APRIL 26, 2020

# CASSINO GAME

CS-C2120 – OHJELMOINTISTUDIO 2, DOCUMENTATION

UHARI VIIVI
885513
Tietotekniikan kandidaatti, grade 2020

## General description

My program is a virtual Cassino game with a graphic user interface and with the possibility to play against computer opponents which means the project is completed on the demanding assignment's level. As I described in my plan, 2 to 12 players can play the game and there has to be at least one real human player and the rest of the 11 can be computer opponents.

## User interface

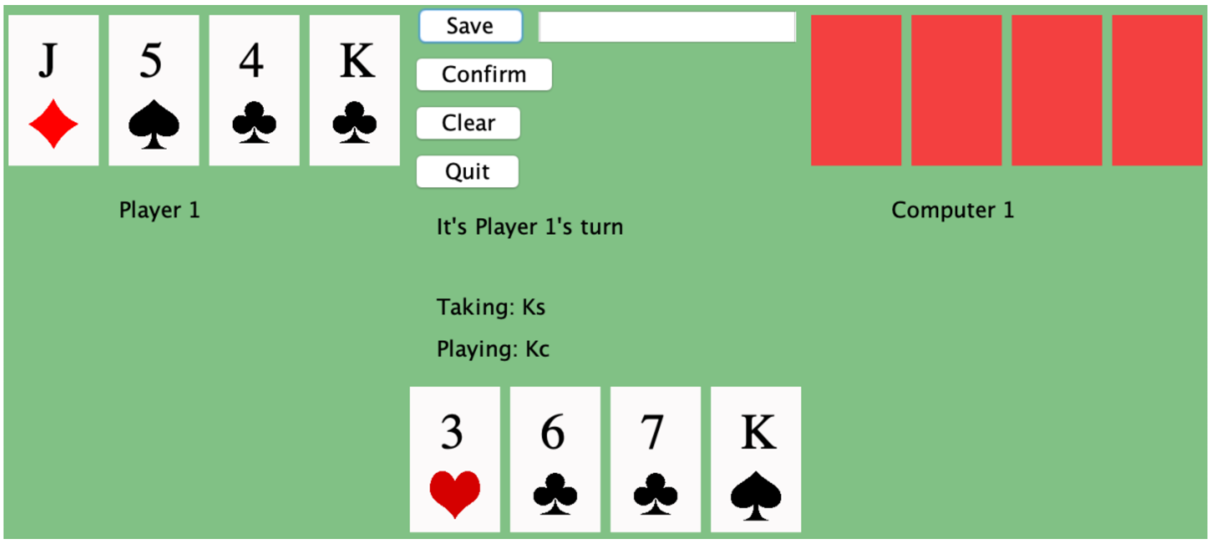
The user interface consists of three different windows: the start window, the game/play window and the end window. At the start window it is possible to enter the number of the human players and the number of the computer opponents. The human players' number needs to be given first and the program reminds the user of this if needed.

It is also possible the load a game after which it isn't possible to give the number of the players or computer opponents anymore. If inputs are false, the program reports about this to the user and tells what needs to be changed. After loading the game or entering the number of players, the user can start the game by clicking the start button. Contrary to my plan, it isn't possible to change the players' names and they just get their default names Player 1, Computer 1, etc.

When the game starts, the display is switched to the game window. The game window is similar to the window I described in my plan and is shown in picture 1 with two players. Real players can click on their cards and the table's cards and the player confirms its turn by pressing the *confirm* button. The computer opponents play their turn when clicking on the names. The player's, whose turn it is, hand cards are face up and the others are face down. The count of the hand cards displayed decreases when the deck is empty.

In addition to the buttons described in my plan, there is a *save* button and a text field to write the file's name where the game should be saved. The game is saved after entering this file name and clicking on the *save* button. There's also a *clear* button during the game to clear the cards the player wants to take. They also clear automatically when pressing the confirm button, whether the move is valid or not.

