Chapter 12 Object-Oriented Design

CHAPTER GOALS

- To learn about the software life cycle
- To learn how to discover new classes and methods
- To understand the use of CRC cards for class discovery
- To be able to identify inheritance, aggregation, and dependency relationships between classes
- To master the use of UML class diagrams to describe class relationships
- To learn how to use object-oriented design to build complex programs

To implement a software system successfully, be it as simple as your next homework project or as complex as the next air traffic monitoring system, some amount of planning, design, and testing is required. In fact, for larger projects, the amount of time spent on planning is much higher than the amount of time spent on programming and testing.

If you find that most of your homework time is spent in front of the computer, keying in code and fixing bugs, you are probably spending more time on your homework than you should. You could cut down your total time by spending more on the planning and design phase. This chapter tells you how to approach these tasks in a systematic manner, using the object-oriented design methodology.

12.1 The Software Life Cycle

In this section we will discuss the *software life cycle*: the activities that take place between the time a software program is first conceived and the time it is finally retired.

The life cycle of software encompasses all activities from initial analysis until obsolescence.

A software project usually starts because a customer has a problem and is willing to pay money to have it solved. The Department of Defense, the customer of many programming projects, was an early proponent of a *formal process* for software development. A formal process identifies and describes different phases and gives guidelines for carrying out the phases and when to move from one phase to the next.

A formal process for software development describes phases of the development process and gives guidelines for how to carry out the phases.

Many software engineers break the development process down into the following five phases:

- Analysis
- Design
- Implementation
- Testing
- Deployment

In the *analysis* phase, you decide *what* the project is supposed to accomplish; you do not think about *how* the program will accomplish its tasks. The output of the analysis phase is a *requirements document*, which describes in complete detail what the program will be able to do once it is completed. Part of this requirements document can be a user manual that tells how the user will operate the program to derive the promised benefits. Another part sets performance criteria—how many inputs the program must be able to handle in what time, or what its maximum memory and disk storage requirements are. 530

531

In the *design* phase, you develop a plan for how you will implement the system. You discover the structures that underlie the problem to be solved. When you use object-oriented design, you decide what classes you need and what their most important methods are. The output of this phase is a description of the classes and methods, with diagrams that show the relationships among the classes.

In the *implementation* phase, you write and compile program code to implement the classes and methods that were discovered in the design phase. The output of this phase is the completed program.

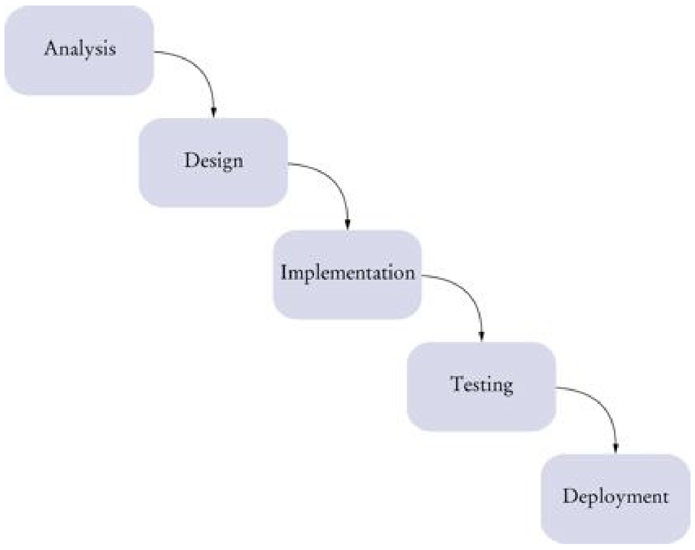
In the *testing* phase, you run tests to verify that the program works correctly. The output of this phase is a report describing the tests that you carried out and their results.

In the *deployment* phase, the users of the program install it and use it for its intended purpose.

When formal development processes were first established in the early 1970s, software engineers had a very simple visual model of these phases. They postulated that one phase would run to completion, its output would spill over to the next phase, and the next phase would begin. This model is called the *waterfall model* of software development (see Figure 1).

The waterfall model of software development describes a sequential process of analysis, design, implementation, testing, and deployment.

Figure 1

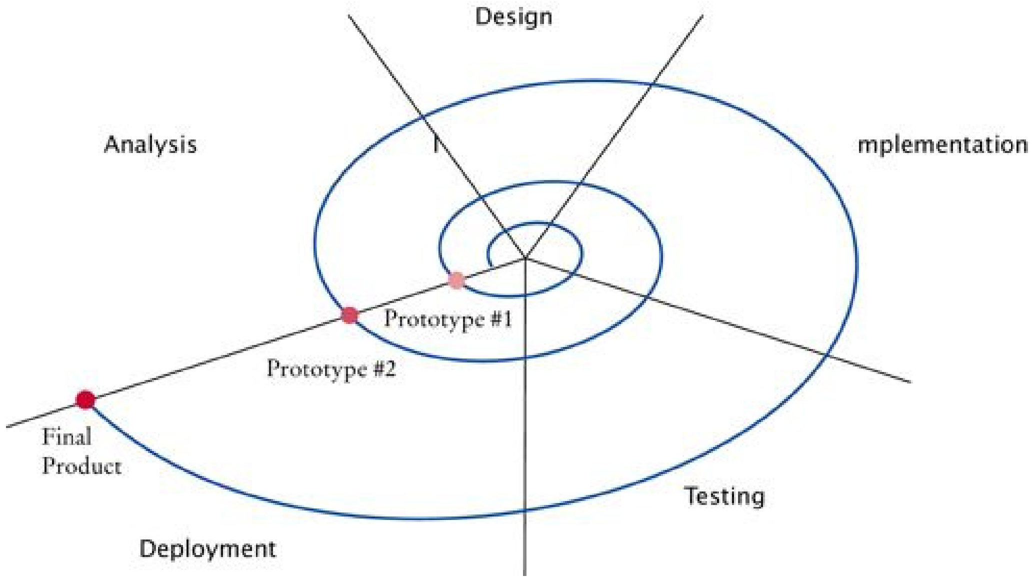


The Waterfall Model

In an ideal world the waterfall model has a lot of appeal: You figure out what to do; then you figure out how to do it; then you do it; then you verify that you did it right; then you hand the product to the customer. When rigidly applied, though, the waterfall model simply did not work. It was very difficult to come up with a perfect 531
requirement specification. It was quite common to discover in the design phase that 532
the requirements were inconsistent or that a small change in the requirements would lead to a system that was both easier to design and more useful for the customer, but the analysis phase was over, so the designers had no choice—they had to take the existing requirements, errors and all. This problem would repeat itself during implementation. The designers may have thought they knew how to solve the problem as efficiently as possible, but when the design was actually implemented, it turned out that the resulting program was not as fast as the designers had thought. The next transition is one with which you are surely familiar. When the program was handed to the quality assurance department for testing, many bugs were found that would best be fixed by reimplementing, or maybe even redesigning, the program, but the waterfall model did not allow for this. Finally, when the customers received the finished product, they were often not at all happy with it. Even though the customers typically were very involved in the analysis phase, often they themselves were not sure exactly what they needed. After all, it can be very difficult to describe how you want to use a product that you have never seen before. But when the customers started using the program, they began to realize what they would have liked. Of course, then it was too late, and they had to live with what they got.

The spiral model of software development describes an iterative process in which design and implementation are repeated.

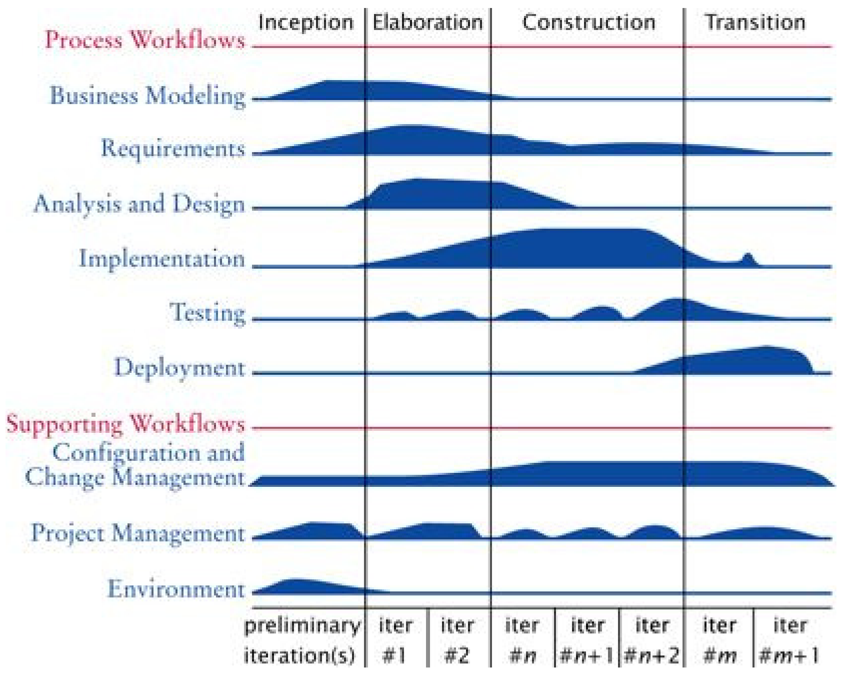
Figure 2



A Spiral Model

Having some level of iteration is clearly necessary. There simply must be a mechanism to deal with errors from the preceding phase. A *spiral model*, originally proposed by Barry Boehm in 1988, breaks the development process down into multiple phases (see Figure 2). Early phases focus on the construction of *prototypes*. A prototype is a small system that shows some aspects of the final system. Because 532 prototypes model only a part of a system and do not need to withstand 533 customer abuse, they can be implemented quickly. It is common to build a *user interface prototype* that shows the user interface in action. This gives customers an early chance to become more familiar with the system and to suggest improvements before the analysis is complete. Other prototypes can be built to validate interfaces with external systems, to test performance, and so on. Lessons learned from the development of one prototype can be applied to the next iteration of the spiral.

Figure 3



Activity Levels in the Rational Unified Process Methodology [1]

By building in repeated trials and feedback, a development process that follows the spiral model has a greater chance of delivering a satisfactory system. However, there is also a danger. If engineers believe that they don't have to do a good job because they can always do another iteration, then there will be many iterations, and the process will take a very long time to complete.

Extreme Programming is a development methodology that strives for simplicity by removing formal structure and focusing on best practices.

Figure 3 shows activity levels in the “Rational Unified Process”, a development process methodology by the inventors of UML. The details are not important, but as you can see, this is a complex process involving multiple iterations.

