In this chapter, we will introduce the main topic of this book: object-oriented design. The chapter introduces a miniature version of a typical object-oriented design methodology that can guide you from the...
functional specification of a program to its implementation. You will see how to find and document classes and the relationships between them, using CRC cards and UML diagrams.

2.1 From Problem to Code

This book discusses the design and implementation of computer programs from the object-oriented point of view. We focus on small and medium-sized problems. Although much of what we say remains valid for large projects, there are added complexities with large projects that we will not address here.

Programming tasks originate from the desire to solve a particular problem. The task may be simple, such as writing a program that generates and formats a report, or complicated, such as writing a word processor. The end product is a working program. To this end, it is a common practice to break up the software development process into three phases:

- Analysis
- Design
- Implementation

This section briefly discusses the goals and methods of these phases. Of course, it is simplistic to assume that development is a simple linear progression through these phases. Successful software products evolve over time. Implementation experiences may suggest an improved design. New requirements are added, forcing another iteration through analysis and design. Experience suggests that object-oriented design can lead to software that facilitates the inevitable evolution better than software developed with traditional methods because the objects and classes that represent the concepts of a problem domain tend to be fairly stable.

2.1.1 The Analysis Phase

In the analysis phase, a vague understanding of the problem is transformed into a precise description of the tasks that the software system needs to carry out. The result of the analysis phase is a detailed textual description, commonly called a functional specification, that has the following characteristics:

- It completely defines the tasks to be performed.
- It is free from internal contradictions.
- It is readable both by experts in the problem domain and by software developers.
- It is reviewable by diverse interested parties.
- It can be tested against reality.
Consider, for example, the task of writing a word-processing program. The analysis phase must define terms, such as fonts, footnotes, multiple columns, and document sections, and the interaction of those features, such as how footnotes in multiple-column text ought to look on the screen and the printed page. The user interface must be documented, explaining, for example, how the user is to enter and move a footnote or specify the font for footnote numbers. One possible format for an analysis document is a user manual, very precisely worded to remove as much ambiguity as possible.

Another common format for describing the behavior of a system is a set of use cases. A use case is a description of a sequence of actions that yields a benefit for a user of a system. At least in principle, it should be possible to enumerate all benefits that a system can confer upon its users and supply use cases that show how they can be obtained. The analysis phase concerns itself with the description of what needs to be done, not how it should be done. The selection of specific algorithms, such as those that insert page breaks or sort the index, will be handled in the implementation phase.

Although we do not do so in this book, it is possible to use object-oriented techniques in the analysis phase as well as the design phase. An advantage of that approach is that the object model of the analysis phase can be carried forward to the design phase. A potential pitfall is that customers of a software product are generally not familiar with the terminology of object orientation. Clients may not find it easy to tell whether the analysis will lead to a product that satisfies their needs.

### 2.1.2  The Design Phase

In the design phase, the program designer must structure the programming tasks into a set of interrelated classes. Each class must be specified precisely, listing both its responsibilities and its relationship to other classes in the system. You will study this process in this book in some detail.

The designer must strive for a result in which the classes are crisply defined and class relationships are of manageable complexity. The exact choice of data structures, for example, hash tables or binary search trees for a collection, is not of concern in the design phase but is deferred until implementation. Even the choice of programming language is not a design issue. It is possible to map an object-oriented design to a programming language without object-oriented features, although that process can be somewhat unpleasant.

Here are the major goals of the design phase:

- Identify the classes
- Identify the responsibilities of these classes
- Identify the relationships among these classes

These are goals, not steps. It is usually not possible to find all classes first, then give a complete description of their responsibilities, then elaborate on their relationships. The discovery process is iterative—the identification of one aspect of a class may force changes in or lead to the discovery of others.
The end result of the design process consists of a number of artifacts:

- A textual description of the classes and their most important responsibilities
- Diagrams of the relationships among the classes
- Diagrams of important usage scenarios
- State diagrams of objects whose behavior is highly state-dependent

Depending on the tool support, this information may be stored on paper, in text and graphics files, or in a CASE (computer-assisted software engineering) tool database.

The information gathered in this phase becomes the foundation for the implementation of the system in an actual programming language. Typically, the design phase is more time-consuming than the actual programming, or—to put a positive spin on it—a good design greatly reduces the time required for implementation and testing.

2.1.3 The Implementation Phase

In the implementation phase, the classes and methods are coded, tested, and deployed. A part of this book concerns itself with the problems of implementing an object-oriented design in Java.

Traditional programming methods rely on completion and unit testing of procedural units, followed by an integration phase. This integration tends to be frustrating and disappointing. Few programs are born according to plan out of a successful “big bang” integration. Object-oriented development encourages the gradual growth of a program by successively attaching more working classes and class clusters and repeated testing.

It is quite common to defer the implementation of some operations and build a “rapid prototype” that displays some functionality of the final product. Such a prototype can be extremely helpful in influencing the design or even the problem analysis, especially in cases where a problem was so incompletely understood that seeing a prototype do some work gives more insights into the solutions that are really desired.

You should not rush the analysis and design phase just to get to a working prototype quickly, nor should you hesitate to reopen the previous phases if a prototype yields no useful insight.

Object-oriented design is particularly suited for prototyping. The objects supporting the prototype are likely to be the same that need to be present in the final product, and attaching the prototype into a complete program is often feasible. Some developers welcome this; others caution against it because prototypes are often rushed and without sufficient time to work them over carefully. In fact, some people recommend implementing prototypes in a language such as Visual Basic and then writing the final product in a language such as Java. For small to medium-sized products, a prototype can expand into a complete product. If you follow this evolutionary approach, be sure that the transition from prototype to final product is well managed and that enough time is allocated to fix mistakes and implement newly discovered improvements.

For the remainder of this chapter, we will mainly be concerned with the design phase of a programming project, focusing on object-oriented design techniques.
The Object and Class Concepts

We assume that you have programmed with classes for some time, and that you are familiar with the mechanics of defining classes and constructing objects. Thus, you have a fairly good idea what objects and classes are in the context of Java. Let's take a higher-level view and think about the concepts of objects and classes outside any particular programming language.

Objects are entities in a computer program that have three characteristic properties:

- State
- Behavior
- Identity

An object can store information that is the result of its prior operations. That information may determine how the object behaves in the future. The collection of all information held by an object is the object's state. An object's state may change over time, but only when an operation has been carried out on the object that causes the state change.

Consider the example of a mailbox in a voice mail system. A mailbox object may be in an empty state (immediately after its creation) or full (after receiving a large number of messages). This state affects the behavior of the mailbox object: A full mailbox may reject new mail messages, whereas an empty mailbox may give a special response ("no messages waiting") when asked to list all new messages.

The behavior of an object is defined by the operations (or methods, as they are called in Java) that an object supports. Objects permit certain operations and do not support others. For example, a mailbox can add a mail message to its collection or retrieve a stored message, but it cannot carry out other operations such as "translate the stored messages into Lithuanian".

Object-oriented programs contain statements in which objects are asked to carry out certain operations. Because not all operations are suitable for all objects, there must be a mechanism for rejecting improper requests. Object-oriented programming systems differ in this regard. Some systems attempt to weed out unsupported operations at compile time; others generate run-time errors.

The momentary state and the collection of admissible operations, however, do not fully characterize an object. It is possible for two or more objects to support the same operations and to have the same state, yet to be different from each other. Each object has its own identity. For example, two different mailboxes may, by chance, have the same contents, yet the program can tell them apart.

Some researchers define objects as entities that have state, behavior, and identity. This definition is somewhat unsatisfactory—what, after all, is an "entity"? The definition is also quite broad. As one computer scientist has pointed out, it then follows that his cat is an object: It has a rich internal state (hungry, purring, sleeping); it carries out certain
The following nouns are typical of those that can be found in the functional description of a voice mail system:

- Mailbox
- Message
- User
- Passcode
- Extension
- Menu

Many, but not necessarily all of them, are good choices for classes.

**TIP** Make sure not to fall into the trap of making your designs too specific. Suppose you are designing software to process orders for kitchen appliances such as toasters and blenders. If you let the object-oriented design process run amok, you end up with classes *KitchenAppliance*, *Toaster*, and *Blender*. But wait—the kitchen appliance hierarchy is irrelevant to our problem, namely to process orders for products. A *Product* class is probably a better choice.

Don't fall into the opposite trap of making your designs unreasonably general. Consider the mail system example. A mailbox is a kind of component, and there are connections between various components. Connections can carry data (such as messages). Should you therefore design classes *Component*, *Connection*, and *Data*? No—those classes are too general. You would not be able to come up with clear responsibilities for them, and you would be no closer to a solution of your design problem.

After you have harvested the classes that are obvious from the program specification, you need to turn your attention to other classes that are required to carry out necessary work. For example, consider the storage of messages in a mailbox. The mailbox owner wants to listen to the messages in the order in which they were added. In other words, messages are inserted and retrieved in a *FIFO* (first in, first out) fashion. Computer scientists defined the *queue* data type to describe this behavior, and they have discovered several implementations of this data type, some of which are more efficient than others. (See the note at the end of this section for more information about queues.) During design time, it makes sense to describe a class *MessageQueue* and its FIFO behavior. However, the exact implementation of the queue is of no interest in the design phase.

Class names should be nouns in the singular form: *Message*, *Mailbox*. Sometimes the noun needs to be prefixed by an adjective or participle: *RectangularShape*, *BufferedReader*. Don't use *Object* in the class name (*MailboxObject*)—it adds no value. Unless you are solving a very generic problem, stay away from generic names such as *Agent*, *Task*, *Item*, *Event*, *User*. If you name your classes after verbs (such as *Deliver* or *Printing*), you are probably on the wrong track.
After you go beyond the technique of finding nouns in the functional specification, it is useful to look at other categories of classes that are often helpful. Here are some of these categories:

- **Tangible things**
- **Agents**
- **Events and transactions**
- **Users and roles**
- **Systems**
- **System interfaces and devices**
- **Foundational classes**

Tangible things are the easiest classes to discover because they are visible in the problem domain. We have seen many examples: Mailbox, Message, Document, Footnote.