The program also reports what is happening to the center of the display. It reports which cards have been chosen from the table and from the player's hand. The program also reports what the computer opponents did on their turn and if the players' moves are valid or not.



It's Player 1's turn

Taking: Ks

Playing: Kc

Picture 1. Game with two players

After all the cards are played the program displays the end window. Here the user can see how many points each player got and who is the winner. The end window is a bit simpler than described in my plan and it doesn't show the pictures of the special cards the players got. At the end it is also possible to save the game the same way as during the game. Again, the program gives feedback whether the game was saved successfully or not.

Contrary to my plan it is not possible to start a new game during the same run of the program. The *quit* buttons during the game and at the end just end the running of the program and close the window.

## Class structure

My class structure is similar to the class structure described in my plan so the classes in my program are Card, Deck, OwnTable, Player, Computer, Game and GUI. The Card class represents the cards used in the game. Each Card has its number and suit and a method to determine its value in a player's hand.

The Player class represents the players and the Computer class the computer opponents which is a subclass of the Player class. The Player class has methods to add cards to their hands and piles and to remove cards from their hands. These methods correspond to the possible actions a player can make. For example, the *placeCard* method removes a card from the player's hand and *takeCards* adds the given cards to the player's pile. This class has also a check method to check whether it is possible to take the given cards with the player's current card and a private variable *points* to keep a track of the player's points.

The Computer class has, in addition to the aspects described above, methods to evaluate its best move. First it evaluates the all the combinations of the table cards and after this the possible combinations of each of its hand cards. Then the computer evaluates is it best to take certain cards off the table or to just place a card on table.

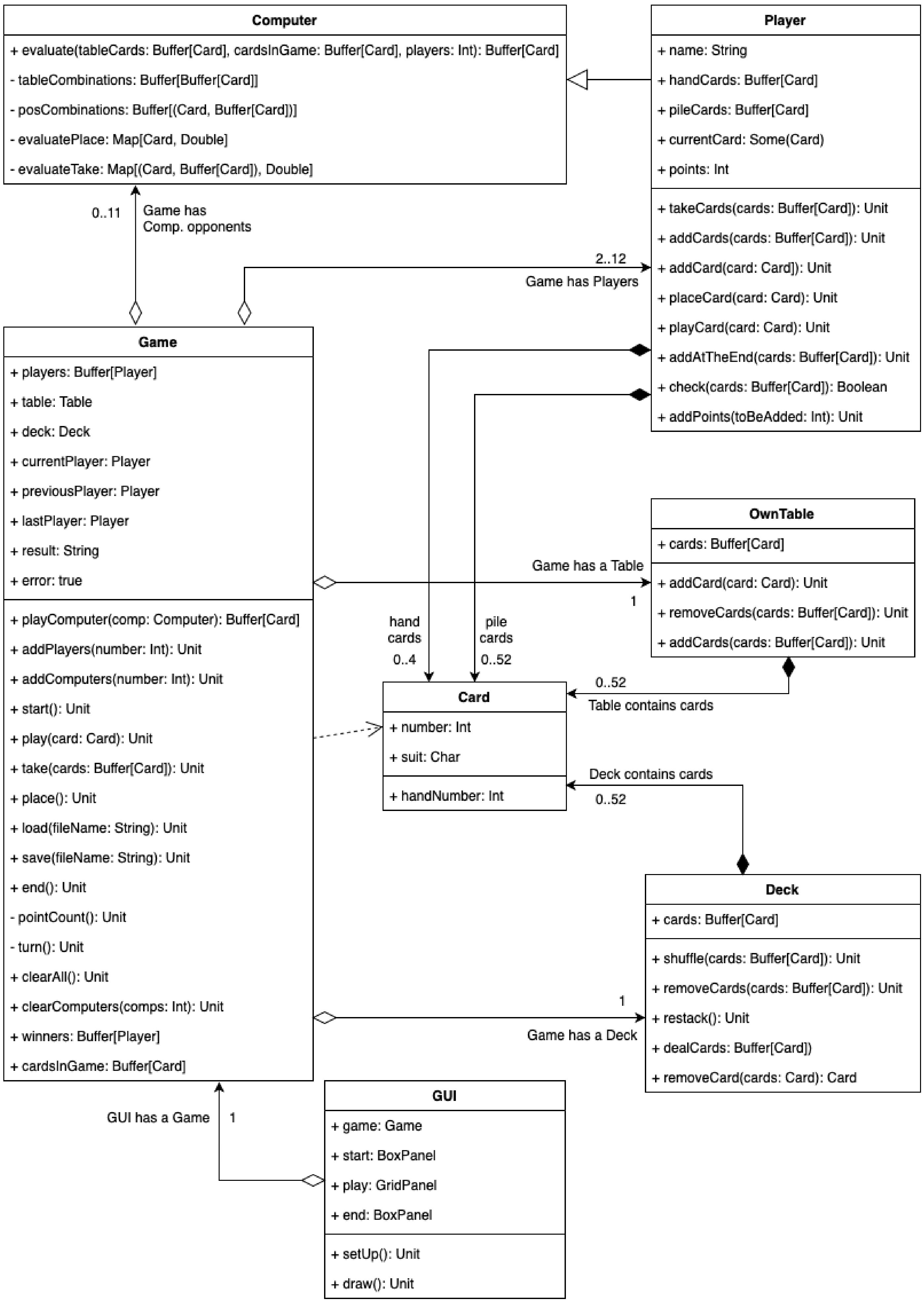
The OwnTable class represents the table of the game ja the Deck class represents the deck. The OwnTable class has similar methods to the Player class to remove and add cards. The Deck class has the method to remove cards but also methods to restack and shuffle the deck. The Deck has also a method *dealCards* which returns the next four cards of the deck and removes these from the deck.

