UNIVERSITY OF BRIGHTON

2013/2014 CI101H Intro to OO Programming

ASSIGNMENT DETAILS:

Assignment Name:	Game of Zuul
Member of staff responsible:	Gerard Allsop
Start/finish:	Hand out of assignment: W\C 17th Feb 2014
	Hand in of assignment: W\C 26th May 2014

Weighting: 25% of module

AIM of Assignment:
Refactor a simple text game taking account of coupling, cohesion and responsibility driven design.

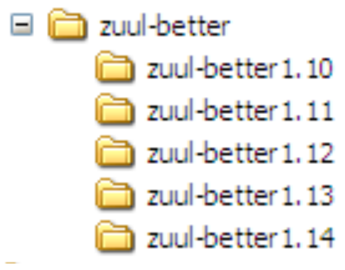
Learning Outcomes/Objectives assessed

LO1	use sequence, selection and iteration to develop simple applications
LO2	solve simple problems by partitioning into components
LO3	detect and correct errors in both logic and syntax LO4 test and debug simple programs
LO5	understand and provide appropriate documentation
LO6	deal with a variety of data types and collections LO7 use data encapsulation

CONTEXT
This assignment utilises the zuul-better project provided in chapter six of Objects First with Java and asks you to develop the game application incrementally, according to a set of defined tasks (see below).

The tasks should be implemented in a manner consistent with the class design techniques (coupling, cohesion, and refactoring) presented in your course book; again see chapter seven.

DELIVERABLES
To complete all tasks, five BlueJ projects should be submitted, according to the task specifications on the following page.


zuul-better

zuul-better1.10	1.10 project addresses grade E requirements
zuul-better1.11	1.11 project addresses grade D requirements
zuul-better1.12	1.12 project addresses grade C requirements
zuul-better1.13	1.13 project addresses grade B requirements
zuul-better1.14	1.14 project addresses grade A requirements

Each deliverable project should:

- summarise the changes made to the previous project in the *readme.txt* file associated with a BlueJ project file,
- provide more details with comments in the appropriate class files.

Assignment Tasks

Where indicated, these tasks should be undertaken with reference to the exercises in chapter 6 of the course book: *Objects First with Java.*

Task 1:

Save your project using its project name suffixed with ‘1.10’.

Add a main method to the *Game* class which constructs a new *Game* object and calls its *play()* method. In other words, the application must be executable outside of the BlueJ environment.

Change the room scenario in the *zuul-better* project provided with *BlueJ***,** to something you find more interesting.

The 1.10 version and all subsequent versions should be executable outside of *BlueJ*.

Task 2:

Save project version 1.10 as 1.11.

Create a method called *display()* in the Game class with a single String parameter, which handles all output to the terminal window produced by the Game class.

Implement the *display()* method so it prints to the terminal window only if a boolean field in the Game class (e.g. ‘debug_mode’) is not true. This will be used when unit testing the Game class.

Task 3

Version 1.11 should include a ***look*** command, which re-displays a description of the current room and its contents.

Task 4

Save version 1.11 of the project to 1.12.

Version 1.12 should allow a single item to be added to a room. Each item has a name, a description, and a weight. The description displayed when a player enters a room, should include the name and weight of the item located in the room**.**

Note: *Before finalising your solution to task 4, take a look at the problem set in task 6. Is your Room class cohesive? Will you be able to extend your solution to cope with an unlimited number of items in each room?*

Task 5:

Extend 1.12 to implement a new command *examine,* followed by the name of an item that exists in the room you are in e.g. **examine brain**. The name and description of the item will be displayed.

Note: *Should the case of the letters be ignored when typing a command?*

Task 6:

Save version 1.12 of the project as 1.13.

Modify version 1.13 of the project to allow a room to hold an unlimited number of items. Any actions that resulted in a description of the room contents, should now describe all of the items in that room.

Task 7

Modify version 1.13 to implement a *back* command, so that using it repeatedly takes you back several rooms.

Task 8

Create a TestGame class which includes a method called testBack() designed to test the operation of the 'back' command. The method should display to the terminal window:

- The test that is running,
- The results of the test (whether it failed or not).

You may need to add some public methods in your Game class so that you are able to invoke them from your test class e.g. the private method:

```
 private boolean processCommand(Command command) {…}
```

could be accessed from the test class, via a new public method:

```
public boolean testProcessCommand(Command command){
        return processCommand(command);
}
```

This approach would avoid changing the existing access modifiers from private to public. The newly added section of test methods could then be commented out, or removed from your final version. Be sure to comment any test methods you add.

To stop interference with the test results, the method should disable printing by the Game class (see task 2).

Task 9 (Re-factoring exercise):

Save version 1.13 of the project to 1.14.

Re-factor version 1.4 to introduce a separate *Player* class. This class should include fields and methods to maintain the state of the current room the player is in, as well as the previous rooms this player has visited.

When starting the game, a list of players could be entered from the command line, or hard-coded into the main method.

During execution of the game, the act of switching players should change the context of the game so that it matches that of the required player e.g. If 'Gerard' is the current player and is in the theatre just before player is switched, the location should change to that next player's last location (outside if this is the new players' first go.) When 'Gerard' takes over play again, the context should revert back to the theatre. In other words, each player will have their own room history.

Task 10:

Implement 3 new commands:

- take,
- drop,
- inventory.

The **take** command should allow the player to pick up an item found in any of the rooms, whilst the **drop** command should allow the player to place a previously taken item into the room they are in. *The inventory command should display a list of items the player is carrying, along with the weight of each item.*

Assessment Criteria

Grade F

- Work submitted, but criteria for grade E are not met.

Grade E (deliverable: project file suffixed 1.10)

- *As per grade F plus*
- Task 1 achieved.

Grade D (deliverable: as for *grade E* plus 1.11 project)

- *As per grade E plus.*
- Task 2 and 3 attempted,
- At least one method is correctly operating,
- ***ALL methods are appropriately commented.***

Grade C (deliverable: as for *grade D* plus 1.12 project)

- *As per grade D plus*
- Task 2 and 3 fully operational,
- Task 4 and 5 attempted,
- At least one of these tasks is operational,
- Code indentation attempted.

Grade B (deliverable: as for *grade C* plus 1.13 project)

- *As per grade C plus*
- Task 4, and 5 operate correctly
- Task 6, 7, and 8 attempted and to a significant extent are operational
- Code indentation style consistently highlights program structure
- JavaDoc styled tags (e.g. @param, @return) used appropriately

Grade A (deliverable: as for *grade B* plus 1.14 project)

- *As per grade B plus*
- Tasks 6,7 and 8 are fully operational,
- The re-factoring required by task 9 has been completed successfully,
- The player name is utilised in the game to change the context,
- The three commands of task 10 are fully implemented.