Assignment 4 - Blackjack

Novel feature idea: addition possibility to enter HIT/STAY and bet via voice input

Requirements

In this game, one user plays Blackjack via the terminal against the computer who is the Dealer.
At the start of the game, the program shows the amount of money that the Player has (the starting value is 100 CHF). Then, the Dealer asks the Player whether they want to make a bet or go away with the money.
If the Player decides to go away with the money, the game finishes. If the Player wants to play, the Dealer asks for the amount to bet in this round. The Player inputs a bet whose amount must be lower or equal than the available money. Then the round starts.

Each round follows the Blackjack rules, also in terms of which cards are visible to the Player and when (e.g., at the start of each round, the Player sees the values of their card, as well as the visible card from the Dealer). The Player has to input their decisions (e.g., 'hit' or 'stay') on the console. When the round is finished, if the money left to the Player is higher than 0, the Dealer asks again whether to play another round or not; otherwise, the game is finished.

Novel feature: In the beginning of the game the player is asked if he would like to have voice input. If yes, then hit/stay as well as bet is handled via microphone. If not - via console. For bet in this case only limited inputs are accepted: `(ten|twenty|forty|fifty|sixty|eighty|hundred)`

RDD

1. Going though the requirements and underlining nouns reveals the first set of candidates.
2. Excluding some obvious non-classes:
    - Physical objects: computer -> are not needed
    - Other than classes:
        - amount of money, value (field in Player)
        - bet, decision (method in Player)
        - start (method in game)
        - round (probably field)
3. Grouping of relevant candidate classes:
    - Game, rules -> Game
    - Card
    - console, terminal,microphone, input-> inputBehaviour (?)
    - Player
    - Dealer
    - Deck (assuming that cards are stored in the deck)
    - Table (although not from requirements, but might be useful to store everyones cards, was rejected further in the design process as it is a more data driven design rather than a responsibility driven design).

The initial idea was to make use of the state pattern. Every round played would be represented by a state. As this assignment is just a singleplayer game, it would make use of four states, one for the first round, one for the player's turn, one for the dealer's turn and one

for the end of the game. The advantage of this pattern would be to have the possibility to easily extend the game with multiple players.
However during the design phase it was decided that the Strategy design pattern better suits the current system.
The final decision was to use:

- Strategy design pattern
  It was noticed that Player and Dealer are similar classes, which differ only in implementation of hit behavior. Additional feature (mic input) introduced essentially one more input behavior for the Player. Additionally, those behaviors might be changed during runtime (e.g. in the case of VoiceInput mic does not work, it throws exceptions, and behavior is changed to TerminalInput). Strategy design pattern allows for exactly that, since it encapsulates those behaviors in separate classes. So, it provides advantages such as:
  1. strategies can be swapped during runtime
  2. system becomes easily extendable for other behaviors (building in new feature was fast, based on this design pattern) (*Open/Closed Principle*)
  3. implementation details of behaviors are encapsulated
  4. Having two classes (player and dealer) which extend an abstract class helps to avoid code repetition and hence minimize error-proneness, making the system more changeable and extensible

  How: The two classes Player and Dealer are extending the abstract class User. User contains methods, common to both classes, such as reset, hit, bust, showCards, takeCards, ect. The class Dealer contains methods specific to it, such as flipCard. The Player class contains methods specific to it such as WinBet, looseBet, isOutofMoney, and makeBet. The class User has two fields HitBehaviour and InputBehaviour. For each of the behaviors, corresponding Interfaces are created. Currently, there are two classes, which implement `HitBehaviour`: PlayerHitBehavior and DealerHitBehaviour. As for InputBehavior, there are currently three classes: Terminal, Voice and DummyInputBehavior. Dummy is introduced because in the current setup Dealer = computer and his moves are performed automatically. However, in any further visions of the game, it could be that the dealer would also input his moves to the console. In that case it would be very easy to switch his behavior to terminal one.