The Game class joins these classes together. It registers the players' moves and moves the cards around the players, the table and the deck. The *place* method places the current player's current card on the table, the *take* method takes the given cards of the table with the current player's current card and the *play* method just determines the given card to be the current player's current card. The *playComputer* method calls the given computer opponent to evaluate its move and returns the cards the computer opponent wants to take. The Game class also contains the *save* and *load* methods.

The Game class has also methods required at the beginning and at the end of a game. In this class it is possible to add players, execute the start set up of a game, count the points of the players and to clear information. For example, during the start set up the deck is restacked and shuffled, cards are dealt, and the first player is determined. Methods to clear information are useful if the user wants to change its inputs. The *end* method adds the last cards left on the table to the correct player's pile, clears the table and counts the players' points. The *winner* method determines who the winner is or is it a tie.

The user interface consists of only one class GUI. It has a method *setUp* to set up the game at the start of it and a *draw* method to update the displayed game window. It has also multiple smaller methods like *updateTake* to update the text in the center of the game window. There are also helper methods to draw the pictures of the cards and the described text labels and to make the displayed window look nicer.

This class structure works well since the smaller classes Card, Deck, OwnTabel and Player were independent from each other and just get together in the Game class. This is also why the GUI only needs to call the Game's methods and not the Player's and other classes' methods. However, with this implementation most of the important methods are implemented in the Game class which makes it a bit heavy. For example, the *load* and *save* methods could be implemented in their own class which would on the other hand add dependencies between this class, the Game class and the graphic user interface.