Even complex development processes with many iterations have not always met with success. In 1999, Kent Beck published an influential book [2] on *Extreme*

Programming, a development methodology that strives for simplicity by cutting out most of the formal trappings of a traditional development methodolgy and instead focusing on a set of *practices*:

- *Realistic planning:* Customers are to make business decisions, programmers are to make technical decisions. Update the plan when it conflicts with reality.
- *Small releases:* Release a useful system quickly, then release updates on a very short cycle.

- *Metaphor:* All programmers should have a simple shared story that explains the system under development.
- *Simplicity:* Design everything to be as simple as possible instead of preparing for future complexity.
- *Testing:* Both programmers and customers are to write test cases. The system is continuously tested.
- *Refactoring:* Programmers are to restructure the system continuously to improve the code and eliminate duplication.
- *Pair programming:* Put programmers together in pairs, and require each pair to write code on a single computer.
- *Collective ownership:* All programmers have permission to change all code as it becomes necessary.
- *Continuous integration:* Whenever a task is completed, build the entire system and test it.
- *40-hour week:* Don't cover up unrealistic schedules with bursts of heroic effort.
- *On-site customer:* An actual customer of the system is to be accessible to team members at all times.
- *Coding standards:* Programmers are to follow standards that emphasize self-documenting code.

Many of these practices are common sense. Others, such as the pair programming requirement, are surprising. Beck claims that the value of the Extreme Programming approach lies in the synergy of these practices—the sum is bigger than the parts.

In your first programming course, you will not develop systems that are so complex that you need a full-fledged methodology to solve your homework problems. This introduction to the development process should, however, show you that successful software development involves more than just coding. In the remainder of this chapter we will have a closer look at the *design phase* of the software development process.

Self Check

1. Suppose you sign a contract, promising that you will, for an agreed-upon price, design, implement, and test a software package exactly as it has been specified in a requirements document. What is the primary risk you and your customer are facing with this business arrangement?
2. Does Extreme Programming follow a waterfall or a spiral model?
3. What is the purpose of the "on-site customer" in Extreme Programming? 534

535

Random Fact 12.1: Programmer Productivity

If you talk to your friends in this programming class, you will find that some of them consistently complete their assignments much more quickly than others. Perhaps they have more experience. However, even when programmers with the same education and experience are compared, wide variations in competence are routinely observed and measured. It is not uncommon to have the best programmer in a team be five to ten times as productive as the worst, using any of a number of reasonable measures of productivity [3].

That is a staggering range of performance among trained professionals. In a marathon race, the best runner will not run five to ten times faster than the slowest one. Software product managers are acutely aware of these disparities. The obvious solution is, of course, to hire only the best programmers, but even in recent periods of economic slowdown the demand for good programmers has greatly outstripped the supply.

Fortunately for all of us, joining the rank of the best is not necessarily a question of raw intellectual power. Good judgment, experience, broad knowledge, attention to detail, and superior planning are at least as important as mental brilliance. These skills can be acquired by individuals who are genuinely interested in improving themselves.

Even the most gifted programmer can deal with only a finite number of details in a given time period. Suppose a programmer can implement and debug one method every two hours, or one hundred methods per month. (This is a generous estimate. Few programmers are this productive.) If a task requires 10,000 methods (which is typical for a medium-sized program), then a single programmer would need 100 months to complete the job. Such a project is sometimes expressed as a "100-man-month" project. But as Fred Brooks explains in his famous book [4], the concept of "man-month" is a myth. One cannot trade months for programmers. One hundred programmers cannot finish the task in one month. In fact, 10 programmers probably couldn't finish it in 10 months. First of all, the 10 programmers need to learn about the project before they can get productive. Whenever there is a problem with a particular method, both the author and its users need to meet and discuss it, taking time away from all of them. A bug in one method may have other programmers twiddling their thumbs until it is fixed.

It is difficult to estimate these inevitable delays. They are one reason why software is often released later than originally promised. What is a manager to do when the delays mount? As Brooks points out, adding more personnel will make a late project even later, because the productive people have to stop working and train the newcomers.

You will experience these problems when you work on your first team project with other students. Be prepared for a major drop in productivity, and be sure to set ample time aside for team communications.

There is, however, no alternative to teamwork. Most important and worthwhile projects transcend the ability of one single individual. Learning to function well in a team is just as important as becoming a competent programmer.

12.2 Discovering Classes

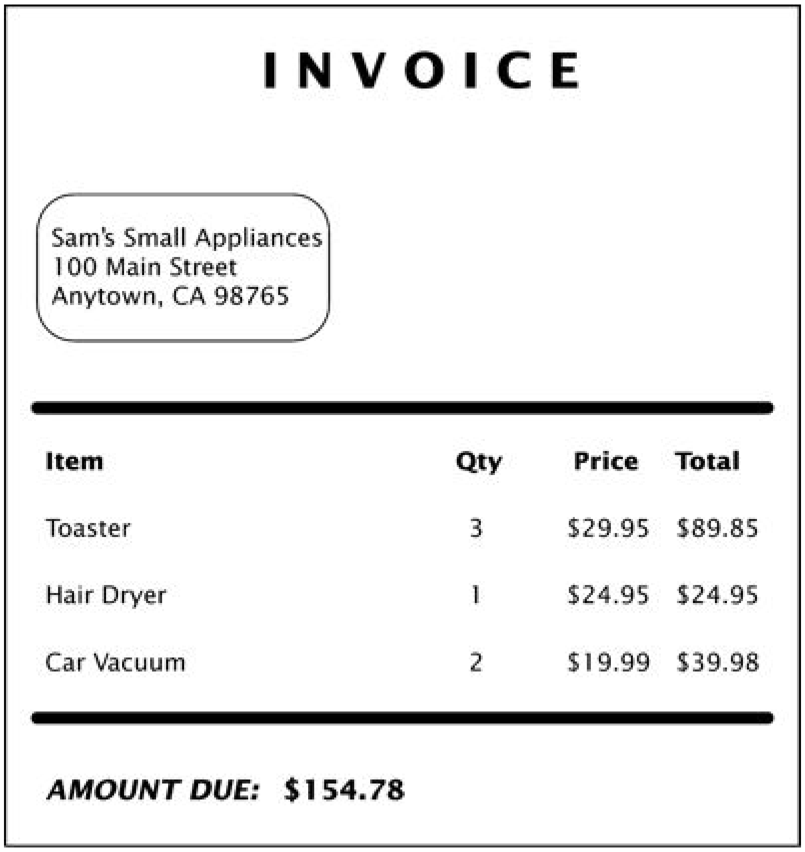
In the design phase of software development, your task is to discover structures that make it possible to implement a set of tasks on a computer. When you use the object-oriented design process, you carry out the following tasks:

1. Discover classes.
2. Determine the responsibilities of each class.
3. Describe the relationships between the classes.

In object-oriented design, you discover classes, determine the responsibilities of classes, and describe the relationships between classes.

A class represents some useful concept. You have seen classes for concrete entities, such as bank accounts, ellipses, and products. Other classes represent abstract concepts, such as streams and windows. A simple rule for finding classes is to look for *nouns* in the task description. For example, suppose your job is to print an invoice such as the one in Figure 4. 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.

Figure 4



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**

An Invoice

When finding classes, keep the following points in mind:

- A class represents a set of objects with the same behavior. Entities with multiple occurrences in your problem description, such as customers or products, are good candidates for objects. Find out what they have in common, and design classes to capture those commonalities.
- Some entities should be represented as objects, others as primitive types. 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 not too limiting or excessively general.

- Not all classes can be discovered in the analysis phase. 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.

Once a set of classes has been identified, you need to define the behavior for each class. That is, you need to find out what methods each object needs to carry out to solve the programming problem. A simple rule for finding these methods is to look for *verbs* in the task description, and 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.

An excellent way to carry out this task is the "CRC card method." *CRC* stands for "*c*lasses", "*r*esponsibilities", "*c*ollaborators", and in its simplest form, the method works as follows. Use an index card for each *class* (see Figure 5). 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*.

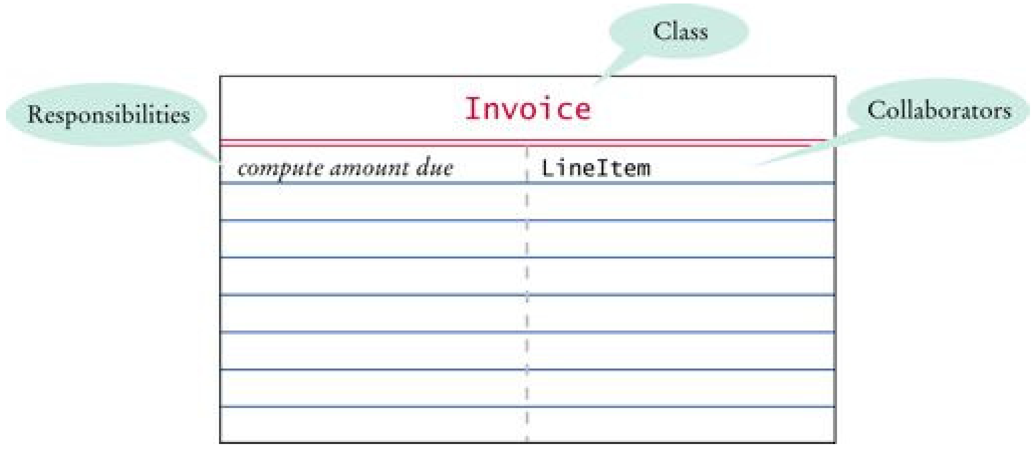
A CRC card describes a class, its responsibilities, and its collaborating classes.

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 5



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 the next section.

SELF CHECK

4. Suppose the invoice is to be saved to a file. Name a likely collaborator.
5. Looking at the invoice in Figure 4, what is a likely responsibility of the `Customer` class?
6. What do you do if a CRC card has ten responsibilities?

12.3 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.

You have seen the inheritance relationship between classes many times in this book. Inheritance is a very important relationship, but, as it turns out, it is not the only useful relationship, and it can be overused.

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. Every circle is an ellipse (with equal width and height).

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

Inheritance is sometimes abused, however. 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. Java lets us model that relationship, too. Use an instance field:

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

The technical term for this relationship is *aggregation*. Each `Tire` aggregates a `Circle` object. In general, a class aggregates another class if its objects have objects of the other class.

