CHAPTER 12

OBJECT-ORIENTED DESIGN

CHAPTER GOALS

To learn how to discover new classes and methods

To use CRC cards for class discovery

To understand the concepts of cohesion and coupling

To identify inheritance, aggregation, and dependency relationships between classes

To describe class relationships using UML class diagrams

To apply object-oriented design techniques to building complex programs

To use packages to organize programs

CHAPTER CONTENTS

Successfully implementing a software system—as simple as your next homework project or as complex as the next air traffic monitoring system—requires a great deal of planning and design. In fact, for larger projects, the amount of time spent on planning and design is much greater than the amount of time spent on programming and testing.

Do you find that most of your homework time is spent in front of the computer, keying in code and fixing bugs? If so, you can probably save time by focusing on a better design before you start coding. This chapter tells you how to approach the design of an object-oriented program in a systematic manner.

12.1 Classes and Their Responsibilities

When you design a program, you work from a *requirements specification*, a description of what your program should do. The designer's task is to discover structures that make it possible to implement the requirements in a computer program. In the following sections, we will examine the steps of the design process.

12.1.1 Discovering Classes

To discover classes, look for nouns in the problem description.

When you solve a problem using objects and classes, you need to determine the classes required for the implementation. You may be able to reuse existing classes, or you may need to implement new ones.

One simple approach for discovering classes and methods is to look for the nouns and verbs in the requirements specification. Often, *nouns* correspond to classes, and *verbs* correspond to methods.

For example, suppose your job is to print an invoice such as the one in Figure 1. Obvious classes that come to mind are `Invoice`, `LineItem`, and `Customer`. It is a good idea to keep a list of *candidate classes* on a whiteboard or a sheet of paper. As you brainstorm, simply put all ideas for classes onto the list. You can always cross out the ones that weren't useful after all.

Concepts from the problem domain are good candidates for classes.

In general, concepts from the problem domain, be it science, business, or a game, often make good classes. Examples are

- `Cannonball`
- `CashRegister`
- `Monster`

The name for such a class should be a noun that describes the concept.

Not all classes can be discovered from the program requirements. Most complex programs need classes for tactical purposes, such as file or database access, user interfaces, control mechanisms, and so on.

Some of the classes that you need may already exist, either in the standard library or in a program that you developed previously. You also may be able to use inheritance to extend existing classes into classes that match your needs.

INVOICE

Sam's Small Appliances
100 Main Street
Anytown, CA 98765

Item	Qty	Price	Total
Toaster	3	$29.95	$89.85
Hair Dryer	1	$24.95	$24.95
Car Vacuum	2	$19.99	$39.98

AMOUNT DUE: **$154.78**

Figure 1 An Invoice

What might not be a good class? If you can't tell from the class name what an object of the class is supposed to do, then you are probably not on the right track. For example, your homework assignment might be to write a program that prints paychecks. Suppose you start by trying to design a class `PaycheckProgram`. What would an object of this class do? An object of this class would have to do everything that the homework needs to do. That doesn't simplify anything. A better class would be `Paycheck`. Then your program can manipulate one or more `Paycheck` objects.

Another common mistake, often made by students who are used to writing programs that consist of static methods, is to turn an action into a class. For example, if your homework assignment is to compute a paycheck, you may consider writing a class `ComputePaycheck`. But can you visualize a "ComputePaycheck" object? The fact that "ComputePaycheck" isn't a noun tips you off that you are on the wrong track. On the other hand, a `Paycheck` class makes intuitive sense. The word "paycheck" is a noun. You can visualize a paycheck object. You can then think about useful methods of the `Paycheck` class, such as `computeTaxes`, that help you solve the assignment.

In a class scheduling system, potential classes from the problem domain include Class, LectureHall, Instructor, and Student.

Finally, a common error is to overdo the class discovery process. For example, should an address be an object of an `Address` class, or should it simply be a string? There is no perfect answer—it depends on the task that you want to solve. If your software needs to analyze addresses (for example, to determine shipping costs), then an `Address` class is an appropriate design. However, if your software will never need such a capability, you should not waste time on an overly complex design. It is your job to find a balanced design; one that is neither too limiting nor excessively general.

12.1.2 The CRC Card Method

Once you have identified a set of classes, you define the behavior for each class. Find out what methods you need to provide for each class in order to solve the programming problem. A simple rule for finding these methods is to look for *verbs* in the task description, then match the verbs to the appropriate objects. For example, in the invoice program, a class needs to compute the amount due. Now you need to figure out *which class* is responsible for this method. Do customers compute what they owe? Do invoices total up the amount due? Do the items total themselves up? The best choice is to make "compute amount due" the responsibility of the `Invoice` class.

A CRC card describes a class, its responsibilities, and its collaborating classes.

An excellent way to carry out this task is the "**CRC card** method." *CRC* stands for "classes", "responsibilities", "collaborators", and in its simplest form, the method works as follows: Use an index card for each *class* (see Figure 2). As you think about verbs in the task description that indicate methods, you pick the card of the class that you think should be responsible, and write that *responsibility* on the card.

For each responsibility, you record which other classes are needed to fulfill it. Those classes are the **collaborators**.

For example, suppose you decide that an invoice should compute the amount due. Then you write "compute amount due" on the left-hand side of an index card with the title `Invoice`.

If a class can carry out that responsibility by itself, do nothing further. But if the class needs the help of other classes, write the names of these collaborators on the right-hand side of the card.

To compute the total, the invoice needs to ask each line item about its total price. Therefore, the `LineItem` class is a collaborator.

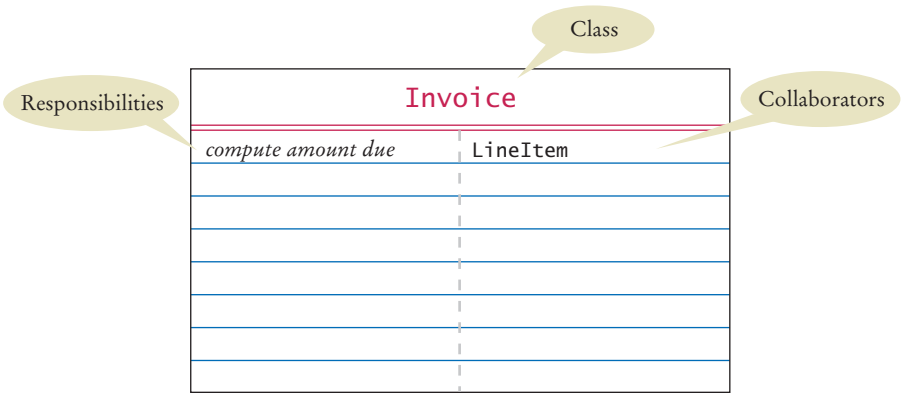


Figure 2 A CRC Card

This is a good time to look up the index card for the `LineItem` class. Does it have a "get total price" method? If not, add one.

How do you know that you are on the right track? For each responsibility, ask yourself how it can actually be done, using the responsibilities written on the various cards. Many people find it helpful to group the cards on a table so that the collaborators are close to each other, and to simulate tasks by moving a token (such as a coin) from one card to the next to indicate which object is currently active.

Keep in mind that the responsibilities that you list on the CRC card are on a *high level*. Sometimes a single responsibility may need two or more Java methods for carrying it out. Some researchers say that a CRC card should have no more than three distinct responsibilities.

The CRC card method is informal on purpose, so that you can be creative and discover classes and their properties. Once you find that you have settled on a good set of classes, you will want to know how they are related to each other. Can you find classes with common properties, so that some responsibilities can be taken care of by a common superclass? Can you organize classes into clusters that are independent of each other? Finding class relationships and documenting them with diagrams is the topic of Section 12.2.

12.1.3 Cohesion

The public interface of a class is cohesive if all of its features are related to the concept that the class represents.

A class should represent a single concept. The public methods and constants that the public interface exposes should be *cohesive*. That is, all interface features should be closely related to the single concept that the class represents.

If you find that the public interface of a class refers to multiple concepts, then that is a good sign that it may be time to use separate classes instead. Consider, for example, the public interface of a `CashRegister` class:

```
public class CashRegister
{
   public static final double NICKEL_VALUE = 0.05;
   public static final double DIME_VALUE = 0.1;
   public static final double QUARTER_VALUE = 0.25;
   . . .
   public void enterPayment(int dollars, int quarters,
         int dimes, int nickels, int pennies) { . . . }
   . . .
}
```

There are really two concepts here: a cash register that holds coins and computes their total, and the values of individual coins. (For simplicity, we assume that the cash register only holds coins, not bills. Exercise P12.2 discusses a more general solution.)

