In The Name Of God

Sharif University of Technology

Computer Science Department

Advanced Programming Project
Phase 3

Hearthstone

Amirhossein Afsharrad

95101077

July 18, 2020

Contents

1 Introduction

In this document we briefly explain the important points of the implementation of Hearthstone project's third phase. The goal for this phase is to complete the implementation of the game, so one can really play it. You may find all packages and classes attached to this report, or you can check them using the following link on GitHub:

https://github.com/AmirAfsharrad/Hearthstone

The java version of this project is java11 and it has been built using *Gradle*. Apart from java standard packages, only the two following dependencies have been added to the `build.gradle` file for the `gson` and `json` respective packages:

- `compile 'com.googlecode.json-simple:json-simple:1.1.1'`
- `compile group: 'com.google.code.gson', name: 'gson', version: '2.8.5'`

All the code has been generated by the author of this report, and no code has been copied from other resources. The resource for the images is also similar to the previous phase of the project.

2 What has changed since phase 2?

The most important advancement of the project is the full implementation of the game logic. Apart from that, there has been some changes made to the graphical user interface. These include the design of the second player on the playground panel, and some slight features such as progress bars for time and quests.

3 The General Structure of the Game Logic

Several elements have been added to the project in order to make the game logic work properly. These elements shall be explained in more detail later, but we are going to go through them in a short review here. First of all, the class `Contestant` has been added to the project to be responsible for each of the two players in the game. To be more clear, each player is actually a `Contestant` with some features and abilities. Using this class, any event that happens during the game for a player is reflected to this class and the proper choice of actions are made using the fields and methods of this class. The two contestants exist within the class `Playground`, which is a `Place` as described in previous reports.

Another useful structure lies within the package `GameLogic`, containing some *visitor* classes responsible for acts such as dealing damage to game objects or restoring their health. Another important class inside this package is `PlayCards`, which is responsible for the procedure of having a card played.

To describe the big picture, game events are transferred from the game panel to the `Playground`, from which appropriate methods of `Contestant`s are called, and these methods usually include calling another method from `PlayCards` or different visitors of the package `GameLogic`.

These packages and classes are explained in more detail in next sections.

4 The Class `Contestant`

This class is the role model for each of the two players playing the game on the playground. There are several fields and methods withing this class, *some* of which are explained below:

4.1 fields

There are many fields inside this class, each holding a game parameter for a player. These include name, mana, passive, hero, deck, hand, planted cards, the state of the player (which is either normal or waiting for target), current spell, current weapon, current quest, etc.

4.2 Methods

There are several methods in this class. Some with more importance are listed below:

- `public void init(Deck inputDeck)`
  This method does all the needed initialization for a player to start the game. There is a deck as the input to this method, which has been selected in the collections panel before starting the game. For the second player who does not select a deck, a default deck has been defined in the `GameHandler`, so that would be the input to this method in case.

  There are other methods used within `init`, including `initDeck`, `initHero`, `initHand`, and `initPassiveProcess`. The latter does what needs to be done in order to apply the effect of the passive selected by the user. Others methods are obvious by their names.

- `public void drawRandomCard(boolean noSpell)`
  This method, as expected from its name, draws a random card from deck to hand. It also has a boolean input which determines whether the drawn card can be a spell or not.

- `public void startTurn()`
  This method is responsible for starting a turn for a contestant. It handles whatever needs to be handled at the start of a turn, such as increasing the mana value (if it has not reached the maximum limit), or discarding some effects remaining from the previous turn of the opponent.

- `public void endTurn()`
  This method is similar to the previous one, but takes the appropriate actions when a player's turn finishes.

- `drawEffectHandle(Card card)`
  Some minions have effects which act at the time of drawing cards. For example, there are minions who would make any drawn cards have less cost than usual. This holds only when the minion is planted. There has been a structure defined to address these effects and this method checks these effects for all planted minions. If there is any, it takes the proper action.

- `private void startTurnMinionEffectHandle()`
  This method is similar to the previous one. The difference is about the time of action.

While the previous method was responsible to take action about the effects of the planted minions at the time of card draw, this method does the same thing at the time of starting a turn.

- `private void startTurnWeaponEffectHandle()`
  This method does the same thing as the previous one, but for the weapons instead of minions.

- `private void endTurnMinionEffectHandle()`
  As described before, the function of this method is almost obvious. It handles the effects of the planted minions when a turn ends.

- `public void checkForDeadMinions()`
  This is a simple but very important and useful method for removing the dead minions from the game panel. Obviously, it is called several times from different places.

- `private void handleQuestCompletion(Card card)`
  This method is responsible to take proper actions when a user finishes a quest successfully. This includes giving the user the predefined reward.

- `public void runHeroPower()`
  This method is run whenever the user chooses to use the hero power of his or her hero. It handles the hero power effect based on the type of the hero.

- `public void initiateAttack(Attacker attacker, int attackValue)`
  This method initiates attack when the user decides to do so. The input is an `Attacker`, an interface implemented by minions and weapons. When a user clicks on a planted minion (and has not been waiting for a target to be received), it initiates an attack. It also can happen by clicking on the active weapon.

- `public void playCard(Card card, int numberOnLeft)`
  This method is responsible for playing a card when it is selected from the hand of the user. There is a class called `PlayCard`, whose methods are called within this method. Actually, this method could be removed and we could have used directly the `PlayCard` class methods, but we have avoided doing so for the sake of respecting the hierarchy. The class `PlayCards` is explained in the next section.

There are many more methods in this class, but they are not as important as the methods explained above. Many of those methods have been declared to do small jobs and do not need to be mentioned here.

5 The Package `GameLogic`

This package contains the classes added to the project in order to implement the game logic in a more structured manner. There are two packages and one class inside this package, which will be explained below.

5.1 The Class `PlayCards`

This class only contains four static methods listed below:

- `public static void playSpell(Spell spell)`
- `public static void playMinion(Card card, int numberOnLeft)`
- `public static void playWeapon(Weapon weapon)`
- `public static void playQuest(Quest quest)`

Obvious from their names, these methods are responsible for playing each of the four types of cards. While the methods for weapons and quests are quite small, the other two have many lines of codes since minions and spells are quite complicated to implement. Further explanations would be too detailed for this report, so we leave that part to the code itself. The only point to be made is about the battlecries of the minions, which have been implemented using the spells. Each battlecry is actually a spell, so we have defined new spell cards for each of the minions who have battlecries, and used this similarity to make our coding more convenient.

5.2 The Package `Visitors`

There are four visitor classes defined in this package:

- `AttackVisitor`
- `DealDamageVisitor`
- `GiveHealthVisitor`
- `ModifyAttackVisitor`

These classes are used for attacking, dealing damage, giving (or restoring) health, and modifying the attack power of the game objects. They use the *visitor* design pattern.

5.3 The Package `Interfaces`

This package contains the interfaces needed for the visitor classes to function properly. There interfaces are listed below:

- `Attackable`
- `Attacker`
- `Damageable`

- `HealthTaker`
- `ModifiableAttack`
- `Target`

The functionality of these interfaces is known by their name. For instance, the `AttackVisitor` operates on `Attackable`, which is implemented by minions and heroes.
Having defined this structure, we have made the process of implementing the game logic much more convenient.