- Singleton design pattern
  Although not stated explicitly in the requirements, it was decided to store the cards in the deck, making sure that a correct deck structure is preserved and only one instance of deck circulates in the game. For this purpose a singleton design pattern was utilized. Additionally, the pointer to the cards array is never returned, so no other class gets access to it.
  How: The constructor in Deck is made private and the method `getInstance()` is introduced. It creates an instance if it is the first call and stores it in the private field. If an instance is already created, then it returns it without creating a new one.

- Iterator

When cards are given to the player/dealer, it could either be one or two cards. So the size is not clear. In order not to return arrays of different sizes and reveal implementation to the client code, the iterator helps to hide these implementation details. Additionally class Dealer can access cards in abstract class User via getting an iterator.

Advantages:

1. Can vary number of cards return and way to store them
2. Encapsulating of implementation details
3. Can provide access to array without passing pointer
4. If implementation is ever changed and responsibility of printing cards is put to another class, createIterator() interface in User makes it easy.
5. Dealer can flip his cards and iterate over private field of User

How: Class CradIterator with interface next() and hasnext() is introduced. Client code: user/dealer calls iterator.has() to iterate over and next() to access that element.

The resulting CRC

Deck (similar to lecture 5)

Responsibilities:

acting as collection of cards and making corresponding operations with them:

- shuffle
- store all cards
- providing cards
- securing that every card exists exactly once in the game and only one instance of deck can be instantiated (singleton)
- provide info (e.g. size)
- resetting (without creating new instance)

Collaborators:

- Card (deck is a collection of cards, so it uses only constructor of card)

Card (similar to lecture 5)

Responsibilities:

being responsible for its own characteristics:

- displaying itself
- providing its rank, value
- flipping itself

Collaborators:

- none, as card only operates with its own fields, and it is other classes who rely on card

CardIterator

Responsibilities:

- implementing interface hasnext() and next()
- allowing others to iterate over cards

Collaborators: none

Player

Responsibilities: cash operations

- having cash (starting with 100) and checking if there's something left
- updating cash in the end of the game

Collaborators:

- *superclass* User (accessing constructor to pass that it is a player)

Dealer

Responsibilities:

- flipping the card

Collaborators:

- CradIterator (getting the first card in order to flip it)
- Card (flipping it)
- *superclass* User (accessing constructor to pass that it is a dealer)

User

Responsibilities:

- assigning correct behaviors
- resetting itself
- counting score
- operations with cards: taking cards, showing cards, giving iterators to its own cards.
- bust/hit/bet

Collaborators:

- CardIterator (iterating over given cards and also building an iterator for its own cards.
- behavior classes (initializing them, changing strategy to terminal if microphone strategy does not work)
- classes implementing InputBehavior: dummyInputBehavior, terminalInputBehavior, voiceInputBehavior (these classes help to correctly implement read input depending on the chosen strategy in the case of hit/stay and bet amount)
- classes implementing HitBehavior: PlayerHitBehavior and DealerHitBehaviour (these classes help to correctly implement hit depending on the chosen strategy)

Game

Responsibilities:

- storing player, dealer and deck
- keeping track of current turn
- logic of the game (handling round, end of the game, ect.)
- giving cards to participants
- getting main input (e.g. deciding whether game is played via terminal or mic input is accepted, whether in general player makes a bet and starts a game or go away)

Collaborators:

- Player, dealer, user (initializing and calling methods, such as count score, asking for change of strategy behavior, giving them cards, ect.)
- Deck (shuffling, resetting, drawing cards)
- CardIterator (initializing it)

Two classes to implement HitBehaviour:

PlayerHitBehavior and DealderHitBehavior

Responsibilities: implement method hit

Collaborators: none

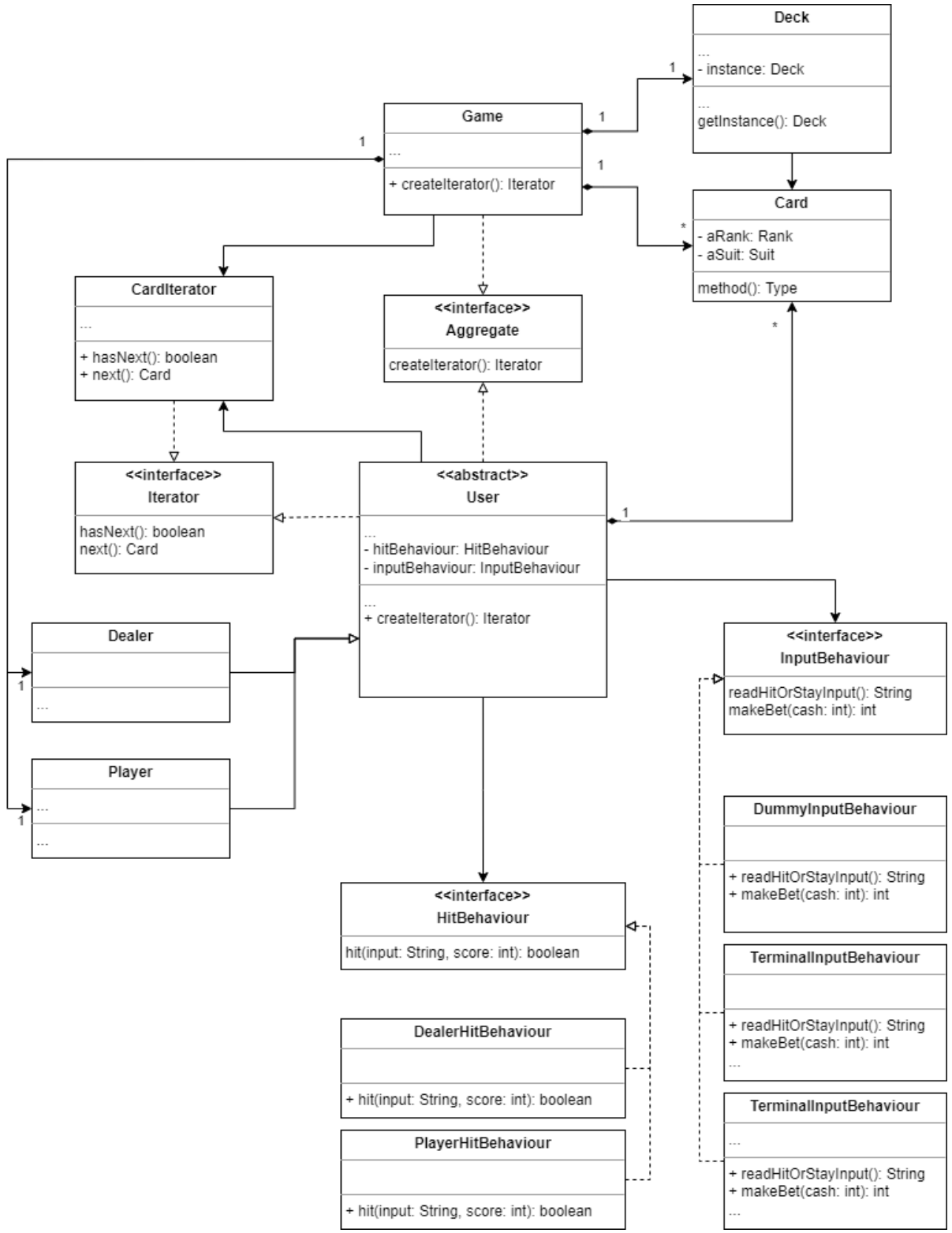
Three classes to implement InputBehaviour:

DummyInputBehavior, TerminalInputBehavior, VoiceInputBehavior

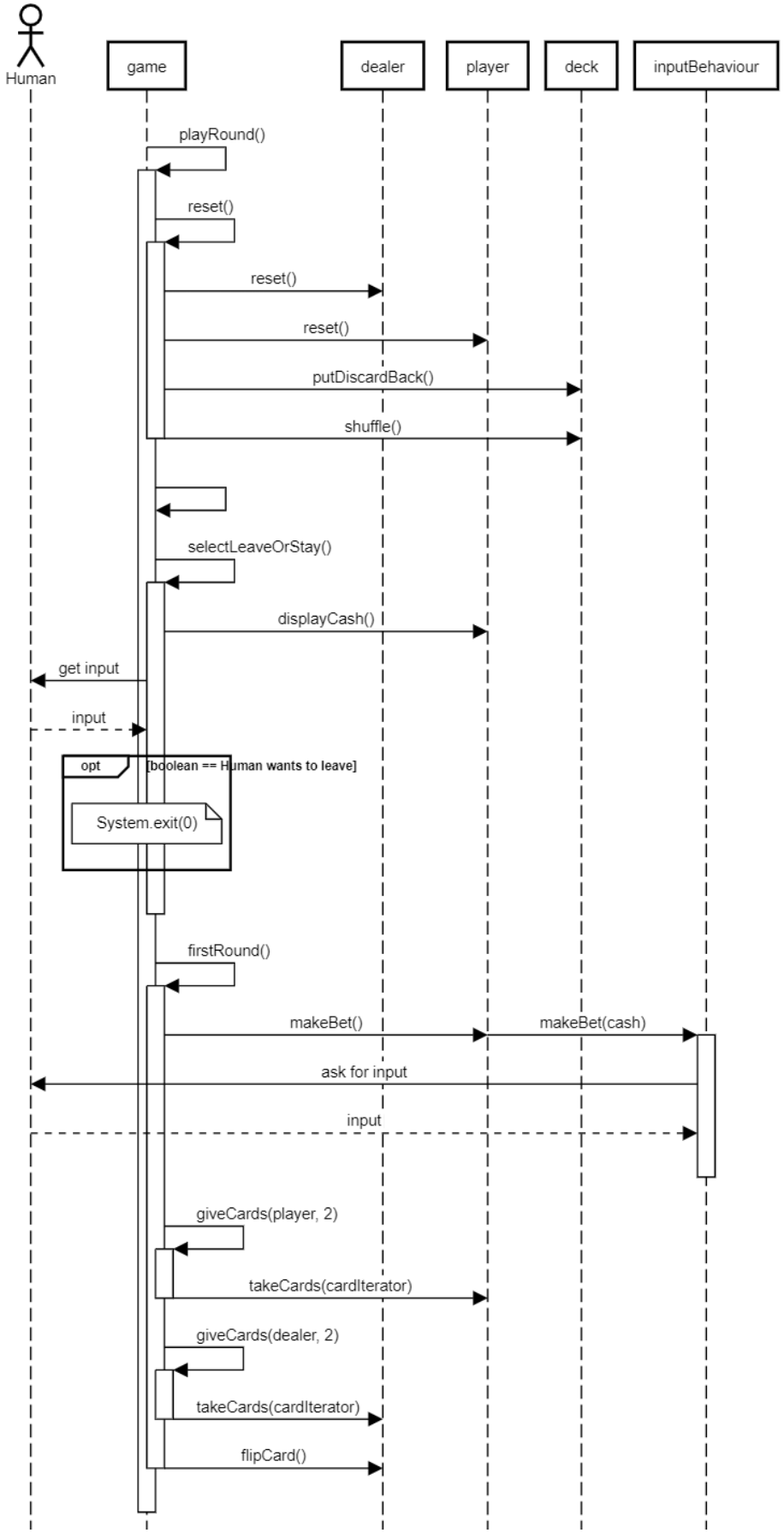
Responsibilities: implementing `readHitOrStayInput()`