Computer
+ evaluate(tableCards: Buffer[Card], cardsInGame: Buffer[Card], players: Int): Buffer[Card]
- tableCombinations: Buffer[Buffer[Card]]
- posCombinations: Buffer[(Card, Buffer[Card])]
- evaluatePlace: Map[Card, Double]
- evaluateTake: Map[(Card, Buffer[Card]), Double]
Player
+ name: String
+ handCards: Buffer[Card]
+ pileCards: Buffer[Card]
+ currentCard: Some(Card)
+ points: Int
+ takeCards(cards: Buffer[Card]): Unit
+ addCards(cards: Buffer[Card]): Unit
+ addCard(card: Card]): Unit
+ placeCard(card: Card): Unit
+ playCard(card: Card): Unit
+ addAtTheEnd(cards: Buffer[Card]): Unit
+ check(cards: Buffer[Card]): Boolean
+ addPoints(toBeAdded: Int): Unit
0..11
Game has
Comp. opponents
2..12
Game has Players
Game
+ players: Buffer[Player]
+ table: Table
+ deck: Deck
+ currentPlayer: Player
+ previousPlayer: Player
+ lastPlayer: Player
+ result: String
+ error: true
+ playComputer(comp: Computer): Buffer[Card]
+ addPlayers(number: Int): Unit
+ addComputers(number: Int): Unit
+ start(): Unit
+ play(card: Card): Unit
+ take(cards: Buffer[Card]): Unit
+ place(): Unit
+ load(fileName: String): Unit
+ save(fileName: String): Unit
+ end(): Unit
- pointCount(): Unit
- turn(): Unit
+ clearAll(): Unit
+ clearComputers(comps: Int): Unit
+ winners: Buffer[Player]
+ cardsInGame: Buffer[Card]
OwnTable
+ cards: Buffer[Card]
+ addCard(card: Card): Unit
+ removeCards(cards: Buffer[Card]): Unit
+ addCards(cards: Buffer[Card]): Unit
Game has a Table
1
hand
cards
0..4
pile
cards
0..52
0..52
Table contains cards
Card
+ number: Int
+ suit: Char
+ handNumber: Int
Deck contains cards
0..52
Deck
+ cards: Buffer[Card]
+ shuffle(cards: Buffer[Card]): Unit
+ removeCards(cards: Buffer[Card]): Unit
+ restack(): Unit
+ dealCards: Buffer[Card])
+ removeCard(cards: Card): Card
1
Game has a Deck
GUI has a Game
1
GUI
+ game: Game
+ start: BoxPanel
+ play: GridPanel
+ end: BoxPanel
+ setUp(): Unit
+ draw(): Unit
Graph 1. UML-Cassino

## Algorithms

The algorithms executed are similar to the ones described in my plan.

### Card checking algorithm

The card checking algorithm is executed in the *check* method of the Player class. First the program checks if any of the wanted cards are bigger than the played one. If this is the case the turn in invalid. Then the program checks if any of the wanted cards are as big as the played one. These cards can't be used for any sums, so they are set aside. If only one card is left after these phases the turn is invalid. Otherwise, the program checks for sums:

1) We add a card to the previous sum/first card of the card sequence. If the sum is equal to the played card, we move onto step 2. If the sum is smaller than the card played, we move onto step 3. If the sum is bigger, we move onto step 4.
2) We perform the step 1 to the rest of the cards. If no cards are left, the turn is valid.
3) We continue with step 1 if there are other cards left. If not, the turn is invalid.
4) We change the cards' order and perform step 1.
   We change the cards' order by keeping the first one in its place but taking the second one to the back of the sequence when the third one comes the second one, the fourth one comes the third one and so on. Now if the second one is the one that was the second one in the beginning of this algorithm, the turn is invalid.

### Computer Opponent

The computer opponent analyzes which cards it should take or which card it should place. Contrary to my plan the computer opponent evaluates the taking and the placing of cards together and doesn't dismiss the possibility of placing a card even thought it could take cards of the table. This is because when taking cards of the table it is possible to give an easy chance for the next player to get a sweep when placing a card on the table could be a better option.

