CHAPTER

8 Designing Classes

Main concepts discussed in this chapter:

- responsibility-driven design
- coupling
- cohesion
- refactoring

Java constructs discussed in this chapter:

enumerated types, `switch`

In this chapter, we look at some of the factors that influence the design of a class. What makes a class design either good or bad? Writing good classes can take more effort in the short term than writing bad classes, but in the long term that extra effort will almost always be justified. To help us write good classes, there are some principles that we can follow. In particular, we introduce the view that class design should be responsibility-driven, and that classes should encapsulate their data.

This chapter is, like many of the chapters before, structured around a project. It can be studied by just reading it and following our line of argument, or it can be studied in much more depth by doing the project exercises in parallel while working through the chapter.

The project work is divided into three parts. In the first part, we discuss the necessary changes to the source code and develop and show complete solutions to the exercises. The solution for this part is also available in a project accompanying this book. The second part suggests more changes and extensions, and we discuss possible solutions at a high level (the class-design level), but leave it to readers to do the lower-level work and to complete the implementation. The third part suggests even more improvements in the form of exercises. We do not give solutions—the exercises apply the material discussed throughout the chapter.

Implementing all parts makes a good programming project over several weeks. It can also be used very successfully as a group project.

8.1 Introduction

It is possible to implement an application and get it to perform its task with badly designed classes. Simply executing a finished application does not usually indicate whether it is well structured internally or not.

The problems typically surface when a maintenance programmer wants to make some changes to an existing application. If, for example, a programmer attempts to fix a bug or wants to add new functionality to an existing program, a task that might be easy and obvious with well-designed classes may well be very hard and involve a great deal of work if the classes are badly designed.

In larger applications, this effect occurs earlier on, during the initial implementation. If the implementation starts with a bad structure, then finishing it might later become overly complex, and the complete program may either not be finished contain bugs, or take a lot longer to build. In reality, companies often maintain, extend, and sell an application over many years. It is not uncommon that an implementation for software that we can buy in a software store today was started more than ten years ago. In this situation, a software company cannot afford to have badly structured code.

Because many of the effects of bad class design become most obvious when trying to adapt or extend an application, we shall do exactly that. In this chapter, we will use an example called *world-of-zuul*, which is a rudimentary implementation of a text-based adventure game. In its original state, the game is not actually very ambitious—for one thing, it is incomplete. By the end of this chapter, however, you will be in a position to exercise your imagination and design and implement your own game and make it really fun and interesting.

world-of-zuul Our *world-of-zuul* game is modeled on the original *Adventure* game that was developed in the early 1970s by Will Crowther and expanded by Don Woods. The original game is also sometimes known as the *Colossal Cave Adventure*. This was a wonderfully imaginative and sophisticated game for its time, involving finding your way through a complex cave system, locating hidden treasure, using secret words, and other mysteries, all in an effort to score the maximum number of points. You can read more about it at sites such as http://jerz.setonhill.edu/if/canon/Adventure.htm and http://www.rickadams.org/adventure/, or try doing a web search for "Colossal Cave Adventure."

While we work on extending the original application, we will take the opportunity to discuss aspects of its existing class design. We will see that the implementation we start with contains examples of bad design, and we will be able to see how this impacts on our tasks and how we can fix them.

In the project examples for this book, you will find two versions of the *zuul* project: *zuul-bad* and *zuul-better*. Both implement exactly the same functionality, but some of the class structure is different, representing bad design in one project and better design in

the other. The fact that we can implement the same functionality in either a good way or a bad way illustrates the fact that bad design is not usually a consequence of having a difficult problem to solve. Bad design has more to do with the decisions that we make when solving a particular problem. We cannot use the argument that there was no other way to solve the problem as an excuse for bad design.

So, we will use the project with bad design so that we can explore why it is bad, and then improve it. The better version is an implementation of the changes we discuss here.

Exercise 8.1 Open the project *zuul-bad*. (This project is called "bad" because its implementation contains some bad design decisions, and we want to leave no doubt that this should not be used as an example of good programming practice!) Execute and explore the application. The project comment gives you some information about how to run it.

While exploring the application, answer the following questions:

- What does this application do?
- What commands does the game accept?
- What does each command do?
- How many rooms are in the scenario?
- Draw a map of the existing rooms.

Exercise 8.2 After you know what the whole application does, try to find out what each individual class does. Write down for each class its purpose. You need to look at the source code to do this. Note that you might not (and need not) understand all of the source code. Often, reading through comments and looking at method headers is enough.

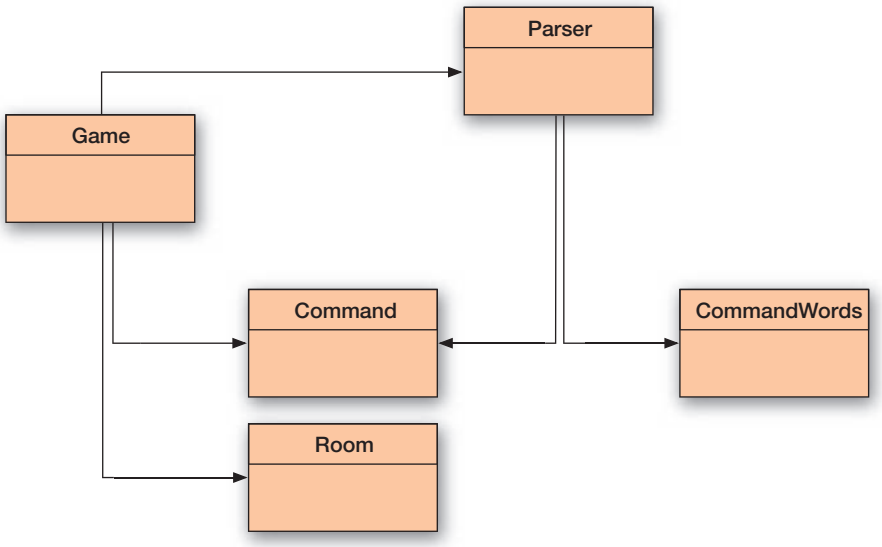
8.2 The *world-of-zuul* game example

From Exercise 8.1, you have seen that the *zuul* game is not yet very adventurous. It is, in fact, quite boring in its current state. But it provides a good basis for us to design and implement our own game, which will hopefully be more interesting.

We start by analyzing the classes that are already there in our first version and trying to find out what they do. The class diagram is shown in Figure 8.1.

The project shows five classes. They are **`Parser`**, **`CommandWords`**, **`Command`**, **`Room`**, and **`Game`**. An investigation of the source code shows, fortunately, that these classes are quite well documented, and we can get an initial overview of what they do by just reading the class comment at the top of each class. (This fact also serves to illustrate that bad design involves something deeper than simply the way a class looks or how good its documentation is.) Our understanding of the game will be assisted by having a look at the source code

Figure 8.1
`Zuul` class diagram



to see what methods each class has and what some of the methods appear to do. Here, we summarize the purpose of each class:

- *CommandWords* The **`CommandWords`** class defines all valid commands in the game. It does this by holding an array of **`String`** objects representing the command words.
- *Parser* The parser reads lines of input from the terminal and tries to interpret them as commands. It creates objects of class **`Command`** that represent the command that was entered.
- *Command* A **`Command`** object represents a command that was entered by the user. It has methods that make it easy for us to check whether this was a valid command and to get the first and second words of the command as separate strings.
- *Room* A **`Room`** object represents a location in a game. Rooms can have exits that lead to other rooms.
- *Game* The **`Game`** class is the main class of the game. It sets up the game and then enters a loop to read and execute commands. It also contains the code that implements each user command.

Exercise 8.3 Design your own game scenario. Do this away from the computer. Do not think about implementation, classes, or even programming in general. Just think about inventing an interesting game. This could be done with a group of people.

The game can be anything that has as its base structure a player moving through different locations. Here are some examples:

- You are a white blood cell traveling through the body in search of viruses to attack . . .
- You are lost in a shopping mall and must find the exit . . .
- You are a mole in its burrow and you cannot remember where you stored your food reserves before winter . . .
- You are an adventurer who searches through a dungeon full of monsters and other characters . . .
- You are on the bomb squad and must find and defuse a bomb before it goes off . . .

Make sure that your game has a goal (so that it has an end and the player can "win"). Try to think of many things to make the game interesting (trap doors, magic items, characters that help you only if you feed them, time limits. . . whatever you like). Let your imagination run wild.

At this stage, do not worry about how to implement these things.

8.3 Introduction to coupling and cohesion

Concept

The term **coupling** describes the interconnectedness of classes. We strive for loose coupling in a system—that is, a system where each class is largely independent and communicates with other classes via a small, well-defined interface.