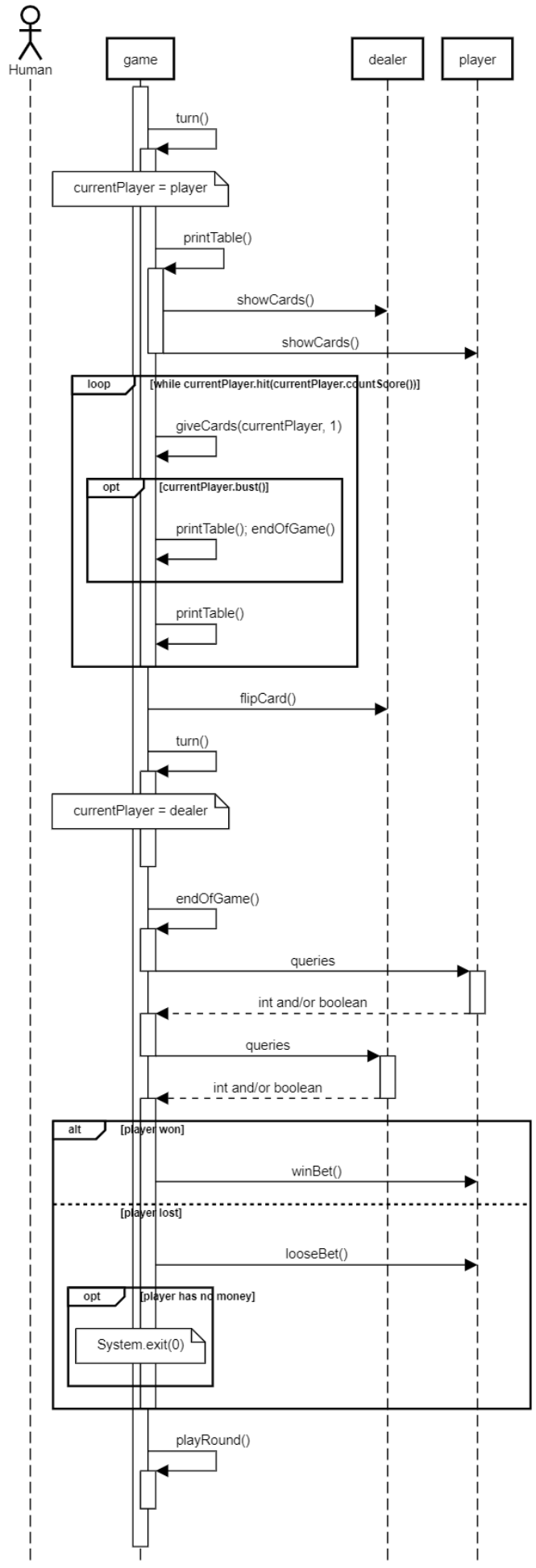
Collaborators: none

Class diagram



Sequence Diagram



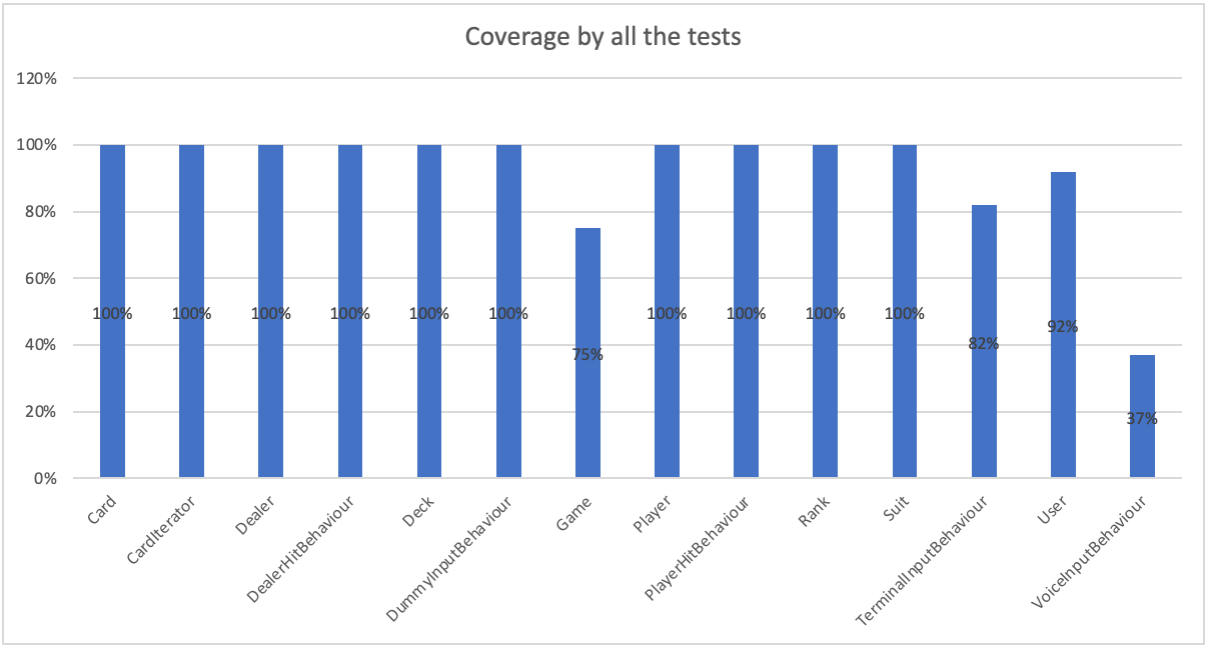

Human
game
dealer
player
turn()
currentPlayer = player
printTable()
showCards()
showCards()
loop
[while currentPlayer.hit(currentPlayer.countScore())]
giveCards(currentPlayer, 1)
opt
[currentPlayer.bust()]
printTable(); endOfGame()
printTable()
flipCard()
turn()
currentPlayer = dealer
endOfGame()
queries
int and/or boolean
queries
int and/or boolean
alt
[player won]
winBet()
[player lost]
