Designing Classes

Java version

Objectives

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.

Main concepts discussed in this chapter

- responsibility-driven design
- cohesion
- coupling
- refactoring

Resources

- Classes needed for this lab - *chapter08.jar*.
- Background on computer adventure games - You are in a twisty maze of passageways, all alike...
  - http://jerz.setonhill.edu/if/canon/Adventure.htm
  - http://www.rickadams.org/adventure

To do

Introduction

Bad code may work, even correctly, but it is generally very hard to maintain. Here we begin with working code and try to add or change its functionality; this will make it evident that the original is really not very good. We look at a better version to see how the project should have been designed.

From bad to better design

Exercises

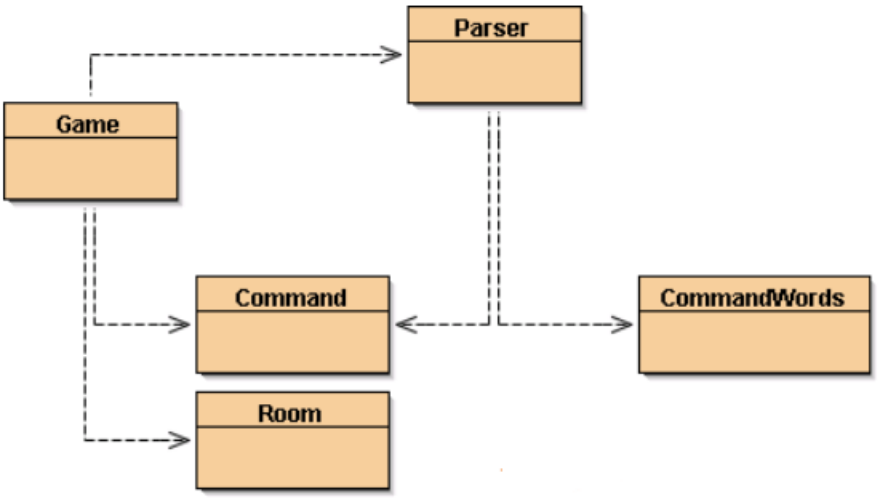
1. Open the *zuul.bad* package. (This package is called bad because its implementation contains some bad design decisions, and we want to leave no doubt that this should not be used as an example of good programming practice!) Execute and explore the application. The project comments gives you some information 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.
2. After you know what the whole application does, try to find out what each individual class does. Write down for each class the purpose of the class. 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.

The *world-of-zuul* game example

As is, the game will not be a commercial success – we've got work to do.

Start by looking at its structure.



It consists of five classes:

- **`CommandWords`**: The **`CommandWords`** class defines all valid commands in the game. It does this by holding an array of strings with 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.

Exercises

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 from 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.

Introduction to coupling and cohesion

Loose coupling means that classes are not particularly interdependent. This is good – we can understand each class without having to look at a whole load of other classes. It also makes it easier to change a class without the change propagating all over the application.

Cohesion refers to what a unit of code (class or method) is responsible for. High cohesion means that the unit does one thing, and one thing only. This is good.

Exercises

4. Draw (on paper) a map for the game you invented in the previous exercise. Open the *zuul.bad* project, and save it under a different name (e.g. *zuul*). This is the project you will use to make improvements and modifications throughout this chapter. You can leave off the *.bad* suffix, since it will soon (hopefully) 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. Test!

Code duplication

This, on the other hand, is bad. Bad. Bad. The problem is that changes must be made in several places to maintain consistency. This increases developer work and can easily lead to inconsistent code. The following code contains a certain amount of duplicated code:

```
package zuul.bad;

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");

        // initialise 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("World of 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;
        }

        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 (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 **printWelcome** and **goRoom** contain the following 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();
```

The problem is that these methods are not cohesive enough, they do two things. In addition to what is indicated in their name, they print the player's current location. This would be better in its own 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

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

Making extensions

zuul.bad works, but it's no fun to modify.