Sometimes it is helpful to change an operation into an agent class. For example, the "compute page breaks" operation on a document could be turned into a Paginator class, which operates on documents. Then the paginator can work on a part of a document while another part is edited on the screen. In this case, the agent class is invented to express parallel execution.

The Scanner class is another example. As described in Chapter 1, a Scanner is used to scan for numbers and strings in an input stream. Thus, the operation of parsing input is encapsulated in the Scanner agent.

Agent classes often end in "er" or "or".

Event and transaction classes are useful to model records of activities that describe what happened in the past or what needs to be done later. An example is a MouseEvent class, which remembers when and where the mouse was moved or clicked.

User and role classes are stand-ins for actual users of the program. An Administrator class is a representation of the human administrator of the system. A Reviewer class in an interactive authoring system models a user whose role is to add critical annotations and recommendations for change. User classes are common in systems that are used by more than one person or where one person needs to perform distinct tasks.

System classes model a subsystem or the overall system being built. Their roles are typically to perform initialization and shutdown and to start the flow of input into the system. For example, we might have a class MainSystem to represent the voice mail system in its entirety.

System interface classes model interfaces to the host operating system, the windowing system, a database, and so on. A typical example is the File class.

Foundation classes are classes such as String, Date, or Rectangle. They encapsulate basic data types with well-understood properties. At the design stage, you should simply assume that these classes are readily available, just as the fundamental types (integers and floating-point numbers) are.
Queues

A queue is a very common data type in computer science. You add items to one end of the queue (the tail) and remove them from the other end of the queue (the head). To visualize a queue, simply think of people lining up (see Figure 1). People join the tail of the queue and wait until they have reached the head of the queue. Queues store items in a first in, first out or FIFO fashion. Items are removed in the same order in which they have been added.

There are many applications for queues. For example, the Java graphical user interface system keeps an event queue of all events, such as mouse and keyboard events. The events are inserted into the queue whenever the operating system notifies the application of the event. Another thread of control removes them from the queue and passes them to the appropriate event listeners. Another example is a print queue. A printer may be accessed by several applications, perhaps running on different computers. If all of the applications tried to access the printer at the same time, the printout would be garbled. Instead, each application places all bytes that need to be sent to the printer into a file and inserts that file into the print queue. When the printer is done printing one file, it retrieves the next one from the queue. Therefore, print jobs are printed using the FIFO rule, which is a fair arrangement for users of the shared printer.

The standard Java library defines a number of queue classes for multithreaded programming, but for simple queues, the library designers suggest that you just use the add and remove methods of the LinkedList class. We will consider a "circular array" implementation of a queue in the next chapter.

Figure 1

A Queue
To discover responsibilities, look for verbs in the problem description. Just as classes correspond to nouns in the problem description, responsibilities correspond to verbs. If you read through the functional description of the voice mail system in Section 2.12, you will find that messages are recorded, played, and deleted; users log in; passcodes are checked. When you discover a responsibility, you must find one class (and only one class) that owns that responsibility.

A responsibility must belong to exactly one class.

For some classes, finding responsibilities is quite easy because we are familiar with the territory. For example, any textbook on data structures will tell us the responsibilities of the MessageQueue class:

- Add a message to the tail of the queue.
- Remove a message from the head of the queue.
- Test whether the queue is empty.

With other classes, finding the right responsibilities is more difficult. Consider the following responsibility in a voice mail system.

- Add the message to a mailbox.

Is this a responsibility of the Message class? That is not a good idea. To see the reason, think how a message could perform the responsibility. In order to add itself to a mailbox, the message would need to know the internal structure of the mailbox. The details would depend on whether the mailbox uses an array list, a queue, or another data structure to hold its messages. But we always assume that those implementation details are private to the Mailbox class, and that the Message class has no insight into them.

In our situation, the responsibility of adding a message to a mailbox lies with the mailbox, not with the message. The mailbox has sufficient understanding of its structure to perform the operation.

When discovering responsibilities, programmers commonly make wrong guesses and assign the responsibility to an inappropriate class. For that reason, it is helpful to have more than one person involved in the design phase. If one person assigns a responsibility to a particular class, another can ask the hard question, “How can an object of this class possibly carry out this responsibility?” The question is hard because we are not yet supposed to get to the nitty-gritty of implementation details. But it is appropriate to consider a “reasonable” implementation, or better, two different possibilities, to demonstrate that the responsibility can be carried out.

TIP When assigning responsibilities, respect the natural layering of abstraction levels. At the lowest levels of any system, we have files, keyboard and mouse interfaces, and other system services. At the highest levels there are classes that tie together the software system, such as MailSystem. The responsibilities of a class should stay at one abstraction level. A class Mailbox that represents a mid-level abstraction should not deal with processing keystrokes, a low-level responsibility, nor should it be concerned with the initialization of the system, a high-level responsibility.
CHAPTER 2 The Object-Oriented Design Process

2.5 Relationships Between Classes

Three relationships are common among classes:

- Dependency ("uses")
- Aggregation ("has")
- Inheritance ("is")

We will discuss these three relationships in detail in this section.

2.5.1 Dependency

A class depends on another class if it manipulates objects of the other class in any way. For example, the class Mailbox in a voice mail system uses the Message class because Mailbox objects manipulate Message objects.

It is almost easier to understand when a class doesn't depend on another. If a class can carry out all of its tasks without being aware that the other class even exists, then it doesn't use that class. For example, the Message class does not need to use the Mailbox class at all. Messages need not be aware that they are stored inside mailboxes. However, the Mailbox class uses the Message class. This shows that dependency is an asymmetric relationship.

One important design goal is to minimize the number of dependency relationships; that is, to minimize the coupling between classes. If one class is unaware of the existence of another, it is also unconcerned about any changes in that other class. A low degree of coupling tends to make it much easier to implement changes in the future.

For example, consider this message class:

```java
public class Message {
    public void print() { System.out.println(text); }
}
```

The `print` method prints the message to `System.out`. Therefore, the `Message` class is coupled with both the `System` and the `PrintStream` classes. (The `System.out` object is an instance of the `PrintStream` class.)

If the class is deployed in an embedded device such as a real voice message system or a toaster oven, then there is no `System.out`. It would be better to have a method

```java
public String getText()
```

that returns the message text as a string. Then it is up to some other part of the system to send the string to `System.out`, to a dialog box, or to a speaker.
Aggregation takes place if objects of one class contain objects of another class over a period of time. For example, `MessageQueue` has `Message` objects, and we say that the `MessageQueue` class aggregates the `Message` class.

Aggregation is a special case of dependency. Of course, if a class contains objects of another class, then it is acutely aware of the existence of that class.

Aggregation is often informally described as the "has-a" relationship. A message queue has a message. Actually, a message queue has several messages. With aggregation relationships, it is useful to keep track of these multiplicities. There may be a 1:1 or 1:n relationship. For example, each mailbox has exactly one greeting (1:1), but each message queue may contain many messages (1:n).

Aggregation is usually implemented through instance fields. For example, if a mailbox has a greeting, then the Java implementation might look like this:

```java
public class Mailbox {
    private Greeting myGreeting;
}
```

This particular implementation can serve as a 1:1 or 1:0...1 relationship (if you allow `myGreeting == null` to indicate that there is no greeting for a particular mailbox). For a 1:n relationship, you need an array or a collection object. For example,

```java
public class MessageQueue {
    private ArrayList<Message> elements;
}
```

However, not all instance fields of a class correspond to aggregation. If an object contains a field of a very simple type such as a number, string, or date, it is considered merely an attribute, not aggregation. For example, suppose a message has a time stamp of type `Date`.

```java
public class Message {
    private Date timestamp;
}
```

We consider `Date` a foundational type, just like a number or a string. Thus, we don’t say that the `Message` class aggregates the `Date` class, but we consider the time stamp an attribute.
The distinction between aggregation and attributes depends on the context of your design. You'll need to make a judgment whether a particular class is "very simple", giving rise to attributes, or whether you should describe an aggregation relationship.

2.5.3 — Inheritance

A class inherits from another if all objects of its class are special cases of objects of the other class, capable of exhibiting the same behavior but possibly with additional responsibilities and a richer state.

Here is a typical example. Many voice mail systems let you forward a message that you received to another user. When the forwarded message is played, it first tells who forwarded it before playing the contents of the original message. We can model this feature by having the ForwardedMessage inherit from the Message class.

We call the more general class the superclass and the more specialized class the subclass. A subclass object must be usable in all situations in which a superclass object is expected. For example, a forwarded message object can be stored and played, just like any other message.

But a greeting in a voice mail system, even though it is in many respects similar to a message, is not usable in the same contexts as messages are. Users cannot store greetings in mailboxes. We conclude that Greeting may not inherit from Message.

Inheritance is often called the "is-a" relationship. This intuitive notion makes it easy to distinguish inheritance from aggregation. For example, a forwarded message is a message (inheritance) while a mailbox has a greeting (aggregation).

As you will see in Chapters 4 and 6, exploiting inheritance relationships can lead to very powerful and extensible designs. However, we must point out that inheritance is much less common than the dependency and aggregation relationships. Many designs can best be modeled by employing inheritance in a few selected places.