If we are to justify our assertion that some designs are better than others, then we need to define some terms that will allow us to discuss the issues that we consider to be important in class design. Two terms are central when talking about the quality of a class design: *coupling* and *cohesion.*

The term *coupling* refers to the interconnectedness of classes. We have already discussed in earlier chapters that we aim to design our applications as a set of cooperating classes that communicate via well-defined interfaces. The degree of coupling indicates how tightly these classes are connected. We strive for a low degree of coupling, or *loose coupling.*

The degree of coupling determines how hard it is to make changes in an application. In a tightly coupled class structure, a change in one class can make it necessary to change several other classes as well. This is what we try to avoid, because the effect of making one small change can quickly ripple through a complete application. In addition, finding all the places where changes are necessary, and actually making the changes, can be difficult and time consuming.

In a loosely coupled system, on the other hand, we can often change one class without making any changes to other classes, and the application will still work. We shall discuss particular examples of tight and loose coupling in this chapter.

The term *cohesion* relates to the number and diversity of tasks for which a single unit of an application is responsible. Cohesion is relevant for units of a single class and an individual method.[1]

[1] We sometimes also use the term *module* (or *package* in Java) to refer to a multi-class unit. *Cohesion* is relevant at this level too.

Concept

The term **cohesion** describes how well a unit of code maps to a logical task or entity. In a highly cohesive system, each unit of code (method, class, or module) is responsible for a well-defined task or entity. Good class design exhibits a high degree of cohesion.

Ideally, one unit of code should be responsible for one cohesive task (that is, one task that can be seen as a logical unit). A method should implement one logical operation, and a class should represent one type of entity. The main reason behind the principle of cohesion is reuse: if a method or a class is responsible for only one well-defined thing, then it is much more likely it can be used again in a different context. A complementary advantage of following this principle is that, when change *is* required to some aspect of an application, we are likely to find all the relevant pieces located in the same unit.

We shall discuss with examples below how cohesion influences the quality of class design.

Exercise 8.4 Draw (on paper) a map for the game you invented in Exercise 8.3. Open the *zuul-bad* project and save it under a different name (e.g., *zuul*). This is the project you will use for making improvements and modifications throughout this chapter. You can leave off the *-bad* suffix, because it will (hopefully) soon not be that bad anymore.

As a first step, change the **`createRooms`** method in the **`Game`** class to create the rooms and exits you invented for your game.

8.4 Code duplication

Code duplication is an indicator of bad design. The **`Game`** class shown in Code 8.1 contains a case of code duplication. The problem with code duplication is that any change to one version must also be made to another if we are to avoid inconsistency. This increases the amount of work a maintenance programmer has to do, and it introduces the danger of bugs. It very easily happens that a maintenance programmer finds one copy of the code and, having changed it, assumes that the job is done. There is nothing indicating that a second copy of the code exists, and it might incorrectly remain unchanged.

Code 8.1
Selected sections of the (badly designed) `Game` class

```
public class Game
{
    Some code omitted.

    private void createRooms()
    {
        Room outside, theater, pub, lab, office;

        // create the rooms
        outside = new Room("outside the main entrance of the university");
        theater = new Room("in a lecture theater");
        pub = new Room("in the campus pub");
        lab = new Room("in a computing lab");
        office = new Room("in the computing admin office");
```

Code 8.1 continued
Selected sections of the (badly designed) `Game` class

Concept

Code duplication (having the same segment of code in an application more than once) is a sign of bad design. It should be avoided.

```
        // initialize room exits
        outside.setExits(null, theater, lab, pub);
        theater.setExits(null, null, null, outside);
        pub.setExits(null, outside, null, null);
        lab.setExits(outside, office, null, null);
        office.setExits(null, null, null, lab);

        currentRoom = outside;  // start game outside
    }
```

Some code omitted.

```
    /**
     * Print out the opening message for the player.
     */
    private void printWelcome()
    {
        System.out.println();
        System.out.println("Welcome to the World of Zuul!");
        System.out.println("Zuul is a new incredibly boring adventure game.");
        System.out.println("Type 'help' if you need help.");
        System.out.println();
        System.out.println("You are " + currentRoom.getDescription());
        System.out.print("Exits: ");
        if(currentRoom.northExit != null) {
            System.out.print("north ");
        }
        if(currentRoom.eastExit != null) {
            System.out.print("east ");
        }
        if(currentRoom.southExit != null) {
            System.out.print("south ");
        }
        if(currentRoom.westExit != null) {
            System.out.print("west ");
        }
        System.out.println();
    }
```

Some code omitted.

```
    /**
     * Try to go in one direction. If there is an exit, enter
     * the new room, otherwise print an error message.
     */
    private void goRoom(Command command)
    {
        if(!command.hasSecondWord()) {
            // if there is no second word, we don't know where to go...
            System.out.println("Go where?");
            return;
        }
```

Code 8.1 continued
Selected sections of the (badly designed) `Game` class

```
        String direction = command.getSecondWord();

        // Try to leave current room.
        Room nextRoom = null;
        if(direction.equals("north")) {
            nextRoom = currentRoom.northExit;
        }
        if(direction.equals("east")) {
            nextRoom = currentRoom.eastExit;
        }
        if(direction.equals("south")) {
            nextRoom = currentRoom.southExit;
        }
        if(direction.equals("west")) {
            nextRoom = currentRoom.westExit;
        }

        if (nextRoom == null) {
            System.out.println("There is no door!");
        }
        else {
            currentRoom = nextRoom;
            System.out.println("You are " + currentRoom.getDescription());
            System.out.print("Exits: ");
            if(currentRoom.northExit != null) {
                System.out.print("north ");
            }
            if(currentRoom.eastExit != null) {
                System.out.print("east ");
            }
            if(currentRoom.southExit != null) {
                System.out.print("south ");
            }
            if(currentRoom.westExit != null) {
                System.out.print("west ");
            }
            System.out.println();
        }
    }

    Some code omitted.
}
```

Both the **`printWelcome`** and **`goRoom`** methods contain the following lines of code:

```
System.out.println("You are" + currentRoom.getDescription());
System.out.print("Exits:");
if(currentRoom.northExit != null) {
    System.out.print("north");
}
if(currentRoom.eastExit != null) {
    System.out.print("east");
}
```

```
if(currentRoom.southExit != null) {
    System.out.print("south");
}
if(currentRoom.westExit != null) {
    System.out.print("west");
}
System.out.println();
```

Code duplication is usually a symptom of poor cohesion. The problem here has its roots in the fact that both methods in question do two things: **printWelcome** prints the welcome message and prints the information about the current location, while **goRoom** changes the current location then prints information about the (new) current location.

Both methods print information about the current location, but neither can call the other, because they also do other things. This is bad design.

A better design would use a separate, more cohesive method whose sole task is to print the current location information (Code 8.2). Both the **printWelcome** and **goRoom** methods can then make calls to this method when they need to print this information. This way, writing the code twice is avoided, and when we need to change it, we need to change it only once.

Code 8.2 `printLocationInfo` as a separate method

```
private void printLocationInfo()
{
    System.out.println("You are " + currentRoom.getDescription());
    System.out.print("Exits: ");
    if(currentRoom.northExit != null) {
        System.out.print("north ");
    }
    if(currentRoom.eastExit != null) {
        System.out.print("east ");
    }
    if(currentRoom.southExit != null) {
        System.out.print("south ");
    }
    if(currentRoom.westExit != null) {
        System.out.print("west ");
    }
    System.out.println();
}
```

Exercise 8.5 Implement and use a separate **printLocationInfo** method in your project, as discussed in this section. Test your changes.

8.5 Making extensions

The *zuul-bad* project does work. We can execute it, and it correctly does everything that it was intended to do. However, it is in some respects quite badly designed. A well-designed alternative would perform in the same way; we would not notice any difference just by executing the program.

Once we try to make modifications to the project, however, we will notice significant differences in the amount of work involved in changing badly designed code, compared with changes to a well-designed application. We will investigate this by making some changes to the project. While we are doing this, we will discuss examples of bad design when we see them in the existing source, and we will improve the class design before we implement our extensions.

8.5.1 The task

The first task we will attempt is to add a new direction of movement. Currently, a player can move in four directions: *north*, *east*, *south*, and *west.* We want to allow for multilevel buildings (or cellars, or dungeons, or whatever you later want to add to your game) and add *up* and *down* as possible directions. A player can then type **"go down"** to move, say, down into a cellar.

8.5.2 Finding the relevant source code

Inspection of the given classes shows us that at least two classes are involved in this change: **`Room`** and **`Game`**.

