an *inspection*, a reader (never the program's author) goes through the requirements, design, or code line by line. The inspection participants are given the material in advance and are expected to have reviewed it carefully. During the inspection, the participants point out errors, which are recorded on an inspection report. Many of the errors have been noted by team members during their preinspection preparation. Other errors are uncovered just by the process of reading aloud. As with the walk-through, the chief benefit of the team meeting is the discussion that takes place among team members. This interaction among programmers, testers, and other team members can uncover many program errors long before the testing stage begins.

If you look back at Figure 1.11, you see that the cost of fixing an error is relatively inexpensive up through the coding phase. After that, the cost of fixing an error increases dramatically. Using the formal inspection process can clearly benefit a project.

Exceptions

At the design stage, you should plan how to handle exceptions in your program. Exceptions are just what the name implies: exceptional situations. They are situations that alter the flow of control of the program, usually resulting in a premature end to program execution. Working with exceptions begins at the design phase: What are the unusual situations that the program should recognize? Where in the program can the situations be detected? How should the situations be handled if they occur?

Exception Associated with an unusual, often unpredictable event, detectable by software or hardware, that requires special processing. The event may or may not be erroneous.

Where—indeed whether—an exception is detected depends on the language, the software package design, the design of the libraries being used, and the platform, that is, on the operating system and hardware. Where an exception *should* be detected depends on the type of exception, on the software package design, and on the platform. Where an exception *is* detected should be well documented in the relevant code segments.

An exception *may* be handled any place in the software hierarchy—from the place in the program module where the exception is first detected through the top level of the program. In Java, as in most programming languages, unhandled built-in exceptions carry the penalty of program termination. Where in an application an exception *should* be handled is a design decision; however, exceptions should be handled at a level that knows what the exception means.

An exception need not be fatal. For non-fatal exceptions, the thread of execution may continue. Although the thread of execution can continue from any point in the program, the execution should continue from the lowest level that can recover from the exception. When an error occurs, the program may fail unexpectedly. Some of the failure conditions may possibly be anticipated and some may not. All such errors must be detected and managed.

Exceptions can be written in any language. Java (along with some other languages) provides built-in mechanisms to manage exceptions. All exception mechanisms have three parts:

- Defining the exception
- Generating (raising) the exception
- Handling the exception

Once your exception plan is determined, Java gives you a clean way of implementing these three phases using the *try-catch* and *throw* statements. We cover these statements at the end of Chapter 2 after we have introduced some additional Java constructs.

Program Testing

Eventually, after all the design verification, deskchecking, and inspections have been completed, it is time to execute the code. At last, we are ready to start testing with the intention of finding any errors that may still remain.

The testing process is made up of a set of test cases that, taken together, allow us to assert that a program works correctly. We say "assert" rather than "prove" because testing does not generally provide a proof of program correctness.

The goal of each test case is to verify a particular program feature. For instance, we may design several test cases to demonstrate that the program correctly handles various classes of input errors. Or we may design cases to check the processing when a data structure (such as an array) is empty, or when it contains the maximum number of elements.

Within each test case, we must perform a series of component tasks:

- We determine inputs that demonstrate the goal of the test case.
- We determine the expected behavior of the program for the given input.
- We run the program and observe the resulting behavior.
- We compare the expected behavior and the actual behavior of the program. If they are the same, the test case is successful. If not, an error exists, either in the test case itself or in the program. In the latter case, we begin debugging.

Unit testing Testing a class or method by itself

For now we are talking about test cases at a class, or method, level. It's much easier to test and debug modules of a program one at a time, rather than trying to get the whole program solution to work all at once. Testing at this level is called unit testing.

How do we know what kinds of unit test cases are appropriate, and how many are needed? Determining the set of test cases that is sufficient to validate a unit of a program is in itself a difficult task. There are two approaches to specifying test cases: cases based on testing possible data inputs and cases based on testing aspects of the code itself.

Functional domain The set of valid input data for a program or method

Data Coverage

In those limited cases where the set of valid inputs, or the functional domain, is extremely small, one can verify a program unit by testing it against every possible input element. This approach, known as *exhaustive testing*, can prove conclusively that the software meets its specifications. For instance, the functional domain of the following method consists of the values `true` and `false`.

```
public void PrintBoolean(boolean boolValue)
// Prints the Boolean value to the output
{
  if (boolValue)
    output.println("true");
  else
    output.println("false");
}
```

It makes sense to apply exhaustive testing to this method, because there are only two possible input values. In most cases, however, the functional domain is very large, so exhaustive testing is almost always impractical or impossible. What is the functional domain of the following method?

```
public void PrintInteger(int intValue)
// Prints the integer value intValue to the output
{
  output.println(intValue);
}
```

It is not practical to test this method by running it with every possible data input; the number of elements in the set of `int` values is clearly too large. In such cases, we do not attempt exhaustive testing. Instead, we pick some other measurement as a testing goal.

You can attempt program testing in a haphazard way, entering data randomly until you cause the program to fail. Guessing doesn't hurt, but it may not help much either. This approach is likely to uncover some bugs in a program, but it is very unlikely to find them all. Fortunately, however, there are strategies for detecting errors in a systematic way.

One goal-oriented approach is to cover general classes of data. You should test at least one example of each category of inputs, as well as boundaries and other special cases. For instance, in method `PrintInteger` there are three basic classes of `int` data: negative values, zero, and positive values. So, you should plan three test cases, one for each of these classes. You could try more than three, of course. For example, you might want to try `Integer.MAX_VALUE` and `Integer.MIN_VALUE`, but because all the program does is print the value of its input, the additional test cases don't accomplish much.

