Plans of the Sorcery Group Project

Yiran Cao & Haochen Wu

Section I: UML diagram

Please refer to uml.pdf.

Section II: Breaking down the project

|  |  |  |
| --- | --- | --- |
| DATE | TASK | CHECK |
| July 22 | Design UML, fill out specified questions (Haochen) |  |
| July 22 | Implement players (Yiran) |  |
| July 18 | Implement game loop, -init command line argument (Yiran) |  |
| July 23 | Implement functionality to load deck from default.deck (Yiran) |  |
| July 23 | Implement cards with no ability (Haochen) |  |
| July 24 | Implement text display (Yiran) |  |
| July 24 | Implement functionality to have a hand of card, and to draw from deck (Yiran) |  |
| July 24 | Implement functionality to start and end turn (Haochen) |  |
| July 24 | Implement minion with no ability, implement functionality to attack player and attack minion (Haochen) |  |
| July 25 | Implement spells which interact with minions (Haochen) |  |
| July 25 | Implement rituals and triggered abilities (Yiran) |  |
| July 25 | Implement simple enchantments (Haochen) |  |
| July 25 | Implement activated abilities (Yiran) |  |
| July 26 | Implement functionality of magic, actions, activation cost (Haochen) |  |
| July 26 | Implement other enchantments (Haochen) |  |
| July 26 | Implement graphic display (Yiran) |  |
| July 27 | Implement -testing command line argument (Yiran) |  |
| July 27 | Implement randomization (Haochen) |  |
| July 27 | Implement -deck1 and -deck2 command line arguments (Yiran) |  |
| July 29 | Test for boundary cases and debug (Yiran and Haochen) |  |
| July 30 | Finalize documentation (Haochen) |  |

Section III: Answers to the questions

Q1: How could you design activated abilities in your code to maximize code reuse?

A: Activated abilities and triggered abilities are both inherited from the class Card, which has a method called playCard with two different set of parameters. By taking the advantage of function overloading, we saved repetitive coding to some extent.

Q2: What design pattern would be ideal for implementing enchantments? Why?

A: Decorator Pattern, because enchantments are added as additional features/modifications of minions, so we can use the decorator pattern to manage (adding and removing) enchantments that are being applied to minions.

Q3: Suppose we found a solution to the space limitations of the current user interface and wanted to allow minions to have any number and combination of activated and triggered abilities. What design patterns might help us achieve this while maximizing code reuse?

A: If this is the case, we may utilize the decorator pattern again to allow minions to have such feature. Similar to what we did with enchantments, each activated and triggered ability will be a concrete class inherited from either an abstract class called activatedAbilityDecorator or triggeredAbilityDecorator. Minion will be the Component and several specific minions will be concrete components.

Q4: How could you make supporting two (or more) interfaces at once easy while requiring minimal changes to the rest of the code?

A: Similar to what we did for assignment 4 question 5, we regard the two players as the subjects and the two observers of the subject are TextDisplay and GraphicDisplay, respectively. By overloading the output operator, we can support two interfaces at once easily.