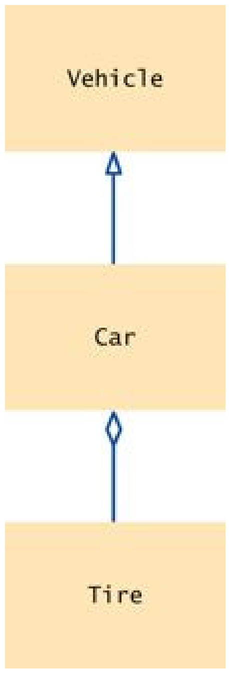
Aggregation (the *has-a* relationship) denotes that objects of one class contain references to objects of another class.

Here is another example. Every car *is a* vehicle. Every car *has a* tire (in fact, it 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;
}
```

539

Figure 6



UML Notation for Inheritance and Aggregation

In this book, we use the UML notation for class diagrams. You have already seen many examples of the UML notation for inheritance—an arrow with an open triangle pointing to the superclass. In the UML notation, aggregation is denoted by a solid line with a diamond-shaped symbol next to the aggregating class. Figure 6 shows a class diagram with an inheritance and an aggregation relationship.

The aggregation relationship is related to the *dependency* relationship, which you saw in Chapter 8. Recall that a class depends on another if one of its methods *uses* an object of the other class in some way.

Dependency is another name for the "uses" relationship.

For example, many of our applications depend on the `Scanner` class, because they use a `Scanner` object to read input.

Aggregation is a stronger form of dependency. If a class has objects of another class, it certainly uses the other class. However, the converse is not true. For example, a class may use the `Scanner` class without ever defining an instance field of class `Scanner`. The class may simply construct a local variable of type `Scanner`, or its methods may receive `Scanner` objects as parameters. 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*.

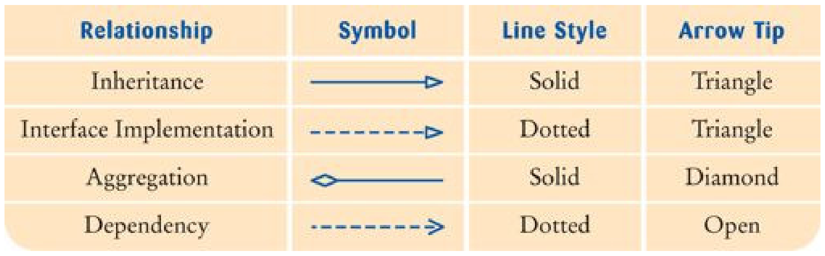
You need to be able to distinguish the UML notations for inheritance, interface implementation, aggregation, and dependency.

As you saw in Chapter 8, the UML notation for dependency is a dashed line with an open arrow that points to the dependent class.

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. 540

541

Table 1 UML Relationship Symbols



Relationship	Symbol	Line Style	Arrow Tip
Inheritance	——————▷	Solid	Triangle
Interface Implementation	- - - - - - - -▷	Dotted	Triangle
Aggregation	◇——————	Solid	Diamond
Dependency	- - - - - - - ->	Dotted	Open

Self Check

7. Consider the `Bank` and `BankAccount` classes of Chapter 7. How are they related?

8. Consider the `BankAccount` and `SavingsAccount` objects of Chapter 10. How are they related?

9. Consider the `BankAccountTester` class of Chapter 3. Which classes does it depend on?

How To 12.1: CRC Cards and UML Diagrams

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:

1. Discover classes.
2. Determine the responsibilities of each class.
3. 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 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 cards tell you which classes use others. Draw usage arrows for the collaborators on the CRC cards.

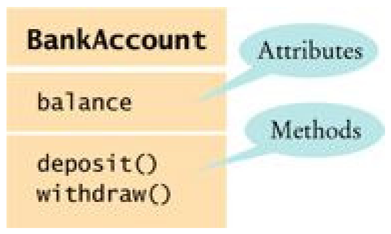
Some dependency relationships give rise to aggregations. For each of the dependency relationships, ask yourself: How does the object locate its collaborator? Does it navigate to it directly because it stores a reference? In that case, draw an aggregation arrow. Or is the collaborator a method parameter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

ADVANCED 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 fields 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 Attributes and Methods in a Class Diagram). You need not list *all* attributes and methods in a particular diagram. Just list the ones that are helpful to understand 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`.



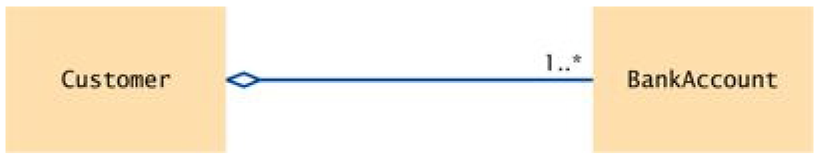
Attributes and Methods in a Class Diagram

ADVANCED 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.



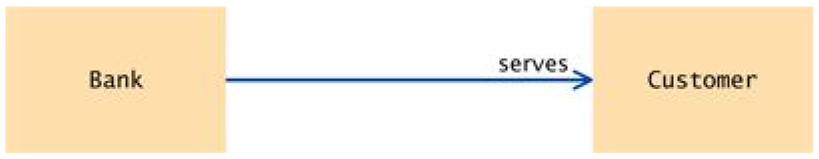
An Aggregation Relationship with Multiplicities

Advanced Topic 12.3: Aggregation and Association

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 field, 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. An Association Relationship 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

543

544

Frankly, the differences between aggregation and association are 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 manage objects of another class. The aggregation or *has-a* relationship accurately describes this phenomenon.

12.4 Case Study: Printing an Invoice

In this chapter, we discuss a five-part 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* [1] 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 example that follows.

12.4.1 Requirements

The task of this 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. 544

```
                  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.

12.4.2 CRC Cards

First, you need to discover classes. Classes correspond to nouns in the requirements description. 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 fields 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.

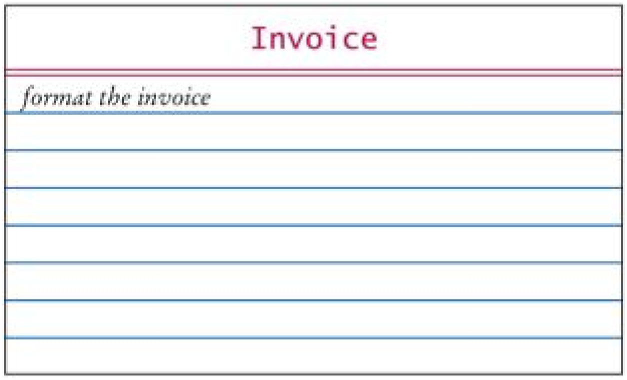
After this process of elimination, we are left with four candidates for classes:

```
Invoice
Address
LineItem
Product
```

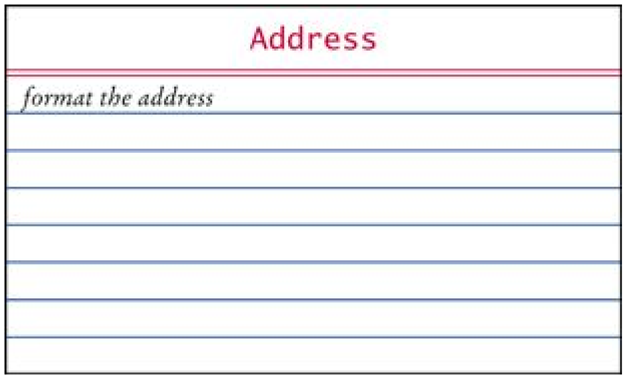
Each of them represents a useful concept, so let's make them all into classes.

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: 545

546

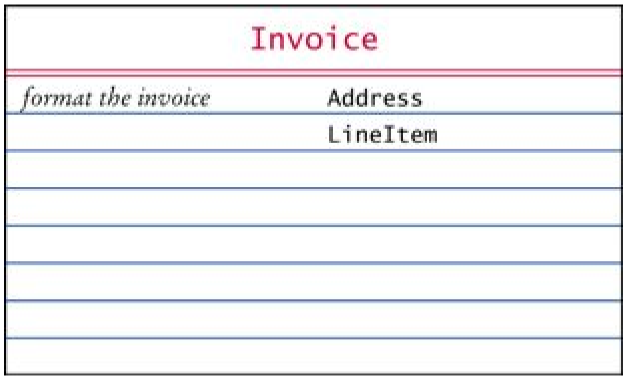


How does an invoice format itself? It must format the billing address, format all line items, and then add the amount due. How can the invoice format an address? It can't—that really is the responsibility of the `Address` class. This leads to a second CRC card:



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`:

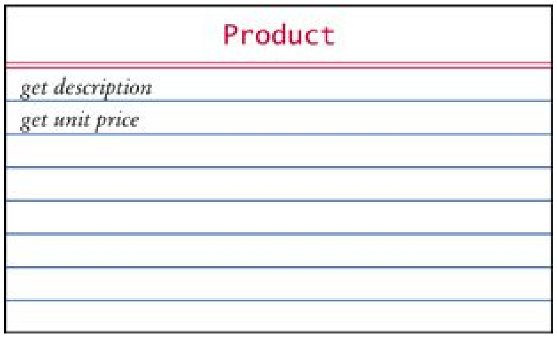


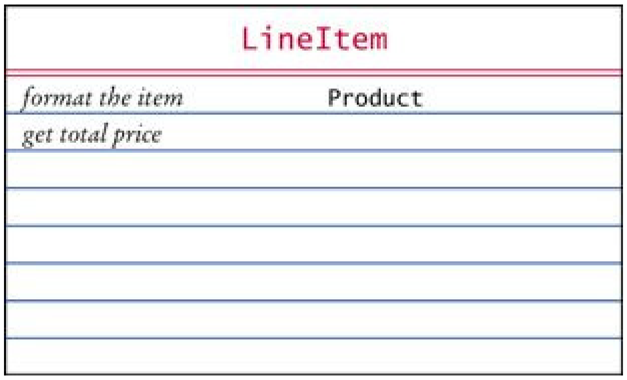
When formatting the invoice, the invoice also needs to compute the total amount due. To obtain that amount, it must ask each line item about the total price of the item.

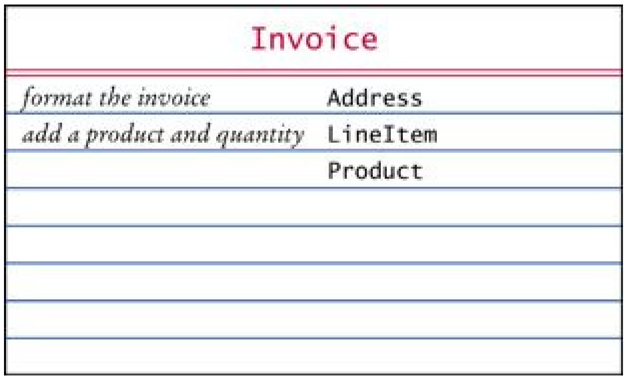
How does a line item obtain that total? It must ask the product for the unit price, and then multiply it by the quantity. That is, the `Product` class must reveal the unit price, and it is a collaborator of the `LineItem` class.

