Exercise 1 - A Blackjack Game - Design and Implementation

The goal is to design and implement a fully playable Blackjack game. The objective is to use the learning from the Software Construction course to implement Blackjack!

The first thing we did is, put the laptops aside. We sat together and figured out what classes we wanted. We decided the roles and responsibilities for each class. We also decided on how classes will collaborate with each other. On our first sitting, we decided to have 8 classes which are:

Game, Casino, Player, Dealer, Bank, Card, Deck, Utils

We further discussed and realised that Dealer and Player classes can be merged since they are both Players who will be playing the game. So our classes became:

Game, Casino, Player, Bank, Card, Deck, Utils

Once we had the classes, we created crc cards for each class and discussed the roles and responsibilities for each class.

Please find the **CRC cards** as follows:

Game	
• Entry point of Game • Setup the game • Decide if player wants to play or leave	• Utils • Casino • Player

Card	Comparable
• Handles individual cards • provide rank and suits of card • print card • provide value of card	

Deck	CardSource
• Provide a deck of 6x52 cards • Shuffle cards • Draw cards	• Card

Player	
• Holds information about Player and Dealer • Provide balance • Add balance • Remove balance	

Casino	
• Blackjack implementation	• Deck • Card • Utils • Player

Utils	
• Scan different inputs from Terminal • Validate inputs • Other helper methods	

Bank	
• Information about Money • Note: This class was decided during responsibility driven design but later merged with player class	

Once we had a good idea about how our Blackjack game implementation will look like, we started coding. During implementation, we figured out that we can merge applications from the Bank class in Player class. Therefore, our final classes are as follows:

Game, Casino, Player, Card, Deck, Utils

Out of the abovementioned 6 classes, important classes are:

Game, Casino, Player, Card, Deck

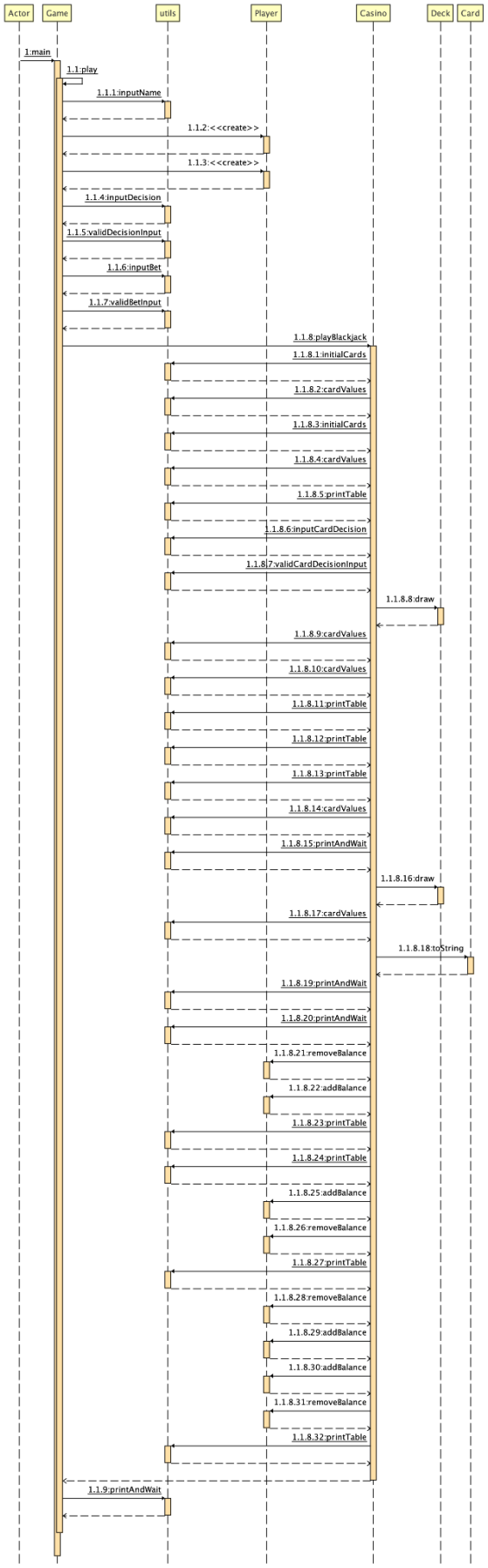
Utils class has some utility methods which are like helping methods. Methods from utils class are used throughout the code.