2.6 Use Cases

Use cases are an analysis technique to describe in a formal way how a computer system should work. Each use case focuses on a specific scenario, and describes the steps that are necessary to bring it to successful completion. Each step in a use case represents an interaction with people or entities outside the computer system (the actor) and the system itself. For example, the use case "Leave a message" describes the steps that a caller must take to dial an extension and leave a message. The use case "Retrieve messages" describes the steps needed to listen to the messages in the mailbox. In the first case, the actor is the caller leaving a message. In the second case, the actor is the mailbox owner.

An essential aspect of a use case is that it must describe a scenario that completes to a point that is of some value to one of the actors. In the case of "Leave a message", the value to the caller is the fact that the message is deposited in the appropriate mailbox. In contrast, merely
dialing a telephone number and listening to a menu would not be considered a valid use
case because it does not by itself have value to anyone.

Of course, most scenarios that potentially deliver a valuable outcome can also fail for one
reason or another. Perhaps the message queue is full, or a mailbox owner enters the
wrong password. A use case should include variations that describe these situations.

Minimally, a use case should have a name that describes it concisely, a main sequence of
actions, and, if appropriate, variants to the main sequence. Some analysts prefer a more
formal writeup that numbers the use cases, calls out the actors, refers to related use cases,
and so on. However, in this book we'll keep use cases as simple as possible.

Here is a sample use case for a voice mail system.

Leave a Message

1. The caller dials the main number of the voice mail system.
2. The voice mail system speaks a prompt.
   Enter mailbox number followed by #.
3. The user types in the extension number of the message recipient.
4. The voice mail system speaks.
   You have reached mailbox xxxx. Please leave a message now.
5. The caller speaks the message.
6. The caller hangs up.
7. The voice mail system places the recorded message in the recipient's mailbox.

Variation #1

1.1. In Step 3, the user enters an invalid extension number.
1.2. The voice mail system speaks.
   You have typed an invalid mailbox number.
1.3. Continue with Step 2.

Variation #2

2.1. After Step 4, the caller hangs up instead of speaking a message.
2.2. The voice mail system discards the empty message.

INTERNET The Web site http://www.usecases.org/ contains a template for a more elabo-
rate use case format. The "Use Case Zone" at http://www.pols.co.uk/use-case-zone/
has many useful links to articles that report on experiences with use cases, including some
interesting cautionary tales.
The CRC card method is an effective *design* technique for discovering classes, responsibilities, and relationships. A *CRC card* is simply an index card that describes one class and lists its responsibilities and *collaborators* (dependent classes). Index cards are a good choice for a number of reasons. They are small, thereby discouraging you from piling too much responsibility into a single class. They are low-tech, so that they can be used by groups of designers gathered around a table. They are more rugged than sheets of paper and can be handed around and rearranged during brainstorming sessions.

You make one card for each discovered class. Write the class name at the top of the card. Below, on the left-hand side, you describe the responsibilities. On the right-hand side, you list other classes that need to collaborate with this class so that it can fulfill its responsibilities.

The CRC card shown in Figure 2 indicates that we have discovered three responsibilities of the mailbox: to manage the passcode, to manage the greeting, and to manage new and saved messages. The latter responsibility requires collaboration with the MessageQueue class. That is, the mailbox needs to interact with MessageQueue objects in some unspecified way.

The responsibilities should be at a *high level*. Don't write individual methods. If a class has more responsibilities than you can fit on the index card, you may need to make two

---

**Figure 2**

A CRC Card

<table>
<thead>
<tr>
<th>Mailbox</th>
</tr>
</thead>
<tbody>
<tr>
<td>manage passcode                     MessageQueue</td>
</tr>
<tr>
<td>manage greeting</td>
</tr>
<tr>
<td>manage new and saved messages</td>
</tr>
</tbody>
</table>

---
new cards, distribute the responsibilities among them, and tear up the old card. Between one and three responsibilities per card is ideal.

**TIP** Programmers who start out with the CRC card technique sometimes equate responsibilities with methods. Keep in mind that responsibilities are at a high level. A single responsibility may give rise to a number of methods. If you find that your card contains lots of related responsibilities, try to express some of them at a higher level. For example, you may want to replace “manage passcode” and “manage greeting” with “manage user options”.

The collaborators don’t have to be on the same lines as the responsibilities. Simply list collaborators as you discover them, without regard for the ordering.

CRC cards are quite intuitive for “walking through” use cases. Consider, for example, the use case “Leave a message”. The caller dials the main number and is connected to the voice mail system. That happens through the public telephone system and is outside our concern. Next, the caller dials the extension. Now “someone” inside the voice mail program needs to locate the mailbox that has the given extension number. Neither the Mailbox nor the Message class can handle this responsibility. Perhaps a mailbox knows its own extension number, but it doesn’t know about the extension numbers of the other mailboxes in the system. And a message doesn’t know anything about mailboxes and extension numbers. A MailSystem knows about all of its mailboxes, so it would be a reasonable choice for a responsible agent. Let’s create a new index card, shown in Figure 3.

**TIP** Beware of the omnipotent system class. You often need a class that is responsible for coordinating the working of the system that you are building, but there is a tremendous danger of overburdening that class with too many responsibilities. Have a look at the evolution of the MailSystem class throughout this chapter and see if we manage to keep it under control.
CHAPTER 2  The Object-Oriented Design Process

**TIP** Beware of classes with magical powers that have no connection with the real world or computer systems. A MailSystem is actually quite real—when you buy a commercial voice mail system, you get a box into which you plug phone cables. But you can’t just define your own “systems”. If in doubt, check with experts that understand the problem domain.

Next, imagine how the mail system is going to locate the mailbox. Maybe each mailbox knows its number, and the mail system asks each one of the mailboxes “are you number $x$”? Or maybe the mailboxes don’t know their numbers, and the mail system keeps a table that lists all extension numbers and their mailboxes? Either one is a reasonable mechanism, and you don’t have to settle for one or the other at this stage. All that matters is that you are convinced that the mail system can do the job.

Let’s finish the use case. The mail system has located the appropriate mailbox. It now needs to deliver the message to that mailbox. Look again at the Mailbox CRC card. It has a responsibility “manage new and saved messages”. Thus, it seems to be up to the job of storing the message. Now you should add the Mailbox class as a collaborator of the MailSystem class. The mail system needs the collaboration of the mailbox to complete the delivery.

**TIP** Avoid “mission creep”. If a class acquires too many responsibilities, then consider splitting it in two. Ideally, a class should not have more than three high-level responsibilities.

**TIP** Watch out for unrelated responsibilities. A class should represent a coherent concept, with related responsibilities. If the Mailbox class gets charged with storing messages and parsing input, make a new class and split the responsibilities.

**TIP** Resist the temptation to add responsibilities just because they can be done. For example, someone may have suggested a Mailbox responsibility “sort messages”. But the task at hand requires no sorting, and you shouldn’t collect unused responsibilities.

**TIP** A class with no responsibilities surely is not useful. Try eliminating classes that don’t seem to contribute to solving the problem at hand. Typical candidates are vague mechanisms such as Connector and Data.

The walkthroughs with CRC cards are particularly suited for group discussion. Let’s assume the analysts are done with their work and have left behind a stack of use cases. Get two or three designers together. Here is a good way to “break the ice” and get started. Let all participants use the “noun and verb” technique to come up with a pool of candidates for classes and operations. Then consider the first use case that looks interesting and perform a walkthrough. Have one person play the protagonist, who proposes a responsible agent and a method for carrying out the task. Invariably the description will be somewhat vague, and the other participants will find it easy to ask for
clarification or to suggest different preferences. Rotate the protagonist role so that each participant gets to play "devil's advocate".

Arrange cards on the table so that classes are physically close to their collaborators. The visual arrangement of the cards can give clues to simple or overly complex relationships. You should not be afraid to tear up cards or to erase, modify, or reorganize operations. Experienced designers will cheerfully admit that they rarely hit upon an optimal division of responsibilities on the first try and that a fair amount of trial and error is necessary even in seemingly simple designs.

You do not necessarily need a group of people for effective class discovery. If you work on your own, though, it helps if you have a "Jekyll and Hyde" personality and can play your own devil's advocate.

CRC cards are a good tool for proposing designs, but they are not particularly suited for documenting them. The better the design discussions, the messier the cards look afterwards. The visual arrangement and movement of the cards are ephemeral. For this reason, the cards should be discarded after a design has been found. They are meant as a discovery tool, not as archival information. We will discuss more permanent documentation tools in the next sections.

In summary, CRC cards are a popular mechanism for discovering classes and operations. Making a new card for each class as the need arises and marking new operations on the cards is easy. Scenarios can be "played out" by moving the cards around while tracing the control flow.

### UML Class Diagrams

Graphical notations are very popular for conveying design information, for a good reason. It is easier to extract relationship information by looking at a diagram than by reading documentation.

To express design information, some convention is required. You may have seen flowcharts that use diamond-shaped symbols for decisions. Of course, there is no logical reason why decisions couldn't be denoted by triangles or circles. The diamond is just the standard choice. For quite some time, there was no similar standard for object-oriented design diagrams. A number of diagramming conventions had been proposed over time that differed greatly in their visual appearance. Finally, three well-known researchers, Booch, Rumbaugh, and Jacobson, got together to unify their disparate notations and developed UML, *the unified modeling language*. We will use UML for all diagrams in this book.

**INTERNET** There are a number of tools available for drawing UML diagrams. The best-known commercial programs are

- Rational Rose (http://www.ibm.com/software/rational/)
- Together (http://www.borland.com/together/)
The commercial programs can be expensive. Freely available programs are
- ArgoUML (http://argouml.tigris.org/) and its commercial cousin Poseidon
  UML Community Edition (http://www.gentleware.com/)
- Dia (http://www.gnome.org/projects/dia; a Windows version is available from
  http://hansbreuer.org/dia/)

For simple UML diagrams, you can use the Violet tool that you can download from the companion Web site for this book or from http://horstmann.com/violet. In Chapter 8, you will learn more about the design of the Violet program.