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.

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







12.4.3 UML Diagrams

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.

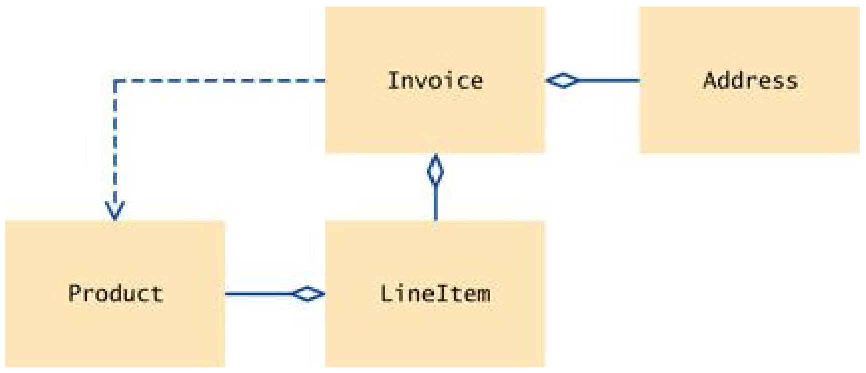
Now ask yourself which of these dependencies are actually aggregations. How does an invoice know about the address, line item, and product objects with which it collaborates? An invoice object must hold references to the address and the line items when it formats the invoice. But an invoice object need 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 7 shows the class relationships that we discovered.

Figure 7



The Relationships Between the Invoice Classes

12.4.4 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
{
```
548

549
```
/**
        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 and its price.
      */
      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 line 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()
      {
      }
}
```

Then run the `javadoc` program to obtain a prettily formatted version of your documentation in HTML format (see Figure 8).

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 of the key methods—a task that less prepared programmers leave for later, and then often neglect for lack of time.

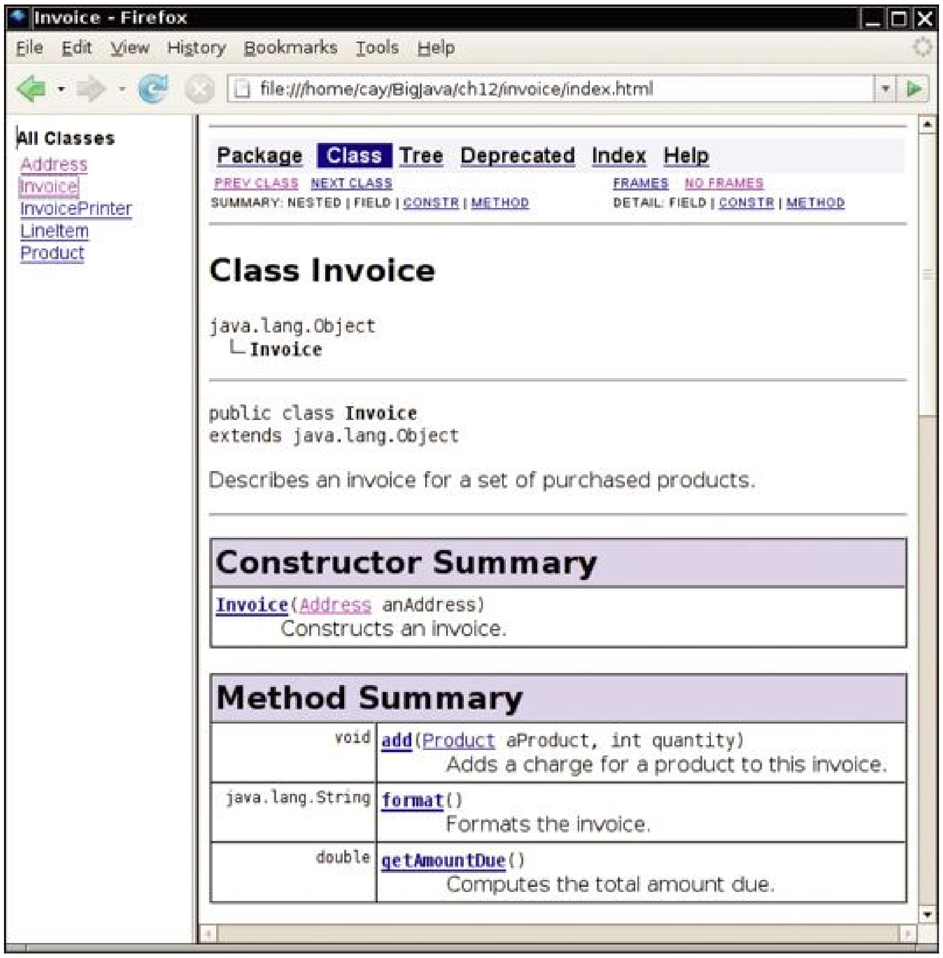
12.4.5 Implementation

Finally, you are ready to implement the classes.

You already have the method signatures and comments from the previous step. Now look at the UML diagram to add instance fields. Aggregated classes yield instance fields. 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 fields of the `Invoice` class:

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

Figure 8


Invoice - Firefox
File Edit View History Bookmarks Tools Help
file:///home/cay/BigJava/ch12/invoice/index.html
All Classes
Address
Invoice
InvoicePrinter
LineItem
Product
Package Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS
FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD

Class Invoice

```
java.lang.Object
  └Invoice
```

```
public class Invoice
extends java.lang.Object
```

Describes an invoice for a set of purchased products.

Constructor Summary
`Invoice(Address anAddress)` Constructs an invoice.

Method Summary	
`void`	`add(Product aProduct, int quantity)` Adds a charge for a product to this invoice.
`java.lang.String`	`format()` Formats the invoice.
`double`	`getAmountDue()` Computes the total amount due.

The Class Documentation in HTML Format

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

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

The methods themselves are now very easy. 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.

Here is the entire program. 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.

ch12/invoice/InvoicePrinter.java

```
/**
    This program demonstrates the invoice classes by
    printing a sample invoice.
*/
public class InvoicePrinter
{
      public static void main(String[] args)
      {
            Address samsAddress
```

```java
                        = new Address("Sam's Small Appliances",
                        "100 Main Street", "Anytown", "CA", "98765");

            Invoice samsInvoice = new Invoice(samsAddress);
            samsInvoice.add(new Product("Toaster", 29.95), 3);
            samsInvoice.add(new Product("Hair dryer", 24.95), 1);
            samsInvoice.add(new Product("Car vacuum", 19.99), 2);

            System.out.println(samsInvoice.format());
      }
}
```

ch12/invoice/Invoice.java

```java
import java.util.ArrayList;

/**
   Describes an invoice for a set of purchased products.
*/
public class Invoice
{
      /**
         Constructs an invoice.
         @param anAddress the billing address
      */
      public Invoice(Address anAddress)
      {
```

552

553

```java
            items = new ArrayList<LineItem>();
            billingAddress = anAddress;
      }

      /**
         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)
   {
      LineItem anItem = new LineItem(aProduct, quantity);
      items.add(anItem);
   }

   /**
      Formats the invoice.
      @return the formatted invoice
   */
   public String format()
   {
      String r = “                    I N V O I C E\n\n”
            + billingAddress.format()
            + String.format(“\n\n%'-30s%'8s%'5s%'8s\n”,
            “Description”, “Price”, “Qty”, “Total”);

      for (LineItem i : items)
      {
         r = r + i.format() + “\n”;
      }

      r = r + String.format(“\nAMOUNT DUE: %$'8.2f”, getAmountDue());

      return r;
   }

   /**
      Computes the total amount due.
      @return the amount due
   */
   public double getAmountDue()
```

```
    {
        double amountDue = 0;
        for (LineItem i : items)
        {
            amountDue = amountDue + i.getTotalPrice();
        }
        return amountDue;
    }

    private Address billingAddress;
    private ArrayList<LineItem> items;
}
```

553

554

ch12/invoice/LineItem.java

```
/**
   Describes a quantity of an article to purchase.
*/
public class LineItem
{
      /**
         Constructs an item from the product and quantity.
         @param aProduct the product
         @param aQuantity the item quantity
    */
    public LineItem(Product aProduct, int aQuantity)
    {
        theProduct = aProduct;
        quantity = aQuantity;
    }

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

    /**
       Formats this item.
       @return a formatted string of this line item
    */
    public String format()
    {
          return String.format(“%'-30s%'8.2f%'5d%'8.2f”,
                theProduct.getDescription(), theProduct.getPrice(),
                quantity, getTotalPrice());
    }

    private int quantity;
    private Product theProduct;
}
```

ch12/invoice/Product.java

```
/**
   Describes a product with a description and a price.
*/
public class Product
{
      /**
         Constructs a product from a description and a price.
         @param aDescription the product description
         @param aPrice the product price
     */
     public Product(String aDescription, double aPrice)
```

554

555

```
     {
           description = aDescription;
           price = aPrice;
      }

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

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

   private String description;
   private double price;
}
```

ch12/invoice/Address.java

```
/**
   Describes a mailing address.
*/
public class Address
{
   /**
      Constructs a mailing address.
      @param aName the recipient name
      @param aStreet the street
      @param aCity the city
      @param aState the two-letter state code
      @param aZip the ZIP postal code
   */
   public Address(String aName, String aStreet,
         String aCity, String aState, String aZip)
   {
      name = aName;
```

```
         street = aStreet;
         city = aCity;
         state = aState;
         zip = aZip;
   }

