CS12320: Main individual Assignment

Panagiotis Karapas (pak27@aber.ac.uk)

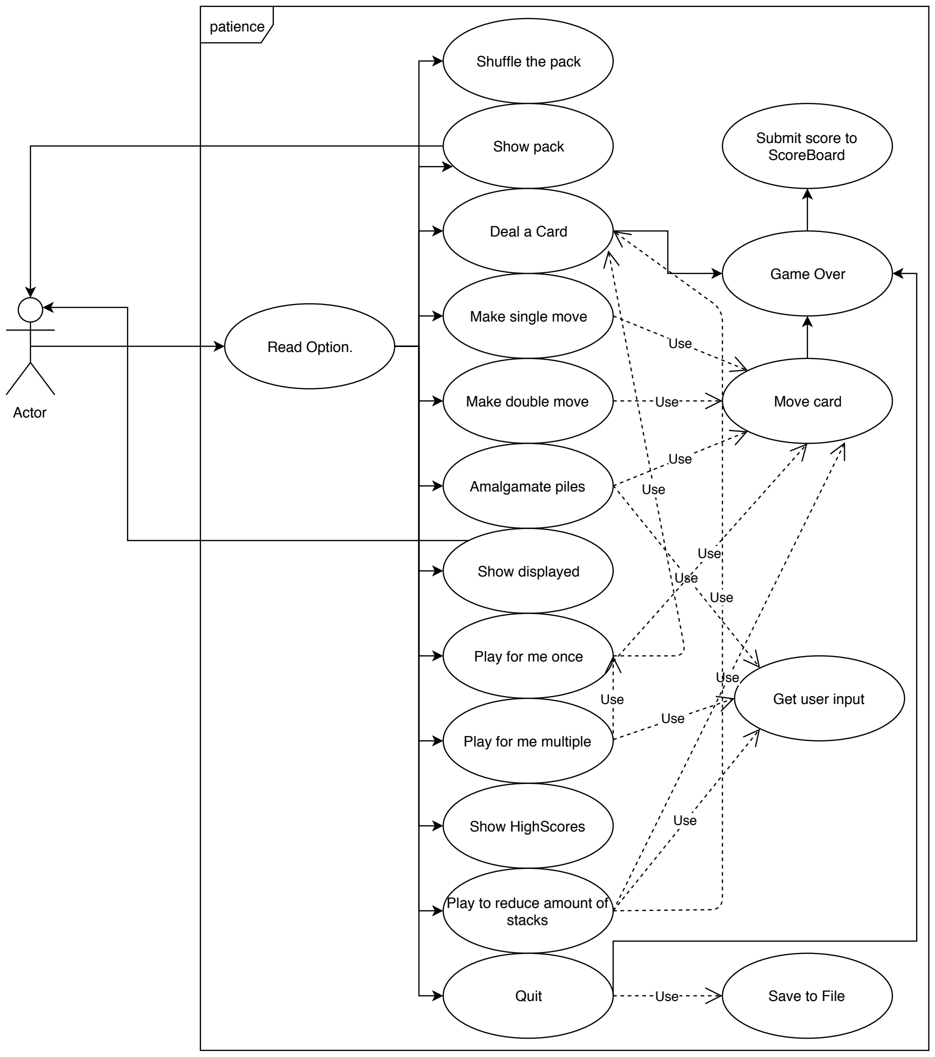
May 10, 2018

Contents

1 Introduction

This is the report part of my main individual assignment for CS12320. One of my main goals when designing the structure of this project was that I wanted the code to be reusable for other card games. The *util*, *scorekeeping* and *playingCards* packages when combined provide the programmer with a lot of useful resources for creating any card game that uses the standard deck of playing cards, with support for coloured output, saving and reading score-boards and more.

1.1 UML use-case diagram



2 Design

In this section I will explain the role of each class and it's connections with other classes. The section is divided by package and then class names (except where only one class exists in the package). A class diagram is included (section 2.8).

