Title of the assignment

Student name
student@aber.ac.uk

Submission date: xx/xx/xxxx

Contents

Introduction

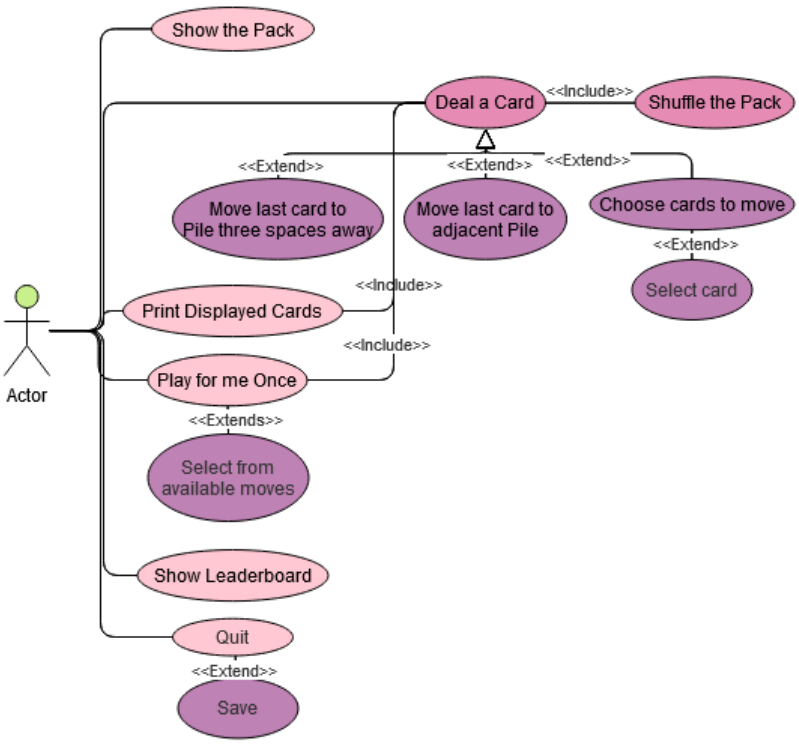
In my patience game program, 10 functional requirements are achieved to create a playable and fully-featured game.

I have created a program which presents a command line menu to the user, alongside a graphical display for the cards. The user has access to many movement options and can save their score once completing the game.

In addition, I have expanded on these requirements to create a drag-and-drop menu, through which the patience game is fully playable.

This document contains details of the development and testing of the game, including screenshots and diagrams. It evaluates how successful the complete result is in achieving the requirements given.

Use Case Diagram



Design

Class Diagram

Textual Class Description

The *Game* class is used to run a game of patience. Each *Game* has many *Decks*, most of which are *Piles* contained within a piles *ArrayList*. One instance of the *Deck* superclass is typically present.

These objects have a *pack* array, from which new cards are taken. It is made up of Strings, each of two characters. One representing the number, and another representing the suit.

Deck has a *getCard()* method which can return the value of a card from a given index in the pack.

The *Pile* subclass adds a *position* attribute, which tracks a pile's location on the table. This is used

instead of checking the index because the *position* value is sometimes needed after the index has already changed, such as when placing a card on top of another.

Many instances of *Score* are utilised in the leaderboard. Each holds a player's *name*, along with their *points*.

A *Game* also has a single *CardTable* which is used when displaying the GUI.

The GUI-only version of this program includes an additional class, *Name*. This is used to create a menu for a user to input their name while saving the game.

It also expands on the *CardTable* class, including additional functions to handle the new graphical elements.

Pseudocode

```
void move(int jump, int start){
//Get cards from Piles
  if ( start >= 0 and <= piles.length() ) {
      pile = piles(start)
      String top = pile.topcard()

int index = start - jump
      if ( index >= 0 and ind <= piles.size() ) {
             pile = piles(ind)
             String bottom = pile.topcard()

            //Check whether move is valid and make move if true
            if (top and bottom match suit or number) and (j = 1 or 3)) {
               piles.remove(pile)
               print("Move successful!")
               arrangePiles()
             }
}
}
  else print("This move is not possible!")
}

//get rid of gaps
void arrangePiles(){
    int pos = 0
    for (i in piles) {
        if (i.position() is not pos) {
            i.setPosition(pos)
        }
        pos++
    }
}
```

