7.8 Localizing change

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.

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 then we 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 this change easy.

7.9 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 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 more 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 by simply 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, by the way, 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 now very easy for us to 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. However, after making this change, when we 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());
}
```

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.