Kai Zinnhardt 19-763-176/ Jia Fu 21-907-662/ Paul Luley 21-741-491/ Joel Meier 16-916-959

Software Construction HS 2021 Group 03: Assignment 4

Responsibility Driven Design

CRC Cards / CRC Session:

For the Responsibility Driven Design approach, we started off by searching for the noun phrases. We found the following nouns: Game, Player, User, Dealer, Terminal, Rounds, Cards, and Deck.
After recording those noun phrases we formulated our candidate classes. We did this by conducting a class rationale selection. Thereby we found conceptual entities (Game, Player, Dealer). Additionally, we found several words for one concept (Player or User). Furthermore, we also found values of attributes (rounds). After doing this step we got the following list of candidate classes: Game, Player, Dealer, Terminal, Rounds, Cards, and Deck.
Before starting the CRC Session we looked at the verb phrases that are inside the description. We got the following list of verbs: show the amount of money, go away with money, ask for the amount, input a bet, visibility, decision.
We started the CRC Session with the role play for the different scenarios. The main scenarios that we covered were: initialization of a round, the interaction between the player and Dealer, how the round should end, and how a new round should be declared. As a consequence, we wanted to cover all the different scenarios of the game. After the CRC session, we noticed that we need to add the classes of the launcher, IO (split into input and scoreboard. This can be extracted from the card “Terminal”.), and Hand. Additionally, we could discard the clases of: Round and Terminal (now split into “input” and “scoreBoard”).

However, during the implementation of the Game, we noticed that we needed to add a further class, naimly the “Round” class. This has the responsibility to keep track of the rounds won, lossed and tied during the game. (It Collaborates with the game and the Scoreboard).This, therefore, leaves us with the following CRC Cards:

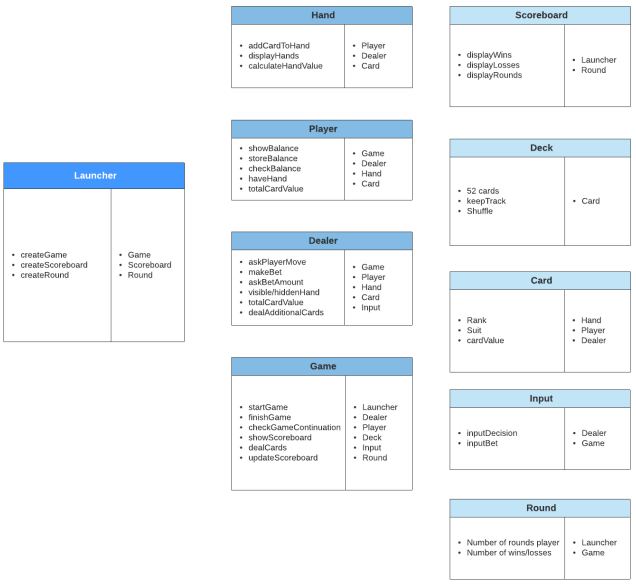

Figure 1: CRC Cards

To give a static overview of how the Blackjack game will work, we have created the following **Class Diagram**:

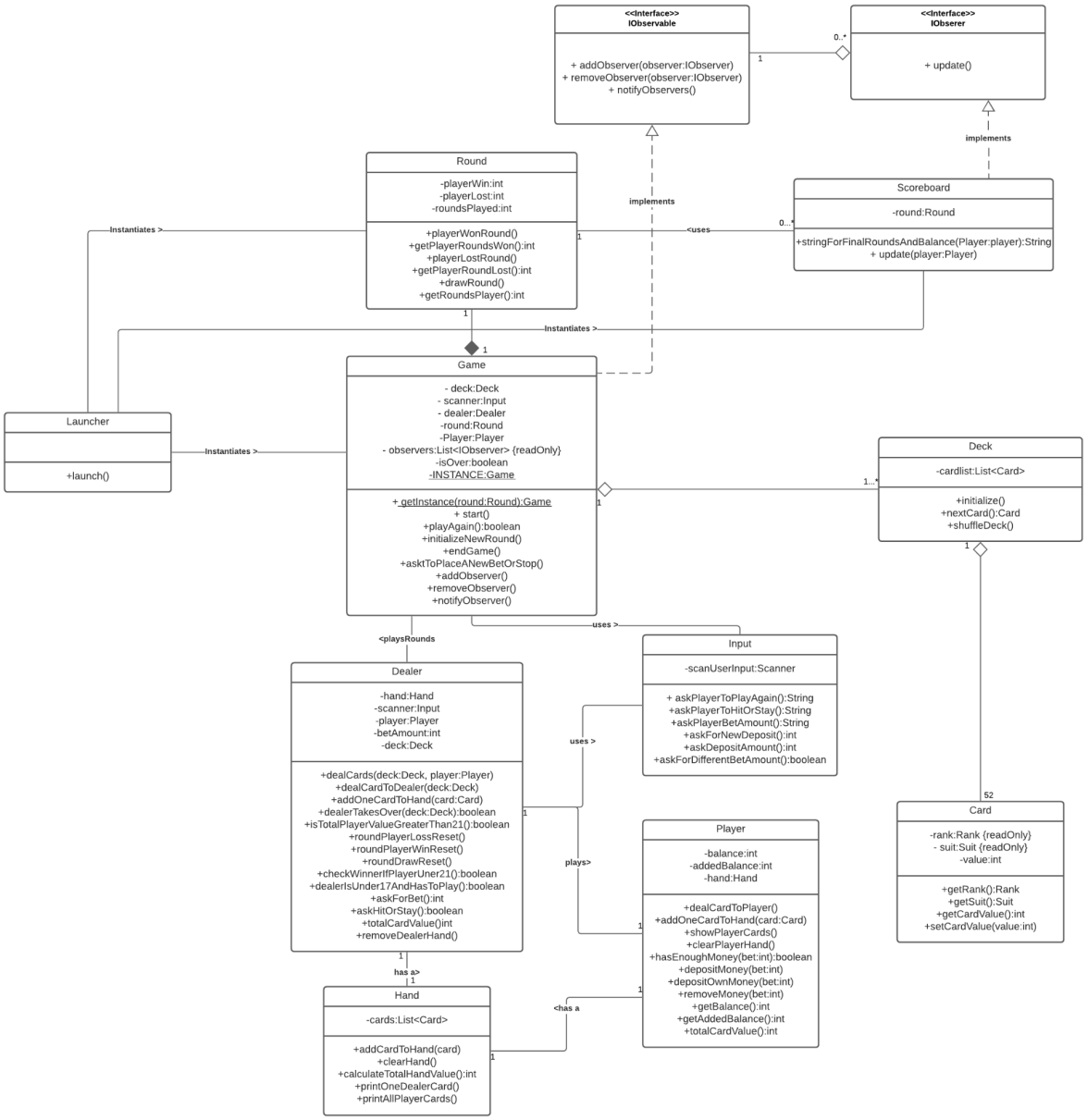

Figure 2: UML Class Diagram

Sequence Diagram:

However to get a full impression of the functioning of the blackjack game, one also needs to know how the game works dynamically. Therefore the following Sequence Diagram will show the dynamic behavior of the game.
For the sake of clarity it makes the following assumptions:

1. It is an initial game, the user starts with 100 CHF
2. When asked whether the user wants to play or not, the user answers ‘Y’ (according to the instructions for “Yes”)
3. When asked for a bet amount the user enters an amount between 1 and 100

4. When asked whether the user wants to hit or stay, the user answers ‘S’ (according to the instructions for “Stay”)
5. A lot of details are displayed but the Game to Scoreboard communication is left out since it is explicitly explained in the next section.