Testing

Test Table

ID	Requirement	Description	Inputs	Expected Outputs	Pass/Fail	Comments
A1	FR1	Show the pack	Enter '1'	The contents of the pack are displayed in the terminal	P	
B1	FR2	Shuffle the pack	Enter '2'	The pack is re-ordered	P	This can be confirmed by outputting the pack
B2		Attempt once game has begun		The input is rejected	P	
C1	FR3	Deal a card	Enter '3'	A card String is moved from the pack array into the list of piles	P	
C2		Attempt when the pack is empty		No card is dealt and the user is told to try another option	F	An Index out of bounds exception occurred.
D1	FR4	The card which has just been dealt is moved onto the pile next to it	Enter '4'	The 'bottom' pile is removed from the list if valid	P	
D2		Attempt when no piles exist		The move is rejected	P	
D3		Attempt when the move is not valid		The move is rejected	P	
E1	FR5	The card which has just been dealt is moved onto the pile three spaces to the left	Enter '5'	The 'bottom' pile is removed from the list if valid	P	
E2		Attempt when no piles exist		The move is rejected	P	
E3		Attempt when the move is not valid		The move is rejected	P	
F1	FR6	The user selects two piles to merge (three spaces away)	Enter the index of two valid cards	The 'bottom' pile is removed from the list if valid	P	
F2		The user selects two piles to merge (adjacent)	Enter the index of two valid cards	The 'bottom' pile is removed from the list if valid	P	
F3		Attempt when no piles exist	Enter '5'	The move is rejected	F	While the input was correctly handled, it still asked the player to give card indexes first
F4		Attempt when the move is not valid	Enter the index of two invalid cards	The move is rejected	P	
F5		Attempt with invalid indexes	Enter an invalid index (such as a card 2 or 5 spaces away)	The move is rejected	P	

F6		Attempt with out-of-range indexes	Enter an index which is too high	The move is rejected	F	Index out of bound exception occurred. Check put in place to prevent this.
G1	FR7	Print Displayed Cards	Enter '7'	All current piles are displayed on the command line	P	
H1	FR8	The Program automatically performs a move	Enter '8'	The 'bottom' pile is removed from the list	F	This occurs when there is only one possible move, or multiple adjacent moves and one three-pile move which takes priority Despite a possible move being available, the function found no moves. This was due to indexes of 0 not being registered by the length checking function. Rather than signifying an empty array slot with '0', I filled them with MAXDECK+1 Additionally, jumps resulting in a 0 index were restricted by the move validator. I replaced the >0 with a >=0
H2		The program presents the user with a list of possible moves			P	This occurs when there are multiple possible moves. If there are three-pile moves, no adjacent moves will be shown
H3		The user selects a move from the list presented	Enter an index from the move list	The 'bottom' pile is removed from the list	P	The pile with the lower index in the Arraylist is the one removed
H4		User selects an invalid move	Enter an out of bounds index	The move is rejected	P	
H5		Attempt when no piles exist	Enter '8'	The move is rejected		
I1	FR9	Show Leaderboard	Enter '9'	The top ten past scores are displayed	P	A player starts with points equal to the number of cards in the deck. One point is removed for each remaining pile.

J1	FR10	Quit	Enter 'Q'	The game ends and the user is directed to close the GUI window	P	The final score is saved after quitting
K1	NFR3	Saving	Enter Name	The score is saved to file if valid	P	
K2		Attempt to quit with a non-empty deck	Enter 'Q'	No score is saved	P	All cards must be dealt for the number of remaining piles to be judged
L1	NFR1	The menu is displayed	N/A	The user is presented with a list of options. All inputs lead to the correct functions	P	
L2		Attempt invalid option	Enter '%'	The input is rejected	P	
M1	NFR2	GUI	N/A	The graphical interface is displayed and updated following each move	P	