`Room` is the class that stores (among other things) the exits of each room. As we saw in Code 8.1, in the **`Game`** class the exit information from the current room is used to print out information about exits and to move from one room to another.

The **`Room`** class is fairly short. Its source code is shown in Code 8.3. Reading the source, we can see that the exits are mentioned in two different places: they are listed as fields at the top of the class, and they get assigned in the **`setExits`** method. To add two new directions, we would need to add two new exits (**`upExit`** and **`downExit`**) in these two places.

Code 8.3
Source code of the (badly designed) `Room` class

```
public class Room
{
    public String description;
    public Room northExit;
    public Room southExit;
    public Room eastExit;
    public Room westExit;

    /**
     * Create a room described "description." Initially, it has
     * no exits. "description" is something like "a kitchen" or
     * "an open court yard."
     * @param description The room's description.
     */
    public Room(String description)
    {
        this.description = description;
    }
```

Code 8.3 continued
Source code of the (badly designed) `Room` class

```
    /**
     * Define the exits of this room.  Every direction either leads
     * to another room or is null (no exit there).
     * @param north The north exit.
     * @param east The east east.
     * @param south The south exit.
     * @param west The west exit.
     */
    public void setExits(Room north, Room east, Room south, Room west)
    {
        if(north != null) {
            northExit = north;
        }
        if(east != null) {
            eastExit = east;
        }
        if(south != null) {
            southExit = south;
        }
        if(west != null) {
            westExit = west;
        }
    }

    /**
     * @return The description of the room.
     */
    public String getDescription()
    {
        return description;
    }
}
```

It is a bit more work to find all relevant places in the **`Game`** class. The source code is somewhat longer (it is not fully shown here), and finding all the relevant places takes some patience and care.

Reading the code shown in Code 8.1, we can see that the **`Game`** class makes heavy use of the exit information of a room. The **`Game`** object holds a reference to one room in the **`currentRoom`** variable, and frequently accesses this room's exit information.

- In the **`createRoom`** method, the exits are defined.
- In the **`printWelcome`** method, the current room's exits are printed out so that the player knows where to go when the game starts.
- In the **`goRoom`** method, the exits are used to find the next room. They are then used again to print out the exits of the next room we have just entered.

If we now want to add two new exit directions, we will have to add the *up* and *down* options in all these places. However, read the following section before you do this.

8.6 Coupling

The fact that there are so many places where all exits are enumerated is symptomatic of poor class design. When declaring the exit variables in the **Room** class, we need to list one variable per exit; in the **setExits** method, there is one if statement per exit; in the **goRoom** method, there is one if statement per exit; in the **printLocationInfo** method, there is one if statement per exit; and so on. This design decision now creates work for us: when adding new exits, we need to find all these places and add two new cases. Imagine the effect if we decided to use directions such as northwest, southeast, etc.!

To improve the situation, we decide to use a **HashMap** to store the exits, rather than separate variables. Doing this, we should be able to write code that can cope with any number of exits and does not need so many modifications. The **HashMap** will contain a mapping from a named direction (e.g., **"north"**) to the room that lies in that direction (a **Room** object). Thus, each entry has a **String** as the key and a **Room** object as the value.

This is a change in the way a room stores information internally about neighboring rooms. Theoretically, this is a change that should affect only the *implementation* of the **Room** class (*how* the exit information is stored), not the *interface* (*what* the room stores).

Ideally, when only the implementation of a class changes, other classes should not be affected. This would be a case of *loose* coupling.

In our example, this does not work. If we remove the exit variables in the **Room** class and replace them with a **HashMap**, the **Game** class will not compile any more. It makes numerous references to the room's exit variables, which all would cause errors.

We see that we have a case here of *tight* coupling. In order to clean this up, we will decouple these classes before we introduce the **HashMap**.

8.6.1 Using encapsulation to reduce coupling

Concept

Proper **encapsulation** in classes reduces coupling and thus leads to a better design.

One of the main problems in this example is the use of public fields. The exit fields in the **Room** class have all been declared **public**. Clearly, the programmer of this class did not follow the guidelines we have set out earlier in this book ("Never make fields public!"). We shall now see the result. The **Game** class in this example can make direct accesses to these fields (and it makes extensive use of this fact). By making the fields public, the **Room** class has exposed in its interface not only the fact that it has exits, but also exactly how the exit information is stored. This breaks one of the fundamental principles of good class design: *encapsulation.*

The encapsulation guideline (hiding implementation information from view) suggests that only information about *what* a class can do should be visible to the outside, not about *how* it does it. This has a great advantage: if no other class knows how our information is stored, then we can easily change how it is stored without breaking other classes.

We can enforce this separation of *what* and *how* by making the fields private, and using an accessor method to access them. The first stage of our modified **Room** class is shown in Code 8.4.

Code 8.4
Using an accessor method to decrease coupling

```
public class Room
{
    private String description;
    private Room northExit;
    private Room southExit;
    private Room eastExit;
    private Room westExit;

    Existing methods unchanged.

    public Room getExit(String direction)
    {
        if(direction.equals("north")) {
            return northExit;
        }
        if(direction.equals("east")) {
            return eastExit;
        }
        if(direction.equals("south")) {
            return southExit;
        }
        if(direction.equals("west")) {
            return westExit;
        }
        return null;
    }
}
```

Having made this change to the **Room** class, we need to change the **Game** class as well. Wherever an exit variable was accessed, we now use the accessor method. For example, instead of writing

```
nextRoom = currentRoom.eastExit;
```

we now write

```
nextRoom = currentRoom.getExit("east");
```

This makes coding one section in the **Game** class much easier as well. In the **goRoom** method, the replacement suggested here will result in the following code segment:

```
Room nextRoom = null;
if(direction.equals("north")) {
    nextRoom = currentRoom.getExit("north");
}
if(direction.equals("east")) {
    nextRoom = currentRoom.getExit("east");
}
if(direction.equals("south")) {
    nextRoom = currentRoom.getExit("south");
}
```

```
if(direction.equals("west")) {
    nextRoom = currentRoom.getExit("west");
}
```

Instead, this whole code segment can now be replaced with:

```
Room nextRoom = currentRoom.getExit(direction);
```

Exercise 8.6 Make the changes we have described to the **Room** and **Game** classes.

Exercise 8.7 Make a similar change to the **printLocationInfo** method of **Game** so that details of the exits are now prepared by the **Room** rather than the **Game**. Define a method in **Room** with the following header:

```
/**
 * Return a description of the room's exits,
 * for example, "Exits: north west".
 * @return A description of the available exits.
 */
public String getExitString()
```

So far, we have not changed the representation of the exits in the **Room** class. We have only cleaned up the interface. The *change* in the **Game** class is minimal—instead of an access of a public field, we use a method call—but the *gain* is dramatic. We can now make a change to the way exits are stored in the room, without any need to worry about breaking anything in the **Game** class. The internal representation in **Room** has been completely decoupled from the interface. Now that the design is the way it should have been in the first place, exchanging the separate exit fields for a **HashMap** is easy. The changed code is shown in Code 8.5.

Code 8.5
Source code of the **Room** class

```
import java.util.HashMap;

Class comment omitted.

public class Room
{
    private String description;
    private HashMap<String, Room> exits;        // stores exits of this room.

    /**
     * Create a room described "description". Initially, it has no exits.
     * "description" is something like "a kitchen" or "an open court yard."
     * @param description The room's description.
     */
    public Room(String description)
    {
        this.description = description;
        exits = new HashMap<>();
    }
```

Code 8.5 continued
Source code of the **Room** class

```
    /**
     * Define an exit from this room.
     * @param direction The direction of the exit.
     * @param neighbor  The room to which the exit leads.
     */
    public void setExit(Room north, Room east, Room south, Room west)
    {
        if(north != null) {
            exits.put("north", north);
        }
        if(east != null) {
            exits.put("east", east);
        }
        if(south != null) {
            exits.put("south", south);
        }
        if(west != null) {
            exits.put("west", west);
        }
    }

    /**
     * Return the room that is reached if we go from this room in direction
     * "direction."  If there is no room in that direction, return null.
     * @param direction The exit's direction.
     * @return The room in the given direction.
     */
    public Room getExit(String direction)
    {
        return exits.get(direction);
    }

    /**
     * @return The description of the room
     * (the one that was defined in the constructor).
     */
    public String getDescription()
    {
        return description;
    }
}
```

It is worth emphasizing again that we can make this change now without even checking whether anything will break elsewhere. Because we have changed only private aspects of the **Room** class, which, by definition, cannot be used in other classes, this change does not impact on other classes. The interface remains unchanged.

A by-product of this change is that our **Room** class is now even shorter. Instead of listing four separate variables, we have only one. In addition, the **getExit** method is considerably simplified.

