In The Name Of God

Sharif University of Technology

Computer Science Department

Advanced Programming Project
Phase 1

Hearthstone

Amirhossein Afsharrad

95101077

March 23, 2020

Contents

1 Introduction

In this document we briefly explain all packages and classes implemented for the first phase of the Hearthstone game project. The goal for this phase is to implement the basic structure of the game, so the game is not still ready to be played. 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'`

2 How does the project work? - The Big Picture

In this section we are going to explain how the project works in general. The details of implementation are explained in next sections.

To start the game, you need to run the following code:

```
GameHandler.getGameHandler().runGame();
```

The `GameHandler` class is a singleton class who is responsible for maintaining the general running process of the game. The code above will result in the following code to be run:

```
CLI cli = CLI.getGetCLI();
cli.run();
```

The `CLI` class is also another singleton class which is responsible for the processing of the input commands from the user.
After calling the `cli.run()` method, the key process of the game starts using the following infinite `while` loop:

```
while (true){
  try {
    currentPlace.defaultResponse();
    command = GetResponseFromUser.getResponse(user);
    currentPlace = currentPlace.runCommand(command, user, currentPlace);
    } catch (Exception e){
      Logger.log(user, "error", Arrays.toString(e.getStackTrace()), true);
  }
}
```

In the above loop three tasks are continuously done:

1. A default response is given to the user. It depends on the `currentPlace` which is an object determining where in the game we currently are. For example we could be in the main menu, collections, store, etc. and a default response could be: ”You are in the main menu. Where to go?”.
2. The process waits until the user enters a command.
3. The command has to be processed so that two decisions are made:
   (a) Take appropriate actions such as printing a response or changing user attributes (such as cards, current hero, deck, etc.)
   (b) Determine the new `currentPlace`

So by inspecting the above code you would conclude that the key method for running the game is the `runCommand` method called for the `currentPlace`.
In order to gain a better understanding of how this project works, note that there exists an abstract class called `Place` who has different subclasses such as `SignInOrSignUp`, `MainMenu`, `Collections`, and `Store`. Each of these classes are singletons since we need one and only one of each place. The `runCommand` method is also an abstract method of the `Place` class, so each of the places have to override `runCommand` for themselves so that they can process the input

command from the user and take appropriate actions. The output of this method is also a `Place` which determines the next place of the game after the command is processed.
Figure 1 shows a schematic of how the project works.

