Chapter 11 Building Quality Software

Chapter Objectives

After studying this chapter, you should be able to:

- Identify characteristics of quality software, both from the users' and programmers' perspectives
- Follow a development process that promotes quality as you develop your programs
- Avoid common pitfalls in designing object-oriented programs
- Include defensive programming measures to make errors more likely to expose themselves so they can be fixed
- Explain characteristics of quality user interfaces and describe an iterative methodology for developing them

Suppose your rich uncle offered you a choice of two automobiles. One is known for its high performance, luxurious leather interior, and precision workmanship. The other is underpowered, needs frequent repair, and will probably have visible body rust before it's five years old. Both are free, no strings attached. Which would you take?

Most of us have a highly developed sense of quality. It's sometimes hard to define exactly what quality is, but given a choice, we prefer the higher quality option.

This chapter begins by describing what to look for in high-quality software. The rest of the chapter describes how to improve the quality of the software we write.

11.1 Defining Quality Software

The dictionary defines quality as "the degree of excellence of a thing." In the automobile example in the introduction, quality was described in terms of performance, interior finishing, workmanship, reliability, and susceptibility to rust. In an article of clothing, the relevant characteristics might include the strength, wrinkle resistance, and color fastness of the fabric, as well as the perceived style of the garment.

Sometimes we can take measurements that correspond very well to quality. For example, an automobile's top speed or time to accelerate from 0 to 60 miles per hour are easily measured and indicate the car's performance pretty well. Similarly, the number of threads per inch is easily measured and often corresponds to the quality of a garment's fabric. The quality of other characteristics such as an automobile's reliability or the style of a garment is more subjective and must be measured less directly. For example, one could survey car owners for the number of repairs they have required over the last 5 years or survey shoppers on their reactions to a garment.

In the remainder of this section we'll examine characteristics that are important to "the degree of excellence" or quality of software.

11.1.1 Quality from a User's Perspective

The user of a program judges its quality based on many characteristics. Three of the most important, however, are correctness, usability, and reliability.

A program is **correct** if it meets its specifications. A payroll program that ignores overtime or withholds too much tax would not be correct. A Web browser that only displays the first five paragraphs of a Web page cannot be considered correct. However, a program can be missing your favorite feature and still be correct. The key is whether it meets its specification—whether it does what it is supposed to do—correctly. It's a fact of life, however, that all but the simplest programs will be less than 100% correct.

The **usability** of a program is determined by the effort required to learn, operate, prepare input, and interpret output when compared to the alternatives. A more usable program is a higher quality program. But usability is subjective and must take the user into account. A beginning digital photographer may want a very simple program to crop photos and touch up "red-eye." The "ease-of-use" of this program would be nothing but frustration to a professional photographer that also wants to manipulate the color balance or merge parts of one image into another.

A third characteristic of quality programs is **reliability**. A program is reliable if it does not crash, does not lose or corrupt data, and is consistent in how it works. Obviously, an unreliable program is not a quality program.

11.1.2 Quality from a Programmer's Perspective

It may seem strange, at first, to consider quality from any other perspective than the user. However, when buying a new car, your mechanic may tell you to avoid certain models because they are difficult to repair. From the mechanic's perspective, the models differ in quality. New home buyers often have the home inspected by someone trained to look beneath the paint. The inspector determines whether the foundation is sound or whether shortcuts were taken in insulating the windows. From the inspector's perspective, two homes that look the same may have different levels of quality.

Similarly, two programs may look exactly the same to the user but have radically different quality levels to programmers. From their perspective, a quality program makes their life easier because it is understandable, testable, and maintainable.

Understandable Programs

A program is understandable if it is easy to determine how the program works. Choosing descriptive variable and method names is one of the easiest ways to increase the understandability of a program. Appropriate comments describing each class and method, as well as explanatory comments for more difficult code, also play a large role in the understandability of a program.

Understandability is important because most programs are read by many people over many years. A program in an insurance company may be written and debugged by a team of programmers over several months. Understandability helps the team work together. Several months later, another programmer may read the code to correct bugs. Five years later, the program may need extensive modifications to accommodate a new product offered by the company and two years after that it may again need modification to accommodate a change to the business practice of the company. In each of these situations, the programmer's life is improved by working with understandable code.

Most professional programs last much longer than those written by students learning to program. A typical student program is written in a few hours or at most a few weeks. After it is handed in, someone reads it, assigns a grade, and the program's useful life is over. With such short lifespans and so few people involved, program understandability has less importance—although it never hurts to ensure that the person assigning the grade can understand what you've written!

Understandability becomes more important as the size of the program grows. A typical student program may contain between twenty and several hundred lines of code. At this scale, programmers can keep most of the important details for the entire program in mind or can easily remind themselves if they forget. Not so for commercial programs that may involve between several thousand and several million lines of code. In such programs, understandability is vital to successfully writing or fixing the program.

Testable Programs

Programmers need to test their code. It's a simple fact of a programmer's life. Designing software that is easy to test makes this part of programming easier and more enjoyable, and improves the program's quality from the programmer's perspective.

It is also likely to improve the program's quality from the user's perspective. Recall that correctness is a major factor. A testable program is more likely to be tested, and is therefore more likely to be correct.

A testable program has classes with a single, well-defined purpose. Methods have a single purpose and few dependencies on other parts of the code. A testable program has an infrastructure for testing, allowing it to be easily tested whenever it is modified using the techniques discussed in Section 7.1.

Maintainable Programs

As noted earlier, a typical program is often changed over its lifetime to accommodate new requirements. A high-quality program is maintainable if it is easy to find and correct bugs, easy to adapt to new requirements, and easy to improve its overall quality in a process called **refactoring**. Refactoring modifies the code to improve its quality but doesn't actually change what the program does.

The maintainability of a program is strongly influenced by how understandable and testable it is. Carefully designed programs are more maintainable. Later in this chapter we will investigate a number of design guidelines such as writing appropriately parameterized methods, avoiding duplicated code, and using private instance variables.

11.2 Using a Development Process

How do we build high-quality software? It doesn't just happen! We need some discipline and a **development process**—a set of steps that help us know what to do next.

Many development processes have been proposed over the years. The one we will use is illustrated in Figure 11-1. It combines the best of the older, plan-first development processes with the newer, object-oriented and agile development processes.

Program development begins with defining the requirements. Upon completing each stage, development proceeds to the next stage as shown by the heavier arrows. At each stage, perhaps with the exception of implementing scenarios, lots of interaction with users should be expected.

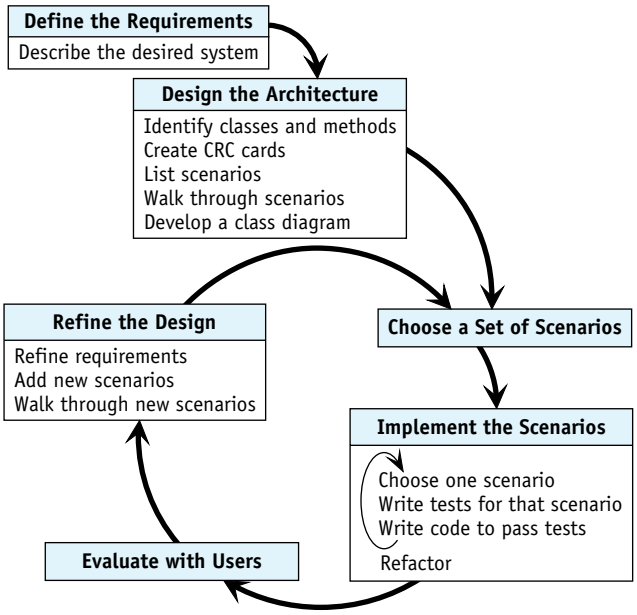
KEY IDEA

User involvement is vital to a successful project.

In the following sections we'll examine each of the major stages shown in Figure 11-1 and illustrate the process with a case study.

(figure 11-1)

Software development process



11.2.1 Defining Requirements

The first stage of the development process is defining the **requirements**, also known as the **specification**. The requirements are a written statement of what the program is supposed to do. Depending on the complexity of the problem, it could be several paragraphs or many pages.

Requirements often start with a single person's idea, but are usually developed by talking with people who are expected to use the program. Questions might include:

- How do you currently do the job (without the program)?
- What are the good parts and the bad parts of your current approach?
- What capabilities would you like to have that you don't have now?

Eventually, the answers to these questions are written down. They might result in a document that contains information similar to the paragraphs shown in Figure 11-2. We will use it as a running example for the remainder of this section.

> The video store system is used to rent videos to the store's customers.
>
> The system has a list of videos and a list of customers. Each video has a unique identifying number, title, and genre. The system must be able to add new customers and videos to these lists, as well as remove inactive customers and videos no longer in circulation.
>
> Customers are charged a rental fee of $.99 for videos released more than one year ago and $2.99 for new releases, plus any accumulated late fees, when they rent a video. Customers are assessed late fees if a video is returned past its due date. Customers have a rental history to help resolve late fee disputes.

(figure 11-2)

Requirements for a video store system

Requirements are seldom complete. This is a fact of life that programmers must deal with, rather than the way it should be. Incomplete requirements mean that programmers typically need to go back to the users with questions about what the system should do. Some of the issues these requirements should address, but don't, include the following:

- Can the store have multiple copies of the same video?
- Can a customer rent more than one video at one time?
- Are there additional fees such as taxes?
- Will other pricing strategies be offered? For example, three videos for three nights at a reduced rate?
- What kind of reports are required?
- What are the possible genres?
- Does this program run on a single computer or many? (Running on many computers with coordinated data is much beyond the scope of this book.)

For this example, we will assume the store can have multiple copies of a video and that customers can rent more than one video at a time. We will ignore the other issues for now.

11.2.2 Designing the Architecture

KEY IDEA

Defining the architecture includes some of the most important and far-reaching decisions of the project.

The second stage of the software development process is designing the program's architecture. The **architecture** refers to how the most important classes interact with each other. Crucial decisions here have consequences throughout the life of the program, so it's important to get it right. Designing the architecture consists of five tasks, as shown in Figure 11-1. The five tasks are as follows:

- Identify the most important classes and methods for the program using an analysis of the nouns and verbs in the requirements.
- Summarize the responsibilities and collaborators for each class on index cards.
- List scenarios in which the software will be used.
- Walk through the scenarios using the index cards to further develop the responsibilities and collaborators.

- Develop a class diagram based on the responsibilities and collaborators listed on the index cards.

These five tasks are elaborated in the sections that follow.

Identify Classes and Methods

In Section 8.3 we introduced an object-based design strategy, reproduced in Figure 11-3. The methodology uses nouns and noun phrases in the requirements to help identify objects and classes, and verbs and verb phrases to help identify methods. Recall that a noun is a word indicating a person, place, thing, or idea, while a verb is a word expressing an action or a state of being.