Recall that the original aim that set off this series of changes was to make it easier to add the two new possible exits in the *up* and *down* direction. This has already become much easier. Because we now use a **HashMap** to store exits, storing these two additional directions will work without any change. We can also obtain the exit information via the **getExit** method without any problem.

The only place where knowledge about the four existing exits (*north*, *east*, *south*, *west*) is still coded into the source is in the **setExits** method. This is the last part that needs improvement. At the moment, the method's header is

```
public void setExits(Room north, Room east, Room south, Room west)
```

This method is part of the interface of the **Room** class, so any change we make to it will inevitably affect some other classes by virtue of coupling. We can never completely decouple the classes in an application; otherwise objects of different classes would not be able to interact with one another. Rather, we try to keep the degree of coupling as low as possible. If we have to make a change to **setExits** anyway, to accommodate additional directions, then our preferred solution is to replace it entirely with this method:

```
/**
 * Define an exit from this room.
 * @param direction The direction of the exit.
 * @param neighbor The room in the given direction.
 */
public void setExit(String direction, Room neighbor)
{
    exits.put(direction, neighbor);
}
```

Now, the exits of this room can be set one exit at a time, and any direction can be used for an exit. In the **Game** class, the change that results from modifying the interface of **Room** is as follows. Instead of writing

```
lab.setExits(outside, office, null, null);
```

we now write

```
lab.setExit("north", outside);
lab.setExit("east", office);
```

We have now completely removed the restriction from **Room** that it can store only four exits. The **Room** class is now ready to store *up* and *down* exits, as well as any other direction you might think of (northwest, southeast, etc.).

Exercise 8.8 Implement the changes described in this section in your own *zuul* project.

8.7 Responsibility-driven design

We have seen in the previous section that making use of proper encapsulation reduces coupling and can significantly reduce the amount of work needed to make changes to an application. Encapsulation, however, is not the only factor that influences the degree of coupling. Another aspect is known by the term *responsibility-driven design.*

Responsibility-driven design expresses the idea that each class should be responsible for handling its own data. Often, when we need to add some new functionality to an application,

we need to ask ourselves in which class we should add a method to implement this new function. Which class should be responsible for the task? The answer is that the class that is responsible for storing some data should also be responsible for manipulating it.

How well responsibility-driven design is used influences the degree of coupling and, therefore, again, the ease with which an application can be modified or extended. As usual, we will discuss this in more detail with our example.

8.7.1 Responsibilities and coupling

Concept

Responsibility-driven design is the process of designing classes by assigning well-defined responsibilities to each class. This process can be used to determine which class should implement which part of an application function.

The changes to the **Room** class that we discussed in Section 8.6.1 make it quite easy now to add the new directions for up and down movement in the **Game** class. We investigate this with an example. Assume that we want to add a new room (the cellar) under the office. All we have to do to achieve this is to make some small changes to **Game**'s **createRooms** method to create the room and to make two calls to set the exits:

```
private void createRooms()
{
    Room outside, theater, pub, lab, office, cellar;
    ...
    cellar = new Room("in the cellar");
    ...
    office.setExit("down", cellar);
    cellar.setExit("up", office);
}
```

Because of the new interface of the **Room** class, this will work without problems. The change is now very easy and confirms that the design is getting better.

Further evidence of this can be seen if we compare the original version of the **printLocationInfo** method shown in Code 8.2 with the **getExitString** method shown in Code 8.6 that represents a solution to Exercise 8.7.

Because information about its exits is now stored only in the room itself, it is the room that is responsible for providing that information. The room can do this much better than any other object, because it has all the knowledge about the internal storage structure of the exit data. Now, inside the **Room** class, we can make use of the knowledge that exits are stored in a **HashMap**, and we can iterate over that map to describe the exits.

Consequently, we replace the version of **getExitString** shown in Code 8.6 with the version shown in Code 8.7. This method finds all the names for exits in the **HashMap** (the keys in the **HashMap** are the names of the exits) and concatenates them to a single **String**, which is then returned. (We need to import `Set` from **java.util** for this to work.)

Exercise 8.9 Look up the **keySet** method in the documentation of **HashMap**. What does it do?

Exercise 8.10 Explain, in detail and in writing, how the **getExitString** method shown in Code 8.7 works.

Code 8.6
The `getExitString` method of `Room`

```
/**
 * Return a description of the room's exits,
 * for example, "Exits: north west."
 * @return A description of the available exits.
 */
public String getExitString()
{
    String exitString = "Exits: ";
    if(northExit != null) {
        exitString += "north ";
    }
    if(eastExit != null) {
        exitString += "east ";
    }
    if(southExit != null) {
        exitString += "south ";
    }
    if(westExit != null) {
        exitString += "west ";
    }
    return exitString;
}
```

Code 8.7
A revised version of `getExitString`

```
/**
 * Return a description of the room's exits,
 * for example "Exits: north west."
 * @return A description of the available exits.
 */
public String getExitString()
{
    String returnString = "Exits:";
    Set<String> keys = exits.keySet();
    for(String exit : keys) {
        returnString += " " + exit;
    }
    return returnString;
}
```

Our goal to reduce coupling demands that, as far as possible, changes to the **Room** class do not require changes to the **Game** class. We can still improve this.

Currently, we have still encoded in the **Game** class the knowledge that the information we want from a room consists of a description string and the exit string:

```
System.out.println("You are " + currentRoom.getDescription());
System.out.println(currentRoom.getExitString());
```

What if we add items to rooms in our game? Or monsters? Or other players?

When we describe what we see, the list of items, monsters, and other players should be included in the description of the room. We would need not only to make changes to the **Room** class to add these things, but also to change the code segment above where the description is printed out.

This is again a breach of the responsibility-driven design rule. Because the **Room** class holds information about a room, it should also produce a description for a room. We can improve this by adding to the **Room** class the following method:

```
/**
 * Return a long description of this room, of the form:
 *     You are in the kitchen.
 *     Exits: north west
 * @return A description of the room, including exits.
 */
public String getLongDescription()
{
    return "You are " + description + ".\n" + getExitString();
}
```

In the **Game** class, we then write

```
System.out.println(currentRoom.getLongDescription());
```

The "long description" of a room now includes the description string and information about the exits, and may in the future include anything else there is to say about a room. When we make these future extensions, we will have to make changes to only the **Room** class.

Concept

One of the main goals of a good class design is that of **localizing change**: making changes to one class should have minimal effects on other classes.

Exercise 8.11 Implement the changes described in this section in your own *zuul* project.

Exercise 8.12 Draw an object diagram with all objects in your game, the way they are just after starting the game.

Exercise 8.13 How does the object diagram change when you execute a *go* command?

8.8 Localizing change

Another aspect of the decoupling and responsibility principles is that of *localizing change.* We aim to create a class design that makes later changes easy by localizing the effects of a change.

Ideally, only a single class needs to be changed to make a modification. Sometimes several classes need change, but we then aim at this being as few classes as possible. In addition, the changes needed in other classes should be obvious, easy to detect, and easy to carry out.

To a large extent, we can achieve this by following good design rules such as using responsibility-driven design and aiming for loose coupling and high cohesion. In addition, however, we should have modification and extension in mind when we create our applications. It is important to anticipate that an aspect of our program might change, in order to make any changes easy.

8.9 Implicit coupling

We have seen that the use of public fields is one practice that is likely to create an unnecessarily tight form of coupling between classes. With this tight coupling, it may be necessary to make changes to more than one class for what should have been a simple modification. Therefore, public fields should be avoided. However, there is an even worse form of coupling: *implicit coupling.*

Implicit coupling is a situation where one class depends on the internal information of another, but this dependence is not immediately obvious. The tight coupling in the case of the public fields was not good, but at least it was obvious. If we change the public fields in one class and forget about the other, the application will not compile any longer, and the compiler will point out the problem. In cases of implicit coupling, omitting a necessary change can go undetected.

We can see the problem arising if we try to add further command words to the game.

Suppose that we want to add the command *look* to the set of legal commands. The purpose of *look* is merely to print out the description of the room and the exits again (we "look around the room"). This could be helpful if we have entered a sequence of commands in a room so that the description has scrolled out of view and we cannot remember where the exits of the current room are.

We can introduce a new command word simply by adding it to the array of known words in the **`validCommands`** array in the **`CommandWords`** class:

```
// a constant array that holds all valid command words
private static final String validCommands[] = {
    "go", "quit", "help", "look"
};
```

This shows an example of good cohesion: instead of defining the command words in the parser, which would have been one obvious possibility, the author created a separate class just to define the command words. This makes it very easy for us to now find the place where command words are defined, and it is easy to add one. The author was obviously thinking ahead, assuming that more commands might be added later, and created a structure that makes this very easy.

