Home Exam - Boomerang - D7032E

Andreas Söderman

October 19, 2023

1 Introduction

This was the home exam and the final assignment of the course Software engineering (D7032E) at Luleå University of Technology. In this home exam a functional but badly designed and implemented version of the card game Boomerang Australia was provided. The goal for the exam was to create a new version of Boomerang that focused on providing modifiability, extensibility and testability along with upholding the SOLID principles and Booch's metrics.

2 Unit Testing

All of the requirements 1-12 are fulfilled by the new implementation of Boomerang. It is also possible to test that these requirement have been fulfilled with tests written using JUnit. Since future modifications 13-14 were not implemented no unit tests where written for those.

3 Software Architecture Design and Refactoring

3.1 Modifiability

To provide modifiability the implementation was divided into different packages that have different functions within the program. This included creating separate packages for: the input and output, the player and all functions related to it (the players scoresheet and hand), the creation of card objects, and the creation of messages. These are not the only parts of the original design that have been split into separate modules but they each help show how cohesion has been increased in the new design.

One example of how modifiability was provided is the abstract factory method used to create the card objects in the game. This is to make it easier to add other card-sets such as those described in future modification 14 and use card sets that may exceed or fail to reach the normal amount of cards used in Boomerang. If such a card-set was to be added a new implementation of the standard rules would also need to be created to change the amount of expected cards and number of cards per player.

An UML model of the abstract card factory for cards used in Boomerang Australia can be seen in Figure 1. As can be seen in this figure, cards are implemented as an abstract class that has one implemented function called *getCardInfo*. This function reads takes the path to a file that has the information about a specific card saved.

The information associated with the different cards where saved in *JSON* files, this can be seen in Figure 2.