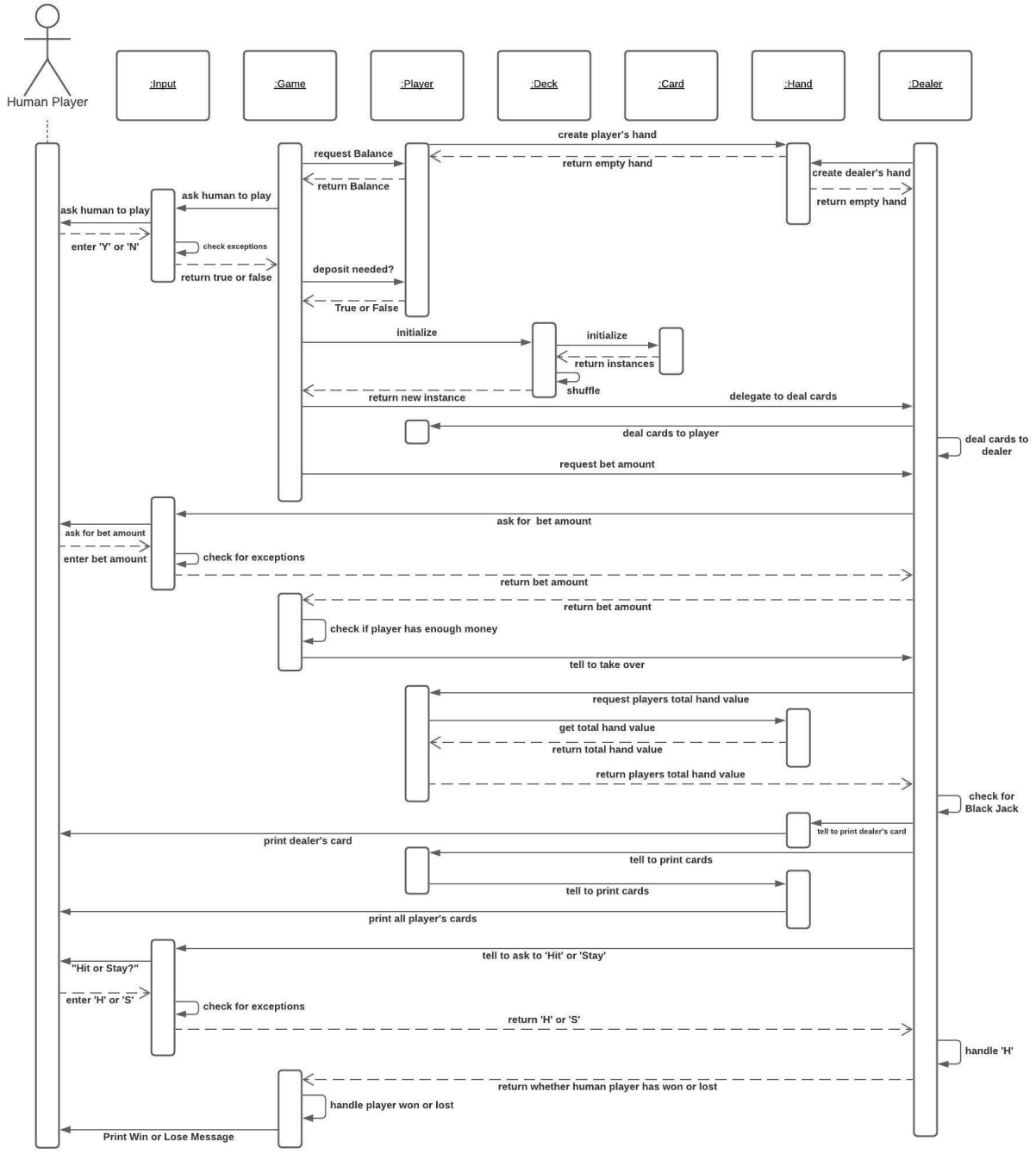

Figure 4: Sequence Diagram of an initial game

The sequence diagram can be read as follows:

1. During initialization an empty Hand is created for the human Player and the Dealer each
2. The Game logic checks the Player’s balance
3. The Game asks the user whether he or she wants to play or not via the Input
    a. The Input checks for exceptions and would throw them accordingly

