Assignment 4

Ex. 1:

The Blackjack Game
For our Blackjack game we added three extensions. The first extension is that the game is playable with up to five players. The amount must be set in the beginning of the game and each player has to introduce himself. The second extension is that a player has the ability to split its hands. This follows the rules of the official Blackjack game, meaning that he can only split when he initially receives two cards with the same value. He can split up to 3 times, having a total of 4 hands. The third extension is a scoreboard where the best 10 players are represented in a Hall of Fame list.
It was taken care of all the variables in the code, especially regarding getters and setters. All variables are private, and setters don't exist. There are getters, but none that would let anyone publicly access a private field. If there are getters that would do so, they have the modifier protected such that they can only be accessed by classes in the same package. This is especially useful e.g., in the case where the player class needs to access the hand list of the 'PlayerSubject' class.

Singleton pattern
The singleton pattern comes into play with the CardDeck class as it is the central card staple that exists exclusively throughout the whole game. As usual, the Singleton ensures the CardDeck class can only be instantiated a single time and when accessed further, will subsequently return its own instance. For this to work, the constructor is accessible only locally through the *getInstance()* method and thus private. Said method is thread-safe and will simulate a static behaviour. The instance reference is again stored in the CardDeck class itself for subsequent retrieval. The pattern attenuates the notion of atomicity for the card deck.
Other classes access the CardDeck using the *getInstance()* method.

Command pattern
The command pattern has been used for the scoreboard. In particular, there are commands for printing and saving the scores. There is no command for loading the score since the method responsible for loading the score is private and is called within the method responsible for printing the score.

Main class and Logic package
The Main class represents the game flow. It contains only a main method with the game loop in it. We decided to design it like this because it is a natural mapping to a real Blackjack game. At the beginning we initialize some variables and greet the players. Then we enter the game loop which first asks for the bets, then let the players play the game and at the end evaluates and concludes the round. This gets repeated until no player is left. New players can of course join at the end of every round if there are seats available.
The Game class on the other hand contains most of the logic. Here again we tried to follow the natural flow of a Blackjack game. There are some main methods like takeBets() or play() and are called directly in the game loop. Other methods like distributeCards() or takeNewPlayers() are just helper methods to avoid duplicate code and are used only in the Game class.
The IOFormatter class is just a helper class. It is used only in the Game class and only when the table must be printed. Error messages are printed directly without the need of the IOFormatter. We also decided not to user the IOFormatter class to format the input since in every state of the game another type of input is required, such as a number between 1 and 3, a string as name or the string "leave" to leave the table. There would be no benefits to implement this in the IOFormatter class and therefore each method in the game class validates the input on its own.

Cards package
At the center of the Blackjack game are the card sets which consist of cards and those are eventually combined on a deck where they are shuffled together. Cards have several properties such as their rank, suit and whether they are hidden. The hand takes care of the cards, can add and reveal them or count their points. A card set is a complete single collection of all ranks and suits that exist in a classical Blackjack card set, amounting to 52 cards. The CardDeck is of great significance as it is the interface between the logical representation of the abstraction classes and the game logic. It is also present on the game table and all the players will draw their cards from it. Upon creation, a new deck is allocated with a given amount of card sets and then shuffled. It further ensures that it does not run out of cards and restocks itself when falling below a certain threshold.

Scoreboard package
One of the extensions to the base game that we have added includes the scoreboard. The scoreboard is always shown once at the start of the game and at the end, when there are no players left, via the printScore command. If no games have been played so far, then the scoreboard is empty. Whenever a player decides to leave or is kicked due to running out of money, then their score, that is, how much money they have managed to accumulate, is saved.
When saving the score, two things happen. The variable as well as the file holding information about the scoreboard are updated. Thus, this allows to easily print the up-to-date scoreboard, while also not losing any information about the scoreboard when the game is terminated. Saving the information simultaneously on a file allows to load the contents of the file at the start of the program execution so that the scores of players across multiple sessions can be saved.
The ScoreBoardEntry is an entry of the ScoreBoard and as opposed to the Player class, the constructor allows to set the name as well as the balance, which is required for loading the contents of the file and saving each entry as ScoreBoardEntry corresponding to a Player.

