

OBJECT-ORIENTED DESIGN

Chapter Goals

- To learn how to discover new classes and methods
- To use CRC cards for class discovery
- To identify inheritance, aggregation, and dependency relationships between classes
- To describe class relationships using UML class diagrams
- To apply object-oriented design techniques to building complex programs

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

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

12.1.1 Discovering Classes

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

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

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

For example, suppose your job is to print an invoice such as the one in Figure 1.

INVOICE			
Sam's Small Appliances 100 Main Street Anytown, CA 98765			
Item	Qty	Price	Total
Toaster	3	\$29.95	\$89.85
Hair Dryer	1	\$24.95	\$24.95
Car Vacuum	2	\$19.99	\$39.98
AMOUNT DUE:			\$154.78

Figure 1 An Invoice

Obvious classes that come to mind are `Invoice`, `LineItem`, and `Customer`. It is a good idea to keep a list of *candidate classes* on a whiteboard or a sheet of paper. As you brainstorm, simply put all ideas for classes onto the list. You can always cross out the ones that weren't useful after all.

Concepts from the problem domain are good candidates for classes.

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

- Cannonball
- CashRegister
- Monster

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

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

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

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

12.1.2 The CRC Card Method

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

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

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

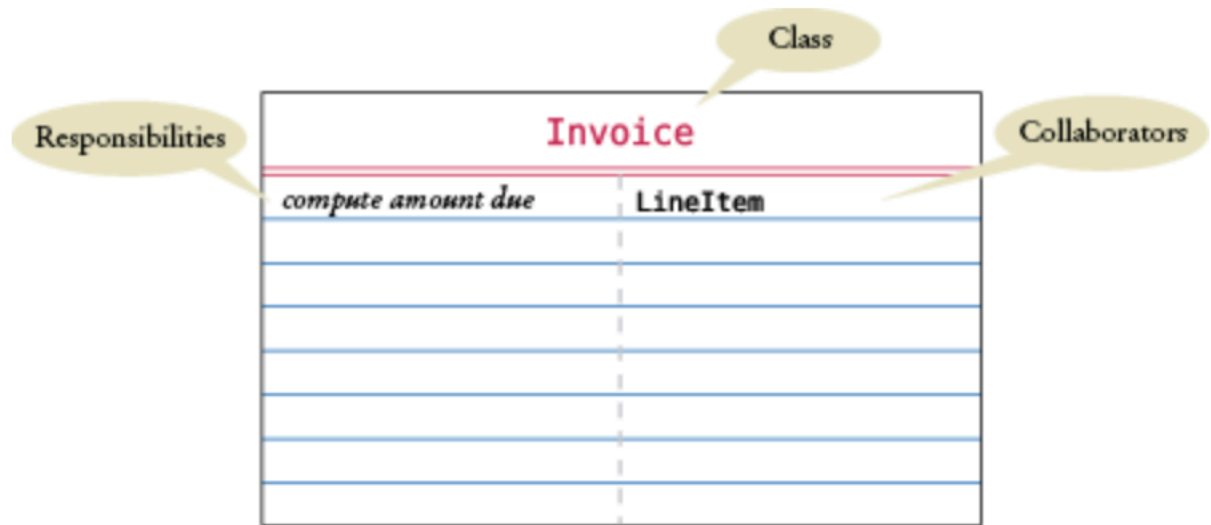


Figure 2 A CRC Card

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

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

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

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

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

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

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

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

12.2 Relationships Between Classes

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

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


12.2.1 Dependency

Too many dependencies make a system difficult to manage.

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

Many classes need other classes in order to do their jobs. For example, in Section 8.2.2, we described a design of a `CashRegister` class that depends on the `Coin` class to determine the value of the payment.

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

As you saw in Section 8.2, dependency is denoted by a dashed line with a -shaped open arrow tip. The arrow tip points to the class on which the other class depends. Figure 3 shows a class diagram indicating that the `CashRegister` class depends on the `Coin` class.

