Project document, Cassino

Aaron Penkkala 1028010
Computer Science, tekniikan kandidaatti ja diplomi-insinööri
2023
24.4.2023

General description

The project which I have made is the card game Casino (sometimes spelt as “Cassino”). The game is in a text-based UI, and satisfies the criteria specified in the “Moderate” requirements (Text-based UI, 2-N human players, saving and loading). My original plan was to make the game with the “Demanding” requirements (moderate requirements + GUI and computer controlled opponents), but unfortunately I did not have enough time. I will most likely continue to program the GUI in my free time, as I believe this will be a good first “big” project, if I also implement the GUI elements. My inexperience with GUIs also meant I believed I did not have enough time to fully learn to program the GUI before the deadline, hence only the “moderate” requirements were fulfilled.

User interface

The card game is started by starting the ‘run’-function in ‘main.scala’ found in the project. The user is greeted with a welcome message, and a reminder that this version is the

“deck-cassino” version of Cassino. The game then prompts the user to start a completely new game, or load in an existing game from the save file included with the project. There is only one save file included which can be loaded to and from.

If the user decides to start a new game, they are then prompted to input the amount of players (2-N), and then input their names. After this, the game starts with the round-based system specified in the A+-page.

If the user decides to load an existing game, the program will try to load the game from the file ‘save.txt’ included with the project. If the loading is successful, the game continues from where it was left off, and continues as normal until the game is won. After a game is won, the program simply closes.

The user can also input the command ‘/halt’. This will pause the current game, whether it be loaded or a new game. When halted, the user can either continue the current game, save the current game to the save file, overwriting the old saved game with the current one, OR they can quit the game and therefore, close the program.

You play the game by inputting the number or index next to the card you want to play from your hand, and then inputting the numbers or indexes of the cards you want to pick up. When picking up multiple cards, please separate the indexes with a comma ‘,’ e.g. ‘0,1,2,3’. You can only play one card at a time from your hand, but you can pick up many cards from the table.

The game will only accept inputs that are numbers, and which are in range of the current table. If the table is empty, you pick a card that you want to place on the table. If you do not have any

combinations or any matches from the table, just pick a card to place on the table, and input any of the indexes of the table cards. The game will then automatically place the selected card on the table.

Minor tweaks have been made from the previous plans, such as not including a ‘rules’-section or -command.

Program structure

The program is split into 10 different classes. These include: Card, Deck, Game, Hand, loader, main, Player, saver, Stack and Table. Most of them are quite self-explanitory.

Card represents a playing card with a value and a suit. This includes both ‘text-based’ names, such as ‘Two of Hearts’ or ‘King of Clubs’, as well as a format I implemented which is used for example in the save file. This format has two digits representing the value of a card, followed by a character representing the suit of a card, e.g. ‘02H’ for Two of Hearts, and ‘KnC’ for King of Clubs.

Deck represents a collection of these cards. Some of its functions are, for example, picking n-amount of random cards from the deck.

Game includes the heart of the program, the humanAlgorithm. This algorithm is at the center of the game, and is what allows players to pick up and place cards from and to the table. This is specified more in the next chapter.

Hand represents the hand the player has at a given time.

Loader allows the game to be loaded from the save file. This took some time to parse from previous projects, as well as the ‘main’ class for continuing the game. It has the functions loadGame(), which parses the text files and returns the information, such as players, the turn etc. The function loadGameFromFile() uses that information and starts a ‘new’ game from those.

Main is the class where the game/program can be started. The game always starts there, and then branches off depending on user inputs. Classes ‘main’ and ‘loader’ include a lot of exception handling for the user input, as the user can input almost anything when it comes to text-based UI, as compared to GUIs. (numbers instead of text, negative numbers, blank inputs etc.)

Player represents a single player, with a name, Hand and a Stack. They also have to have points, since the game is finished when atleast one player has atleast 16 points.