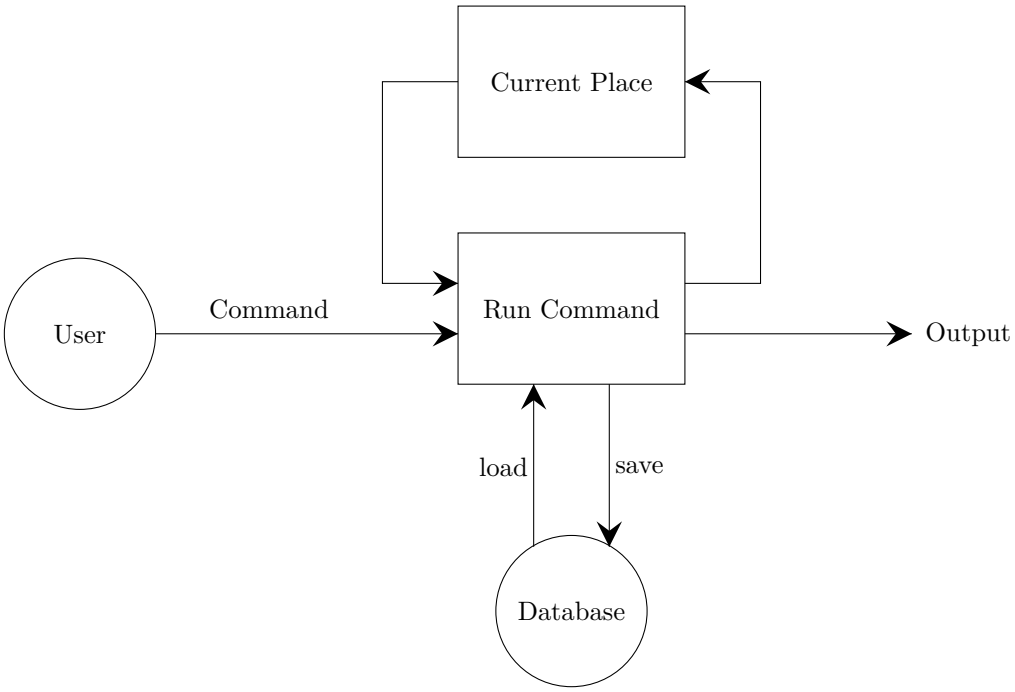

Figure 1: Schematic of the project structure

In the following sections, we explain each package and its classes.

3 Package `Cards`

This package contains classes related to cards of the game. There is an abstract class `Card` which has three subclasses `Minion`, `Spell`, and `Weapon`. There is also another class called `CardCreator` which creates a `Card` object by taking the name of the card as input. The following is a more detailed description of the classes in this package.

3.1 Class `Card`

This class is an implementation for the cards of the game. The class has been declared abstract, since all cards are one of the three subclass types of minion, spell, or weapon and we will not need any direct instances of the abstract type `Card`. The attributes and methods of this class are as follows:

3.1.1 Attributes

- **`private int mana`**
  The mana cost of the card.
- **`private String name`**
  The name of the card.
- **`private String rarity`**
  The rarity of the card, which is one of the four types: common, rare, epic, or legendary.
- **`private String heroClass`**
  The class of hero(es) which could have this card in their deck. It could be Mage, Rogue, Warlock, or neutral.
- **`private String description`**
  Description of the card. For now this field is merely a string, but we may need a class for it in future development of the game.
- **`private String type`**
  This field determines whether the card is a minion, a spell, or a weapon. Since this class is abstract, one could remove this field and use the `getClass()` method on the subclasses of `Card` to know their type, but defining an attribute for type would result in a simpler and more easily-understood code.

3.1.2 Methods

We shall skip the setters and getters and only describe other methods.

- **`public int getPrice()`**
  This method determines the price of the cards. There could be various methods to determine the price, but in this phase we have simply used the card's rarity for this purpose. The price for common, rare, epic, and legendary cards are 10, 20, 30, and 40 (measured in terms of the game's unit of money) respectively.

- `public String toString()`
  We have overridden the `toString()` method to return only the card's name. That is because whenever we need to print a card in our code, we only need to show the card's name. One could tell that we could have used the `getName()` method and avoided overriding the `toString()` method, which is correct but we have done so to make our code simpler and better-understood.

- `public boolean equals(Object object)`
  We have overridden the `equals` method in order to be able to check if two cards are the same. The important point is that for two cards, we know from the game structure that only comparing the names suffices to check whether they are the same or not, and that is what we have done in `equals` method. This is because the cards in this game have unique names and there are no pairs of cards having the same names but different features.

3.2 Classes `Minion`, `Spell`, and `Weapon`

These three classes are the children of `Card` who have a few (or none for `Spell`) more attributes. These attributes are listed below:

- `private int attackPower`
  This is the attack power of the card, which is a shared attribute of minions and weapons.

- `private int hp`
  This is the health point field, which is only an attribute of the minion class and the other two types do not need a health point.

- `private int durability`
  This is a specific feature of weapons, which determines how many times a weapon could be used. The other two types of cards do not need such a feature.

3.3 Class `CardCreator`

The primary goal of this class is to create a card by getting the card name in the input. As it was mentioned before, the cards are uniquely determined by their names and there are no pairs of cards with the same name but different features. This would suggest us to even remove the card classes and only work with cards as strings, but this will result in a lot of problems in future when we are going to use cards in the game (although this approach would make this phase of the project much simpler since we wouldn't have to convert card objects to strings and vice versa for saving and loading as they would have been mere strings).
As a result of this feature, we can create a card by just knowing its name and looking in a database of cards' features. As explained in sections 10, we have such database saved in the file `All Cards.json`. So This class is only responsible to have a method to take the card name, look into the database and extract other features of that specific card, and create an object of the card. That is what happens in the following methods:

- `private static Minion createMinion(JSONObject cardObj)`

This method takes the card to be created as a `JSONObject` and extracts the features of the card and finally creates a `Card` object. Note that this method is specified for minions. There are two more similar methods responsible for spells and weapons. Before going to the next methods, there are two important points to be stated: First, this method is private, and that is because this method (along with the next two who do the same thing for spells and weapons) are used by the public method `createCard` which is is explained below. Second, the input of this method is a `JSONObject`, and that is because our database file is a JSON file. This will also be more clear after going through the method `CreateCard`.