A class diagram shows classes and the relationships among them.

There are a number of different types of UML diagrams. In this book, we will use class diagrams, sequence diagrams, and state diagrams.

The basic UML notation for class diagrams is fairly simple. Classes are drawn as boxes, which contain the class name and, when appropriate, the names of attributes and methods in additional compartments (see Figure 4). The UML defines an attribute as a named property of a class that describes a range of values that instances of the property may hold. Often, an attribute simply corresponds to an instance field. Occasionally, an attribute is conceptually at a higher level than the actual implementation. You usually do not list all attributes and methods, only the most important ones.

Tip If you have lots of attributes, check whether you can group some of them into classes. For example, if a Student class has attributes name, street, city, state, and zip, then you missed the opportunity of discovering a class Address.

You can also specify the type of an attribute. Unlike in Java, where the type precedes a variable, the UML format is attribute: Type, for example,

```plaintext
text : String
```

Similarly, you can specify the parameter and return types of a method, for example

```plaintext
getMessage(index : int) : Message
```

Figure 4

UML Notation for Classes
Often, the types of attributes, parameters, and return values are omitted to conserve space. Thus, if you see `methodName()`, you cannot automatically assume that the method has no parameters.

Classes are joined by various kinds of connections (see Figure 5). You are already familiar with the first three relationships. We will discuss the others in this section.

You have to be careful about the shapes of the arrow tips. The inheritance arrow is closed, whereas the dependency arrow is open. Also note that the arrow tips for inheritance and dependency are at the end of the dependent class, but the diamond for aggregation is at the end of the aggregating class.

For the “has” relationship, you can also write the multiplicity on the end points of the connection. The most common choices for the multiplicity are:

- any number (zero or more): *
- one or more: 1..*  
- zero or one: 0..1  
- exactly one: 1

For example, Figure 6 denotes that a message queue can hold any number of messages, and a message is in exactly one message queue.
CHAPTER 2  The Object-Oriented Design Process

TIP  Make sure that you use either aggregation or an attribute for a particular feature, but not both. For example, suppose the class Message has a field timestamp of type Date. If you consider the time stamp an attribute, then you should not draw a box and an aggregation connector for the Date class.

TIP  Challenge counts of one. Does a mailbox really only have one greeting? Many real systems have multiple greetings: one for inside callers and one for outside callers, and yet another one for weekends. Once you have a class, you can construct as many objects as you need.

Some designers differentiate between aggregation and composition. Composition is a stronger form of aggregation where the contained objects do not have an existence independent of their container. For example, in the voice mail system, the message queues are permanently contained in the mailboxes—a message queue never exists outside a mailbox. The UML notation for composition is a line with a solid diamond at the end (see Figure 7). In contrast, messages move throughout the mail system and don't always reside in a message queue. Thus, messages are aggregated in message queues, but a message queue is not composed of messages. We will not make that distinction in this book, but you may encounter it elsewhere.

TIP  Use aggregation (or composition) only if a class actively manages objects of another class. For example, does a gas station have cars? Of course it does. Should you therefore draw an aggregation between the class GasStation and the class Car? Not necessarily. If the gas station objects do not need to keep track of the cars, then aggregation is not appropriate.

Some designers do not like the aggregation relationship because they feel it is too implementation-specific. UML defines a more general association between classes. An association is drawn as a solid line without a diamond. You can write roles at the ends of the lines (see Figure 8).
Figure 8
An Association with Roles

Here we model the fact that students register for courses and courses have students as participants. Early in a design, this general relationship makes a lot of sense. As you move closer to implementation, you will want to resolve whether a Course object manages a collection of students, a Student object manages a collection of courses, or both courses and students manage collections of each other.

The relationship between courses and students is bidirectional—Course objects will need to know about the students in the course, and Student objects need to know about the courses for which they are registered. Quite often, an association is directed, that is, it can only be navigated in one way. For example, a message queue needs to be able to locate the messages inside, but a message need not know in which message queue it is. A directed association is drawn with an open arrow tip (see Figure 9). It is easy to confuse that connector with inheritance—you have to pay close attention to the shapes of the arrow tips when drawing UML diagrams.

In Chapter 4, we will introduce the notion of an interface type. An interface type describes a set of methods, but it contains no implementation at all. A class can implement the interface by supplying implementations of its methods. In the UML notation, you denote an interface by adding the stereotype descriptor «interface» above the interface name. (The « and » characters are called guillemets or French quotation marks. They have Unicode values \u00AB = 171 and \u00BB = 187.) If a class implements an interface, you draw a dotted arrow with a closed arrow tip. Figure 10 shows an example.
CHAPTER 2  The Object-Oriented Design Process

Because the javadoc documentation and the class browsers in integrated development environments only display the inheritance relationship, they give the false impression that inheritance is the most important of the relationships between classes. Actually, that is not the case—inheritance is simply easy to determine from the program code. The most important relationship to control is the dependency or "uses" relationship. Too many dependencies make it difficult to evolve a design over time.

**TIP** You should *not* aim for a comprehensive diagram that shows all classes and relationships in the system. A diagram with too much information becomes just a blur. The reason for drawing diagrams is to communicate design decisions. To achieve this purpose, each UML diagram should focus on a particular aspect of the design, and it should be accompanied by text that explains its relevance. When drawing a diagram, you should only include those elements that are needed to make a particular point, and omit all distractions.

### 2.9  Sequence Diagrams

Class diagrams are static—they display the relationships among the classes that exist throughout the lifetime of the system. In contrast, a sequence diagram shows the dynamics of a particular scenario. You use sequence diagrams to describe communication patterns among objects. Figure 11 shows the key elements of a sequence diagram—a method call from one object to another.

Sequence diagrams describe interactions between objects. In UML, you use underline to distinguish object rectangles from class rectangles. The text inside an object rectangle has one of the following three formats:

- `objectName : ClassName (full description)`
- `objectName (class not specified)`
- `: ClassName (object not specified)`

The dashed vertical line that emanates from the object is called the *lifeline*. In some object-oriented programming languages, objects can be explicitly destroyed, which causes their lifeline to end at the point of destruction. However, we will always draw the lifeline so that it goes on indefinitely.

![Figure 11](image_url)

**Figure 11**
A Sequence Diagram
The rectangles along the lifeline are called *activation bars*. They show when the object has control, executing a method. When you call a method, start an activation bar at the end of the call arrow. The activation bar ends when the method returns. (Note that the activation bar of a called method should always be smaller than that of the calling method.)

In the most common form, a sequence diagram illustrates the behavior of a single method. Then the leftmost object has one long activation bar, from which one or more call arrows emanate. For example, the diagram in Figure 11 illustrates the `add` method of the `MessageQueue` class. A message is added to the message queue that holds the new messages. The diagram corresponds to the Java statement

```
newMessages.add( . . . )
```

You cannot tell from the diagram what parameter was passed to the method.

A method can call another method on the same object. Then draw the activation bar of the called method over the one of the calling method, as in Figure 12.

If a method constructs a new object, you can use the stereotype «create» to indicate the timing of the creation. Arrange the object rectangle of the created object as in Figure 13.

When drawing a sequence diagram, you omit a large amount of detail. Generally, you do not indicate branches or loops. (The UML defines a notation for that purpose, but it is a bit cumbersome and rarely used.) The principal purpose of a sequence diagram is to show the objects that are involved in carrying out a particular scenario and the order of the method calls that are executed.
CHAPTER 2 The Object-Oriented Design Process

Sequence diagrams are valuable for documenting complex interactions between objects. These interactions are common in object-oriented programs where any one object tends to have limited responsibilities and requires the collaboration of several other objects. You will see examples in the case study at the end of this chapter.

**TIP** If you played through a use case when using CRC cards, then it is probably a good idea to use a sequence diagram to document that scenario. On the other hand, there is no requirement to use sequence diagrams to document every method call.

State Diagrams

Some objects have a discrete set of states that affect their behavior. For example, a voice mail system is in a "connected" state when a caller first connects to it. After the caller enters an extension number, the system enters the "recording" state where it records whatever the caller speaks. When the caller enters a passcode, the system is in the "mailbox menu" state. The state diagram in Figure 14 shows these states and the transitions between them.

The state has a noticeable impact on the behavior. If the caller speaks while the system is in the "mailbox menu" state, the spoken words are simply ignored. Voice input is recorded only when the system is in the "recording" state.

States are particularly common with objects that interact with the program user. For example, suppose a user wants to retrieve recent voice mail messages. The user must

- Enter the mailbox number.
- Enter the passcode.
- Enter a menu command to start playing messages.

![Figure 14](A State Diagram)
The telephone touchpad has no concept of these steps—it keeps no state. Whenever the user presses a key, that key might be a part of the mailbox number, passcode, or menu command. Some part of the voice mail system must keep track of the current state so that it can process the key correctly. We will discuss this issue further in the case study at the end of this chapter.

### Using javadoc for Design Documentation

You already saw in Chapter 1 how to use the javadoc tool to document classes and methods: Add documentation comments to your source file and run the javadoc tool to generate a set of hyperlinked documents. You can also use javadoc to document your designs. Simply write a skeleton class with no fields and leave all method implementations blank. Of course, supply the class and method comments.

Here is an example:

```java
/**
 * A mailbox contains messages that the mailbox owner can manage.
 */
public class Mailbox {
    /**
     * Adds a message to the end of the new messages.
     * @param aMessage a message
     */
    public void add(Message aMessage) {
    }

    /**
     * Returns the current message.
     * @return the current message
     */
    public Message getCurrentMessage() {
    }
}
```

Do not compile this file—the compiler will complain about unknown types and methods with no return statements. Instead, simply run the javadoc program to extract the HTML documentation. This approach has two benefits. You can post the HTML documentation on the Web and easily share it with your team members. And you can carry the Java files into the implementation phase, with the comments for the key methods already in place.