First the computer opponent evaluates all the possible combinations of the cards on the table and then which card in its hand could take which combinations. If the computer opponent does match its handcards to possible combinations, it evaluates each of these combinations based on the played card and on the cards it takes. The computer opponent then considers these points:

1) How many cards are in the combination?
2) How many spades are in the combination?
3) How many special cards are in the combination?
4) Is it possible for the next player to get a sweep with the cards left on the table?

Here the combinations also include the card that is played since it is also important to evaluate that one too.

Then the computer opponent evaluates the best card to place on the table by considering these points:

5) Is it a special card?

6) Is it a spade?
7) Could it make combinations for special cards?
8) Is it a small or a big card?

The computer opponent evaluates a corresponding number to these combinations/cards by which it then decides which move to make. The bigger the number the better the move. Here are the equations to calculate the actual numeric values of the moves. The numbers correspond to the previous listed points.

1) count of the cards * 1/52 = count of the cards * 0,019
2) count of the cards * 1/13 * 2 = count of the cards * 0,154
3) point count of the special cards * 1
4) possibility to get a sweep * (-1)
   - possibility to get a sweep = cards that could get a sweep and are still in the play/all the cards that are still in the play * hand cards count
5) points from this card * (-1)
6) -1/13 * 2 = -0,154
7) possibility to get with a special card * (-1)
   - possibility to get with a special card = point count of the possible special cards/all the cards that are still in the * hand cards count
8) card number * (-0,0001)

The equations try to return values that correspond to points a player can get during a game (except equation 8) and the negative values represent points the other players could get instead. The equations 4 and 7 could be improved to be more specifically calculated so that they would work better with the other equations even though the computer opponent works well already.

In the equations 4 and 7 the points of a sweep or the special cards are first divided by the count of all the cards still in the game and then multiplied by the number of the next player's hand cards because the next player has to have the correct cards. In the actual code the equation is multiplied by the number of the computer opponent's hand cards to make the code simpler. The equation 8 just makes the last difference when deciding between cards to place that have otherwise the same values.

## Data structures

My program uses mostly mutable buffers especially when describing the cards of the players, table and deck since the cards rotate a lot during the game. The Game class uses a mutable buffer for its players because it is useful to know their indices for example when saving and loading a game since the file format has player blocks that include the players' indices. It is also useful to know their indices and keep them in order to clear computer opponents and to count how many of the players are human players and how many computer opponents. Clearing in addition to adding players explains why it is useful to use a mutable data structure. Mutable indexed buffers are also used in the GUI class when matching cards and players to their Swing components.

Maps are used in the Computer class in the *evaluateTake* and *evaluatePlace* methods to match possible combinations and cards to their numerical values. Vectors are used in multiple methods when the specified collection doesn't need to be modified which decreases accidental errors for them to be modified.

## Files and their format

The file format the program uses is displayed below. It is a text file and consists of the header CASSINO and the blocks PLR, CMP, TBL, TRN and END. The PLR and CMP blocks represent players and computer opponents and the number after these three letters tells which player or computer opponent it is. After the letters PLR and the player number, there should be a number that tells the player's name's size which is followed by the name. Then comes the strings describing the player's hand and pile cards that are separated by a colon. The TBL block tells what cards are on the table, the TRN block tells whose turn it is, and the END block ends the file.

```
CASSINO
PLR18Player 110c7sJc3s:8h8s
CMP1Kh8c4dAs:9s5d4s
CMP29d5s2h6c:
TBL6dQs6s
TRNcmp2
END
```

A moore specific explanation of PLR18Player 110c7sJc3s:8h8s :
Player number 1 with the name Player 1 that is 8 Chars long. The cards in the player's hand are ten of clubs (10c), seven of spades (7s), jack of clubs (Jc) and three of spades (3s). The pile cards are eight or hears (8s) and eight of spades (8s). The hand and pile cards are separated by a colon.