It makes sense to have a separate `Coin` class and have coins responsible for knowing their values.

```
public class Coin
{
   . . .
   public Coin(double aValue, String aName) { . . . }
   public double getValue() { . . . }
   . . .
}
```

Then the `CashRegister` class can be simplified:

```
public class CashRegister
{
   . . .
   public void enterPayment(int coinCount, Coin coinType) { . . . }
   . . .
}
```

ONLINE EXAMPLE

A sample program using the `Coin` and `CashRegister` classes.

Now the `CashRegister` class no longer needs to know anything about coin values. The same class can equally well handle euros or zorkmids!

This is clearly a better solution, because it separates the responsibilities of the cash register and the coins.

1. What is the rule of thumb for finding classes?
2. Your job is to write a program that plays chess. Might `ChessBoard` be an appropriate class? How about `MovePiece`?
3. Suppose the invoice is to be saved to a file. Name a likely collaborator.
4. Looking at the invoice in Figure 1, what is a likely responsibility of the `Customer` class?
5. What do you do if a CRC card has ten responsibilities?

Practice It Now you can try these exercises at the end of the chapter: R12.4, R12.5, R12.12.

12.2 Relationships Between Classes

When designing a program, it is useful to document the relationships between classes. This helps you in a number of ways. For example, if you find classes with common behavior, you can save effort by placing the common behavior into a superclass. If you know that some classes are *not* related to each other, you can assign different programmers to implement each of them, without worrying that one of them has to wait for the other.

In the following sections, we will describe the most common types of relationships.

12.2.1 Dependency

A class depends on another class if it uses objects of that class.

Many classes need other classes in order to do their jobs. For example, in Section 12.1.3, we described a design of a `CashRegister` class that depends on the `Coin` class to determine the value of the payment.

The dependency relationship is sometimes nicknamed the "knows about" relationship. The cash register in Section 12.1.3 knows that there are coin objects. In contrast, the `Coin` class does *not* depend on the `CashRegister` class. Coins have no idea that they are being collected in cash registers, and they can carry out their work without ever calling any method in the `CashRegister` class.

To visualize relationships, such as dependency between classes, programmers draw class diagrams. In this book, we use the UML ("Unified Modeling Language") notation for objects and classes. UML is a notation for object-oriented analysis and

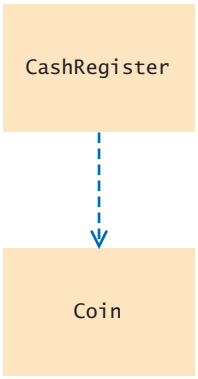


Figure 3
Dependency Relationship Between the `CashRegister` and `Coin` Classes

design invented by Grady Booch, Ivar Jacobson, and James Rumbaugh, three leading researchers in object-oriented software development. The UML notation distinguishes between *object diagrams* and *class diagrams*. An object diagram shows individual objects, their attributes, and the relationships between them. Chapter 8 has several object diagrams. A class diagram shows classes and the relationships between them. In Chapter 9, you saw class diagrams that show inheritance relationships. In the UML notation, we underline the names of classes in object diagrams but not in class diagrams.

In a class diagram, you denote dependency by a dashed line with a ≫-shaped open arrow tip. The arrow tip points to the class on which the other class depends. Figure 3 shows a class diagram indicating that the `CashRegister` class depends on the `Coin` class.

If many classes of a program depend on each other, then we say that the **coupling** between classes is high. Conversely, if there are few dependencies between classes, then we say that the coupling is low (see Figure 4).

It is a good practice to minimize the coupling (i.e., dependency) between classes.

Why does coupling matter? If the `Coin` class changes in the next release of the program, all the classes that depend on it may be affected. If the change is drastic, the coupled classes must all be updated. Furthermore, if we would like to use a class in another program, we have to take with it all the classes on which it depends. Thus, we want to remove unnecessary coupling between classes.

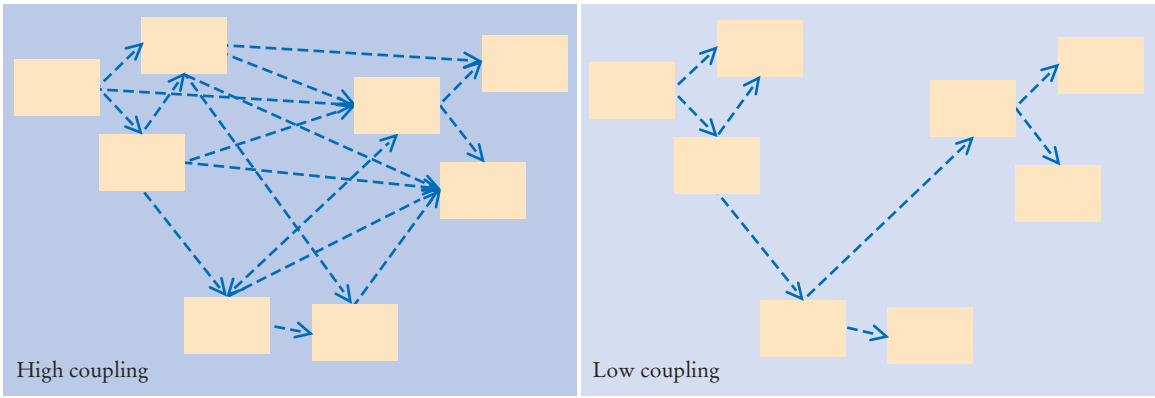


Figure 4 High and Low Coupling Between Classes

12.2.2 Aggregation

A class aggregates another if its objects contain objects of the other class.

Another fundamental relationship between classes is the "aggregation" relationship (which is informally known as the "has-a" relationship).

The **aggregation** relationship states that objects of one class contain objects of another class. Consider a quiz that is made up of questions. Because each quiz has one or more questions, we say that the class `Quiz` *aggregates* the class `Question`. In the UML notation, aggregation is denoted by a line with a diamond-shaped symbol attached to the aggregating class (see Figure 5).

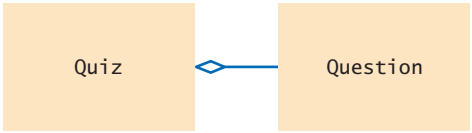


Figure 5
Class Diagram Showing Aggregation

Finding out about aggregation is very helpful for deciding how to implement classes. For example, when you implement the `Quiz` class, you will want to store the questions of a quiz as an instance variable.

Because a quiz can have any number of questions, an array or array list is a good choice for collecting them:

```
public class Quiz
{
   private ArrayList<Question> questions;
   . . .
}
```

Aggregation is a stronger form of dependency. If a class has objects of another class, it certainly knows about the other class. However, the converse is not true. For example, a class may use the `Scanner` class without ever declaring an instance variable of class `Scanner`. The class may simply construct a local variable of type `Scanner`, or its methods may receive `Scanner` objects as arguments. This use is not aggregation because the objects of the class don't contain `Scanner` objects—they just create or receive them for the duration of a single method.

Generally, you need aggregation when an object needs to remember another object *between method calls*.

ONLINE EXAMPLE

The complete `Quiz` and `Question` classes.

A car has a motor and tires. In object-oriented design, this "has-a" relationship is called aggregation.

12.2.3 Inheritance

Inheritance is a relationship between a more general class (the superclass) and a more specialized class (the subclass). This relationship is often described as the "is-a" relationship. Every truck *is a* vehicle. Every savings account *is a* bank account.

Inheritance (the *is-a* relationship) is sometimes inappropriately used when the *has-a* relationship would be more appropriate.

Inheritance is sometimes abused. For example, consider a `Tire` class that describes a car tire. Should the class `Tire` be a subclass of a class `Circle`? It sounds convenient. There are quite a few useful methods in the `Circle` class—for example, the `Tire` class may inherit methods that compute the radius, perimeter, and center point, which should come in handy when drawing tire shapes. Though it may be convenient for the programmer, this arrangement makes no sense conceptually. It isn't true that every tire is a circle. Tires are car parts, whereas circles are geometric objects. There is a relationship between tires and circles, though. A tire *has a* circle as its boundary. Use aggregation:

```
public class Tire
{
   private String rating;
   private Circle boundary;
   . . .
}
```

Aggregation (the *has-a* relationship) denotes that objects of one class contain references to objects of another class.

