

Project document

Personal information

Eetu Mustonen

782632

Informaatioverkostot

3rd year

30.4.2022

General Description

I chose the casino card game as the topic of my project because I am interested in card games and the possibilities of a deck of cards. So, I find it natural to combine my interest in card games and programming with this project, and I hope to learn something about combining these that I could use in the future.

In the game, points are achieved by collecting cards on the table, with player's hand cards. I will create algorithms for executing user commands and tracking the progress of the game. Before algorithms, however, appropriate data structures must be created for parts of the game, such as cards and players.

I was aiming for a moderate implementation of the project and, where possible, some demanding level features. In the end, only the moderate implementation was achieved, due to lack of time that I was able to dedicate for this project.

Rules of the game

One must collect points which are calculated after every round.

The game continues until someone reaches 16 points.

A player can play out one of his/her cards: it can be used either for "Capturing" cards from the table or to just "Trail" it on the table.

If the player cannot take anything from the table, he/she must put one of his/her cards on the table.

The next two tasks are done automatically:

If the player takes cards from the table, he/she puts them in a separate pile of his/her own. The pile is used to count the points after the round has ended.

Player must draw a new card from the deck after using a card so that he/she has always 4 cards in his/her hand.

The number of cards on the table can vary. For example, if someone takes all the cards from the table, the next player must put a card on the empty table.

(When the deck runs out, everyone plays until there are no cards left in any player's hand).

Capturing:

Player can use a card to capture one or more cards of the same value and/or cards such that their summed value is equal to the used card.

For example, a player can capture with a J (value 11):

{J}, {4,7}, {4,7, J}

If some player gets all the cards from the table at the same time, he/she gets a so called sweep which adds 1 point to the player's points.

Special Cards:

There are a couple of cards that are more valuable in the hand than in the table,

Aces: 14 in hand, 1 on table

Diamonds-10: 16 in hand, 10 on table

Spades-2: 15 in hand, 2 on table

For example: you can take two eights from the table with Diamonds-10

For the other cards the values are same in the hand and on the table.

Scoring:

When every player runs out of cards, the last player to take cards from the table gets the rest of the cards from the table. After this the points are calculated and added to the existing scores.

The following things grant points:

Every sweep grants 1 point.

Every Ace grants 1 point.

The player with most cards gets 1 point.

The player with most spades gets 2 points.

The player with Diamonds-10 gets 2 points.

The player with Spades-2 gets 1 point.

User interface

The program is started by running the Cassino object. The user interface is a very simple textual interface in the console of an IDE. User commands are very limited and the program provides lots of guidance throughout the game. User commands are for adding players by giving a name for each one, and moving cards around by giving the move's type and which cards are to be moved. In certain points the user is asked if the game continues. Saving and loading the game is also possible through the user interface.

Program structure

Class: Game

Game is an instance of the Cassino game and includes game rounds, players and player score. The class has an ID for files. Game class is able to add players, maintain points, start a new round and save the current points. This class is associated with the Player and Round classes.

Class: Player

Player objects have a name that acts as an identifier and it does not need other variables or functions because the player's hand cards, stack, and score are processed in other structures.

Class: Card

An instance of Card models a playing card with suit and value.

Class: Deck

A Deck is a structure for maintaining some amount (0-52) of cards in a collection and modifying that collection. A full deck of playing cards contains 52 different cards that are a combination of 4 suits and 13 values. Player's hand cards and stacks, table cards and the main deck are instances of a Deck. The cards are moved between the decks with the class's methods. The class also has a method for counting the amount of spades and points that the collection of cards adds up to.