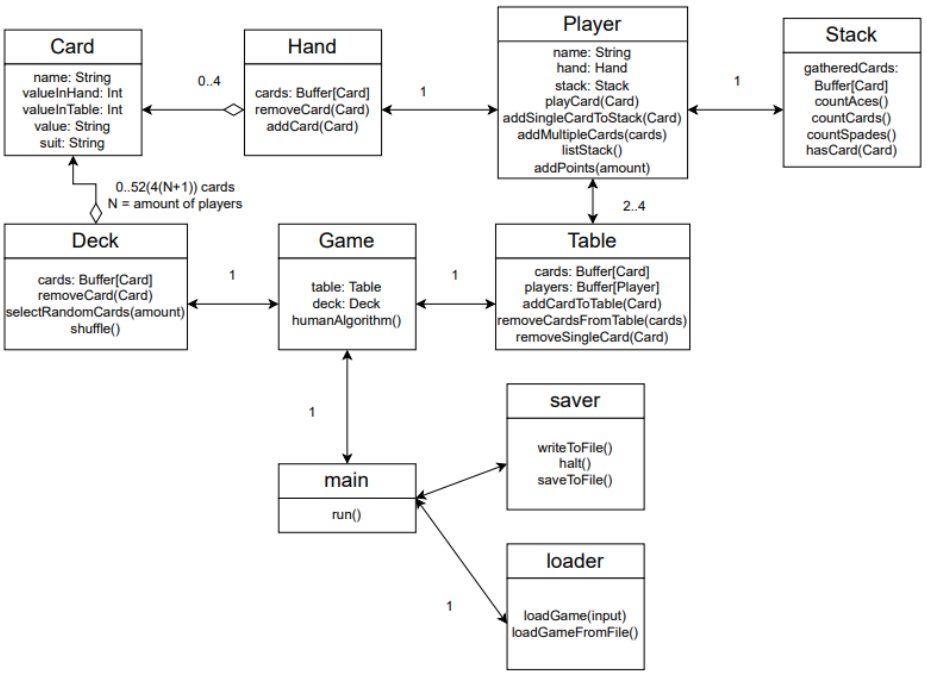
Saver allows the game to be saved. It takes information from the current game and transforms it into text form, which can then be written into the save file. It also has the functionality of quitting the program and continuing it after saving.

Stack represents a stack of cards the player has collected during the current game. This includes the cards taken from the table as well as the cards which the players used to get those cards from the table. These Stacks are counted up at the end of the game to determine the points each player earned that round.

Table represents a playing table. It has players in it and cards placed on it.

Most of these classes are totally transparent with eachother, allowing their variables to be accessed from other classes. This was a concern of mine, as I felt like every time I tried setting something to private, I discovered I needed to access it from another class. Therefore almost all of the classes are public.

The original UML-diagram holds up pretty well, but there are a few things changed, added and tweaked from it.



UML-diagram of the project

Algorithms

The main algorithm of the program is the humanAlgorithm, found within ‘Game.scala’. The algorithm takes a single card played by the user, as well as the cards the player wants in exchange for that card.

The algorithm works by first picking up any cards from the table that have the same value as the played card. These cards are removed from the table, and added to the player's stack. Next the algorithm produces combinations of the selected cards on the table with Scala's built-in functions, first as combinations of 2, then 3, all the way up to the amount of cards wanted by the player. If a combination's sum is the same value as the played card, these cards are removed from the table and from future combination checks. This checking goes on until all suitable combinations are found. If no suitable combinations are found, the played card gets placed on the table, and the turn changes. If suitable combinations are found, those are removed from the table, placed into the players stack, and the game continues.

This algorithm also takes care of sweeps by checking if the table is empty after a player has successfully taken cards from the table. A simple check in 'main.scala' takes care of when a player plays a card when the table is empty, and simply places the played card on the table.