looseBet()
opt
[player has no money]
System.exit(0)
playRound()

Testing

The given requirements for this assignment are properly tested. The additional feature appeared to be hard to test because of the voice input (hard to automate) and path dependencies. Automating voice input would probably go beyond the scope of the assignment. Therefore this class was tested only to a limited extent. The overall coverage reached 75 percent.

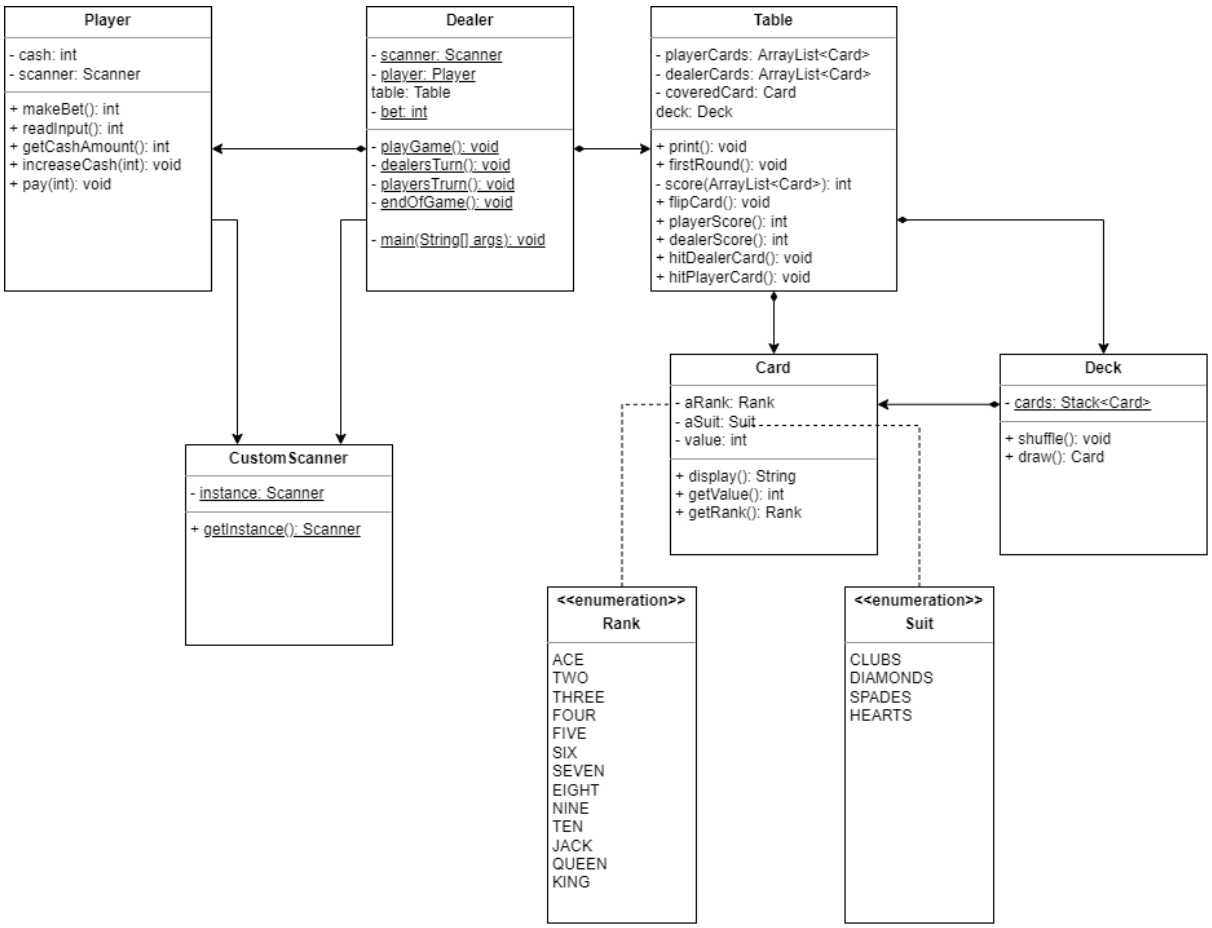
100% classes, 75% lines covered in 'all classes in scope'