Class: Round

The Round category models one round of play that is launched from the Game class. It models players taking turns and moving cards around based on player's decision. According to user commands, this class handles cards between players, the table, stacks, and deck, and checks the validity of capturing before executed. In the start of a round a full deck is shuffled and 4 cards are dealt to each player and on the table. Then turns are taken until each

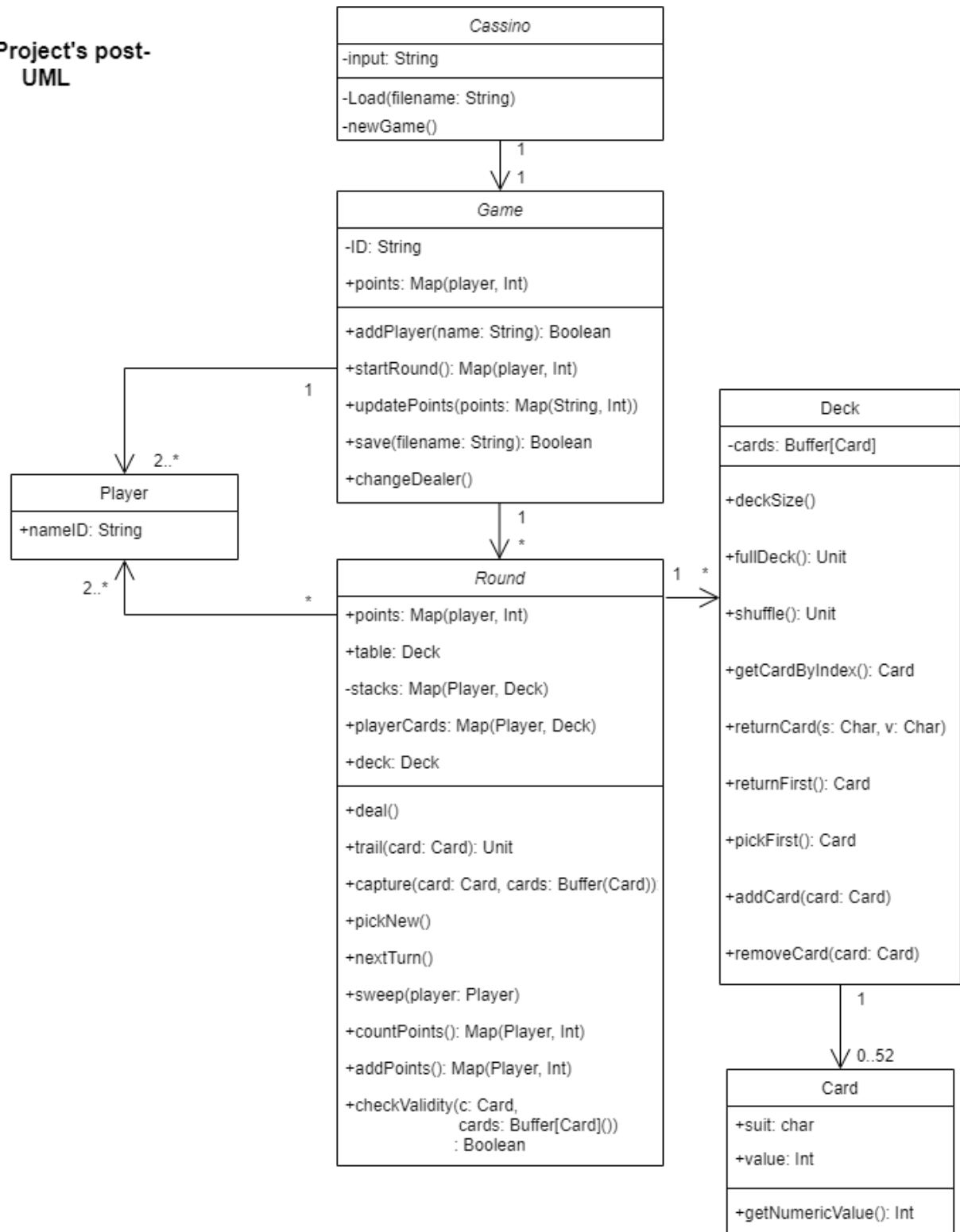
player runs out of cards. In the end of a round points are counted based on the players' stacks. The Round class has an association with the Deck and Player classes.

