7.5 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. Just by executing the program we would not notice any difference.

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 changing 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.

7.5.1 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.

7.5.2 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, and, as we saw in Code 7.1, 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 7.3. 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 7.3
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".
     */
    public Room(String description)
    {
        this.description = description;
    }

    /**
     * Define the exits of this room. Every direction either
     * leads to another room or is null (no exit there).
     */
    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 (the one that was
     * defined in the constructor).
     */
    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 shown fully here), and finding all the relevant places takes some patience and care.

Reading the code shown in Code 7.1, 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.

7.6 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.

7.6.1 Using encapsulation to reduce coupling

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*.

Concept:

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

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 7.4.

Code 7.4
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 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);
```