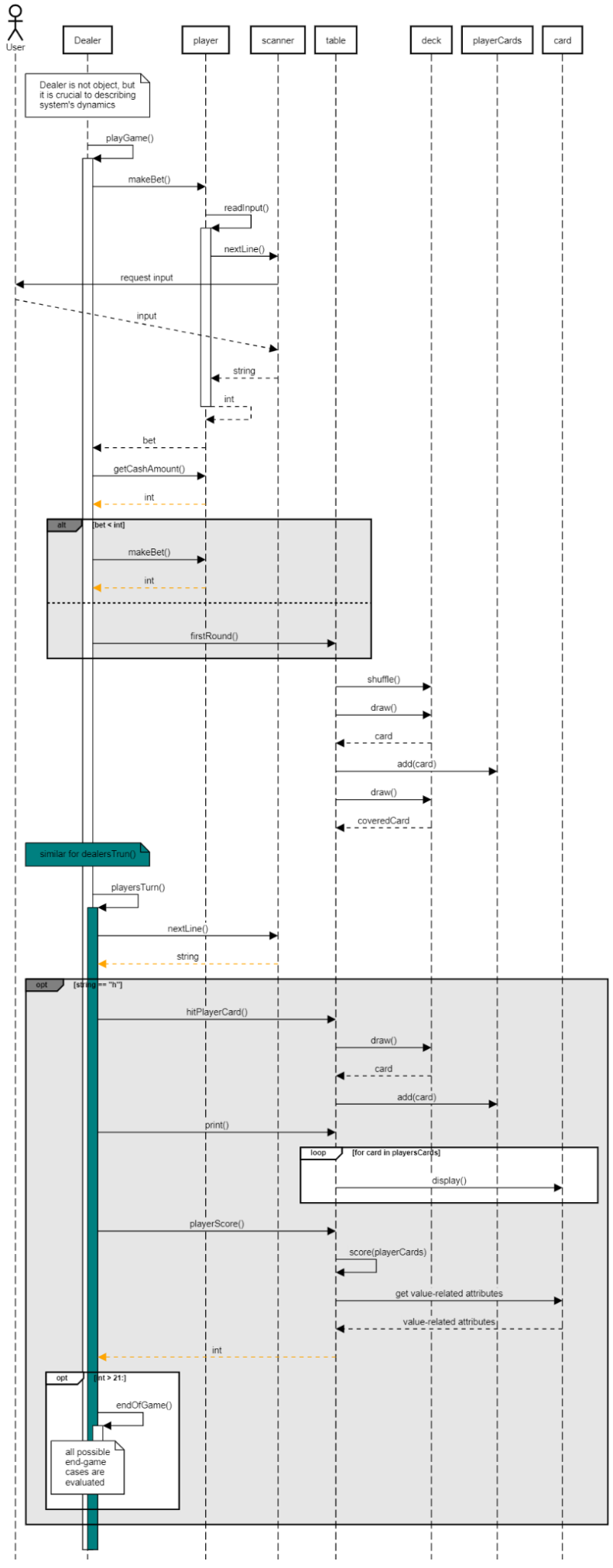
Element	Class, %	Method, %	Line, %
ch.uzh.group38	100% (15/15)	91% (66/72)	75% (220/292)