Here is another example: Every car *is a* vehicle. Every car *has a* tire (in fact, it typically has four or, if you count the spare, five). Thus, you would use inheritance from `Vehicle` and use aggregation of `Tire` objects:

```
public class Car extends Vehicle
{
   private Tire[] tires;
   . . .
}
```

See Figure 6 for the UML diagram.

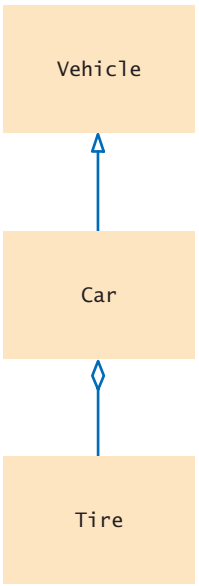


Figure 6
UML Notation for Inheritance and Aggregation

You need to be able to distinguish the UML notation for inheritance, interface implementation, aggregation, and dependency.

The arrows in the UML notation can get confusing. Table 1 shows a summary of the four UML relationship symbols that we use in this book.

Table 1 UML Relationship Symbols

Relationship	Symbol	Line Style	Arrow Tip
Inheritance		Solid	Triangle
Interface Implementation		Dotted	Triangle
Aggregation		Solid	Diamond
Dependency		Dotted	Open

6. Consider the `CashRegisterTester` class of Chapter 8. On which classes does it depend?
7. Consider the `Question` and `ChoiceQuestion` objects of Chapter 9. How are they related?
8. Consider the `Quiz` class described in Section 12.2.2. Suppose a quiz contains a mixture of `Question` and `ChoiceQuestion` objects. Which classes does the `Quiz` class depend on?
9. Why should coupling be minimized between classes?
10. In an e-mail system, messages are stored in a mailbox. Draw a UML diagram that shows the appropriate aggregation relationship.
11. You are implementing a system to manage a library, keeping track of which books are checked out by whom. Should the `Book` class aggregate `Patron` or the other way around?
12. In a library management system, what would be the relationship between classes `Patron` and `Author`?

Practice It Now you can try these exercises at the end of the chapter: R12.8, R12.9, R12.13.

HOW TO 12.1 Using CRC Cards and UML Diagrams in Program Design

Before writing code for a complex problem, you need to design a solution. The methodology introduced in this chapter suggests that you follow a design process that is composed of the following tasks:

- Discover classes.
- Determine the responsibilities of each class.
- Describe the relationships between the classes.

CRC cards and UML diagrams help you discover and record this information.

Step 1 Discover classes.

Highlight the nouns in the problem description. Make a list of the nouns. Cross out those that don't seem to be reasonable candidates for classes.

Step 2 Discover responsibilities.

Make a list of the major tasks that your system needs to fulfill. From those tasks, pick one that is not trivial and that is intuitive to you. Find a class that is responsible for carrying out that task. Make an index card and write the name and the task on it. Now ask yourself how an object of the class can carry out the task. It probably needs help from other objects. Then make CRC cards for the classes to which those objects belong and write the responsibilities on them.

Don't be afraid to cross out, move, split, or merge responsibilities. Rip up cards if they become too messy. This is an informal process.

You are done when you have walked through all major tasks and are satisfied that they can all be solved with the classes and responsibilities that you discovered.

Step 3 Describe relationships.

Make a class diagram that shows the relationships between all the classes that you discovered.

Start with inheritance—the *is-a* relationship between classes. Is any class a specialization of another? If so, draw inheritance arrows. Keep in mind that many designs, especially for simple programs, don't use inheritance extensively.

The "collaborators" column of the CRC card tells you which classes are used by that class. Draw dependency arrows for the collaborators on the CRC cards.

Some dependency relationships give rise to aggregations. For each of the dependency relationships, ask yourself: How does the object locate its collaborator? Does it navigate to it directly because it stores a reference? In that case, draw an aggregation arrow. Or is the collaborator a method parameter variable or return value? Then simply draw a dependency arrow.

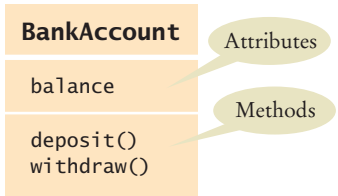
Special Topic 12.1

Attributes and Methods in UML Diagrams

Sometimes it is useful to indicate class *attributes* and *methods* in a class diagram. An **attribute** is an externally observable property that objects of a class have. For example, `name` and `price` would be attributes of the `Product` class. Usually, attributes correspond to instance variables. But they don't have to—a class may have a different way of organizing its data. For example, a `GregorianCalendar` object from the Java library has attributes `day`, `month`, and `year`, and it would be appropriate to draw a UML diagram that shows these attributes. However, the class doesn't actually have instance variables that store these quantities. Instead, it internally represents all dates by counting the milliseconds from January 1, 1970—an implementation detail that a class user certainly doesn't need to know about.

You can indicate attributes and methods in a class diagram by dividing a class rectangle into three compartments, with the class name in the top, attributes in the middle, and methods in the bottom (see the figure below). You need not list *all* attributes and methods in a particular diagram. Just list the ones that are helpful for understanding whatever point you are making with a particular diagram.

Also, don't list as an attribute what you also draw as an aggregation. If you denote by aggregation the fact that a `Car` has `Tire` objects, don't add an attribute `tires`.



Attributes and Methods in a Class Diagram

Special Topic 12.2

Multiplicities

Some designers like to write *multiplicities* at the end(s) of an aggregation relationship to denote how many objects are aggregated. The notations for the most common multiplicities are:

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

The figure below shows that a customer has one or more bank accounts.

An Aggregation Relationship with Multiplicities

Special Topic 12.3

Aggregation, Association, and Composition

Some designers find the aggregation or *has-a* terminology unsatisfactory. For example, consider customers of a bank. Does the bank "have" customers? Do the customers "have" bank accounts, or does the bank "have" them? Which of these "has" relationships should be modeled by aggregation? This line of thinking can lead us to premature implementation decisions.

Early in the design phase, it makes sense to use a more general relationship between classes called **association**. A class is associated with another if you can *navigate* from objects of one class to objects of the other class. For example, given a `Bank` object, you can navigate to `Customer` objects, perhaps by accessing an instance variable, or by making a database lookup.

The UML notation for an association relationship is a solid line, with optional arrows that show in which directions you can navigate the relationship. You can also add words to the line ends to further explain the nature of the relationship. The figure below shows that you can navigate from `Bank` objects to `Customer` objects, but you cannot navigate the other way around. That is, in this particular design, the `Customer` class has no mechanism to determine in which banks it keeps its money.

An Association Relationship

The UML standard also recognizes a stronger form of the aggregation relationship called **composition**, where the aggregated objects do not have an existence independent of the containing object. For example, composition models the relationship between a bank and its accounts. If a bank closes, the account objects cease to exist as well. In the UML notation, composition looks like aggregation with a filled-in diamond.

A Composition Relationship

Frankly, the differences between aggregation, association, and composition can be confusing, even to experienced designers. If you find the distinction helpful, by all means use the relationship that you find most appropriate. But don't spend time pondering subtle differences between these concepts. From the practical point of view of a Java programmer, it is useful to know when objects of one class have references to objects of another class. The aggregation or *has-a* relationship accurately describes this phenomenon.

Programming Tip 12.1

Make Parallel Arrays into Arrays of Objects

Sometimes, you find yourself using arrays or array lists of the same length, each of which stores a part of what conceptually should be an object. In that situation, it is a good idea to reorganize your program and use a single array or array list whose elements are objects.

For example, suppose an invoice contains a series of item descriptions and prices. One solution is to keep two arrays:

```
String[] descriptions;
double[] prices;
```

Each of the arrays will have the same length, and the `i`th *slice*, consisting of `descriptions[i]` and `prices[i]`, contains data that need to be processed together. These arrays are called **parallel arrays** (see Figure 7).

Parallel arrays become a headache in larger programs. The programmer must ensure that the arrays always have the same length and that each slice is filled with values that actually belong together. Moreover, any method that operates on a slice must get all values of the slice as arguments, which is tedious to program.