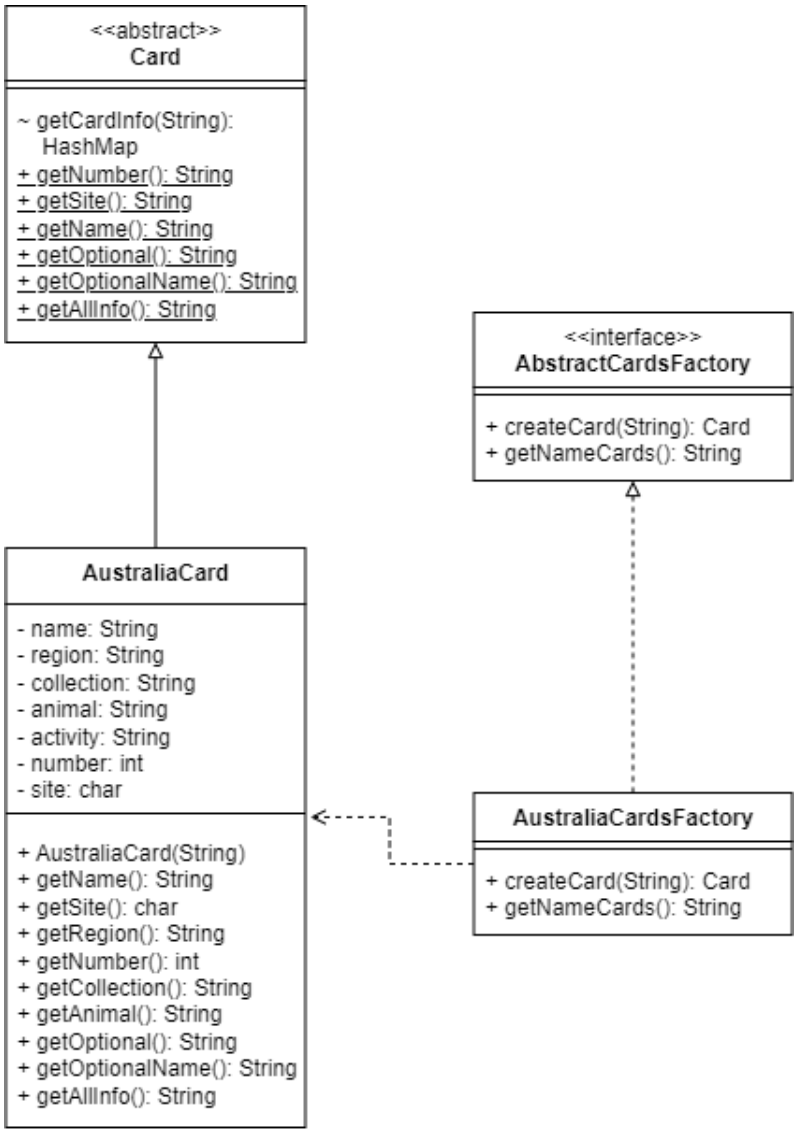

Figure 1: An UML class diagram of the Abstract Card Factory created for the assignment.

Figure 2: Shows the JSON files used to store the information associated with the cards.

3.2 Testability

In order to provide high testability in the new implementation there was a focus on limiting complexity by dividing the different functionality into separate modules and classes that made use of specialised interfaces and abstract data structures. One example of this is the abstract class *Scoresheet* which was designed to get extended into different scoresheets depending on the version of the game being played. The only version currently implemented is the *ScoresheetAustralia*.

In this Scoresheet all the unique scouring mechanics of Boomerang Australia described in requirement 10 have been implemented. This makes the new design simpler than the original design by not having the score be saved in the player class and having the score for the round be determined in one long function. It also makes it easier to test the functionality of individual scouring functions. A simplified UML showing the scoresheet interface can be seen in Figure 3.

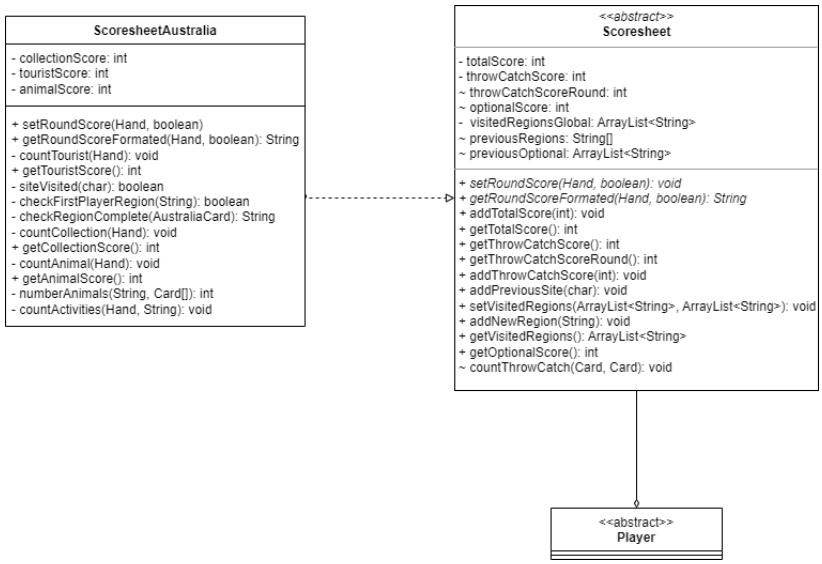

Figure 3: A simplified state diagram showing the Scoresheet interface.

The process of playing the game was also divided into different states to make it easier to test the individual parts of the game. The main game loop can be seen in Figure 4, and a simplified UML diagram of the of different States of the game can be seen in Figure 5.

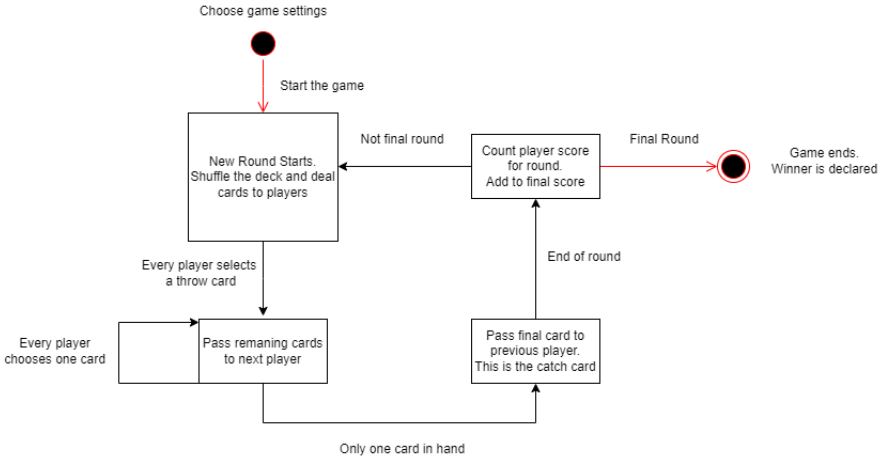

Figure 4: A simplified state diagram showing the game loop.