Previous design versions

1. Design with table without design patterns:




User
Dealer
player
scanner
table
deck
playerCards
card
Dealer is not object, but it is crucial to describing system's dynamics
playGame()
makeBet()
readInput()
nextLine()
request input
input
string
int
bet
getCashAmount()
int
alt
[bet < int]
makeBet()
int
firstRound()
shuffle()
draw()
card
add(card)
draw()
coveredCard
similar for dealersTrun()
playersTurn()
nextLine()
string
opt
[string == "h"]
hitPlayerCard()
draw()
card
add(card)
print()
loop
[for card in playersCards]
display()
playerScore()
score(playerCards)
get value-related attributes
value-related attributes
int
opt
[int > 21:]
endOfGame()
all possible end-game cases are evaluated

2. CRC cards for observer design pattern:
   There was an attempt to use the observer design pattern at first, where the player (potentially multiple players) as well as the dealer would be notified. The game was built according to this design and it was working and was extendable to multiple players. However, it was decided to try a more fitting design. Below are two versions of the observer design:

version 1:

Game (Notifier (part of observer dp))

Responsibility:

- initialize itself, ask the user how many players are playing
- initialize players and dealer and store them in its fields
- know the rules of the game, i.e. when is who's turn, when cards are dealt, what happens when dealer busts etc.

Collaborators:

- Dealer: create, store and tell dealer what to do and when
- Player: create, store and tell what to do and when

Dealer

Responsibility:

- create the deck, store it and interact with it
- create multiple decks in case of too many players (like one for every 3 players)
- give cards to players
- store its own cards
- perform its own turn and count its own score
- get the cards and scores from players and dealer and print them out
- paying player in case of a win, kicking player out if broke

Collaborators:

- Player: give cards (player is passed to dealer from game)
- Deck: create, store and shuffle
- Card: store personal cards
- Game

Player (Observer (part of observer dp))

Responsibility:

- make bet
- handling its cash
- store its own cards
- perform its own turn and count its own score
- decide when to print his and dealer's cards

Collaborators:

- Card: store personal cards
- Game
- Dealer

Deck (Singleton)

Responsibility:

- create and store all cards
- shuffle the cards

Collaborators:

- Card: create and store a “deck” of them
- Dealer

Card

Responsibility:

- represent a card

Collaborators:

- Dealer
- Player
- Deck

version 2:

Game

Responsibility:

- initialize itself
- initialize players and dealer and store them in its fields
- know and implements the procedure of the game, e.g. who gets cards first, what happens when someone busts

Collaborators:

- Dealer: create, store and tell dealer what to do and when
- Player: create, store and tell what to do and when

Dealer (Subject (observer dp), Aggregate (iterator dp))

Responsibility:

- create the deck, store it and interact with it
- give cards to players through iterator through notifyObserver()
- store its own cards
- perform its own turn and count its own score
- (showing its own cards)

Collaborators:

- Player: give cards (player is passed to dealer from game)
- Deck: create, store and shuffle
- Card: store personal cards
- CardIterator: create and send
- Game

Player (Observer (part of observer dp))

Responsibility:

- make bet
- handle its cash
- store its own and copy of dealers cards
- perform its own turn and count its own score
- decide when to print its and dealer’s cards (showing its own cards)

Collaborators:

- Card: store cards

- CardIterator: receive and use
- Game
- Dealer

Deck (Singleton)

Responsibility:

- create and store all cards
- shuffle the cards

Collaborators:

- Card: create and store a “deck” of them
- Dealer

Card

Responsibility:

- represent a card
- represent a hidden card

Collaborators:

- Dealer
- Player
- Deck

CardIterator

Responsibility:

- pass multiple cards without disclosing implementation of their storage

Collaborators:

- Dealer
- Player