- **`private static Spell createSpell(JSONObject cardObj)`**
  Similar as the previous method but for spell cards.

- **`private static Weapon createWeapon(JSONObject cardObj)`**
  Similar as the previous method but for weapon cards.

- **`public static Card createCard(String cardName)`**
  This method, as we mentioned before, is the only public method of this class which creates a card by taking its name as a string in the input. It first opens the JSON database file and reads the list of all cards as a `JSONArray`, then looks for the card with the specified name. Then it looks at the type of the card and decides which of the three previous methods to call. Now it is more clear why we set the input type of the previous methods to `JSONObject`, since in this method we have a `JSONObject` extracted from a `JSONArray` which needs to turn into a `Card` object.

4 Package `Heroes`

This package contains the implementation of all heroes of the game. There are three types of heroes to be implemented in Hearthstone: Mage, Rogue, and Warlock. For each type, we have designed a class, and also there is an abstract class `Hero` which is extended by all three real hero types.

4.1 Class `Hero`

As it was just mentioned, this is an abstract class who is responsible for the general structure of a hero. The fields and methods of this class are explained below. We skip the setters and getters since their function is obvious.

4.1.1 Attributes

- **`protected int defaultHp`**
  This is the default value for the hero's health point. It has been set to 30 as the project instructions asked for, but it could be set to any other value. It's access modifier has been set to `protected` since other subclasses of `Hero` need to know this value.
- **`private int hp`**
  This is the health point of the hero, which varies during the game-play.
- **`private String type`**
  This field is responsible to determine which type of hero (Mage, Rogue, or Warlock) an object of type `Hero` is. Actually we could do without this feature since `Hero` is an abstract class and the objects to be instantiated are of the types `Mage`, `Rogue`, or `Warlock`; and we could run the `getClass()` method on these objects to know their hero type. However creating a string field of type would result in a simpler and better-looking code.
- **`private String heroPower`**
  This is one of the features for the heroes, about which there is almost nothing implemented in the first phase. As a result, it has been defined in this class as a string field which only contains the description of the hero power. We may define a new class for hero power in future if we need to.
- **`private specialPower`**
  About this field everything is similar to `heroPower`. We may design a separate class for `heroPower` but for now a simple string would suffice.
- **`private ArrayList<Card> deck`**
  This field is an `ArrayList` of cards, which holds the deck for the hero.
- **`private int deckCapacity`**
  This is the maximum number of cards heroes can have in their deck. It has been set to a default value of 15 according to the project instructions.

- `private int maxRepetitiveCardsInDeck`
  This item is the maximum number of the same card that a hero can have in the deck. The default value for this parameter is set to 2 according to the project instructions, but it could easily be modified if needed.

4.1.2 Methods

- `public int getNumberOfCardsInDeck()`
  This method returns the number of the cards present in the hero's deck. This method is useful when the game controller wants to check whether a card could be added to the hero's deck or not.
- `public int getHowManyOfThisCardInDeck(Card card)`
  This method takes a `Card` object as the input and returns how many of that card exists in the hero's deck. It is used to prevent the user from adding more than the specified limit `maxRepetitiveCardsInDeck` of the same card to a hero's deck.
- `public ArrayList<String> getDeckAsArrayOfString()`
  This method returns the hero's deck as an `ArrayList` of Strings. We need to transform the cards to string in order to be able to save them as JSON objects.
- `public JSONObject getJson()`
  This method returns a `JSONObject` containing all attributes of a hero which we need to save when a user logs out of his or her account.

