D7032E - Home exam

Course supervisor: Josef Hallberg

Anton Tiberg - anttib-5@student.ltu.se

November 2, 2018

Introduction

Cards Against Humanity is a card game in which the goal is to match white cards that the player gets with black scenario cards. Then one player that does not play a card acts as a judge to pick the winning card. This game works in the same way as another game called Apples to Apples. This is a report that covers the project of going from a poorly written Apples 2 Apples program to refactoring it into a better written Cards Against Humanity program.

Contents

1 Unit testing

Rule/requirement 15 did not work as expected. The game was always finished as soon as a player got four points. In figure 1 we can see that the if-statement will be true as soon as a player hits four points and thus a player is going to win without taking the player count into consideration.

```
//Check if any player have enough green apples to win
if(players.get(i).greenApples.size() >= 4) {
    gameWinner = i;
    finished=true;
}
```

Figure 1: Image of code where it checks for the winner.

This is not testable since it is a small part in a large constructor, so there's no return value for JUnit testing to test against.

2 Quality attributes, requirements and testability

Both requirements are poorly written since both of them lack details which makes them open to interpretation for the developer. It asks for the program to be easy to modify, extend and test for future changes and implementations. But this is also required with any details or concrete examples. This could end up with the implementation to not be as expected, since developers can interpret the requirements differently.

3 Software Architecture and code review

3.1 Extensibility

It is quite hard to add functionality to the program with the existing code without re-factoring it, since most of it is in one constructor and there are very few methods, this leads to the code having a high level of coupling which creates a lot of unnecessary dependencies.

Dividing the code into more methods and classes will make the program more extensible and easier to futher implenet.

3.2 Modifiability

The code, as for now, is quite annoying and hard to modify. So changing rules or game mechanics will be a problem. Mainly because the lack of methods, which will make it so that the developer has to do more changes in the code when needed, instead of just one change and the methods will take care of the rest.

An example for why the program lacks in modifiability now is that a lot of limits in comparison statements are hard coded. It could easily be change into a final variable and used instead so the developer doesn't have to search for the correct row in the code for a small change.

3.3 Testability

This quality attribute could be done, if the developer does print statements as tests. But, if the developer wants automated tests with, for example JUnit, then it is poorly supported because of the lack of methods that actually return a value. So the testability of the code with JUnit is very poor. There's one method that actually returns a value, figure 2, so this could be testable. But it is far from enough to be able to say that the code has a high level of testability.

```
public PlayedApple judge() {
    if(isBot){
        return Apples2Apples.playedApple.get(0);
    } else if(online){
        int playedAppleIndex = 0;
        try {
            playedAppleIndex = Integer.parseInt(inFromClient.readLine());
        } catch(Exception e) {}
        return Apples2Apples.playedApple.get(playedAppleIndex);
    }  else {
        System.out.println("Choose which red apple wins\n");
        int choice = 0;
        try {
            BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
            String input=br.readLine();
            choice = Integer.parseInt(input);
        } catch (NumberFormatException e) {
            System.out.println("That is not a valid option");
            judge();
        } catch (Exception e) {}
        return Apples2Apples.playedApple.get(choice);
    }
}
```

Figure 2: Image of method that returns values.

4 Software Architecture design and re-factoring

4.1 Classes

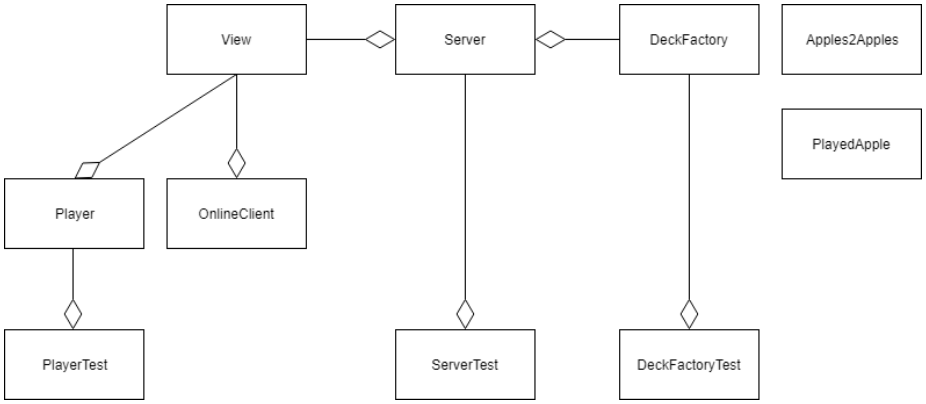

Figure 3: The main design for the program.

A Model View Controller approach was made when designing the structure of the code. Where Model is the DeckFactory, since it holds all the data. The View is the View class, which holds all of the prints. And lastly the controller is the Server, since it takes care of the main parts of game.
The server has a DeckFactory since it needs the cards for the game, it also has a View so it can call for the necessary prints for the game.

4.1.1 Apples2Apples

The main method and takes in some arguments which determines if it should start as a host, sometimes with bots, or a regular player that has to connect to a host.

4.1.2 PlayedApple

A helper class for the played cards.

4.1.3 DeckFactory

Takes in the cards as ArrayLists from txt-files. And it also shuffles the lists after they gets all the cards.

4.1.4 View

Contains, as mentioned before, all the prints that are needed to communicate with the user of the program. Hence why there are many classes that has a View object.

4.1.5 Online client

Takes care of the game for the users that are connecting through an ip.