(figure 11-3)

Object-based design methodology (reproduced from Figure 8-13)

1. Read the description of what the program is supposed to do, highlighting the nouns and noun phrases. These are the objects your program must declare. If there are any objects that cannot be directly represented using existing types, define classes to represent such objects.

2. Highlight the verbs and verb phrases in the description. These are the services. If a service is not predefined:
   a. Define a method to perform the service.
   b. Store it in the class responsible for providing the service.

3. Apply the services from Step 2 to the objects from Step 1 in a way that solves the problem.

Applying this methodology is easier if the specification is rewritten using simpler sentences. All sentences have a **subject** and a **predicate**. The subject is a noun or noun phrase and provides the answer of who or what did the action. The predicate contains a verb and explains the action or condition of the subject.

The rewritten specification should use a verb in the active voice. Such a sentence has a subject that does something or is in a state of being. The alternative is a passive voice where the subject receives the action. "Customers are charged a rental fee of $.99 …" is passive. "Customers pay a rental fee of $.99 …" is active.

The rewritten specification should also remove connecting words like "and" in the predicates, wherever possible. This will introduce some verbal redundancy. For example, "The system has a list of videos and a list of customers" will turn into two sentences: "The system has a list of videos" and "The system has a list of customers." Such rewriting must be done carefully to ensure that the meaning is unchanged.

The result of rewriting the requirements in Figure 11-2 is shown in Figure 11-4. The verb is underlined in each case.

The system rents videos to customers.
The system has a list of videos.
The system has a list of customers.
Each video has a unique identifying number.
Each video has a title.
Each video has a genre.
The system adds new customers (to its list of customers).
The system adds new videos (to its list of videos).
The system removes inactive customers (from its list of customers).
The system removes videos (from its list of videos).
Customers pay a rental fee of $.99 for releases more than one year old.
Customers pay a rental fee of $2.99 for new releases.
Customers pay a late fee after a video has been returned late.
Customers have a rental history to help resolve late fee disputes.

(figure 11-4)

Requirements rewritten using the active voice and simpler sentences

With the requirements in this form, we can more easily use the nouns and verbs to identify classes and methods, as suggested by the methodology in Figure 11-3. The following guidelines are relevant:

- Nouns in the sentence's subject are almost always relevant classes.
- Some nouns will represent instance variables in a class. Examples from Figure 11-4 include "unique identifying number" and "title." Nouns that can be represented using existing types such as `int` and `String` or occur with a verb such as *has* are particularly likely to be instance variables rather than new classes.
- Look at nouns in the predicate as well. For example, "rental history" looks like it might be a class we need to write.
- Name classes with singular nouns. If we need many videos or customers, we will construct many `Video` or `Customer` objects.
- Don't let adjectives fool you. They describe or modify a noun, but rarely represent a new class. For example, we do not need one class for customers and another class for inactive customers ("inactive" is the adjective).
- Sometimes synonyms or abbreviations are given for the same thing—for example, "video store system" and "system." Choose just one name to represent all of these different ways of saying the same thing.
- Class names are important. Take enough time to find just the right words to describe the objects they represent. In this case, `Video`, `Customer`, and `RentalHistory` are fairly obvious choices. "The system" needs more work. `VideoStore` is one reasonable choice to encompass the whole "system."

The predicates in the rewritten requirements represent the **responsibilities** of each class. Responsibilities come in two flavors: information the class must know or derive, and actions the class must be able to carry out. The first kind of responsibility is often represented by possessive verbs such as have, has, or keeps. The second kind is often represented by active verbs such as add, remove, rent, or charge.

Create CRC Cards

KEY IDEA

CRC cards list a class's responsibilities and collaborators.

CRC cards are a handy way to record the classes and responsibilities identified in the previous step. **CRC** stands for Classes, Responsibilities, and Collaborators. The cards are usually made from 4 x 6 inch index cards and are divided into three areas, as shown in Figure 11-5.

(figure 11-5)

Sample CRC card

VideoStore	
rent videos to customers	video
has a list of videos	customer
has a list of customers	
add and remove videos	
add and remove customers	
charge customers for videos	
assess late fees	

The top area of the CRC card contains the name of the class. Below the class name, the responsibilities are listed on the left side. The right side contains a list of the **collaborators**. The collaborators section is a recognition that classes usually do not act alone. They collaborate with other classes to do their work. For example, to rent a video, the `VideoStore` class will likely need to work with instances of the `Video` and `Customer` classes. Hence, both classes are listed to the right. A collaborator is only listed once even though it may be involved with several responsibilities.

It's important to use real index cards rather than making these lists on a computer because in one of the following steps we will distribute the CRC cards to people in a group. The size is also important. These cards are meant to represent an overview of the class. If we can't express it in the space on one card, we are using too much detail. That's why the closely related actions of adding and removing customers are compressed into a single responsibility in Figure 11-5.

These CRC cards represent the classes likely to play the most important roles within the program. As such, they form the foundation of the program's architecture.

However, we now set the CRC cards aside while we develop scenarios. The scenarios will eventually be used with the cards to further develop their responsibilities and collaborations.

Develop Scenarios

A **scenario** is a specific task that a user might want to do with the program. Scenarios are also known as **use cases**. We will use scenarios to simulate the program to gain a better understanding of what classes are needed, the services they need to support, and how the classes interact with each other.

Scenarios for the video store system might include:

- The program is started.
- A new customer is added.
- An existing customer rents a video.
- An existing customer rents three videos.
- A new customer rents a video.
- A video is returned on time.
- An overdue video is returned.
- A customer returns an overdue video and wants to pay the late fee without renting another video.
- A customer would like a list of all the drama videos released in the last two years that he or she has not already rented.
- A report is prepared of all customers with videos more than one week overdue.
- A video has been lost and must be removed from the system.

A complex system could have hundreds or even thousands of scenarios. List as many as you can think of.

Walk Through Several Scenarios

CRC cards and scenarios are brought together in a **walk-through**. A walk-through proceeds through one scenario in an orderly fashion to develop the tasks that each class must perform. A walk-through works best with a group of people in which each person is assigned one of the CRC cards. That person gives voice to the class's responsibilities as the group walks through the scenario. If a group of people is not available, the process can be simulated by just one person.

A walk-through is an active process that is best illustrated by examining a transcript. The "speaking parts" are people holding their assigned CRC card. We'll identify them in the

transcript by the classes named on their cards. The first scenario is an existing customer, Ashley Wong, renting the video *Star Wars: A New Hope*.

`VideoStore:` "Well, I already have a responsibility to 'rent videos to customers,' so I guess I'll start this one. I'd better find Ashley in my list of customers."

KEY IDEA

Responsibilities are often added during a walk-through.

`Customer:` "That sounds like a new responsibility. You'd better write it down on your card." (`VideoStore` adds ***find customer*** to her card.)

`VideoStore:` "I have a list of customer objects. `Customer`, what kind of information do I need to find one of you in the list?"

KEY IDEA

Defining responsibilities often identifies required instance variables.

`Customer:` "I don't have any responsibilities related to that. I guess I could have a name or an ID number. I'll write that down on my card." (`Customer` adds ***has info like name and ID*** to his card.) "I could also have a responsibility to determine if a given name or ID matches me." (`Customer` adds ***determine if given name or ID matches me***.)

`VideoStore:` "So I need to have a way to get a customer name or an ID from the user. I can also see that eventually I'm going to need information to identify a video, too. But I've already got a long list of responsibilities from the noun/verb analysis!"

`Video:` "What if we had a user interface (`UI`) class to collect that kind of information? That would off-load some of that responsibility from you."

`VideoStore:` "Great idea!"

KEY IDEA

New CRC cards are often added during a walk-through.

`VideoStore` adds `UI` as a collaborator. Someone makes a new CRC card for `UI` and adds the responsibilities ***accept customer ID or name*** and ***accept video ID*** as responsibilities. `VideoStore` is added as a collaborator.

`VideoStore:` "OK. So now I can get a customer number from the `UI` and collaborate with `Customer` to find the specific customer. I can do the same with `Video` to find the video. So `Video`, you already have the responsibility to ***have info like ID, title, and genre*** from our noun/verb analysis. You'd better do like `Customer` did and add ***determine if given ID matches me***. I'll also add the responsibility ***find a video***.

"Now I've got a `Video` and a `Customer` and I'm supposed to rent one to the other. I'd really like to give that responsibility to someone else. `Video`, can you rent yourself to the `Customer`?"

`Video:`	"I suppose so. But I recall that the `Customer` class has responsibilities for having a rental history and paying rental fees. I think it would make more sense for him to have that responsibility."
`Customer:`	"Yeah, I can do that. I'll add *rent video* as a responsibility. To rent a video, I need to find out if the video is a new release or not. `Video`, can you answer that question for me?"
`Video:`	"You really want that information so you know how much to charge, right? I think it would be better if you just asked me that question directly rather than whether I'm a new release."
`Customer:`	"You're right!"

`Video` adds *get rental fee* to its responsibilities.

`Customer:`	I also need to add the video to the rental history. So I'll add the responsibility *add video to rental history* and add `RentalHistory` as a collaborator.
`RentalHistory:`	"That makes me think that I'm just a list of some other kind of object. I think "rental history" is really just an `ArrayList` in the `Customer` class. I propose renaming myself to just `Rental`. Then I would have the responsibility of having information about one rental—mainly the video, the date rented, and the due date."

The `RentalHistory` card is thrown away and a new one named `Rental` is created. The responsibility to *have info like video, date rented, date due* is added. `RentalHistory` collaborates with `Video`. The collaborator on the `Customer` card is changed from `RentalHistory` to just `Rental`.

We'll stop our transcript here. To finish this scenario the participants need to decide how to charge the customer for the rental and perhaps for accumulated late charges.

After finishing this scenario, the group should walk through additional scenarios. It's good to do several radically different scenarios early to verify that the evolving architecture can handle them. The architecture often changes quite a bit while walking through the first several scenarios, but then settles down to a stable design. Eventually scenarios will not result in new CRC cards or collaborations and will produce only a few new responsibilities. At that point, the walk-through process can stop.

KEY IDEA

Walk through several scenarios.

A walk-through blurs the distinction between objects and classes. In practice, it doesn't matter. The same person can represent all the instances of one class and generally doesn't need to keep track of them as distinct objects.