4.2 Classes `Mage`, `Rogue`, and `Warlock`

These three classes are children of the `Hero` class who inherit all its fields and methods. They are very similar so we explain them in the same section. These classes need no more fields than their superclass, and they are only different in `type`, `heroPower`, and `specialPower`. These differences have been handled in the constructors of each of the three `Hero` subclasses. As a result one might think of not creating three different classes for hero types and only using a single hero class. This idea could also work properly but we shall see that having three separate classes has resulted in more simplicity in our design. The only other thing to be mentioned is the `toString()` method which is overridden for these three classes and returns their `type`. It is obvious that we could avoid overriding `toString()` method and user `hero.getType()` whenever needed, but we have done so for simplicity.

4.3 Class `HeroCreator`

This class contains only a single static method:

- `public static Hero createHeroFromJson(JSONObject heroJsonObject)`
  It returns a `Hero` object after taking a `JSONObject` in the input. This method actually does the reverse of the `JSONObject getJson()` method in `Hero` class who would turn a `Hero` into a `JSONObject`. These two work as complements for save and load processes.

5 Package `UserHandle`

This package is responsible for users and their data. It plays the key role in save and load processes, and it also defines the key object `User`. We shall review the two important classes of this package in this section.

5.1 Class `User`

This class is responsible for keeping all the data of a user who is playing the game. The fields and methods of the `User` object are as follows:

5.1.1 Attributes

- **`private int id`**
  The user ID which is unique among all users ever playing the game (even if they remove their account).
- **`private String name`**
  The name that user registers his or her account with.
- **`private String password`**
  The password of the user. Note that, as it shall be discussed in more details later, the game does not store the explicit password of any user, but it stores the hashed form of the passwords using SHA256 hashing method.
- **`private int gold`**
  The amount of gold (money) a user has (which could be used in the store).
- **`private ArrayList<Card> cards`**
  The list of all the cards a user owns. For a newly registered user this list consists of ten cards. The method `initializeCardsAndHeroesAsDefault` is responsible for initializing the cards.
- **`private ArrayList<Hero> heroes`**
  The list of all the heroes a user owns. For a newly registered user this list only consists of Mage.
- **`private int currentHeroIndex`**
  A user must always be aware of his or her current hero. One way to do so was to create a `Hero` object as `currentHero`, but for the sake of code simplicity we used another approach. We only keep the index of the current hero as an index of the `ArrayList heroes`. This makes many things simpler. One important example is that if we created an independent field for `currentHero`, we always had to take care of updating the respective hero in the `heroes` list when a change occurred to the `currentHero`, but working with only an index lets us work directly with the `heroes` list and modify any of the heroes in that list which is being used as the current hero.

- `private boolean loggedIn`
  For any user we would need to know whether it is logged in as the current user or not. This field keeps a `boolean` to answer this question.

5.1.2 Methods

- `public void setCurrentHero(String heroName)`
  This method takes a hero (or hero name) and modifies `currentHeroIndex` to change and set the current hero.

- `public void copy (User user)`
  This method takes a `User` object in the input and copies all its values to the current user. We had to write this method since we needed to modify a `User` object inside a method, while the real object was outside the method body. As a result, we had to change the attributes one by one and we could not only change the reference since what we have inside the method is not the original reference but a copy of that.

- `public boolean hasHero(String hero)`
  This method simply returns a `boolean` to show whether a user has a hero or not.

- `public boolean hasCard(String cardName)`
  This method simply returns a `boolean` to show whether a user has a card or not.

- `public void addHero(Hero hero)`
  This method adds a hero to the user's list of heroes.

- `public ArrayList<String> getHeroesAsArrayOfString()`
  This method returns the list of a user's heroes as a string list. This might come in handy when we want to know all the heroes a user has, while we could do without this method only using `hasHero` method.

- `public void initializeCardsAndHeroesAsDefault()`
  This method is responsible for initializing the cards and heroes of a user. The heroes list is set to a default list only containing a Mage, and the card list is initialized to 10 cards which could be accessed in a file named `Default User Cards.txt`. This file is generated with the methods of the package `Initialization` explained in the section 10.

