CHAPTER

7 Designing classes

Main concepts discussed in this chapter:

- responsibility-driven design
- coupling
- cohesion
- refactoring

Java constructs discussed in this chapter:

`static` (for methods), `Math`, enumerated types

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 often 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 with 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 done very well as a group project.

7.1 Introduction

It is possible to implement an application and to get it to perform its task with badly designed classes. Just executing a finished application does not usually indicate whether it is structured well 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 already occurs during the original 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, or contain bugs, or take a lot longer to build than necessary. 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 10 years ago. In this situation, a software company cannot afford to have badly structured code.

Since 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 simple, 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 places 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 has examples of bad design decisions in it, and we will 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 the 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.

7.2 The *world-of-zuul* game example

From Exercise 7.1, 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 7.1.

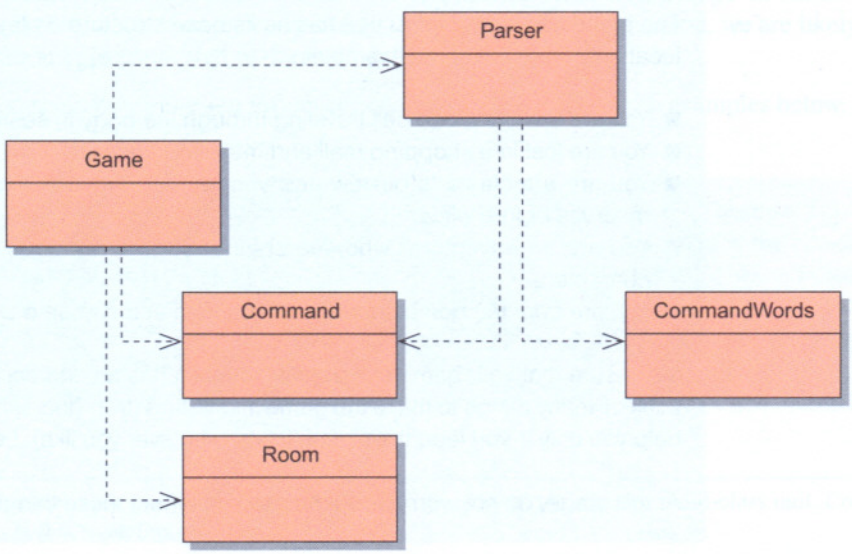

Figure 7.1
Zuul class diagram

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 that 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 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 the game up, and then enters a loop to read and execute commands. It also contains the code that implements each user command.