Object: Cassino

Cassino is the interface to the program. It has an association relationship with the Game and creates a new Game object that is controlled by user commands. The necessary instructions are printed out to the user at the beginning of the game and as the game progresses, giving appropriate instructions for different situations.

The user interface works as a while loop that stops with the "quit" command or when one of the players reaches 16 points. Inside this loop is other loops for checking valid user input and running the rounds.

OS2 Project's post-UML



Algorithms

The most important algorithm in the program is the `checkValidity()` method in class `Round`. What it does is that it checks if the player's attempt to capture, i.e., pick cards from the table with a card in the player's hand, is valid without the user telling how the cards should be interpreted (check the Rules of the game to see how capturing works).

The method takes two parameters, the player's card and a collection of table cards the player wishes to capture. First the cards must be given a numeric value because there are the picture cards and some cards have special values. The table cards' values are inserted into a collection `C` that is then modified recursively until it is empty or some condition shows that the move is not valid.

Two conditions are checked before the recursive search: if the table cards contain a card bigger than the player's or if the sum of the table cards is not a multiple of the player's card the move is invalid.

The recursive function takes two parameters: "values" that is a set of cards which's validity is checked based on the "target" value. First the function is called with the player's card's value and the table cards' values that the player wishes to capture. The values are sorted in a descending order and if the difference between the target and the biggest value is included in the values, the difference and the biggest value are removed from the original collection `C`. If "values" doesn't contain the difference then all values bigger than the difference are filtered out. The difference is removed from the original collection `C` and the recursive function is called again, parameters being the difference and the filtered values. If at some point `C` is empty then the capture is valid because all the valid card combinations are removed. Otherwise, a condition is met where it is clear that the move cannot be valid.

Another way of doing the search is a very exhausting one called Brute Force search that goes through all possible combinations of forming sub-sums that equal the target value.

Data structures

Mutable `LinkedHashMaps` are used for storing player data. Keys are players and values are points or decks. `LinkedHashMap` is better than regular `Map` because it maintains the inserting order. All other collections are mutable `Buffers` because they are easily modifiable with inserting, removing and searching elements independently. The collections are mutable so that the same collection is modified instead of creating new ones. This way it is easier to avoid bugs from mixing up the collections etc.

My own data structure is the `Deck` class, that stores 0-52 cards in an array. The class has methods for modifying and searching the deck.

Files and Internet access

The game results can be saved at any point of the game but only the point situation between rounds can be saved.

A game can be loaded at the start running the program.

The file format is as follows:

Cassino	//indicator that it's a game of Cassino
id	//game's id
player : points	//first player and his/her points
...	
player : points	//until last player's name and points
END	//"END" to signal the file is at the end

So for example:

```
Cassino
12345
Adam : 3
Brian: 2
Charlie : 5
END
```

Testing

The testing was done along the development. Some of the early tests are deleted but there is multiple test objects in the file test.scala.test that cover most of the programs functionality and especially the most bug sensitive areas of the program like more complex methods and file management. During the implementation process I used test objects, debugger and println-method for checking certain values in the console. The testing done is not going to guarantee the program being bug-free, but so far all the tests pass. Only thing missing from the original plan is Unit testing. With Unit testing I would have tested the same things as with the regular test objects but it would have been good practise.

Known bugs and missing features

Currently there is no known bugs. All the moderate features are implemented.

3 best sides and 3 weaknesses

Best sides:

1. Good coding style with function and variable names as well as whitespace usage.
2. Classes and methods are divided efficiently and logically except for user interface.
3. As a textual user interface the UI is user friendly.

Weaknesses:

1. The user interface works fine but could have been divided into helper functions instead of a big chunk of code inside while loops.
2. Lack of testing different scenarios
3. Deck's methods could have been given more thought and simplified.

Deviations from the plan, realized process and schedule

The order of implementations and the hours used were close to the original plan. Of course, the plan didn't cover 100 percent of the program, but it was complete enough to not face any major changes along the implementation. The main difference is the 5 week break I had to have from this course due to other urgency and therefore didn't have time to develop a graphical user interface. Other differences are:

The user interface actually took about 10 hours when the plan was 6 and it was implemented after all the other classes, because it was not clear if it was going to be graphical or textual.