- `public ArrayList<String> getCardsAsArrayOfString()`
  This method does the same thing as `getHeroesAsArrayOfString()` but for the cards.

- `public void addCard(Card card)`
  This method adds a card to the user's list of cards.

- `public void removeCard(Card card)`
  This method removes a card from the user's list of cards.

- `public boolean isForSale(Card card)`
  This method is responsible to answer the questions "Can the user sell this card?". This actually means that this method has to check whether any of the user's heroes have this

card in their decks or not. If the card is not present in any of the heroes' decks, the user can sell it. If not, he or she has to first remove it from those decks. This method is used in the warning system of the store where one wants to sell or buy cards.

- **`public JSONArray getHerosJsonArray()`**
  This method returns a `JSONArray` containing the data of all heroes of a user. This `JSONArray` is used to save (and later load) the needed data for a user's heroes.

- **`public String toString()`**
  We have also overridden the `toString()` method for `User` object to print its attributes when needed.

5.2 Class `UserDataHandler`

This class is simply the interface of the game and the data that is saved outside the program. There are four main methods in this class for adding, saving, loading, and removing user accounts. There are also some other private methods who help the four main methods perform in a good way. These methods are as follows:

- **`private static int getUserIndexIfExists(String userName)`**
  This method returns the ID of a given user if it is an active user (i.e. a user which has been created and not removed yet). This could help the system load the saved data. Also, it has been declared private since only other methods in this class need to use it.

- **`private static boolean userMatchCheck(String userName, JSONObject user)`**
  This method does a very simple task. It takes a `JSONObject` and a string to check whether the object has the same name as the string or not.

- **`public static User add(User user)`**
  This method adds a user to the system if there is no active user with the same user name. It reads the previously saved data from the `users.json` file, adds a new user to the end of the file, and saves it again.

- **`public static User load(String userName, String password)`**
  This method loads the needed data when a user who has previously created an account wants to sign in. It reads data from `users.json` file after checking whether the account exists and the entered password matches the saved password. As previously stated, the passwords saved in the file `users.json` are hashed, so to check the input password, this method calculates its hash and compares it with what was saved in the database file.

- **`public static void save(User user)`**
  When a user decides to sign out of the game, this method has to save its data. It acts similar to `add` method. It opens the `users.json` file and loads its data into a `JSONArray`, finds the user in that array and replaces its data with the updated data. After that it writes the `JSONArray` into the database file again and the saving process in complete.

- **`public static void remove(String userName, String password)`**
  As we mentioned before, user IDs are unique even after the user is removed. So the `remove`

method only changes an attribute of the user in the data file by adding the exact time of deletion of the account. After this happens, one is allowed to create a new user with the same name as the removed user. This method updates the `users.json` file and adds the deletion timestamp to it in a very similar manner as the `save` method works.

6 Package `Places`

6.1 Class `Place`

This game contains different places. The user is permitted to do certain things in each place. For this phase of the project, we have defined four different places, each of which is taken care of by a separate class. Class `Place` is the supperclass to all places and it has been declared abstract since all real places are the instances of its subclasses.

6.1.1 Attributes

- **`private HashMap<String, String> instructions`**
  This field is a `HashMap`, whose keys are the valid commands a user can enter in this place. The values of this `HashMap` are the instructions for each command. We use this field whenever the user enters the command `hearthstone --help`.

- **`private String instructionsPath`**
  This is the address of the file in which the valid commands and their instructions are saved. We use methods from the package `Initialization` (explained in section 10) to generate these files.

6.1.2 Methods

- **`public void loadInstructions()`**
  This method reads the JSON file from the path `instructionsPath` and loads its data into the `HashMap instructions`. We run this method for each subclass of `Place` in the constructor (with its specific `instructionsPath`).

- **`public abstract Place runCommand(String command, User user, Place currentPlace)`**
  We always need to process the input from the user during the run time of the game. In each place, there must be a `runCommand` method to do this job. That is why we have declared it as an abstract method, so it would be implemented in each of the subclasses according to the set of valid commands in that specific subclass.
  We have set the output type of this method a `Place` so that it will return the place *after the command is processed* as commands could obviously result in a change of place during the game.

