CM100227 Coursework 2
Dungeon of Doom - Part 1

November 25, 2016

1 Introduction

The coursework of this unit consists of three parts: the lab sheets and two larger Java courseworks. This document provides the specification for the second large coursework: **Dungeons of Doom**.

You can use any Integrated Development Environment (IDE) for the development of your scripts, but **your scripts have to run when we use the command-line on LCPU without requiring the installation of libraries, modules or other programs**.

Questions regarding the coursework can always be posted on the Moodle Forum and programming1@lists.bath.ac.uk mailing list.

2 Learning Objectives

At the end of this part of the coursework you will be able to design and write medium-sized program using appropriate object oriented software techniques.

3 Dungeons of Doom

For your first coursework, we asked you to investigate and replicate a piece of code that we supplied (SRPN). In this second piece of coursework you will need to design and write a program that allows a single human player and a computer player ('bot') to play the game 'Dungeons of Doom', described below. (Connect4).

3.1 Game overview

The *Dungeon of Doom* is played on a rectangular grid, which serves as the game's board, on which the player acting as a brave fortune-hunter, can move and pick up gold. The goal is to get enough gold and then exit the dungeon. The game is played on rounds. On each round, the player sends a command and if the command is successful, an action takes place. A full list of the available commands, the *game-protocol*, is available below.

3.1.1 Board representation

The dungeon is made out of square tiles. Each tile can be:

1. **Player:** This tile represent a human player. It is displayed as the letter *P*.
2. **Bot:** This tile represent an AI-controlled player, a bot. It is displayed as the letter *B*.
3. **Empty Floor:** Allows a player to walk over it, some may also contain items such as gold. If empty, it is displayed as a *dot(.)*.
4. **Gold:** A special floor tile, allows a player to walk over it and pickup the gold in it. If the player pickups the gold, then the tile is converted into an *empty floor* tile. It is displayed as the letter *G*.
5. **Exit:** A special floor tile, that the player can use to exit the dungeon and win the game. It is displayed as the letter *E*.
6. **Wall:** Blocks the player from moving through it. It is displayed as a *hash sign(#)*.

3.2 Set-up

You start the game with no gold, and at a random location within the dungeon. This position must not contain any existing players/bots or any gold, but it may be an exit tile. You should not be placed inside a wall.

3.3 Protocol commands

Your software should support the human player as they enter and see the results of the following 'game protocol' i.e. the following commands:

> HELLO:
>
> Returns a string, displaying the amount of gold required to be eligible for exit. The format of the reply is: *Gold: ¡number¿*.
>
> `MOVE<direction>`
>
> Moves the player one square in the indicated direction. The direction must be either *N, S, E or W*.
>
> PICKUP:
>
> Pickups the gold on the player's current location. On success, returns the new total of gold collected.

LOOK:

Returns back a 5x5 grid, showing the map around the player. The LOOK window should show walls, empty tiles, Moodle tiles, pieces coursework, other players, and bots each with repented character or symbol. As shown in the example bellow, in the middle of the grid, the player is represented as a P.

```
#.#.E
..P..
#....
X..GX
```

QUIT: Allows the player to quit the game, loosing all progress.

3.4 Code Specifications

We have supplied basic code to get you started with this project (on Moodle). The base code provides an extremely basic definition of fours classes and all the key method signatures within those classes e.g.

```
public class Map {

  /**
   * Returns Gold required to exit the current map.
   */
  protected int getGoldRequired() {
      return 0;
  }

//further code ommitted
```

The four skeleton classes that you need to extend are as follows:

1. Map: You need to extend the class *Map* and its methods. The class Map will be responsible for reading the map from a specified *.txt* file and holding a reference of the current state of the map in memory, stored in a 2D char array. Any gold picked up should be removed from the array.

2. GameLogic: You also need to extend the class *GameLogic* and its methods. The class GameLogic will be responsible for processing the commands send by the player.

