

Individual Assignment: Patience is a Virtue

Michał Wojciech Goly [mwg2]

1st May 2015

Contents

1	Introduction	2
1.1	Project description	2
1.2	Game controls	2
2	Requirements Analysis	3
2.1	UML Use Case diagram	3
3	Design	4
3.1	UML Class diagram	4
3.2	Description and relationships of each Class	5
4	Testing	8
4.1	Test tables	8
4.2	Screenshots	10
5	Evaluation	21
5.1	Completely graphical user interface	22
5.2	Initial implementation	22
5.3	Towards the working application	22
5.4	Summary	23

1 Introduction

1.1 Project description

Patience is a simple card game for one player, in which the objective is to end up with one pile of cards on the table. There are 52 playing cards at the start of the game, all facing downwards in a pack. Player can then deal a card which will be removed from the deck and put on the table facing upwards. By continuing to do so, there could potentially be 52 cards on the table all facing upwards, unless a different move was made. Apart from dealing cards there are two additional valid moves in the game. Cards can be joined together if they have the same suit or value and if:

- They are next to each other
- There are two other cards between them

When joined, the card further to the right will be placed on top of the other, regardless of the order in which they have been selected. Each move is worth 10 points in the game. Therefore because there are 51 available moves in total, the highest possible score is 510 points.

1.2 Game controls

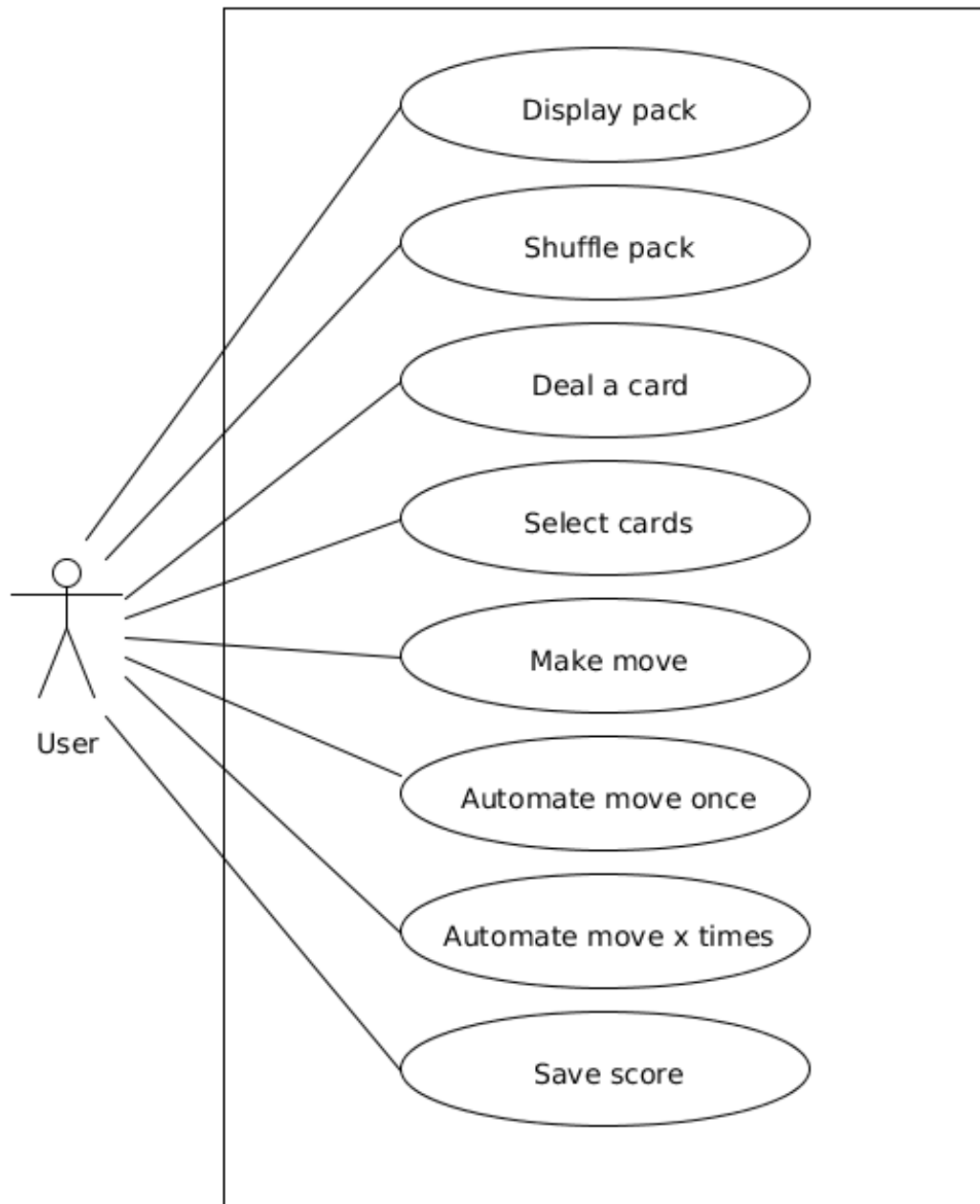
Because the user interface is fully graphical, to play the game user can simply click on the cards. For example if the player wants to deal a card, he should click on the pack and a move will be made. Similarly, to select a card user has to click on it. In order to indicate which card was selected, a blue border will be painted around it.

Additional options are available within the button panel at the bottom of the window. Player can display contents of the pack, as well as shuffle it. Second option is only permitted once and should typically be selected at the start of the game. Pack is not randomized by default due to the requirements specification of the assignment. Last two options allow an automation of the gameplay. Player can either make use of the 'Play for me once' option which as expected will make one valid move in the game, if there is one, or specify the amount of moves to be made by clicking the 'Play for me x times' button.