There are other cases of data coverage. For example, if the input consists of commands, you must test each command and varying sequences of commands. If the input is a fixed-sized array containing a variable number of values, you should test the maximum number of values; this is the boundary condition. A way to test for robustness is to try one more than the maximum number of values. It is also a good idea to try an array in which no values have been stored or one that contains a single element. Testing based on data coverage is called black-box testing. The tester must know the external interface to the module—its inputs and expected outputs—but does not need to consider what is being done inside the module (the inside of the black box). (See Figure 1.13)

Black-box testing Testing a program or method based on the possible input values, treating the code as a "black box"

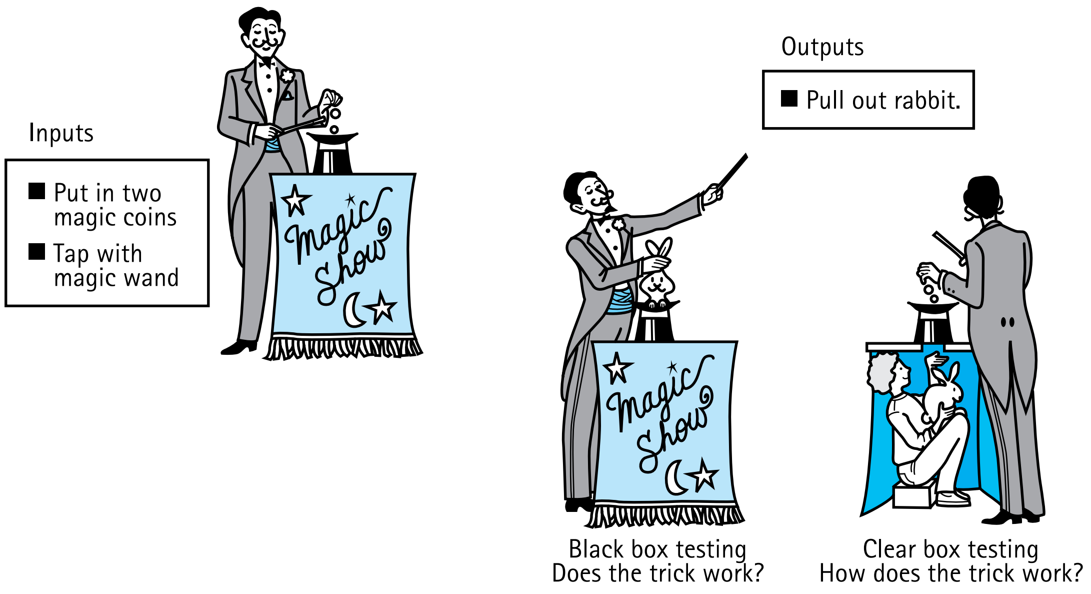


Figure 1.13 *Testing approaches*

Code Coverage

A number of testing strategies are based on the concept of code coverage, the execution of statements or groups of statements in the program. This testing approach is called clear (or white) box testing. The tester must look inside the module (through the clear box) to see the code that is being tested.

Clear (white) box testing Testing a program or method based on covering all of the branches or paths of the code

Branch A code segment that is not always executed; for example, a Switch statement has as many branches as there are case labels

Path A combination of branches that might be traversed when a program or method is executed

Path testing A testing technique whereby the tester tries to execute all possible paths in a program or method

One approach, called *statement coverage*, requires that every statement in the program be executed at least once. Another approach requires that the test cases cause every branch, or code section, in the program to be executed. A single test case can achieve statement coverage of an *if-then* statement, but it takes two test cases to test both branches of the statement.

A similar type of code-coverage goal is to test program paths. A path is a combination of branches that might be traveled when the program is executed. In path testing, we try to execute all the possible program paths in different test cases.

Test Plans

Deciding on the goal of the test approach—data coverage, code coverage, or (most often) a mixture of the two, precedes the development of a test plan. Some test plans are very informal—the goal and a list of test cases, written by hand on a piece of paper. Even this type of test plan may be more than you have ever been required to write for a class programming project. Other test plans (particularly those submitted to management or to a customer for approval) are very formal, containing the details of each test case in a standardized format.

Test plan A document showing the test cases planned for a program or module, their purposes, inputs, expected outputs, and criteria for success

For program testing to be effective, *it must be planned*. You must design your testing in an organized way, and you must put your design in writing. You should determine the required or desired level of testing, and plan your general strategy and test cases before testing begins. In fact, you should start planning for testing before writing a single line of code.

Debugging

In the previous section we talked about checking the output from our test and debugging when errors were detected. We can debug "on the fly" by adding output statements in suspected trouble spots when problems are found. For example, if you suspect an error in the `IncDate increment` method, you could augment the method as follows:

```
public void increment()
{
  // For debugging
  output.println("IncDate method increment entered.");
  output.println("year = " + year);
  output.println("month = " + month);
  output.println("day = " + day);

  // Increment algorithm goes here
  // It updates the year, month, and day values

  // For debugging
  output.println("IncDate method increment exiting.");
  output.println("year = " + year);
  output.println("month = " + month);
  output.println("day = " + day);
  output.println("IncDate method increment terminated.");
}
```

Note that the new output is only for debugging; these output lines are meant to be seen only by the tester, not by the user of the program. But it's annoying for debugging output to show up mixed with your application's real output, and it's difficult to debug when the debugging output isn't collected in one place. One way to separate the debugging output from the "real" program output is to declare a separate file to receive these debugging lines.

Usually the debugging output statements are removed from the program, or "commented out," before the program is delivered to the customer or turned in to the professor. (To "comment out" means to turn the statements into comments by preceding them with `//` or enclosing them between `/*` and `*/`.) An advantage of turning the debugging statements into comments is that you can easily and selectively turn them back on for later tests. A disadvantage of this technique is that editing is required throughout the program to change from the testing mode (with debugging) to the operational mode (without debugging).

Another popular technique is to make the debugging output statements dependent on a Boolean flag, which can be turned on or off as desired. For instance, a section of code known to be error-prone may be flagged in various spots for trace output by using the Boolean value `debugFlag`:

```
// Set debugFlag to control debugging mode
static boolean debugFlag = true;
.
.
.
if (debugFlag)
  debugOutput.println("method Complex entered.");
```

This flag may be turned on or off by assignment, depending on the programmer's need. Changing to an operational mode (without debugging output) merely involves redefining `debugFlag` as `false` and then recompiling the program. If a flag is used, the debugging statements can be left in the program; only the *if* checks are executed in an operational run of the program. The disadvantage of this technique is that the code for the debugging is always there, making the compiled program larger and slower. If there are a lot of debugging statements, they may waste needed space and time in a large program. The debugging statements can also clutter up the program, making it harder to read. (This is another example of the tradeoffs we face in developing software.)

Some systems have online debugging programs that provide trace outputs, making the debugging process much simpler. If the system at your school or workplace has a run-time debugger, use it! Any tool that makes the task easier should be welcome, but remember that no tool replaces thinking.

A warning about debugging: Beware of the quick fix! Program bugs often travel in swarms, so when you find a bug, don't be too quick to fix it and run your program again. As often as not, fixing one bug generates another. A superficial guess about the cause of a program error usually does not produce a complete solution. In general, the time that it takes to consider all the ramifications of the changes you are making is time well spent.

If you constantly need to debug, there's a deficiency in your design process. The time that it takes to consider all the ramifications of the design you are making is time spent best of all.