Please find below detailed description of each classes:

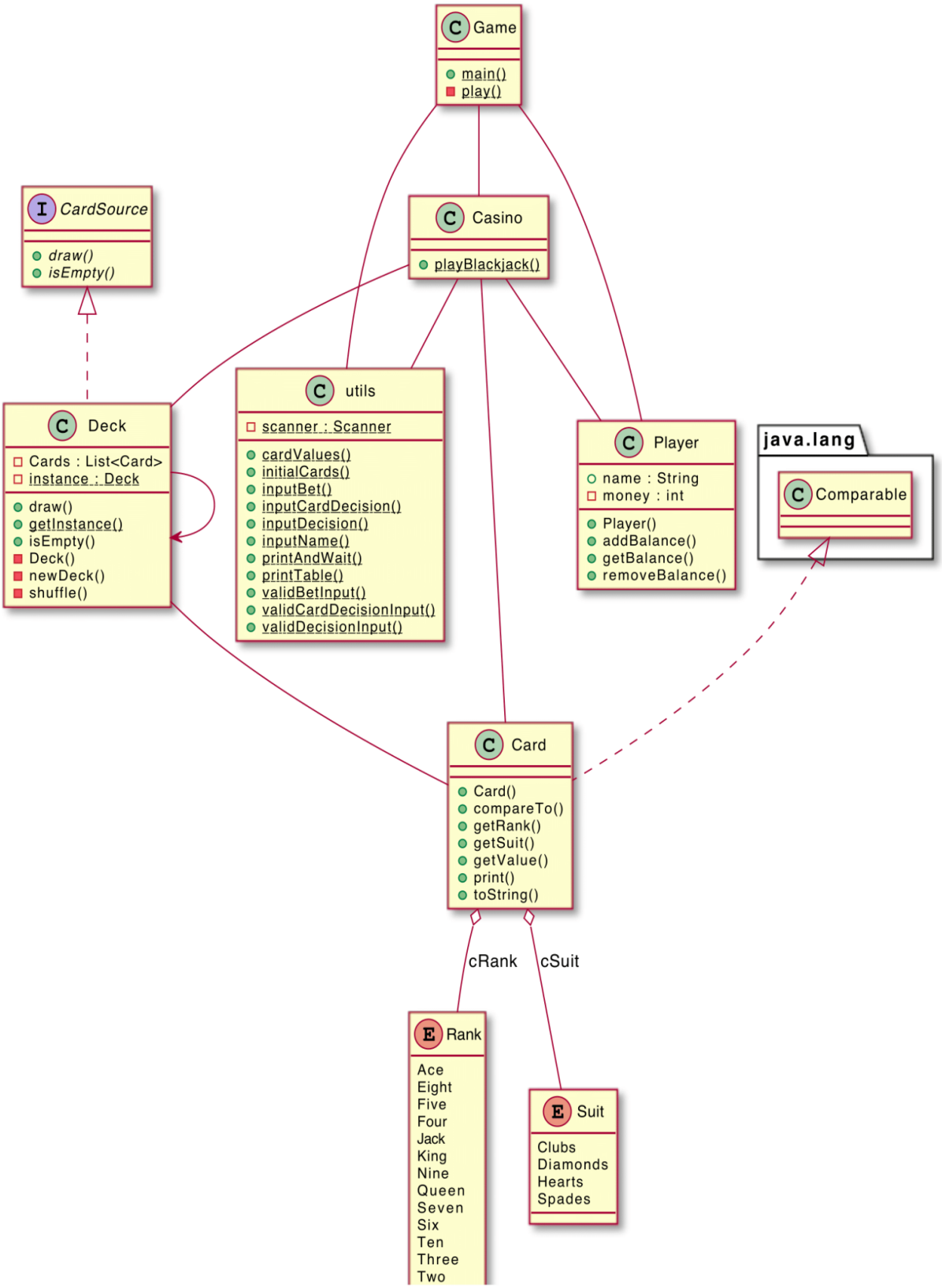
Class name	**Detailed description**
Game	This is the entry point of the game. This class is responsible for running the entire game. It takes input from the user such as player name, if they wanna play or go home with remaining money, and if they wanna play, how much they wanna bet, and so on. Once it receives all the information and validates if the user provided correct information, it further asks the Casino to play the black jack game. It is also responsible for game over and publishing the results if the player loses, ties, or wins the round.
Casino	This class is responsible to play a round of black jack game. There is only one method in this game called playBlackjack which was called from Game class. This method plays a round of a game and returns the result. Result is either between Win, Lose, and Tie.
Player	Player class is a class of Player. It has 2 class variables namely name and money. It can add or remove money from a Player's account. Dealer and User are players in our implementation.
Card	This class has 2 private variables cRank and cSuit of Rank and Suit enum type respectively. This class implements Comparable and can therefore be compared.

Deck	This class is Single object class (**Singleton pattern** implemented). It implements the CardSource interface. It is responsible for creating a new Deck of 6 x 52 cards. We kept 6 x 52 cards instead of just 52 cards because Casinos use this trick to avoid players counting the Rank on cards. Deck is also responsible for shuffling and drawing cards from the deck.