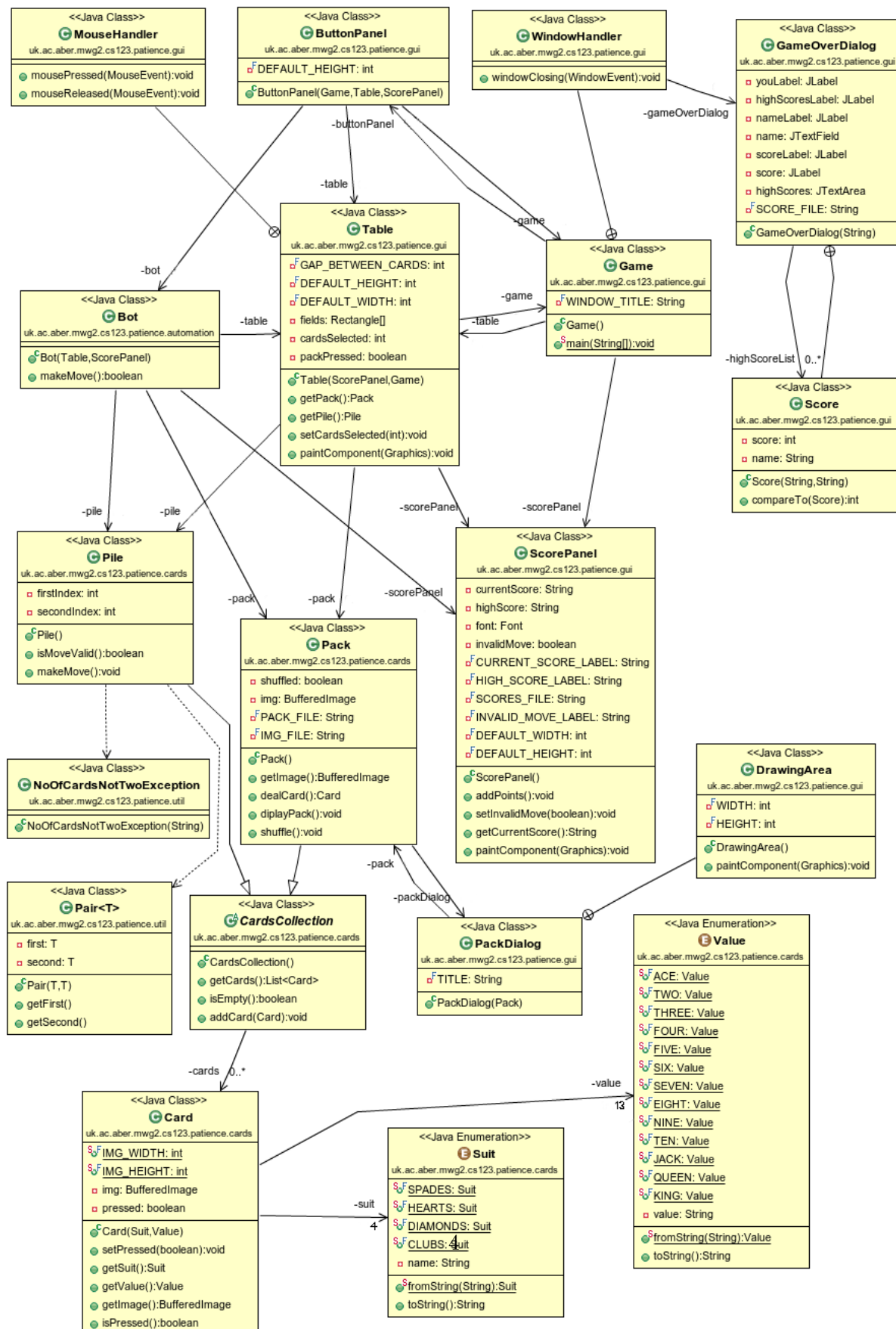
Game ends either if the automation algorithm detects that there are no more moves available, user presses the 'x' exit button on the top of the window or game is won. Before the application closes, a smaller window will pop up to ask the player to enter his name in order to save his score. User can choose not to store his result by leaving the name field blank.

2 Requirements Analysis

2.1 UML Use Case diagram



3.1 UML Class diagram

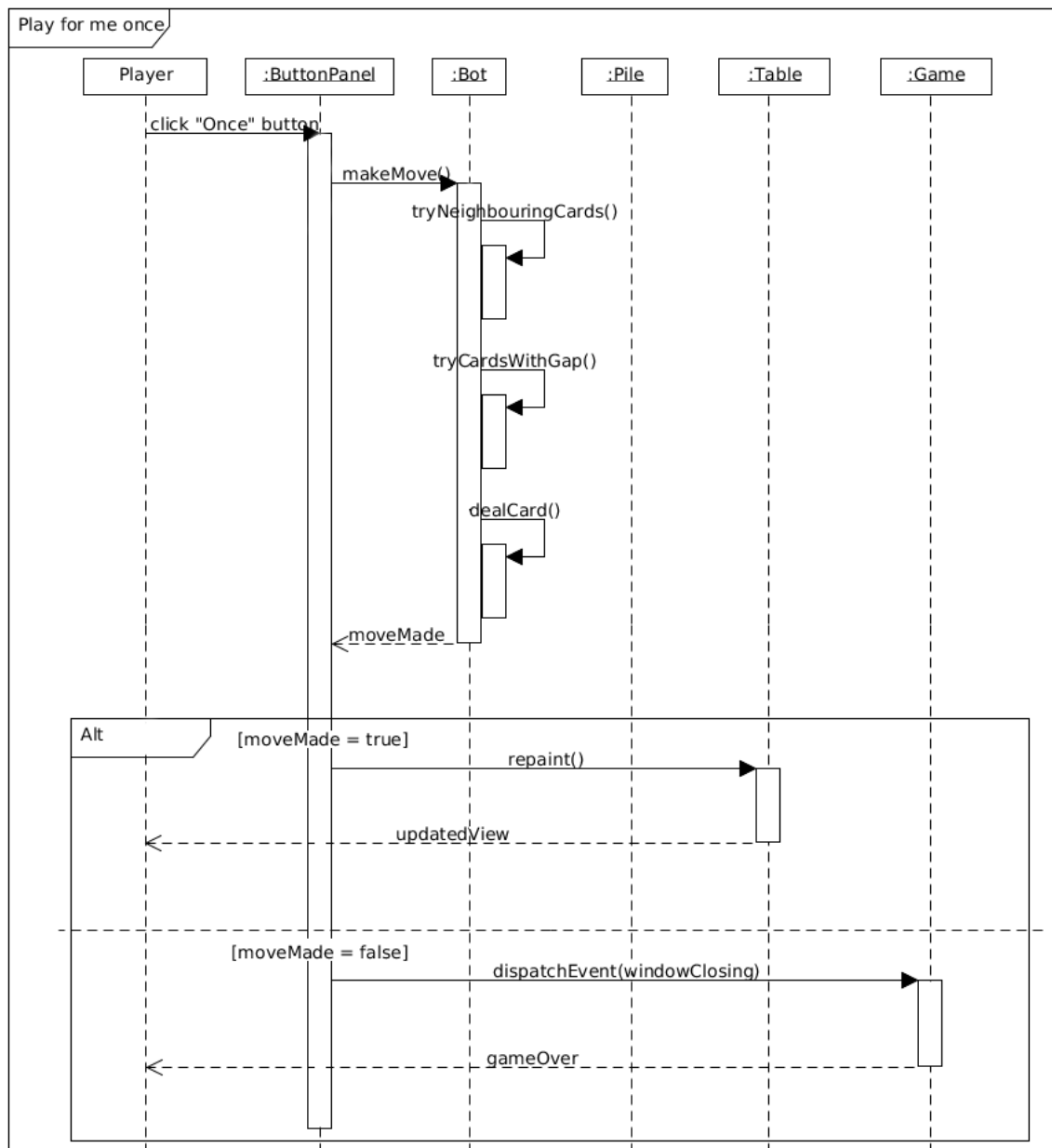


3.2 Description and relationships of each Class

1. **Game** class contains the **main** method and is therefore the starting point of the application. It also extends the **JFrame** class and takes care of the initialization and layout of the **ScorePanel**, **Table** and **ButtonPanel** components within itself.
2. **WindowHandler** is an inner class implemented inside the **Game** class. It extends the **WindowAdapter** class, which means that it can listen and react to the events generated by the window. Whenever a window closing event is being dispatched, this class displays a **GameOverDialog** dialog.
3. **GameOverDialog** is displayed whenever user wins the game or decides to exit the window. He will be then asked to enter his name which will be saved alongside his score. Previously recorded scores are stored inside a txt file **scores.txt** and are loaded into a list of objects of type **Score** whenever game is launched.
4. **Score** is an inner class located inside the **GameOverDialog**. It holds an information about the name of the player, his score and enables the list of scores to be sorted in a descending order.
5. **ScorePanel** is a GUI **JPanel** on top of the window. It displays the current score of the player, as well as the highest score recorded in the past. Object of this class can be notified if an invalid move has been detected in the game, via the **setInvalidMove** method. Appropriate message will be displayed to notify the player about it. Similarly, other objects can call the **addPoints** method to add 10 points to the current score and update the view.
6. **Table** extends the **JPanel** class and represents the playing table on which cards are being printed. **Pile** is printed on the top of the panel, while **Pack** on the bottom. **Table** object acts the view, whereas its inner class **MouseHandler** as the controller. It will update itself whenever it is notified about the change of either the **Pack** or the **Pile** contents. Furthermore, a blue border will be painted around each selected card in the game.
7. **MouseHandler** extends the **MouseAdapter** class and reacts to the events generated by player's mouse. It acts as the controller and enables the user to manipulate cards by clicking and selecting them. Whenever a card is selected, **MouseHandler** will notify the view **Table** to update itself. **MouseHandler** is also making sure that the maximum number of cards selected at any time is not greater than 2. If user selects two cards, **MouseHandler** will try to make a move by calling the **isMoveValid** and **makeMove** methods from the **Pile** object. Then it will tell the **ScorePanel** to either add 10 points to the current score or display the "INVALID MOVE" notification. Finally, it will dispatch the window closing event in case the user wins the game.
 - (a) Detect mouse pressed event
 - Clear the 'invalid move' notification from the panel
 - Check if any of the cards contain coordinates of the event
 - If so, select this card
 - Ask table to update itself