Professional object-oriented design tools can also produce HTML reports of classes and methods as well as skeleton source code. If you use one of those tools for your design work, then you do not need to use javadoc.
To walk through the basic steps of the object-oriented design process, we will consider the task of writing a program that simulates a telephone voice mail system, similar to the message system that many companies use.

In a voice mail system, a person dials an extension number and, provided the other party does not pick up the telephone, leaves a message. The other party can later retrieve the messages, keep them, or delete them. Real-world systems have a multitude of fancy features: Messages can be forwarded to one or more mailboxes; distribution lists can be defined, retained, and edited; and authorized persons can send broadcast messages to all users.

We will design and implement a program that simulates a voice mail system, without creating a completely realistic working phone system. We will simply represent voice mail by text that is entered through the keyboard. We need to simulate the three distinct input events that occur in a real telephone system: speaking, pushing a button on the telephone touchpad, and hanging up the telephone. We use the following convention for input: An input line consisting of a single character 1 ... 9 or # denotes a pressed button on the telephone touchpad. For example, to dial extension 13, you enter

```
1
3
#
```

An input line consisting of the single letter H denotes hanging up the telephone. Any other text denotes voice input.

The first formal step in the process that leads us toward the final product (the voice mail system) is the analysis phase. Its role is to crisply define the behavior of the system. In this example, we will define the behavior through a set of use cases. Note that the use cases by themselves are not a full specification of a system. The functional specification also needs to define system limitations, performance, and so on.

### Use Cases for the Voice Mail System

#### Reach an Extension

1. The user dials the main number of the voice mail system.
2. The voice mail system speaks a prompt.
   
   Enter mailbox number followed by #.
3. The user types in the extension number of the message recipient.
4. The voice mail system speaks.
   
   You have reached mailbox xxxx. Please leave a message now.
2.12 Case Study: A Voice Mail System

Leave a Message

1. The caller carries out Reach an Extension.
2. The caller speaks the message.
3. The caller hangs up.
4. The voice mail system places the recorded message in the recipient’s mailbox.

Log in

1. The mailbox owner carries out Reach an Extension.
2. The mailbox owner types the passcode, followed by the # key. (The default passcode is the same as the mailbox number. The mailbox owner can change it—see Change the Passcode.)
3. The voice mail system plays the mailbox menu:
   Enter 1 to retrieve your messages.
   Enter 2 to change your passcode.
   Enter 3 to change your greeting.

Retrieve Messages

1. The mailbox owner carries out Log in.
2. The mailbox owner selects the “retrieve your messages” menu option.
3. The voice mail system plays the message menu:
   Enter 1 to listen to the current message.
   Enter 2 to save the current message.
   Enter 3 to delete the current message.
   Enter 4 to return to the mailbox menu.
4. The mailbox owner selects the “listen to the current message” menu option.
5. The voice mail system plays the current new message, or, if there are no new messages, the current old message. Note that the message is played, not removed from the queue.
6. The voice mail system plays the message menu.
7. The user selects “delete the current message”. The message is permanently removed.
8. Continue with Step 3.

Variation #1. Saving a message

1.1. Start at Step 6.
1.2. The user selects “save the current message”. The message is removed from its queue and appended to the queue of old messages.
1.3. Continue with Step 3.
CHAPTER 2  The Object-Oriented Design Process

Change the Greeting

1. The mailbox owner carries out Log in.
2. The mailbox owner selects the "change your greeting" menu option.
3. The mailbox owner speaks the greeting.
4. The mailbox owner presses the # key.
5. The mail system sets the new greeting.

Variation #1. Hang up before confirmation

1.1. Start at Step 3.
1.2. The mailbox owner hangs up the telephone.
1.3. The mail system keeps the old greeting.

Change the Passcode

1. The mailbox owner carries out Log in.
2. The mailbox owner selects the "change your passcode" menu option.
3. The mailbox owner dials the new passcode.
4. The mailbox owner presses the # key.
5. The mail system sets the new passcode.

Variation #1. Hang up before confirmation

1.1. Start at Step 3.
1.2. The mailbox owner hangs up the telephone.
1.3. The mail system keeps the old passcode.

2.12.2 CRC Cards for the Voice Mail System

Let us walk through the process of discovering classes for the voice mail system. Some obvious classes, whose nouns appear in the functional specification, are

- Mailbox
- Message
- MailSystem

Let's start with Mailbox since it is both important and easy to understand. The principal job of the mailbox is to keep messages. The mailbox should keep track of which messages are new and which are saved. New messages may be deposited into the mailbox, and users should be able to retrieve, save, and delete their messages.
The messages need to be kept somewhere. Since we retrieve messages in a first-in, first-out fashion, a queue is an appropriate data structure. Since we need to differentiate between new and saved messages, we'll use two queues, one for the new messages and one for the saved messages. So far, the CRC cards looks like this:

**Mailbox**

<table>
<thead>
<tr>
<th>keep new and saved messages</th>
<th>MessageQueue</th>
</tr>
</thead>
</table>

Where are the mailboxes kept? There needs to be a class that contains them all. We'll call it MailSystem. The responsibility of the mail system is to manage the mailboxes.

**MailSystem**

<table>
<thead>
<tr>
<th>manage mailboxes</th>
<th>Mailbox</th>
</tr>
</thead>
</table>
We can't go much further until we resolve how input and output is processed. Since we have been simulating telephone equipment, let's start with a class `Telephone`. A telephone has two responsibilities: to take user input (button presses, voice input, and hangup actions), and to play voice output on the speaker.

<table>
<thead>
<tr>
<th>Telephone</th>
</tr>
</thead>
<tbody>
<tr>
<td>take user input from touchpad, microphone, hangup</td>
</tr>
<tr>
<td>speak output</td>
</tr>
</tbody>
</table>

When the telephone gets user input, it must communicate it to some object. Could it tell the mail system? Superficially, that sounds like a good idea. But it turns out that there is a problem. In a real voice mail system, it is possible for multiple telephones to be connected to the voice mail system. Each connection needs to keep track of the current state (recording, retrieving messages, and so on). It is possible that one connection is currently recording a message while another is retrieving messages. It seems a tall order for the mail system to keep multiple states, one for each connection. Instead, let's have a separate `Connection` class. A connection communicates with a telephone, carries out the user commands, and keeps track of the state of the session.

<table>
<thead>
<tr>
<th>Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>get input from telephone</td>
</tr>
<tr>
<td>carry out user commands</td>
</tr>
<tr>
<td>keep track of state</td>
</tr>
</tbody>
</table>

Consider reasonable generalizations when designing a system. What features might the next update contain? What features do competing products implement already? Check that these features can be accommodated without radical changes in your design.
2.12 Case Study: A Voice Mail System

For example, to arrive at the design of the voice mail system in this chapter, I considered two reasonable generalizations:

- Can the system be extended to support two telephones?
- Can the system use a graphical user interface instead of a command-line interface?

Now that we have some idea of the components of the system, it is time for a simple scenario walkthrough. Let's start with the Leave a Message use case.

1. The user dials an extension. The Telephone sends the dialed extension number to the Connection. (Add Connection as a collaborator of Telephone. Place the two cards next to each other.)

2. The Connection asks the MailSystem to find the Mailbox object with the given extension number. (This is at least vaguely included in the "manage mailboxes" responsibility. Arrange the MailSystem and Mailbox cards close to the Connection card.)

3. The Connection asks the Mailbox for its greeting. (Add "manage greeting" to the Mailbox responsibilities, and add Mailbox as a collaborator of Connection.)

4. The Connection asks the Telephone to play the greeting on the speaker.

5. The user speaks the message. The Telephone asks the Connection to record it. (Add "record voice input" to the responsibilities of Connection.)

6. The user hangs up. The Telephone notifies the Connection.

7. The Connection constructs a Message object that contains the recorded message. (Add Message as a collaborator of Connection. Make a Message card with a responsibility "manage message contents".)

8. The Connection adds the Message object to the Mailbox.

As a result of this walkthrough, the Telephone, Connection, and Mailbox cards have been updated, and a Message card has been added.
Now let's consider the use case Retrieve Messages. The first steps of the scenario are the same as that of the preceding scenario. Let's start at the point where the user types in the passcode.

1. The user types in the passcode. The Telephone notifies the Connection.
2. The Connection asks the Mailbox to check the passcode. (Add “manage passcode” to the responsibilities of the Mailbox class.)
3. Assuming the passcode was correct, the Connection sets the Mailbox as the current mailbox and asks the Telephone to speak the mailbox menu.

4. The user types in the “retrieve messages” menu option. The Telephone passes it on to the Connection.

5. The Connection asks the Telephone to speak the message menu.

6. The user types in the “listen to current message” option. The Telephone passes it on to the Connection.

7. The Connection gets the first Message from the current Mailbox and sends its contents to the Telephone. (Add “retrieve messages” to the responsibilities of Mailbox.)

8. The Connection asks the Telephone to speak the message menu.

9. The user types in the “save current message” menu option. The Telephone passes it on to the Connection.

10. The Connection tells the Mailbox to save the current message. (Modify the responsibilities of Mailbox to “retrieve, save, delete messages”.)

11. The Connection asks the Telephone to speak the message menu.

That finishes the scenario. As a result, the Mailbox CRC card has been updated.