2.1 uk.ac.aber.dcs.cs12320.cards.gui.javafx.CardTable

The *CardTable* class provides all of the methods needed for creating and displaying the GUI. The main methods to note are:

- *drawImage* that takes one stack of cards and returns an image containing all of them in a way that only the top card can be fully seen, and for the rest of them only ***PERCENT_SHOWN*** percent can be seen (current ***PERCENT_SHOWN*** value is 0.15 (so 15%) but any value would work).
- *cardDisplay* that creates a display and displays all the face-up cards in the board provided. A *HBox* and a *ScrollPane* are used to create a scrollable window, allowing all the cards to be displayed.

2.2 pak27.playingCards

The *playingCards* package contains classes that work with each other to create a library that can be used to create most card games. It should be noted that it requires that *pak27.util* (section 2.4) is also included in the project.

2.2.1 Card

The *Card* class represents a single card. Cards can then be arranged in a *Deck* (section 2.2.3) or a *Stack* (section 2.2.2).The *image* variable is transient because I did not want Gson to save the full paths of the images, since then moving the files would break the game. It's worth mentioning that the *Card* class expects all the card images to be in the correct folder, and that they are named correctly.

Note: The *copy* method has a boolean argument that if set to false means that the copy's image will not be loaded, which (even though it will not save memory space) will speed the copying process, but also means that the card can not be displayed.

2.2.2 Stack

The *Stack* class implements a FILO (First In Last Out) system using an *ArrayList*. This would be better implemented using *java.util.Stack* but I did not know that it existed, until most of my code was already done, and I do not have enough time to change it. This class is used as the super class for *Deck* (section 2.2.3).

2.2.3 Deck

The *Deck* class inherits from the *Stack* class (section 2.2.2). It also provides methods for shuffling, reading the cards from a file but also generating a file if it's missing.

On Shuffling: Before knowing about the *Colections.shuffle* method, I implemented my own, and since shuffling is a really important for every card game,I ended up, without knowing it, implementing the same algorithm that is implemented in *Colections.shuffle* (Durstenfeld's version of Fisher-Yates shuffle). In the end I chose *Colections.shuffle* implementation because it is most probably more efficient than my own. My *shuffle* method was:

```
  Random r = new Random();
public void shuffle() {//I chose this algorithm since its quite efficient , and does not require extra space.
    ArrayList<Card> cards = getCards();//makes it easier by not having to use getters and setters
    for (int i = cards.size(); i > 0; i--) {
        int index = r.nextInt(i);//get a random index between 0 (inclusive) and i (exclusive)
        //swap the element in the "index"th place with the last un-shuffled element
        Card selected = cards.get(index);
        cards.set(index, cards.get(i - 1));
        cards.set(i - 1, selected);
    }
    setCards(cards);
}

```

2.2.4 Colour

The *Colour* enumeration includes just 2 values *RED* and *BLACK*. The purpose of this is to know which cards are black and which are red, which can then be used for using colours on the console output.

2.2.5 Suit

The *Suit* enumeration includes 4 values, one for each possible suit. Each suit also includes a *Colour* (section 2.2.4) and a *char* used as an identifier.

2.2.6 Value

The *Value* enumeration includes 14 values (Ace to King plus Joker). Each value also includes a *char* and an *int* as an identifier and numerical value respectively. *Note*: The Joker has 'O' as his char since 'J' is used by the Jack.

2.3 pak27.scoreKeeping

The *scoreKeeping* package contains classes that work with each other to create a library that can be used to create and save a score-board. The scoreboard can be of any size (within memory limitations) and it also provides support for using colours. It should be noted that it requires that *pak27.util* (section 2.4) is also included in the project.

2.3.1 Score

The *Score* class represents a single score, which includes a name and an integer score. It also includes two optional values:

- *wasAssisted* should be true only if the computer helped the player in any way (hints/autoplay).
- *colour* which defaults to *BLACK* and defines the colour the name should be displayed in.

2.3.2 ScoreBoard

The *Scoreboard* class's main role is to encapsulate a Score (section 2.3.1) array. The size of the array is defined on instantiation of the class. The default value is 10. The scoreboard also has a non-optional boolean variable *isBiggerBetter* which defines the way the scores should be arranged. Assisted scores are always worse than their non-assisted counterparts. The scoring of the patience game specifically is discussed in section 2.6.