   /**
      Formats the address.
      @return the address as a string with three lines
   */
   public String format()
   {
         return name + "\n" + street + "\n"
                   + city + ", " + state + " " + zip;
   }

   private String name;
   private String street;
   private String city;
   private String state;
   private String zip;
}
```

SELF CHECK

10. Which class is responsible for computing the amount due? What are its collaborators for this task?

11. Why do the `format` methods return `String` objects instead of directly printing to `System.out`?

12.5 Case Study: An Automatic Teller Machine

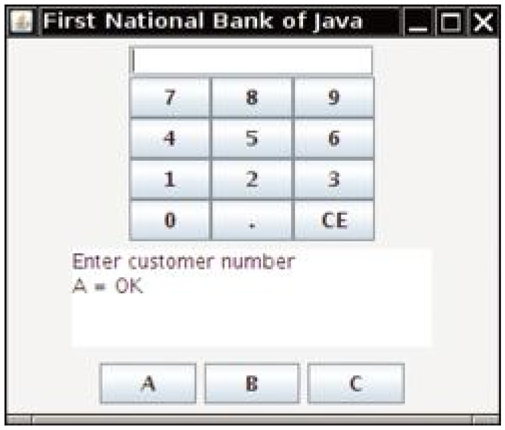
12.5.1 Requirements

The purpose of this project is to design a simulation of an automatic teller machine (ATM). The ATM is used by the customers of a bank. Each customer has two accounts: a checking account and a savings account. Each customer also has a customer number and a personal identification number (PIN); both are required to gain access to the accounts. (In a real ATM, the customer number would be recorded on the magnetic strip of the ATM card. In this simulation, the customer will need to type it in.) With the ATM, customers can select an account (checking or savings). The balance of the selected account is displayed. Then the customer can deposit and withdraw money. This process is repeated until the customer chooses to exit.

The details of the user interaction depend on the user interface that we choose for the simulation. We will develop two separate interfaces: a graphical interface that closely mimics an actual ATM (see Figure 9), and a text-based interface that allows you to test the ATM and bank classes without being distracted by GUI programming.

In the GUI interface, the ATM has a keypad to enter numbers, a display to show messages, and a set of buttons, labeled A, B, and C, whose function depends on the state of the machine.

Figure 9



Enter customer number
A = OK

Graphical User Interface for the Automatic Teller Machine

Specifically, the user interaction is as follows. When the ATM starts up, it expects a user to enter a customer number. The display shows the following message:

```
Enter customer number
A = OK
```

The user enters the customer number on the keypad and presses the A button. The display message changes to

```
Enter PIN
A = OK
```

Next, the user enters the PIN and presses the A button again. If the customer number and ID match those of one of the customers in the bank, then the customer can proceed. If not, the user is again prompted to enter the customer number.

If the customer has been authorized to use the system, then the display message changes to

```
Select Account
A = Checking
B = Savings
C = Exit
```

If the user presses the C button, the ATM reverts to its original state and asks the next user to enter a customer number.

If the user presses the A or B buttons, the ATM remembers the selected account, and the display message changes to

```
Balance = balance of selected account
Enter amount and select transaction
A = Withdraw
B = Deposit
C = Cancel
```

If the user presses the A or B buttons, the value entered in the keypad is withdrawn from or deposited into the selected account. (This is just a simulation, so no money is dispensed and no deposit is accepted.) Afterwards, the ATM reverts to the preceding state, allowing the user to select another account or to exit.

If the user presses the C button, the ATM reverts to the preceding state without executing any transaction. 557

558

In the text-based interaction, we read input from `System.in` instead of the buttons. Here is a typical dialog:

```
Enter account number: 1
Enter PIN: 1234
A=Checking, B=Savings, C=Quit: A
Balance=0.0
A=Deposit, B=Withdrawal, C=Cancel: A
Amount: 1000
A=Checking, B=Savings, C=Quit: C
```

In our solution, only the user interface classes are affected by the choice of user interface. The remainder of the classes can be used for both solutions—they are decoupled from the user interface.

Because this is a simulation, the ATM does not actually communicate with a bank. It simply loads a set of customer numbers and PINs from a file. All accounts are initialized with a zero balance.

12.5.2 CRC Cards

We will again follow the recipe of Section 12.2 and show how to discover classes, responsibilities, and relationships and how to obtain a detailed design for the ATM program.

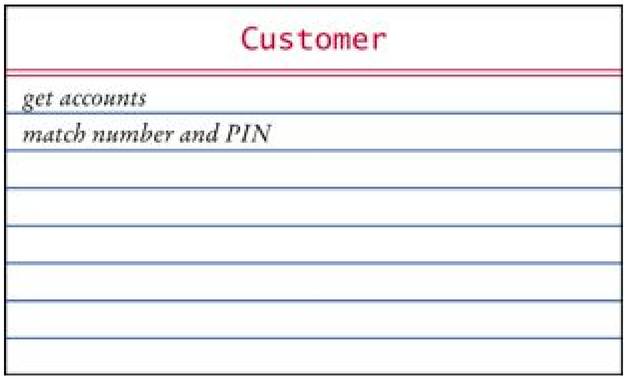
Recall that the first rule for finding classes is "Look for nouns in the problem description". Here is a list of the nouns:

```
ATM
User
Keypad
Display
Display message
Button
State
Bank account
Checking account
Savings account
Customer
Customer number
PIN
Bank
```

Of course, not all of these nouns will become names of classes, and we may yet discover the need for classes that aren't in this list, but it is a good start.

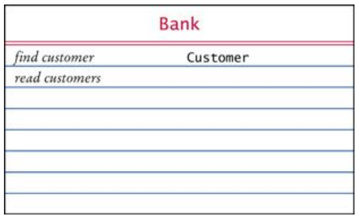
Users and customers represent the same concept in this program. Let's use a class `Customer`. A customer has two bank accounts, and we will require that a `Customer` object should be able to locate these accounts. (Another possible design would make the `Bank` class responsible for locating the accounts of a given customer—see Exercise P12.9.)

A customer also has a customer number and a PIN. We can, of course, require that a customer object give us the customer number and the PIN. But perhaps that isn't 558 so secure. Instead, simply require that a customer object, when given a customer 559 number and a PIN, will tell us whether it matches its own information or not.



A bank contains a collection of customers. When a user walks up to the ATM and enters a customer number and PIN, it is the job of the bank to find the matching customer. How can the bank do this? It needs to check for each customer whether its customer number and PIN match. Thus, it needs to call the *match number and PIN* method of the `Customer` class that we just discovered. Because the *find customer* method calls a `Customer` method, it collaborates with the `Customer` class. We record that fact in the right-hand column of the CRC card.

When the simulation starts up, the bank must also be able to read account information from a file.



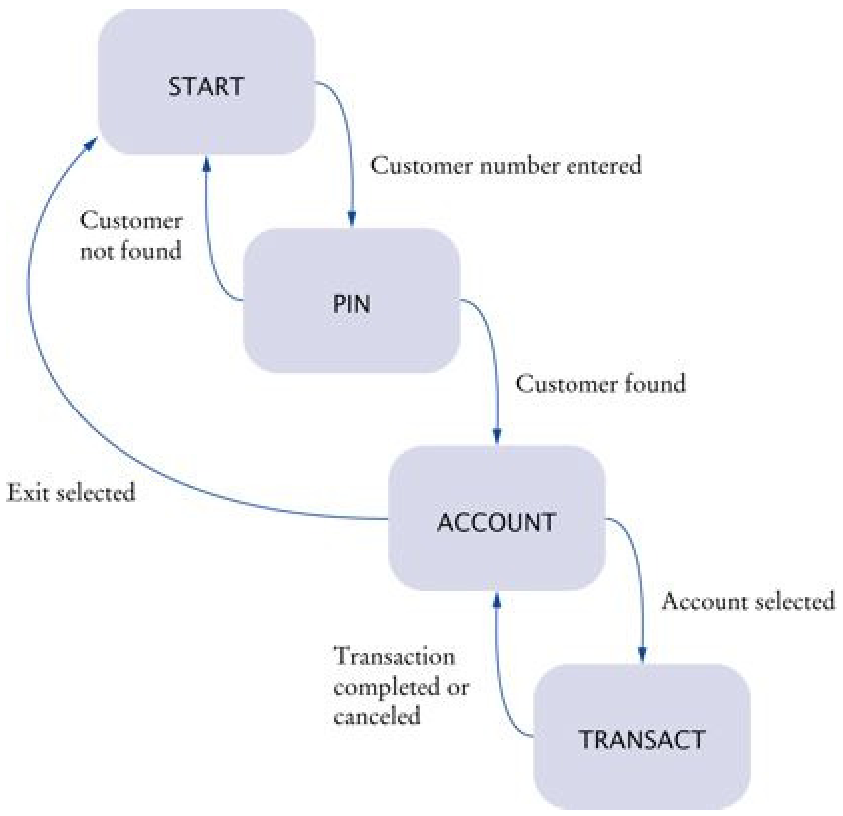
The `BankAccount` class is our familiar class with methods to get the balance and to deposit and withdraw money.

In this program there is nothing that distinguishes checking accounts from savings accounts. The ATM does not add interest or deduct fees. Therefore, we decide not to implement separate subclasses for checking and savings accounts.

Finally, we are left with the ATM class itself. An important notion of the ATM is the *state*. The current machine state determines the text of the prompts and the function of the buttons. For example, when you first log in, you use the A and B buttons to select an account. Next, you use the same buttons to choose between deposit and withdrawal. The ATM must remember the current state so that it can correctly interpret the buttons. 559

560

Figure 10



State Diagram for the ATM Class

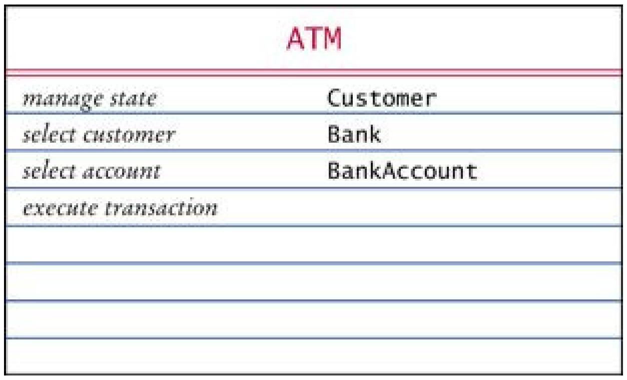
There are four states:

1. START: Enter customer ID
2. PIN: Enter PIN
3. ACCOUNT: Select account
4. TRANSACT: Select transaction

To understand how to move from one state to the next, it is useful to draw a *state diagram* (Figure 10). The UML notation has standardized shapes for state diagrams. Draw states as rectangles with rounded corners. Draw state changes as arrows, with labels that indicate the reason for the change.

The user must type a valid customer number and PIN. Then the ATM can ask the bank to find the customer. This calls for a *select customer* method. It collaborates with the bank, asking the bank for the customer that matches the customer number and PIN. Next, there must be a *select account* method that asks the current customer for the checking or savings account. Finally, the ATM must carry out the selected transaction on the current account. 560

561



ATM	
manage state	Customer
select customer	Bank
select account	BankAccount
execute transaction	

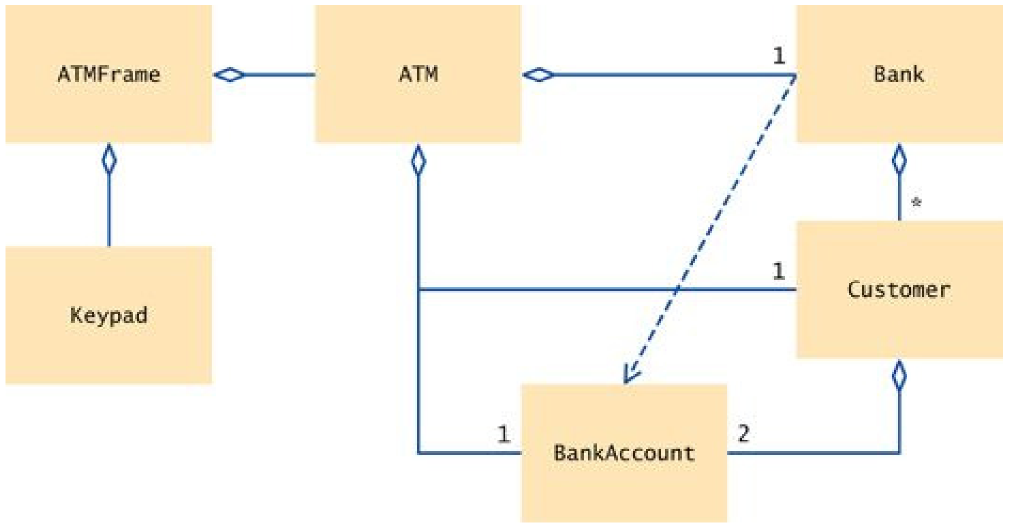
Of course, discovering these classes and methods was not as neat and orderly as it appears from this discussion. When I designed these classes for this book, it took me several trials and many torn cards to come up with a satisfactory design. It is also important to remember that there is seldom one best design.

This design has several advantages. The classes describe clear concepts. The methods are sufficient to implement all necessary tasks. (I mentally walked through every ATM usage scenario to verify that.) There are not too many collaboration dependencies between the classes. Thus, I was satisfied with this design and proceeded to the next step.

12.5.3 UML Diagrams

Figure 11 shows the relationships between these classes, using the graphical user interface. (The console user interface uses a single class `ATMSimulator` instead of the `ATMFrame` and `Keypad` classes.)

Figure 11



Relationships Between the ATM Classes

To draw the dependencies, use the “collaborator” columns from the CRC cards. Looking at those columns, you find that the dependencies are as follows:

- `ATM` uses `Bank`, `Customer`, and `BankAccount`.
- `Bank` uses `Customer`.
- `Customer` uses `BankAccount`.

It is easy to see some of the aggregation relationships. A bank has customers, and each customer has two bank accounts.

Does the `ATM` class aggregate `Bank`? To answer this question, ask yourself whether an ATM object needs to store a reference to a bank object. Does it need to locate the same bank object across multiple method calls? Indeed it does. Therefore, aggregation is the appropriate relationship.

Does an ATM aggregate customers? Clearly, the ATM is not responsible for storing all of the bank's customers. That's the bank's job. But in our design, the ATM remembers the *current* customer. If a customer has logged in, subsequent commands refer to the same customer. The ATM needs to either store a reference to the customer, or ask the bank to look up the object whenever it needs the current customer. It is a design decision: either store the object, or look it up when needed. We will decide to store the current customer object. That is, we will use aggregation. Note that the choice of aggregation is not an automatic consequence of the problem description—it is a design decision.

Similarly, we will decide to store the current bank account (checking or savings) that the user selects. Therefore, we have an aggregation relationship between `ATM` and `BankAccount`.

The class diagram is a good tool to visualize dependencies. Look at the GUI classes. They are completely independent from the rest of the ATM system. You can replace the GUI with a console interface, and you can take out the `Keypad` class and use it in another application. Also, the `Bank`, `BankAccount`, and `Customer` classes, although dependent on each other, don't know anything about the `ATM` class. That makes sense—you can have banks without ATMs. As you can see, when you analyze relationships, you look for both the absence and presence of relationships

12.5.4 Method Documentation

Now you are ready for the final step of the design phase: to document the classes and methods that you discovered. Here is a part of the documentation for the `ATM` class:

```
/**
  An ATM that accesses a bank.
*/
public class ATM
```

```
{
       /**
          Constructs an ATM for a given bank.
          @param aBank the bank to which this ATM connects
       */
       public ATM(Bank aBank) { }
```

```
       /**
          Sets the current customer number
          and sets state to PIN.
          (Precondition: state is START)
          @param number the customer number
       */
       public void setCustomerNumber(int number) { }
       /**
          Finds customer in bank.
          If found sets state to ACCOUNT, else to START.
          (Precondition: state is PIN)
          @param pin the PIN of the current customer
       */
       public void selectCustomer(int pin) { }
       /**
          Sets current account to checking or savings. Sets
          state to TRANSACT.
          (Precondition: state is ACCOUNT or TRANSACT)
          @param account one of CHECKING or SAVINGS
       */
       public void selectAccount(int account) { }
       /**
          Withdraws amount from current account.
          (Precondition: state is TRANSACT)
          @param value the amount to withdraw
       */
       public void withdraw(double value) { }
       . . .
}
```

Then run the `javadoc` utility to turn this documentation into HTML format.

For conciseness, we omit the documentation of the other classes.

12.5.5 Implementation

Finally, the time has come to implement the ATM simulator. The implementation phase is very straightforward and should take *much less time than the design phase*.

A good strategy for implementing the classes is to go “bottom-up”. Start with the classes that don't depend on others, such as `Keypad` and `BankAccount`. Then implement a class such as `Customer` that depends only on the `BankAccount` class. This “bottom-up” approach allows you to test your classes individually. You will find the implementations of these classes at the end of this section.

The most complex class is the `ATM` class. In order to implement the methods, you need to define the necessary instance variables. From the class diagram, you can tell that the ATM has a bank object. It becomes an instance variable of the class:

```
public class ATM
{
      . . .
      private Bank theBank;
}
```

From the description of the ATM states, it is clear that we require additional instance variables to store the current state, customer, and bank account.

```
public class ATM
{
      . . .
      private int state;
      private Customer currentCustomer;
      private BankAccount currentAccount;
      . . .
}
```

Most methods are very straightforward to implement. Consider the `selectCustomer` method. From the design documentation, we have the description

```
/**
   Finds customer in bank.
   If found sets state to ACCOUNT, else to START.
   (Precondition: state is PIN)
```

```
     @param pin the PIN of the current customer
    */
```

This description can be almost literally translated to Java instructions:

```
public void selectCustomer(int pin)
{
       assert state == PIN;
       currentCustomer =
theBank.findCustomer(customerNumber, pin);
       if (currentCustomer == null)
         state = START;
      else
         state = ACCOUNT;
}
```

We won't go through a method-by-method description of the ATM program. You should take some time and compare the actual implementation against the CRC cards and the UML diagram.

ch12/atm/ATM.java

```
/**
   An ATM that accesses a bank.
*/
public class ATM
{
   /**
     Constructs an ATM for a given bank.
     @param aBank the bank to which this ATM connects
   */
   public ATM(Bank aBank)
   {
          theBank = aBank;
          reset();
   }

   /**
       Resets the ATM to the initial state.
   */
   public void reset()
   {
```

564

565

```
            customerNumber = -1;
            currentAccount = null;
            state = START;
      }

      /**
         Sets the current customer number
         and sets state to PIN.
         (Precondition: state is START)
         @param number the customer number
      */
      public void setCustomerNumber(int number)
      {
             assert state == START;
             customerNumber = number;
             state = PIN;
      }

      /**
         Finds customer in bank.
         If found, sets state to ACCOUNT, else to START.
         (Precondition: state is PIN)
         @param pin the PIN of the current customer
      */
      public void selectCustomer(int pin)
      {
             assert state == PIN;
             currentCustomer =
theBank.findCustomer(customerNumber, pin);
             if (currentCustomer == null)
                state = START;
            else
                state = ACCOUNT;
      }

      /**
         Sets current account to checking or savings. Sets
         state to TRANSACT.
         (Precondition: state is ACCOUNT or TRANSACT)
         @param account one of CHECKING or SAVINGS
      */
```

```java
    public void selectAccount(int account)
    {
          assert state == ACCOUNT || state == TRANSACT;
          if (account == CHECKING)
              currentAccount = currentCustomer.getCheckingAccount();
          else
              currentAccount = currentCustomer.getSavingsAccount();
          state = TRANSACT;
```

```java
    }

    /**
       Withdraws amount from current account.
       (Precondition: state is TRANSACT)
       @param value the amount to withdraw
    */
    public void withdraw(double value)
    {
          assert state == TRANSACT;
          currentAccount.withdraw(value);
    }

    /**
       Deposits amount to current account.
       (Precondition: state is TRANSACT)
       @param value the amount to deposit
    */
    public void deposit(double value)
    {
          assert state == TRANSACT;
          currentAccount.deposit(value);
    }

    /**
       Gets the balance of the current account.
       (Precondition: state is TRANSACT)
       @return the balance
    */
    public double getBalance()
```

```
     {
        assert state == TRANSACT;
      return currentAccount.getBalance();
   }

   /**
      Moves back to the previous state.
   */
   public void back()
   {
      if (state == TRANSACT)
         state = ACCOUNT;
      else if (state == ACCOUNT)
         state = PIN;
      else if (state == PIN)
         state = START;
   }

   /**
      Gets the current state of this ATM.
    @return the current state
   */
   public int getState()
   {
      return state;
   }

   private int state;
   private int customerNumber;
   private Customer currentCustomer;
   private BankAccount currentAccount;
   private Bank theBank;

   public static final int START = 1;
   public static final int PIN = 2;
   public static final int ACCOUNT = 3;
   public static final int TRANSACT = 4;

   public static final int CHECKING = 1;
   public static final int SAVINGS = 2;
}
```

566

567

ch12/atm/Bank.java

```java
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Scanner;

/**
   A bank contains customers with bank accounts.
*/
public class Bank
{
   /**
      Constructs a bank with no customers.
   */
   public Bank()
   {
      customers = new ArrayList<Customer>();
   }

   /**
      Reads the customer numbers and pins
      and initializes the bank accounts.
      @param filename the name of the customer file
   */
   public void readCustomers(String filename)
         throws IOException
   {
      Scanner in = new Scanner(new FileReader(filename));
      while (in.hasNext())
      {
         int number = in.nextInt();
         int pin = in.nextInt();
         Customer c = new Customer(number, pin);
         addCustomer(c);
```

567

568

```java
      }
      in.close();
   }

```

```
    /**
        Adds a customer to the bank.
        @param c the customer to add
    */
    public void addCustomer(Customer c)
    {
        customers.add(c);
    }