4. The Game checks with the Player's balance whether a deposit is needed or not
5. The Game initializes the Deck
    a. The Deck in turn initializes the Cards and shuffles them
6. The Game tells the Dealer to deal the Cards
7. The Dealer deals the Cards to the Player and itself
8. The Game requests the Dealer to ask for a bet amount
9. The Dealer utilizes the Input to ask the user for a bet amount and returns it to the Game
    a. The Input checks for exceptions and would throw them accordingly
10. The Game checks whether the user has enough money or not
11. The Dealer takes over and asks the Player for its total Hand value
12. The Player asks the Hand for computation of the value and returns it to the Dealer
13. The Dealer checks for an immediate black jack
14. The Dealer uses the Hand to print its card
15. The Dealer tells the Player to print its cards
    a. The Printer tells the Hand to print its cards
16. The Dealer uses the Input to ask for "Hit or Stay"
    a. The Input checks for exceptions, would throw them accordingly, and returns the answer to the Dealer
17. The Dealer would handle the answer 'H' (according to the instructions for "Hit", but in our case the user answered 'S')
18. The Dealer returns to the Game whether the Player has won or lost
19. The Game handles a Win, Loss or Draw accordingly, prints an according message to the console, adjusts the player's balance, clears the hands, counts the rounds played, the wins or losses

Design Patterns

Observer Pattern:

As it can be seen by the Class diagram above, we used an Observer Design Pattern for the Game to Scoreboard association. The Observer Design Pattern was used to create loose coupling between these two classes. This is as we saw the need that if we want to change or add a different type of Scoreboard, this kind of loose coupling is needed. This was implemented in such a way, that the Scoreboard is the Observer and the Game is the Observable. The game contains a list of observer objects. If needed an observer then can be added or deleted from this list. Additionally, the Game can then update all the observers that are registered inside the list at any given moment in time.

In the concrete implementation phase, we had trouble generating the loose coupling, as the game has information that the scoreboard needs to know. This is as the game knows when a new round is started/ended and who won the round. To not destroy the loose coupling and to not give the game too many responsibilities we created another class (“Round”). The “Round” then collects this information. Then the specific Scoreboard, if it needs to, can gather this information from the round class. The part of the class diagram, that covers this instant is the following (thereby one has to exclude the Deck class):

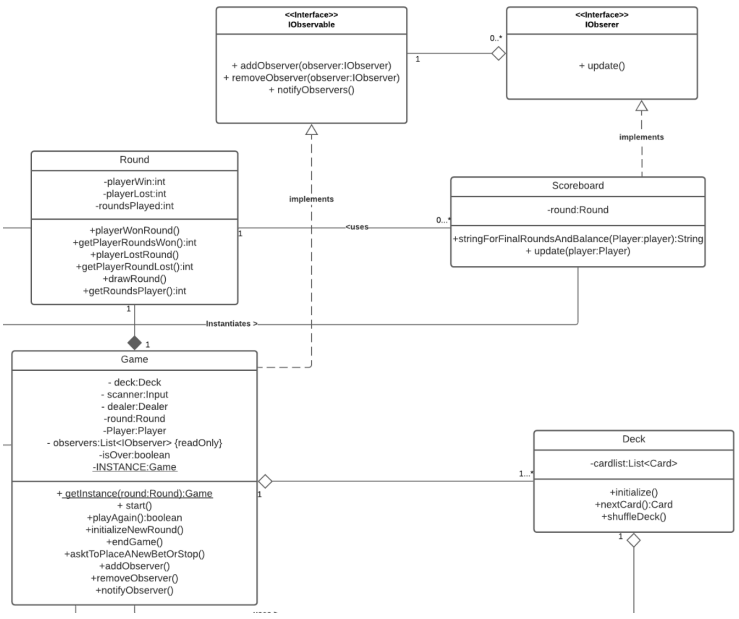

Figure 5: UML Class Diagram for the Observer Pattern

To show the dynamic behavior of the observer pattern a sequence diagram of the user interacting with the game is shown in *figure 6*. The following sequence diagram assumes the Game was already played, and the Player wants to end the Blackjack game. Thus, it focuses on the Game to Scoreboard communication.