2.4 pak27.util

The *util* package provides utility functionality that is used throughout the program. The *Util* class (section 2.4.1) combined with the *ANSIColour* enumeration (section 2.4.2) can produced text in any of the 7 colours defined in *ANSIColour*.

2.4.1 Util

The *Util* class is comprised only by static methods. It provides functionality that is used throughout the rest of the program. A last minute addition that was added to *Util* was the *readInteger* method. This was added because for an unknown reason when the game was run from the command line the *Scanner* delimiter would not work.

2.4.2 ANSIColour

The *ANSIColour* enumeration includes 8 values (7 colours and 1 reset). Each value also includes a *String* that is the ANSI code needed to be used.

2.5 pak27.autoplay.Autoplay

The *AutoPlay* class is part of the things that were added in the game by me. It is a brute-force solver of the game. It has only on public method *solve* that takes the provided board, and tries to reduce it to the provided number of stacks. If a solution is found, then a 2 dimensional *int* array is returned containing the steps it took to get to that point,if a solution is not found, or something is wrong with the arguments, null is returned. I chose to return the steps and not just the solved board, because I wanted it to be able to be used in cases like stepping through the solution. I know this is not an efficient way of solving the game, but it works (even if it's slow) and for my purposes is good enough.

2.6 pak27.Game

The *Game* class is the main class, it contains the main method but is also responsible for running the game loop. It inherits from the *Application* class, allowing us to use the *javafx* library for our GUI. Some things to note about the *Game* class are:

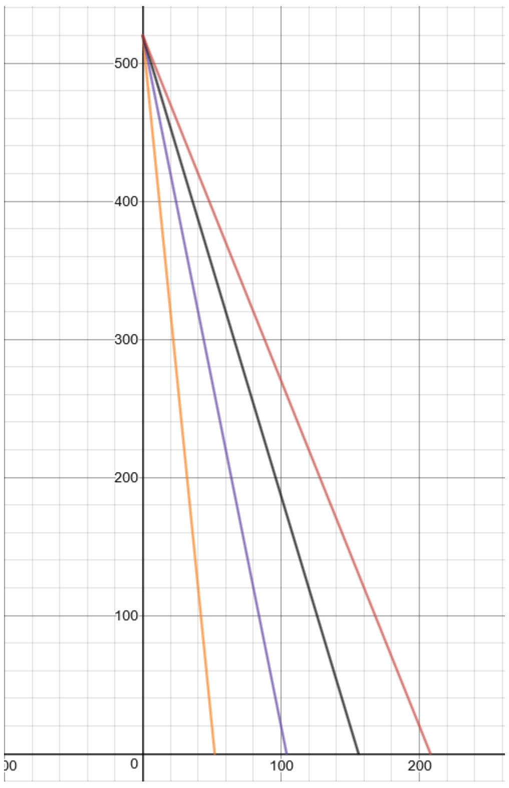
- The use of the boolean *update* allows to save a lot of resources since updating the GUI is not always necessary.
- The scoring system was completely changed, since it had to accommodate for the usage of more than a single deck. In the *Game* class their are 2 scoring functions, one if the game ends by the player having no remaining moves, and one if the player decides to exit by pressing 'Q'.

If the game ends properly.

If the game ends properly then the following equation is used to calculate the score:

$$\left\lceil \frac{\lfloor 2 * (520 - 10 * (\frac{s}{n})) \rfloor}{2} \right\rceil$$

Or in code: Math.round(520 - 10*(s/n)) Where s is the amount of stacks left minus 1 and n is the number of decks you started with. Which means that if you use more decks, then you need a lesser amount of stacks left to get a good score. The relation can be seen below:



Where the yellow, purple, black and red lines represent starting with 1, 2, 3 and 4 decks respectively, with the x axis being the amount of stacks remaining at the end and the y axis being the score.

520 is the perfect score for any amount of starting decks.

If the player exits.

If the player exits before the game properly ends (no remaining moves available), then the score is calculated the same equation as before with the difference that n is now the number of decks you started with, plus the number of cards remaining on the starting pile. This results you in getting the worst possible score you can get at this time.