- **`public abstract void defaultResponse()`**
  When the game is running, each place needs a default response (for example ”You are in main menu. Where to go?”) to show to the user. This method is responsible to produce this response and it is declared abstract so that all places implement it for themselves.

6.2 Classes `SignInOrSignUp`, `MainMenu`, `Collections`, and `Store`

For this phase of the project, there are four places in the game. For each of these places we have designed a separate class. The most important task for each class is to override the

method `runCommand`. In fact, the key process which makes the CLI a good interface is what happens inside these `runCommand` methods. They are responsible for processing the user input and return appropriate outputs.
We are not going to explain the code in each of these methods in this document, but they are generally in the form of a big `switch case` who checks different cases of the input command.
Also an important point is that all these four classes have been designed using the *singleton* design pattern since for each place, there has to be one and only one instance of that place.

7 Package `GameHandler`

This package contains a class named `GameHandler` and another internal package `CLI`. They are explained in this section.

7.1 Class GameHandler

This class is responsible for general operation of the game. That is why it has been created using the *singleton* design pattern. There has to be one and only one game handler it the game.

7.1.1 Attributes

- **`private static GameHandler gameHandler`**
  We need a `GameHandler` object for the singleton design pattern.
- **`private HashMap<String, Card> allCards`**
  This field is the set of all cards in the game. We could do without this field and create a card whenever needed using the `CardCreator.createCard` method, but since this method needs to read from a file, it will take much more time if it is going to be called many times during the game. To avoid this problem, we only do this once and create an instance of all possible cards and save them in this field, and use the cards whenever needed.
- **`private ArrayList<String> allCardNames`**
  This field keeps the names of all game cards in an `ArrayList` to be accessed whenever needed. It is actually used when listing all possible cards a user can buy in the store.
- **`private String path`**
  This is the address of the cards database from which the information needed for initializing `allCards` and `allCardNames` is extracted.

7.1.2 Methods

- **`public void refreshCardsDatabase()`**
  This method uses the address in `path` to initialize or refresh `allCards` and `allCardNames`. In general, it is run once inside the constructor of the class.
- **`public Card getCard(String cardName)`**
  This method returns a `Card` object by taking the name of the desired card at the input.
- **`public static void runGame()`**
  This method gets the instance of CLI (which is a singleton class) and runs the `run()` method of the CLI, so the game shall run. One could run the game and start playing by just calling `GameHandler.getGameHandler().runGame()`.

7.2 Package `CLI`

This package is responsible for the command line interface of the game. In contains three small but important classes.

7.2.1 Class `CLI`

This is a singleton class who is responsible to maintain the command line interface. Apart from the single `CLI` object and its getter, this class only contains one method:

- **`public void run()`**
  This method uses an infinite `while` loop and performs the following three steps:

  1. Run a default response for the current place of the game.
  2. Get a command from the user.
  3. Process the command and do the appropriate task.

  Doing these three steps in an infinite loop is what makes the game work.

7.2.2 Class `RespondToUser`

This is a simple class which gives the desired response to the user. For this phase it only contains a simple print, but it needs further development for the next phases (graphical interface). It also uses the output command for logging. This simple method has the following signature:

- **`public static void respond(Object message, User user)`**
  Note that it has two inputs, a message and a user. The message is clearly taken to be printed. The user, on the other hand, is only taken as an input for the logging process and has nothing to do with the on-screen printing.

7.2.3 Class `getResponseFromUser`

This class has a single method which is overloaded with four different signatures:

- **`public static String getResponse()`**
- **`public static String getResponse(Object message)`**
- **`public static String getResponse(User user)`**
- **`public static String getResponse(Object message, User user)`**

These methods are simply responsible for taking the input command from the user, but they sometimes use that command for logging, and they sometimes print an output simultaneously. As a result, four different signatures (do the logging or not, and print something before getting the input or not) exist for this method.

8 Package Logger

This package is responsible for logging the game and has a single class `Logger`.