- (b) Detect mouse released event
 - If two cards are selected, make a move or unselect if its illegal
 - Ask table to update itself
 - Check if player won and finish the game if he did
- 8. **ButtonPanel** is located on the bottom of the window and contains buttons user can click to access additional functions. Displaying & shuffling the pack and automation of the gameplay.
- 9. **CardsCollection** is an abstract class which represents a collection of cards. Internally cards are stored in a list and can be accessed and removed using provided methods.
- 10. **Pack** extends the **CardsCollection** and represents a 52 cards pack. Cards can be shuffled once and removed from the pack using the **shuffle** and **dealCard** methods respectively. If the **displayPack** method is called, a dialog will pop up showing the contents of the **Pack**.
- 11. **PackDialog** is a modal dialog box which consists of two components. In the centre of the box, a **DrawingArea** paints current contents of the **Pack**. A button panel is situated on the bottom of the window enabling the player to close the dialog.
- 12. **DrawingArea** is an inner class inside the **PackDialog**, which extends the **JPanel** and overrides the **paintComponent** method. Its only purpose is to graphically display contents of the **Pack**.
- 13. **Pile** extends the **CardsCollection** and represents cards dealt from the **Pack** on to the **Table**. It also holds the information about the indexes of the cards which are currently selected. It has two very important methods **isMoveValid** and **makeMove**. First one checks if the two selected cards correspond to any of the known moves in the game, while the second one actually makes the move. **isMoveValid** has to be called before the other one.
- 14. **NoOfCardsNotTwoException** is thrown whenever the number of pressed cards is not as expected equal to 2.
- 15. **Pair<T>** a simple generic class which represents a pair of any two objects of the same class. Allows to easily pass around two objects together.
- 16. **Card** class represents a single playing card. Each card has its value, suit and an image. An appropriate picture is loaded depending on the suit and the value of the card.
- 17. **Suit** enum represents a suit of a card. Each suit has its corresponding single character String representation, accessible with the **toString** method.
- 18. **Value** enum represents a value of a card. Each value has its String representation, accessible with the **toString** method.
- 19. **Bot** object can be used to automate the game. Its only public method **makeMove** tries to make a move in the game and returns **true** if it succeeds or **false** otherwise. Cards in the **Pile** are firstly checked from right to left to see whether it is possible to join any (make a move). If none of the cards can be combined, bot will try to deal a card from

the **Pack**. If that fails then the game is over as there are no more available moves, and an appropriate event will be dispatched to display the **GameOverDialog**.



4 Testing

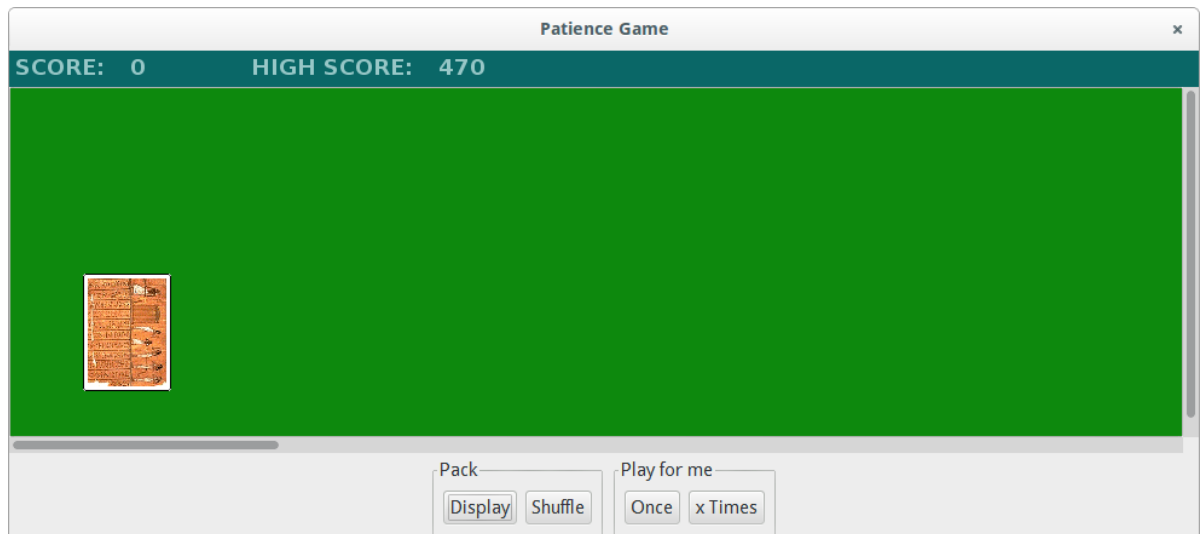
4.1 Test tables

ID	Requirement	Description	Inputs	Expected Outputs	Pass / Fail	Comments
A1.1	FR1	Load the proper look and feel depending on The OS	Start game on Ubuntu GNOME	SS1 screenshot displayed	P	
			Start game on MS Windows	SS2 screenshot displayed	P	
			Start game on Mac OS	SS3 screenshot displayed	P	
A1.2	FR2	Show current Contents of The pack	Click "Display" button when pack Is not empty	SS4 screenshot displayed	P	
			Click "Display" button when pack Is empty	SS5 screenshot displayed	P	
A1.3	FR3	Shuffle the pack	Click "Shuffle" button once	SS6 screenshot displayed	P	
			Click "Shuffle" button twice	SS7 screenshot displayed	P	
A1.4	FR4	Deal a card	Click on the deck	SS8 screenshot displayed	P	
A1.5	FR5	Select cards on the table	Click on an unselected card	SS9 screenshot displayed	P	
			Click on a selected card	SS10 screenshot displayed	P	
A1.6	FR6	Make a move	Select two cards next to each Other see (SS11)	SS12 screenshot displayed	P	
			Select two cards with a gap 2 card Gap in between see (SS13)	SS14 screenshot displayed	P	
			Make an invalid Move See (SS15)	SS16 screenshot displayed	P	