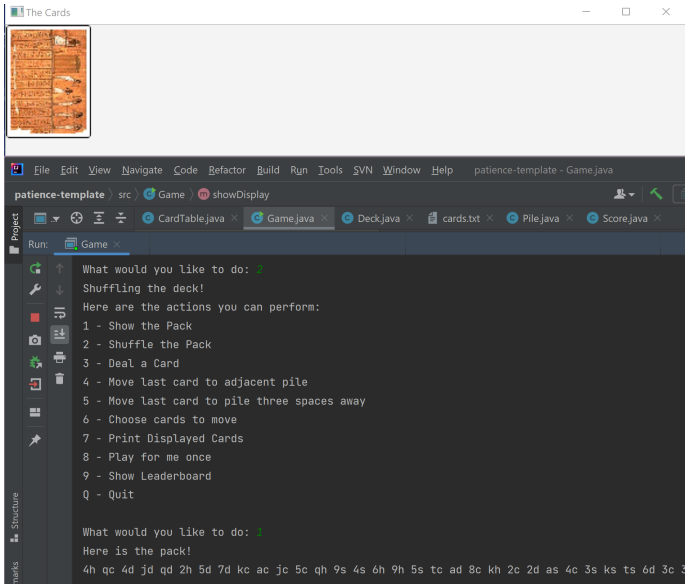
Screenshots

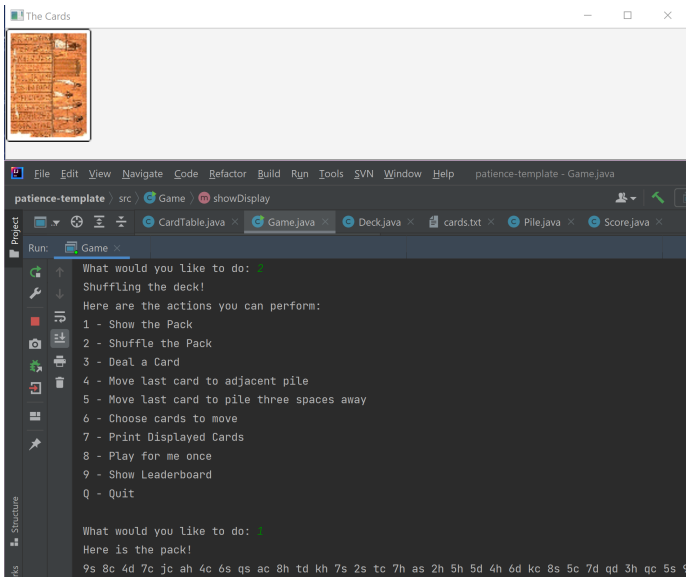
Most screenshots were taken while the program was running in IntelliJ. The command line version was not compiled until after testing was complete.

FR1 - Show the pack

FR2 - Shuffle the cards

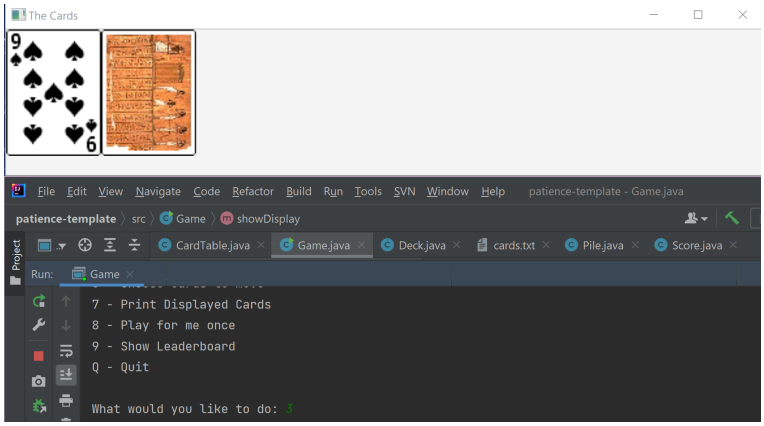
The *deck* is shuffled and printed to the command line.