The remedy is simple. Look at the slice and find the *concept* that it represents. Then make the concept into a class. In this example, each slice contains the description and price of an item; turn this into a class:

Avoid parallel arrays by changing them into arrays of objects.

```
public class Item
{
   private String description;
   private double price;
   . . .
}
```

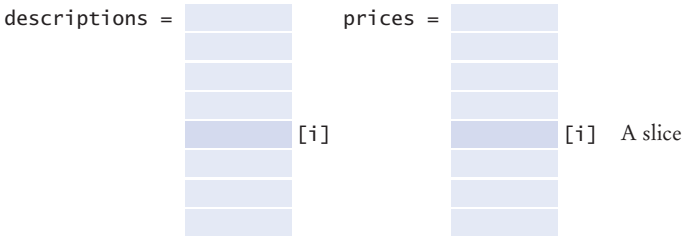


Figure 7
Parallel Arrays

You can now eliminate the parallel arrays and replace them with a single array:

```
Item[] items;
```

Each slot in the resulting array corresponds to a slice in the set of parallel arrays (see Figure 8).

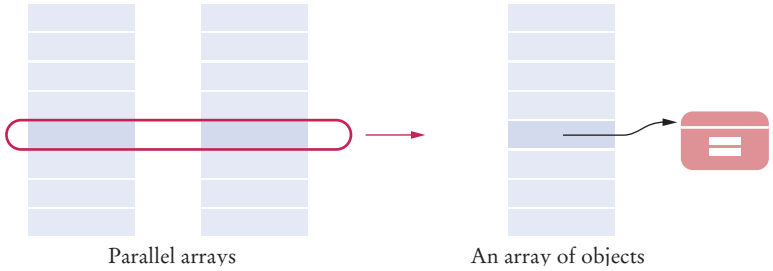


Figure 8 Eliminating Parallel Arrays

Programming Tip 12.2

Consistency

In this chapter you learned of two criteria for improving the quality of the public interface of a class. You should maximize cohesion and remove unnecessary coupling. There is another criterion that we would like you to pay attention to—*consistency*. When you have a set of methods, follow a consistent scheme for their names and parameter variables. This is simply a sign of good craftsmanship.

Sadly, you can find any number of inconsistencies in the standard Java library. Here is an example. To show an input dialog box, you call

```
JOptionPane.showInputDialog(promptString)
```

To show a message dialog box, you call

```
JOptionPane.showMessageDialog(null, messageString)
```

What's the `null` argument? It turns out that the `showMessageDialog` method needs an argument to specify the parent window, or `null` if no parent window is required. But the `showInputDialog` method requires no parent window. Why the inconsistency? There is no reason. It would have been an easy matter to supply a `showMessageDialog` method that exactly mirrors the `showInputDialog` method.

Inconsistencies such as these are not fatal flaws, but they are an annoyance, particularly because they can be so easily avoided.

12.3 Application: Printing an Invoice

In this book, we discuss a five-part program development process that is particularly well suited for beginning programmers:

1. Gather requirements.
2. Use CRC cards to find classes, responsibilities, and collaborators.
3. Use UML diagrams to record class relationships.
4. Use `javadoc` to document method behavior.
5. Implement your program.

There isn't a lot of notation to learn. The class diagrams are simple to draw. The deliverables of the design phase are obviously useful for the implementation phase—you simply take the source files and start adding the method code. Of course, as your projects get more complex, you will want to learn more about formal design methods. There are many techniques to describe object scenarios, call sequencing, the large-scale structure of programs, and so on, that are very beneficial even for relatively simple projects. *The Unified Modeling Language User Guide* gives a good overview of these techniques.

In this section, we will walk through the object-oriented design technique with a very simple example. In this case, the methodology may feel overblown, but it is a good introduction to the mechanics of each step. You will then be better prepared for the more complex programs that you will encounter in the future.

12.3.1 Requirements

Start the development process by gathering and documenting program requirements.

Before you begin designing a solution, you should gather all requirements for your program in plain English. Write down what your program should do. It is helpful to include typical scenarios in addition to a general description.

The task of our sample program is to print out an invoice. An invoice describes the charges for a set of products in certain quantities. (We omit complexities such as dates, taxes, and invoice and customer numbers.) The program simply prints the billing address, all line items, and the amount due. Each line item contains the description and unit price of a product, the quantity ordered, and the total price.

An invoice lists the charges for each item and the amount due.

```
                    I N V O I C E

  Sam's Small Appliances
  100 Main Street
  Anytown, CA 98765

  Description                    Price  Qty  Total
  Toaster                        29.95    3  89.85
  Hair dryer                     24.95    1  24.95
  Car vacuum                     19.99    2  39.98

  AMOUNT DUE: $154.78
```

Also, in the interest of simplicity, we do not provide a user interface. We just supply a test program that adds line items to the invoice and then prints it.

12.3.2 CRC Cards

Use CRC cards to find classes, responsibilities, and collaborators.

When designing an object-oriented program, you need to discover classes. Classes correspond to nouns in the requirements specification. In this problem, it is pretty obvious what the nouns are:

```
Invoice          Address          LineItem
Product          Description      Price
Quantity         Total            Amount due
```

(Of course, `Toaster` doesn't count—it is the description of a `LineItem` object and therefore a data value, not the name of a class.)

Description and price are attributes of the `Product` class. What about the quantity? The quantity is not an attribute of a `Product`. Just as in the printed invoice, let's have a class `LineItem` that records the product and the quantity (such as "3 toasters").

The total and amount due are computed—not stored anywhere. Thus, they don't lead to classes.

After this process of elimination, we are left with four candidates for classes:

```
Invoice
Address
LineItem
Product
```

Each of them represents a useful concept, so let's make them all into classes.

The purpose of the program is to print an invoice. However, the `Invoice` class won't necessarily know whether to display the output in `System.out`, in a text area, or in a file. Therefore, let's relax the task slightly and make the invoice responsible for *formatting* the invoice. The result is a string (containing multiple lines) that can be printed out or displayed. Record that responsibility on a CRC card:

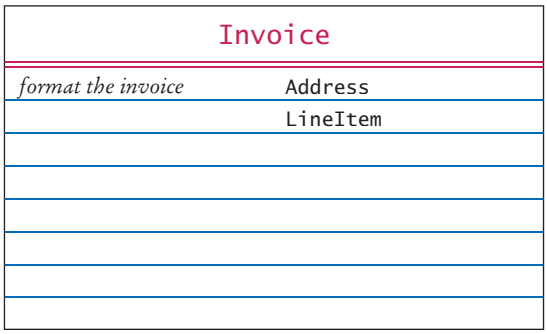
Invoice
format the invoice

How does an invoice format itself? It must format the billing address, format all line items, and then add the amount due. How can the invoice format an address? It can't—that really is the responsibility of the `Address` class. This leads to a second CRC card:

Address
format the address

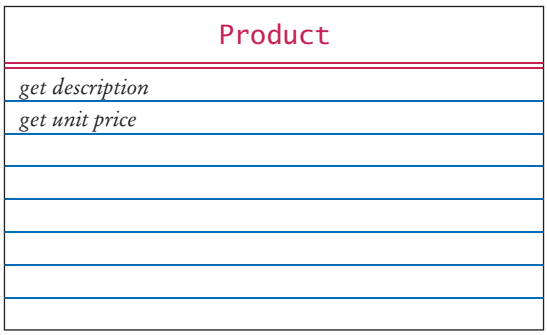
Similarly, formatting of a line item is the responsibility of the `LineItem` class.

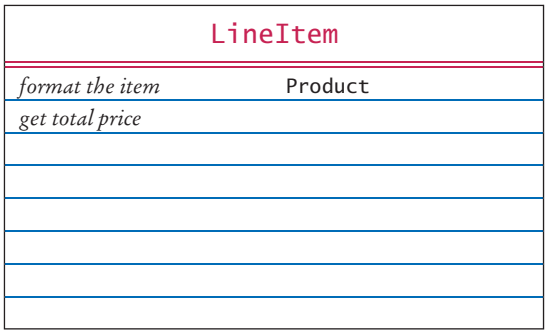
The `format` method of the `Invoice` class calls the `format` methods of the `Address` and `LineItem` classes. Whenever a method uses another class, you list that other class as a collaborator. In other words, `Address` and `LineItem` are collaborators of `Invoice`:



When formatting the invoice, the invoice also needs to compute the total amount due. To obtain that amount, it must ask each line item about the total price of the item.

How does a line item obtain that total? It must ask the product for the unit price, and then multiply it by the quantity. That is, the `Product` class must reveal the unit price, and it is a collaborator of the `LineItem` class.





Finally, the invoice must be populated with products and quantities, so that it makes sense to format the result. That too is a responsibility of the `Invoice` class.

Invoice	
format the invoice	Address
add a product and quantity	LineItem
	Product

We now have a set of CRC cards that completes the CRC card process.

12.3.3 UML Diagrams

Use UML diagrams to record class relationships.