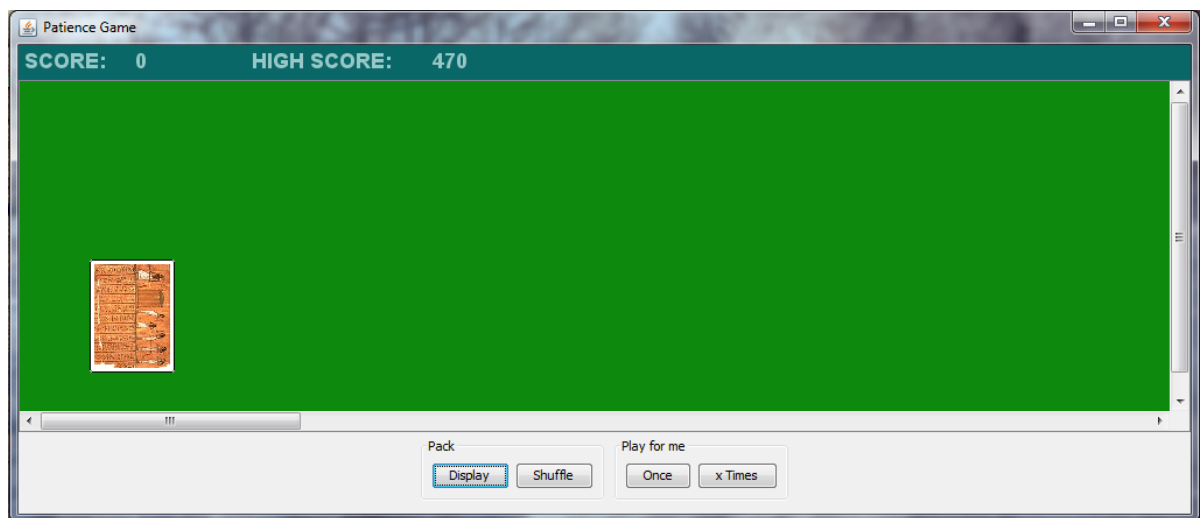
ID	Requirement	Description	Inputs	Expected Outputs	Pass / Fail	Comments
B1.1	FR7	Play for me once	Click "Once" button when cards on the table can Be joined see (SS17)	SS18 screenshot displayed	P	
			Click "Once" button when cards on the table Cannot be joined see (SS19)	SS20 screenshot displayed	P	
			Click "Once" button when there are no more moves in the game	SS21 screenshot displayed	P	
B1.2	FR8	Play for me a specified number of times	Click "X times" button, enter 20 and press "Ok" (see SS22)	SS23 screenshot displayed	P	
			Click "X times" button, enter -10 and press "Ok"	SS24 screenshot displayed	P	
			Click "X times" button, enter "abc" and press "Ok"	SS24 screenshot displayed	P	
B1.3	FR9	Quitting the game	Press 'x' exit button on the top of the window	SS21 screenshot displayed	P	
			Win the game See (SS25)	SS21 screenshot displayed	P	
B1.4	FR10	Saving the score see (SS21)	Leave the name field blank and press "save & exit" button	Program closes and player's score is not remembered	P	
			Enter name "Kevin" and exit. Then relaunch the game and quit immediately	SS26 screenshot displayed	P	

