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 *actors*) 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.

A use case lists a sequence of actions that yields a result that is of value to an actor.

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.

2.7. CRC Cards

A CRC card is an index card that describes a class, its high-level responsibilities, and its collaborators.

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.

The original article describing CRC cards is: Kent Beck and Ward Cunningham, "A Laboratory for Teaching Object-Oriented Thinking", OOPSLA '89 Conference Proceedings October 1–6, 1989, New Orleans, Louisiana. You can find an electronic version at `http://c2.com/doc/oopsla89/paper.html`.

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