We can test this already. When we make this change and then execute the game and type the command *look*, nothing happens. This contrasts with the behavior of an unknown command word; if we type any unknown word, we see the reply

```
I don't know what you mean...
```

Thus, the fact that we do not see this reply indicates that the word was recognized, but nothing happens because we have not yet implemented an action for this command.

We can fix this by adding a method for the *look* command to the **Game** class:

```
private void look()
{
    System.out.println(currentRoom.getLongDescription());
}
```

You should, of course, also add a comment for this method. After this, we only need to add a case for the *look* command in the **processCommand** method, which will invoke the **look** method when the *look* command is recognized:

```
if(commandWord.equals("help")) {
   printHelp();
}
else if(commandWord.equals("go")) {
   goRoom(command);
}
else if(commandWord.equals("look")) {
   look();
}
else if(commandWord.equals("quit")) {
   wantToQuit = quit(command);
}
```

Try this out, and you will see that it works.

Exercise 8.14 Add the *look* command to your version of the zuul game.

Exercise 8.15 Add another command to your game. For a start, you could choose something simple, such as a command *eat* that, when executed, just prints out "`You have eaten now and you are not hungry any more`." Later, we can improve this so that you really get hungry over time and you need to find food.

Coupling between the **Game**, **Parser**, and **CommandWords** classes so far seems to have been very good—it was easy to make this extension, and we got it to work quickly.

The problem that was mentioned before—implicit coupling—becomes apparent when we now issue a help command. The output is

```
You are lost. You are alone. You wander
around at the university.
Your command words are:
go quit help
```

Now we notice a small problem. The help text is incomplete: the new command, *look*, is not listed.

This seems easy to fix: we can just edit the help text string in the **Game's printHelp** method. This is quickly done and does not seem a great problem. But suppose we had not noticed this error now. Did you think of this problem before you just read about it here?

This is a fundamental problem, because every time a command is added, the help text needs to be changed, and it is very easy to forget to make this change. The program compiles and runs, and everything seems fine. A maintenance programmer may well believe that the job is finished and release a program that now contains a bug.

This is an example of implicit coupling. When commands change, the help text must be modified (coupling), but nothing in the program source clearly points out this dependence (thus implicit).

A well-designed class will avoid this form of coupling by following the rule of responsibility-driven design. Because the **CommandWords** class is responsible for command words, it should also be responsible for printing command words. Thus, we add the following method to the **CommandWords** class:

```
/**
 * Print all valid commands to System.out.
 */
public void showAll()
{
    for(String command : validCommands) {
        System.out.print(command + " ");
    }
    System.out.println();
}
```

The idea here is that the **printHelp** method in **Game**, instead of printing a fixed text with the command words, invokes a method that asks the **CommandWords** class to print all its command words. Doing this ensures that the correct command words will always be printed, and adding a new command will also add it to the help text without further change.

The only remaining problem is that the **Game** object does not have a reference to the **CommandWords** object. You can see in the class diagram (Figure 8.1) that there is no arrow from **Game** to **CommandWords**. This indicates that the **Game** class does not even know of the existence of the **CommandWords** class. Instead, the game just has a parser, and the parser has command words.

We could now add a method to the parser that hands the **CommandWords** object to the **Game** object so that they could communicate. This would, however, increase the degree of coupling in our application. **Game** would then depend on **CommandWords**, which it currently does not. Also, we would see this effect in the class diagram: **Game** would then have an arrow to **CommandWords**.

The arrows in the diagram are, in fact, a good first indication of how tightly coupled a program is—the more arrows, the more coupling. As an approximation of good class design, we can aim at creating diagrams with few arrows.

Thus, the fact that **Game** did not have a reference to **CommandWords** is a good thing! We should not change this. From **Game's** viewpoint, the fact that the **CommandWords** class

exists is an implementation detail of the parser. The parser returns commands, and whether it uses a **CommandWords** object to achieve this or something else is entirely up to the parser's implementation.

A better design just lets the **Game** talk to the **Parser**, which in turn may talk to **CommandWords**. We can implement this by adding the following code to the **printHelp** method in **Game**:

```
System.out.println("Your command words are:");
parser.showCommands();
```

All that is missing, then, is the **showCommands** method in the **Parser**, which delegates this task to the **CommandWords** class. Here is the complete method (in class **Parser**):

```
/**
 * Print out a list of valid command words.
 */
public void showCommands()
{
    commands.showAll();
}
```

Exercise 8.16 Implement the improved version of printing out the command words, as described in this section.

Exercise 8.17 If you now add another new command, do you still need to change the **Game** class? Why?

The full implementation of all changes discussed in this chapter so far is available in your code examples in a project named *zuul-better*. If you have done the exercises so far, you can ignore this project and continue to use your own. If you have not done the exercises but want to do the following exercises in this chapter as a programming project, you can use the *zuul-better* project as your starting point.

8.10 Thinking ahead

The design we have now is an important improvement to the original version. It is, however, possible to improve it even more.

One characteristic of a good software designer is the ability to think ahead. What might change? What can we safely assume will stay unchanged for the life of the program?

One assumption that we have hard-coded into most of our classes is that this game will run as a text-based game with terminal input and output. But will it always be like this?

It might be an interesting extension later to add a graphical user interface with menus, buttons, and images. In that case, we would not want to print the information to the text

terminal anymore. We might still have command words, and we might still want to show them when a player enters a help command. But we might then show them in a text field in a window, rather than using `System.out.println`.

It is good design to try to encapsulate all information about the user interface in a single class or a clearly defined set of classes. Our solution from Section 8.9, for example—the `showAll` method in the `CommandWords` class—does not follow this design rule. It would be nice to define that `CommandWords` is responsible for *producing* (but not *printing*!) the list of command words, but that the `Game` class should decide how it is presented to the user.

We can easily achieve this by changing the `showAll` method so that it returns a string containing all command words instead of printing them out directly. (We should probably rename it `getCommandList` when we make this change.) This string can then be printed in the `printHelp` method in `Game`.

Note that this does not gain us anything right now, but we might profit from the improved design in the future.

Exercise 8.18 Implement the suggested change. Make sure that your program still works as before.

Exercise 8.19 Find out what the *model-view-controller* pattern is. You can do a web search to get information, or you can use any other sources you find. How is it related to the topic discussed here? What does it suggest? How could it be applied to this project? (Only *discuss* its application to this project, as an actual implementation would be an advanced challenge exercise.)

8.11 Cohesion

We introduced the idea of cohesion in Section 8.3: a unit of code should always be responsible for one, and only one, task. We shall now investigate the cohesion principle in more depth and analyze some examples.

The principle of cohesion can be applied to classes and methods: classes should display a high degree of cohesion, and so should methods.

8.11.1 Cohesion of methods

When we talk about cohesion of methods, we seek to express the ideal that any one method should be responsible for one, and only one, well-defined task.

We can see an example of a cohesive method in the `Game` class. This class has a private method named `printWelcome` to show the opening text, and this method is called when the game starts in the `play` method (Code 8.8).

Code 8.8
Two methods with a good degree of cohesion

```
/**
 *  Main play routine. Loops until end of play.
 */
public void play()
{
    printWelcome();

    // Enter the main command loop. Here we repeatedly read commands and
    // execute them until the game is over.

    boolean finished = false;
    while (! finished) {
        Command command = parser.getCommand();
        finished = processCommand(command);
    }
    System.out.println("Thank you for playing.  Good bye.");
}
```

```
/**
 * Print out the opening message for the player.
 */
private void printWelcome()
{
    System.out.println();
    System.out.println("Welcome to the World of Zuul!");
    System.out.println("Zuul is a new, boring adventure game.");
    System.out.println("Type 'help' if you need help.");
    System.out.println();
    System.out.println(currentRoom.getLongDescription());
}
```

Concept

Method cohesion
A cohesive method is responsible for one, and only one, well-defined task.

From a functional point of view, we could have just entered the statements from the **`printWelcome`** method directly into the **`play`** method and achieved the same result without defining an extra method and making a method call. The same can, by the way, be said for the **`processCommand`** method that is also invoked in the **`play`** method: this code, too, could have been written directly into the **`play`** method.

It is, however, much easier to understand what a segment of code does and to make modifications if short, cohesive methods are used. In the chosen method structure, all methods are reasonably short and easy to understand, and their names indicate their purposes quite clearly. These characteristics represent valuable help for a maintenance programmer.

8.11.2 Cohesion of classes

The rule of cohesion of classes states that each class should represent one single, well-defined entity in the problem domain.