4.2 Screenshots

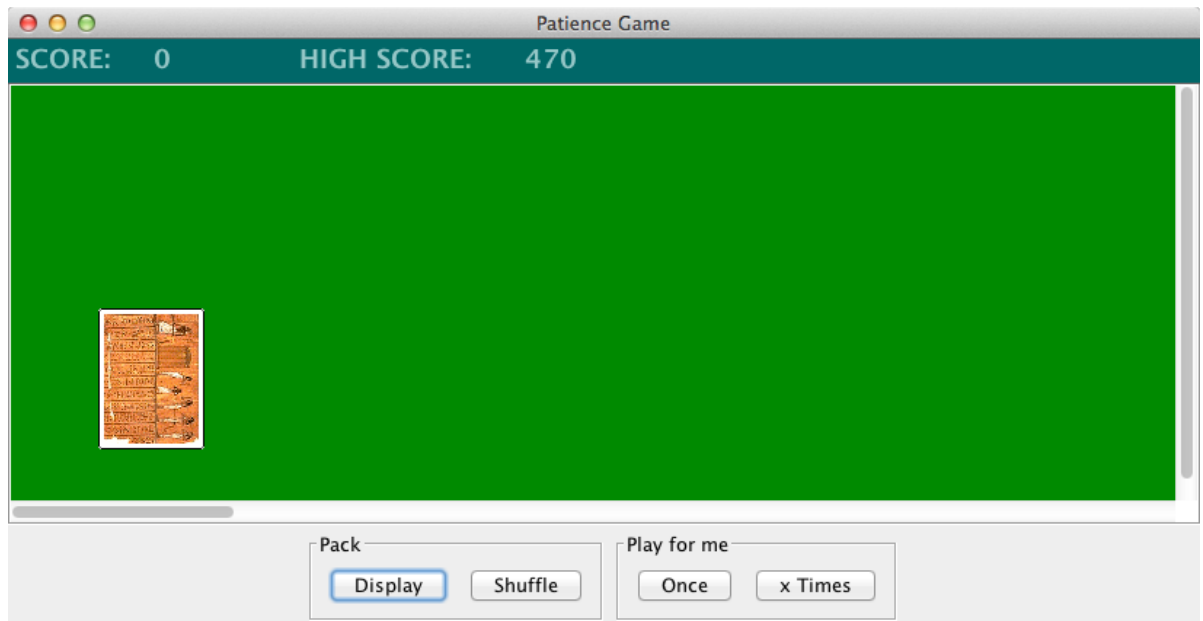
SS 1



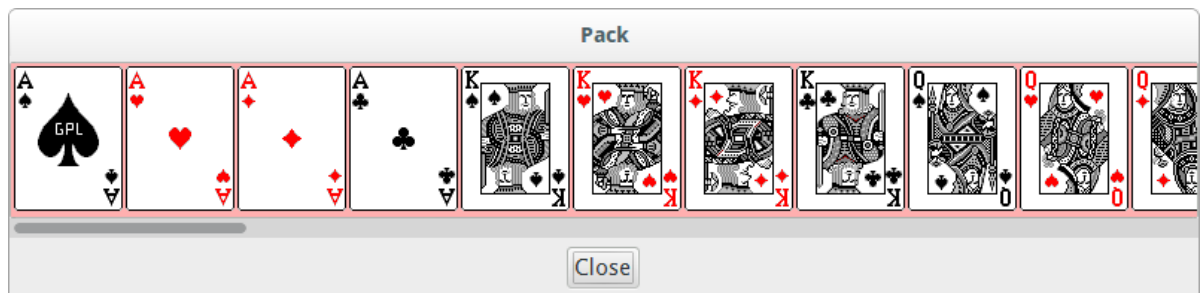
SS 2



SS 3



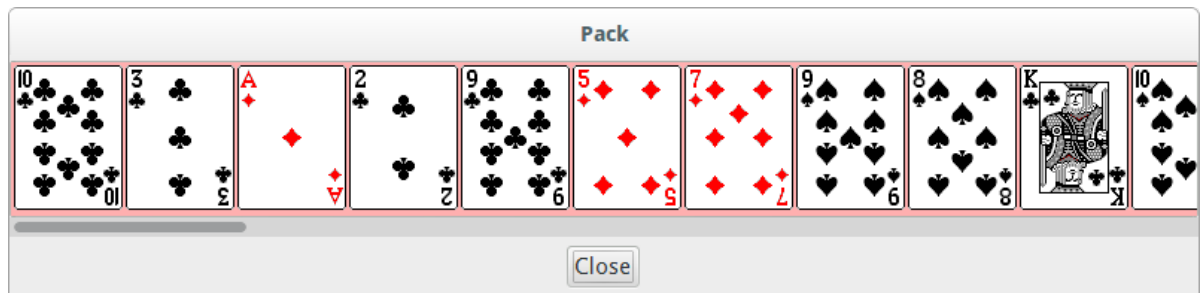
SS 4



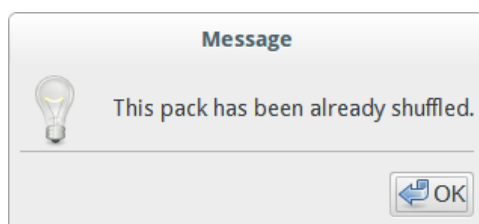
SS 5



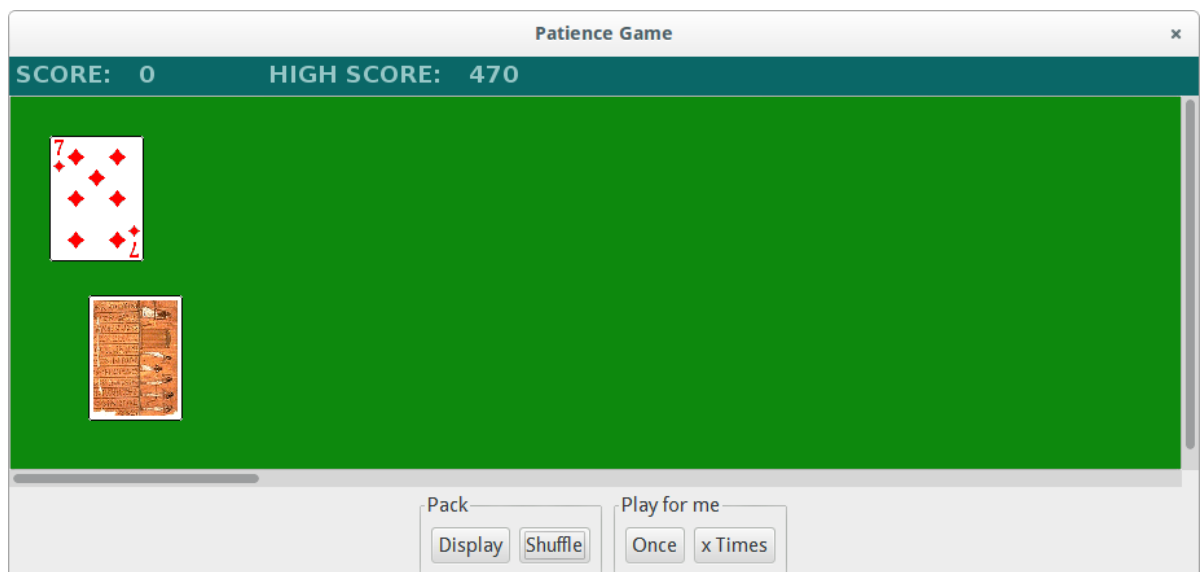
SS 6



SS 7