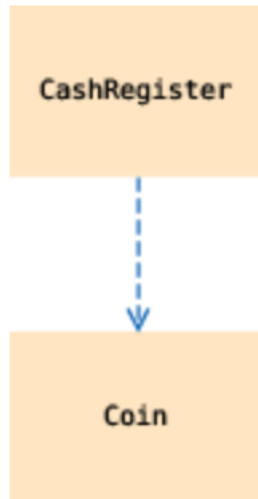


Figure 3 Dependency Relationship Between the `CashRegister` and `Coin` Classes

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

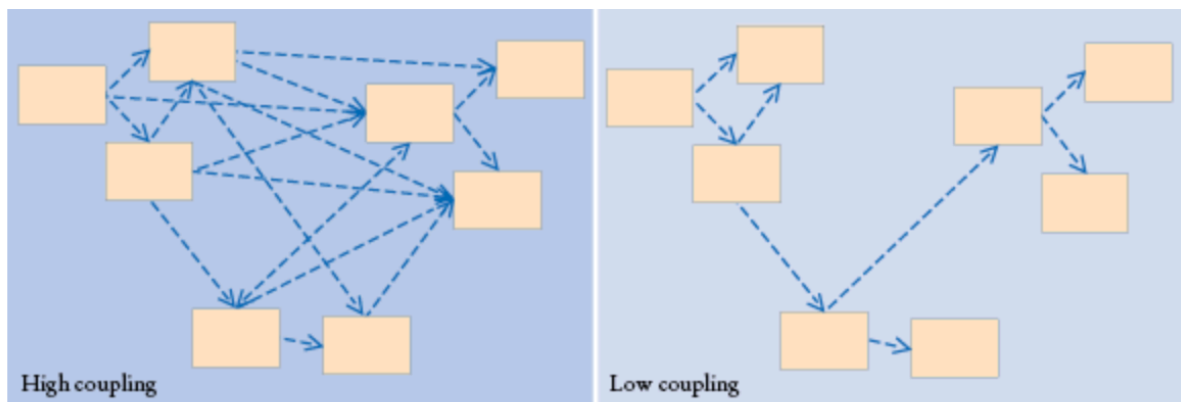


Figure 4 High and Low Coupling Between Classes

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

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

12.2.2 Aggregation

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

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

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

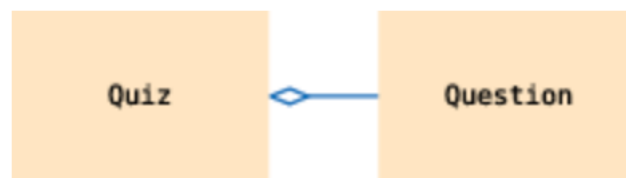


Figure 5 Class Diagram Showing Aggregation

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

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

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

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

Generally, you need aggregation when an object needs to remember another object *between method calls*. For example, a car has a motor and tires which is a has-a relationship or in other words an aggregation.

Example Code This program demonstrates the `Quiz` and `Question` classes.

sec02/QuizDemo.java

```
1 public class QuizDemo
2 {
3     public static void main(String[] args)
4     {
5         Question first = new Question();
6         first.setText("Who was the inventor of Java?");
7         first.setAnswer("James Gosling");
8
9         ChoiceQuestion second = new ChoiceQuestion();
10        second.setText("In which country was the inventor of Java born?");
11        second.addChoice("Australia", false);
12        second.addChoice("Canada", true);
13        second.addChoice("Denmark", false);
14        second.addChoice("United States", false);
15
16        Quiz q = new Quiz();
17        q.addQuestion(first);
18        q.addQuestion(second);
19        q.presentQuestions();
20    }
21 }
```