4.1.6 Player

Contains all of the support for playing cards, receiving cards, it has all of the necessary variables such as ID and their hand. Bot functionality is also in the Player class, since the bot is a kind of player and I did not see any reason to split them up since the difference of the two is that a bot does things randomly.

4.2 Testing

Unit tests were also implemented (see Fig. 3), however all of the rules could not be tested. Mainly because they appear in the game loop, and there was no way to create Unit testing for the part where the program went into the game loop. The Unit testing that were implemented was for the setup (rule 1-5) and for winning the game (rule 14 and 15).
The thing that was kept in mind while implementing the code was to create methods that easily could be tested (methods that returns values). But I did not think about the game loop and that it kept me from implement tests for the methods that were used in the loop.

4.3 Modifiability and Extensability

Modifying the rules should be quite simple, for example, the required points to win is set as a variable that is used instead of just a hard coding of a number where needed in the code. There are more things like this in the code that makes it easier to modify the game.
The code is also divided into more classes and methods than before (see Fig. 4), which also makes it more modifiable since it is easier to change things in methods and still maintaining a working program. It is also easier to see where things go wrong in the code.
The program should also be more extensible now from before since there is a lot more methods that can be re-used if, for example, additional rules or maybe expansion cards is to be added to the game.

4.4 Quality Attributes

The reason for the Model View Controller design patter was to keep the whole program at a good structure and to make it clear which classes has to do with what part in the whole program. A Model View Controller structure also provides with a high level of cohesion and a low level of coupling and a higher level of modifiability.
The deck factory was used mainly so there was a single class that created the decks. But also so that it is completely separated from the other classes so no other class generates or holds the decks that are needed for the game.

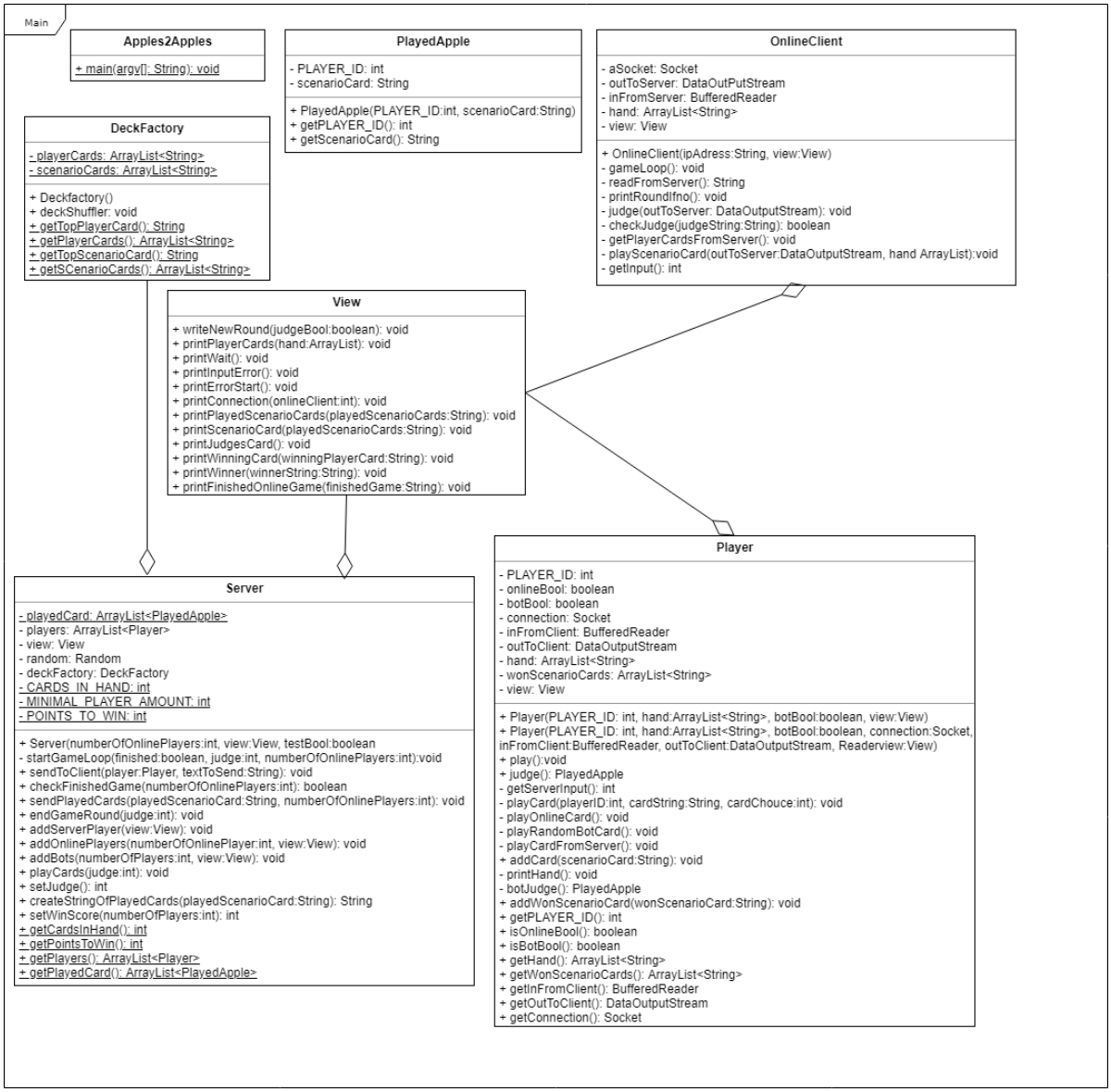

Figure 4: The UML diagram for the program.