Each block needs to be on their own line for the program to read them and the file needs to have an END block for the program to function. However, the file can have line breaks and it checks if other blocks or the CASSINO header are missing. The file has to have at least two players (PLR or CMP) of which at least one needs to be a human player (PLR). The program also checks if the TRN block has the right format so if it reads a player corresponding to the PLR and CMP blocks given before it. If there's no TBL block the table just doesn't have any cards.

The PLR and CMP blocks should be in order and start from the number one so that they match the idea and other factors of the game. It would be weird if there would be players 2 and 3 but no player number 1. If two blocks start with the same four characters, for example CMP1, then only the last one is registered. The program also checks if both numbers, the player number and the size of its name are given. However, the program doesn't check if a correct number for the player's name's size is given.

The file doesn’t need the colon in the PLR or CMP block. This just means all the cards described are the player’s hand cards. However, there can’t be more than four hand cards which the program checks. It also checks if there are misspelled cards. There can’t be written cards with numbers such as 32 or with suits such as “r”. Acceptable numbers for cards are the numbers from 2 to 10 and the letters “A”, “J”, “Q” and “K”. Acceptable letters for suits are “s”, “d”, “h” and “c”. The program also checks that there are no same cards, and that the cards’ information is written in the right order which means that the number should be written first. The program does not check if the pile cards have valid combinations since this would be quite complicated.

## Testing

The program was tested with three different test classes, the TestGame, TestFile and UnitTests classes. TestGame tests the players’ moves during a game and other methods of the Game class. TestFile tests the load and save methods and UnitTests executes some unit tests concerning similar actions than in the TestGame class.

Tests were added to the TestGame class during the making of the program. First methods concerning the player moves like *place*, *take* and *play* were tested. After these, other methods implemented to the Game class were tested like the *end*, *start* and player adding methods. Then the *check* method of the Player class was tested with multiple test cases and improved throughout the development of the project as some bugs were detected. Finally, after the computer opponent was implemented its methods *evaluate*, *evaluatePlace* and *evaluateTake* were tested in the TestGame class.

The *load* and *save* methods were tested in the TestFile class during their development and actually created in this class. After they were working properly the methods were moved to the Game class after which they were tested in the TestFile class just by calling them from the Game class.

The UnitTests class tests the *check*, *evaluate*, *evaluatePlace* and *take* methods. It tests these methods because they were some of the most complex methods and they use algorithms I personally created. They were also the methods I had most difficulty with. These tests were implemented at the last stages of the project to make absolutely sure that they were working. During every test, I developed the method until the program passed the respective test and the method was working as hoped. The testing was continuous and it proceeded according to my plan.

## Bugs and deficiencies

A distinct weakness in my program is that the table can have at most 24 cards according to the graphic user interface since the panel tablePanel is a 6 x 4 grid panel. If more cards are added the card pics don’t look as they should and they become narrower. It isn’t likely that a scenario where this many cards would be on the table would happen but theoretically there is the possibility. I decided to make the decision of working with constant size displays because working with Scala Swing was new for me and I wanted to get the game otherwise working without major disruptions. Also, the program crashes if a text file is given to it without an END block.

## Strengths and weaknesses

I'm glad that I decided to implement the computer opponent as the class Computer being a subclass of the Player class. This allows the Computer class and its methods to concentrate on its AI and tactics. I also think it was a good idea to divide the different tactics into their own methods, so it is possible to comment/erase them to see with which tactics the computer opponent is at its best. It's also easy to develop them forward and implement new ones.

I'm also happy with the other features of my class structure in the sense that most of the classes are independent from each other. This gives the possibility to add methods to them and edit their methods without major disruptions. If developing the smaller classes Deck, OwnTable, Deck or Player only the Game and GUI classes need to be modified to take into account these changes.

However, my classes have pretty wide interfaces and their variables' values can be changed easily. Most variables are defined to be *var* variables and I also use mutable buffers as data structures. Even though this is useful since cards rotate a lot during the game, it does make my program vulnerable for mistakes in the code since the values can be easily changed. The code could be improved with more *private* statements and additional methods and by changing some of the *var* variables into *val* variables.