Concept

Class cohesion
A cohesive class represents one well-defined entity.

As an example of class cohesion, we now discuss another extension to the *zuul* project. We now want to add *items* to the game. Each room may hold an item, and each item has a description and a weight. An item's weight can be used later to determine whether it can be picked up or not.

A naïve approach would be to add two fields to the **`Room`** class: **`itemDescription`** and **`itemWeight`**. We could now specify the item details for each room, and we could print out the details whenever we enter a room.

This approach, however, does not display a good degree of cohesion: the **`Room`** class now describes both a room and an item. It also suggests that an item is bound to a particular room, which we might not wish to be the case.

A better design would create a separate class for items, probably called **`Item`**. This class would have fields for a description and weight, and a room would simply hold a reference to an item object.

Exercise 8.20 Extend either your adventure project or the *zuul-better* project so that a room can contain a single item. Items have a description and a weight. When creating rooms and setting their exits, items for this game should also be created. When a player enters a room, information about an item in this room should be displayed.

Exercise 8.21 How should the information be produced about an item present in a room? Which class should produce the string describing the item? Which class should print it? Why? Explain in writing. If answering this exercise makes you feel you should change your implementation, go ahead and make the changes.

The real benefits of separating rooms and items in the design can be seen if we change the specification a little. In a further variation of our game, we want to allow not only a single item in each room, but an unlimited number of items. In the design using a separate **`Item`** class, this is easy. We can create multiple **`Item`** objects and store them in a collection of items in the room.

With the first, naïve approach, this change would be almost impossible to implement.

Exercise 8.22 Modify the project so that a room can hold any number of items. Use a collection to do this. Make sure the room has an **`addItem`** method that places an item into the room. Make sure all items get shown when a player enters a room.

8.11.3 Cohesion for readability

There are several ways in which high cohesion benefits a design. The two most important ones are *readability* and *reuse*.

The example discussed in Section 8.11.1, cohesion of the **printWelcome** method, is clearly an example in which increasing cohesion makes a class more readable and thus easier to understand and maintain.

The class-cohesion example in Section 8.11.2 also has an element of readability. If a separate **Item** class exists, a maintenance programmer will easily recognize where to start reading code if a change to the characteristics of an item is needed. Cohesion of classes also increases readability of a program.

8.11.4 Cohesion for reuse

The second great advantage of cohesion is a higher potential for reuse.

The class-cohesion example in Section 8.11.2 shows an example of this: by creating a separate **Item** class, we can create multiple items and thus use the same code for more than a single item.

Reuse is also an important aspect of method cohesion. Consider a method in the **Room** class with the following header:

```
public Room leaveRoom(String direction)
```

This method could return the room in the given direction (so that it can be used as the new **currentRoom**) and also print out the description of the new room that we just entered.

This seems like a possible design, and it can indeed be made to work. In our version, however, we have separated this task into two methods:

```
public Room getExit(String direction)
public String getLongDescription()
```

The first is responsible for returning the next room, whereas the second produces the room's description.

The advantage of this design is that the separate tasks can be reused more easily. The **getLongDescription** method, for example, is now used not only in the **goRoom** method, but also in **printWelcome** and the implementation of the *look* command. This is only possible because it displays a high degree of cohesion. Reusing it would not be possible in the version with the **leaveRoom** method.

Exercise 8.23 Implement a *back* command. This command does not have a second word. Entering the *back* command takes the player into the previous room he/she was in.

Exercise 8.24 Test your new command. Does it work as expected? Also, test cases where the command is used incorrectly. For example, what does your program do if a player types a second word after the *back* command? Does it behave sensibly?

Exercise 8.25 What does your program do if you type "back" twice? Is this behavior sensible?

Exercise 8.26 *Challenge exercise* Implement the *back* command so that using it repeatedly takes you back several rooms, all the way to the beginning of the game if used often enough. Use a **`Stack`** to do this. (You may need to find out about stacks. Look at the Java library documentation.)

8.12 Refactoring

Concept

Refactoring is the activity of restructuring an existing design to maintain a good class design when the application is modified or extended.

When designing applications, we should attempt to think ahead, anticipate possible changes in the future, and create highly cohesive, loosely coupled classes and methods that make modifications easy. This is a noble goal, but we cannot always anticipate all future adaptations, and it is not feasible to prepare for all possible extensions we can think of.

This is why *refactoring* is important.

Refactoring is the activity of restructuring existing classes and methods to adapt them to changed functionality and requirements. Often in the lifetime of an application, functionality is gradually added. One common consequence is that, as a side-effect of this, methods and classes slowly grow in length.

It is tempting for a maintenance programmer to add some extra code to existing classes or methods. Doing this for some time, however, decreases the degree of cohesion. When more and more code is added to a method or a class, it is likely that at some stage it will represent more than one clearly defined task or entity.

Refactoring is the rethinking and redesigning of class and method structures. Most commonly, the effect is that classes are split in two or that methods are divided into two or more methods. Refactoring can also include the joining of multiple classes or methods into one, but that is less common than splitting.

8.12.1 Refactoring and testing

Before we provide an example of refactoring, we need to reflect on the fact that, when we refactor a program, we are usually proposing to make some potentially large changes to something that already works. When something is changed, there is a likelihood that errors will be introduced. Therefore, it is important to proceed cautiously; and, prior to refactoring, we should establish that a set of tests exists for the current version of the program. If tests do not exist, then we should first decide how we can reasonably test the functionality of the

program and record those tests (for instance, by writing them down) so that we can repeat the same tests later. We will discuss testing more formally in the next chapter. If you are already familiar with automated testing, use automated tests. Otherwise, manual (but systematic) testing is sufficient for now.

Once a set of tests has been decided, the refactoring can start. Ideally, the refactoring should then follow in two steps:

- The first step is to refactor in order to improve the internal structure of the code, but without making any changes to the functionality of the application. In other words, the program should, when executed, behave exactly as it did before. Once this stage is completed, the previously established tests should be repeated to ensure that we have not introduced unintended errors.
- The second step is taken only once we have reestablished the baseline functionality in the refactored version. Then we are in a safe position to enhance the program. Once that has been done, of course, testing will need to be conducted on the new version.

Making several changes at the same time (refactoring and adding new features) makes it harder to locate the source of problems when they occur.

Exercise 8.27 What sort of baseline functionality tests might we wish to establish in the current version of the game?

8.12.2 An example of refactoring

As an example, we shall continue with the extension of adding items to the game. In Section 8.11.2, we started adding items, suggesting a structure in which rooms can contain any number of items. A logical extension to this arrangement is that a player should be able to pick up items and carry them around. Here is an informal specification of our next goal:

- The player can pick up items from the current room.
- The player can carry any number of items, but only up to a maximum weight.
- Some items cannot be picked up.
- The player can drop items in the current room.

To achieve these goals, we can do the following:

- If not already done, we add a class **`Item`** to the project. An item has, as discussed above, a description (a string) and a weight (an integer).
- We should also add a field **`name`** to the **`Item`** class. This will allow us to refer to the item with a name shorter than that of the description. If, for instance, there is a book in the current room, the field values of this item might be:

  name: **`book`**
  description: **`an old, dusty book bound in gray leather`**
  weight: **`1200`**

If we enter a room, we can print out the item's description to tell the player what is there. But for commands, the name will be easier to use. For instance, the player might then type *take book* to pick up the book.

- We can ensure that some items cannot be picked up, by just making them very heavy (more than a player can carry). Or should we have another **`boolean`** field **`canBePickedUp`**? Which do you think is the better design? Does it matter? Try answering this by thinking about what future changes might be made to the game.
- We add commands *take* and *drop* to pick up and drop items. Both commands have an item name as a second word.
- Somewhere we have to add a field (holding some form of collection) to store the items currently carried by the player. We also have to add a field with the maximum weight the player can carry, so that we can check it each time we try to pick up something. Where should these go? Once again, think about future extensions to help you make the decision.

This last task is what we will discuss in more detail now, in order to illustrate the process of refactoring.

The first question to ask ourselves when thinking about how to enable players to carry items is: Where should we add the fields for the currently carried items and the maximum weight? A quick look over the existing classes shows that the **`Game`** class is really the only place where it can be fit in. It cannot be stored in **`Room`**, **`Item`**, or **`Command`**, because there are many different instances of these classes over time, which are not all always accessible. It does not make sense in **`Parser`** or **`CommandWords`** either.

Reinforcing the decision to place these changes in the **`Game`** class is the fact that it already stores the current room (information about where the player is right now), so adding the current items (information about what the player has) seems to fit with this quite well.

This approach could be made to work. It is, however, not a solution that is well designed. The **`Game`** class is fairly big already, and there is a good argument that it contains too much as it is. Adding even more does not make this better.