Testing Java Data Structures

The major topic of this textbook is data structures: what they are, how we use them, and how we implement them using Java. This chapter has been an overview of software engineering. In Chapter 2 we begin our concentration on data and how to structure it. It seems appropriate to end this section about verification with a look at how we test the data structures we implement in Java.

In Chapter 2, we implement a data structure using a Java class, so that many different application programs can use the structure. When we first create the class that models the data structure, we do not necessarily have any application programs ready to use it. We need to test it by itself first, before creating the applications.

Every data structure that we implement supports a set of operations. For each structure, we would like to create a test driver program that allows us to test the operations in a variety of sequences. How can we write a single test driver that allows us to test numerous operation sequences? The solution is to separate the specific set of operations that we want to test from the test driver program itself. We list the operations, and the necessary parameters, in a text file. The test driver program reads the operations from the text file one line at a time, performs the listed operation by invoking the methods of the class being tested, and reports the results to an output file. The test program also reports its general results on the screen.

The testing approach described here allows us to easily change our test case—we just have to change the contents of the input file. However, it would be even easier if we could dynamically change the name of the input file, whenever we run the program. Then we could organize our test cases, one per file, and easily rerun a test case whenever we needed. Therefore, we construct our test driver to accept the name of the input file as a command line parameter; we do the same for the output file. Figure 1.14 displays a model of our test architecture.

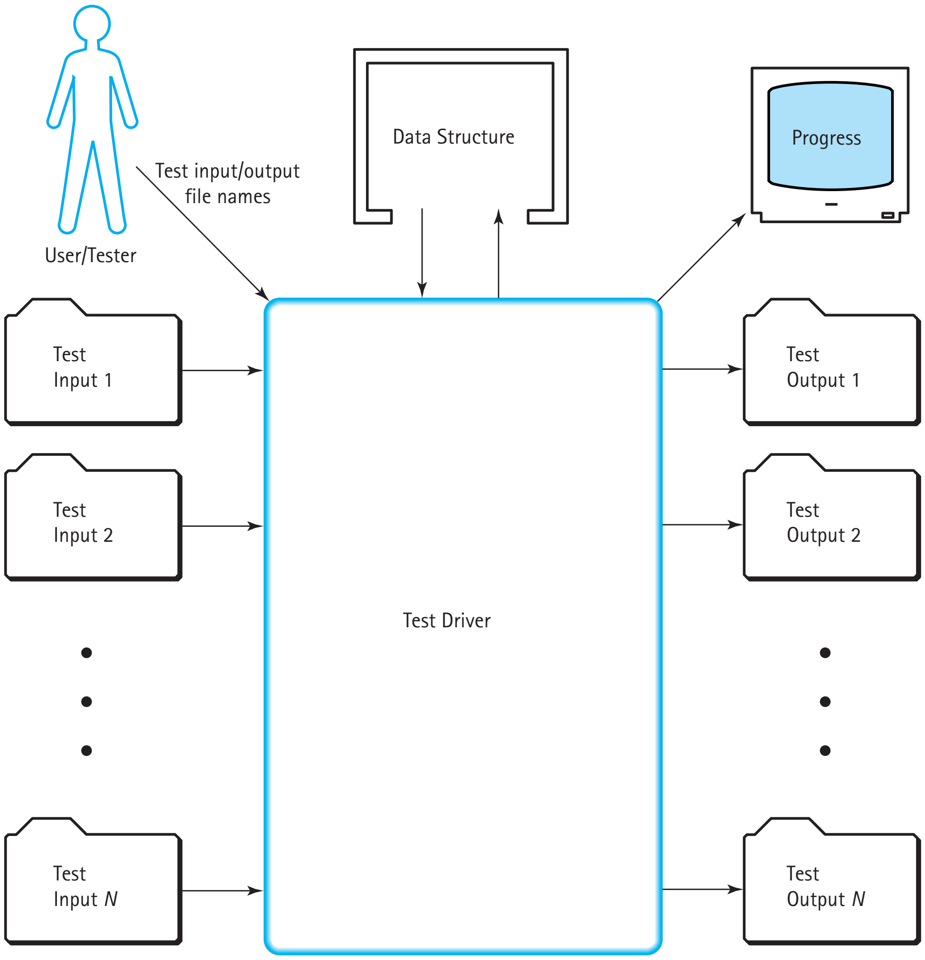


Figure 1.14 *Model of test architecture*

Our test drivers all follow the same basic algorithm; here is a pseudocode description:

```
Obtain the names of the input and output files from the command line
Open the input file for reading and the output file for writing
Read the first line from the input file
Print "Results " plus the first line of the input file to the output file
Print a blank line to the output file
Read a command line from the input file
Set numCommands to 0
While the command read is not 'quit'
    Execute the command by invoking the public methods of the data structure
    Print the results to the output file
    Print the data structure to the output file (if appropriate)
    Increment numCommands by 1
    Print "Command " + numCommands + " completed" to the screen
    Read the next command from the input file
Close the input and output files.
Print "Testing completed" to the screen
```

This algorithm provides us with maximum flexibility for minimum extra work when we are testing our data structures. Once we implement the algorithm by creating a test driver for a specific data structure, we can easily create a test driver for a different data structure by changing only three steps.

Notice that the third and fourth commands copy a "header line" from the input test file to the output file. This helps us manage our test cases by allowing us to label each test case file with an identifying string on its first line; the same string always begins the corresponding output file.

Suppose we want to test the `IncDate` class that was defined earlier in this chapter. We first create a test plan. Let's use a goal-oriented approach. We first test the constructor and each of the observer methods. Next we test the transformer method `increment`. To test `increment` we identify general categories of dates, with respect to the effect of the `increment` method. We test dates that represent each of these categories, with special attention given to the boundaries of the categories. Thus, we test some dates in the middle of months, and at the beginning and end of months. We test the end of years also. We pay careful attention to testing how the method handles leap years, by including tests concentrated at the end of February in many different years. Several more test cases, besides those listed below, would be needed to ensure that the `increment` method works correctly.

Operation to be Tested and Description of Action	Input Values	Expected Output
`IncDate`		
invoke and print	5, 6, 2000	5/6/2000
Observers		
print `monthIs`		5
print `dayIs`		6
print `yearIs`		2000
Transformer		
`increment` and print		5/7/2000
`IncDate`	5,30,2000	
`increment` and print		5/31/2000
`IncDate`	5,31,2000	
`increment` and print		6/1/2000
`IncDate`	6,30,2000	
`increment` and print		7/1/2000
`IncDate`	2,28,2002	
`increment` and print		3/1/2002
etc.		