After you have discovered classes and their relationships with CRC cards, you should record your findings in a UML diagram. The dependency relationships come from the collaboration column on the CRC cards. Each class depends on the classes with which it collaborates. In our example, the `Invoice` class collaborates with the `Address`, `LineItem`, and `Product` classes. The `LineItem` class collaborates with the `Product` class.

Now ask yourself which of these dependencies are actually aggregations. How does an invoice know about the address, line item, and product objects with which it collaborates? An invoice object must hold references to the address and the line items when it formats the invoice. But an invoice object need not hold a reference to a product object when adding a product. The product is turned into a line item, and then it is the item's responsibility to hold a reference to it.

Therefore, the `Invoice` class aggregates the `Address` and `LineItem` classes. The `LineItem` class aggregates the `Product` class. However, there is no *has-a* relationship between an invoice and a product. An invoice doesn't store products directly—they are stored in the `LineItem` objects.

There is no inheritance in this example.

Figure 9 shows the class relationships that we discovered.

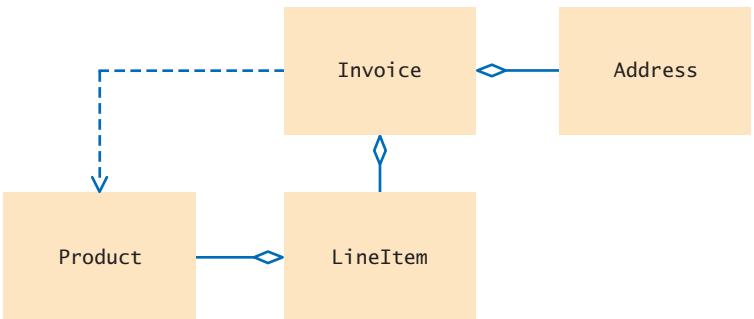


Figure 9 The Relationships Between the Invoice Classes

12.3.4 Method Documentation

Use javadoc comments (with the method bodies left blank) to record the behavior of classes.

The final step of the design phase is to write the documentation of the discovered classes and methods. Simply write a Java source file for each class, write the method comments for those methods that you have discovered, and leave the bodies of the methods blank.

```
/**
   Describes an invoice for a set of purchased products.
*/
public class Invoice
{
   /**
      Adds a charge for a product to this invoice.
      @param aProduct the product that the customer ordered
      @param quantity the quantity of the product
   */
   public void add(Product aProduct, int quantity)
   {
   }

   /**
      Formats the invoice.
      @return the formatted invoice
   */
   public String format()
   {
   }
}


/**
   Describes a quantity of an article to purchase.
*/
public class LineItem
{
   /**
      Computes the total cost of this line item.
      @return the total price
   */
   public double getTotalPrice()
   {
   }

   /**
      Formats this item.
      @return a formatted string of this item
   */
   public String format()
   {
   }
}


/**
   Describes a product with a description and a price.
*/
public class Product
{
```

```
   /**
      Gets the product description.
      @return the description
   */
   public String getDescription()
   {
   }

   /**
      Gets the product price.
      @return the unit price
   */
   public double getPrice()
   {
   }
}


/**
   Describes a mailing address.
*/
public class Address
{
   /**
      Formats the address.
      @return the address as a string with three lines
   */
   public String format()
   {
   }
}
```

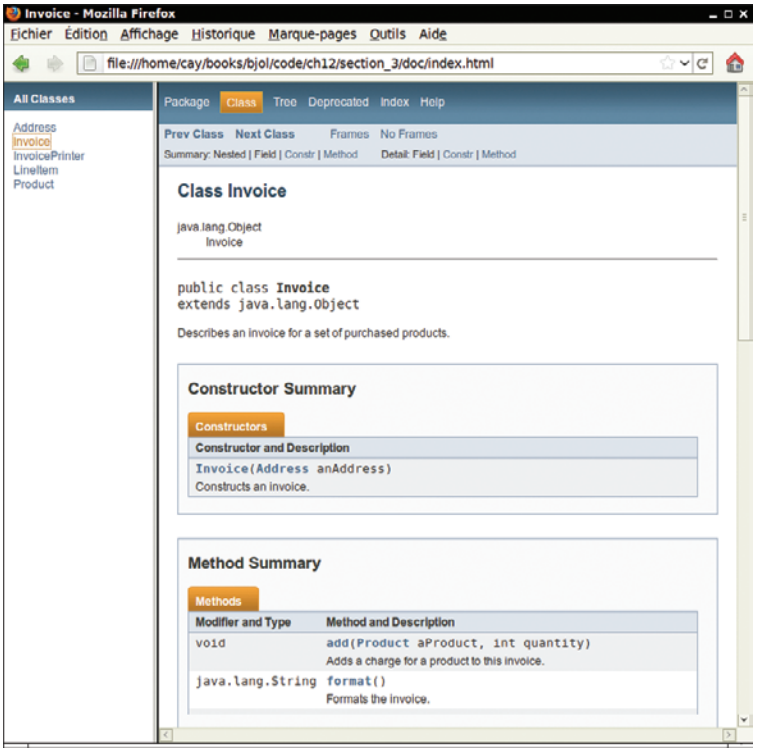


Figure 10
Class Documentation in HTML Format

Then run the javadoc program to obtain a neatly formatted version of your documentation in HTML format (see Figure 10).

This approach for documenting your classes has a number of advantages. You can share the HTML documentation with others if you work in a team. You use a format that is immediately useful—Java source files that you can carry into the implementation phase. And, most importantly, you supply the comments for the key methods—a task that less prepared programmers leave for later, and often neglect for lack of time.

12.3.5 Implementation

After completing the design, implement your classes.

After you have completed the object-oriented design, you are ready to implement the classes.

You already have the method parameter variables and comments from the previous step. Now look at the UML diagram to add instance variables. Aggregated classes yield instance variables. Start with the `Invoice` class. An invoice aggregates `Address` and `LineItem`. Every invoice has one billing address, but it can have many line items. To store multiple `LineItem` objects, you can use an array list. Now you have the instance variables of the `Invoice` class:

```
public class Invoice
{
   private Address billingAddress;
   private ArrayList<LineItem> items;
   . . .
}
```

A line item needs to store a `Product` object and the product quantity. That leads to the following instance variables:

```
public class LineItem
{
   private int quantity;
   private Product theProduct;
   . . .
}
```

The methods themselves are now easy to implement. Here is a typical example. You already know what the `getTotalPrice` method of the `LineItem` class needs to do—get the unit price of the product and multiply it with the quantity.

```
/**
   Computes the total cost of this line item.
   @return the total price
*/
public double getTotalPrice()
{
   return theProduct.getPrice() * quantity;
}
```

We will not discuss the other methods in detail—they are equally straightforward.

Finally, you need to supply constructors, another routine task.

The entire program is shown below. It is a good practice to go through it in detail and match up the classes and methods against the CRC cards and UML diagram.

In this chapter, you learned a systematic approach for building a relatively complex program. However, object-oriented design is definitely not a spectator sport. To really learn how to design and implement programs, you have to gain experience by repeating this process with your own projects. It is quite possible that you don't

immediately home in on a good solution and that you need to go back and reorganize your classes and responsibilities. That is normal and only to be expected. The purpose of the object-oriented design process is to spot these problems in the design phase, when they are still easy to rectify, instead of in the implementation phase, when massive reorganization is more difficult and time consuming.