    /**
        Finds a customer in the bank.
        @param aNumber a customer number
        @param aPin a personal identification number
        @return the matching customer, or null if no customer
         matches
    */
    public Customer findCustomer(int aNumber,
int aPin)
    {
        for (Customer c : customers)
        {
            if (c.match(aNumber, aPin))
                return c;
        }
        return null;
    }

    private ArrayList<Customer> customers;
}
```

ch12/atm/Customer.java

```
/**
    A bank customer with a checking and a savings account.
*/
public class Customer
{
    /**
        Constructs a customer with a given number and PIN.
        @param aNumber the customer number
```

```
         @param aPinthe personal identification number
      */
      public Customer(int aNumber, int aPin)
      {
         customerNumber = aNumber;
         pin = aPin;
         checkingAccount = new BankAccount();
         savingsAccount = new BankAccount();
      }

```

568

569

```
      /**
         Tests if this customer matches a customer number
          and PIN.
         @param aNumber a customer number
         @param aPin a personal identification number
         @return true if the customer number and PIN match
      */
      public boolean match(int aNumber, int aPin)
      {
         return customerNumber == aNumber && pin == aPin;
      }

      /**
         Gets the checking account of this customer.
         @return the checking account
      */
      public BankAccount getCheckingAccount()
      {
         return checkingAccount;
      }

      /**
        Gets the savings account of this customer.
        @return the checking account
      */
      public BankAccount getSavingsAccount()
      {
         return savingsAccount;
      }

```

```
   private int customerNumber;
   private int pin;
   private BankAccount checkingAccount;
   private BankAccount savingsAccount;
}
```

The following class implements a console user interface for the ATM.

ch12/atm/ATMSimulator.java

```
import java.io.IOException;
import java.util.Scanner;

/**
   A text-based simulation of an automatic teller machine.
*/
public class ATMSimulator
{
   public static void main(String[] args)
   {
      ATM theATM;
      try
      {
```

569

570

```
         Bank theBank = new Bank();
         theBank.readCustomers("customers.txt");
         theATM = new ATM(theBank);
      }
      catch(IOException e)
      {
         System.out.println("Error opening accounts file.");
         return;
      }

      Scanner in = new Scanner(System.in);

      while (true)
      {
         int state = theATM.getState();
         if (state == ATM.START)
         {
            System.out.print(“Enter customer number: ”);
            int number = in.nextInt();
            theATM.setCustomerNumber(number);
         }
         else if (state == ATM.PIN)
         {
            System.out.print(“Enter PIN: ”);
            int pin = in.nextInt();
            theATM.selectCustomer(pin);
         }
         else if (state == ATM.ACCOUNT)
         {
            System.out.print(“A=Checking, B=Savings, C=Quit: ”);
            String command = in.next();
            if (command.equalsIgnoreCase(“A”))
               theATM.selectAccount(ATM.CHECKING);
            else if (command.equalsIgnoreCase(“B”))
               theATM.selectAccount(ATM.SAVINGS);
            else if (command.equalsIgnoreCase(“C”))
               theATM.reset();
            else
               System.out.println(“Illegal input!”);
         }
         else if (state == ATM.TRANSACT)
         {
            System.out.println(“Balance=” + theATM.getBalance());
            System.out.print(“A=Deposit, B=Withdrawal, C=Cancel: ”);
            String command = in.next();
            if (command.equalsIgnoreCase(“A”))
            {
               System.out.print(“Amount: ”);
               double amount = in.nextDouble();
               theATM.deposit(amount);
               theATM.back();
            }
            else if (command.equalsIgnoreCase("B"))
            {
               System.out.print("Amount: ");
               double amount = in.nextDouble();
               theATM.withdraw(amount);
               theATM.back();
            }
            else if (command.equalsIgnoreCase("C"))
               theATM.back();
            else
               System.out.println("Illegal input!");
         }
      }
   }
}
```

570

571

Output

```
Enter account number: 1
Enter PIN: 1234
A=Checking, B=Savings, C=Quit: A
Balance=0.0
A=Deposit, B=Withdrawal, C=Cancel: A
Amount: 1000
A=Checking, B=Savings, C=Quit: C
. . .
```

Here are the user interface classes for the GUI version of the user interface.

ch12/atm/ATMViewer.java

```
import java.io.IOException;
import javax.swing.JFrame;
import javax.swing.JOptionPane;

/**
   A graphical simulation of an automatic teller machine.
*/
public class ATMViewer
```

```java
{
    public static void main(String[] args)
    {
       ATM theATM;

       try
       {
          Bank theBank = new Bank();
          theBank.readCustomers("customers.txt");
          theATM = new ATM(theBank);
       }
       catch(IOException e)
       {
          JOptionPane.showMessageDialog(null,
                "Error opening accounts
file.");
          return;
```
571

572

```java
       }

       JFrame frame = new ATMFrame(theATM);
       frame.setTitle("First National Bank of
Java");
       frame.setDefaultCloseOperation(JFrame.EXIT_O
       frame.setVisible(true);
    }
}
```

ch12/atm/ATMFrame.java

```java
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextArea;

 /**
    A frame displaying the components of an ATM.
 */
 public class ATMFrame extends JFrame
```

```
{
   /**
      Constructs the user interface of the ATM frame.
   */
   public ATMFrame(ATM anATM)
   {
      theATM = anATM;

      // Construct components
      pad = new KeyPad();

      display = new JTextArea(4, 20);

      aButton = new JButton("  A  ");
      aButton.addActionListener(new ABButtonListener());

      bButton = new JButton("  B  ");
      bButton.addActionListener(new BButtonListener());

      cButton = new JButton("  C  ");
      cButton.addActionListener(new CButtonListener());

      // Add components

      JPanel buttonPanel = new JPanel();
      buttonPanel.add(aButton);
      buttonPanel.add(bButton);
      buttonPanel.add(cButton);

```

572

573

```
      setLayout(new FlowLayout());
      add(pad);
      add(display);
      add(buttonPanel);
      showState();

      setSize(FRAME_WIDTH, FRAME_HEIGHT);
   }

   /**
```

```
        Updates display message.
    */
    public void showState()
    {
        int state = theATM.getState();
        pad.clear();
        if (state == ATM.START)
            display.setText("Enter customer
number\nA = OK");
        else if (state == ATM.PIN)
            display.setText("Enter PIN\nA = OK");
        else if (state == ATM.ACCOUNT)
            display.setText("Select Account\n"
                    + "A = Checking\nB =
Savings\nC = Exit");
        else if (state == ATM.TRANSACT)
            display.setText("Balance = "
                    + theATM.getBalance()
                    + "\nEnter amount and select
transaction\n"
                    + "A = Withdraw\nB =
Deposit\nC = Cancel");
    }

    private class AButtonListener implements
ActionListener
    {
        public void actionPerformed(ActionEvent
event)
        {
            int state = theATM.getState();
            if (state == ATM.START)
                theATM.setCustomerNumber((int)
pad.getValue());
            else if (state == ATM.PIN)
                theATM.selectCustomer((int)
pad.getValue());
            else if (state == ATM.ACCOUNT)
                theATM.selectAccount(ATM.CHECKING);
            else if (state == ATM.TRANSACT)
            {
                theATM.withdraw(pad.getValue());
                theATM.back();
```

```
         }
         showState();
      }
   }

   private class BButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
```

```
         int state = theATM.getState();
         if (state == ATM.ACCOUNT)
           theATM.selectAccount(ATM.SAVINGS);
          else if (state == ATM.TRANSACT)
          {
             theATM.deposit(pad.getValue());
             theATM.back();
          }
          showState();
       }
   }

   private class CButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         int state = theATM.getState();
         if (state == ATM.ACCOUNT)
            theATM.reset();
         else if (state == ATM.TRANSACT)
            theATM.back();
         showState();
      }
   }

   private JButton aButton;
   private JButton bButton;
   private JButton cButton;

   private KeyPad pad;
```

```
    private JTextArea display;

    private ATM theATM;

    private static final int FRAME_WIDTH = 300;
    private static final int FRAME_HEIGHT =
300;
}
```

This class uses layout managers to arrange the text field and the keypad buttons. See Chapter 18 for more information about layout managers.

ch12/atm/KeyPad.java

```
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JPanel;
import javax.swing.JTextField;

/**
      A component that lets the user enter a number, using
      a keypad labeled with digits.
 */
public class KeyPad extends JPanel
{
    /**
          Constructs the keypad panel.
    */
    public KeyPad()
    {
        setLayout(new BorderLayout());

        // Add display field

        display = new JTextField();
        add(display, "North");

        // Make button panel
```

```java

        buttonPanel = new JPanel();
        buttonPanel.setLayout(new GridLayout(4, 3));

        //Add digit buttons

        addButton("7");
        addButton("8");
        addButton("9");
        addButton("4");
        addButton("5");
        addButton("6");
        addButton("1");
        addButton("2");
        addButton("3");
        addButton("0");
        addButton(".");

        // Add clear entry button

        clearButton = new JButton("CE");
        buttonPanel.add(clearButton);

        class ClearButtonListener implements ActionListener
        {
           public void actionPerformed(ActionEvent event)
           {
              display.setText("");
           }
        }
        ActionListener listener = new ClearButtonListener();

        clearButton.addActionListener(new
              ClearButtonListener());

        add(buttonPanel, "Center");
    }

    /**
```

```
         Adds a button to the button panel.
         @param label the button label
      */
      private void addButton(final String label)
      {
         class DigitButtonListener implements ActionListener
         {
            public void actionPerformed(ActionEvent event)
            {

               // Don't add two decimal points
               if (label.equals(".")
                     && display.getText().indexOf(".") != -1)
                  return;

               // Append label text to button
               display.setText(display.getText() + label);
            }
         }

         JButton button = new JButton (label);
         buttonPanel.add(button);
         ActionListener listener = new DigitButtonListener();
         button.addActionListener(listener);
      }

      /**
         Gets the value that the user entered.
         @return the value in the text field of the keypad
      */
      public double getValue()
      {
         return Double.parseDouble(display.getText());
      }