<table>
<thead>
<tr>
<th>Mailbox</th>
</tr>
</thead>
<tbody>
<tr>
<td>keep new and saved messages</td>
</tr>
<tr>
<td>manage greeting</td>
</tr>
<tr>
<td>manage passcode</td>
</tr>
<tr>
<td>retrieve, save, delete messages</td>
</tr>
</tbody>
</table>

The remaining use cases do not add any new information, so we omit the scenarios here. There are a few points to keep in mind when using CRC cards. It is not easy to reason about objects and scenarios at a high level. It can be extremely difficult to distinguish between operations that are easy to implement and those that sound easy but actually pose significant implementation challenges. The only solution to this problem is lots of practice. Try your best with the CRC cards, and when you run into trouble with the implementation, try again. There is no shame in redesigning the classes until a system actually works. In fact, I redesigned the mail system classes at least twice before arriving at the current design.
Also, don't be deceived by the seemingly logical progression of thoughts in this section. Generally, when using CRC cards, there are quite a few false starts and detours. Describing them in a book would be pretty boring, so the process descriptions that you get in books tend to give you a false impression. One purpose of CRC cards is to fail early, to fail often, and to fail inexpensively. It is a lot cheaper to tear up a bunch of cards than to reorganize a large amount of source code.

2.12.3 — UML Class Diagrams for the Voice Mail System

The "collaboration" parts of the CRC cards show the following dependency relationships:

- Mailbox depends on MessageQueue
- MailSystem depends on Mailbox
- Connection depends on Telephone, MailSystem, Message, and Mailbox
- Telephone depends on Connection

Figure 15 shows these dependencies.
Next, consider the aggregation relationships. From the previous discussion, we know the following:

- A mail system has mailboxes.
- A mailbox has two message queues.
- A message queue has some number of messages.
- A Connection has a current mailbox. It also has references to the MailSystem and Telephone objects that it connects.

There is no inheritance relationship between the classes. Figure 16 shows the completed UML diagram. Note that an aggregation relationship "wins" over a dependency relationship. If a class aggregates another, it clearly uses it, and you don't need to record the latter.

---

**2.12.4 UML Sequence and State Diagrams**

The purpose of a sequence diagram is to understand a complex control flow that involves multiple objects, and to assure oneself at design time that there will be no surprises during the implementation.

In our case, the interactions between the Telephone, Connection, MailSystem, and Mailbox classes are not easy to understand. Let us draw a sequence diagram for the use case Leave a Message (see Figure 17).
The Telephone class reads user input one line at a time and passes it on to the Connection class. Let’s postulate three methods for the Connection class:

- `dial` passes on a button press.
- `record` passes on speech.
- `hangup` tells the connection that the telephone has hung up.

First, the caller keys in the extension number, resulting in several calls to `dial`. We show only one of them—there is no advantage in modeling the repetition.

Once the Connection has the complete mailbox number, it needs to play the greeting. How does it know what greeting to play? It needs to get the mailbox and ask it for the greeting. How does it get the mailbox? It asks the mail system, calling a method that we call `findMailbox`.

The `findMailbox` method returns a `Mailbox` object. You don’t see parameters and return values in the sequence diagram. You have to keep track of the objects yourself and realize that the Mailbox object to the right of the figure is meant to be the object returned by the `findMailbox` call.

Now that the connection has access to the mailbox, it needs the greeting. Thus, it invokes the `getGreeting` method on the mailbox and gets the greeting, which it then
plays on the telephone speaker. Note that the greeting does not show up at all in the sequence diagram since it is entirely passive—no methods are invoked on it.

Next, the telephone reads the message text from the user and passes it on to the connection. Then the telephone reads the hangup signal and calls the hangup method. That is the signal for the connection to construct a message object and to add it to the mailbox.

Which mailbox? The same one that was previously obtained by calling findMailbox. How does the connection remember that mailbox? After all, it had called findMailbox in another method call. This is an indication that the Connection class holds on to the current mailbox.

Figure 18 shows the sequence diagram for the use case Retrieve Messages. It is a good exercise for you to analyze the sequence calls one by one. Ask yourself exactly where the objects of the diagram come from and how the calling methods have access to them.

**Figure 18**
Sequence Diagram for Retrieving a Message
One complexity of the voice mail system is that it is not in control of the input. The user may provide touchpad or spoken input in any order, or simply hang up the phone. The telephone notifies the connection when such an event occurs. For example, notice that the connection is called at least three times in the "Leave a Message" scenario. (As already mentioned, the dial method is called for each separate key. The connection needs to aggregate keys until the user hits the # key. We didn't show that detail in the sequence diagrams.) The connection needs to keep track of the various states so that it can pick up at the right place when it receives the next user input. Figure 19 shows the state diagram.

### 2.12.5 Java Implementation

Now we are ready to implement the system in Java. The files below give the implementation, which at this point is quite straightforward. You should compile and run the program to see the mail system in action. When you run the program, type Q to terminate it. After running the program, have a look at each of the classes. Read the documentation comments and compare them with the CRC cards and the UML class diagrams. Look
again at the UML sequence diagrams and trace the method calls in the actual code. Find
the state transitions of the Connection class.

This simulation has a somewhat unsightly keyboard interface. In Chapter 5, you will see
how to attach a graphical user interface (with buttons for the telephone keys and a text
area to enter simulated voice). That change will require modification of just two classes:
Telephone and MailSystemTester. Because the other classes have been decoupled from
input and output, they require no changes whatsoever. Furthermore, in that program,
you will be able to use two simulated telephones that can interact with the voice mail
system at the same time, just like in a real voice mail system. This is possible because
each connection between a telephone and the voice mail system is managed by a separate
Connection object.

Ch2/mail/Message.java

```java
1 /**
2 * A message left by the caller.
3 */
4 public class Message
5 {
6     /**
7      * Construct a message object.
8      * @param messageText the message text
9     */
10     public Message(String messageText)
11     {
12         text = messageText;
13     }
14 }
15 /**
16     * Get the message text.
17     * @return message text
18     */
19     public String getText()
20     {
21         return text;
22     }
23     private String text;
24 }
```

Ch2/mail/MessageQueue.java

```java
1 import java.util.ArrayList;
2 /**
3 * A first-in, first-out collection of messages. This
4 * implementation is not very efficient. We will consider
5 * a more efficient implementation in Chapter 3.
6 */
7 public class MessageQueue
8 {
```
/**
 * Constructs an empty message queue.
 */
public MessageQueue()
{
    queue = new ArrayList<Message>();
}

/**
 * Remove message at head.
 * @return message that has been removed from the queue
 */
public Message remove()
{
    return queue.remove(0);
}

/**
 * Append message at tail.
 * @param newMessage the message to be appended
 */
public void add(Message newMessage)
{
    queue.add(newMessage);
}

/**
 * Get the total number of messages in the queue.
 * @return the total number of messages in the queue
 */
public int size()
{
    return queue.size();
}

/**
 * Get message at head.
 * @return message that is at the head of the queue, or null
 * if the queue is empty
 */
public Message peek()
{
    if (queue.size() == 0) return null;
    else return queue.get(0);
}

private ArrayList<Message> queue;

Ch2/mail/Mailbox.java
/*
 * A mailbox contains messages that can be listed, kept or discarded.
 */
public class Mailbox
{
Case Study: A Voice Mail System

/**
 * Creates Mailbox object.
 * @param aPasscode passcode number
 * @param aGreeting greeting string
 */

public Mailbox(String aPasscode, String aGreeting)
{
    passcode = aPasscode;
    greeting = aGreeting;
    newMessages = new MessageQueue();
    keptMessages = new MessageQueue();
}

/**
 * Check if the passcode is correct.
 * @param aPasscode a passcode to check
 * @return true if the supplied passcode matches the mailbox passcode
 */

public boolean checkPasscode(String aPasscode)
{
    return aPasscode.equals(passcode);
}

/**
 * Add a message to the mailbox.
 * @param aMessage the message to be added
 */

public void addMessage(Message aMessage)
{
    newMessages.add(aMessage);
}

/**
 * Get the current message.
 * @return the current message
 */

public Message getCurrentMessage()
{
    if (newMessages.size() > 0)
        return newMessages.peek();
    else if (keptMessages.size() > 0)
        return keptMessages.peek();
    else
        return null;
}

/**
 * Remove the current message from the mailbox.
 * @return the message that has just been removed
 */

public Message removeCurrentMessage()
{
    if (newMessages.size() > 0)
        return newMessages.remove();
    else if (keptMessages.size() > 0)
        return keptMessages.remove();
}
CHAPTER 2 The Object-Oriented Design Process

```java
62 else
63    return null;
64 }
65
66 /**<
67  * Save the current message.
68  */
69 public void saveCurrentMessage()
70 {
71    Message m = removeCurrentMessage();
72    if (m != null)
73        keptMessages.add(m);
74 }
75
76 /**<
77  * Change mailbox's greeting.
78  * @param newGreeting the new greeting string
79  */
80 public void setGreeting(String newGreeting)
81 {
82    greeting = newGreeting;
83 }
84
85 /**<
86  * Change mailbox's passcode.
87  * @param newPasscode the new passcode
88  */
89 public void setPasscode(String newPasscode)
90 {
91    passcode = newPasscode;
92 }
93
94 /**<
95  * Get the mailbox's greeting.
96  * @return the greeting
97  */
98 public String getGreeting()
99 {
100    return greeting;
101 }
102 private MessageQueue newMessages;
103 private MessageQueue keptMessages;
104 private String greeting;
105 private String passcode;
106
```

Ch2/mail/Connection.java

```java
1 /**<
2  * Connects a phone to the mail system. The purpose of this
3  * class is to keep track of the state of a connection, because
4  * the phone itself is just a source of individual key presses.
5  */
6 public class Connection
7 {
```
private void resetConnection()
{
    currentRecording = "";
    accumulatedKeys = "";
    state = CONNECTED;
    phone.speak(INITIAL_PROMPT);
}