The task

We'll add two new directions of movement, *up* and *down.*

Finding the relevant source code

Exits are mentioned in two classes, **`Room`** (given below) and **`Game`**.

```
package zuul.bad;

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;
    }
```

```
    /**
     * 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.
     */
    public String getDescription() {
        return description;
    }
}
```

Room seems simple enough to modify. For **Game** this may be more complicated.

- 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.

Coupling

The problem with **Room** is that the different exit directions are stored as variables. If we add the two new directions, we need to add two new variables. And don't get us started on diagonal directions!

We would be better off to use a **HashMap** data structure to store rooms by keywords indicating the directions. Ideally, such a change would be local to **Room**, and other classes would not be affected. This is loose coupling, but unfortunately this is not the case in this implementation as the **Game** class uses **Room**'s public variables **northExit**, **southExit**, etc. **Game** knows too much about the internal organization of **Room**. This makes modification time-consuming.

Using encapsulation to reduce coupling

Game should never have had access to **Room**'s variables. This breaks encapsulation and reveals the implementation of **Room**. A class should only ever reveal what it does and never how it does it; bad **Room**. We change **Room** to at first just hide its variables and add an accessor:

```
class Room {
    private final String description;
    private Room northExit;
```

```
    private Room southExit;
    private Room eastExit;
    private Room westExit;

    Existing methods unchanged.

    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;
    }
}
```

In **Game**, this allows us to write

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

and to replace several lines of code in the **goRoom** method with

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

Exercises

6. Make the changes we have described to the **Room** and **Game** classes.
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 signature:

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

Ok, what did we just accomplish? We have decoupled the internal implementation of Room from its interface. It's now possible to change the implementation without breaking anything that relies on the interface. So we could replace the individual exit fields by a data structure, eg,

```
import java.util.HashMap;

class Room {
    private final String description;
```

```
    private final 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".
     */
    Room(String description) {
        this.description = description;
        exits = new HashMap<>();
    }

    /**
     * Define the exits of this room. Every direction either leads to
another
     * room or is null (no exit there).
     */
    void setExits(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 description of the room (the one that was defined in
     * the constructor).
     */
    String getDescription() {
        return description;
    }

    /**
     * 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.
     */
    Room getExit(String direction) {
        return exits.get(direction);
    }
}
```

Now most of **Room** has become independent of specific exit directions, and we can get rid of the last dependence by defining a method **setExit**

```
    /**
     * Define an exit from this room.
     * @param direction The direction of the exit.
     * @param neighbor The room to which the exit leads.
     */
```

```
void setExit(String direction, Room neighbor) {
    exits.put(direction, neighbor);
}
```

In **Game**, we can replace

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

by

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

and adding new directions like *up* and *down* becomes trivial.

Exercises

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

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. However, responsibility-driven design can also make life easier. This is the idea that a class which stores some data should also be responsible for manipulating it, which can decrease coupling and increase cohesion.

Responsibilities and coupling

Adding a new room in the down direction can be done with

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

Since the exits are stored in a **HashMap**, we can use iteration to get the exit strings:

```
private String getExitString() {
    String returnString = "Exits:";
    Set<String> keys = exits.keySet();
    for (String exit : keys) {
        returnString += " " + exit;
    }
    return returnString;
}
```

Exercises

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

10. Explain, in detail, how the **getExitString** method shown in the above code works.

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

Currently, we still have 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 zombies?

When we describe what we see, the list of items, and zombies, 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. Since 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.
 */
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, 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 a single class: the **Room** class.

Exercise

11. Implement the changes described in this section in your own *zuul* project.
12. Draw an object diagram with all objects in your game, the way they are just after starting the game.
13. How does the object diagram change when you execute a **go** command?

Localizing change

Decoupling and responsibility driven design lead naturally to the result that most changes only have local effects, minimizing what has to in fact be changed.

Implicit coupling

This can be a more subtle form of coupling.