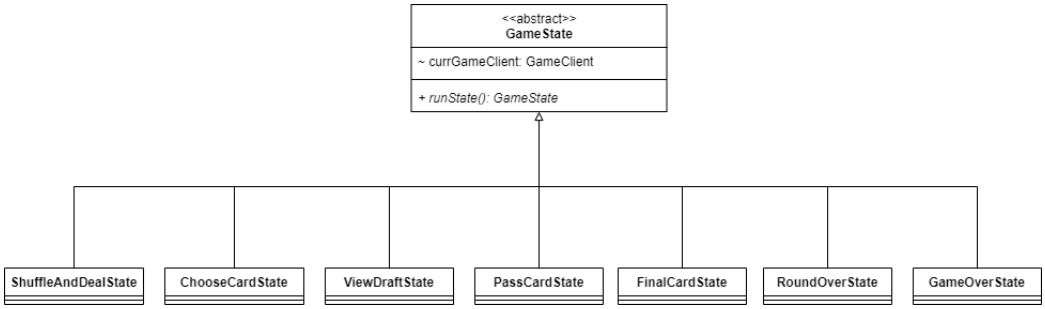

Figure 5: A simplified UML class diagram of the different game states.

A Builder for the GameClient was created to make it easier to test specific configurations. The Builder was made in such a way that the only parameters that need to be set are the number of real players and bot players, and all the other parameters for the GameClient such as the card factory, rule set, scoresheet, and the behavior of bot players were all set to the default implementation.

A UML model of the game client builder can be seen in Figure 6. As can be seen in this figure the builder contains functions to not only change the number of players in a game but also the specifics and behavior of the game being played. This was done to make it easy to extend different parts of the game. At the moment each of these functions contains a switch statement that only contains one case, and that is for the value of '1'. This is because no functionality beyond that described in the assignment has been implemented. If the Boomerang Europe and Boomerang USA modes where to be implemented their respective abstract card factory and scoresheet could simply be added to these switch statements, for example *setEdition(2)* could set it to Boomerang Europe and *setEdition(3)* could set it to Boomerang USA.

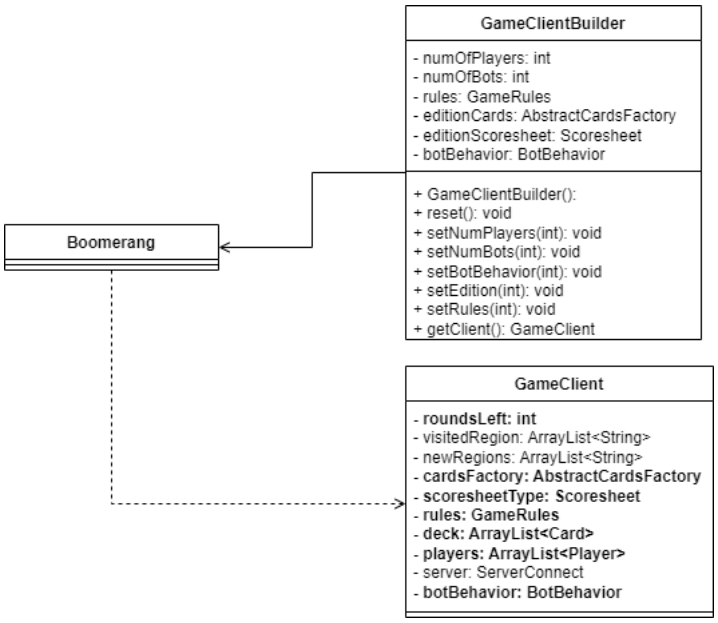

Figure 6: A UML class diagram showing the game client builder.