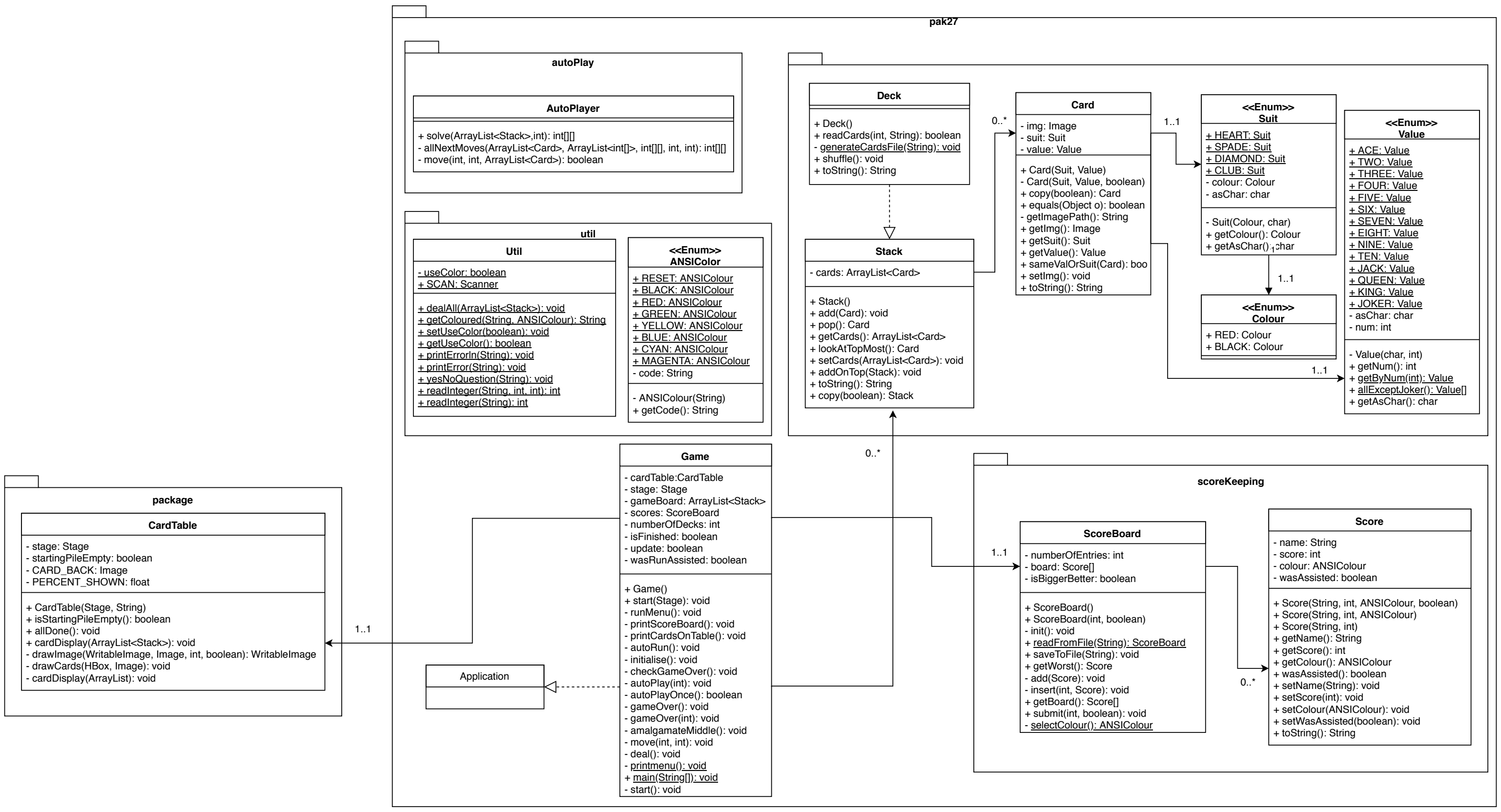
2.6.1 Amalgamate pseudo code

1. Ask the user which piles they want to move (make sure the inputs are integer).
2. Check if the piles exist (so that the indexes are between 0 and the amount of stacks on the table).
3. If step 1 and 2 are successful then move all the cards from the 'from' pile to the 'to' pile, then remove the 'from' pile.
4. Make sure that the GUI is updated.

