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

7.7 Responsibility-driven design

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.

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.

7.7.1 Responsibilities and coupling

The changes to the `Room` class that we discussed in Section 7.6.1 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 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 the `Game`'s `createRooms` method to create the room, and make two calls to set the exits:

```
private void createRooms()
{
    Room outside, theatre, 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. This 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 7.2 with the getExitString method shown in Code 7.6 that represents a solution to Exercise 7.7.

Code 7.6
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;
}
```

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, since 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 7.6 with the version shown in Code 7.7. 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 the classes Set and Iterator from java.util for this to work.)

Exercise 7.9 Look up the keySet method in the documentation of HashMap. What does it do?

Exercise 7.10 Explain, in detail and in writing, how the getExitString method shown in Code 7.7 works.

Code 7.7
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;
}
```