3.3 Extensibility

Extensibility is a quality attribute that build on top of the attributes modifiability, integrability and testability. This meant that a lot of the work done to provide modifiability and testability also helped with providing extensibility.

Extensibility is provided in the packages that make use of interfaces. This can be seen in the following interfaces:

- The *GameRules* interface which is intended to be implemented for different rulesets such as the one described in future modifications 13.
- The *HandleIO* interface which was made so that the game easily be extended into using some sort of UI instead of just using a console. It is currently being implemented by two classes, the *Console* class and *Remote* class. The *Console* handles local IO presented in a console, and the *Remote* class handles communication over a remote connection.
- The *BotBehavior* interface makes it possible to extend the functionality of bot players in the future. A new implementation of this interface was created where the bots choices were completely predictable in order to test rules 5-9.

UML class diagrams for the described interfaces can be seen in Figure 7.

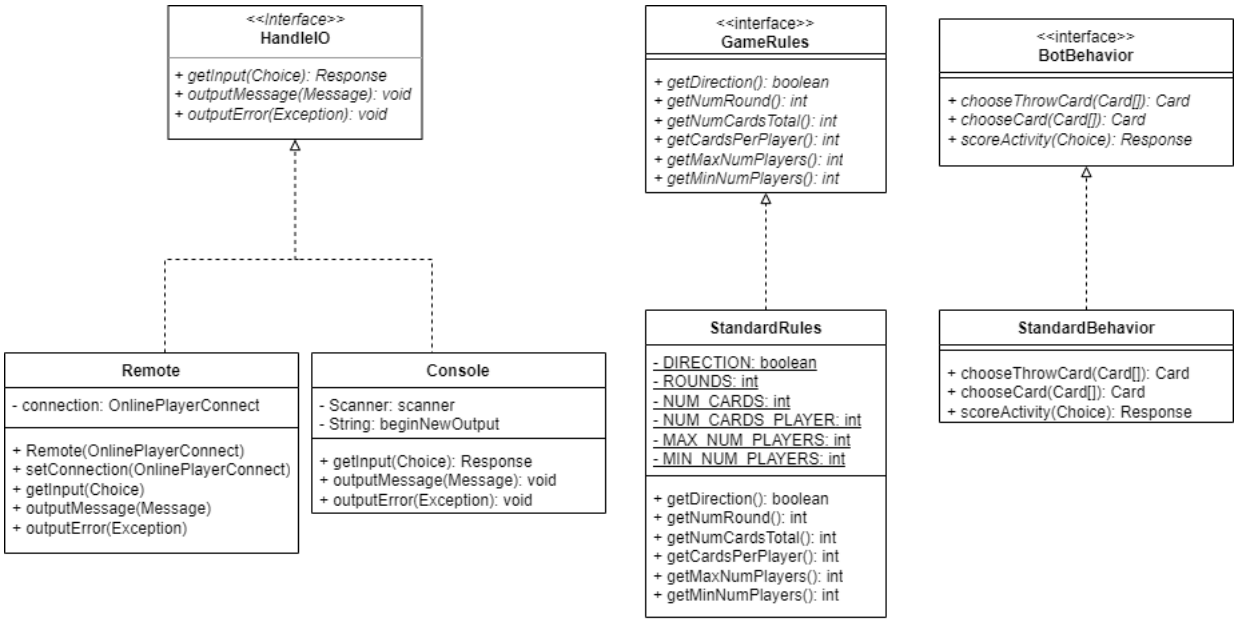

Figure 7: Shows UML class diagrams for some of the interfaces used in the new implementation.

4 Running the Game

The game can be run through the executable JAR file in the file folder. The following commands are used: `java -jar "D7032E - Boomerang.jar"` [#Players] [#Bots] to create a game, and `java -jar "D7032E - Boomerang.jar"` [IP-address] to join a game. The syntax to launch a game with specific settings is `java -jar "D7032E - Boomerang.jar"` [#Players] [#Bots] [Edition] [Rules] [Bot Behavior], although currently only the standard version of Boomerang Australia described in the instructions for this assignment have been implemented so these parameters can not change anything.