Curtin University

COMP2003 ASSIGNMENT

Role Playing Game – Combat Tool

Name: Priyank Joshi

Student ID - 17812234

Date: 12/10/2017

Task 1 – Design in UML

These are attached on a separate folder named UML which consists of Whole_UML.png (UML diagram for the whole system), factory.png, strategy.png, observer1.png, observer2.png and observer3.png (these are parts of the whole system to elaborate the design patterns that uses polymorphism).

Task 2 – Coding

This consists of a set of ordinary .java files which are attached on a separate folder named "AssignmentCoding".

The instruction for compiling and running the game (program) is as follows:

1) Download the "Assignment.zip", extract the whole folder, and you may place "AssignmentCoding" folder under oose/Assignment
2) Then open the terminal and make sure you are in the correct directory "cd oose/Assignment/AssignmentCoding" to compile the program.
3) Compile all the .java files by typing "javac *.java"
4) Run the program by typing "java Game"
5) Follow the instructions on the terminal to interact with and play the game. (NB: the files ability.txt and character.txt can be in any directory – you will need to specify in the terminal, the directory it is in when entering the ability and character files)

Task 3 - Polymorphism

I have used factory method pattern that uses polymorphism to decide which subclasses to use between CreateNewGame or LoadExistingGame which share a common super class Read, this provides flexibility and ability to switch between objects. Furthermore, polymorphism was also used in strategy pattern where by the CombatTool is the context and owns the TruthTeller and StoryTeller (strategy objects) which implement a common interface named Side. This pattern is used because CombatTool does not need to know whether the side is Truth Teller, StoryTeller or combination of those and to support extensibility of a broader classification of allies and foes, precisely to support more than two "sides". Additionally, three different observer patterns have been used that demonstrates polymorphism. The first observer pattern is used to produce an event when a character dies, the event source (Character) will notify the observers (TruthTeller and StoryTeller which have many characters and implement a common superclass named DeadCharacterObserver) about a dead character event. If so, then remove that character in story teller or truth teller from being an observer(listener). The second observer pattern is used to generate an event when there is a change in turn from Story Teller to Truth Teller and vice versa. The event source (CombatTool) will know when to notify the observers (TruthTeller and StoryTeller which implement a common superclass named ChangeTurnObserver) about the event. The third observer pattern is used to notify the

CombatTool observer for example whenever no characters are left in StoryTeller side, then show TruthTeller side has won the game and vice versa. Observer patterns were used to enable separation of concerns that is decouple the “when” from “what” as the “when” is handled by the event source and “what” is handles by the observers. These are all capable for supporting for more than two sides.

Refer to factory.png, strategy.png, observer1.png, observer.png and observer3.png for further illustration

Task 4 – Testability

I have used factory design pattern to make unit testing easier. That is in ReadFactory class, I have included a static class field called testRead which is of type Read which you can use it to be the test. Additionally, the setter method, is to set testRead to read and can achieve testing by mocking objects and pass it to the method. When we call readGame (option, ui) method, it contains the normal factory functionality under the if statement. If the testRead equals null then you execute the factory functionality indicating that testing is not achieved hence create the game or load an existing game, else if testRead is not null then this indicates that testing is achieved so read equals to testRead instead and return read of type Read.

Furthermore, I have used dependency injection in almost all classes (CombatTool.java, CreateNewGame.java, LoadExistingGame.java, Game.java, ReadFactory.java, StoryTeller.java, TruthTeller.java, Turn.java) to improve maintainability and testability. This is done by minimising the use of static calls and direct object creation using “new” keyword. Introducing injector code in the Game.java in the main() method that contains direct object creation connecting objects together which is needed for the game operation, otherwise the game functionality will not be able to operate. All in all, the use of dependency injection allows testing to be done easily by creating test harness and mocking objects to perform unit testing.

Task 5 – Alternative Design Choices

A) Another alternate design pattern that would fit into this system would be to have a strategy pattern just for target selection procedure, that is have Heal and Damage as interface classes and having Single (methods relating to single healing and single damage) and Multiple (methods related to multiple healing and multiple damage) as subclasses inherit both these interfaces. The owning object would be Target class. Currently the Turn class has the responsibility of target selection (what happens at each turn for a character). One of the advantage of this design is the ability to change the behaviour at dynamically (making them interchangeable). Also, this will allow addition of different target selections for example selection of 2 or more characters (but not all characters). However, the disadvantage of this design is that it

will increase the number of objects hence tighter coupling between Target object and the Single and Multiple objects.

B) Another alternate design choice that could be used is template method design pattern for the turn process. Specifically, change the Turn class (abstract super class) to have the common functionality such as: character chooses an action (no action or use an ability), then if character uses an ability, he then chooses an ability from the list, and then targetSelection method (single target heal/damage and multi target heal/damage) will be the template method. Then have the TruthTeller and StoryTeller to implement this superclass as they will both have the same functionality from the Turn class. The TruthTeller and StoryTeller can override the method for target selection, as this will vary by one or few steps depending on whom the target applies. The pros for this is that it can support bonuses provided from equipment's/damaging opponent by making bonus () method as template and both the subclasses will override this method. This ensures effective code reuse and reduced code duplication. However, the disadvantage of this pattern is you must understand which methods needs to be overridden, and which ones stay in common superclass as more sides are involved with extra extensibility features. Therefore, it could lead to difficulty in maintainability and understandability.