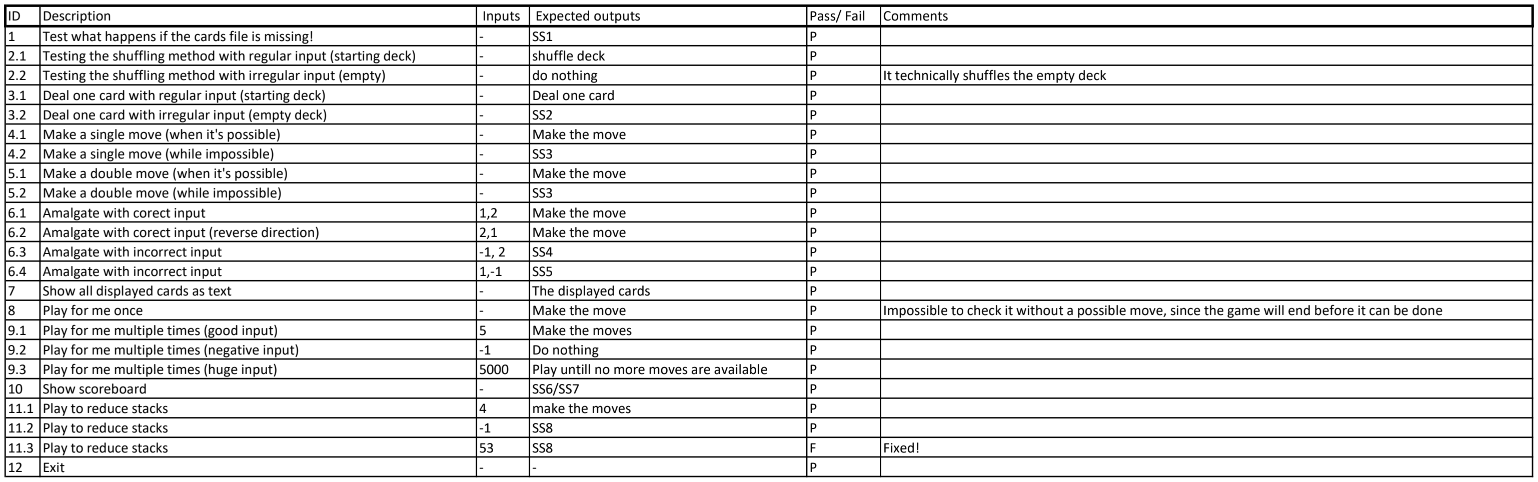
2.7 cards

The cards folder contains all the images used throughout the game. The original images were replaced.