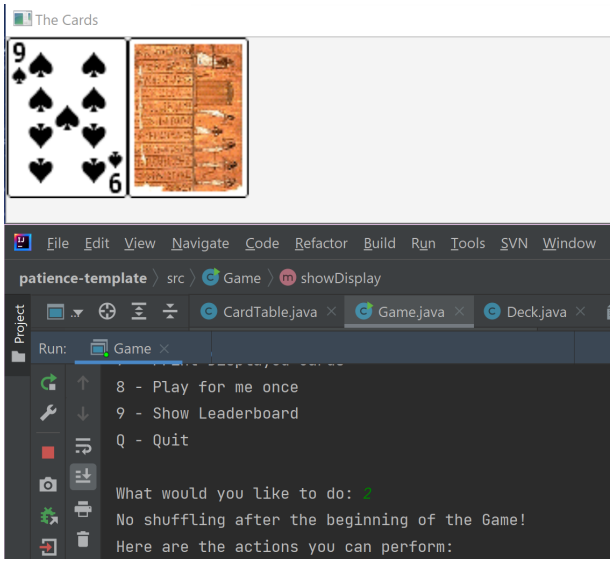


FR3 - Deal a card

A card is dealt from the *deck*

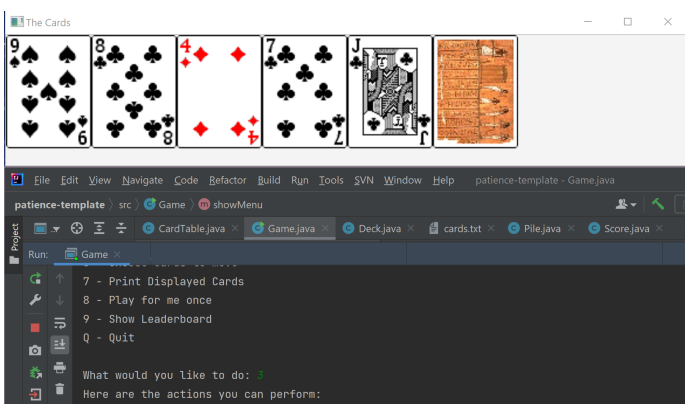


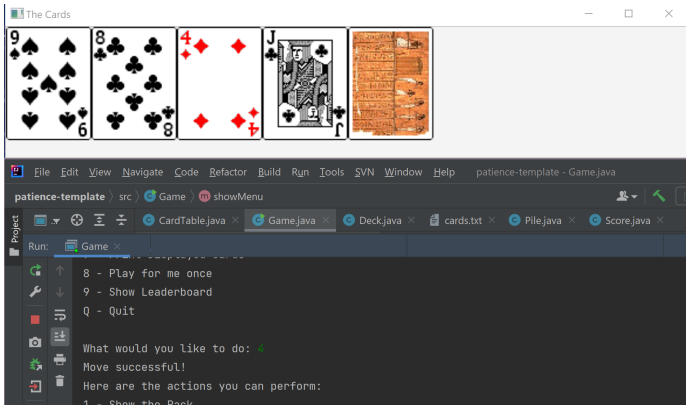
The player is not allowed to shuffle once the game has begun