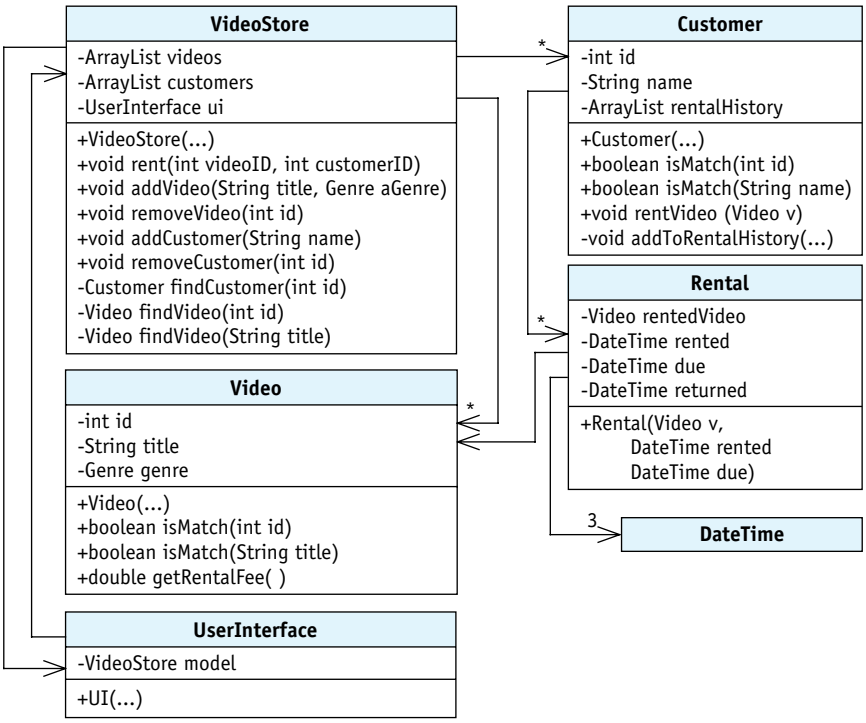
Beginners often skip walking through scenarios, perhaps because they feel embarrassed by the role playing this process demands. Don't! A walk-through is a *very* effective way to understand how the system will work and to come to a common agreement on the design.

Develop a Class Diagram

The final step in designing the architecture is to develop a class diagram from the CRC cards. "Knowing" responsibilities often turn into instance variables. The other responsibilities turn into methods. Collaborations turn into associations between classes. The CRC cards developed so far for the video store yield the class diagram shown in Figure 11-6.

(figure 11-6)

Class diagram for the video store system



11.2.3 Iterative Development

The main body of the development methodology, shown in Figure 11-1, consists of four repeated steps called the **development cycle**:

- Choose a set of scenarios to implement.
- Implement those scenarios.
- Evaluate the resulting system with users.
- Refine the design based on user evaluation, perhaps by adding new requirements or scenarios or revising the existing ones.

In a large project, the development team will repeat this development cycle many times. With each iteration, they have a program that is closer to the finished product, and with adequate feedback from users, refined into a product that actually meets their needs.

Choose a Set of Scenarios

The development cycle begins with choosing a subset of the scenarios to implement. The choice should be made with the users. Which scenarios will they find most useful? Which scenarios are required for the most basic functionality? For example, we need to add videos and customers before we can rent videos. So we might choose adding videos and customers as the first scenarios to implement.

These scenarios form the basis for the remainder of the cycle.

Implement the Chosen Scenarios

The implementation phase is when the code is actually written. As illustrated by the development methodology diagram in Figure 11-1, implementation itself is iterative within the larger iterative process.

Each implementation iteration begins with choosing one scenario—for example, adding a video. This scenario is represented in the class diagram by the `addVideo` method.

Next, write tests to determine if the scenario is implemented correctly. Recall that a test involves five steps:

- Decide which method you want to test. In this case, `addVideo` will be our primary concern.
- Set up a known situation. For example, a brand new `VideoStore` object that has zero videos.
- Determine the expected results of executing the chosen method.
- Execute the code you want to test.
- Verify the results. For example, verify that calling `addVideo` causes the `VideoStore` object to have one more video than before. Verifying that the video can also be retrieved is another good test.

LOOKING BACK

Testing was first discussed in Section 7.1.

Testing a command such as `addVideo` also requires queries to determine the current state of the `VideoStore` class.

These suggestions result in tests such as those shown in Listing 11-1.

Listing 11-1: *Tests for the* `VideoStore` *class*

```
1  public class VideoStore extends Object
2  {  // Methods omitted.
3
4    public static void main(String[] args)
5    {
6     System.out.println("Testing adding a video...");
7     VideoStore vs = new VideoStore();
8     Test.ckEquals("no videos", 0, vs.getNumVideos());
9     vs.addVideo("Star Wars: A New Hope", Genre.SCI_FI);
10    Test.ckEquals("one video", 1, vs.getNumVideos());
11
12    // test finding by name
13    Video v1 = vs.findVideo("Star Wars: A New Hope");
14    Test.ckEquals("found video", true,
15                  v1.isMatch("Star Wars: A New Hope"));
16
17    // test finding by id
18    Video v2 = vs.findVideo(v1.getID());
19    Test.ckEquals("found video", true,
20                  v2.isMatch("Star Wars: A New Hope"));
21
22    // test not found condition
23    Video v3 = vs.findVideo("Gone with the Wind");
24    Test.ckIsNull("not found", v3);
25   }
26 }
```

Another set of tests should be written in the `Video` class. Among them, tests for `isMatch` and tests to verify that each instance of `Video` is assigned a unique identification number.

KEY IDEA

Some programmers use the mantra "Test a little, code a little; test a little, code a little...."

After writing the tests, implement the methods required so that the tests pass. This means actually compiling the program (and fixing the compile-time errors) and running it (and fixing any bugs the tests expose).

When the scenario passes its tests, choose another scenario and repeat the process. Keep choosing new scenarios, writing tests, and writing code until all the scenarios for this development cycle are implemented.

Finally, reexamine the program in light of the new code that has been added. There may be areas that are overly complex or where code is duplicated. Take the time to simplify it (**refactor**) before moving on.

Evaluate with Users

After the scenarios for this development cycle have been implemented, evaluate the resulting program with users. Does it do what they expect? Are there ways to improve how they interact with the program? Does it spark new ideas for how the program can be used to do their jobs more efficiently?

Users' needs change over time. In fact, introducing the program itself changes users and their needs. It could be that what they thought was needed when the project began is very different now—and the project must adapt to that new reality.

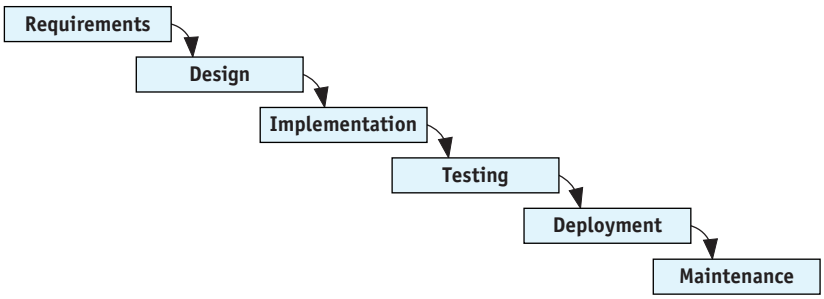
Refine the Design

The evaluation step may require refining the design. Perhaps the requirements themselves need to be refined. Perhaps a key scenario needs to be changed with a new walkthrough, or it is realized that some scenarios won't actually be needed, or (more likely) that some scenarios need to be added.

After refining the design, repeat the development cycle again, beginning with choosing a new set of scenarios to develop.

Advantages of the Iterative Approach

The iterative approach to implementation has a number of advantages, especially compared to an older approach known as the **waterfall model**. It was named the waterfall model because the results of each stage of development fell into the next, much like a series of waterfalls. The waterfall model is illustrated in Figure 11-7.



(figure 11-7)

Waterfall model

The iterative approach offers at least six specific advantages (in no particular order):

- In the iterative approach, bugs are produced, identified, and fixed in small groups. In the waterfall model, many bugs are produced all at once during the implementation phase, and then must be found and fixed all at once during

the testing phase. This is considerably harder because one bug can interfere with finding and fixing another.

- The program is always close to working with the iterative approach. If a deadline is looming (you need to hand in the assignment or your customer wants to see how the program is coming along), you have all the completed scenarios to show. Using the waterfall model, it may be that a lot of work has been done but nothing can actually be demonstrated because the debugging is incomplete.
- The iterative approach does a better job of maintaining programmer morale. Each small victory of seeing another test pass boosts morale, but a long debugging process in the waterfall model saps morale.
- Choosing one scenario to implement and writing tests for it gives programmers many specific goals to focus their programming efforts. It also provides an objective means to determine when the goals have been met.
- As the program changes over time, the tests generated in the iterative approach are useful for verifying that everything that used to work still works.
- Frequent user evaluation helps keep the project on track with the real, changing needs of the users.

11.3 Designing Classes and Methods

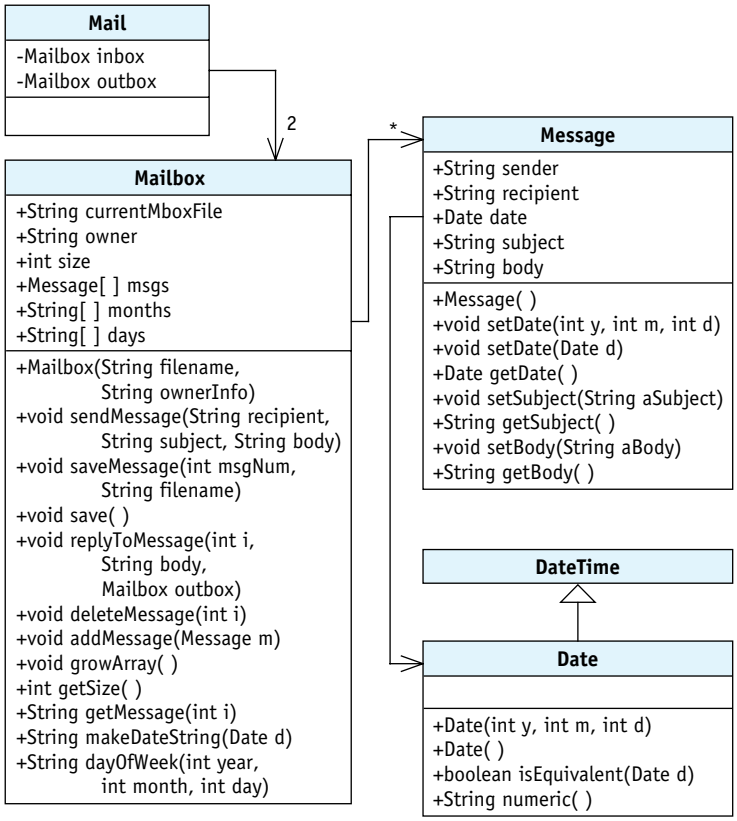
In Section 11.2 we discussed a development process for discovering and implementing classes. In this section we will explore some concrete rules or heuristics for classes and methods that increase the understandability, testability, and maintainability of a program. We will look at a pair of classes that illustrate a number of *very* poor design decisions. We will examine the consequences of those decisions and what better decisions could have been made.

