PR3S02

Instructions

Ce document contient :
(voir aussi la page web http://www.esiee.fr/~bureaud/fi/fihome.html)

- Les objectifs détaillés de cette unité.
- Le cadre « administratif » du projet.
- Les caractéristiques du jeu d'aventure et des exemples de thèmes.
- Le chapitre 7 du livre en français, reprenant les énoncés de la plupart des exercices avec plus de détails, et surtout expliquant les notions qu'il faut acquérir en réalisant le projet.
- La liste officielle de tous les exercices à faire pour mener à bien le projet. Elle est en anglais sauf pour les exercices ajoutés à ceux du livre.
- Les documentations des sources de la première version du projet.
- Un modèle de Compte-Rendu d'Avancement.

Exercises from the book "Objects First with Java", chapter 7, by D.Barnes & M.J.Kölling, modified by D.Bureau.

Exercise 7.1 Open the project *zuul-bad*. (This project 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 comment gives you some information about how to run it.
While exploring the application, answer the following questions:

- **Q** What does this application do?
- **Q** What commands does the game accept?
- **Q** What does each command do?
- **Q** How many rooms are in the scenario?
- **P** Draw a map of the existing rooms.

Exercise 7.2 After you know what the whole application does, try to find out what each individual class does. **P** 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.

Exercise 7.3 **P** 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.

Exercise 7.4 **P** Draw (on paper) a map for the game you invented in exercise 7.3. 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. **J** As a first step, change the `createRooms` method in the `Game` class to create the rooms and exits you invented for your game. Test!

Exercise 7.5 **J** Implement and use a separate `printLocationInfo` method in your project, as discussed in this section. Test your changes.

Exercise 7.6 Make the changes we have described to the `Room` and `Game` classes.

- **J?** 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.
- **J?** We can enforce this separation of *what* and *how* by making the fields private and using an accessor method to access them. Having made this change to the `Room` class, we need to change the `Game` class as well.

Exercice 7.6bis *(voir l'énoncé en dernière page)*

Exercise 7.7 **J** Make a similar change to the `printLocation` method of `Game` so that details of the exits are now prepared by the `Room` rather than the `Game`. **J** Define a method in `Room` with the following signature:

```
/**
* Return a string describing the room's exits,
* for example, "Exits: north west".
*/
public String getExitString()
```

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

- **J?** Now that the design is the way it should have been in the first place, exchanging the separate exit fields for a `HashMap` is easy.

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

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

Exercise 7.11 Implement the changes described in this section in your own *zuul* project.

- **J?** 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.

Exercise 7.12 **P** Draw an object diagram with all java objects in your game, the way they are just after starting the game.

Exercise 7.13 **Q** How does the object diagram change when you execute a `go` command?

Exercise 7.14 **J** Add the *look* command to your version of the zuul game.

Exercise 7.15 **J** 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 anymore*.' Later, we can improve this so that you really get hungry over time and you need to find food.

Exercise 7.16 Implement the improved version of printing out the command words, as described in this section.

- **J?** Since the `CommandWord` class is responsible for command words, it should also be responsible for printing command words.
- **J?** It follows that a better design just lets the `Game` talk to the `Parser`, which in turn may talk to `CommandWords`.

Exercise 7.17 **Q** If you now add another new command, do you still need to change the `Game` class? **Q** Why?

Exercise 7.18 Implement the suggested change. Make sure that your program still works as before.

- **J?** We can easily achieve this by changing the `showAll` method so that it returns a string containing all command words instead of printing them out directly. (We should probably rename it `getCommandList` when we make this change.)

Exercice 7.18bis *(voir l'énoncé en dernière page)*

Exercise 7.19 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. **Q** How is it related to the topic discussed here? **Q** What does it suggest? **Q** 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.)

Exercice 7.19bis *(voir l'énoncé en dernière page)*

Exercise 7.20 **J** Extend either your adventure project, or the *zuul-better* project, so that a room can contain a single item. Items have a description and a weight (and/or a price). **J** When creating rooms and setting their exits, items for this game should also be created. **J** When a player enters a room, information about an item present in this room should be displayed.