section_3/InvoicePrinter.java

```
1  /**
2     This program demonstrates the invoice classes by
3     printing a sample invoice.
4  */
5  public class InvoicePrinter
6  {
7     public static void main(String[] args)
8     {
9        Address samsAddress
10             = new Address("Sam's Small Appliances",
11             "100 Main Street", "Anytown", "CA", "98765");
12
13       Invoice samsInvoice = new Invoice(samsAddress);
14       samsInvoice.add(new Product("Toaster", 29.95), 3);
15       samsInvoice.add(new Product("Hair dryer", 24.95), 1);
16       samsInvoice.add(new Product("Car vacuum", 19.99), 2);
17
18       System.out.println(samsInvoice.format());
19    }
20 }
```

section_3/Invoice.java

```
1  import java.util.ArrayList;
2
3  /**
4     Describes an invoice for a set of purchased products.
5  */
6  public class Invoice
7  {
8     private Address billingAddress;
9     private ArrayList<LineItem> items;
10
11    /**
12       Constructs an invoice.
13       @param anAddress the billing address
14    */
15    public Invoice(Address anAddress)
16    {
17       items = new ArrayList<LineItem>();
18       billingAddress = anAddress;
19    }
20
21    /**
22       Adds a charge for a product to this invoice.
23       @param aProduct the product that the customer ordered
24       @param quantity the quantity of the product
25    */
26    public void add(Product aProduct, int quantity)
27    {
28       LineItem anItem = new LineItem(aProduct, quantity);
```

```
29          items.add(anItem);
30       }
31
32       /**
33          Formats the invoice.
34          @return the formatted invoice
35       */
36       public String format()
37       {
38          String r = "                       I N V O I C E\n\n"
39                + billingAddress.format()
40                + String.format("\n\n%-30s%8s%5s%8s\n",
41                "Description", "Price", "Qty", "Total");
42
43          for (LineItem item : items)
44          {
45             r = r + item.format() + "\n";
46          }
47
48          r = r + String.format("\nAMOUNT DUE: $%8.2f", getAmountDue());
49
50          return r;
51       }
52
53       /**
54          Computes the total amount due.
55          @return the amount due
56       */
57       private double getAmountDue()
58       {
59          double amountDue = 0;
60          for (LineItem item : items)
61          {
62             amountDue = amountDue + item.getTotalPrice();
63          }
64          return amountDue;
65       }
66    }
```

section_3/LineItem.java

```
1  /**
2     Describes a quantity of an article to purchase.
3  */
4  public class LineItem
5  {
6     private int quantity;
7     private Product theProduct;
8
9     /**
10       Constructs an item from the product and quantity.
11       @param aProduct the product
12       @param aQuantity the item quantity
13    */
14    public LineItem(Product aProduct, int aQuantity)
15    {
16       theProduct = aProduct;
17       quantity = aQuantity;
18    }
19
```

```
20      /**
21         Computes the total cost of this line item.
22         @return the total price
23      */
24      public double getTotalPrice()
25      {
26         return theProduct.getPrice() * quantity;
27      }
28
29      /**
30         Formats this item.
31         @return a formatted string of this line item
32      */
33      public String format()
34      {
35         return String.format("%-30s%8.2f%5d%8.2f",
36            theProduct.getDescription(), theProduct.getPrice(),
37            quantity, getTotalPrice());
38      }
39   }
```

section_3/Product.java

```
1   /**
2      Describes a product with a description and a price.
3   */
4   public class Product
5   {
6      private String description;
7      private double price;
8
9      /**
10        Constructs a product from a description and a price.
11        @param aDescription the product description
12        @param aPrice the product price
13     */
14     public Product(String aDescription, double aPrice)
15     {
16        description = aDescription;
17        price = aPrice;
18     }
19
20     /**
21        Gets the product description.
22        @return the description
23     */
24     public String getDescription()
25     {
26        return description;
27     }
28
29     /**
30        Gets the product price.
31        @return the unit price
32     */
33     public double getPrice()
34     {
35        return price;
36     }
37  }
```

section_3/Address.java

```
1  /**
2     Describes a mailing address.
3  */
4  public class Address
5  {
6     private String name;
7     private String street;
8     private String city;
9     private String state;
10    private String zip;
11
12    /**
13       Constructs a mailing address.
14       @param aName the recipient name
15       @param aStreet the street
16       @param aCity the city
17       @param aState the two-letter state code
18       @param aZip the ZIP postal code
19    */
20    public Address(String aName, String aStreet,
21          String aCity, String aState, String aZip)
22    {
23       name = aName;
24       street = aStreet;
25       city = aCity;
26       state = aState;
27       zip = aZip;
28    }
29
30    /**
31       Formats the address.
32       @return the address as a string with three lines
33    */
34    public String format()
35    {
36       return name + "\n" + street + "\n"
37             + city + ", " + state + " " + zip;
38    }
39 }
```

13. Which class is responsible for computing the amount due? What are its collaborators for this task?
14. Why do the `format` methods return `String` objects instead of directly printing to `System.out`?

Practice It Now you can try these exercises at the end of the chapter: R12.18, P12.6, P12.7.

WORKED EXAMPLE 12.1 Simulating an Automatic Teller Machine

This Worked Example applies the object-oriented design methodology to the simulation of an automatic teller machine that works with both a console-based and graphical user interface.

Available online in WileyPLUS and at `www.wiley.com/college/horstmann`.

12.4 Packages

A Java program consists of a collection of classes. So far, most of your programs have consisted of a small number of classes. As programs get larger, however, simply distributing the classes over multiple files isn't enough. An additional structuring mechanism is needed.

A package is a set of related classes.

In Java, packages provide this structuring mechanism. A Java **package** is a set of related classes. For example, the Java library consists of dozens of packages, some of which are listed in Table 2. The following sections show how you can make use of packages in your programs.

Table 2 Important Packages in the Java Library

Package	Purpose	Sample Class
`java.lang`	Language support	`Math`
`java.util`	Utilities	`Scanner`
`java.io`	Input and output	`PrintStream`
`java.awt`	Abstract Windowing Toolkit	`Color`
`java.net`	Networking	`Socket`
`java.sql`	Database access through Structured Query Language	`ResultSet`
`javax.swing`	Swing user interface	`JButton`
`org.w3c.dom`	Document Object Model for XML documents	`Document`

12.4.1 Organizing Related Classes into Packages

To put a class in a package, you must place

```
package packageName;
```

as the first statement in its source file. A package name consists of one or more identifiers separated by periods. (See Section 12.4.3 for tips on constructing package names.)

For example, let's put a `BankAccount` class into a package named `com.horstmann`. The `BankAccount.java` file must start as follows:

```
package com.horstmann;

public class BankAccount
{
   . . .
}
```

In addition to the named packages (such as `java.util` or `com.horstmann`), there is a special package, called the *default package*, which has no name. If you did not include any `package` statement at the top of your source file, the class is placed in the default package.

In Java, related classes are grouped into packages.

12.4.2 Importing Packages

If you want to use a class from a package, you can refer to it by its full name (package name plus class name). For example, `java.util.Scanner` refers to the `Scanner` class in the `java.util` package:

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

The `import` directive lets you refer to a class from a package by its class name, without the package prefix.

Naturally, that is somewhat inconvenient. You can instead *import* a name with an `import` statement:

```
import java.util.Scanner;
```

Then you can refer to the class as `Scanner` without the package prefix.

You can import *all classes* of a package with an `import` statement that ends in `.*`. For example, you can use the statement

```
import java.util.*;
```

to import all classes from the `java.util` package. That statement lets you refer to classes like `Scanner` or `ArrayList` without a `java.util` prefix.

However, you never need to import the classes in the `java.lang` package explicitly. That is the package containing the most basic Java classes, such as `Math` and `Object`. These classes are always available to you. In effect, an automatic `import java.lang.*;` statement has been placed into every source file.

Finally, you don't need to import other classes in the same package. For example, in the source code of the class `problem1.Tester`, you don't need to import the class `problem1.BankAccount`. The compiler will find the `BankAccount` class without an `import` statement because it is located in the same package, `problem1`.

12.4.3 Package Names

Placing related classes into a package is clearly a convenient way to organize classes. However, there is a more important reason for packages: to avoid **name clashes**. In a large project, it is inevitable that two people will come up with the same name for the same concept. This even happens in the standard Java class library (which has now grown to thousands of classes). There is a class `Timer` in the `java.util` package and another class called `Timer` in the `javax.swing` package. You can still tell the Java compiler exactly which `Timer` class you need by referring to them as `java.util.Timer` and `javax.swing.Timer`.

Of course, for the package-naming convention to work, there must be some way to ensure that package names are unique. It wouldn't be good if the car maker BMW placed all its Java code into the package `bmw`, and some other programmer (perhaps Britney M. Walters) had the same bright idea. To avoid this problem, the inventors of Java recommend that you use a package-naming scheme that takes advantage of the uniqueness of Internet domain names.

Use a domain name in reverse to construct an unambiguous package name.

For example, I have a domain name `horstmann.com`, and there is nobody else on the planet with the same domain name. (I was lucky that the domain name `horstmann.com` had not been taken by anyone else when I applied. If your name is Walters, you will sadly find that someone else beat you to `walters.com`.) To get a package name, turn the domain name around to produce a package name prefix, such as `com.horstmann`.

If you don't have your own domain name, you can still create a package name that has a high probability of being unique by writing your e-mail address backwards. For example, if Britney Walters has an e-mail address `walters@cs.sjsu.edu`, then she can use a package name `edu.sjsu.cs.walters` for her own classes.