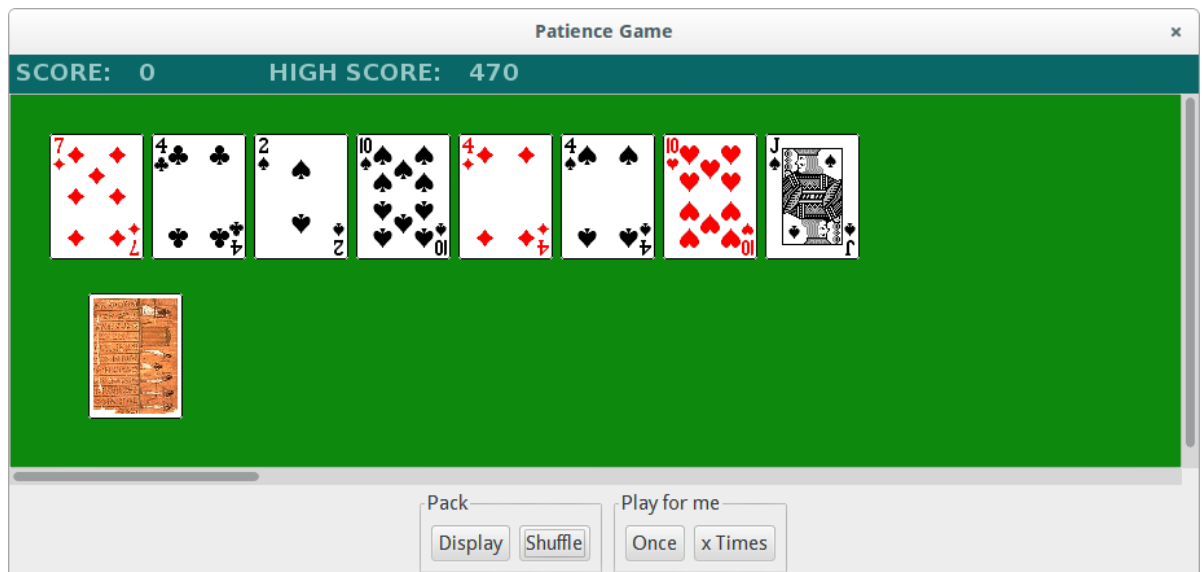
SS 8



SS 9



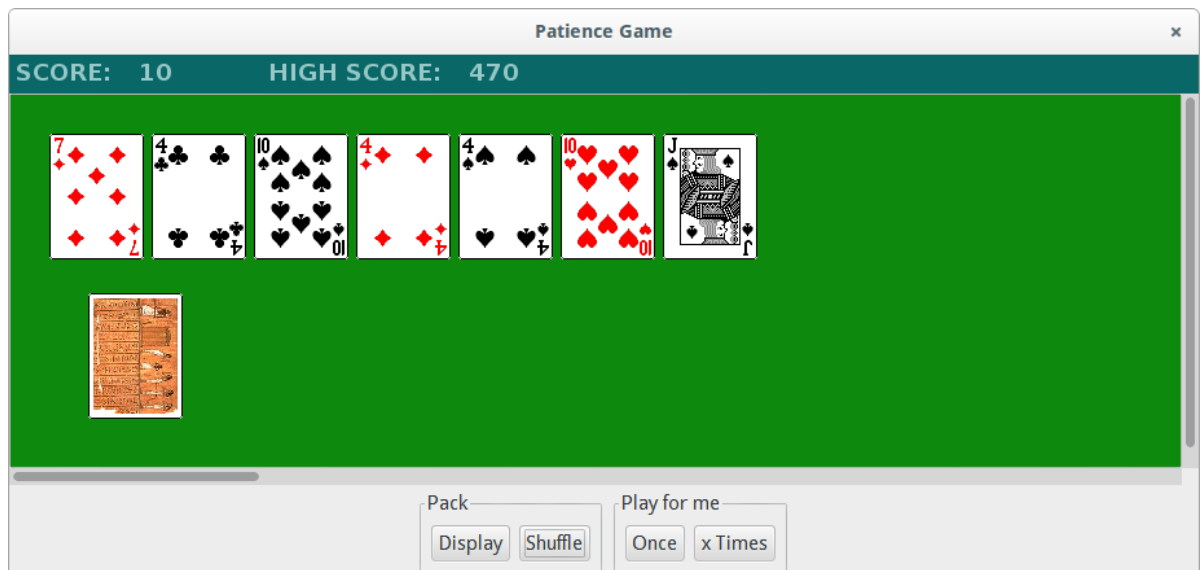
SS 10



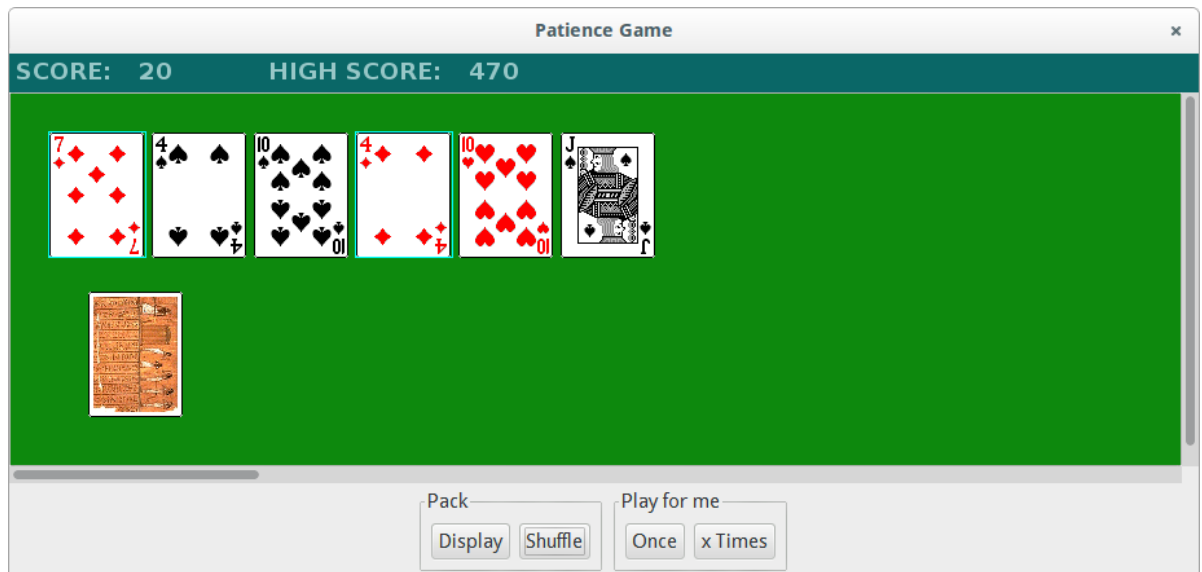
SS 11



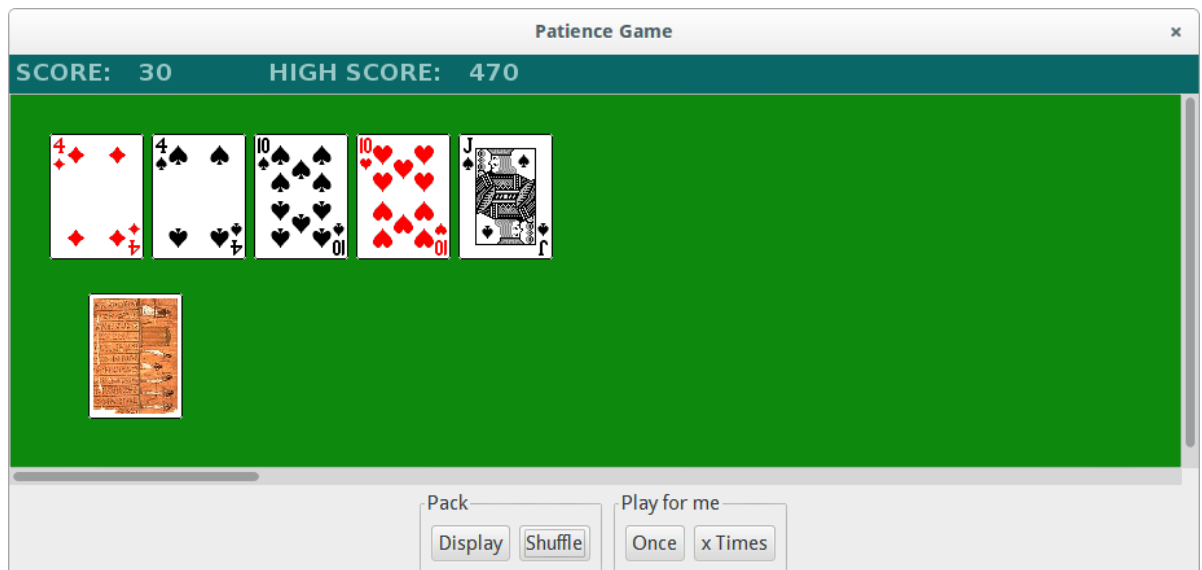
SS 12



SS 13



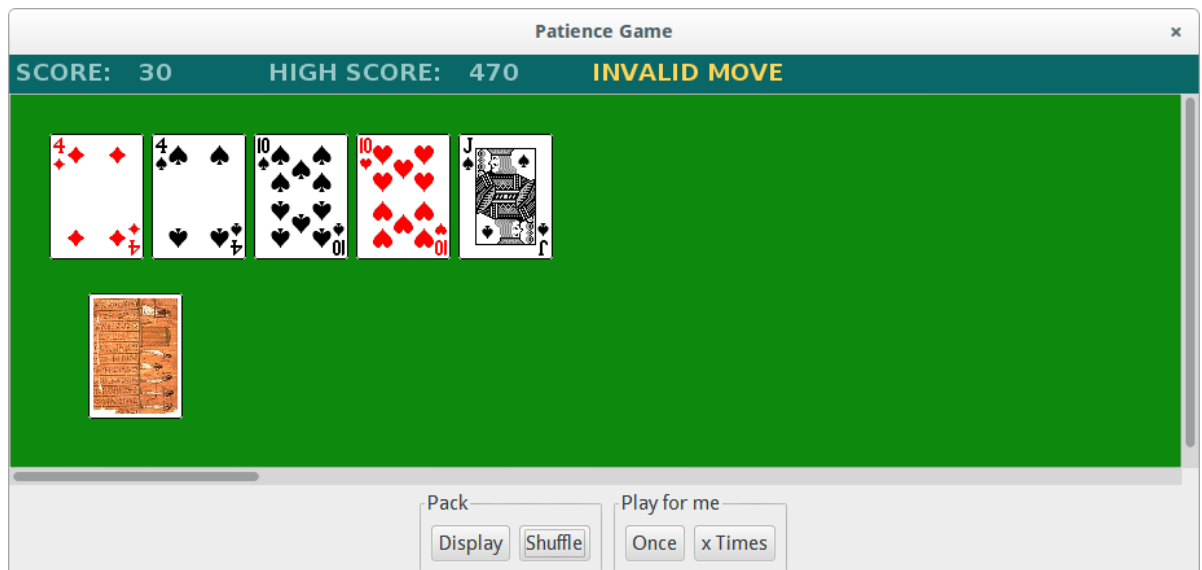
SS 14



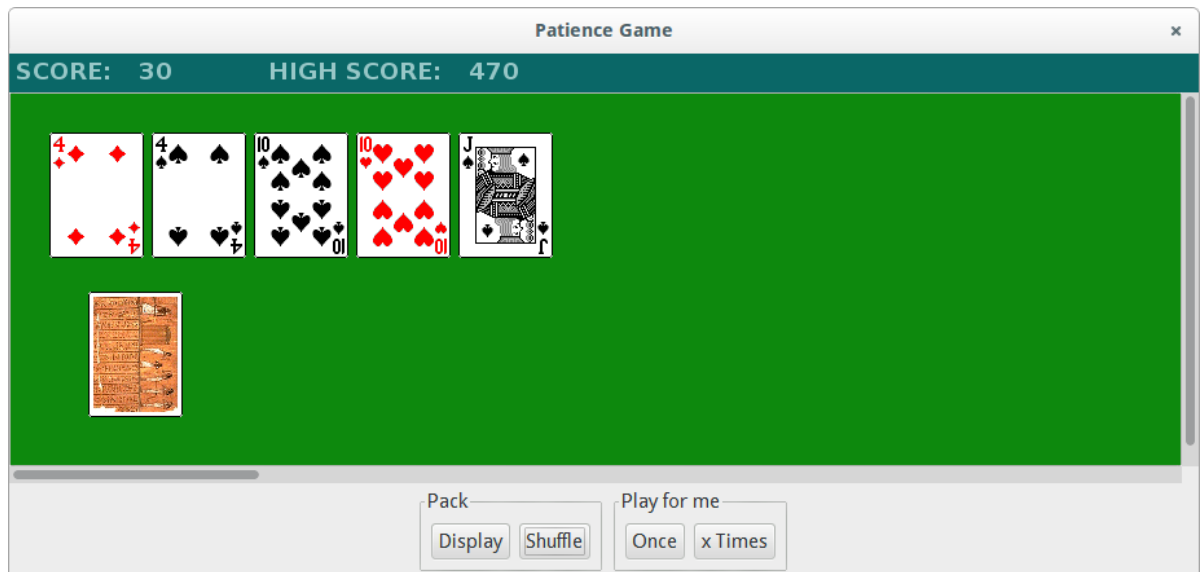