3. Human: You need to implement the class *Human* and its methods. Human should contain a *main* method to run the game.

4. Bot: Finally, you need to extend the class *Bot*. This class should contain any code needed by the bot to catch the player. In their most basic form, Bot objects should move randomly around the Map (avoiding walls). More

marks will be given, however, for programs which make Bots actively chase the player (again, avoiding walls). Bots should move once for each Human move.

Your task is to (substantially) extend the code that we have provided such that we can use it to play Dungeon of Doom. More specifically, your code should allow us to start a game of Dungeons of Doom and enter the commands listed in the Protocol section below as many times as we like (and in an order that we chose) until we have completed the game i.e. until we are either caught by the bot or have gathered enough gold and exit the dungeon. You should provide a sample dungeon with your code but we will use multiple dungeons during marking.

There are some fairly strict limitations on the changes that you may make to our basic code:

you must use our four classes

you may not change the method signatures that we have provided e.g. you may not alter "protected int getGoldRequired()" in the examp, above.

you may change the return values in the methods that we have provided but not the return types e.g. you may return values other than 0 in the example, above, but must always return an int.

you may add other "helper" classes and methods (and interfaces / abstract classes) if you wish to but we must be able to test the game without knowing about them. In other words, the additional functionality that you provide must either be written within or called from the classes and methods that we provide.

We value clean code and good practices. You will be marked on the quality of the code/comments provided and the on design decisions that you make.

3.5 Advance Feature: GUI

For extra marks, you can research and implement a basic GUI. At a minimum, that GUI must:

1. Allow the game to operate in a window.
2. Have a control panel consisting of buttons.
3. Have a pane that shows the user the outcome of their actions. This pane can be either console-like or for additional points, a graphic pane which would display the game in a 2D graphical manner using sprites.

GUI code should not replace code from the core assignment.

Please note that we will not be lecturing on Java GUIs as part of CM10227. As the 'Marking scheme', below indicates, it is entirely possible to gain a good pass on this courework without implementing a GUI. We will ,however, give a small number of marks to those who read up on this area for themselves and successfully implement a basic GUI. In other words, we will reward self-learning.

4 Marking Scheme

The below are a rough guide to how you might expect to be marked by an indicative description of what might achieve particular degree categories.

4.1 Marks Table

Criteria	Max Score	Description
Core Specifications		
Functionality satisfying requirements	max 45	Code satisfied the specification to play the core game as a human-player.
Commenting, formatting and clarity	max 30	Code is clearly written and as simple as possible, implements object-oriented programming techniques, and is consistently commented.
Readme	max 5	The Readme.txt submitted contains useful installation, run, and how to play instructions.
Graphical User Interface (GUI)		
Functionality satisfying requirements	max 15	GUI fulfill all specifications and provides at least one additional innovation.
Commenting, formatting and clarity	max 5	Code related to the bot uses clean code practices, object-oriented programming techniques, and is consistently commented.

4.2 Indiciative Marks

4.2.1 Approx 40%:

1. Code satisfied the specification to play the core game as a human-player.
2. Code is properly formatted and commented.
3. Very sparse ReadMe provided (¡1 page).

4.2.2 Approx 60%:

1. Meets all the criteria of a 40% pass.
2. Code architecture follows Object-Oriented Principles.
3. Readme contains comprehensive information on the game and your implementation (1-2 pages).
4. A basic bot is included.

4.2.3 70% plus:

1. Meets all the criteria of a 60% pass.
2. Bot uses non-random movement.
3. Game can be played via a GUI.

5 Submission

You should upload a zip file to Moodle by **5pm on 16th December 2016**. The name of the zip file should be in the form:

CW2-your id.zip

The zip file must contain:

1. A project folder, titled *Project*, containing your source code and any resources files needed. You should not include packages or other subfolders. No compiled code or uneccessary files, e.g. no version control files, should be included.
2. A Readme file containing an introduction to your game, how to install and run the game, and a few words about your code (max 2 pages).

Failure to follow the submission specifications, by providing non-needed files or not following the .zip structured specified, will result in a penalty being applied to the final mark, as a tutor will need to manually edit or remove some of your files.