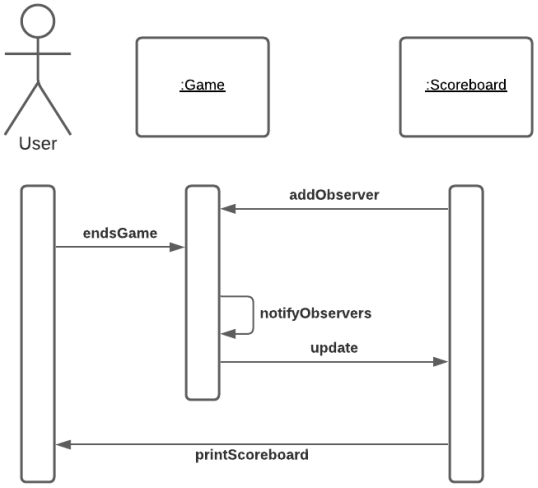

Figure 6: UML Sequence Diagram for the Observer Pattern

Singleton Design Pattern:

The Game is heavily used: it interacts with the Launcher, Input, Deck, Round, Player, and especially with the Dealer. To ensure at all times that the same instance is used by the different parts of the code that are interested in the Game, the Singleton Design Pattern was applied to it. It guarantees that there is only a single instance of the game running at a time.
To achieve that the Game's constructor was made private. That way it is only accessible inside the Game's class. To get the one and only instance of the Game from the outside a public static method getInstance(Round round) was introduced, which checks whether an instance of the Game already exists, if not it will create one, and returns the instance afterwards. If the instance already exists, it will return this specific instance. If another developer decides to work on this Blackjack game, it will be seen easily that the Game is meant to be a Singleton and that it is not possible to create a second instance.

Class Diagram:
To show the usage of the Singleton Design Pattern in a static world, a class diagram will be used. Thereby this class diagram will only show the Game class, that contains the Singleton instance. The Singleton class Diagram is the following:

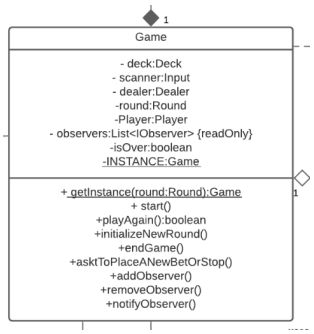

Figure 7: UML Class Diagram for the Singleton Pattern

Sequence Diagram:

To look at the Singleton implementation from another perspective, a sequence diagram will be used, to show its dynamic Behavior.

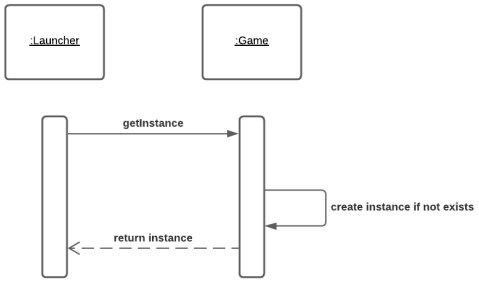

Figure 8: UML Sequence Diagram for the Singleton Pattern

Kai Zinnhardt 19-763-176/ Jia Fu 21-907-662/ Paul Luley 21-741-491/ Joel Meier 16-916-959

Implementation of the Blackjack game

General remarks:

We started the implementation by following our responsibility driven design session and applying the class design. The first steps were to make sure all rules were followed accordingly in our Blackjack game. That meant for example, the correct behavior of the Ace cards. We did that by checking that at most one Ace can be counted as value “1” and in general the values are counted correctly as “1” or “11”. If the player has an Ace it can be counted as “11” so that the total Player card value is still “<=21”, otherwise it will count as “1”. If there are several Aces in play, only one will count as “1” and the other one automatically counts as “11”. The card value calculation is done by the program and shown to the player - to make sure no calculation errors occur.
In order to ease visual comprehensibility we decided to show the player all of his cards on the terminal after each draw of any card. That way the person playing the game always knows what his hand currently holds, what the value is and can make decisions based off of that.
We tried in general to guide the player in a way that is easy to understand by applying the following rules:

- If the bet amount is higher than the available balance, “0” or negative, a warning is raised.
- The player can see his hand (with all cards currently in it), his total value and is asked to hit or stay
- After each round is finished, the win or loss is added to the stats and a new round starts asking the player if he wants to continue
- etc.