After identifying a test plan, we create a test driver using our algorithm. Then we use the test driver to carry out our plan. The `IncDate` class supports five operations: `IncDate` (the constructor), `yearIs`, `monthIs`, `dayIs`, and `increment`. We represent these operations in the test input file simply by using their names. In that file, the word `IncDate` is followed by three lines, each containing an integer, to supply the three *int* parameters of the constructor. Figure 1.15 shows an example of a test input file, the resulting output file, and the screen information that would be generated.

Study the test driver program on page 51 to make sure you understand our testing approach. You should be able to follow the control logic of the program. Note that we assume the inclusion of a reasonable `toString` method in the `Date` class, as described at the end of the Object-Oriented Design section. (The `Date.java` file on our web site includes a `toString` method.)

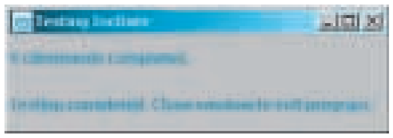
```
IncDate Test Data A
IncDate
5
6
2000
monthIs
dayIs
increment
dayIs
quit
```

File: TestDataA

```
Results IncDate Test Data A

Constructor invoked with 5 6 2000
theDate: 5/6/2000
Month is 5
theDate: 5/6/2000
Day is 6
theDate: 5/6/2000
increment invoked
theDate: 5/7/2000
Day is 7
theDate: 5/7/2000
```

File: TestOutputA



```
5 commands completed.

Testing completed. Close window to exit program.
```

Screen

```
Command: java TDIncDate TestDataA TestOutputA
```

Figure 1.15 *Example of a test input file and resulting output file*

We realize that the students using this textbook come from a wide variety of Java backgrounds, especially with respect to the Java I/O approach. You may have learned Java in an environment where the Java input/output statements were "hidden" behind a package provided with your introductory textbook. Or you may have learned graphical input/output techniques, but never learned how to do file input/output. You may not be familiar with "command-line parameters;" or you might have been using command-line parameters since the first week you studied Java. You may have learned how to use the Java AWT; you may have learned Swing; you may have learned neither. Our approach to testing requires only simple file input and output, in addition to screen output. It does not require any direct user input during execution, which can be complicated in Java.

The feature section on Java Input/Output (after the following code) introduces the input/output techniques used for our test drivers. We use these same techniques in test drivers and example programs throughout the rest of the text, so it is a good idea for you to study them carefully now. The only places in the text where more advanced I/O approaches are used are in the chapter Case Studies. Beginning with Chapter 3, we develop case studies as examples of real programs that use the data structures you are studying. These case studies use progressively more advanced graphical interfaces, and are accompanied by additional feature sections as needed to explain any new constructs.

Therefore, the case studies not only provide examples of object-oriented design and uses of data structures, but they also progressively introduce you to user interface techniques.

Within the following test driver code we have emphasized, with underlining, all the commands related to input/output. As you can see, these statements make up a large percentage of the program; this is not unusual.

```
//----------------------------------------------------------------------------
// TDIncDate.java             by Dale/Joyce/Weems                     Chapter 1
//
// Test Driver for the IncDate class
//----------------------------------------------------------------------------

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import IncDate.*;

// Test Driver for the IncDate class
public class TDIncDate
{
  public static void main(String[] args) throws IOException
  {
    String testName   = "IncDate";
    String command    = null;
    int numCommands   = 0;
    IncDate  theDate = new IncDate(0,0,0);
    int month, day, year;

    //Get file name arguments from command line as entered by user
    String dataFileName = args[0];
    String outFileName  = args[1];

    //Prepare files
    BufferedReader dataFile = new BufferedReader(new FileReader(dataFileName));
    PrintWriter outFile     = new PrintWriter(new FileWriter(outFileName));

    //Get test file header line and echo print to outFile
    String testInfo = dataFile.readLine();
    outFile.println("Results " + testInfo);
    outFile.println();
    command = dataFile.readLine();

    //Process commands
    while(!command.equals("quit"))
```

```
{
  if (command.equals("IncDate"))
  {
    month  = Integer.parseInt(dataFile.readLine());
    day    = Integer.parseInt(dataFile.readLine());
    year   = Integer.parseInt(dataFile.readLine());
    outFile.println("Constructor invoked with " + month + " "
                     + day + " " + year);
    theDate = new IncDate(month, day, year);
  }
  else if (command.equals("yearIs"))

  {
    outFile.println("Year is " + theDate.yearIs());
  }
  else if (command.equals("monthIs"))

  {
    outFile.println("Month is " + theDate.monthIs());
  }
  else if (command.equals("dayIs"))

  {
    outFile.println("Day is " + theDate.dayIs());
  }
  else if (command.equals("increment"))

  {
    theDate.increment();
    outFile.println("increment invoked ");
  }

  outFile.println("theDate: " + theDate);
  numCommands++;
  command = dataFile.readLine();
}

//Close files
dataFile.close();
outFile.close();

//Set up output frame
JFrame outputFrame = new JFrame();
outputFrame.setTitle("Testing " + testName);
outputFrame.setSize(300,100);
outputFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
      // Instantiate content pane and information panel
      Container contentPane = outputFrame.getContentPane();
      JPanel infoPanel      = new JPanel();

      // Set layout
      infoPanel.setLayout(new GridLayout(2,1));

      // Create labels
      JLabel countInfo = new JLabel(numCommands + " commands completed.  ");
      JLabel finishedInfo = new JLabel("Testing completed. "
                                  + "Close window to exit program.");

      // Add information
      infoPanel.add(countInfo);
      infoPanel.add(finishedInfo);
      contentPane.add(infoPanel);

      // Show information
      outputFrame.show();
    }
  }
```

Note that the test driver gets the test data and calls the methods to be tested. It also provides written output about the effects of the method calls, so that the tester can check the results. Sometimes test drivers are used to test hundreds or thousands of test cases. In such situations it is best if the test driver automatically verifies whether or not the test cases were handled successfully. Exercise 36 asks you to expand this test driver to include automatic test-case verification.

This test driver does not do any error checking to make sure that the inputs are valid. For instance, it doesn't verify that the input command code is really a legal command. Furthermore, it does not handle possible I/O exceptions; instead it just throws them out to the run-time environment (exception handling is discussed in Chapter 2). Remember that the goal of the test driver is to act as a skeleton of the real program, not to be the real program. Therefore, the test driver does not need to be as robust as the program it simulates.