We should ask ourselves again which class or object this information should belong to. Thinking carefully about the type of information we are adding here (carried items, maximum weight), we realize that this is information about a *player*! The logical thing to do (following responsibility-driven design guidelines) is to create a **`Player`** class. We can then add these fields to the **`Player`** class, and create a **`Player`** object at the start of the game, to store the data.

The existing field **`currentRoom`** also stores information about the player: the player's current location. Consequently, we should now also move this field into the **`Player`** class.

Analyzing it now, it is obvious that this design better fits the principle of responsibility-driven design. Who should be responsible for storing information about the player? The **`Player`** class, of course.

In the original version, we had only a single piece of information for the player: the current room. Whether we should have had a **`Player`** class even back then is up for discussion.

There are arguments both ways. It would have been nice design, so, yes, maybe we should. But having a class with only a single field and no methods that do anything of significance might have been regarded as overkill.

Sometimes there are gray areas such as this one, where either decision is defensible. But after adding our new fields, the situation is quite clear. There is now a strong argument for a **`Player`** class. It would store the fields and have methods such as **`dropItem`** and **`pickUpItem`** (which can include the weight check and might return false if we cannot carry it).

What we did when we introduced the **`Player`** class and moved the **`currentRoom`** field from **`Game`** into **`Player`** was refactoring. We have restructured the way we represent our data, to achieve a better design under changed requirements.

Programmers not as well trained as us (or just being lazy) might have left the **`currentRoom`** field where it was, seeing that the program worked as it was and there did not seem to be a great need to make this change. They would end up with a messy class design.

The effect of making the change can be seen if we think one step further ahead. Assume that we now want to extend the game to allow for multiple players. With our nice new design, this is suddenly very easy. We already have a **`Player`** class (the **`Game`** holds a **`Player`** object), and it is easy to create several **`Player`** objects and store in **`Game`** a collection of players instead of a single player. Each player object would hold its own current room, items, and maximum weight. Different players could even have different maximum weights, opening up the even wider concept of having players with quite different capabilities—their carrying capability being just one of many.

The lazy programmer who left **`currentRoom`** in the **`Game`** class, however, has a serious problem now. Because the entire game has only a single current room, current locations of multiple players cannot easily be stored. Bad design usually bites back to create more work for us in the end.

Doing good refactoring is as much about thinking in a certain mindset as it is about technical skills. While we make changes and extensions to applications, we should regularly question whether an original class design still represents the best solution. As the functionality changes, arguments for or against certain designs change. What was a good design for a simple application might not be good any more when some extensions are added.

Recognizing these changes and actually making the refactoring modifications to the source code usually saves a lot of time and effort in the end. The earlier we clean up our design, the more work we usually save.

We should be prepared to *factor out* methods (turn a sequence of statements from the body of an existing method into a new, independent method) and classes (take parts of a class and create a new class from it). Considering refactoring regularly keeps our class design clean and saves work in the end. Of course, one of the things that will actually mean that refactoring makes life harder in the long run is if we fail to adequately test the refactored version against the original. Whenever we embark on a major refactoring task, it is essential to ensure that we test well, before and after the change. Doing these tests manually (by creating and testing objects interactively) will get tedious very quickly. We shall investigate how we can improve our testing—by automating it—in the next chapter.

Exercise 8.28 Refactor your project to introduce a separate `Player` class. A `Player` object should store at least the current room of the player, but you may also like to store the player's name or other information.

Exercise 8.29 Implement an extension that allows a player to pick up one single item. This includes implementing two new commands: *take* and *drop*.

Exercise 8.30 Extend your implementation to allow the player to carry any number of items.

Exercise 8.31 Add a restriction that allows the player to carry items only up to a specified maximum weight. The maximum weight a player can carry is an attribute of the player.

Exercise 8.32 Implement an *items* command that prints out all items currently carried and their total weight.

Exercise 8.33 Add a *magic cookie* item to a room. Add an *eat cookie* command. If a player finds and eats the magic cookie, it increases the weight that the player can carry. (You might like to modify this slightly to better fit into your own game scenario.)

8.13 Refactoring for language independence

One feature of the *zuul* game that we have not commented on yet is that the user interface is closely tied to commands written in English. This assumption is embedded in both the **`CommandWords`** class, where the list of valid commands is stored, and the **`Game`** class, where the **`processCommand`** method explicitly compares each command word against a set of English words. If we wish to change the interface to allow users to use a different language, then we would have to find all the places in the source code where command words are used and change them. This is a further example of a form of implicit coupling, which we discussed in Section 8.9.

If we want to have language independence in the program, then ideally we should have just one place in the source code where the actual text of command words is stored and have everywhere else refer to commands in a language-independent way. A programming language feature that makes this possible is *enumerated types*, or *enums*. We will explore this feature of Java via the *zuul-with-enums* projects.

8.13.1 Enumerated types

Code 8.9 shows a Java enumerated type definition called **`CommandWord`**.

Code 8.9
An enumerated type for command words

```
/**
 * Representations for all the valid command words for the game.
 *
 * @author  Michael Kölling and David J. Barnes
 * @version 2016.02.29
 */
public enum CommandWord
{
    // A value for each command word, plus one for unrecognized commands.
    GO, QUIT, HELP, UNKNOWN
}
```

In its simplest form, an enumerated type definition consists of an outer wrapper that uses the word **enum** rather than **class**, and a body that is simply a list of variable names denoting the set of values that belong to this type. By convention, these variable names are fully capitalized. We never create objects of an enumerated type. In effect, each name within the type definition represents a unique instance of the type that has already been created for us to use. We refer to these instances as **CommandWord.GO**, **CommandWord.QUIT**, etc. Although the syntax for using them is similar, it is important to avoid thinking of these values as being like the numeric class constants we discussed in Section 6.14. Despite the simplicity of their definition, enumerated type values are proper objects, and are not the same as integers.

How can we use the **CommandWord** type to make a step toward decoupling the game logic of *zuul* from a particular natural language? One of the first improvements we can make is to the following series of tests in the **processCommand** method of **Game**:

```
if(command.isUnknown()) {
    System.out.println("I don't know what you mean...");
    return false;
}
String commandWord = command.getCommandWord();
if(commandWord.equals("help")) {
    printHelp();
}
else if(commandWord.equals("go")) {
    goRoom(command);
}
else if(commandWord.equals("quit")) {
    wantToQuit = quit(command);
}
```

If **commandWord** is made to be of type **CommandWord** rather than **String**, then this can be rewritten as:

```
if(commandWord == CommandWord.UNKNOWN) {
    System.out.println("I don't know what you mean...");
}
else if(commandWord == CommandWord.HELP) {
    printHelp();
}
```

```
else if(commandWord == CommandWord.GO) {
    goRoom(command);
}
else if(commandWord == CommandWord.QUIT) {
    wantToQuit = quit(command);
}
```

In fact, now that we changed the type to **CommandWord**, we could also use a *switch statement* instead of the series of if statements. This expresses the intent of this code segment a little more clearly.[2]

```
switch (commandWord) {
    case UNKNOWN:
        System.out.println("I don't know what you mean...");
        break;
    case HELP:
        printHelp();
        break;
    case GO:
        goRoom(command);
        break;
    case QUIT:
        wantToQuit = quit(command);
        break;
}
```

Concept

A **switch statement** selects a sequence of statements for execution from multiple different options.

The switch statement takes the variable in the parentheses following the **switch** keyword (**commandWord** in our case) and compares it to each of the values listed after the **case** keywords. When a case matches, the code following it is executed. The **break** statement causes the switch statement to abort at that point, and execution continues after the switch statement. For a fuller description of the switch statement, see Appendix D.

Now we just have to arrange for the user's typed commands to be mapped to the corresponding **CommandWord** values. Open the *zuul-with-enums-v1* project to see how we have done this. The most significant change can be found in the **CommandWords** class. Instead of using an array of strings to define the valid commands, we now use a map between strings and **CommandWord** objects:

```
public CommandWords()
{
    validCommands = new HashMap<>();
    validCommands.put("go", CommandWord.GO);
    validCommands.put("help", CommandWord.HELP);
    validCommands.put("quit", CommandWord.QUIT);
}
```

[2] In fact, string literals can also be used as case values in switch statements, in order to avoid a long sequence of if-else-if comparisons.

The command typed by a user can now easily be converted to its corresponding enumerated type value.

Exercise 8.34 Review the source code of the *zuul-with-enums-v1* project to see how it uses the **CommandWord** type. The classes **Command**, **CommandWords**, **Game**, and **Parser** have all been adapted from the *zuul-better* version to accommodate this change. Check that the program still works as you would expect.