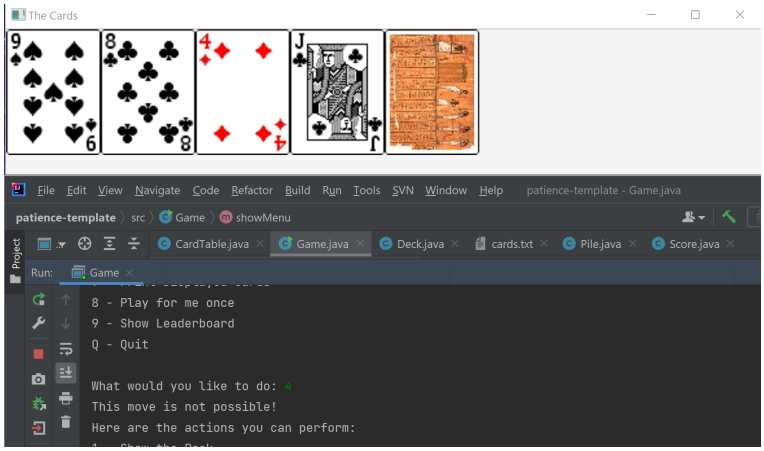
FR4 - Move last card to adjacent pile

The last card is amalgamated with the pile beside it



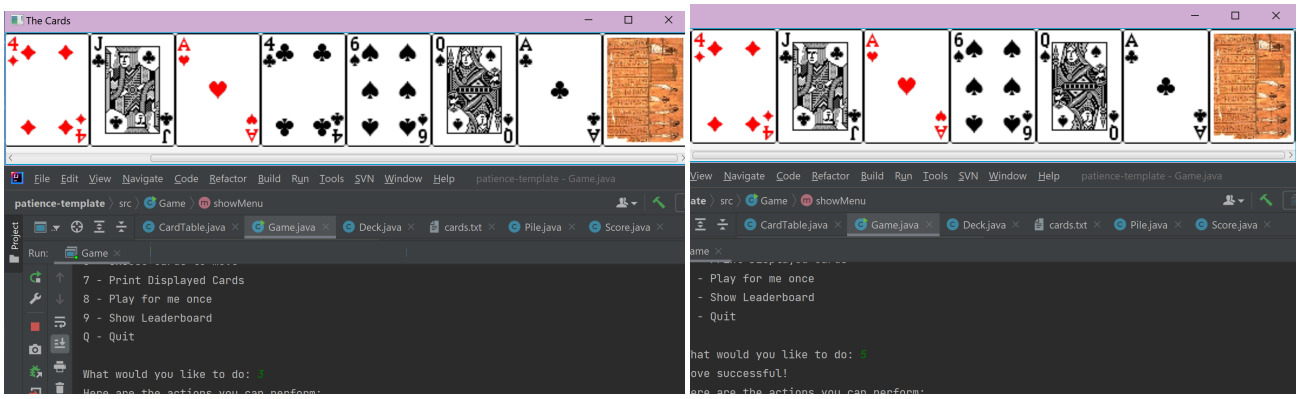


An invalid move is rejected

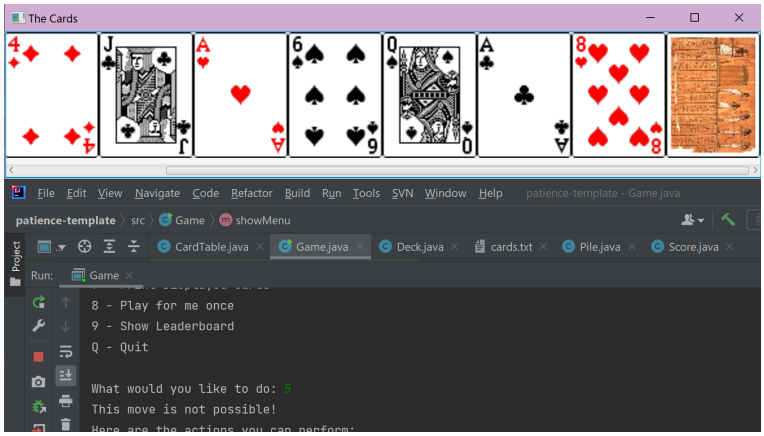


FR5 - Move last card to pile three spaces away

The last card is amalgamated with the pile three spaces away

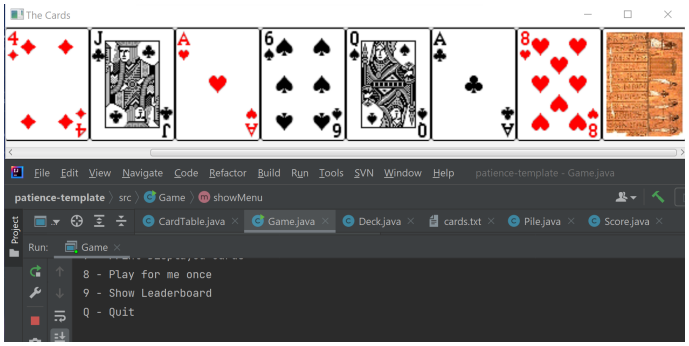


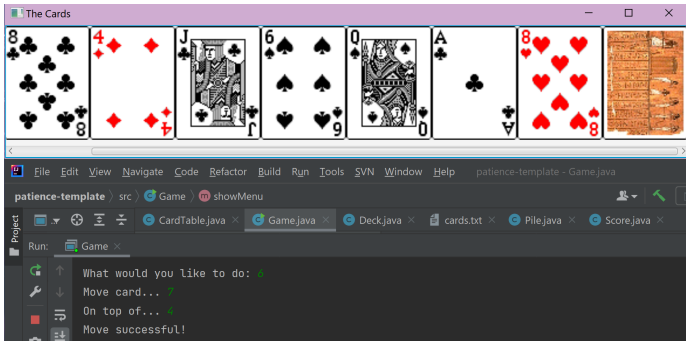
An invalid move is rejected



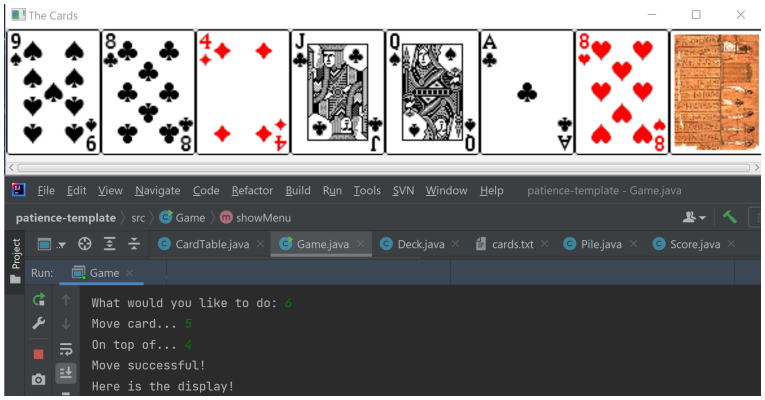
FR6 - Choose cards to move

The two specified piles are amalgamated

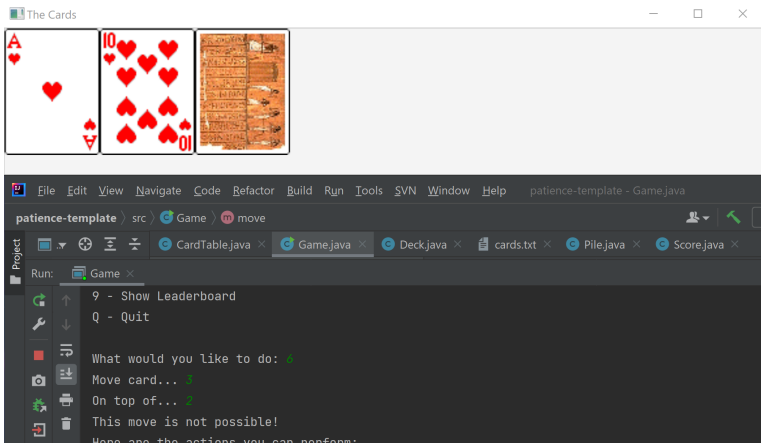




Specified adjcacent piles are amalgamated

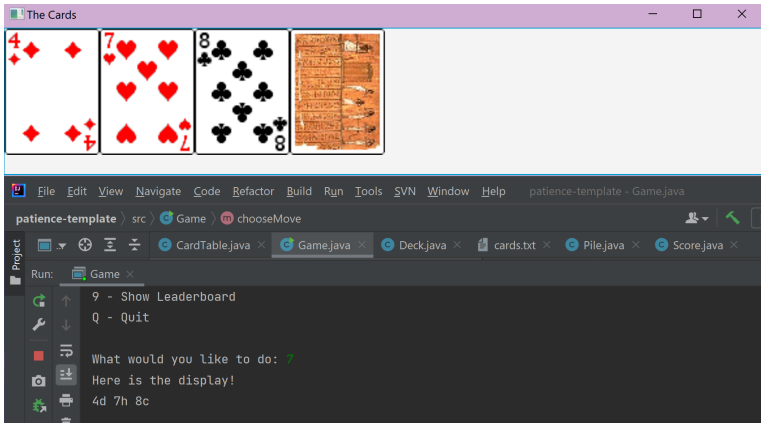


An invalid move is rejected



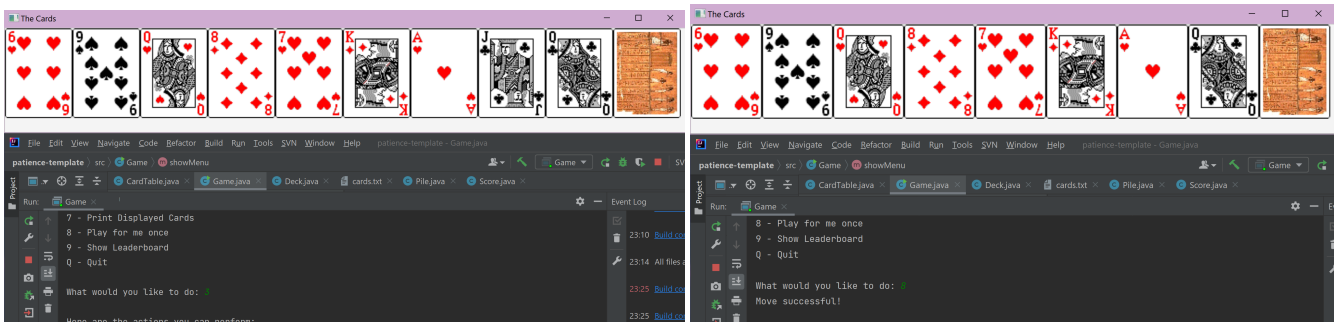
FR7 - Print displayed cards

The cards on the table are printed to the command line

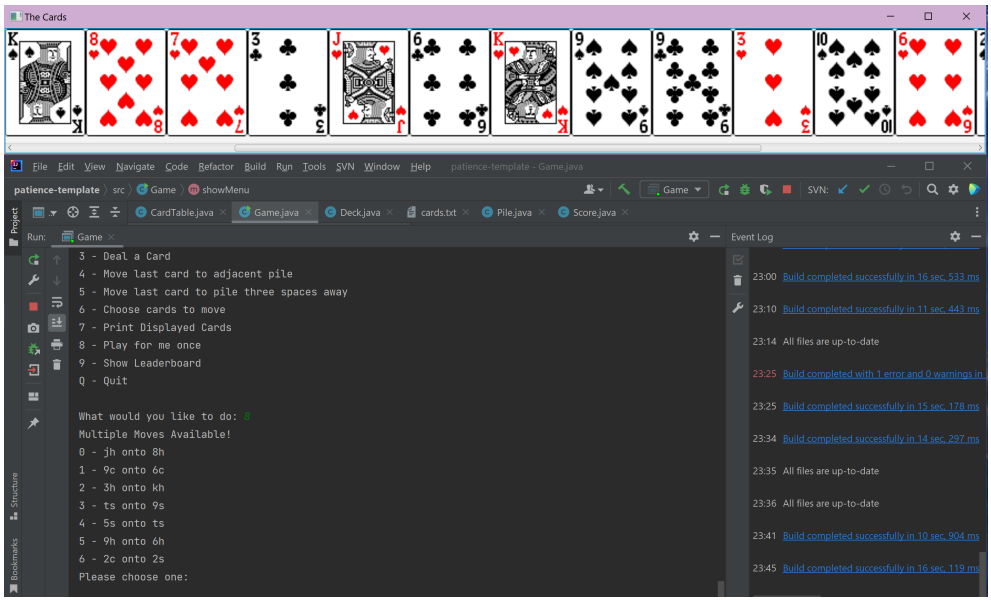


FR8 - Play for me once

A move is automatically made

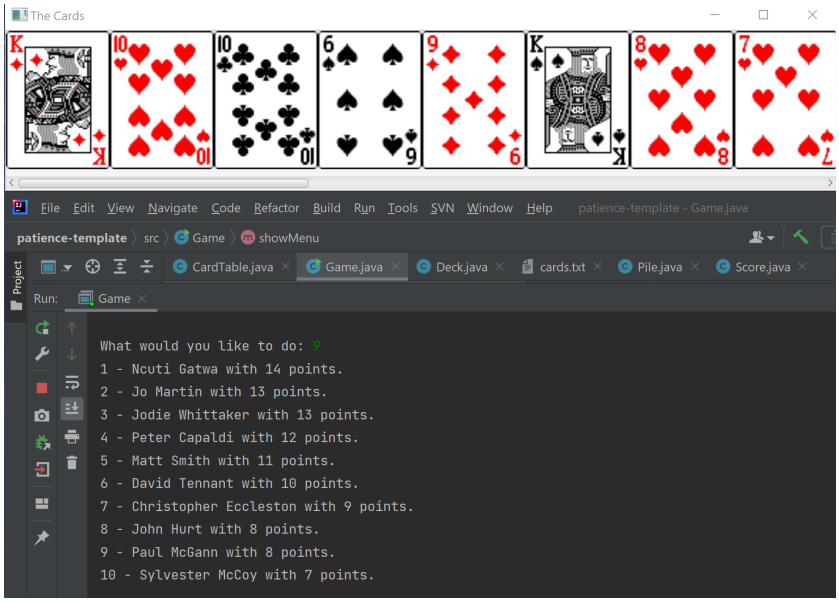


The move selection list is shown



FR9 - Show leaderboard

The leaderboard is displayed



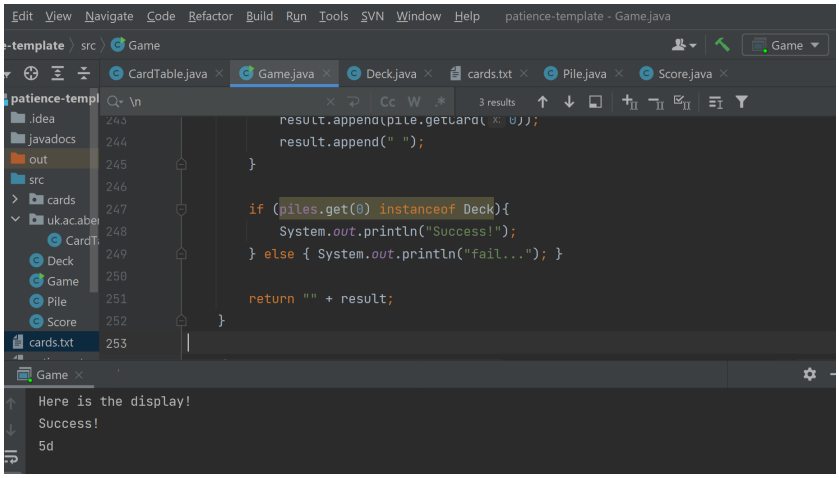
On quitting, the player's name is requested in order to save

```
What would you like to do: q