Java Input/Output I

The Java class libraries provide varied and robust mechanisms for input and output. Hundreds of classes related to the user interface provide programmers with a multitude of options. I/O is not the topic of this textbook. We use straightforward I/O approaches that support the study of data structures.

In this feature section, we examine the I/O commands used in the `TDIncDate` program (we examine more I/O commands as needed later in the text). The relevant commands are highlighted in the program text. As modeled in Figure 1.14, this program uses screen output and file input and output. The program also uses command-line arguments to obtain the names of the files—this is a form of input. Figure 1.15 shows an example of an input file, the resultant output file, the screen output, and the corresponding command line. If you're interested in learning more, you might begin by studying the documentation provided on the Sun Microsystems Inc. web site of the various classes and methods we use.

Command-Line Input

A simple way to pass string information to a Java program is with command-line arguments. Command-line arguments are read by the program each time it is run; a different set of arguments will invoke different behavior from the program. For example, suppose you want to run the `TDIncDate` program using a file called `TestDataA` as the input file and a file called `TestOutputA` as the output file. If you are working from the command line, you invoke the Java interpreter, asking it to "execute" the `TDIncDate.class` file using as arguments the strings "TestDataA" and "TestOutputA" by entering:

```
java TDIncDate TestDataA TestOutputA
```

The program runs; it takes its input from the `TestDataA` file; a small output window appears on your screen informing you when the program is finished; and the `TestOutputA` file holds the results of the test. You end the program by closing the output window. Now, if you want the program to run again using different input and output files, say, `TestDataB` and `TestOutputB`, you simply invoke the interpreter with a different command line:

```
java TDIncDate TestDataB TestOutputB
```

Note that if you are using an integrated development environment, instead of working from the command line, you compile and run your program using a pull-down menu or a shortcut key. Consult your environment's documentation to learn how to pass command-line arguments in this situation.

How do you access the command-line arguments within your program? Through the `main` method's array of strings parameter. By convention, this parameter is usually called `args`, to represent the command-line arguments. In our example, `args[0]` references the string "TestDataA" and `args[1]` references the string "TestOutputA". We use these string values to initialize string variables that represent the input and output files of the program:

```
String dataFileName = args[0];
String outFileName = args[1];
```

With this approach, we can change the test input and output files each time we run the program by simply entering a different command on the command line.

File Output

Java provides a stream output model. As an abstract concept, a stream is just a sequence of bytes. A Java program can direct an output stream to a file, a network connection, or even a specific block of memory. We use files.

The Java class library supports more than 60 different stream types. We use classes that inherit from the abstract class `Writer`. Abstract classes are discussed in Chapter 3. For now, all you need to know is that you cannot instantiate objects of abstract classes, but you can extend the classes. In our program we use the `PrintWriter` class and the `FileWriter` class, both of which are library subclasses of `Writer`. To make these classes available within our program, we must include the *import* statement:

```
import java.io.*;
```

The `Writer` class and its subclasses allow us to perform text output in a standard environment. You may recall from your previous studies that Java uses the Unicode character set as its base character set. A Unicode character uses 16 bits; therefore, the Unicode character set can represent 65,536 unique characters. This large character set helps make Java suitable as a programming language around the world, since there are many languages that do not use the standard Western alphabet. However, most of our environments do not yet support the Unicode character set. For example, text files, which we often use to provide input to a program or output from a program, are based on the much smaller ASCII character set. The `Writer` class provides methods to translate the Unicode characters used within a Java program to the ASCII characters required by text files.

To perform stream output using ASCII characters, we instantiate an object of the class `PrintWriter`. The `PrintWriter` class provides methods for printing all of Java's primitive types, strings, generic objects (using the object's `toString` method), and arrays of characters. It also provides a method to close the output stream (`close`), methods to check and set errors (`checkError` and `setError`), and a method to flush the stream (`flush`). The `flush` method is used to force all of the current output to go immediately to the file. In `TDIncDate` we only use `PrintWriter`, `println`, and `close` methods. The `println` method sends a textual representation of its parameter to the output stream, followed by a linefeed. For example, the code:

```
outFile.println("Month is " + theDate.monthIs());
```

transforms the `int` returned by the `monthIs` method into a string, concatenates that string to the string "Month is", transforms the entire string into an ASCII representation, appends a linefeed character, and sends the whole thing to the output stream. You can see many other uses of the `println` method throughout the rest of the program. The `close` method is invoked when processing is finished:

```
outFile.close();
```

Invoking `close` informs the system that we are finished using the file. It is important for system efficiency and stability for a program to close files when it is finished using them.

So far in this discussion, we have referred to sending textual information to the "output stream." But how is this output stream associated with the correct file? The answer to this question is found by looking at the declaration of the `PrintWriter` object used in the program:

```
PrintWriter outFile = new PrintWriter(new FileWriter(outFileName));
```

Embedded within the `PrintWriter` declaration is an invocation of a `FileWriter` constructor:

```
new FileWriter(outFileName)
```

The `FileWriter` class is another subclass of `Writer`. The code invokes the `FileWriter` constructor and instantiates an object of the class `Writer` that is associated with the file represented by the variable `outFileName`. Recall that `outFileName` is the name of the output file that was passed to the program as a command-line argument. By embedding this code within the `PrintWriter` declaration, we associate the `PrintWriter` object `outFile` with the text file represented by `outFileName`. In our example above this is the `OutFileA` file. Therefore, a command such as:

```
outFile.println("Month is " + theDate.monthIs());
```

sends its output to the `OutFileA` file.

File Input

Most of the previous discussion about file output can be applied to file input. Instead of using the abstract class `Writer` we use the abstract class `Reader`; instead of `PrintWriter` we use `BufferedReader`; instead of the `println` method we use the `readLine` method; instead of the `FileWriter` class we use the `FileReader` class. We leave it to the reader to look over the `TDIncDate` program to see how the various file reading statements interact with each other. We do, however, briefly discuss the `readLine` method.

The `BufferedReader readLine` method returns a string that holds the next line of characters from the input stream. Therefore, a statement such as:

```
command = dataFile.readLine();
```

sets the string variable `command` to reference the next line of characters from the file associated with the object `dataFile`. In some cases we need to transform this line of characters into an integer. To do this we use the `parseInt` method of the `Integer` wrapper class:

```
day = Integer.parseInt(dataFile.readLine());
```

An alternate approach is to use the `intValue` method of the `String` class, and the `valueOf` method of the `Integer` wrapper class as follows:

```
day = Integer.valueOf(dataFile.readLine()).intValue;
```

Wrapper classes are discussed in Chapter 2.