Suppose we want to add the command word *look* to allow the player to look around a room. It seems easy enough to change the **CommandWords** class

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

If we write the command *look* to the game...nothing happens. We need to add a method to handle this command in the **Game** class:

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

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);
}
```

Exercises

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

So far so good, the game works. However, the *help* command prints

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

But no *look* command! This is an example of implicit coupling. Every time a command is added, the help text must be updated. This can easily lead to inconsistencies.

Responsibility-driven design says classes should be responsible for manipulating their data, and so **CommandWords** should print its command words, eg, with the method

```
/**
* Print all valid commands to System.out.
*/
void showAll() {
    for(String command : validCommands) {
```

```
            System.out.print(command + " ");
        }
        System.out.println();
    }
```

Problem solved...almost. **Game** doesn't know about **CommandWords** so it can't print the commands. We could introduce **CommandWords** directly into **Game**, but this would increase coupling (one more arrow would be added to the diagram at the beginning of this section). It follows that 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.
*/
void showCommands() {
    commands.showAll();
}
```

Exercise

16. Implement the improved version of printing out the command words, as described in this section.
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 the package *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 *zuul.better* as your starting point.

Thinking ahead

There are two somewhat conflicting ideas here. One is that it's useful to think about what might be added in future to the application, and to try to set up the architecture to make this easier. For instance, the user interface is via command line, which is very unlikely to remain in the final version – we'd probably add a graphical user interface. So we'd like to restrict the user interface to a few well-defined modules that can be easily replaced. This is not the case with the actual implementation – **CommandWords#showAll** prints a list of command words to the terminal. It would be nicer 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. The **toString** method should behave similarly.

Exercises

18. Implement the suggested change. Make sure that your program still works as before.
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.)

On the other hand, you shouldn't enthusiastically rush ahead to *implement* a graphical user interface just because the client might like it. If the client hasn't asked for it don't do it – they may instead already have one and you would have wasted your time!

Cohesion

A unit of code does one thing and one thing only. This can apply to methods and to classes.

Cohesion of methods

For example,

```
    /**
     * 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("World of Zuul is a new, incredibly boring
adventure game.");
        System.out.println("Type 'help' if you need help.");
        System.out.println();
        System.out.println(currentRoom.getLongDescription());
    }
```

The play method calls the method **`printWelcome`** to print information for the user instead of having the code in-line. This reduces its size and makes it more readable.

Cohesion of classes

Open the *zuul.better* package and compare its design to the *zuul.bad* package design. Explain the differences between them in terms of code duplication, cohesion, coupling, responsibility-driven design. Help yourself with the available resources.

You are now going to extend the *zuul.better* project. Keep in mind the rule of cohesion of classes. We want to add *items* to the game. A naïve approach would be to add two fields to the **`Room`** class: **`itemDescription`** and **`itemWeight`**. This could work. 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.

Exercises

20. Extend the *zuul.better* package 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 present in this room should be displayed.
21. Think about the following: How should the information about an item present in a room be produced? Which class should produce the string describing the item? Which class should print it? Why? Make changes as necessary to the implentation.

Our approach adds flexibility should the client decide that a room could have multiple items.

Exercises

22. Modify the *zuul.better* package 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.

Cohesion for readability

Your colleague is not going to have an easy time of reading that 100+ line method. Nor trying to maintain it.

In the above, if **`Item`** is a separate class, it's pretty obvious where to start changing its features, if necessary; much harder if they're wrapped up in a **`Room`** as attributes.

Cohesion for reuse

Smaller, more cohesive code units are easier to recycle. An **`Item`** class can be used for building multiple items. The **`getLongDescription`** method can be used in both **`goRoom`** and in **`printWelcome`**.

Exercises

23. Implement a ***back*** command. This command does not have a second word. Entering the ***back*** command takes the player into the last room they were in.
24. Test your new command properly. Test cases where a bad command is entered - do not forget negative testing! What does your program do if a player types a second word after the *back* command?
25. What does your program do if you type *back* twice? Is this behavior sensible?
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.)

Refactoring

Applications grow by adding new stuff. At some point they'll probably become really ugly, even though they still work. This is the time to rethink the code in terms of low coupling and high cohesion and to *refactor* the code.

Refactoring is the rethinking and redesigning of class and method structures, without changing the functionality of the code.. Most commonly the effect is that classes are split into two, or that methods are divided into two or

more methods. Refactoring can also include the joining of classes or methods into one, but that case is less common.