/**
 * Try to connect the user with the specified mailbox.
 * @param key the phone key pressed by the user
 */
private void connect(String key)
{
    if (key.equals("#"))
    {
        currentMailbox = system.findMailbox(accumulatedKeys);
        if (currentMailbox != null)
        {
            state = RECORDING;
            phone.speak(currentMailbox.getGreeting());
        }
        else
            phone.speak("Incorrect mailbox number. Try again!");
        accumulatedKeys = "";
    }
    else
        accumulatedKeys += key;

/**
 * Try to log in the user.
 * @param key the phone key pressed by the user
 */
private void login(String key)
{
    if (key.equals("#"))
    {
        if (currentMailbox.checkPasscode(accumulatedKeys))
        {
            state = MAILBOX_MENU;
            phone.speak(MAILBOX_MENU_TEXT);
        }
        else
            phone.speak("Incorrect passcode. Try again!");
        accumulatedKeys = "";
    }
    else
        accumulatedKeys += key;

/**
 * Change passcode.
 * @param key the phone key pressed by the user
 */
private void changePasscode(String key)
2.12 Case Study: A Voice Mail System

```java
{  if (key.equals("#"))
    {  currentMailbox.setPasscode(accumulatedKeys);
        state = MAILBOX_MENU;
        phone.speak(MAILBOX_MENU_TEXT);
        accumulatedKeys = "";
    }  
    else
        accumulatedKeys += key;
}
/**
  * Change greeting.
  * @param key the phone key pressed by the user
  */
private void changeGreeting(String key)
{  if (key.equals("#"))
    {  currentMailbox.setGreeting(currentRecording);
        currentRecording = "";
        state = MAILBOX_MENU;
        phone.speak(MAILBOX_MENU_TEXT);
    }  
}
/**
  * Respond to the user's selection from mailbox menu.
  * @param key the phone key pressed by the user
  */
private void mailboxMenu(String key)
{  if (key.equals("1"))
    {  state = MESSAGE_MENU;
        phone.speak(MESSAGE_MENU_TEXT);
    }  
    else if (key.equals("2"))
    {  state = CHANGE_PASSCODE;
        phone.speak("Enter new passcode followed by the # key");
    }  
    else if (key.equals("3"))
    {  state = CHANGE_GREETING;
        phone.speak("Record your greeting, then press the # key");
    }  
}
/**
  * Respond to the user's selection from message menu.
  * @param key the phone key pressed by the user
  */
private void messageMenu(String key)
{  
```
private int state;

private static final int DISCONNECTED = 0;
private static final int CONNECTED = 1;
private static final int RECORDING = 2;
private static final int MAILBOX_MENU = 3;
private static final int MESSAGE_MENU = 4;
private static final int CHANGE_PASSCODE = 5;
private static final int CHANGE_GREETING = 6;

private static final String INITIAL_PROMPT
    = "Enter mailbox number followed by ";
private static final String MAILBOX_MENU_TEXT
    = "Enter 1 to listen to your messages\n" + "Enter 2 to change your passcode\n" + "Enter 3 to change your greeting\n";
private static final String MESSAGE_MENU_TEXT
    = "Enter 1 to listen to the current message\n" + "Enter 2 to save the current message\n" + "Enter 3 to delete the current message\n" + "Enter 4 to return to the main menu\n";
2.12 Case Study: A Voice Mail System

```java
import java.util.ArrayList;

/**
 * A system of voice mailboxes.
 */
public class MailSystem {
    /**
     * Constructs a mail system with a given number of mailboxes.
     * @param mailboxCount the number of mailboxes
     */
    public MailSystem(int mailboxCount) {
        mailboxes = new ArrayList<Mailbox>();
        // Initialize mailboxes.
        for (int i = 0; i < mailboxCount; i++) {
            String passcode = i + 1;
            String greeting = "You have reached mailbox " + (i + 1);
        }
    }
}
```

Ch2/mail/MailSystem.java
```java
62    else
63        return null;
64    }
65
66    /**
67     * Save the current message.
68     */
69    public void saveCurrentMessage()
70    {
71        Message m = removeCurrentMessage();
72        if (m != null)
73            keptMessages.add(m);
74    }
75
76    /**
77     * Change mailbox's greeting.
78     * @param newGreeting the new greeting string
79     */
80    public void setGreeting(String newGreeting)
81    {
82        greeting = newGreeting;
83    }
84
85    /**
86     * Change mailbox's passcode.
87     * @param newPasscode the new passcode
88     */
89    public void setPasscode(String newPasscode)
90    {
91        passcode = newPasscode;
92    }
93
94    /**
95     * Get the mailbox's greeting.
96     * @return the greeting
97     */
98    public String getGreeting()
99    {
100        return greeting;
101    }
102
103    private MessageQueue newMessages;
104    private MessageQueue keptMessages;
105    private String greeting;
106    private String passcode;
107 }

Ch2/mail/Connection.java
1    /**
2     * Connects a phone to the mail system. The purpose of this
3     * class is to keep track of the state of a connection, because
4     * the phone itself is just a source of individual key presses.
5     */
6    public class Connection
7    {
```
2.12 Case Study: A Voice Mail System

```java
/**
 * Construct a Connection object.
 * @param s a MailSystem object
 * @param p a Telephone object
 */
public Connection(MailSystem s, Telephone p) {
    system = s;
    phone = p;
    resetConnection();
}

/**
 * Respond to the user's pressing a key on the phone touchpad.
 * @param key the phone key pressed by the user
 */
public void dial(String key) {
    if (state == CONNECTED)
        connect(key);
    else if (state == RECORDING)
        login(key);
    else if (state == CHANGE_PASSCODE)
        changePasscode(key);
    else if (state == CHANGE_GREETING)
        changeGreeting(key);
    else if (state == MAILBOX_MENU)
        mailboxMenu(key);
    else if (state == MESSAGE_MENU)
        messageMenu(key);
}

/**
 * Record voice.
 * @param voice voice spoken by the user
 */
public void record(String voice) {
    if (state == RECORDING || state == CHANGE_GREETING)
        currentRecording += voice;
}

/**
 * The user hangs up the phone.
 */
public void hangup() {
    if (state == RECORDING)
        currentMailbox.addMessage(new Message(currentRecording));
    resetConnection();
}

/**
 * Reset the connection to the initial state and prompt
 * for mailbox number.
 */
```
CHAPTER 2  The Object-Oriented Design Process

```java
private void resetConnection()
{
    currentRecording = "";
    accumulatedKeys = "";
    state = CONNECTED;
    phone.speak(INITIAL_PROMPT);
}

/**
 * Try to connect the user with the specified mailbox.
 * @param key the phone key pressed by the user
 */
private void connect(String key)
{
    if (key.equals("#"))
    {
        currentMailbox = system.findMailbox(accumulatedKeys);
        if (currentMailbox != null)
        {
            state = RECORDING;
            phone.speak(currentMailbox.getGreeting());
        }
        else
            phone.speak("Incorrect mailbox number. Try again!");
        accumulatedKeys = "";
    }
    else
        accumulatedKeys += key;
}

/**
 * Try to log in the user.
 * @param key the phone key pressed by the user
 */
private void login(String key)
{
    if (key.equals("#"))
    {
        if (currentMailbox.checkPasscode(accumulatedKeys))
        {
            state = MAILBOX_MENU;
            phone.speak(MAILBOX_MENU_TEXT);
        }
        else
            phone.speak("Incorrect passcode. Try again!");
        accumulatedKeys = "";
    }
    else
        accumulatedKeys += key;
}

/**
 * Change passcode.
 * @param key the phone key pressed by the user
 */
private void changePasscode(String key)
```
```java
    { if (key.equals("#"))
        { currentMailbox.setPasscode(accumulatedKeys);
          state = MAILBOX_MENU;
          phone.speak(MAILBOX_MENU_TEXT);
          accumulatedKeys = "";
        }
        else accumulatedKeys += key;
    }

    /**
     * Change greeting.
     * @param key the phone key pressed by the user
     */
    private void changeGreeting(String key)
    {
        if (key.equals("#"))
            { currentMailbox.setGreeting(currentRecording);
              currentRecording = "";
              state = MAILBOX_MENU;
              phone.speak(MAILBOX_MENU_TEXT);
            }
    }

    /**
     * Respond to the user's selection from mailbox menu.
     * @param key the phone key pressed by the user
     */
    private void mailboxMenu(String key)
    {
        if (key.equals("1"))
            { state = MESSAGE_MENU;
              phone.speak(MESSAGE_MENU_TEXT);
            }
        else if (key.equals("2"))
            { state = CHANGE_PASSCODE;
              phone.speak("Enter new passcode followed by the # key");
            }
        else if (key.equals("3"))
            { state = CHANGE_GREETING;
              phone.speak("Record your greeting, then press the # key");
            }
    }

    /**
     * Respond to the user's selection from message menu.
     * @param key the phone key pressed by the user
     */
    private void messageMenu(String key)
    {
```
The Object-Oriented Design Process

```java
if (key.equals("1"))
{
    String output = "";
    Message m = currentMailbox.getCurrentMessage();
    if (m == null) output += "No messages.\n";
    else output += m.getText() + "\n";
    output += MESSAGE-MENU-TEXT;
    phone.speak(output);
}
else if (key.equals("2"))
{
    currentMailbox.saveCurrentMessage();
    phone.speak(MESSAGE-MENU-TEXT);
}
else if (key.equals("3"))
{
    currentMailbox.removeCurrentMessage();
    phone.speak(MESSAGE-MENU-TEXT);
}
else if (key.equals("4"))
{
    state = MAILBOX-MENU;
    phone.speak(MAILBOX-MENU-TEXT);
}

private MailSystem system;
private Mailbox currentMailbox;
private String currentRecording;
private String accumulatedKeys;
private Telephone phone;
private int state;

private static final int DISCONNECTED = 0;
private static final int CONNECTED = 1;
private static final int RECORDING = 2;
private static final int MAILBOX_MENU = 3;
private static final int MESSAGE_MENU = 4;
private static final int CHANGE_PASSCODE = 5;
private static final int CHANGE_GREETING = 6;

private static final String INITIAL_PROMPT
    = "Enter mailbox number followed by ";
private static final String MAILBOX_MENU_TEXT
    = "Enter 1 to listen to your messages\n" + "Enter 2 to change your passcode\n" + "Enter 3 to change your greeting";
private static final String MESSAGE_MENU_TEXT
    = "Enter 1 to listen to the current message\n" + "Enter 2 to save the current message\n" + "Enter 3 to delete the current message\n" + "Enter 4 to return to the main menu";
```
import java.util.ArrayList;