8.1 Class Logger

8.1.1 Attributes

- **`private static String defaultPath`**
  This is the only attribute of this class which stores the default path for saving the logs.

8.1.2 Methods

- **`public static void InitializeNewUserLogFile(User user)`**
  This method creates a new log file for a new user and writes the header of the log file using the name and ID of the user.
- **`public static void log(User user, String event, String eventDescription, boolean... systemLog)`**
  This method takes a `User` as input to know which user's log file is to be updated. It then takes two more inputs `event` and `eventDescription` and writes them in the user's log file. Also there is another input `systemLog` which is an arbitrary `boolean` input that determines whether to register a system log or not. System logs are explained below (the `systemLog` method).
- **`public static void terminateUserLog(User user)`**
  This method is called whenever a user is removed. It opens the user's log file and updates the deletion time in the file.
- **`public static void systemLog(String event, String eventDescription)`**
  Apart from the log files for each user, there is another log file named `hearthstone.log` in which general occurrences of the game are logged. These include user acts and system errors. This method is responsible to log these events.

9 Package `Utilities`

This package contains some helper classes which are needed in different other classes. Note that some of the methods in this package have been taken from online websites whose names the author has unfortunately forgotten.

9.1 Class `FileHandler`

This class contains some useful methods for working with files. They are listed below:

- **`public static List<String> readFileInList(String path)`**
  This method reads a text file (line by line) into a list.
- **`public static String readFileInString(String path)`**
  This method reads a text file into a single string.
- **`public static void writeListIntoFile(List list, String path)`**
  This method writes a list into a text file (line by line).
- **`public static void writeStringIntoFile(String text, String path)`**
  This method writes a string into a text file.
- **`public static void writeJsonObjectToFile(JSONObject jsonObject, String path)`**
  This method writes a `JSONObject` into a JSON file.
- **`public static void writeJsonArrayToFile(JSONArray jsonArray, String path)`**
  This method writes a `JSONArray` into a JSON file.
- **`public static JSONObject getJsonObjectFromFile(String path)`**
  This file reads a `JSONObject` from a JSON file.
- **`public static JSONArray getJsonArrayFromFile(String path)`**
  This file reads a `JSONArray` from a JSON file.

9.2 Class `SHA256Hash`

This class has a single method which returns the hash of an input string using the SHA256 hashing method. The method's signature is as follows:

- **`public static String getHashSHA256(String inputString)`**

9.3 Class `TextProcessingTool`

This class has three simple methods for text processing. They are specifically designed for the `runCommand()` methods in `Place` subclasses who are responsible to process the user input command.

- **`public static boolean stringFirstWordMatch(String mainString, String doesMatchString)`**
  This method checks whether an input command starts with a specific word or not.

- `public static boolean isInBrackets(String string)`
  This method checks whether or not a string is bracketed, i.e., in the form `[inputString]` where `inputString` could be any arbitrary string.
- `public static String unBracket(String string)`
  This method returns the unbracketed version of a bracketed string.

10 Package `Initialization`

This package is responsible for some initialization tasks which need to be done once before the first ever run of the game. They will not be needed unless database files are lost or a change in game structure has taken place.

10.1 Class `CardsDataCreator`

This class has one important method with the following signature which creates a JSON file containing the data of all cards in the game.

- `public static void createCardsData()`

10.2 Class `DefaultUserCardsDataCreator`

This class has a single method to create a text file containing a list of default cards a user has when his or her account has just been created. For this phase it contains ten randomly selected (but fixed) cards. The method signature is as below:

- `public static void createDefaultUserCardsData()`

10.3 Class `PlaceDataCreator`

This class is responsible for creating JSON files for each of the four places in the game, each of the files containing the set of valid commands and their instructions. As mentioned before, this data is used to respond to user's `hearthstone --help` command. There are five methods in this class, four of which responsible for creating the command instruction files for each of the four places, and a fifth method which runs all previous four:

- `public static void initSignInOrSignUp(String... path)`
- `public static void initMainMenu(String... path)`
- `public static void initCollections(String... path)`
- `public static void initStore(String... path)`
- `public static void initAllPlaces()`