The program we will examine is the beginnings of an e-mail client such as *Thunderbird*, *Eudora*, or *Outlook*. The program doesn't have any code to actually send or receive e-mail, but the core classes to store messages and manipulate them in a mailbox exist. A provided graphical user interface allows new messages to be created, and existing messages to be saved to a file, replied to, or deleted.

Our examination will concentrate on just two classes: `Mailbox` and `Message`. The `Message` class models a single message that has been either sent or received. It has instance variables to store the sender, recipient, date, subject, and the body of the message. The `Mailbox` class stores and manipulates a number of `Message` objects. It has an instance variable, `msgs`, that is a partially filled array of `Message` objects. It also has methods such as `sendMessage`, `replyToMessage`, and `deleteMessage`.

The program uses two instances of `Mailbox`, one for the "in box" (received messages) and another for the "out box" (sent messages).

The code for these two classes is shown in Listings 11-2 and 11-3. A class diagram for the program is shown in Figure 11-8. You should take some time to examine these classes and try to understand what they do and how they work. Remember, the code has many poor design decisions. This is *not* code to emulate! Many of these poor choices are shown in annotations within the listing.



(figure 11-8)

Class diagram for the e-mail program

Listing 11-2: *A very poor implementation of the* `Mailbox` *class for an e-mail program*

```
1  import java.util.Scanner;
2  import java.io.*;
3  import becker.util.*;
4
5  /** A mailbox holds messages for an e-mail program.
6  *
7  *  @author Jack Rehder, Byron Weber Becker */
8  public class Mailbox extends Object
```

FIND THE CODE

ch11/email/

Listing 11-2: *A very poor implementation of the* `Mailbox` *class for an e-mail program* (continued)

```
9  {
10   public String currentMboxFile = "";
11   public String owner; // who's mailbox is this?
12
13   public int size = 0; // number of messages
14   public Message[] msgs = new Message[5];
15
16   public static final String[] months = {"Jan", "Feb", "Mar",
17     "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
18   public static final String[] days = {"Sunday", "Monday",
19     "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"};
20
21   public Mailbox(String filename, String ownerInfo)
22                                     throws FileNotFoundException
23   { super();
24     this.owner = ownerInfo;
25
26     this.currentMboxFile = filename;
27     Scanner in = new Scanner(new File(filename));
28
29     while (in.hasNextLine())
30     { Message msg = new Message();   // start a new message
31       in.next();                     // skip From: tag
32       msg.sender = in.nextLine().trim();
33       in.next();                     // skip To: tag
34       msg.recipient = in.nextLine().trim();
35       in.next();                     // skip Date: tag
36       msg.setDate(in.nextInt(),
37                   in.nextInt(), in.nextInt());
38       in.next();                     // skip Subject: tag
39       msg.setSubject(in.nextLine().trim());
40
41       String body = "";
42       while (true)
43       { String line = in.nextLine();
44         if (line.equals("EOM"))
45         { break;
46         }
47         body = body + line + "\n";
48       }
49       msg.setBody(body);
50       this.addMessage(msg);
51     }
```

public instance variables

Instance variables that have little to do with the class's core purpose.

Many set methods leave data in `Message` objects unprotected.

Code to read a message belongs in the `Message` class.

Nested loops are hard to understand.

Listing 11-2: *A very poor implementation of the* `Mailbox` *class for an e-mail program* (continued)

```
52      in.close();
53    }
54
55    public void sendMessage(String recipient,
56                            String subject, String body)
57    { Message m = new Message();
58      m.recipient = recipient;
59      m.setSubject(subject);
60      m.date = new Date();
61      m.setBody(body);
62      m.sender = this.owner;
63
64      this.addMessage(m);
65
66      PrintWriter out = this.openOutputFile("outbox.txt");
67      out.println("From:" + m.sender + "\nTo:" + m.recipient
68          + "\nDate:" + m.getDate().numeric() + "\nSubject:"
69          + m.getSubject() + "\n" + m.getBody());
70      out.close();
71    }
72
73    public void saveMessage(int msgNum, String filename)
74    { PrintWriter out = this.openOutputFile(filename);
75      out.println("From:" + this.msgs[msgNum].sender + "\nTo:"
76          + this.msgs[msgNum].recipient + "\nDate:"
77          + this.msgs[msgNum].getDate().numeric() + "\nSubject:"
78          + this.msgs[msgNum].getSubject() + "\n"
79          + this.msgs[msgNum].getBody());
80      out.close();
81    }
82
83    public void save()
84    { PrintWriter out =
85                  this.openOutputFile(this.currentMboxFile);
86
87      for (int i = 0; i < this.size; i++)
88      { Message m = this.msgs[i];
89        out.print("From:" + m.sender + "\nTo:" + m.recipient
90            + "\nDate:" + m.getDate().numeric() + "\nSubject:"
91            + m.getSubject() + "\n" + m.getBody() + "EOM\n");
92      }
93      out.close();
94    }
```

Constructor apparently does nothing.

Public instance variables leave data unprotected.

This code is repeated five times (look for the other four times).

Listing 11-2: *A very poor implementation of the* `Mailbox` *class for an e-mail program* (continued)

```
95
96    public void replyToMessage(int i, String body,
97                               Mailbox outBox)
98    { Message reply = new Message();
99
100     reply.setDate(new Date());
101     String sender = this.msgs[i].recipient;
102     reply.sender = sender;
103     String recipient = this.msgs[i].sender;
104     reply.recipient = recipient;
105     String subject = this.msgs[i].getSubject();
106     reply.setSubject("Re:" + subject);
107     Date d = this.msgs[i].getDate();
108
109     String who = this.msgs[i].sender;
110     String BEGIN = "**On " + d.numeric() + ","
111         + who.substring(0, who.indexOf('<')).trim()
112         + " wrote:\n";
113     String END = "**end original message**\n\n";
114
115     String origMsg = this.msgs[i].getBody();
116     String replyBody = BEGIN + origMsg + END + body;
117     reply.setBody(replyBody);
118
119     PrintWriter out = this.openOutputFile("outbox.txt");
120     out.println("From:" + reply.sender + "\nTo:"
121                         + reply.recipient
122         + "\nDate:" + reply.getDate().numeric() + "\nSubject:"
123         + reply.getSubject() + "\n" + reply.getBody());
124     out.close();
125     outBox.addMessage(reply);
126   }
127
128   public void deleteMessage(int n)
129   { for (int i = n; i < this.size - 1; i++)
130     { this.msgs[i] = this.msgs[i + 1];
131     }
132     this.size--;
133   }
134
135   public void addMessage(Message m)
136   { if (this.size == this.msgs.length)
```

This method is too long and complex. Much of the code belongs elsewhere.

More repeated code.

No documentation.

Listing 11-2: *A very poor implementation of the* `Mailbox` *class for an e-mail program* (continued)

```
137     { this.growArray();
138     }
139     this.msgs[this.size] = m;
140     this.size++;
141   }
142
143   public void growArray()
144   { Message[] temp = new Message[this.msgs.length * 2];
145     for (int i = 0; i < this.msgs.length; i++)
146     { temp[i] = this.msgs[i];
147     }
148     this.msgs = temp;
149   }
150
151   public int getSize()
152   { return this.size;
153   }
154
155   public String getMessage(int i)
156   { return "From:" + this.msgs[i].sender
157                    + "\nTo:" + this.msgs[i].recipient
158                    + "\nDate:" + this.msgs[i].getDate().numeric()
159                    + "\nSubject:" + this.msgs[i].getSubject() + "\n"
160                    + this.msgs[i].getBody();
161   }
162
163   public String makeDateString(Date d)
164   { if (d.isEquivalent(new Date()))
165     { return "Today";
166     }
167
168     int month = d.getMonth();
169     int year = d.getYear();
170     int day = d.getDay();
171
172     String wkDay = this.dayOfWeek(year, month, day);
173     return wkDay + "," + months[month-1] + "" + day + "" + year;
174   }
175
176   public String dayOfWeek(int year, int month, int day)
177   { int a = (int) Math.floor((14 - month) / 12);
178     int y = year - a;
179     int m = month + 12 * a - 2;
```

Many methods that should be `private` are `public`.

This kind of processing belongs in the `Message` class.

These methods have little to do with the core purpose of the class and should be elsewhere.

Listing 11-2: *A very poor implementation of the* `Mailbox` *class for an e-mail program* (continued)

```
180     int d = (day + y + (int)Math.floor(y / 4)
181         - (int)Math.floor(y/100) + (int)Math.floor(y/400)
182         + (int) Math.floor((31 * m)/12)) % 7;
183
184     return this.days[d];
185   }
186
187   public PrintWriter openOutputFile(String filename)
188   { try
189     { return new PrintWriter(filename);
190     }
191     catch (Exception ex)
192     { ex.printStackTrace();
193       System.exit(1);
194     }
195     return null;
196   }
197 }
```

No attempt to handle the error.

FIND THE CODE

ch11/email/

Listing 11-3: *A very poor implementation of the* `Message` *class for an e-mail program*

```
1  /** Store one message in the mail program.
2  *
3  *  @author Jack Rehder; Byron Weber Becker */
4  public class Message extends Object
5  {
6    public String sender;
7    public String recipient;
8    public Date date;
9    public String subject;
10   public String body = "";
11
12   public Message()
13   {}
14
15   public void setDate(int y, int m, int d)
16   { this.date = new Date(y, m, d);
17   }
18
19   public void setDate(Date d)
```

`public` instance variables are open to abuse.

Constructor doesn't initialize instance variables.

Set methods required to overcome inadequacies in the constructor.

This class should provide services for its clients other than simply storing information.

Listing 11-3: *A very poor implementation of the* `Message` *class for an e-mail program* (continued)

```
20    { this.date = d;
21    }
22
23    public Date getDate()
24    { return this.date;
25    }
26
27    public void setSubject(String aSubject)
28    { this.subject = aSubject;
29    }
30
31    public String getSubject()
32    { return this.subject;
33    }
34
35    public void setBody(String aBody)
36    { this.body = aBody;
37    }
38
39    public String getBody()
40    { return this.body;
41    }
42 }
```

11.3.1 Rules of Thumb for Writing Quality Code

In this section we will discuss a number of rules of thumb for writing quality code. In Section 11.3.2 we will put them into a larger framework.

Document Classes and Methods

You probably noticed that the `Mailbox` and `Message` classes have almost no documentation. If you read the code for comprehension, you probably wished that it had more documentation to help you understand it.

Documentation, or the lack of it, obviously affects how easily a class or method can be understood. That, in turn, affects how easily the code can be maintained and, to some extent, tested.

KEY IDEA

Document what the method should do, then write the code.