/**
 * A system of voice mailboxes.
 */
public class MailSystem
{
    /**
     * Constructs a mail system with a given number of mailboxes.
     * @param mailboxCount the number of mailboxes
     */
    public MailSystem(int mailboxCount)
    {
        mailboxes = new ArrayList<Mailbox>();
        // Initialize mailboxes.
        for (int i = 0; i < mailboxCount; i++)
        {
            String passcode = "" + (i + 1);
            String greeting = "You have reached mailbox " + (i + 1) + 
                              " Please leave a message now."
                            + ".
            mailboxes.add(new Mailbox(passcode, greeting));
        }
    }

    /**
     * Locate a mailbox.
     * @param ext the extension number
     * @return the mailbox or null if not found
     */
    public Mailbox findMailbox(String ext)
    {
        int i = Integer.parseInt(ext);
        if (1 <= i && i <= mailboxes.size())
            return mailboxes.get(i - 1);
        else return null;
    }

    //private ArrayLIst<Mailbox> mailboxes;
}

import java.util.Scanner;

/**
 * A telephone that takes simulated keystrokes and voice input
 * from the user and simulates spoken text.
 */
public class Telephone
{
The Object-Oriented Design Process

```java
9  /**
10   * Construct phone object.
11   * @param aScanner that reads text from a character-input stream
12   */
13  public Telephone(Scanner aScanner) {
14      scanner = aScanner;
15  }
16
17  /**
18   * Speak a message to System.out.
19   * @param output the text that will be "spoken"
20   */
21  public void speak(String output) {
22      System.out.println(output);
23  }
24
25  /**
26   * Loops reading user input and passes the input to the
27   * Connection object’s methods dial, record, or hangup.
28   * @param c the connection that connects this phone to the
29   * voice mail system
30   */
31  public void run(Connection c) {
32      boolean more = true;
33      while (more) {
34          String input = scanner.nextLine();
35          if (input == null) return;
36          if (input.equalsIgnoreCase("HW"))
37              c.hangup();
38          else if (input.equalsIgnoreCase("Q"))
39              more = false;
40          else if (input.length() == 1
41                    && "1234567890#".indexOf(input) >= 0)
42              c.dial(input);
43          else
44              c.record(input);
45      }
46  }
47
48  private Scanner scanner;
49 }
```

Ch2/mail/MailSystemTester.java

```java
1  import java.util.Scanner;
2
3  /**
4   * This program tests the mail system. A single phone
5   * communicates with the program through System.in/System.out.
6   */
7  public class MailSystemTester
8 {  
```
public static void main(String[] args) {
    MailSystem system = new MailSystem(MAILBOX_COUNT);
    Scanner console = new Scanner(System.in);
    Telephone p = new Telephone(console);
    Connection c = new Connection(system, p);
    p.run(c);
}

private static final int MAILBOX_COUNT = 20;

### Exercises

**Exercise 2.1.** Consider the development of an online course registration system that allows students to add and drop classes at a university. Describe the activities that will take place during the analysis, design, and implementation phases. Give specific examples of activities that relate to the registration system.

**Exercise 2.2.** What is the difference between an object and a class? What is the difference between a class and a type?

**Exercise 2.3.** Consider cars in a car-racing video game. Explain the notions of state, behavior, and identity as they relate to car objects.

**Exercise 2.4.** Download the Mozilla Rhino implementation of ECMAScript. Implement the `Greeter` class and write a program that tests two instances of `Greeter`.

**Exercise 2.5.** Implement a class `Car` in ECMAScript. A car has a fuel efficiency (in miles per gallon or the metric equivalent) and a certain fuel level. Supply methods to add fuel, find out the fuel remaining in the tank, and drive a given distance.

**Exercise 2.6.** List at least eight classes that can be used in an online course registration system that allows students to add and drop classes at a university.

**Exercise 2.7.** Consider the development of a computer system for car rentals. Name one class that might be useful in this context from each of the following categories:

(a) Tangible things
(b) Agents
(c) Events and transactions
(d) Users and roles
(e) Systems
(f) System interfaces and devices
(g) Foundational classes
Exercise 2.8. What relationship is appropriate between the following classes: aggregation, inheritance, or neither?

(a) University-Student
(b) Student-TeachingAssistant
(c) Student-Freshman
(d) Student-Professor
(e) Car-Door
(f) Truck-Vehicle
(g) Traffic-TrafficSign
(h) TrafficSign-Color

Exercise 2.9. Consider an online course registration system that allows students to add and drop classes at a university. Give the multiplicities of the associations between these class pairs.

(a) Student-Course
(b) Course-Section
(c) Section-Instructor
(d) Section-Room

Exercise 2.10. Consider an airline reservation system with classes Passenger, Itinerary, Flight, and Seat. Consider a scenario in which a passenger adds a flight to an itinerary and selects a seat. What responsibilities and collaborators will you record on the CRC cards as a result?

Exercise 2.11. How does the design of the preceding exercise change if you have a group of passengers that fly together?

Exercise 2.12. Consider an online store that enables customers to order items from a catalog and pay for them with a credit card. Draw a UML diagram that shows the relationships between these classes:

Customer
Order
RushOrder
Product
Address
CreditCard

Exercise 2.13. Consider this test program:

```java
public class Tester {
    public static void main(String[] args) {
        String s = "Hello World";
        Scanner in = new Scanner(s);
        while (in.hasNext())
            System.out.println(in.next());
    }
}
```
Draw a sequence diagram that shows the method calls of the main method.

Exercise 2.14. Consider a program that plays TicTacToe with a human user. A class TicTacToeBoard stores the game board. A random number generator is used to choose who begins and to generate random legal moves when it's the computer's turn. When it's the human's turn, the move is read from a Scanner, and the program checks that it is legal. After every move, the program checks whether the game is over. Draw a sequence diagram that shows a scenario in which the game starts, the computer gets the first turn, and the human gets the second turn. Stop the diagram after the second turn.

Exercise 2.15. Look up the API documentation of the URLConnection class and draw a state diagram of the states of an object of this class.

Exercise 2.16. Consider the scenario "A user changes the mailbox passcode" in the voice mail system. Carry out a walkthrough with the mail system's CRC cards. What steps do you list in your walkthrough? What collaborations and responsibilities do you record as a result of the walkthrough?

Exercise 2.17. In our voice mail simulation, the Connection objects hold a reference to the "current mailbox". Explain how you can change the design so that the Connection class does not depend on the Mailbox class. Hint: Add responsibilities to the MailSystem class.

Exercise 2.18. Design and implement a program that simulates a vending machine. Products can be purchased by inserting the correct number of coins into the machine. A user selects a product from a list of available products, adds coins, and either gets the product or gets the coins returned if insufficient money was supplied or if the product is sold out. Products can be restocked and money removed by an operator. Follow the design process that was described in this chapter.

Exercise 2.19. Design and implement a program that manages an appointment calendar. An appointment includes the description, date, starting time, and ending time; for example,

- Dentist 2006/10/1 17:30 18:30
- CS1 class 2006/10/2 08:30 10:00

Supply a user interface to add appointments, remove canceled appointments, and print out a list of appointments for a particular day. Follow the design process that was described in this chapter.

Exercise 2.20. Airline seating. Design and implement a program that assigns seats on an airplane. Assume the airplane has 20 seats in first class (5 rows of 4 seats each, separated by an aisle) and 180 seats in economy class (30 rows of 6 seats each, separated by an aisle). Your program should take three commands: add passengers, show seating, and quit. When passengers are added, ask for the class (first or economy), the number of passengers traveling together (1 or 2 in first class; 1 to 3 in economy), and the seating preference (aisle or window in first class; aisle, center, or window in economy). Then try to find a match and assign the seats. If no match exists, print a message. Follow the design process that was described in this chapter.
EXERCISES

Exercise 2.1. Consider the development of an online course registration system that allows students to add and drop classes at a university. Describe the activities that will take place during the analysis, design, and implementation phases. Give specific examples of activities that relate to the registration system.

Exercise 2.2. What is the difference between an object and a class? What is the difference between a class and a type?

Exercise 2.3. Consider cars in a car-racing video game. Explain the notions of state, behavior, and identity as they relate to car objects.

Exercise 2.4. Download the Mozilla Rhino implementation of ECMAScript. Implement the Greeter class and write a program that tests two instances of Greeter.

Exercise 2.5. Implement a class Car in ECMAScript. A car has a fuel efficiency (in miles per gallon or the metric equivalent) and a certain fuel level. Supply methods to add fuel, find out the fuel remaining in the tank, and drive a given distance.

Exercise 2.6. List at least eight classes that can be used in an online course registration system that allows students to add and drop classes at a university.

Exercise 2.7. Consider the development of a computer system for car rentals. Name one class that might be useful in this context from each of the following categories:

(a) Tangible things
(b) Agents
(c) Events and transactions
(d) Users and roles
(e) Systems
(f) System interfaces and devices
(g) Foundational classes