Frame Output

We really cannot do justice to the topic of graphical user interfaces (GUIs) in this textbook. The topic is a nontrivial, important area of computing and deserves serious study. Nevertheless, modern programming approaches demand the use of GUIs and we make moderate use of them in our programs. So, without trying to explain all of the underlying concepts and supporting classes, we look at the purpose of each of the statements related to frame output. (Figure 1.15 shows the displayed frame.)

Note that our `TDIncDate` class includes the following *import* statements:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

The first statement imports classes from the Java library `awt` package; the second imports classes related to event handling, also from the Java library `awt` package; the third imports the classes of the Java `swing` package. The AWT (Abstract Window Toolkit) was the set of graphical interface tools included with the original version of Java. Developers found that this set of tools was too limited for professional program development, so the Java designers included a new set of graphical components, called the "Swing" components, when they released the Java Foundation Classes in 1997. The Swing components are more portable and flexible than their AWT counterparts. We use Java Swing components throughout the text. Note that Java Swing is built on top of Java AWT, so we still need to import AWT classes.

The code related to the frame output begins with the comment:

```
//Set up output frame
```

and continues to the end of the program listing. First, let's address the set-up of the frame itself. A frame is a top-level window with a title, a border, a menu bar, a content pane, and more. We declare our frame with the statement:

```
JFrame outputFrame = new JFrame();
```

`JFrame` is the Java Swing frame component (you can recognize Java Swing components since they begin with the letter "J" to differentiate them from their AWT counterparts). Therefore, our `outputFrame` object is a `JFrame`, and can be manipulated with the library methods defined for `JFrames`.

We immediately make use of three of these methods to set up our frame:

```
outputFrame.setTitle("Testing " + testName);
outputFrame.setSize(300,100);
outputFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

These statements set the title and size for the instantiated frame, and define how the frame should react to the user closing the frame's window. Setting the title and size are very straightforward. The title of our frame is "Testing IncDate," since the variable `testname` was set to "IncDate" at the beginning of the main method. The size of the frame is set to 300 pixels wide by 100 pixels tall.

Defining how the frame reacts to the user closing the frame's window is a little more complicated. When the frame is eventually displayed, it appears in its own window. Normally, when you define a window from within a Java program, you must define how the window reacts to various events: closing the window, resizing the window, activating the window, and so on. You must define methods to handle all of these events. However, in our program we want to handle only one of these events, the window-closing event. Java provides a special method, just for handling this event; the `setDefaultCloseOperation` method. This method tells the `JFrame` what to do when its window is closed, as long the action is one of a small set of common choices. The `JFrame` class provides the following class constants that name these choices:

```
JFrame.DISPOSE_ON_CLOSE
JFrame.DO_NOTHING_ON_CLOSE
JFrame.HIDE_ON_CLOSE
JFrame.EXIT_ON_CLOSE
```

In our program we use the `EXIT_ON_CLOSE` option, so the program disposes of the window and exits when the user closes the window.

The following two lines set up our frame output:

```
Container contentPane = outputFrame.getContentPane();
JPanel infoPanel      = new JPanel();
```

The first line provides us a "handle" for the content pane of the new frame. Remember that frames have many parts; the part where we display information is called the "content pane." We now have access to the content pane of our frame through the `contentPane` variable. This variable is an object of the class `Container`, which means we can place other objects into it for display purposes. What can we place into it? We can place almost anything: buttons, labels, drawings, text boxes; but to help us organize our interfaces we prefer to place yet another container object, called a panel, into content panes. The second line instantiates a `JPanel` object (the Swing version of a panel) called `infoPanel`. It is here where we place the information we want to display.

We next set a particular layout scheme for the `infoPanel` panel with the command:

```
infoPanel.setLayout(new GridLayout(2,1));
```

When we add items to the panel, they are organized according to the layout scheme defined in the above statement. We have chosen to use the grid layout scheme with 2 rows and 1 column. The Java Library provides many other layout schemes.

Next we create a new "label," containing information we wish to display on the screen. A label is a component that can hold one line of text; nothing fancy, just a line of text. That is all we need here. This is accomplished by the statements:

```
JLabel countInfo = new JLabel(numCommands + " commands completed.  ");
JLabel finishedInfo = new JLabel("Testing completed. "
                                  + "Close window to exit program.");
```

Finally, we add our information to the panel and display it with:

```
infoPanel.add(countInfo);
infoPanel.add(finishedInfo);
contentPane.add(infoPanel);
outputFrame.show();
```

The first two `add` method invocations add the labels to the `infoPanel`. The third `add` method invocations adds the `infoPanel` to the `contentPane` (which is already associated with the `outputFrame`). The `show` method displays the `outputFrame` on the monitor. That's it.

In summation, to perform frame output, the `TDIncDate` program does the following:

1. Imports classes from the `awt` and `swing` packages
2. Instantiates a new `JFrame` object
3. Obtains the content pane of the new frame
4. Creates a panel to hold information
5. Defines the layout of the panel
6. Instantiates labels with the information to display
7. Adds these labels to the panel
8. Adds the panel to the content pane
9. Shows the frame

Using this frame output approach allows us to use window output without getting bogged down in too much detail. When we run our test driver program, it reads data from the input file and writes results to the output file. It then creates an output frame as a separate program thread and reports summary information about the test results there. Note that when the main thread of the program finishes, the frame thread is still running. It will run until the user closes the frame's window, activating the window-closing event that we defined through the `set-DefaultCloseOperation` method.

Practical Considerations

It is obvious from this chapter that program verification techniques are time-consuming and, in a job environment, expensive. It would take a long time to do all of the things discussed in this chapter, and a programmer has only so much time to work on any particular program. Certainly not every program is worthy of such cost and effort. How can you tell how much and what kind of verification effort is necessary?

A program's requirements may provide an indication of the level of verification needed. In the classroom, your professor may specify the verification requirements as part of a programming assignment. For instance, you may be required to turn in a written, implemented test plan. Part of your grade may be determined by the completeness of your plan. In the work environment, the verification requirements are often specified by a customer in the contract for a particular programming job. For instance, a contract with a customer may specify that formal reviews or inspections of the software product be held at various times during the development process.

A higher level of verification effort may be indicated for sections of a program that are particularly complicated or error-prone. In these cases, it is wise to start the verification process in the early stages of program development in order to prevent costly errors in the design.

A program whose correct execution is critical to human life is obviously a candidate for a high level of verification. For instance, a program that controls the return of astronauts from a space mission would require a higher level of verification than a program that generates a grocery list. As a more down-to-earth example, consider the potential for disaster if a hospital's patient database system had a bug that caused it to lose information about patients' allergies to medications. A similar error in a database program that manages a Christmas card mailing list, however, would have much less severe consequences.

Summary

How are our quality software goals met by the strategies of abstraction and information hiding? When we hide the details at each level, we make the code simpler and more readable, which makes the program easier to write, modify, and reuse. Object-oriented design processes produce modular units that are also easier to test, debug, and maintain.

One positive side effect of modular design is that modifications tend to be localized in a small set of modules, and thus the cost of modifications is reduced. Remember that whenever we modify a module we must retest it to make sure that it still works correctly in the program. By localizing the modules affected by changes to the program, we limit the extent of retesting needed.

Finally, we increase reliability by making the design conform to our logical picture and delegating confusing details to lower levels of abstraction. By understanding the wide range of activities involved in software development—from requirements analysis through the maintenance of the resulting program—we gain an appreciation of a disciplined software engineering approach. Everyone knows some programming wizard who can sit down and hack out a program in an evening, working alone, coding without a formal design. But we cannot depend on wizardry to control the design, implementation, verification, and maintenance of large, complex software projects that involve the efforts of many programmers. As computers grow larger and more powerful, the problems that people want to solve on them also become larger and more complex. Some people refer to this situation as a software *crisis*. We'd like you to think of it as a software *challenge*.

It should be obvious by now that program verification is not something you begin the night before your program is due. Design verification and program testing go on throughout the software life cycle.

Verification activities begin when we develop the software specifications. At this point, we formulate the overall testing approach and goals. Then, as program design work begins, we apply these goals. We may use formal verification techniques for parts of the program, conduct design inspections, and plan test cases. During the implementation phase, we develop test cases and generate test data to support them. Code inspections give us extra support in debugging the program before it is ever run. Figure 1.16 shows how the various types of verification activities fit into the software development cycle. Throughout the life cycle, one thing remains the same: the earlier in this cycle we can detect program errors, the easier (and less costly in time, effort, and money) they are to remove. Program verification is a serious subject; a program that doesn't work isn't worth the disk it's stored on.

Analysis	Make sure that requirements are completely understood.
	Understand testing requirements.
Specification	Verify the identified requirements.
	Perform requirements inspections with your client.
Design	Design for correctness (using assertions such as preconditions and postconditions).
	Perform design inspections.
	Plan testing approach.
Code	Understand programming language well.
	Perform code inspections.
	Add debugging output statements to the program.
	Write test plan.
	Construct test drivers.
Test	Unit test according to test plan.
	Debug as necessary.
	Integrate tested modules.
	Retest after corrections.
Delivery	Execute acceptance tests of complete product.
Maintenance	Execute regression test whenever delivered product is changed to add new functionality or to correct detected problems.

Figure 1.16 *Life-cycle verification activities*

Summary of Classes and Support Files

In this section at the end of each chapter we summarize, in tabular form, the classes defined in the chapter. The classes are listed in the order in which they appear in the text. We also include information about any other files, such as test input files, that support the material. The summary includes the name of the file, the page on which the class or support file is first referenced, and a few notes. The notes explain how the class or support file was used in the text, followed by additional notes if appropriate. The class and support files are available on our web site. They can be found in the `ch01` subdirectory of the `bookFiles` subdirectory.

Classes and Support Files Defined in Chapter 1

File	First Ref.	Notes
`Date.java`	page 14	Example of a Java class with instance and class variables. Unlike the original code in the text, the code on our web site includes a `toString` method.
`IncDate.java`	page 18	Demonstrates inheritance. The code for the `increment` command is not included (see Exercise 34).
`TDIncDate.java`	page 51	Example of a test driver; test driver for the `IncDate` class. In Exercise 36 we ask the student to enhance the code to include automated test verification.
`TestDataA`	page 50	Input file for `TDIncDate`.

We also include in this summary section a list of any Java library classes that were used for the first time for the classes defined in the chapter. For each library class we list its name, its package, any of its methods that are explicitly used, and the name of the program/class where they are first used. The classes are listed in the order in which they are first used. Note that in some classes the methods listed might not be defined directly in the class; they might be defined in one of its superclasses. With the classes we also list constructors, if appropriate. For more information about the library classes and methods, check the Sun Java documentation.

Library Classes Used in Chapter 1 for the First Time

Class Name	Package	Overview	Methods Used	Where Used
`JFrame`	`swing`	Manages a graphical window	`addWindowListener`, `getContentPane`, `show`, `setSize`, `setTitle`	`TDIncDate`
`String`	`lang`	Creates and parses strings	`equals`, `String`	`TDIncDate`
`BufferedReader`	`io`	Provides a buffered stream of character data	`BufferedReader`, `readLine`, `close`	`TDIncDate`
`FileReader`	`io`	Allows reading of characters from a file	`FileReader`	`TDIncDate`
`PrintWriter`	`io`	Outputs a buffered stream of character data	`PrintWriter`, `println`, `close`	`TDIncDate`
`FileWriter`	`io`	Allows reading of characters from a file	`FileWriter`	`TDIncDate`
`Container`	`awt`	Provides a container that can hold other containers	`add`	`TDIncDate`
`Jpanel`	`swing`	Provides a container for organizing display information	`add`, `JPanel`, `setLayout`	`TDIncDate`
`GridLayout`	`awt`	Creates a rectangular grid scheme for output	`GridLayout`	`TDIncDate`
`JLabel`	`swing`	Holds one line of text for display	`JLabel`	`TDIncDate`
`WindowAdapter`	`awt`	Provides null methods for window events	`WindowAdapter`	`TDIncDate`
`System`	`lang`	Various system-related methods	`exit`	`TDIncDate`
`Integer`	`lang`	Wraps the primitive `int` type	`parseInt`	`TDIncDate`

Exercises

1.1 The Software Process

1. Explain what we mean by "software engineering."
2. List four goals of quality software.
3. Which of these statements is always true?
   a. All of the program requirements must be completely defined before design begins.
   b. All of the program design must be complete before any coding begins.
   c. All of the coding must be complete before any testing can begin.
   d. Different development activities often take place concurrently, overlapping in the software life cycle.
4. Explain why software might need to be modified
   a. in the design phase.
   b. in the coding phase.
   c. in the testing phase.
   d. in the maintenance phase.
5. Goal 4 says, "Quality software is completed on time and within budget."
   a. Explain some of the consequences of not meeting this goal for a student preparing a class programming assignment.
   b. Explain some of the consequences of not meeting this goal for a team developing a highly competitive new software product.
   c. Explain some of the consequences of not meeting this goal for a programmer who is developing the user interface (the screen input/output) for a spacecraft launch system.
6. Name three computer hardware tools that you have used.
7. Name two software tools that you have used in developing computer programs.
8. Explain what we mean by "ideaware."

1.2 Program Design

9. For each of the following, describe at least two different abstractions for different viewers (see Figure 1.1).
   a. A dress
   b. An aspirin
   c. A carrot
   d. A key
   e. A saxophone
   f. A piece of wood
10. Describe four different kinds of stepwise refinement.
11. Explain how to use the nouns and verbs in a problem description to help identify candidate design classes and methods.
12. Find a tool that you can use to create UML class diagrams and recreate the diagram of the `Date` class shown in Figure 1.3.

13. What is the difference between an object and a class? Give some examples.
14. Describe the concept of inheritance, and explain how the inheritance tree is traversed to bind method calls with method implementations in an object-oriented system.
15. Make a list of potential objects from the description of the automated-teller-machine scenario given in this chapter.
16. Given the definition of the `Date` and `IncDate` classes in this chapter, and the following declarations:

    ```
    int temp;
    Date date1 = new Date(10,2,1989);
    Date date2 = new Date(4,2,1992);
    IncDate date3 = new IncDate(12,25,2001);
    ```

    indicate which of the following statements are illegal, and which are legal. Explain your answers.

    a. `temp = date1.dayIs();`
    b. `temp = date3.yearIs();`
    c. `date1.increment();`
    d. `date3.increment();`
    e. `date2 = date1;`
    f. `date2 = date3;`
    g. `date3 = date2;`

1.3 Verification of Software Correctness

17. Have you ever written a programming assignment with an error in the specifications? If so, at what point did you catch the error? How damaging was the error to your design and code?
18. Explain why the cost of fixing an error is increasingly higher the later in the software cycle the error is detected.
19. Explain how an expert understanding of your programming language can reduce the amount of time you spend debugging.
20. Explain the difference between program verification and program validation.
21. Give an example of a run-time error that might occur as the result of a programmer making too many assumptions.
22. Define “robustness.” How can programmers make their programs more robust by taking a defensive approach?
23. The following program has two separate errors, each of which would cause an infinite loop. As a member of the inspection team, you could save the programmer a lot of testing time by finding the errors during the inspection. Can you help?