Writing documentation is one of a programmer's more dreaded jobs, but it definitely pays dividends later when someone needs to read and understand the code. An acceptable practice is to write the documentation and code in parallel, while both are still fresh in your mind. Don't write more code until the code you have is documented. However, many people agree that the best practice is to write the documentation *before* writing the code. Writing the documentation first can often help clarify in your mind what the code is supposed to do, making the code easier to write and more likely to be correct.

Avoid Nested Loops

The `Mailbox` constructor contains a loop within a loop. Such structures are difficult for programmers to understand. Simplify them by putting the inner loop into an appropriately named helper method. Naming the task helps you clarify the responsibilities of both the outer loop and the inner loop.

Keep Methods Short

Research published by psychologist George A. Miller in 1956 shows that people can only store and process a limited amount of information in short term, or working, memory. The limit is often given as seven chunks of information, plus or minus two.

This research supports writing short methods. Methods are more likely to be written and understood correctly if they have fewer chunks of information. The most appropriate way to write such methods is using stepwise refinement, as discussed in Chapter 3.

The `Mailbox` class has at least two methods that are too long: the constructor and `replyToMessage`. Instead of a constructor that is 32 lines long, consider one that uses a helper method to read one message:

```
public Mailbox(String filename, String ownerInfo)
                throws FileNotFoundException
{ super();
  this.owner = ownerInfo;

  this.currentMboxFile = filename;
  Scanner in = new Scanner(new File(filename));

  while (in.hasNextLine())
  { Message msg = this.readOneMessage(in);
    this.addMessage(msg);
  }
  in.close();
}
```

In this version, all the details of reading a message, lines 30–49 in Listing 11-2, are collapsed into `readOneMessage`. That method might be further refined using two more helper methods, `readHeader` and `readBody`.

Using helper methods provides at least three benefits. First, each helper method gives the code it contains a name. A well-chosen name helps identify what the code does, making the helper method easier to understand. Second, the name summarizes the code, making the client that calls it easier to read and understand. Third, helper methods make testing easier. The code in each helper method can often be tested separately, which is easier than testing the same code written as a single method.

Make Helper Methods Private

Methods that do not need to be called from outside their containing class should be declared `private`. For example, there is no reason to call `readOneMessage` from outside the `Mailbox` class. The same is true for `growArray` and probably for `makeDateString` and `dayOfWeek`.

Making these methods `private` offers two advantages. First, it prevents programmers from using them inappropriately, either maliciously or by mistake. For example, a programmer may think it is his responsibility to call `growArray` before adding a message to the mailbox. Doing so could slow the program dramatically and could waste a huge amount of memory. Taking active steps to prevent such problems can make the program as a whole more bug free.

Second, when a bug does occur, appropriate use of accessibility helps track it down by elimination. If a problem occurs within a private method, we know with absolute certainty that it was called from within the same class. With non-private methods, we might be surprised to find that it was called from some place completely unexpected.

Put Duplicated Code in a Helper Method

The code in Listing 11-2 has five places where essentially the same code is repeated. Take a moment to find them. The code itself isn't exactly the same, but the results are.

The repeated code assembles an e-mail message into a string to be saved or displayed in the user interface and occurs in lines 67–69, 75–79, 89–91, 120–123, and 156–160. This is undesirable for a number of reasons:

- It makes the class longer, with more code for programmers to read and understand.
- If a change to the way messages are represented is required, there are five copies of the code that need to change.

- If a bug is found, parallel changes must be made in five places. It will be easy to overlook at least one of them—maybe all except for one!
- All of the repeated code must be tested. That's a silly waste of time and energy if the code is essentially the same.

The solution is to make a helper method that can be called from many places in the program rather than duplicating the code itself. In fact, it may appear that `getMessage` is precisely the helper method we need:

```
155   public String getMessage(int i)
156   { return "From:" + this.msgs[i].sender
157                    + "\nTo:" + this.msgs[i].recipient
158                    + "\nDate:" + this.msgs[i].getDate().numeric()
159                    + "\nSubject:" + this.msgs[i].getSubject()
160                    + "\n" + this.msgs[i].getBody();
161   }
```

This method can be used to replace code in `saveMessage` (lines 75–79) and `save` (lines 89–91), but is much more difficult for `sendMessage` (lines 67–69) and `replyToMessage` (lines 120–123). Passing an index as the argument works for the first two methods, but the last two work with `Message` objects that are not yet in an array and thus can't be accessed with an index.

LOOKING AHEAD

Later, we consider the question "Which class should contain the helper method?" It will lead to an even better solution.

One solution is writing a helper method, `formatMessage`, with a `Message` object as a parameter. Then `getMessage`, above, can be written as follows:

```
155   public String getMessage(int i)
156   { return this.formatMessage(this.msgs[i]);
161   }
```

and lines 89–91 can be rewritten as follows:

```
89      out.print(this.formatMessage(m) + "EOM\n");
```

In addition to reducing wasted time and energy, putting duplicate code into a method gives you an added abstraction to work with. If you've already used the same code several times, chances are good that it represents a higher-level idea, or abstraction, within your code. Putting it in a method increases the chances that you'll recognize when it is appropriate to use it, and makes using it trivial—just call the method.

Make Instance Variables Private

`Mailbox` and `Message` both have many public instance variables. This is unfortunate for several reasons. First, it makes the classes more difficult to change. For example, the `Message` class uses `String`s to store the sender and receiver. They both have two

distinct parts, the real name and the e-mail address in angle brackets, as in the following example:

```
Byron Weber Becker <bwbecker@email.com>
```

It may be decided later that it would be better to define a class named `Contact` for this information. Instances could store the real name in one field and the e-mail address in another. Other information might be added such as nicknames, telephone numbers, or mailing addresses.

If the instance variables were private, making this change would be straightforward. We would have to find all the places inside the `Message` class that accessed either `sender` or `recipient` and change them to use the new `Contact` class. However, if the instance variables are public our search must expand beyond the containing class to include the entire program. In fact, it is possible for many programs to use a given class—and any of them might need changing if they used the instance variable instead of an appropriate method.

A second reason to keep instance variables private is to prevent misuse, either accidentally or maliciously. For example, a programmer writing the user interface may need to know the number of messages in an instance of `Mailbox`. He might write `this.mBox.msgs.length`, failing to realize that `msgs` is a partially filled array and that the number of messages does not correspond to the space in the array. Or perhaps you wrote the `Mailbox` class and the programmer working on the user interface has either a grudge against you or a really wacky sense of humor and adds the following in an obscure part of the user interface code:

```
if (Math.random() < 0.01)
{ this.mBox.size--;
}
```

The effect of this insertion is that one message is lost from the mailbox 1% of the times the above code executes. It seems like a bug in the `Mailbox` class, but who would think to look in a completely unrelated part of the user interface? You could spend a lot of time tracking down this "bug" and face a lot of pressure from management while you're doing it! The best policy is to simply avoid the issue by making instance variables private.

Write Powerful Constructors

The previous advice to make instance variables private is largely circumvented if instance variables have `set` methods. Public `get` and `set` methods allow a client to change a private instance variable just as if it were public.

KEY IDEA

A constructor should return an object that is ready to use.

In the `Message` class, the only reason to have `set` methods is to give the instance variables their initial values—the constructor's job. By the time the constructor has finished executing, the object should be ready to use. All the instance variables should have meaningful initial values. Because all of the values are available at once in a well-written constructor, more stringent checking for inconsistent data can be performed.

In the case of the `Message` class, two constructors might be appropriate. The first would be used in the `Mailbox` constructor to read one message from a file. It would take a single argument, an open `Scanner` object. The second could be used by the `replyToMessage` and `sendMessage` methods to construct an object from the constituent pieces.

Powerful constructors make the `Message` class more understandable because it eliminates the need for many `set` methods. Many bugs arise from variables being given incorrect values. By minimizing the places where new values are given, bugs are made less likely and those that do exist are easier to find.

Powerful constructors also eliminate the problem of incompletely initialized objects. With an appropriate constructor it is impossible for a programmer to overlook setting an instance variable. This might occur, for example, when a new instance variable is added to the class. In the current program all the places where an instance is constructed must be found and modified. This is a much easier maintenance task if all the initialization is done in the constructors.

Keep Data and Processing Together

The largest change, and the biggest benefit, to the `Mailbox` and `Message` classes comes from keeping data and processing together. The major clues that these classes don't keep them together are:

- `Message` is a "container" class. It contains information, but doesn't have any methods that actually process that information.
- `Mailbox` gets many individual pieces of data from instances of `Message` and then processes them in some way.

KEY IDEA

The class with the data should do the processing.

A better approach is to have the class with the data do the processing. For example, the `formatMessage` helper method we described earlier really belongs in the `Message` class, not the `Mailbox` class. By moving it, you can avoid passing an argument and use the instance variables directly instead of using the `get` methods.

A more dramatic example comes from moving much of `replyToMessage` to the `Message` class. Because much of the information for a reply comes from the original message, it makes sense to keep the data and processing together by asking the message to construct the reply. Using this approach, the `replyToMessage` method in `Mailbox` could become only three lines of code, as shown in Listing 11-4.

Listing 11-4: *The* `replyToMessage` *method, written assuming a* `constructReply` *method exists in the* `Message` *class*

```
/** Reply to the given message with the given body.
 * @param i          the index of the message to reply to
 * @param body       the body of the reply
 * @param outBox     the mailbox instance collecting sent mail. */
public void replyToMessage(int i, String body,
                           Mailbox outBox)
{ Message reply = this.msgs[i].constructReply(body);
  outBox.addMessage(reply);
  reply.save(Mailbox.SEND_BOX);
}
```

This code also strongly hints that a message should save itself to a file rather than expecting the `Mailbox` class to do it. After all, the message is where the data that requires saving is located.

It should be obvious that understanding and maintaining `replyToMessage` has become much easier with this change. It is true that programmers may need to find three additional methods (`constructReply`, `addMessage`, and `save`) and understand them. On the other hand, each of these methods has a clear focus that is clearly indicated by its name. That may be enough to allow the programmer to avoid needing to understand their details. If the programmer does need to understand them, the fact that they are focused on a single task and have a clear name will make understanding them much easier.

Write Immutable Classes

A class is **mutable** if its instances can be changed after they are constructed. A class is **immutable** if instances can not be changed after construction. An example of an immutable class is `String`. There are no methods to change a string once it has been created, only methods that create a new string that is similar to the old one in some way. For example, the `name.toUppercase()` method does *not* change the string `name`; it creates a new string like `name` except that lowercase characters are converted to uppercase.

Immutable classes are simple. Their objects have only a single state—the state in which they were created. If the state is correct when it's created, it will be correct for all time without any further work by you or the programmers using the class. References to immutable classes can be shared freely because there is no way to change the object's state. Mutable classes, on the other hand, can have objects in a wide variety of states

that change over time. That makes them harder to use and understand than immutable classes.