The file format and saving and loading was done as the last thing of the project, when the original plan was to do it in the beginning to think about file format and implementation of other methods. But it took a little less time than planned, because saving is only possible between rounds, so only points are saved.

The Deck structure was supposed to be only a full deck of cards, but I found out it's efficient to use it for all possible Card storages.

Also, I realized that class SpecialCard is not needed, because the special values are only used when capturing. And the only function that uses the cards' actual values is checkValidity(). So in the implementation of checkValidity() the special cards are handled differently when used.

In terms of hours used to testing I did only about 10 when the plan was to do 20 hours.

Things that went according to plan:

The order of implementing the lowest- and highest level classes first, combining the top-down and bottom-up approaches worked and the hours used for Player, Card, Deck and Game was about 4 hours as planned.

The Round class was a more difficult class to implement as planned and took 12 hours as was the plan.

Final evaluation

Based on the testing done so far, the game works as it should. But I wouldn't bet on it being bug-free as testing does not cover that many scenarios. What I would improve about the coding style is definitely the "engine" in the user interface. It is really hard to follow due to loops inside loops and all the code being out there without using helper methods. I could have used helper methods and the actual engine more readable. Another thing is the Deck class where the methods weren't given that much of a thought but just implemented when necessary. This leads to the program not being easily modifiable.

I really wish I would've had the time to make a graphic user interface as it would have been quite simple but also very efficient with this kind of a game where it requires just clicking. I'm still happy with the textual user interface I came up with.

I'm happy with how I divided classes and methods and satisfied with my coding style. I'm also happy with the use of Exceptions to keep the user interface running.

If I was to do this project again the most important difference is that during the development, when faced unplanned features or changes in the original plan I would stop and actually plan what I'm going to do instead of just starting to write code and see where it goes. I'm not saying everything should be perfectly planned but at least in implementing the user interface and Deck I rushed with writing the code. Also I would do a more covering testing and the graphical user interface.

References

How To Build A Simple Card Game With JavaScript

<https://www.youtube.com/watch?v=NxRwIZWjLtE&t=387s>

Scala library

<https://www.scala-lang.org/api/2.12.9/index.html>

Geeksforgeeks for tutorials and examples

<https://www.geeksforgeeks.org/>

Stackoverflow for similar problems other have run into


<https://stackoverflow.com/>

Appendixes

Example of a situation where a round has started, here the player sees his/her own cards and the table cards and is asked to give a command for executing a move.

A, it's your turn.

● A: ♥7 ♦8 ♠9 ♥A

B:    

C:    

Table: ♣8 ♥5 ♦Q ♥T

Write

'c' to capture

't' to trail.

'help' for rules.

'quit' to save and exit the game.

/*

Let's say the player chooses to capture the ♣8 with the player's ♦8. So first the player writes 'c' for capturing and press enter. Then player is asked to insert the indexes of the cards used for the move. Selecting cards is not that smooth with textual user interface but works once gotten used to.

*/

c

Give the index of your card you would like to capture with.

Indexing starts from 0

1

Now give the indexes of cards to capture from the table, separated with commas. For example 0,1

0


Capture was succesful!

/*

Now it's the next player's turn and both of 8's are moved to A's stack.

*/

B, it's your turn.

A: 

● B: K 8 J 4

C: 

Table: 5 Q T

Write

'c' to capture

't' to trail.

'help' for rules.

'quit' to save and exit the game.

Example of a situation where round has come to an end:

The round is over. If you wish to save the current results write 'save', else write 'play'.

/*

If user inserts “play” the next round starts, but let's see what happens with saving at this point.

*/

save

Game saved, you can load it later with this game's id: 12345

If you wish to start the next round write 'play', else write 'quit'.

/*

Now the player is given an opportunity to exit the game or keep playing. The player is also reminded of the game ID which is needed to load the game later.

*/