Exercise 7.21 **Q** How should the information about an item present in a room be produced? **Q** Which class should produce the string describing the item? **Q** Which class should print it? **Q** Why? Explain in writing. **J** If answering this exercise makes you feel you should change your implementation, go ahead and make the changes.

Exercise 7.22 **J** Modify the project so that a room can hold any number of items. Use a collection to do this. **J** Make sure the room has an `addItem` method that places an item into the room. **J** Make sure all items get shown when a player enters a room.

Exercise 7.23 **J** Implement a *back* command. This command does not have a second word. Entering the *back* command takes the player into the last room he/she has been in.

Exercise 7.24 Test your new command properly. Do not forget negative testing! **Q** What does your program do if a player types a second word after the *back* command? **Q** Does it behave sensibly? **Q** Are there more cases of negative testing?

Exercise 7.25 **Q** What does your program do if you type *back* twice? **Q** Is this behaviour sensible?

Exercise 7.26 **J** *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.)

Exercise 7.27 **Q** What sort of baseline functionality tests might we wish to establish on the current version of the game?

Exercise 7.28 **Q** How might tests be automated in a program that reads interactive input? **Q** Is it possible to introduce some form of scripting? **Q** For instance, could a user's input be stored in a file rather than read interactively? **Q** What classes would need to be changed to make this possible?

Exercise 7.29 **J** 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.

Exercise 7.30 **J** Implement an extension that allows a player to pick up one single item. This includes implementing two new commands: **J** *take* and **J** *drop*.

Exercise 7.31 **J** Extend your implementation to allow the player to carry (and/or to buy) any number of items.

Exercise 7.32 **J** Add a restriction that allows the player to carry (and/or to buy) items only up to a specified maximum weight (and/or price). The maximum weight a player can carry is an attribute of the player.

Exercise 7.33 **J** Implement an *items* command that prints out all items currently carried and their total weight.

Exercise 7.34 **J** Add a *magic cookie* item to a room. **J** 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.)

It is important to understand these 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. Dans ce cas, demander préalablement à Denis Bureau son accord pour toute différence souhaitée sur un exercice.

Exercise 7.35 **J** 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.

Exercise 7.36 **J** Implement a trapdoor somewhere (or some other form of door that you can only cross one way).

Exercise 7.37 **J** Add locked doors to your game. The player needs to find (or otherwise obtain) a key to open a door.

Exercise 7.38 **J** 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. **P** It might be interesting to discuss design alternatives for this with other students. (We discuss design alternatives for this task at the end of Chapter 9. The adventurous or advanced reader may want to skip ahead and have a look.)

Exercise 7.39 *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. **Q** Can you improve this design? **J** Design the classes so that handling of commands is more modular, and new commands can be added more easily. Implement it. Test it.

Exercice 7.39bis *(voir l'énoncé en dernière page)*

Exercise 7.40 **J** Add characters to the game. Characters are similar to items, but they can talk. **J** They speak some text when you first meet them, and they may give you some help if you give them the right item.

Exercise 7.41 **J** Add moving characters. These are like other characters, but every time the player types a command, these characters may move into an adjoining room.

Exercice 7.41bis *(voir l'énoncé en dernière page)*

Exercise 7.42 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. **Q** What is its signature?

Exercise 7.43 **Q** Why do you think the methods in the `Math` class are static? **Q** Could they be written as instance methods?

Exercise 7.44 **J** Write a test class that has a method to test how long it takes to count from 1 to 10000000 in a loop. You can use the method `currentTimeMillis` from class `System` to help with the time measurement.

Exercise 7.45 **J** 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. Test the `main` method by invoking it from BlueJ. Class methods can be invoked from the class's popup menu.

Exercise 7.46 **J** Execute your game without BlueJ.

Exercices 7.47, 7.48, et 7.49 *(voir les énoncés en dernière page)*

Légende

Q fournir une réponse à une **Q**uestion
P fournir des explications ou un schéma sur **P**apier
J incorporation en **J**ava dans chaque jeu
C incorporation en **C** dans chaque jeu
? nécessite de se référer au chapitre 7 pour plus de détails

Exercices supplémentaires pour PR3S02 (par Denis Bureau)

Exercice 7.6bis PJ Ajouter une pièce à un niveau différent (plus haut ou plus bas), si ce n'est pas déjà prévu dans le scénario.