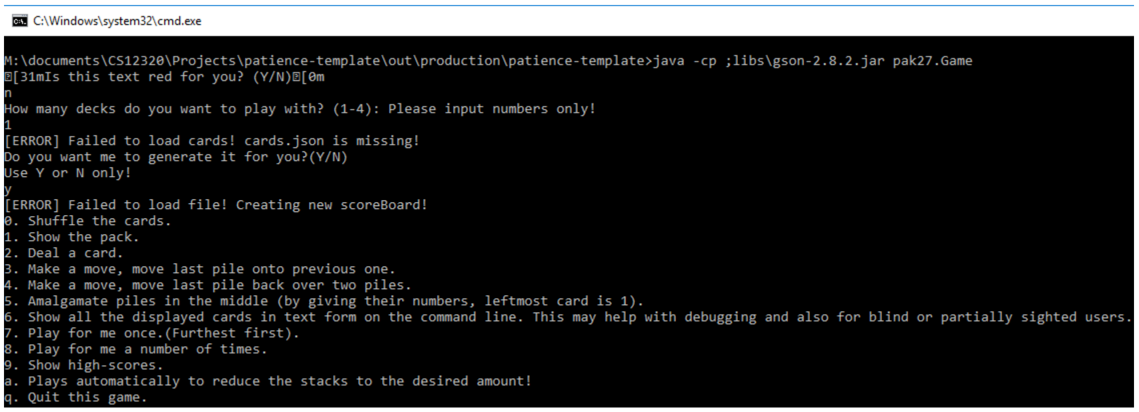
2.8 UML class diagram



3 Testing

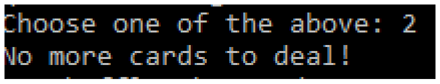


ID	Description	Inputs	Expected outputs	Pass/ Fail	Comments
1	Test what happens if the cards file is missing!	-	SS1	P	
2.1	Testing the shuffling method with regular input (starting deck)	-	shuffle deck	P	
2.2	Testing the shuffling method with irregular input (empty)	-	do nothing	P	It technically shuffles the empty deck
3.1	Deal one card with regular input (starting deck)	-	Deal one card	P	
3.2	Deal one card with irregular input (empty deck)	-	SS2	P	
4.1	Make a single move (when it's possible)	-	Make the move	P	
4.2	Make a single move (while impossible)	-	SS3	P	
5.1	Make a double move (when it's possible)	-	Make the move	P	
5.2	Make a double move (while impossible)	-	SS3	P	
6.1	Amalgate with corect input	1,2	Make the move	P	
6.2	Amalgate with corect input (reverse direction)	2,1	Make the move	P	
6.3	Amalgate with incorrect input	-1, 2	SS4	P	
6.4	Amalgate with incorrect input	1,-1	SS5	P	
7	Show all displayed cards as text	-	The displayed cards	P	
8	Play for me once	-	Make the move	P	Impossible to check it without a possible move, since the game will end before it can be done
9.1	Play for me multiple times (good input)	5	Make the moves	P	
9.2	Play for me multiple times (negative input)	-1	Do nothing	P	
9.3	Play for me multiple times (huge input)	5000	Play untill no more moves are available	P	
10	Show scoreboard	-	SS6/SS7	P	
11.1	Play to reduce stacks	4	make the moves	P	
11.2	Play to reduce stacks	-1	SS8	P	
11.3	Play to reduce stacks	53	SS8	F	Fixed!
12	Exit	-	-	P	



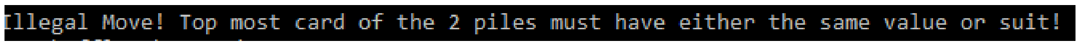
```
M:\documents\CS12320\Projects\patience-template\out\production\patience-template>java -cp ;libs\gson-2.8.2.jar pak27.Game
[31mIs this text red for you? (Y/N)[0m
n
How many decks do you want to play with? (1-4): Please input numbers only!
1
[ERROR] Failed to load cards! cards.json is missing!
Do you want me to generate it for you?(Y/N)
Use Y or N only!
y
[ERROR] Failed to load file! Creating new scoreBoard!
0. Shuffle the cards.
1. Show the pack.
2. Deal a card.
3. Make a move, move last pile onto previous one.
4. Make a move, move last pile back over two piles.
5. Amalgamate piles in the middle (by giving their numbers, leftmost card is 1).
6. Show all the displayed cards in text form on the command line. This may help with debugging and also for blind or partially sighted users.
7. Play for me once.(Furthest first).
8. Play for me a number of times.
9. Show high-scores.
a. Plays automatically to reduce the stacks to the desired amount!
q. Quit this game.
```

Figure 1: SS1



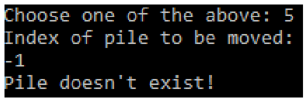
```
Choose one of the above: 2
No more cards to deal!
```

Figure 2: SS2



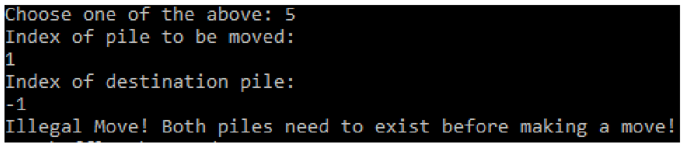
```
Illegal Move! Top most card of the 2 piles must have either the same value or suit!
```