The computer AI for the "Demanding" stage of this project could probably have been coded similarly as this one. There would need to be some sort of weighting system, where the AI has different weights for getting aces, spades, special cards, as many cards as it can, and lastly just anything at all from the table. I'm not sure if I will try to implement that in the future.

Data structures

The main data structure used in this program is mutable Buffers. These were the best option in general, considering the

nature of a card game. You have the player's hands, which change as the game goes on. You have the deck which changes, the table's cards change and also the Stack which changes. As I predicted in the general and technical plans aswell, Buffers were the best data structures to use because of all the functionality they have, but also their mutable nature.

Much of the information inside the classes and functions are also variables. This was again because these values change so much during the course of the game.

The cards, as mentioned before, have two 'names' in two different forms. They have the long form, e.g. 'Two of Hearts', and then the short form '02H', which I have just called originalForm in the program.

Files and Internet access

This program has no access to the internet. The save file, 'save.txt' is included within the program. As the name suggests, this is a simple text file with the information being written in strings in human-readable form. This is a format I designed myself, and the information is not for instance in JSON format.

For the saving/loading to be successful, the save file needs atleast this information:

- Every save file starts with 'save'
- Keywords start with '#', for example '#game metadata' or 'deck'
- The save file needs at least two players under '#game metadata'

- ‘#turn’ is needed so that the game knows where to continue from
- Players are represented by 3 things; ‘#playerN’ contains the cards in the players Hand, ‘#stackN’ contains the cards in the player’s Stack, and ‘#pointsN’ contains the points that player has. (N is the player number, ranging from 1..13”). These can be missing, and the loader should interpret these as the player having no cards left, no points etc. Saving the program will rewrite these keywords if they are missing
- ‘#table’ represents the cards on the table
- ‘#deck’ is the current deck being played, whereas ‘#origDeck’ is the original deck, which the players will be given cards from when a new round starts

An example of a working save file can be found here.

Exceptions / corruptions within the game file cause an exception to be thrown. The exception has information on why the loading/saving was unsuccessful, and usually prints out where the problem is.

Other files/settings should not be needed to start the program, as it is a very simple Scala terminal based program. I have tested the program and got it working on at least 2 different Windows computers, and a Linux machine. If you have any trouble, please send me a message.

Testing

The testing of the program was mainly done through actually trying to play the game. I felt like programming unit tests was

far more difficult to test something like a card game. When programming new methods for example, I started out in Scala REPL, then moved on to actually implementing it in the program, and then testing it through playing the game. I felt like this worked for me in this particular project, but for other projects building solid unit tests will definitely be more efficient. I also built test functions within the classes, and tested things in smaller packages.

From my testing and testing by playing with other people, I think I have covered most bases for errors. I programmed exceptions for every user input I can think of, and for problems such as a save file corruption, I tried giving the user feedback on what went wrong, as fixing a corruption I felt like was out of my league.

The testing plan described in my technical plan differs somewhat. I did not eventually build any unit tests for the algorithms, as I felt like it would have taken away time compared to just testing the algorithms by playing and analysing what went wrong there. Repetition was definitely a big part of the testing.

Known bugs and missing features

The main missing feature was the GUI. I knew pretty early that I was not going to make the computer players because of the lack of time, but I wanted to atleast get the GUI working. Unfortunately that was not the case, but I think I will implement it on my own time. I’ve dedicated quite a lot of time for this project, and the GUI would be the ‘icing’ on the moderately

satisfactory cake. After the GUI I feel confident showing this to people as my first actual project I have coded from scratch.

The loading and saving portion of the program was not really specified anywhere, so I felt it was fair to just throw an exception and halt the program if the save file was corrupted etc. Also the requirement of '2-N' players was also a bit hazy, so AT MOST my program will be able to support 13 players. After 13 players there are not enough cards to give everyone, even if a full deck is at play. I assume this is fine, since it was not specified, and playing a card game with 13 players is not really ideal even in the real world.