Classes that represent a value, like `Date` or `String`, should almost always be made immutable. All other classes should be made as immutable as possible. It obviously isn't possible to make `Mailbox` immutable because messages need to be added and deleted from it. But `Message` can be made immutable. Once created, there is no reason to change a message.

There are a few simple rules to make a class immutable:

- Don't provide methods that modify the object.
- Make all the instance variables private. This prevents anyone from changing them directly.
- Make sure that you don't allow aliases to mutable components. For example, instances of `Message` have references to a `Date` object. Because a `getDate` method is provided, the designer of `Message` must ensure that either `Date` is immutable or that `getDate` makes a copy of the date object to return. Another option is to provide queries such as `getYear` to answer questions about the mutable component. It's also important the other way. If a client passes a reference to a mutable object, make a copy of that object before storing it in an instance variable.

LOOKING BACK

This issue is discussed in more detail beginning in Section 8.2.

If you want to really go the extra mile for your immutable classes, you need to follow two more rules:

- Use the `final` keyword for instance variables. For example, `private final String subject` in the `Message` class emphasizes that `subject`'s value should not change after the first assignment. This is enforced by the compiler.
- Use the `final` keyword for the class as a whole. For example, `public final class Message` prevents someone from overriding the methods in the `Message` class and changing their behaviors.

The `DateTime` class is mutable, even though it represents a value and should be immutable according to the criteria given earlier. Suppose you wanted an immutable `Date` class. Extending `DateTime` doesn't work because methods like `addDays` would still be available via inheritance. But `DateTime` has a lot of functionality that would be good to reuse.

The solution is for the `Date` class to have an instance of `DateTime` as a private instance variable. It can then provide exactly the methods it requires and omit the problematic ones. For example, see Listing 11-5. Of course, nothing prevents you from adding new methods to the class as well.

Listing 11-5: *Making an immutable class using a mutable class*

```
1  public final class Date extends Object
2  {
3    private final DateTime myDate;
4
5    public Date(int year, int month, int day)
6    { super();
7      this.myDate = new DateTime(year, month, day);
8    }
9
10   // For internal use only. Assumes that the caller does NOT keep a reference to d.
11   private Date(DateTime d)
12   { super();
13     this.myDate = d;
14   }
15
16   public int getYear()  {  return this.myDate.getYear();  }
17   public int getMonth() {  return this.myDate.getMonth(); }
18
19   // Return a new date, adding the given number of days.
20   public Date addDays(int howMany)
21   { DateTime copy = new DateTime(this.myDate);
22     copy.addDays(howMany);
23     return new Date(copy);
24   }
25   // etc.
26 }
```

Delegate Work to Helper Classes

A class should represent a single abstraction. The `Message` class should model an e-mail message and the `Mailbox` class should be focused only on managing a group of messages. Mixing in peripheral concepts makes a class harder to understand, test, and maintain. However, in Listing 11-2, the `Mailbox` class does just that.

The problem is that, in addition to its core responsibility of managing messages, the `Mailbox` class also has responsibilities for formatting dates. The major clues are instance variables storing the names of days and months along with methods to turn a date into a formatted string and calculate the day of the week.

Particularly in a program where a `Date` class already exists, these instance variables and methods should be moved there. In general, a class should delegate work to "helper

classes" using the "has-a" or containment relationship discussed in Section 8.1.3 so that each class remains focused on a single idea.

Limiting each class to one cohesive set of responsibilities makes the class easier to understand. It's easier to test one set of responsibilities than to test two or more that are intertwined in the same class. Finally, if changes need to be made, it's easier to figure out where and actually make the changes in well-focused classes.

11.3.2 Managing Complexity

Many of these coding heuristics relate to four features that have been long recognized as crucial to quality code. These four features have stood the test of time, across many different programming languages and development methodologies. They seem to be invariant. Defined in terms of Java, they are:

- **Encapsulation**—Grouping data and related services into a class.
- **Cohesion**—The extent to which each class models a single, well-defined abstraction and each method implements a single, well-defined task.
- **Information hiding**—Hiding and protecting the details of a class's operation from others.
- **Coupling**—The extent to which interactions and dependencies between classes are minimized.

KEY IDEA

Managing complexity is one of the hardest parts of writing software.

Each of these relates to a fundamental problem in constructing software: managing complexity. Most programs have a huge number of details, each affecting the overall correctness of the system. Managing them so that a change to one detail does not create a problem with another is difficult!

Let's define **detail**, for the moment, as either a method or an instance variable, and define **interaction** as a method calling another method or accessing an instance variable.

We can get an idea of the complexity of a program by making a diagram with circles representing details and lines representing interactions. For example, consider a class with one instance variable and three methods. Two methods access the instance variable and one method calls the third as a helper method. The complexity diagram would appear as shown in Figure 11-9. The instance variable is shown with crossed lines within it.

(figure 11-9)

Complexity diagram for a very simple class

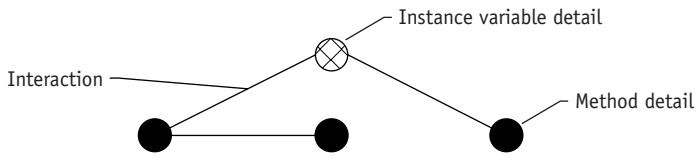
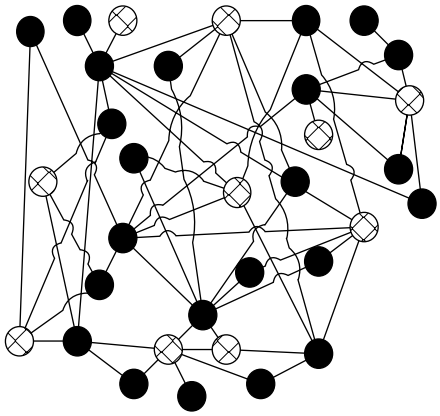


Figure 11-10 shows a diagram for a considerably more complex program. We will use it to show how the ideas of encapsulation, cohesion, information hiding, and coupling can help us manage complexity.



(figure 11-10)

Unconstrained interactions between details

The best solution to date for managing complexity is to impose voluntary constraints on how we write programs so that some interactions can't happen, and to organize the other interactions so that they are easier to think about. Encapsulation, cohesion, information hiding, and coupling all have a role to play in managing complexity through voluntary constraints.

KEY IDEA

Java allows us to write poorly designed programs. As programmers, we must choose to write excellent programs.

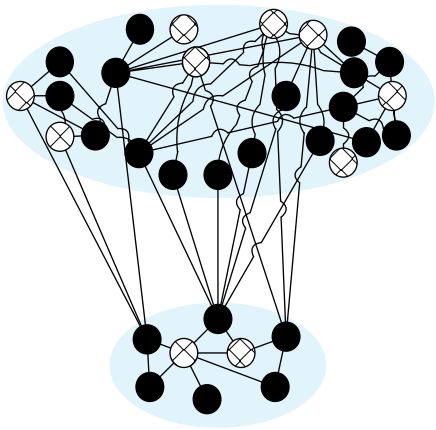
Encapsulation

Think of a class as a capsule—something that encapsulates or encloses a number of related instance variables and methods (details). By putting them into the same capsule, we clearly indicate that they belong together.

Encapsulation is illustrated in Figure 11-11. It's the same as Figure 11-10 except that details have been grouped and encapsulated using ovals. Encapsulation makes it easier to see how interactions are organized because now they fall into two groups.

(figure 11-11)

Classes that are encapsulated, but do meet the ideal for cohesion, information hiding, or coupling



KEY IDEA

Encapsulation divides interactions into those within a class and those between classes.

The first group are interactions between details within the same class. Of all the possible interactions in our program, encapsulation focuses our attention on a group that works together using closely related details. These details, and the interactions between them, are the primary concern of the programmer or small group of programmers responsible for implementing and maintaining the class.

The second group of interactions are the primary concern of programmers using the class—the interactions between details in different classes. Of all the possible interactions within a program, encapsulation helps these programmers focus on the most relevant ones. By grouping interactions inside classes and between classes, we help manage complexity.

Java's class mechanism provides a natural way to group details. Unfortunately, Java does not force them to be grouped—that requires good design decisions on our part. The heuristics noted earlier that support encapsulation include the following:

- Keep Data and Processing Together
- Delegate Work to Helper Classes

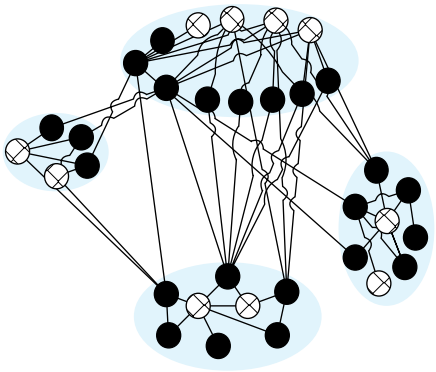
Cohesion

Cohesion emphasizes the "relatedness" of the details that we encapsulate. We should not encapsulate just any details that happen to interact. We should make sure those details represent a cohesive whole. In concrete terms, a class should model a single, well-defined entity. A method should have one, and only one, task.

In the mail program presented earlier in this chapter, one could imagine the `Message` class containing instance variables and methods related to both dates and contacts (senders and receivers). This would represent "low cohesion." The ideal is "high cohesion" in which details related to dates are split into their own class, as are the details related to contacts. This kind of change is illustrated in Figure 11-12.

KEY IDEA

In a highly cohesive program, each class is focused on one abstraction.



(figure 11-12)

Encapsulated, cohesive classes that do not yet meet the ideal for information hiding and coupling

Cohesive classes and methods make it easier for us to understand them. It's easier to focus on just one abstraction or one task than to understand two or more that are mixed together.

Heuristics related to cohesion include:

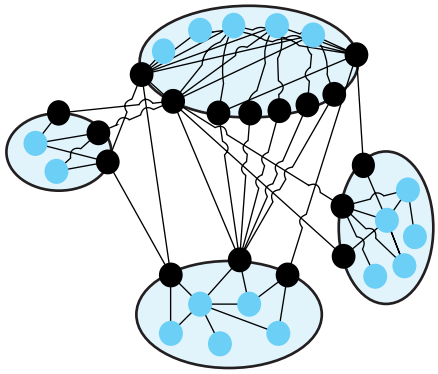
- Delegate Work to Helper Classes
- Keep Methods Short
- Put Duplicated Code in a Helper Method

Information Hiding

Encapsulation groups details of a program together in a class so that a programmer can focus primarily on them. Information hiding says which of those details are important to programmers using the class and hides the rest behind the capsule wall, as shown in Figure 11-13. Note the dark wall around each oval and that public details have moved to straddle the capsule wall.