sec02/ChoiceQuestion.java

```
1 import java.util.ArrayList;
2
3 /**
4  * A question with multiple choices.
5  */
6 public class ChoiceQuestion extends Question
7 {
8     private ArrayList<String> choices;
9
10    /**
11     * Constructs a choice question with no choices.
12     */
13    public ChoiceQuestion()
14    {
15        choices = new ArrayList<String>();
16    }
17
18    /**
19     * Adds an answer choice to this question.
20     * @param choice the choice to add
21     * @param correct true if this is the correct choice, false otherwise
22     */
23    public void addChoice(String choice, boolean correct)
24    {
25        choices.add(choice);
26        if (correct)
27        {
28            // Convert choices.size() to string
29            String choiceString = "" + choices.size();
30            setAnswer(choiceString);
31        }
32    }
33
34    public void display()
35    {
36        // Display the question text
37        super.display();
38        // Display the answer choices
39        for (int i = 0; i < choices.size(); i++)
40        {
41            int choiceNumber = i + 1;
42            System.out.println(choiceNumber + ": " + choices.get(i));
43        }
44    }
45 }
```

sec02/Question.java

```
1  /**
2   A question with a text and an answer.
3  */
4  public class Question
5  {
6      private String text;
7      private String answer;
8
9      /**
10     Constructs a question with empty question and answer.
11     */
12     public Question()
13     {
14         text = "";
15         answer = "";
16     }
17
18     /**
19     Sets the question text.
20     @param questionText the text of this question
21     */
22     public void setText(String questionText)
23     {
24         text = questionText;
25     }
26
27     /**
28     Sets the answer for this question.
29     @param correctResponse the answer
30     */
31     public void setAnswer(String correctResponse)
32     {
33         answer = correctResponse;
34     }
35
36     /**
37     Checks a given response for correctness.
38     @param response the response to check
39     @return true if the response was correct, false otherwise
40     */
41     public boolean checkAnswer(String response)
42     {
43         return response.equals(answer);
44     }
45
46     /**
47     Displays this question.
48     */
49     public void display()
50     {
51         System.out.println(text);
52     }
53 }
```

sec02/Quiz.java

```
1 import java.util.ArrayList;
2 import java.util.Scanner;
3
4 /**
5  * A quiz contains a list of questions.
6  */
7 public class Quiz
8 {
9     private ArrayList<Question> questions;
10
11     /**
12      * Constructs a quiz with no questions.
13      */
14     public Quiz()
15     {
16         questions = new ArrayList<Question>();
17     }
18
19     /**
20      * Adds a question to this quiz.
21      * @param q the question
22      */
23     public void addQuestion(Question q)
24     {
25         questions.add(q);
26     }
27
28     /**
29      * Presents the questions to the user and checks the response.
30      */
31     public void presentQuestions()
32     {
33         Scanner in = new Scanner(System.in);
34
35         for (Question q : questions)
36         {
37             q.display();
38             System.out.print("Your answer: ");
39             String response = in.nextLine();
40             System.out.println(q.checkAnswer(response));
41         }
42     }
43 }
```

12.2.3 Inheritance

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

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

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

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

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

Here is another example: Every car *is a* vehicle. Every car *has a* tire (in fact, it typically has four or, if you count the spare, five). Thus, you would use inheritance from `Vehicle` and use aggregation of `Tire` objects (see Figure 6 for the UML diagram):

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

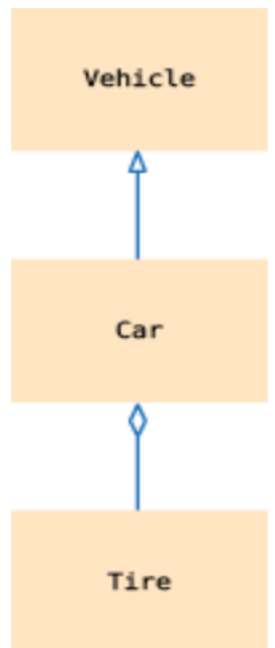

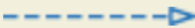
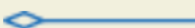



Figure 6 UML Notation for Inheritance and Aggregation

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

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

Table 1 UML Relationship Symbols			
Relationship	Symbol	Line Style	Arrow Tip
Inheritance		Solid	Triangle
Interface Implementation		Dotted	Triangle
Aggregation		Solid	Diamond
Dependency		Dotted	Open