Some instructors will want you to place each of your assignments into a separate package, such as `problem1`, `problem2`, and so on. The reason is again to avoid name collision. You can have two classes, `problem1.BankAccount` and `problem2.BankAccount`, with slightly different properties.

12.4.4 How Classes Are Located

The path of a class file must match its package name.

A package is located in a subdirectory that matches the package name. For example, a package `homework1` is located in a directory `homework1`. If the package name has multiple parts, such as `com.horstmann.javabook`, then you use a subdirectory for each part: `com/horstmann/javabook`.

ONLINE EXAMPLE

The complete `BankAccount` and `BankAccountTester` classes, with the proper directory structure.

For example, if you do your homework assignment in a *base directory* `/home/britney/assignments`, then you can place the class files for the `problem1` package into the directory `/home/britney/assignments/problem1`, as shown in Figure 11. (Here, we are using UNIX-style file names. Under Windows, you would use a directory such as `c:\Users\Britney\assignments\problem1`.)

Figure 11
Base Directories and Subdirectories for Packages

15. Which of the following are packages?

a. `java`

b. `java.lang`

c. `java.util`

d. `java.lang.Math`

16. Is a Java program without `import` statements limited to using the default and `java.lang` packages?

17. Suppose your homework assignments are located in the directory `/home/me/cs101` (`c:\Users\me\cs101` on Windows). Your instructor tells you to place your homework into packages. In which directory do you place the class `hw1.problem1.TicTacToeTester`?

Practice It Now you can try these exercises at the end of the chapter: R12.19, P12.15, P12.16.

CHAPTER SUMMARY

Recognize how to discover classes and their responsibilities.

- To discover classes, look for nouns in the problem description.
- Concepts from the problem domain are good candidates for classes.
- A CRC card describes a class, its responsibilities, and its collaborating classes.
- The public interface of a class is cohesive if all of its features are related to the concept that the class represents.

Categorize class relationships and produce UML diagrams that describe them.

- A class depends on another class if it uses objects of that class.
- It is a good practice to minimize the coupling (i.e., dependency) between classes.
- A class aggregates another if its objects contain objects of the other class.
- Inheritance (the *is-a* relationship) is sometimes inappropriately used when the *has-a* relationship would be more appropriate.

- Aggregation (the *has-a* relationship) denotes that objects of one class contain references to objects of another class.
- You need to be able to distinguish the UML notation for inheritance, interface implementation, aggregation, and dependency.
- Avoid parallel arrays by changing them into arrays of objects.

Apply an object-oriented development process to designing a program.

- Start the development process by gathering and documenting program requirements.
- Use CRC cards to find classes, responsibilities, and collaborators.
- Use UML diagrams to record class relationships.
- Use `javadoc` comments (with the method bodies left blank) to record the behavior of classes.
- After completing the design, implement your classes.

Use packages to structure the classes in your program.

- A package is a set of related classes.
- The `import` directive lets you refer to a class from a package by its class name, without the package prefix.
- Use a domain name in reverse to construct an unambiguous package name.
- The path of a class file must match its package name.

REVIEW EXERCISES

R12.1 List the steps in the process of object-oriented design that this chapter recommends for student use.

R12.2 Give a rule of thumb for how to find classes when designing a program.

R12.3 Give a rule of thumb for how to find methods when designing a program.

R12.4 After discovering a method, why is it important to identify the object that is *responsible* for carrying out the action?

R12.5 Look at the public interface of the `java.lang.System` class and discuss whether or not it is cohesive.

R12.6 On which classes does the class `Integer` in the Java standard library depend?

R12.7 On which classes does the class `java.awt.Rectangle` in the standard library depend?

R12.8 What relationship is appropriate between the following classes: aggregation, inheritance, or neither?

a. `University–Student`
b. `Student–TeachingAssistant`
c. `Student–Freshman`
d. `Student–Professor`
e. `Car–Door`
f. `Truck–Vehicle`
g. `Traffic–TrafficSign`
h. `TrafficSign–Color`

R12.9 Every BMW is a vehicle. Should a class `BMW` inherit from the class `Vehicle`? BMW is a vehicle manufacturer. Does that mean that the class `BMW` should inherit from the class `VehicleManufacturer`?

R12.10 Some books on object-oriented programming recommend using inheritance so that the class `Circle` extends the class `java.awt.Point`. Then the `Circle` class inherits the `setLocation` method from the `Point` superclass. Explain why the `setLocation` method need not be overridden in the subclass. Why is it nevertheless not a good idea to have `Circle` inherit from `Point`? Conversely, would inheriting `Point` from `Circle` fulfill the *is-a* rule? Would it be a good idea?

R12.11 Write CRC cards for the `Coin` and `CashRegister` classes described in Section 12.1.3.

R12.12 Write CRC cards for the `Quiz` and `Question` classes in Section 12.2.2.

R12.13 Draw a UML diagram for the `Quiz`, `Question`, and `ChoiceQuestion` classes. The `Quiz` class is described in Section 12.2.2.

R12.14 A file contains a set of records describing countries. Each record consists of the name of the country, its population, and its area. Suppose your task is to write a program that reads in such a file and prints

- The country with the largest area
- The country with the largest population
- The country with the largest population density (people per square kilometer)

Think through the problems that you need to solve. What classes and methods will you need? Produce a set of CRC cards, a UML diagram, and a set of `javadoc` comments.

■■■ **R12.15** Discover classes and methods for generating a student report card that lists all classes, grades, and the grade point average for a semester. Produce a set of CRC cards, a UML diagram, and a set of `javadoc` comments.

■■ **R12.16** Consider the following problem description:

> **Users place coins in a vending machine and select a product by pushing a button. If the inserted coins are sufficient to cover the purchase price of the product, the product is dispensed and change is given. Otherwise, the inserted coins are returned to the user.**

What classes should you use to implement a solution?

■■ **R12.17** Consider the following problem description:

> **Employees receive their biweekly paychecks. They are paid their hourly rates for each hour worked; however, if they worked more than 40 hours per week, they are paid overtime at 150 percent of their regular wage.**

What classes should you use to implement a solution?

■■ **R12.18** Consider the following problem description:

> **Customers order products from a store. Invoices are generated to list the items and quantities ordered, payments received, and amounts still due. Products are shipped to the shipping address of the customer, and invoices are sent to the billing address.**

Draw a UML diagram showing the aggregation relationships between the classes `Invoice`, `Address`, `Customer`, and `Product`.

■■ **R12.19** Every Java program can be rewritten to avoid `import` statements. Explain how, and rewrite `BabyNames.java` from Worked Example 7.1 to avoid `import` statements.

■ **R12.20** What is the default package? Have you used it before this chapter in your programming?

PROGRAMMING EXERCISES

■■ **P12.1** Modify the `giveChange` method of the `CashRegister` class in the sample code for Section 12.1 so that it returns the number of coins of a particular type to return:

```
int giveChange(Coin coinType)
```

The caller needs to invoke this method for each coin type, in decreasing value.

■ **P12.2** Real cash registers can handle both bills and coins. Design a single class that expresses the commonality of these concepts. Redesign the `CashRegister` class and provide a method for entering payments that are described by your class. Your primary challenge is to come up with a good name for this class.

■ **P12.3** Enhance the invoice-printing program by providing for two kinds of line items: One kind describes products that are purchased in certain numerical quantities (such as "3 toasters"), another describes a fixed charge (such as "shipping: $5.00"). *Hint:* Use inheritance. Produce a UML diagram of your modified implementation.

■■ **P12.4** The invoice-printing program is somewhat unrealistic because the formatting of the `LineItem` objects won't lead to good visual results when the prices and quantities have varying numbers of digits. Enhance the `format` method in two ways: Accept an `int[]` array of column widths as an argument. Use the `NumberFormat` class to format the currency values.

■■ **P12.5** The invoice-printing program has an unfortunate flaw—it mixes "application logic" (the computation of total charges) and "presentation" (the visual appearance of the invoice). To appreciate this flaw, imagine the changes that would be necessary to draw the invoice in HTML for presentation on the Web. Reimplement the program, using a separate `InvoiceFormatter` class to format the invoice. That is, the `Invoice` and `LineItem` methods are no longer responsible for formatting. However, they will acquire other responsibilities, because the `InvoiceFormatter` class needs to query them for the values that it requires.