Refactoring and testing

Refactoring means modifying the code, and modifying working code always runs the risk of introducing bugs. Fortunately, you've got tests to verify your code. You do have tests, right?

Okay, if tests do not exist, then the first stage should be to create some tests that will be suitable for conducting regression testing on the refactored version. Only when these tests exist should the refactoring start. Ideally, the refactoring should then follow in two steps:

- The first step is to refactor in order to provide the same functionality as that of the original version. In other words, we restructure the source code to improve its quality, not to change or increase its functionality. Once this stage is completed, the regression tests should be run to ensure that we have not introduced unintended errors.
- The second step is taken only once we have re-established 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. And they will occur.

Exercise

27. What sort of baseline functionality tests might we wish to establish on the current version of the game?

An example of refactoring

As an example, we shall continue with the extension of adding items to the game. In a previous section 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 shorter name than 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 grey 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 fitted in. It cannot be stored in **`Room`** , **`Item`**, or **`Command`**, since 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 be regarded as overkill.

Sometimes there are gray areas like this 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 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 possibly many.

The lazy programmer who left **currentRoom** in the **Game** class, however, has a serious problem now. Since the whole game has only a single current room, current locations of multiple players cannot be easily stored. Bad design usually bites back later 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 anymore 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 test adequately the refactored version against the original version. Whenever we embark on a major refactoring task it is essential to ensure that our existing test coverage is adequate beforehand, and that it is kept up to date through the refactoring process. Bear this in mind as you attempt the following exercises.

Exercises

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.
29. Refactor your project by to implement an extension that allows a player to pick up one single item. This includes implementing two new commands: **take** and **drop**.
30. Extend your implementation to allow the player to carry any number of items.
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.
32. Implement an *items* command that prints out all items currently carried and their total weight.
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.)

Refactoring for language independence

Language independence means creating applications for a global audience. This is not the case for the versions of the game of *zuul* so far. For example, in **CommandWords** the commands are hard-wired English words. To change the commands to another language, we'd have to find all the English words and then actually edit those bits of code.
A much better solution is to gather all the language-dependent features in one place. For this, we'll use a Java feature called *enumerated types* or *enums*, introduced in the two *zuul.withenums* packages.

Enumerated types

The following is a Java enumerated type

```
/**
```

```
 * Representations for all the valid command words for the game.
 *
 * @author Michael Kolling and David J. Barnes
 * @version 2011.08.07
 */
enum CommandWord {
    // A value for each command word, plus one for unrecognised
    // 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 numeric class constants. 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**:

```
        CommandWord commandWord = command.getCommandWord();

        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;
        }
        return wantToQuit;
```

Now we just have to arrange for the user's typed commands to be mapped to the corresponding **CommandWord** values. Open the *zuul.withenums.v1* package 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:

```
    CommandWords() {
        validCommands = new HashMap<String, CommandWord>();
        validCommands.put("go", CommandWord.GO);
```

```
        validCommands.put("help", CommandWord.HELP);
        validCommands.put("quit", CommandWord.QUIT);
    }
```

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

Exercises

34. Review the source code of the *zuul.withenums.v1* package 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 accomodate the change. Check that the program still works as you would expect.
35. Add a **look** command to the game, along with the lines described in Section *Implicit coupling* above.
36. 'Translate' the game to use the French command words for **"go"** and **"quit"** (**"lancer"** and **"quitter"** respectively) for the **GO** and **QUIT** commands. Do you only have to edit the **CommandWords** class to make the change work?
37. Choose a different command word instead of **"help"** for 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?
38. In a new project, define your own enumerated type called **Position** with values **TOP**, **MIDDLE**, and **BOTTOM**.

Further decoupling of the command interface

We still have a dependence on English in the **CommandWords** class with the *key:value* mapping given by **HashMap<String, CommandWord>** where the *key* is an English word. We can move all the English into one place by modifying **CommandWord** as follows in the *zuul.withenums.v2* project

```
package zuul.withenums.v2;

/**
 * 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 2011.08.10
 */
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;

    /**
     * Initialise 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;
    }
}
```

Note that

- Each type value is followed by a string parameter value.
- There is a constructor which associates the parameter value with the **`commandString`** variable. Not that the constructor is not public because it is always invoked automagically by Java.
- The **`toString`** method returns the string associated with each type.

The map is now used differently in **`CommandWords`**

```
CommandWords() {
    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 which returns an array filled with the value objects from the type.