Please find the **sequence diagram** of our game as follows (For better visibility we added SequenceDiagram.png file in assign_4 folder):


Actor
Game
utils
Player
Casino
Deck
Card
1:main
1.1:play
1.1.1:inputName
1.1.2:<<create>>
1.1.3:<<create>>
1.1.4:inputDecision
1.1.5:validDecisionInput
1.1.6:inputBet
1.1.7:validBetInput
1.1.8:playBlackjack
1.1.8.1:initialCards
1.1.8.2:cardValues
1.1.8.3:initialCards
1.1.8.4:cardValues
1.1.8.5:printTable
1.1.8.6:inputCardDecision
1.1.8.7:validCardDecisionInput
1.1.8.8:draw
1.1.8.9:cardValues
1.1.8.10:cardValues
1.1.8.11:printTable
1.1.8.12:printTable
1.1.8.13:printTable
1.1.8.14:cardValues
1.1.8.15:printAndWait
1.1.8.16:draw
1.1.8.17:cardValues
1.1.8.18:toString
1.1.8.19:printAndWait
1.1.8.20:printAndWait
1.1.8.21:removeBalance
1.1.8.22:addBalance
1.1.8.23:printTable
1.1.8.24:printTable
1.1.8.25:addBalance
1.1.8.26:removeBalance
1.1.8.27:printTable
1.1.8.28:removeBalance
1.1.8.29:addBalance
1.1.8.30:addBalance
1.1.8.31:removeBalance
1.1.8.32:printTable
1.1.9:printAndWait

Please find the **class diagram** below:



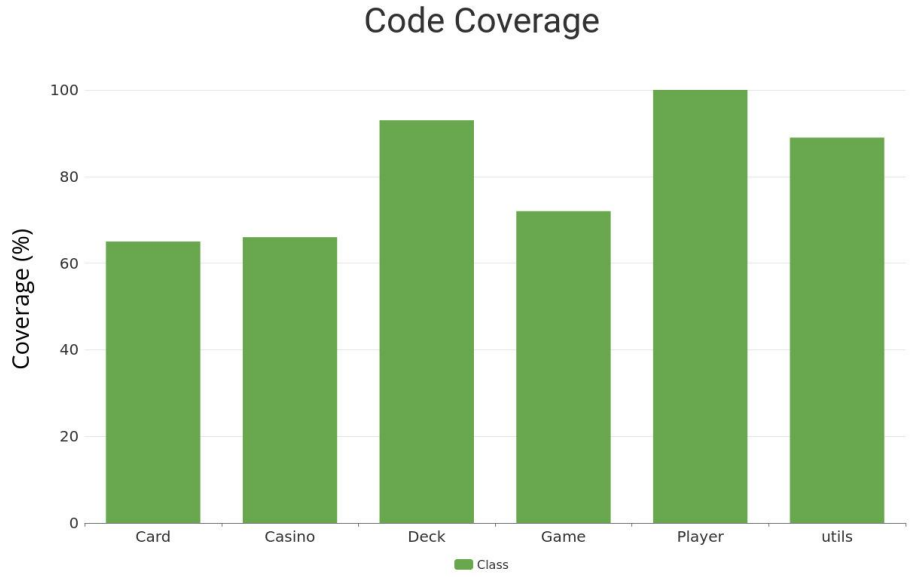
Unit testing:

Note: We used JUnit 4

We wrote unit tests for all the 6 classes and saved them in tests/java folder. We have line coverage of 74% to 78% depending upon who won the game in which stage**.** Writing tests for some classes were more complicated than other classes. For example, tests for classes such as Utils, Player, and Card were quite straightforward. We did not test the Casino class because it only has one method and that method is tested along with the Game class. On the other hand classes such as Game and Deck were harder to test. We will look into the harder classes in details:

GameTest:
This class is very challenging because it has just two methods 1. main 2. play. Both the methods are void and the main method calls play method. Since play is a void method so it will not return anything on which we can assert. In addition, this class takes input from the terminal and runs as many times as the user wants. This makes it extremely hard to test. On the other hand, testing this class is extremely important to achieve desired coverage. Therefore, we used a trick. We gave a string of input where we provided lines from the terminal and then onces the lines provided are exhausted, the game sends an exception of No line found and we just check that. We acknowledge that this isn't the best way!

DeckTest:
The method which needs to be tested in Deck class is draw. Other methods can be included in the test of draw since they are void methods but they are called from the constructor. Therefore when we get the Deck instance, we already are calling other methods. The challenge here is when we draw a card, the rank and suit of the card is randomly drawn from a shuffled deck. Therefore it is hard to assert it with an expected card. But the least we can do is check if the output is a card instance or not. That's exactly what we did. If so, the test is passed.



Conclusion and Acknowledgement:

There are lots of takeaways from Software Construction assignments. We learned about planning, execution, testing, and improving softwares. We learned about OOP, encapsulation, Design patterns, anti-patterns, Sequence diagrams, Class diagrams, Unit testing, to name a few. In addition to betterment of our Software skills, we also improved some soft skills while working together in a team of 4. We learned a lot from each other. We would like to thank Prof. Dr. Alberto Bacchelli for teaching us this beautiful course. Special thanks to Enrico Fregnan and Deborah Jakobi for supporting and helping us throughout the assignments.