Figure 3: SS3



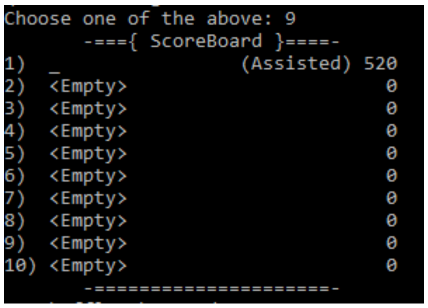
```
Choose one of the above: 5
Index of pile to be moved:
-1
Pile doesn't exist!
```

Figure 4: SS4



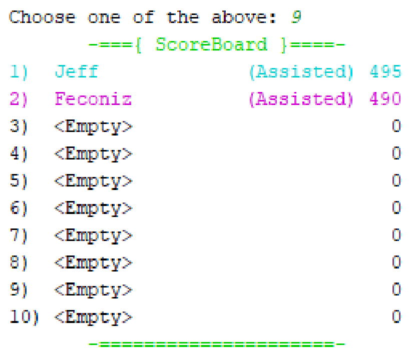
```
Choose one of the above: 5
Index of pile to be moved:
1
Index of destination pile:
-1
Illegal Move! Both piles need to exist before making a move!
```

Figure 5: SS5



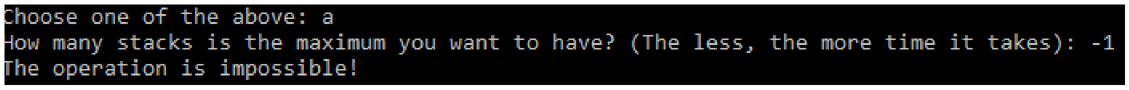
```
Choose one of the above: 9
       -===={ ScoreBoard }====-
1)  _                   (Assisted) 520
2)  <Empty>                          0
3)  <Empty>                          0
4)  <Empty>                          0
5)  <Empty>                          0
6)  <Empty>                          0
7)  <Empty>                          0
8)  <Empty>                          0
9)  <Empty>                          0
10) <Empty>                          0
       -======================-
```

Figure 6: SS6 (where colour is not supported)



```
Choose one of the above: 9
       -===={ ScoreBoard }====-
1)  Jeff                (Assisted) 495
2)  Feconiz             (Assisted) 490
3)  <Empty>                          0
4)  <Empty>                          0
5)  <Empty>                          0
6)  <Empty>                          0
7)  <Empty>                          0
8)  <Empty>                          0
9)  <Empty>                          0
10) <Empty>                          0
       -======================-
```

Figure 7: SS7 (where colour is supported)



```
Choose one of the above: a
How many stacks is the maximum you want to have? (The less, the more time it takes): -1
The operation is impossible!
```

Figure 8: SS8

4 Evaluation

All in all, I think that I managed to fulfil all of the required functionality. Other than what was required the following functionality was added (for flair):

- The ability to have a score-board where each name has a colour selected by the player.
- The ability to have coloured output.
- Saving to Json files using Gson (version 2.8.2).
- The *AutoPlayer* class that reduces the amount of stacks to a desired amount.
- In the GUI not only the top card on each pile is shown, but a small percentage of each of the previous ones can also be seen.
- The ability to work even if both the scores and Cards files are missing, by having the program generate new ones.

Of course my work is not perfect, even though I have eliminated most bugs and most, if not all, user inputs are checked, there are still some things that I would change if I have had more time, these are:

- I would probably move the *getColoured* method to *ANSIColour*. (Even though this would not be difficult to do now, I would have to completely remake my class diagram, since my save file got lost)
- Put a background to the GUI (I tried but it was not cooperating with the *ScrollPane*).
- Make it so the game can be fully played using just the GUI by adding buttons.
- Create a new better *AutoPlayer* that would not try to brute-force the solution, but use some other faster technique to achieve the goal. Or at least not brute-force it recursively.

- I would make it so two instances of the same card (same *Suit* and *Value*) would not have to load the image in memory twice (or more). Maybe using an *Images* enumeration.

4.1 What I learned

Even though I have used Java before, this was a good chance to get introduced to a variety of things that I had not used before. The main thing that I learned was how to use *javaFx* (even at a basic level).

4.2 Final thoughts

In conclusion I think the grade I deserve is 75% (± 5%) since I, in my opinion, satisfactorily covered the requirements of the assignment.