Player package
The player package contains three classes, an abstract class and two child classes. The abstract class 'PlayerSubject' provides methods that are used by player objects and a dealer object. Since player and dealer do not share every method and implement certain methods on their own, an abstract method was the better choice than an interface.
Players are a bit more complex and need some more methods than the dealer. They need to track their balance and are able to split their current hand. However, both objects can receive cards, track hands – even though a dealer only has one, the implementation is shared – and clear them. Some methods will throw custom exceptions.

Exceptions
Cards, hands, and players have their own exceptions. Each exception has its own class and has a constructor with a preset default message. Those exceptions are thrown when a method gets called in a situation where it should not be called.

Tests
Most of the classes are tested. We got an overall line coverage of at least 90%. Therefore, each tested class has a line coverage of at least 88%. The reason why the overall coverage varies roughly between 90% and 98% is that the line coverage of the Game class depends on the generated deck which has always random cards in it. Therefore, some lines are covered with some combinations of cards and others not. The covered lines differ on every run such that after a few runs every line is tested with a very high possibility. We tried to stay as general as possible in these test cases, but this could not be avoided. There was also a difficulty in testing since the most methods in the game class require an input from the console. There again we tried to reconstruct as many variations of inputs as possible.

Easter Egg
To round this project off, we implemented a small easter egg. Do you need a small hint, maettuu?

CRC / UML / Sequence Diagram

Main	
call game functions	

Game	
provide logic functions: take bets provide cards, etc	I/O Player, Dealer Deck, Card

I/O	
input output	

Iterator (Interface)
+ next():Card + hasNext():boolean

Table	
Slots	Player, Dealer

PlayerSubject	
track hands	Hand

Player (PlayerSubject)	
name money inital bet give card split hand pay, bet	 Card, Hand Hand

Dealer (PlayerSubject)	
give card	Card

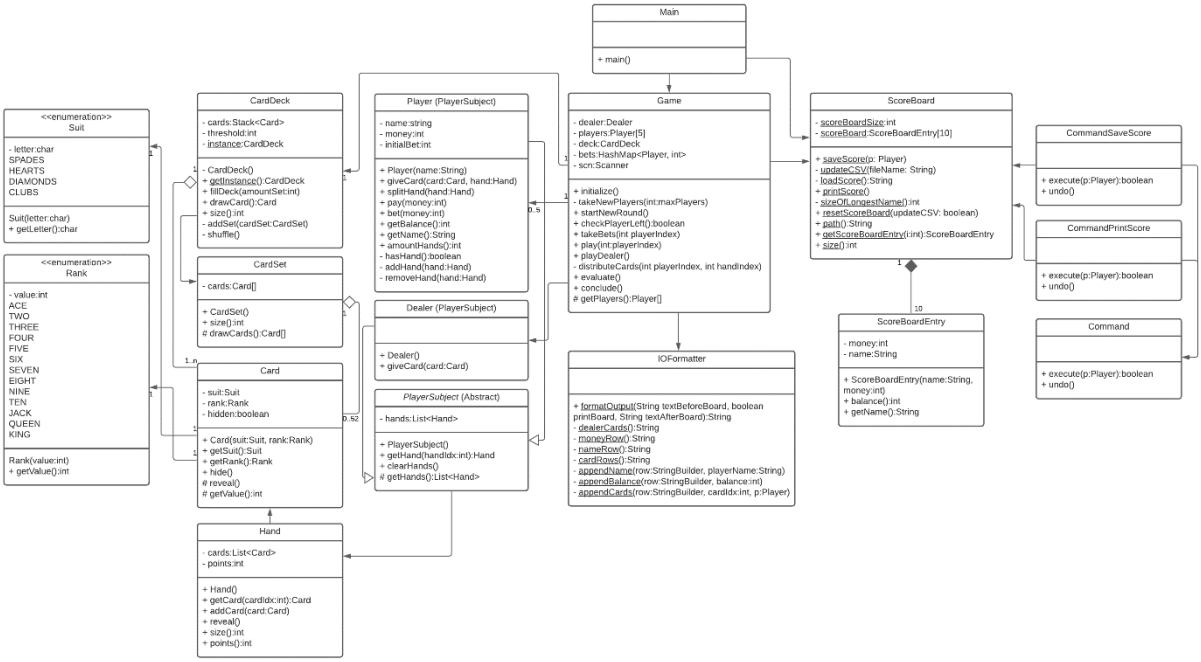
ScoreBoard	
save score load score print score	ScoreBoardEntry