(figure 11-13)

Information hiding emphasizes some details and hides the rest from view



Information hiding distinguishes *what* a class can do from *how* it does it. The parts that are left exposed (declared `public`) are always methods. Their names and documentation indicate what can be done with the class. The details of the instance variables required and the helper methods used are all hidden inside the class.

KEY IDEA

Information hiding removes many possible interactions from a programmer's consideration.

For programmers who want to use a class, information hiding eliminates many possible interactions from their consideration and helps manage the complexity.

Another advantage, as noted before, is that hiding details allows us to change how the class operates without affecting the code that uses the class. As long as the public methods continue to behave as before, the details of just *how* they work can change to accommodate better approaches.

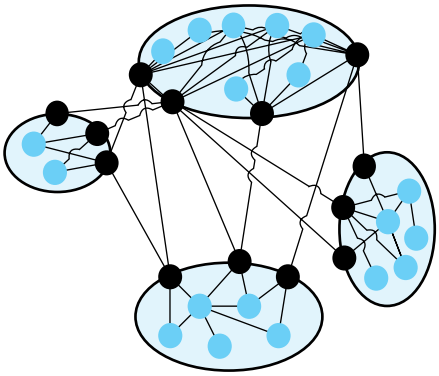
Information hiding also allows us to limit our testing to only the public parts. If they are tested thoroughly, we can be more relaxed about testing internal details.

Heuristics that are related to information hiding include:

- Make Instance Variables Private
- Make Helper Methods Private
- Write Powerful Constructors
- Keep Data and Processing Together

Coupling

Ideally, we would like to be able to understand or change one class with only minimal knowledge or changes of other classes. When this is true, we say the classes are "weakly coupled." See Figure 11-14.



(figure 11-14)

Weakly coupled classes have few dependencies on each other

Information hiding already reduces the coupling between classes by forcing the classes to interact only through public methods. We can go one step further, however, and ask ourselves whether we have the right public methods.

For example, the `Message` class in Listing 11-3 has public instance variables but no methods that really do anything, resulting in strong coupling. The `Mailbox` class is forced to interact often with the `Message` class in order to do its work.

Simply hiding the instance variables and making public accessor methods would not improve the coupling, however. The `Mailbox` class would still need many interactions with the `Message` class. The coupling between these two classes improved dramatically, however, when we wrote higher-level methods like `constructReply` and `save`, substantially reducing the dependency on accessor methods like `getSubject`.

KEY IDEA

Weakly coupled classes use powerful methods to minimize dependencies between classes.

Heuristics that affect coupling include:

- Keep Data and Processing Together
- Make Instance Variables Private
- Write Powerful Constructors
- Write Immutable Classes

Working together, the heuristics in Section 11.3.1 and the overall goals of encapsulation, high cohesion, information hiding, and weak coupling yield classes and programs that are easier to understand, easier to test, and easier to change. All of these are attributes contributing to quality code, from a programmer's perspective.

11.4 Programming Defensively

The quality of software generally increases with the disciplined use of a development methodology. Quality also increases dramatically through good design. But even with these efforts, users may still enter erroneous input, a necessary file may still be accidentally deleted, or a program bug may still bring the program to an abrupt halt. Quality software will proactively attempt to detect and, if possible, handle such errors. This section discusses important techniques for doing so, including exceptions, design by contract, and assertions.

11.4.1 Exceptions

We learned about exceptions in Section 8.4. We learned that exceptions are thrown when an exceptional circumstance arises. The exception interrupts the program's normal flow of control and if nothing is done to intervene, the program will stop with an error message displayed on the console. Programmers can intervene in this process with the `try-catch` statement. Statements that may throw an exception are placed in the `try` clause. A series of `catch` clauses after it can include code to handle exceptions that are thrown.

Using exceptions effectively is an important part of writing quality software for a number of reasons. First, exceptions provide a uniform approach to reporting and handling errors. Languages that do not support exceptions force programmers to adopt ad hoc methods for reporting errors using an instance variable to signal that an error occurred or returning an error code from a method. Figuring out a variety of error reporting methods takes time and increases the probability of mistakes being made.

When using ad hoc methods to report errors, programmers often do not bother to check the error indicators, resulting in software that sometimes fails. Forcing programmers to confront possible errors and decide how to handle them is the second advantage of exceptions. For example, Java forces the programmer to think about how to

handle a `FileNotFoundException`. The programmer could decide to simply report the error and stop, ask the user to enter the filename again, or read an alternate file—but the error can't simply be ignored with the hope that it won't happen.

Third, exceptions separate error-handling code from the code for normal processing. This separation makes the logic for both the normal processing and handling errors easier to understand.

Fourth, an exception's stack trace provides valuable information for programmers when debugging a program.

Many beginning programmers dread exceptions because they are often the first sign of an unwelcome debugging session. This is the wrong attitude. Instead, exceptions should be welcomed as a programmer's friends. They tell us as soon as possible, with helpful debugging information, what went wrong. Finding a bug that is exposed right away, with helpful information, is much easier than debugging in languages without exceptions. In those languages, an error may go undetected until much later in the program's execution. When it is finally noticed, finding its cause may be very difficult.

11.4.2 Design by Contract

One excellent use for exceptions is to inform programmers when a method has been called with inappropriate arguments. For example, consider the `deleteMessage` method in the e-mail program. It takes a single argument, the number of the message to delete. This number is really the index into the partially filled array of messages and must be between 0 and `this.getSize()-1`, inclusive. If the message number is outside of this range, it represents a bug and an exception, typically `IllegalArgumentException`, should be thrown:

```
public void deleteMessage(int msgNum)
{ if (msgNum < 0 || msgNum > this.getSize()-1)
  { throw new IllegalArgumentException("msgNum = " + msgNum
                        + "; must be 0.." + (this.getSize()-1));
  }

  // existing code to delete the message
}
```

The requirement that `msgNum` be between 0 and `this.getSize()-1` is called a **precondition**. More generally, a precondition is anything that must be true when the method is called for it to execute correctly. It is the responsibility of the method's client to ensure that the preconditions are met. Checking the preconditions inside the method is simply a favor to those using the method: the method fails quickly with an appropriate exception that helps them find their bugs more easily. But it is a favor that should

KEY IDEA

Verifying preconditions is a huge favor to those using the method—which usually includes the programmer who wrote it.

always be extended. Consider what might happen if an invalid argument slips through and is used in a method:

- The method might fail in the middle of its processing with a confusing exception.
- The method might complete normally but compute a wrong result that affects the correctness of the program.
- The method might complete normally but leave the object in an invalid state, causing an error later in the program in an unrelated part of the code.

All of these results are undesirable and easily prevented by checking parameters for validity.

If the method's client has met the preconditions, the method is obligated to meet its **postconditions**. A postcondition is what should be true after the method executes. If the preconditions have not been met, the postconditions likely won't be met either.

The postcondition for `deleteMessage` is that the given message, `n`, has been removed from the list and all the remaining messages are renumbered to fill the "hole."

Both pre- and postconditions should be documented. Unfortunately, the standard JavaDoc tool does not support them explicitly. It does have a `@throws` tag which can be used instead. An appropriate comment for `deleteMessage` might be

```
/** Delete message number n from this mailbox,  renumbering all messages from
 *  n+1..this.getSize()-1 to have numbers n...this.getSize()-2. The renumbering
 *  preserves the order of the messages.
 *  @param n    The number of the message to delete
 *  @throws IllegalArgumentException if n is outside the range 0..this.getSize()-1  */
```

The word after `@throws` is expected to be the name of an exception class.

Pre- and postconditions are often viewed as contracts between the client and the method and using them consistently is called **design by contract** [1]. The core idea of this phrase is that each interaction between a client and a method is bound by a **contract**. The contract specifies what the client and the server can each expect of the other.

The contract between a client and method is similar to the contracts we encounter in everyday life. For example, a cell phone provider may have a contract that says if you (the client) pay $20.00 per month, they (the server) will provide up to 500 minutes of cell phone service per month. If you sign the contract and live up to your responsibilities of paying $20.00 per month, they are obligated to provide the service. If they don't, you could take them to court and sue for breach of contract. On the other hand,

[1] This phrase was trademarked by Bertrand Meyer, the developer of the Eiffel programming language. Eiffel makes extensive use of the ideas behind "Design by Contract."

if you don't pay the $20.00, they are under no obligation to provide the cell phone service. They might, but they certainly don't have to.

11.4.3 Assertions

An **assertion** is something the programmer believes will always be true at a certain point in the code. Starting with Java 1.4, the keyword `assert` is available to test assertions. It is followed by a Boolean test that should be true at that point in the program. For example, in the e-mail program, `growArray` should only be called if the partially filled array of messages is full. Thus, `growArray` should have an assertion:

```
private void growArray()
{ assert this.size == this.msgs.length;
  ...
```

If the assertion fails, the program behaves as if an exception named `AssertionError` were thrown. If the behavior is the same, what advantages do assertions have over using an exception? There are two advantages. First, assertions are easier and faster to write because they combine the test with implicitly creating and throwing the exception object.

Second, assertions can be turned off easily. Some assertions might slow the program down unacceptably. For example, some searching techniques require the array being searched to be sorted. Checking that precondition takes longer than doing the actual search. Such a precondition can be used during development and debugging, but then turned off so they have no effect on the program when deployed to users. To turn assertion checking on, execute the program using the following command line:

```
java –enableassertions «ClassName»
```

If you use an IDE, find the place where command line arguments are set and add `–enableassertions`.

It may seem natural to use `assert` to check preconditions as well. Throwing a specified exception, however, is a better solution in this case because it can document the error more accurately in terms the method's user can understand.

11.5 GUIs: Quality Interfaces

Just as the notion of quality applies to programs as a whole, it also applies to the user interface in particular. This has been driven home to me personally in a program I use to access information about students. For example:

- The information I need 90% of the time is buried in four levels of menus. Most of those menus have a single selection.

- The information I most often need is at the top of the screen but to search for another student I need to scroll through several pages of text to reach the "New Search" button.
- The program presents a substantial amount of unimportant information. The useful information is harder to use because of the clutter.

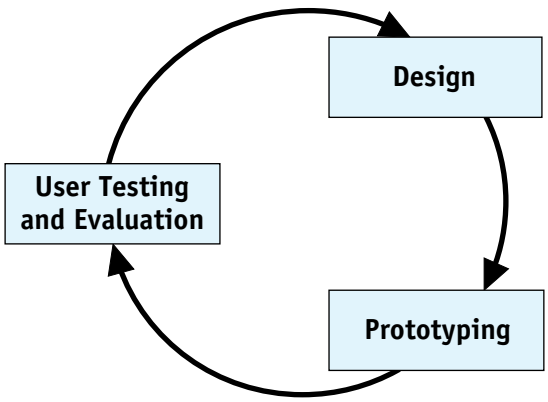
As you can tell, I am not a happy user!