SS 15



SS 16



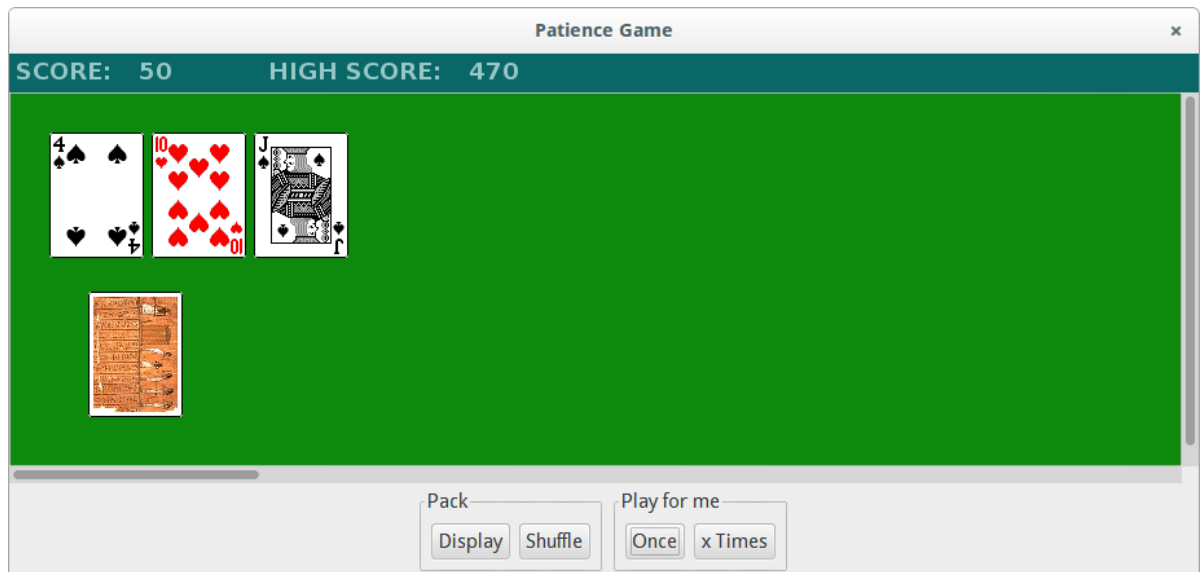
SS 17



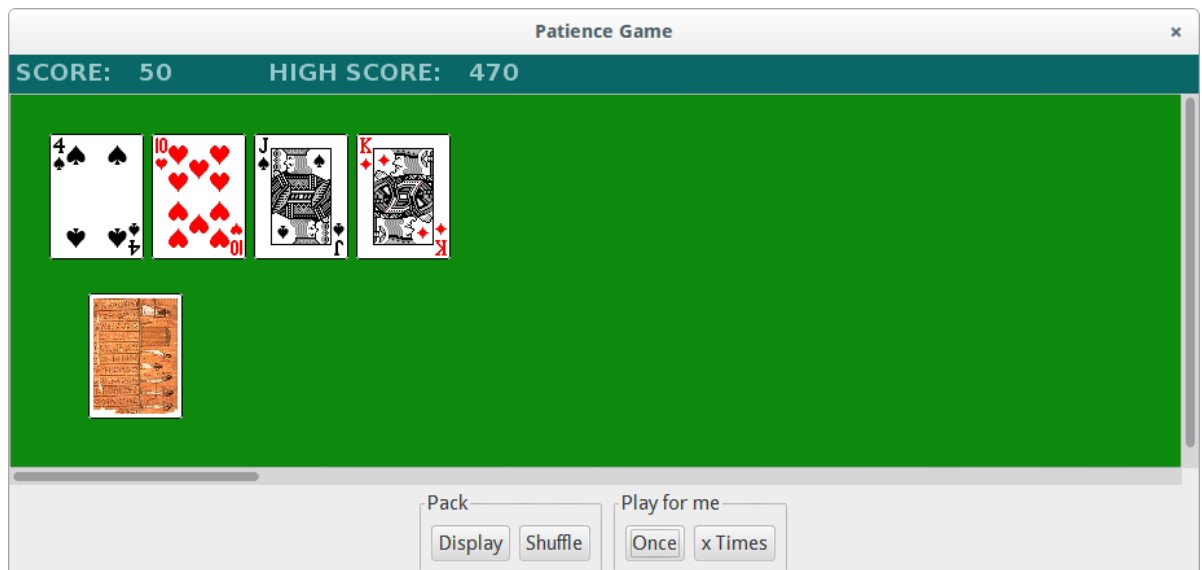
SS 18



SS 19



SS 20



SS 21

Game Over!


You	High Scores
Name: <input type="text"/>	470 Michal
	470 wojt
	220 John
	100 Bob
	60 James Jones
	60 Adam
	30 Michal