```
import java.io.PrintWriter;
public class TryIncrement
{
  static PrintWriter output = new PrintWriter(System.out,true);

  public static void main(String[] args) throws Exception
  {
    int count = 1;
    while(count < 10)
      output.println(" The number after " + count);    /* Now we will
      count = count + 1;                              add 1 to count */
      output.println(" is " + count);
  }
}
```

24. Is there any way a single programmer (for example, a student working alone on a programming assignment) can benefit from some of the ideas behind the inspection process?
25. When is it appropriate to start planning a program's testing?
    a. During design or even earlier
    b. While coding
    c. As soon as the coding is complete
26. Describe the contents of a typical test plan.
27. Devise a test plan to test the `increment` method of the `IncDate` class.
28. A programmer has created a module `sameSign` that accepts two `int` parameters and returns `true` if they are both the same sign, that is, if they are both positive, both negative, or both zero. Otherwise, it returns `false`. Identify a reasonable set of test cases for this module.
29. Explain the advantages and disadvantages of the following debugging techniques:
    a. Inserting output statements that may be turned off by commenting them out
    b. Using a Boolean flag to turn debugging output statements on or off
    c. Using a system debugger
30. Describe a realistic goal-oriented approach to data-coverage testing of the method specified below:

public boolean FindElement(list, targetItem)

Effect:	Searches list for targetItem.
Preconditions:	Elements of list are in no particular order; list may be empty.
Postcondition:	Returns true if targetItem is in list; otherwise, returns false.

31. A program is to read in a numeric score (0 to 100) and display an appropriate letter grade (A, B, C, D, or F).
    a. What is the functional domain of this program?
    b. Is exhaustive data coverage possible for this program?
    c. Devise a test plan for this program.
32. Explain how paths and branches relate to code coverage in testing. Can we attempt 100% path coverage?
33. Explain the phrase "life-cycle verification."
34. Create a `Date` class and an `IncDate` class as described in this chapter (or copy them from the web site). In the `IncDate` class you must create the code for the `increment` method, since that was left undefined in the chapter. Remember to follow the rules of the Gregorian calendar: A year is a leap year if either (i) it is divisible by 4 but not by 100 or (ii) it is divisible by 400. Include the preconditions and postconditions for `increment`. Use the `TDIncDate` program to test your program.
35. You should experiment with the frame output of the `TDIncDate` program. Follow the directions and record the results:
    a. Create a test input file called `MyTest.dat`.
    b. Run the program using `MyTest.dat` as the test input file, and `MyTest.out` as the output file.
    c. Change the `TestDriverFrame.java` class so that it sets the frame size to 500 × 300, and run the program again.
    d. Change the grid layout statement from a grid of 2,1 to a grid of 1,2, and run the program again.
    e. Experiment with other layout managers; use the available resources for information about them.
36. Enhance the `TDIncDate` program to include automatic test-case verification. For each of the commands that can be listed in the test-input file, you need to identify a test-result value, to be used to verify that the command was executed properly. For example, the constructor command `IncDate` can be verified by comparing the resultant value of the `IncDate` object to the date represented by the parameters of the command; the observer command `monthIs` can be verified by checking the value returned by the `monthIs` method to the expected month. The values needed to verify each command should follow the command and its parameters in the test input file. For example, a test input file could look like this:

```
IncDate Test Data B
IncDate
10
5
2002
10/5/2002
monthIs
10
quit
```

The test driver should read a command, read the command's parameters if necessary, execute the command by invoking the appropriate method, and then validate that the command completed successfully by comparing the results of the command to the test result value from the input file. The results of the test (pass or fail) should be written to the output file, and a count of the number of test cases passed and failed should be written to the screen.

37. Create a new program that uses the same basic architecture as the test driver program modeled in Figure 1.14, and that uses the same set of Java I/O statements as `TDIncDate(readLine`, `setLayout`, and so on). This is an open problem; your program can do whatever you like. For example, the input file could contain a list of student names plus three test grades for each student:

```
Smith
100
90
80
Jones
95
95
95
```

And the corresponding output file could contain the student's names and averages:

```
Smith
90
Jones
95
```

Finally, the output frame could contain summary information: for example, the number of students, the total average, the highest average, and so on. Remember to design your program so that the user can indicate the input and output file names through command-line parameters.