11.5.1 Iterative User Interface Design

Like developing a program, developing a user interface should follow a development process. It shouldn't be surprising that excellent user interfaces are rarely achieved on the first try, so iteration is important. A simplified approach is shown in Figure 11-15.

(figure 11-15)

Iterative user interface design and evaluation process



Prototyping refers to making a model of the completed design. It may be a **low-fidelity prototype** drawn on paper index cards or a **high-fidelity prototype** that actually performs many of the operations of the finished program—or anywhere between these two extremes.

In the user evaluation and testing phase, users work with the prototype to evaluate it. Evaluation often involves the **five E's**:

- **Effective**—The completeness and accuracy with which users achieve their goals.
- **Efficient**—The speed and accuracy with which users can complete their tasks.
- **Engaging**—The degree to which the tone and style of the interface makes the product pleasant or satisfying to use.

- **Error tolerant**—How well the design prevents errors or helps with recovery from those that do occur.
- **Easy to learn**—How well the product supports both initial orientation and deepening understanding of its capabilities.

Ideally, each of the five E's is objectively measured and compared to a stated goal. For example, effectiveness might be measured by having a group of users complete a prescribed set of tasks, measuring their error rate. Efficiency might be measured by counting keystrokes and mouse clicks on a set of realistic tasks. Whether the user interface is engaging might be measured with user interviews or questionnaires. Counting the "false starts" users make would be one way to measure whether the user interface is easy to learn.

Obviously, evaluation of a user interface requires users. Having the developers act like users is not good enough—they know too much about the application and how it works.

11.5.2 User Interface Design Principles

Just as the principles of encapsulation, cohesion, information hiding, and coupling have emerged as being important to quality software, a number of design principles are important to quality user interfaces. Well-designed user interfaces are:

- Controlled by the user
- Responsive
- Understandable
- Forgiving

Controlled by the User

Modern user interfaces should give the user as much control over the process as is consistent with the user's knowledge and skill level. Whenever possible, allow the user to choose the ordering of tasks and subtasks. Allow the user to choose between using the keyboard or a mouse. Assume that the user will be interrupted and need to come back to the task later. Allow the user to customize the interface to suit his own preferences.

Responsive

Users need constant feedback to tell them how the system has interpreted their commands. To understand why, put on a blindfold and attempt to send an e-mail message. How far can you get (without specialized tools to give you feedback)?

Feedback happens in many ways: echoing characters typed by the user, highlighting buttons to show they have been "activated" but can still be aborted, disabling controls that are not appropriate in the current context, providing progress bars during long-running tasks, and so on.

Another side of a responsive system is that the feedback comes fast enough to keep the user working at full speed, whenever possible.

Understandable

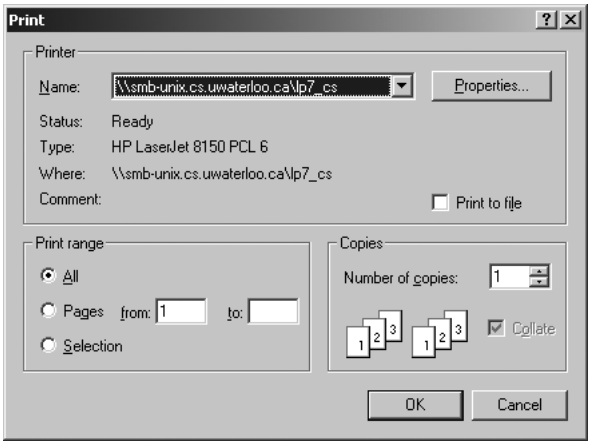
A quality user interface will be as understandable as possible.

Consistency is one way to keep a user interface understandable. Users will be able to transfer what they have learned in one part of the application to another, and probably even from one application to another. Examples of consistency include using the same language to describe the same concepts, organizing the controls on dialog boxes in the same way, using the same kinds of controls to achieve distinct but similar tasks, and so on. A set of design guidelines helps achieve consistency, as does using a standard library of user interface controls (such as `javax.swing`).

Structuring information and controls also helps promote understandability. For example, the print dialog box shown in Figure 11-16 has information grouped into three areas: Printer, Print Range, and Copies. All of the information in the Printer area is about the physical printer—which one to use, its type, whether it's ready, and so on. Many psychology studies have shown that such structure makes it easier for users to find, organize, and use the information presented to them.

(figure 11-16)

Print dialog box showing structured information and controls



Finally, make use of the fact that people recognize information much more easily than they recall it. For example, the drop-down list of printer names contains two names not shown

in Figure 11-16, `\\smb-unix.cs-uwaterloo.ca\ljp_dc3109` and `Splash G620 DocuColor 12PM`. Recognizing which of these three printers is desired is much easier than recalling 35 or more characters and typing them into a text field.

Forgiving

Finally, a quality user interface should be forgiving of user mistakes. Ideally, the interface should prevent as many mistakes as possible by, for example, disabling commands that are not applicable in the current context. But users will still make mistakes due to fatigue, distraction, uncertainty, and so on. The user interface should make it easy to correct these mistakes. It might do this by allowing the user to undo commands, or by allowing users to correct their input and reissue commands.

11.6 Summary and Concept Map

Quality software is a pleasure to work with, both as a user and as a programmer. Quality is generally increased by using an iterative development methodology and paying particular attention to the design. The concepts of encapsulation, cohesion, information hiding, and coupling all play a significant role in maintaining the internal quality of software. Similar kinds of principles apply to user interface design as well. Finally, proactively detecting and reporting bugs and exceptional events via exception objects also increases the quality of the software we write.

11.7 Problem Set

Written Exercises

11.1 Make CRC cards based on the transcript in Section 11.2.2. For each subproblem, a–d, hand in updated CRC cards and a transcript of the walk-through.

a. Finish walking through the scenario of renting one video. For example, how does the system's user know how much to charge the customer?

b. Walk through adding a new customer.

c. Walk through the scenario of returning a video on time.

d. Walk through producing a report of all customers with videos more than one week late.

11.2 The alarm clock case study in Section 8.3 uses the `newAudioClip` method in the `Applet` class to load a sound file from the disk drive. If the sound file does not exist or is in the wrong format, the `newAudioClip` method doesn't do anything at all. Is this a good idea? Defend your answer.

11.3 Is the `Video` class shown in Figure 11-6 immutable? Why or why not?

11.4 Explain why the `Customer` class, as shown in Figure 11-6, is mutable. Is this class a good candidate for an immutable class? Why?

11.5 Expand the list of scenarios for the video store given in Section 11.2.1. (*Hint*: It would be fruitful to imagine the program having just been installed in a store. What must be done before it can be used to rent a video?) Now consider the iterative nature of the software development process, as shown in Figure 11-1. Organize the scenarios into three groups: the set to implement first, the set to implement next, and the set to implement last. Give a brief rationale describing why you grouped the scenarios as you did.

11.6 For each of the following requirements documents, hand in the following:

a. Rewritten requirements (see Figure 11-4)

b CRC cards that result from the rewritten requirements

c. List of scenarios

d. Transcript of walking through one scenario

e. CRC cards as refined by the walk-through

f. Class diagram constructed from the CRC cards

Requirements Document 1: The concert hall's ticketing system is used to sell concert tickets to concert hall patrons. The system has a list of patrons who have previously purchased tickets as well as a list of upcoming concerts. It also has a map of the concert hall showing how many rows of seats there are and how many seats are in each row.

Users must be able to add new patrons, add newly scheduled concerts, and sell tickets to a particular concert to patrons. Patrons may request of block of adjacent seats in the same row.

The hall's manager will want periodic reports of how many tickets have been sold for each upcoming concert as well as patrons who have purchased tickets to more than four concerts in the last 18 months.

Requirements Document 2: A program is required to synchronize the files in two directories. This is useful, for example, to synchronize a person's laptop computer with files on their primary desktop computer.

Each file has a name and a modification date (the date and time when it was last changed). Each directory has a list of files it contains.

Input to the program are the paths to the two directories to synchronize. The program also has a list of the files that existed the last time the directories were synchronized. Let's refer to the two directories as A and B and to the list as L.

For each file *f* in directory A, the program will copy *f* to B if it does not appear in either B or L (it's a newly created file). It will delete *f* from A if *f* appears in L but not in B (it was previously deleted from B). It will copy *f* to B if it appears in B and L, and the modification date of *f* is newer than the modification date of the copy in B and the copy in B is older than L (it was changed in A but not B). It will ask the user what to do if *f* and the corresponding copy in B are both newer than L.

Similar processing also occurs for each file in B.

After both directories have been processed, they should both have exactly the same files. Write the list of those files out to use the next time the directories are synchronized.

Requirements Document 3: The game of Adventure has a series of rooms in a cave. Each room has a passage to at least one other room. Rooms may also have treasures such as lamps, keys, gold, and so on. Each treasure has an associated weight and value. A player moves between rooms with commands such as "north," "west," and "down." Each room is described when the player enters it.

A player can pick up treasures (but the player has a limit to how much he can carry—it can't carry all of the treasures). The player can also put down treasures it has previously picked up.

The game is over when the player enters the "quit" command. If the value of the player's accumulated treasures is high enough, the player is added to the game's top ten players list.

(For more information on the original adventure game, search the Web for "Colossal Cave Adventure.")

Requirements Document 4: An online auction service has a list of items for sale. Each item has a description, a seller, a current bid, and an auction close date. Buyers may search the list for items with descriptions that contain the search terms they enter. If buyers see an item they want to buy, they may bid on the item, provided the auction close date has not passed and their bid is higher than all previous bids for the item.

Once per day, the system notifies buyers and sellers of auctions that closed that day.

11.7 This chapter mentions the term *refactor* but does not describe it extensively. Research this term on the Web and write a short essay on what the term means. Some sites give concrete refactoring patterns. Describe two or three of these patterns and how they can improve code quality.

Programming Exercises

11.8 Listing 11-4 shows how `replyToMessage` could be written if there was a `constructReply` method in the `Message` class. Write `constructReply`.

Programming Projects

11.9 Find the code to the e-mail program shown in Listings 11-2 and 11-3. Rewrite the program using the heuristics in Section 11.3 and adding exceptions where appropriate. You should not need to modify `MailUI.java`, but other classes may need changing and new classes may be added. The user of your rewritten program should not be able to detect any differences between it and the original. Only the programmers working with it will know how much the quality has improved.

11.10 Consider the video store program discussed in Section 11.2.

a. Write code so that the tests given in Listing 11-1 will pass.

b. Walk through the scenario of adding a customer. Develop tests for this scenario and implement the code required to pass the tests. Assume that a user interface gathers the required information about the customer and provides it to your code.

c. Walk through the scenario of renting a video to an existing customer. Develop tests and implement the code to pass the tests. Assume that a user interface gathers a customer identification number and a video identification number and provides them to your code.