      /**
```

```
          Clears the display.
      */
      public void clear()
      {
         display.setText("");
      }

      private JPanel buttonPanel;
      private JButton clearButton;
      private JTextField display;
   }
```

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 576 577 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.

SELF CHECK

12. Why does the Bank class in this example not store an array list of bank accounts?

13. Suppose the requirements change—you need to save the current account balances to a file after every transaction and reload them when the program starts. What is the impact of this change on the design?

RANDOM FACT 12.2: Software Development–Art or Science?

There has been a long discussion whether the discipline of computing is a science or not. We call the field "computer science", but that doesn't mean much.

Except possibly for librarians and sociologists, few people believe that library science and social science are scientific endeavors.

A scientific discipline aims to discover certain fundamental principles dictated by the laws of nature. It operates on the *scientific method*: by posing hypotheses and testing them with experiments that are repeatable by other workers in the field. For example, a physicist may have a theory on the makeup of nuclear particles and attempt to confirm or refute that theory by running experiments in a particle collider. If an experiment cannot be confirmed, such as the "cold fusion" research in the early 1990s, then the theory dies a quick death.

Some software developers indeed run experiments. They try out various methods of computing certain results or of configuring computer systems, and measure the differences in performance. However, their aim is not to discover laws of nature.

Some computer scientists discover fundamental principles. One class of fundamental results, for instance, states that it is impossible to write certain kinds of computer programs, no matter how powerful the computing equipment is. For example, it is impossible to write a program that takes as its input any two Java program files and as its output prints whether or not these two programs always compute the same results. Such a program would be very handy for grading student homework, but nobody, no matter how clever, will ever be able to write one that works for all input files. However, the majority of computer scientists are not researching the limits of computation.

Some people view software development as an *art* or *craft*. A programmer who writes elegant code that is easy to understand and runs with optimum efficiency can indeed be considered a good craftsman. Calling it an art is perhaps far-fetched, because an art object requires an audience to appreciate it, whereas the program code is generally hidden from the program user.

Others call software development an *engineering discipline*. Just as mechanical engineering is based on the fundamental mathematical principles of statics, computing has certain mathematical foundations. There is more to mechanical 577
engineering than mathematics, such as knowledge of materials and of project 578
planning. The same is true for computing. A *software engineer* needs to know about planning, budgeting, design, test automation, documentation, and source

code control, in addition to computer science subjects, such as programming, algorithm design, and database technologies.

In one somewhat worrisome aspect, software development does not have the same standing as other engineering disciplines. There is little agreement as to what constitutes professional conduct in the computer field. Unlike the scientist, whose main responsibility is the search for truth, the software developer must strive to satisfy the conflicting demands of quality, safety, and economy. Engineering disciplines have professional organizations that hold their members to standards of conduct. The computer field is so new that in many cases we simply don't know the correct method for achieving certain tasks. That makes it difficult to set professional standards.

What do you think? From your limited experience, do you consider software development an art, a craft, a science, or an engineering activity?

CHAPTER SUMMARY

1. The life cycle of software encompasses all activities from initial analysis until obsolescence.
2. A formal process for software development describes phases of the development process and gives guidelines for how to carry out the phases.
3. The waterfall model of software development describes a sequential process of analysis, design, implementation, testing, and deployment.
4. The spiral model of software development describes an iterative process in which design and implementation are repeated.
5. Extreme Programming is a development methodology that strives for simplicity by removing formal structure and focusing on best practices.
6. In object-oriented design, you discover classes, determine the responsibilities of classes, and describe the relationships between classes.
7. A CRC card describes a class, its responsibilities, and its collaborating classes.

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

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

10. Dependency is another name for the "uses" relationship.

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

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

FURTHER READING

1. Grady Booch, James Rumbaugh, and Ivar Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.
2. Kent Beck, *Extreme Programming Explained*, Addison-Wesley, 1999.
3. W. H. Sackmann, W. J. Erikson, and E. E. Grant, "Exploratory Experimental Studies Comparing Online and Offline Programming Performance", *Communications of the ACM, vol. 11, no. 1* (January 1968), pp. 3–11.
4. F. Brooks, *The Mythical Man-Month*, Addison-Wesley, 1975.

REVIEW EXERCISES

★ **Exercise R12.1.** What is the software life cycle?

★★ **Exercise R12.2.** List the steps in the process of object-oriented design that this chapter recommends for student use.

★ **Exercise R12.3.** Give a rule of thumb for how to find classes when designing a program.

★ **Exercise R12.4.** Give a rule of thumb for how to find methods when designing a program.

★★ **Exercise R12.5.** After discovering a method, why is it important to identify the object that is *responsible* for carrying out the action?

★ **Exercise R12.6.** What relationship is appropriate between the following classes: aggregation, inheritance, or neither?

a. `University-Student`

b. `Student-TeachingAssistant`

c. `Student-Freshman`

d. `Student-Professor`

e. `Car-Door`

f. `Truck-Vehicle`

g. `Traffic-TrafficSign`

h. `TrafficSign-Color`

★★ **Exercise R12.7.** 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`?

★★ **Exercise R12.8.** Some books on object-oriented programming recommend using inheritance so that the class `Circle` extends the class `Point`. Then the `Circle` class inherits the `setLocation` method from the `Point` 579
superclass. Explain why the `setLocation` method need not be redefined 580
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?

★ **Exercise R12.9.** Write CRC cards for the `Coin` and `CashRegister` classes described in Section 8.2.

★ **Exercise R12.10.** Write CRC cards for the `Bank` and `BankAccount` classes in Section 7.2.

★★ **Exercise R12.11.** Draw a UML diagram for the `Coin` and `CashRegister` classes described in Section 8.2.

★★★ **Exercise R12.12.** 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.

★★★ **Exercise R12.13.** 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.

★★★ **Exercise R12.14.** Consider a quiz grading system that grades student responses to quizzes. A quiz consists of questions. There are different types of questions, including essay questions and multiple-choice questions. Students turn in submissions for quizzes, and the grading system grades them. Draw a UML diagram for classes `Quiz`, `Question`, `EssayQuestion`, `MultipleChoiceQuestion`, `Student`, and `Submission`.

- Additional review exercises are available in WileyPLUS.

PROGRAMMING EXERCISES

★★ **Exercise P12.1.** 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.

★★ **Exercise P12.2.** 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 a parameter. Use the `NumberFormat` class to format the currency values.

★★ **Exercise P12.3.** The invoice-printing program has an unfortunate flaw—it mixes “business 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.

★★★ **Exercise P12.4.** Write a program that teaches arithmetic to your younger brother. 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 non-negative difference. Generate random problems and get the player 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.

★★★ **Exercise P12.5.** Design 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.

★★ **Exercise P12.6.** 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 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.

★★★ **Exercise P12.7.** Write a program to design an appointment calendar. An appointment includes the date, starting time, ending time, and a description; for example,

```
Dentist 2007/10/1 17:30 18:30
CS1 class 2007/10/2 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.

★★★ **Exercise P12.8.** *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 581
window in economy). Then try to find a match and assign the seats. If no 582
match exists, print a message. Your solution should include a class `Airplane` that is not coupled with the `Scanner` or `PrintSream` classes. Follow the design process that was described in this chapter.

★★ **Exercise P12.9.** Modify the implementations of the class in the ATM example 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.

★★★ **Exercise P12.10.** 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.

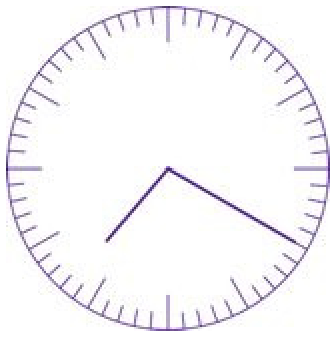
A quiz is specified in a text file. Each question starts with a letter indicating the question type (T, N, S, M), followed by a line containing the question text. The next line of a non-choice question contains the answer. Choice questions have a list of choices that is terminated by a blank line. Each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

```
T
Which Java keyword is used to define 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.

★★★G Exercise P12.11. Implement a program to teach your baby sister to read the clock. In the game, present an analog clock, such as the one in Figure 12. Generate random times and display the clock. Accept guesses from the player. Reward the player for correct guesses. After two 582
583 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.

Figure 12



An Analog Clock

★★★G Exercise P12.12. 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.

★★★G Exercise P12.13. 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.

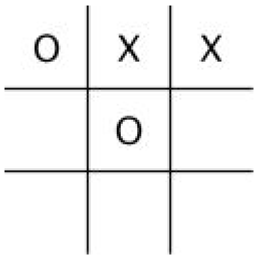
- Additional programming exercises are available in WileyPLUS.

PROGRAMMING PROJECTS

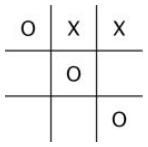
★★★ **Project 12.1.** Produce a requirements document for a program that allows a company to send out personalized mailings, either by e-mail or through the postal service. Template files contain the message text, together with variable fields (such as Dear [Title] [Last Name] …). A database (stored as a text file) contains the field values for each recipient. Use HTML as the output file format. Then design and implement the program.

★★★ **Project 12.2.** 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 583

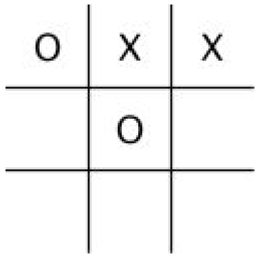
584 immediately preceding combination is stored as losing. For example, suppose that X = computer and O = human. Suppose the current combination is



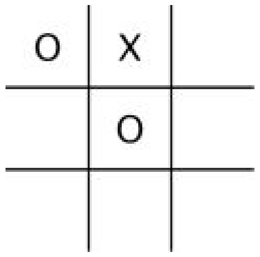
Now it is the human's turn, who will of course choose



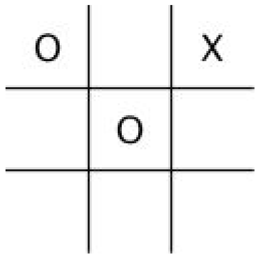
The computer should then remember the preceding combination



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



or



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

584

585

ANSWERS TO SELF-CHECK QUESTIONS

1. It is unlikely that the customer did a perfect job with the requirements document. If you don't accommodate changes, your customer may not like the outcome. If you charge for the changes, your customer may not like the cost.

2. An “extreme” spiral model, with lots of iterations.
3. To give frequent feedback as to whether the current iteration of the product fits customer needs.
4. `FileWriter`
5. To produce the shipping address of the customer.
6. Reword the responsibilities so that they are at a higher level, or come up with more classes to handle the responsibilities.
7. Through aggregation. The bank manages bank account objects.
8. Through inheritance.
9. The `BankAccount`, `System`, and `PrintStream` classes.
10. The `Invoice` class is responsible for computing the amount due. It collaborates with the `LineItem` class.
11. 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.
12. The bank needs to store the list of customers so that customers can log in. We need to locate all bank accounts of a customer, and we chose to simply store them in the customer class. In this program, there is no further need to access bank accounts.
13. The `Bank` class needs to have an additional responsibility: to load and save the accounts. The bank can carry out this responsibility because it has access to the customer objects and, through them, to the bank accounts.