Exercise 8.35 Add a *look* command to the game, along the lines described in Section 8.9.

Exercise 8.36 "Translate" the game to use different command words for the **GO** and **QUIT** commands. These could be from a real language or just made-up words. Do you only have to edit the **CommandWords** class to make this change work? What is the significance of this?

Exercise 8.37 Change the word associated with the HELP command and check that it works correctly. After you have made your changes, what do you notice about the welcome message that is printed when the game starts?

Exercise 8.38 In a new project, define your own enumerated type called **Direction** with values NORTH, SOUTH, EAST, and WEST.

8.13.2 Further decoupling of the command interface

The enumerated **CommandWord** type has allowed us to make a significant decoupling of the user interface language from the game logic, and it is almost completely possible to translate the commands into another language just by editing the **CommandWords** class. (At some stage, we should also translate the room descriptions and other output strings, probably by reading them from a file, but we shall leave this until later.) There is one further piece of decoupling of the command words that we would like to perform. Currently, whenever a new command is introduced into the game, we must add a new value to the **CommandWord**, and an association between that value and the user's text in the **CommandWords** classes. It would be helpful if we could make the **CommandWord** type self-contained—in effect, move the text–value association from **CommandWords** to **CommandWord**.

Java allows enumerated type definitions to contain much more than a list of the type's values. We will not explore this feature in much detail, but just give you a flavor of what is possible. Code 8.10 shows an enhanced **CommandWord** type that looks quite similar to an ordinary class definition. This can be found in the *zuul-with-enums-v2* project.

Code 8.10
Associating command strings with enumerated type values

```
/**
 * Representations for all the valid command words for the game
 * along with a string in a particular language.
 *
 * @author  Michael Kölling and David J. Barnes
 * @version 2016.02.29
 */
public enum CommandWord
{
    // A value for each command word along with its
    // corresponding user interface string.
    GO("go"), QUIT("quit"), HELP("help"), UNKNOWN("?");

    // The command string.
    private String commandString;

    /**
     * Initialize with the corresponding command string.
     * @param commandString The command string.
     */
    CommandWord(String commandString)
    {
        this.commandString = commandString;
    }

    /**
     * @return The command word as a string.
     */
    public String toString()
    {
        return commandString;
    }
}
```

The main points to note about this new version of **`CommandWord`** are that:

- Each type value is followed by a parameter value—in this case, the text of the command associated with that value.
- Unlike the version in Code 8.9, a semicolon is required at the end of the list of type values.
- The type definition includes a constructor. This does not have the word **`public`** in its header. Enumerated type constructors are never public, because we do not create the instances. The parameter associated with each type value is passed to this constructor.
- The type definition includes a field, **`commandString`**. The constructor stores the command string in this field.
- A **`toString`** method has been used to return the text associated with a particular type value.

With the text of the commands stored in the **CommandWord** type, the **CommandWords** class in *zuul-with-enums-v2* uses a different way to create its map between text and enumerated values:

```
validCommands = new HashMap<String, CommandWord>();
for(CommandWord command : CommandWord.values()) {
    if(command != CommandWord.UNKNOWN) {
        validCommands.put(command.toString(), command);
    }
}
```

Every enumerated type defines a **values** method that returns an array filled with the value objects from the type. The code above iterates over the array and calls the **toString** method to obtain the command **String** associated with each value.

Exercise 8.39 Add your own *look* command to *zuul-with-enums-v2*. Do you only need to change the **CommandWord** type?

Exercise 8.40 Change the word associated with the *help* command in **CommandWord**. Is this change automatically reflected in the welcome text when you start the game? Take a look at the **printWelcome** method in the **Game** class to see how this has been achieved.

8.14 Design guidelines

An often-heard piece of advice to beginners about writing good object-oriented programs is, "Don't put too much into a single method" or "Don't put everything into one class." Both suggestions have merit, but frequently lead to the counter-questions, "How long should a method be?" or "How long should a class be?"

After the discussion in this chapter, these questions can now be answered in terms of cohesion and coupling. A method is too long if it does more than one logical task. A class is too complex if it represents more than one logical entity.

You will notice that these answers do not give clear-cut rules that specify exactly what to do. Terms such as *one logical task* are still open to interpretation, and different programmers will decide differently in many situations.

These are *guidelines*—not cast-in-stone rules. Keeping these in mind, though, will significantly improve your class design and enable you to master more complex problems and write better and more interesting programs.

It is important to understand the following exercises as suggestions, not as fixed specifications. This game has many possible ways in which it can be extended, and you are encouraged to invent your own extensions. You do not need to do all the exercises here to create an interesting game; you may want to do more,

or you may want to do different ones. Here are some suggestions to get you started.

Exercise 8.41 Add some form of time limit to your game. If a certain task is not completed in a specified time, the player loses. A time limit can easily be implemented by counting the number of moves or the number of entered commands. You do not need to use real time.

Exercise 8.42 Implement a trapdoor somewhere (or some other form of door that you can only cross one way).

Exercise 8.43 Add a *beamer* to the game. A beamer is a device that can be *charged* and *fired*. When you charge the beamer, it memorizes the current room. When you fire the beamer, it transports you immediately back to the room it was charged in. The beamer could either be standard equipment or an item that the player can find. Of course, you need commands to charge and fire the beamer.

Exercise 8.44 Add locked doors to your game. The player needs to find (or otherwise obtain) a key to open a door.

Exercise 8.45 Add a transporter room. Whenever the player enters this room, he/she is randomly transported into one of the other rooms. Note: Coming up with a good design for this task is not trivial. It might be interesting to discuss design alternatives for this with other students. (We discuss design alternatives for this task at the end of Chapter 11. The adventurous or advanced reader may want to skip ahead and have a look.)

8.15 Summary

In this chapter, we have discussed what are often called the *nonfunctional aspects* of an application. Here, the issue is not so much to get a program to perform a certain task, but to do this with well-designed classes.

Good class design can make a huge difference when an application needs to be corrected, modified, or extended. It also allows us to reuse parts of the application in other contexts (for example, for other projects) and thus creates benefits later.

There are two key concepts under which class design can be evaluated: coupling and cohesion. Coupling refers to the interconnectedness of classes, cohesion to modularization into appropriate units. Good design exhibits loose coupling and high cohesion.

One way to achieve a good structure is to follow a process of responsibility-driven design. Whenever we add a function to the application, we try to identify which class should be responsible for which part of the task.

When extending a program, we use regular refactoring to adapt the design to changing requirements and to ensure that classes and methods remain cohesive and loosely coupled.

Terms introduced in this chapter:

code duplication, coupling, cohesion, encapsulation, responsibility-driven design, implicit coupling, refactoring

Concept summary

- **coupling** The term *coupling* describes the interconnectedness of classes. We strive for *loose coupling* in a system—that is, a system where each class is largely independent and communicates with other classes via a small, well-defined interface.
- **cohesion** The term *cohesion* describes how well a unit of code maps to a logical task or entity. In a *highly cohesive* system, each unit of code (method, class, or module) is responsible for a well-defined task or entity. Good class design exhibits a high degree of cohesion.
- **code duplication** Code duplication (having the same segment of code in an application more than once) is a sign of bad design. It should be avoided.
- **encapsulation** Proper encapsulation in classes reduces coupling and thus leads to a better design.
- **responsibility-driven design** Responsibility-driven design is the process of designing classes by assigning well-defined responsibilities to each class. This process can be used to determine which class should implement which part of an application function.
- **localizing change** One of the main goals of a good class design is that of localizing change: making changes to one class should have minimal effects on other classes.
- **method cohesion** A cohesive method is responsible for one, and only one, well-defined task.
- **class cohesion** A cohesive class represents one well-defined entity.
- **refactoring** Refactoring is the activity of restructuring an existing design to maintain a good class design when the application is modified or extended.
- **switch statement** A switch statement selects a sequence of statements for execution from multiple different options.

Exercise 8.46 *Challenge exercise* In the `processCommand` method in `Game`, there is a switch statement (or a sequence of if statements) to dispatch commands when a command word is recognized. This is not a very good design, because every time we add a command, we have to add a case here. Can you improve this design? Design the classes so that handling of commands is more modular and new commands can be added more easily. Implement it. Test it.

Exercise 8.47 Add characters to the game. Characters are similar to items, but they can talk. They speak some text when you first meet them, and they may give you some help if you give them the right item.

Exercise 8.48 Add moving characters. These are like other characters, but every time the player types a command, these characters can move into an adjoining room.

Exercise 8.49 Add a class called **`GameMain`** to your project. Define just a **`main`** method within it and have them method create a **`Game`** object and call its **`play`** method.