Exercises

39. Add your own ***look*** command to *zuul.withenums.v2*. Do you only need to change the **`CommandWord`** type?
40. Change the word associated with the ***help*** command in **`CommandWord`**. Is this change reflected automatically 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 done.

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 what exactly 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 guidelines 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.

Exercises

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.
42. Implement a trap door somewhere (or some other form of door that you can only cross one way).
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.
44. Add locked doors to your game. The player needs to find (or otherwise obtain) a key to open a door.
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.
46. Challenge exercise In the **`processCommand`** method in **`Game`** there is a sequence of if statements to dispatch commands when a command word is recognized. This is not a very nice design, since every time we add a command, we have to add a case here in this if statement. 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.
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.
48. Add moving characters. These are like other characters, but every time the player types a command, these characters may move into an adjoining room.

Executing on the command line

When our game is finished, we may want to pass it on to others to play. To do this, it would be nice if people could play the game without the need to start any specific IDE. To be able to do this, we need one more thing: *class methods*, which in Java are also referred to as *static methods*.

Class methods

So far, all methods we have seen have been instance methods: they are invoked on an instance of a class. What distinguishes class methods from instance methods is that class methods can be invoked without an instance—having the class is enough.

In a previous section, we discussed class variables. Class methods are conceptually related and use a related syntax (the keyword static in Java). Just as class variables belong to the class rather than to an instance, so do class methods.

A class method is defined by adding the keyword static in front of the type name in the method's signature:

```
public static int getNumberOfDaysThisMonth() {
    ...
}
```

Such a method can then be called by specifying the name of the class in which it is defined, before the dot in the usual dot notation. If, for instance, the above method is defined in a class called **`Calendar`**, the following call

invokes it:

```
int days = Calendar.getNumberOfDaysThisMonth();
```

Note that the name of the class is used before the dot—no object has been created.

Exercises

49. Read the class documentation for class **`Math`** in the package *java.lang*. It contains many static methods. Find the method that computes the maximum of two integer numbers. What is its signature?
50. Why do you think the methods in the **`Math`** class are static? Could they be written as instance methods?
51. Write a test class that has a method to test how long it takes to count from 1 to 100 in a loop. You can use the method **`currentTimeMillis`** from class **`System`** to help with the time measurement.

The `main` method

If we want to start a Java application without any IDE, we need to use a class method. An application starts without any object in existence. Classes are the only things we have initially, so the first method that can be invoked must be a class method.

The Java definition for starting applications is quite simple: the user specifies the class that should be started, and the Java system will then invoke a method called **`main`** in that class. This method must have a specific signature. If such a method does not exist in that class, an error is reported.

Exercises

52. Find out the details of the **`main`** method and add such a method to your **`Game`** class. The method should create a **`Game`** object and invoke the **`play`** method on it.
53. Execute your game.

Limitations of class methods

Because class methods are associated with a class rather than an instance, they have two important limitations. The first limitation is that a class method may not access any instance fields defined in the class. This is logical, because instance fields are associated with individual objects. Instead, class methods are restricted to accessing class variables from their class. The second limitation is like the first: a class method may not call an instance method from the class. A class method may only invoke other class methods defined in its class.

You will find that we make very little use of class methods in the examples in this book.

Summary

In this chapter we have discussed what are often called the non-functional 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.

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 expression 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.

Exercises

54.

55. Can you call a static method from an instance method? Can you call an instance method from a static method? Can you call a static method from a static method? Answer these questions on paper, then create a test project to check your answers and try it. Explain in detail your answers and observations.
56. Can a class count how many instances have been created of that class? What is needed to do this? Write some code fragments that illustrate what needs to be done. Assume that you want a static method called **`numberOfInstances`** that returns the number of instances created.