Score: 390

SS 22

Patience Game

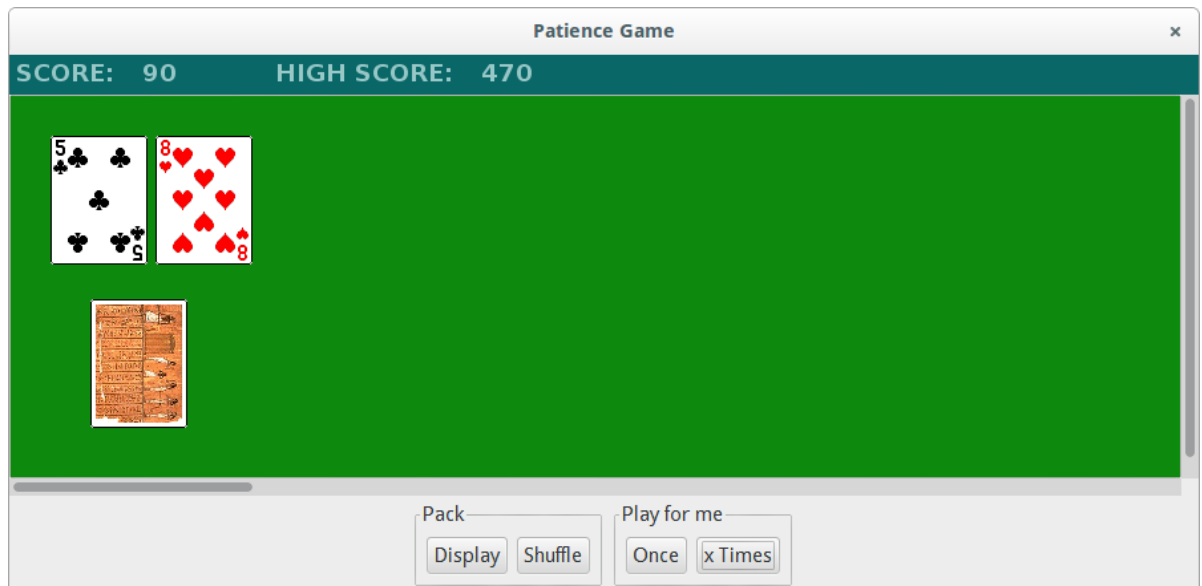
SCORE: 0 HIGH SCORE: 470



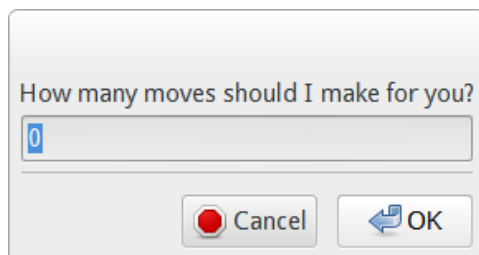
Pack—

How many moves should I make for you?

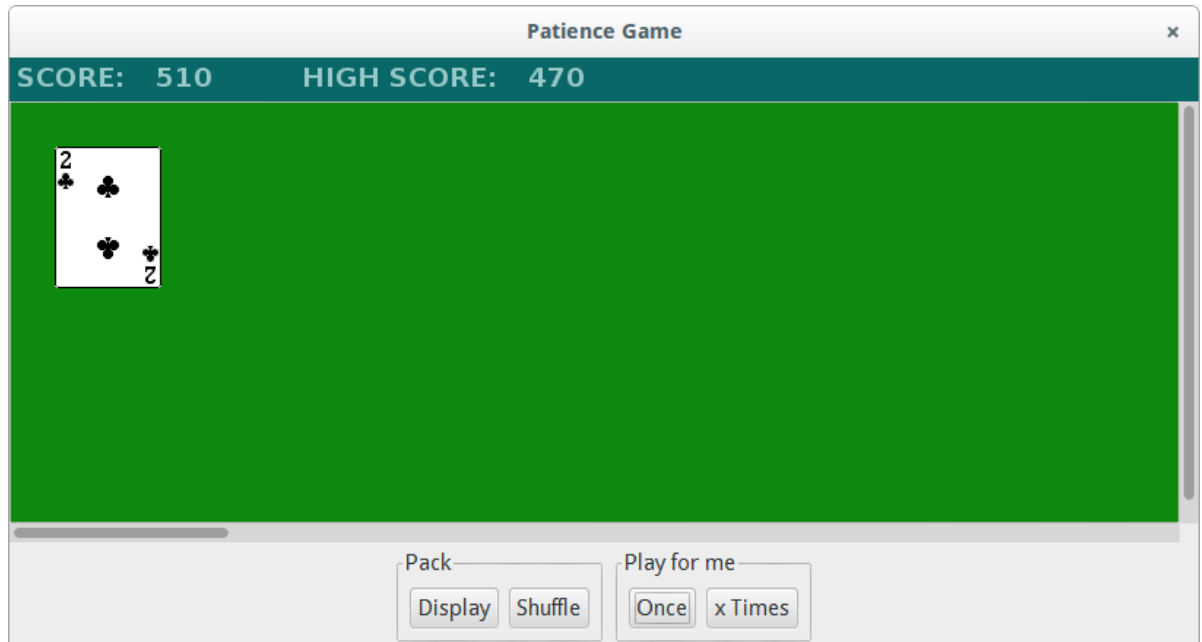
SS 23



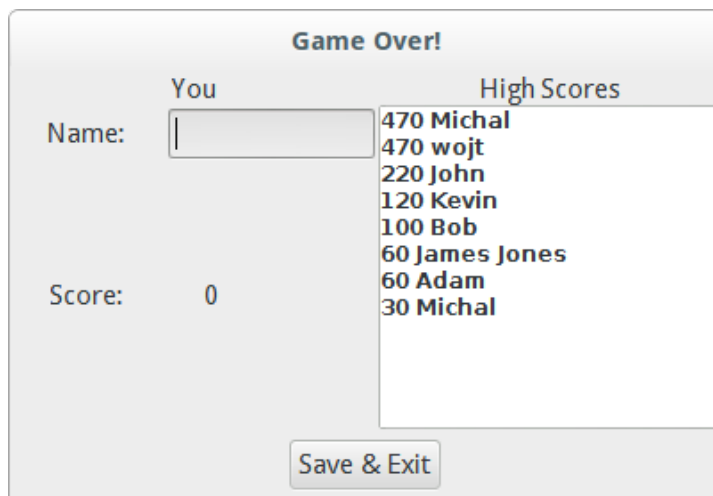
SS 24



SS 25



SS 26



5 Evaluation

I began by reading the requirements specification and trying to understand the rules of the game. Unlike in the previous mini-assignments, we did not receive an initial implementation of the program. Therefore I realised that I should not rush into writing the code, before creating a simple design of the system. At this point I knew, that I am going to need to represent playing cards, their suits and values, pack and some sort of collection of cards the player could select.

5.1 Completely graphical user interface

I have been asked to create a console application which could also take the advantage of the GUI framework provided. Player could then type in commands from the menu and make moves in the game. The GUI frame would be used to graphically display his progress.

In order to make the assignment more challenging and fun, I have decided to make the application completely graphical. User could then click on cards and buttons rather than typing commands into console. This approach however, eliminated the point of the two requirements in the specification of the assignment. Without the console, "control text display" option could not be implemented, nor could be printing of the output into a text file. Furthermore, the "display pack" option could not just print deck's contents. Instead it displays a modal dialog box which graphically presents cards in the pack.

5.2 Initial implementation

Card was the first class I wrote and straight away I knew that apart from keeping the track of its suit and value, it would also need to know whether it has been pressed or not. A boolean instance variable was a sufficient solution to that problem. Subsequently, I moved on to create a **Pack** which would initially store 52 ordered playing cards. **Pile** class represents the cards which were delt from the pack on to the table. Because it has similar basic behaviour as the **Pack**, I have decided to use inheritance and abstract this functionality into a parent class called **CardsCollection**.

5.3 Towards the working application

I knew that the GUI framework provided, would not be sufficient to make the game completely "terminal free". Instead I started from scratch, keeping in mind the MVC design objectives I have read about in the past. I wanted to keep the data, view and controller separate and began to implement the GUI.

User interface (view) consists of the three main components:

- **ScorePanel** which displays current score, highest score ever recorded and an "invalid move" warning when an illegal move has been detected
- **Table** which is a "drawing area" on which cards are being drawn
- **ButtonPanel** which provides the player with additional functions like "shuffle the pack".

Most of the game logic (controller) has been implemented in these classes:

- **MouseHandler** is an inner private class inside the **Table**, which listens to mouse events and enables the player to play the game by clicking on cards.
- **Pile** which also acts a model, makes sure that when two cards are selected it checks whether there is a valid move available. It also can actually perform the move by joining selected cards.
- **Bot** where the game automation code is located.

5.4 Summary

This assignment was definitely more difficult and complex than the previous two mini-assignments. However starting from scratch, enabled me to go through almost every software development life cycle, apart from the "maintenance" one. I enjoyed coding itself the most, but the testing experience will probably help greatly next year and in the future career. Because I have fulfilled all the assignment requirements and created a more complex, fully graphical game, I would be happy to receive a mark of 90% for this assignment.