■■■ **P12.6** Write a program that teaches arithmetic to a young child. The program tests addition and subtraction. In level 1, it tests only addition of numbers less than 10 whose sum is less than 10. In level 2, it tests addition of arbitrary one-digit numbers. In level 3, it tests subtraction of one-digit numbers with a nonnegative difference.

Generate random problems and get the player's input. The player gets up to two tries per problem. Advance from one level to the next when the player has achieved a score of five points.

■■■ **P12.7** Implement a simple e-mail messaging system. A message has a recipient, a sender, and a message text. A mailbox can store messages. Supply a number of mailboxes for different users and a user interface for users to log in, send messages to other users, read their own messages, and log out. Follow the design process that was described in this chapter.

■■ **P12.8** Write a program that simulates a vending machine. Products can be purchased by inserting coins with a value at least equal to the cost of the product. A user selects a product from a list of available products, adds coins, and either gets the product or gets the coins returned. The coins are returned if insufficient money was supplied or if the product is sold out. The machine does not give change if too much money was added. Products can be restocked and money removed by an operator. Follow the design process that was described in this chapter. Your solution should include a class `VendingMachine` that is not coupled with the `Scanner` or `PrintStream` classes.

■■■ **P12.9** Write a program to design an appointment calendar. An appointment includes the date, starting time, ending time, and a description; for example,

```
Dentist 2012/10/1 17:30 18:30
CS1 class 2012/10/2 08:30 10:00
```

Supply a user interface to add appointments, remove canceled appointments, and print out a list of appointments for a particular day. Follow the design process that was described in this chapter. Your solution should include a class `AppointmentCalendar` that is not coupled with the `Scanner` or `PrintStream` classes.

■■ **P12.10** Modify the implementation of the classes in the ATM simulation in Worked Example 12.1 so that the bank manages a collection of bank accounts and a separate collection of customers. Allow joint accounts in which some accounts can have more than one customer.

■■■ **P12.11** Write a program that administers and grades quizzes. A quiz consists of questions. There are four types of questions: text questions, number questions, choice questions with a single answer, and choice questions with multiple answers. When grading a text question, ignore leading or trailing spaces and letter case. When grading a numeric question, accept a response that is approximately the same as the answer.

A quiz is specified in a text file. Each question starts with a letter indicating the question type (T, N, S, M), followed by a line containing the question text. The next line of a non-choice question contains the answer. Choice questions have a list of choices that is terminated by a blank line. Each choice starts with + (correct) or - (incorrect). Here is a sample file:

```
T
Which Java reserved word is used to declare a subclass?
extends
S
What is the original name of the Java language?
- *7
- C--
+ Oak
- Gosling

M
Which of the following types are supertypes of Rectangle?
- PrintStream
+ Shape
+ RectangularShape
+ Object
- String

N
What is the square root of 2?
1.41421356
```

Your program should read in a quiz file, prompt the user for responses to all questions, and grade the responses. Follow the design process that was described in this chapter.

■■ **P12.12** Produce a requirements document for a program that allows a company to send out personalized mailings, either by e-mail or through the postal service. Template files contain the message text, together with variable fields (such as Dear [Title] [Last Name] . . .). A database (stored as a text file) contains the field values for each recipient. Use HTML as the output file format. Then design and implement the program.

■■■ **P12.13** Write a tic-tac-toe game that allows a human player to play against the computer. Your program will play many turns against a human opponent, and it will learn. When it is the computer's turn, the computer randomly selects an empty field, except that it won't ever choose a losing combination. For that purpose, your program must keep an array of losing combinations. Whenever the human wins, the immediately preceding combination is stored as losing. For example, suppose that X = computer and O = human. Suppose the current combination is

O	X	X
	O	

Now it is the human's turn, who will of course choose

O	X	X
	O	
		O

The computer should then remember the preceding combination

O	X	X
	O	

as a losing combination. As a result, the computer will never again choose that combination from

O	X	
	O	

or

O		X
	O	

Discover classes and supply a UML diagram before you begin to program.

■ **P12.14** Place the `CashRegister` and `Coin` classes of the sample program in Section 12.1 into the package `com.horstmann`. Keep the `CashRegisterTester` class in the default package.

■ **P12.15** Place all classes of the sample program in Section 12.3 into the package `com.horstmann`. How do you start the program in your programming environment?

■ **P12.16** Place the classes from Worked Example 12.1 in a package whose name is derived from your e-mail address, as described in Section 12.4.3.

■■■ **Business P12.17** Implement a program that prints paychecks for a group of student assistants. Deduct federal income and Social Security taxes. (You may want to use the tax computation used in Chapter 3. Find out about Social Security taxes on the Internet.) Your program should prompt for the name, hourly wage, and hours worked for each student.

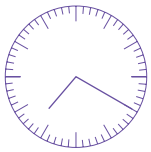
■■■ **Business P12.18** *Airline seating.* Write a program that assigns seats on an airplane. Assume the airplane has 20 seats in first class (5 rows of 4 seats each, separated by an aisle) and 90 seats in economy class (15 rows of 6 seats each, separated by an aisle). Your program should take three commands: add passengers, show seating, and quit. When passengers are added, ask for the class (first or economy), the number of passengers traveling together (1 or 2 in first class; 1 to 3 in economy), and the seating preference (aisle or window in first class; aisle, center, or window in economy). Then try to find a match and assign the seats. If no match exists, print a message. Your solution should include a class `Airplane` that is not coupled with the `Scanner` or `PrintStream` classes. Follow the design process that was described in this chapter.

■■■ **Business P12.19** In an airplane, each passenger has a touch screen for ordering a drink and a snack. Some items are free and some are not. The system prepares two reports for speeding up service:

1. A list of all seats, ordered by row, showing the charges that must be collected.
2. A list of how many drinks and snacks of each type must be prepared for the front and the rear of the plane.

Follow the design process that was described in this chapter to identify classes, and implement a program that simulates the system.

■■■ **Graphics P12.20** Implement a program to teach a young child to read the clock. In the game, present an analog clock, such as the one shown at left. Generate random times and display the clock. Accept guesses from the player. Reward the player for correct guesses. After two incorrect guesses, display the correct answer and make a new random time. Implement several levels of play. In level 1, only show full hours. In level 2, show quarter hours. In level 3, show five-minute multiples, and in level 4, show any number of minutes. After a player has achieved five correct guesses at one level, advance to the next level.

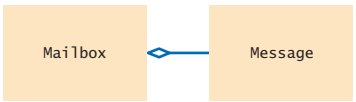


An Analog Clock

■■■ **Graphics P12.21** Write a program that can be used to design a suburban scene, with houses, streets, and cars. Users can add houses and cars of various colors to a street. Write more specific requirements that include a detailed description of the user interface. Then, discover classes and methods, provide UML diagrams, and implement your program.

■■■ **Graphics P12.22** Write a simple graphics editor that allows users to add a mixture of shapes (ellipses, rectangles, and lines in different colors) to a panel. Supply commands to load and save the picture. Discover classes, supply a UML diagram, and implement your program.

ANSWERS TO SELF-CHECK QUESTIONS

1. Look for nouns in the problem description.
2. Yes (`ChessBoard`) and no (`MovePiece`).
3. `PrintStream`
4. To produce the shipping address of the customer.
5. Reword the responsibilities so that they are at a higher level, or come up with more classes to handle the responsibilities.
6. The `CashRegisterTester` class depends on the `CashRegister` and `System` classes.
7. The `ChoiceQuestion` class inherits from the `Question` class.
8. The `Quiz` class depends on the `Question` class but probably not `ChoiceQuestion`, if we assume that the methods of the `Quiz` class manipulate generic `Question` objects, as they did in Chapter 9.
9. If a class doesn't depend on another, it is not affected by interface changes in the other class.
10. 

11. Typically, a library system wants to track which books a patron has checked out, so it makes more sense to have `Patron` aggregate `Book`. However, there is not always one true answer in design. If you feel strongly that it is important to identify the patron who checked out a particular book (perhaps to notify the patron to return it because it was requested by someone else), then you can argue that the aggregation should go the other way around.
12. There would be no relationship.

13. The `Invoice` class is responsible for computing the amount due. It collaborates with the `LineItem` class.
14. This design decision reduces coupling. It enables us to reuse the classes when we want to show the invoice in a dialog box or on a web page.
15. (a) No; (b) Yes; (c) Yes; (d) No
16. No—you simply use fully qualified names for all other classes, such as `java.util.Random` and `java.awt.Rectangle`.
17. `/home/me/cs101/hw1/problem1` or, on Windows, `c:\Users\me\cs101\hw1\problem1`.