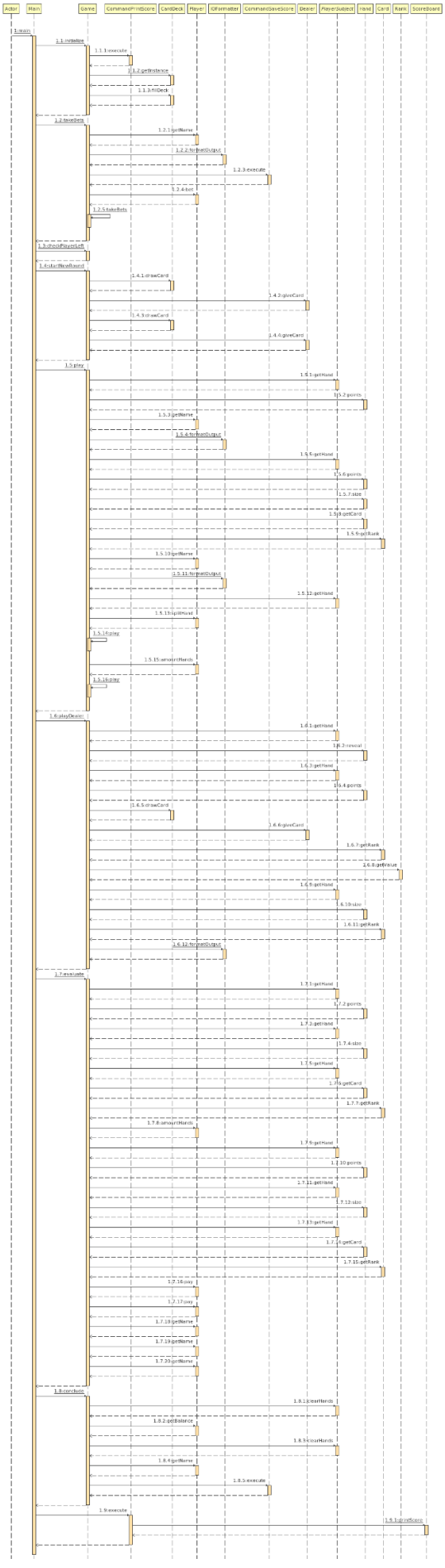
Card	
suit rank hide/reveal	Suit (enum) Rank (enum)

CardSet	
draw cards provide card set	Card

CardDeck	
cards add set draw card shuffle	Card CardSet Card

Hand	
cards points add/remove card	Card Card




Actor
Main
Game
CommandPrintScore
CardDeck
Player
IOFormatter
CommandSaveScore
Dealer
PlayerSubject
Hand
Card
Rank
ScoreBoard
1:main
1.1:initialize
1.1.1:execute
1.1.2:getInstance
1.1.3:fillDeck
1.2:takeBets
1.2.1:getName
1.2.2:formatOutput
1.2.3:execute
1.2.4:bet
1.2.5:takeBets
1.3:hasPlayersLeft
1.4:startNewRound
1.4.1:drawCard
1.4.2:giveCard
1.4.3:drawCard
1.4.4:giveCard
1.5:play
1.5.1:getHand
1.5.2:points
1.5.3:getName
1.5.4:formatOutput
1.5.5:getHand
1.5.6:points
1.5.7:size
1.5.8:getCard
1.5.9:getRank
1.5.10:getName
1.5.11:formatOutput
1.5.12:getHand
1.5.13:splitHand
1.5.14:play
1.5.15:amountHands
1.5.16:play
1.6:playDealer
1.6.1:getHand
1.6.2:reveal
1.6.3:getHand
1.6.4:points
1.6.5:drawCard
1.6.6:giveCard
1.6.7:getRank
1.6.8:getValue
1.6.9:getHand
1.6.10:size
1.6.11:getRank
1.6.12:formatOutput
1.7:evaluate
1.7.1:getHand
1.7.2:points
1.7.3:getHand
1.7.4:size
1.7.5:getHand
1.7.6:getCard
1.7.7:getRank
1.7.8:amountHands
1.7.9:getHand
1.7.10:points
1.7.11:getHand
1.7.12:size
1.7.13:getHand
1.7.14:getCard
1.7.15:getRank
1.7.16:play
1.7.17:play
1.7.18:getName
1.7.19:getName
1.7.20:getName
1.8:conclude
1.8.1:clearHands
1.8.2:getBalance
1.8.3:clearHands
1.8.4:getName
1.8.5:execute
1.9:execute
1.9.1:printScore