No matter what the player does, they receive “visual” feedback on the terminal with what to do in the next step with very clear instructions. This allows for fluid gameplay.
Last but not least once a player is done playing the game the scoreboard will show them the amount of rounds they played, the wins and losses and also the total amount of money that was won or lost during playtime.

Our extension:

The Player can recharge their deposit.

- The player is allowed to deposit more money if he wants to play with more. This is as the player can bet with more money than he has deposited. However, if this case occurs, he is asked to deposit more money or “revisit” his bet, such that the deposit covers his bet.
- Furthermore, if the player lost all of the deposits he had, he can continue to play. Thereby, after getting asked if he wants to continue, the player has to deposit more money.
- The deposit is limited to 1000 CHF at a time.

Testing Report:

For testing we wanted to cover the main parts of the game, thereby we focussed on whether or not the game is stable to play. Thereby, we covered some use cases, to see if the code is behaving like it should be doing. The test cases cover all of the classes except for the Launcher class. This class was perceived as less important for testing. During our testing, we achieved a total line coverage of 79.84% (including the Launcher class) and 90.09% excluding the Launcher class. In Table 1 the distribution of the line coverage can be seen. Thereby one can notice that 5 out of 9 classes under test have a line coverage of 100%. Additionally, we also created a histogram (figure 9) to visualize this table.

Classes	Line Coverage	Line Covered	Total Lines
Game	40,91%	27	66
Round	100,00%	12	12
Card	100,00%	8	8
Deck	100,00%	11	11
Hand	100,00%	25	25
Input	92,31%	36	39
Scoreboard	88,89%	8	9
Dealer	88,71%	55	62
Player	100,00%	16	16
Total	79,84%	198	248

Table 1: Displays the numbers of the line coverage, including the total line coverage.

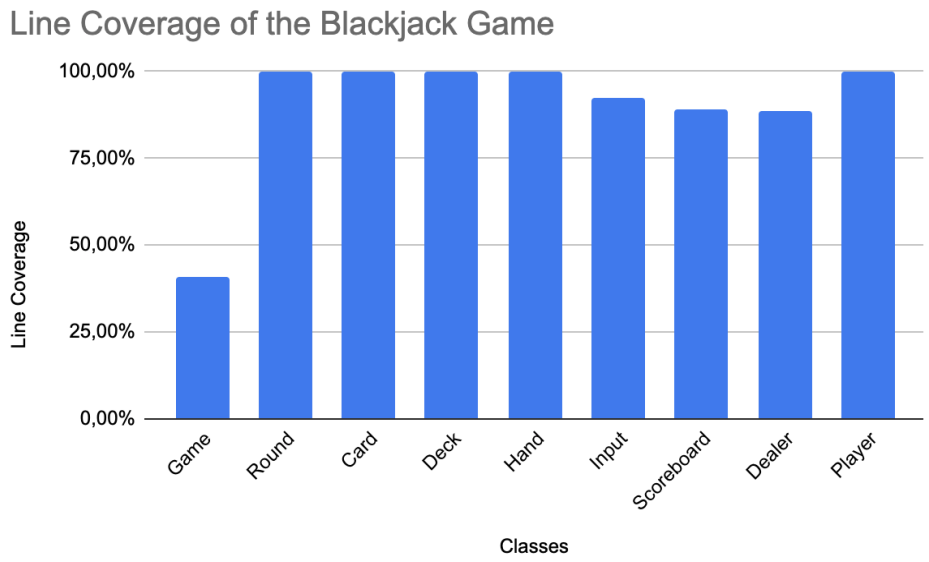

Figure 9: Histogram showing the line coverage of the classes we tested