Also, the GUI class in my program could be improved. For example, there's some repetitiveness. The *setUp* and *draw* methods have some lines that are exactly the same, but the main difference is that the *setUp* method creates the Swing components, and the *draw* method just modifies them. Also, the center panel in the GUI is modified by getting its components with their indices in the *draw* and the smaller update methods which makes the code prone to errors when adding components to the center. The windows' sizes are also fixed, which is a problem for the table cards and if the user wants to write a long name for a file in the game window since this text field isn't very wide.

Finally, the load method could also be more readable. It is useful that it checks for multiple different errors in the loaded text file, but this makes the actual method long and complicated. It has multiple match – case structures and a lot of if – if else branches.

## Process and schedule

The start of the development of my project went according to plan except the implementation of the Card, Table, Deck and Player classes took a lot less time than I expected. Because of this I got the main methods of the Game class implemented also during the first two weeks and was then two weeks ahead of schedule.

Because I was so ahead of my schedule, I decided to start the development of the graphic user interface and test if it wouldn't be too much work for me. So, I developed the GUI during the next two weeks and decided to stick with it and turn down the possibility of making a text user interface. After these two weeks, according to my plan, it was time to implement the *load* and *save* methods and the computer opponent. I succeeded in this and got the GUI working with these additions.

Then at the start of the last two weeks according to my plan I would have only started the implementation of my GUI but luckily, I was almost done with it. The implementation of the graphic user interface took a lot more time than I expected, and I probably wouldn't have been able to get it working in the last two weeks. Since I had time and I was ahead of schedule I improved the *load* method, the *save* method and the computer opponent. I also implemented the unit tests in the UnitTests class. Also a few other improvements were made to the program. For example, I separated the commands from the original *PlayTurn* method to their own separate methods.

As I said, the biggest surprise was how much time it took to get the GUI working as wanted. I didn't have a lot of experience with the Scala Swing library, and it took some learning and getting used to. So, I learned about the Scala Swing library and that it takes a lot of time to get used to a new library especially when there isn't much information about it and how to use its features.

## Final evaluation

In summary, I'm happy with my program even though it could use some improvements. The GUI could be developed further with some more time to meet the requirement of 52 cards being placed on table. The GUI class could also be clearer especially concerning its reactions and maybe even divided into some subclasses. The user interface could be prettier and the possibilities of changing the players' names and starting a new game could be added. These improvements shouldn't be too hard to make except the adding of more cards to the table. This could need some excessive changes to the existing code especially if the sizes of the windows would be changed from fixed to editable.

The class structure of my project is clear and modifying the interfaces of the classes smaller shouldn't be too hard. Some of the buffers could be switched to immutable data structures in some of the methods but keeping them as buffers decreases the need to change the back to buffers again. I'm glad that the *load* method takes so many errors in the text file into account, but it could be made even better and clearer. The computer opponent could also be improved but luckily, it's easy to continue its development.

Some improvements would have been wise to make at the beginning of the development process of my program. For example, the interfaces of the classes could have been more thought out but overall, I'm pleased with the process and the results of my program. I'm also excited that I learned more about the Scala Swing library.

## References

Scala Standard Library API (2.12.0): https://www.scala-lang.org/api/2.12.0/scala-swing/scala/swing

Java 2D API: https://docs.oracle.com/javase/7/docs/api/java/awt/Graphics.html

GUI Programming: https://otfried.org/scala/gui.html

## Appendixes

Source code: os2.cassino

Example file: exampleFile.txt

Hearts pic: heart.png
Clubs pic: clubs.png
Diamonds pic: diamond.png
Spades pic: spades.png

User Interface, Player 1's turn: Player1.png
User interface, After Computer 2's move: Computer2.png
User Interface, End of game: EndOfGame.png