The code might not be the most optimal, or easy to read, but I have tried to atleast comment to the best of my abilities. Some of the code was parsed from previous exercises, such as 'Kalevala' or 'Chess' from this OS2 class, so they are a bit messy.

I could have found a better implementation within the 'loader' when addressing the '#player' keyword. Right now it is a mess of the same code repeated over 1-13 players. I did not find a clever way to repeat code in a match-case structure, therefore I had to make all of them their own case.

3 best sides and 3 weaknesses

Best :

- I am quite happy with the UI, even if it is text-based. I feel like everything you need to know is explained, granted you know how to play Cassino in the first place.

- The exception handling was pretty good in my opinion. It gives you explanations on why your input is wrong, and I haven't found a way to bypass it
- The saving and loading worked surprisingly well in my opinion. Granted, it is just a text file and not some fancy save format, but as for the experience we have so far, I'd say it is pretty good.

Weaknesses :

- Messy code. Sometimes hard to read and analyze. Repetition within the code.
- Optimization / data structures used. Optimization does not matter in a program on the scale of this project, but for future projects possibly finding faster/more valid data structures could be useful, instead of using an all-round and flexible solution such as Buffers. Also turning public variables that could be turned into private, same with functions.
- GUI missing, not very hard to only get something working within the Scala terminal.

Deviations from the plan, realized process and schedule

As said multiple times by now, my original plan was to make this project in the "Demanding" requirements (or atleast the GUI). The schedule was followed by its main points, so the implementation of things did hold up. First programming the classes, then the algorithm for human players, then the interface and lastly saving and loading.

However the timing was a bit messy at times. Whenever it was exam week or Titeenit over the weekend, the time-schedule flew out the window. Then again, sometimes I got a huge boost of inspiration and almost finished something I planned to do for a week in two days.

Other classes such as Databases took some time aswell, and personal ‘issues’ such as moving houses took a couple days as well. Overall I’m happy with what I achieved in the time I was able to use on the project.

Final evaluation

As I am writing this I just finished finding a couple of bugs in the code, such as a ‘cheat’ where the player could input multiple indexes when picking up cards from the table, therefore duplicating cards in their stack. Problems such as these were very common during the development of the game, as giving the user the freedom to input anything they want is truly frustrating. There are so many things to consider when a user inputs anything, and how you handle those inputs is very important. I also thought at some point to just program the game with the GUI first, as this would eliminate the input-mess that the text-based UI brings. However, I am happy with how this turned out, and I feel like I have taken most of the possible things into account when testing this program.

Overall I am quite happy with myself and this project. It is not all it could’ve been, but hopefully I still have the motivation in me to finish the GUI, and then truly claim this as my first real project. Better implementation of data structures etc. will come

with time and practice, and I am quite happy that I was able to overcome any challenges I faced along the way.

Extending this project I feel is not particularly hard, but changing something might cause some headaches. For instance I have two nearly identical but not totally identical versions of the actual game, one in ‘main.scala’ (a new game) and one in ‘loader.scala’ (loading an old game). As of right now a change in one does not change the other, so stuff such as UI-changes need to be written into both separately.

If I was to start this program over again, I think I would still do it the same way I did now, as building a good foundation with the text-based UI enables you to more easily transform that into graphical form. If you decide to build the GUI first, then troubleshooting and testing can be quite hard, because bugs may also occur because of the GUI, not the algorithms for example.

References

As for references, I mainly used the scala documentation, https://www.scala-lang.org/api/3.2.2/. I also consulted the previous exercises and materials of this course, as well as the materials of course Programming 1 and Programming 2. Helpful websites for Scala such as https://www.baeldung.com/scala/, https://alvinalexander.com/ and https://www.geeksforgeeks.org/ also helped during the programming of this project.

Appendixes

An example of a successful save file and how to start a new game can be found here. The project can also be found at https://version.aalto.fi/gitlab/penkkaa1/casino-project-final.