Exercice 7.18bis Étudier le projet *zuul-better* et vérifier que votre vous avez bien programmé dans votre jeu toutes les modifications demandées dans les exercices précédents :

- **J CommandWords** : `showAll`
- **J Parser** : `showCommands`
- **J Game** : `(setExit), (getLongDescription), (showCommands), (getExit)`
- **J Room** : `Set, HashMap, Iterator, setExit, getShortDescription, getLongDescription, getExitString, getExit`

Exercice 7.19bis Étudier le projet *zuul-with-images* pour comprendre son fonctionnement et intégrer dans votre jeu cette nouvelle conception qui vous permettra d'opter éventuellement par la suite pour une interface graphique plus élaborée :

- **J GameEngine** : nouveau nom de l'ancienne classe `Game, (UserInterface), setGUI / play,` fichiers `.gif, interpretCommand / processCommand, gui.println / System.out.println, showImage, gui.enable, endGame / quit`
- **J Game, UserInterface** : nouvelles classes
- **J CommandWords** : `showAll`
- **J Parser** : `getCommand, showCommands`
- **J Room** : `imageName, getImageName`

Exercice 7.39bis Étudier le projet *zuul-even-better* et vérifier que la solution que vous avez adoptée dans votre jeu pour l'exercice 7.39 est aussi bonne que celle qui est mise en œuvre dans cette version :

- **J Command** : abstraite, `commandWord`, `isUnknown`, `execute`
- **J GoCommand, HelpCommand, QuitCommand** : nouvelles classes
- **J CommandWords** : `HashMap, Iterator, get, showAll`
- **J Parser** : `getCommand`
- **J Game** : `play, processCommand, printHelp, goRoom, quit`

Exercice 7.41bis Sauvegarde/restauration de l'état du jeu (en Java) :

- **P** Recenser toutes les informations à sauvegarder pour pouvoir retrouver le jeu dans le même état après une interruptionet définir le format de sauvegarde (simple texte ou xml ou ...).
- **J** Vérifier que toutes les classes à sauvegarder possèdent bien une méthode `toString` .
- **J** Écrire une classe Java qui sauvegarde toutes ces données en appelant une future méthode d'écriture dans un fichier de texte. Dans un premier temps, cette méthode affichera simplement à l'écran, pour permettre un débogage plus aisé.
- **P** Tester en vérifiant que toutes les informations nécessaires figurent dans les affichages générés.
- **J** Ajouter la restauration des données sauvegardées, donc recréer les objets nécessaires correctement initialisés, en appelant une future méthode de lecture dans un fichier de texte. Dans un premier temps, cette méthode lira simplement au clavier.

Exercice 7.47 J Ajouter la possibilité d'utiliser le jeu en applette tout en conservant la possibilité de le lancer comme application. Tester l'applette à partir d'une page web personnelle à l'ESIEE. Les affichages à l'écran lors de la sauvegarde et les saisies au clavier lors de la restauration devront se faire dans des zones de textes appropriées.

Exercice 7.48 J Fabriquer un fichier .jar pouvant être utilisé soit comme applette soit comme application. Tester l'applette à partir d'une page web personnelle à l'ESIEE.

Exercice 7.49 Sauvegarde/restauration de l'état du jeu en C :

- **J** Créer une classe Java qui regroupera les appels à des fonctions écrites en C. Ces extensions en C devront être proprement désactivées lorsque le jeu est lancé comme applette.
- **C** Écrire en C les fonctions de sauvegarde nécessaires en utilisant notamment `fprintf` et les fonctions de restauration nécessaires en utilisant notamment `fscanf` . **C** Il est évidemment nécessaire de prévoir des fonctions d'ouverture et de fermeture du fichier texte de sauvegarde.
- **J** Remplacer les affichages/saisies évoqués à l'exercice 7.41bis par des appels aux fonctions C d'écriture/lecture dans un fichier texte, et vérifier le contenu du fichier produit.
- **P** Tester l'ensemble en vérifiant que tout fonctionne normalement en continuant le jeu après une interruption.