To fully quit, close the graphical window!
Saving score...
What is your name?
Cassidy

Process finished with exit code 0
```

The IS-A test

The IS-A test is performed



Command Line

The Patience game runs from the command line

Discussion

The patience game does not require much specific user input, so most test data is interaction with the menu. I tested valid and invalid options to ensure the menu could handle unexpected characters.

The game at first attempted to perform actions such as pulling cards from the deck even when it was empty. I implemented a check to make sure the deck existed first.

Functional requirements 6 and 8 require users to specify indexes.

For requirement 6, no specific indexes could be included in the test table, as the positions of *Piles* are shuffled each game.

While the system handled most incorrect indexes, selecting an out-of-bounds index caused an exception. I introduced additional defensive programming techniques to rectify this, and other similar errors.

In requirement 8, *addMoves()* occasionally didn't recognise moves that were present on the table. Using the IntelliJ debugger, I discovered *checkLength()* did not count moves containing piles of index 0, due to 0 being used to signify an empty array slot.

To fix this, I initialised empty slots with *MAXDECK+1*, a number the amount of piles cannot reach.

Requirement 10 used a *cards.txt* file as input. This file should not be altered by a user, so is stored as a static variable. It is correctly read by the program to form the leader board. The *Score* for a user can also be saved to file.

After testing was complete, I noticed a further error in *arrangePiles()*. It overwrote the positions of the piles to their index in the *ArrayList*. I implemented a comparable to sort them in order before removing the gaps. This required the position attribute to be changed from a primitive *int* to an *Integer* data type.

Evaluation

Discussion

FR1 & FR2

I first created *Deck* and its *shuffler()* method. These create the deck object used to play the game. This is a private function called by a public *shuffle()* method, maintaining encapsulation.

The *getPackString()* function resulted in a null value due to the *shuffler()* function using a > symbol rather than <, and never adding cards to the list. The bound in the random method was also set incorrectly. Cards were repeated, as they remained selectable after already being used.

FR3 & FR7

Once *shuffle()* was ready, I attempted to implement the GUI. However, I was unable to do so correctly.

Instead, I moved onto the *deal()* function. This creates instances of *Pile* to represent the cards on the table.

I also created *showDisplay()* to output cards to the command line. This let me test the program without a GUI.

FR4 & FR5

I created *move()* to amalgamate piles together. This calls on two smaller functions – one to validate a move and one to re-arrange the piles.

I decided to have the card with the higher index considered the 'top' card, which remains after a move is complete. This allowed the last placed card to be easily moved onto adjacent piles, but resulted in an error. The 'top' card would remain at its own position rather than taking the place of the bottom card. I realised this at the end of testing, and as such, it is visible in some screenshots within this document.

move() takes parameters for the index of the 'top' card and how many spaces away the 'bottom' card is. The *jump* parameter is changed from 1 to 3 in requirement 5.

FR6

chooseMove() allows two indexes to be input, then passes them to the *move()* function. It will only succeed in the validation check if the jump is 1 or 3, either way.

FR8

auto() uses a 3-dimensional array to store an array of jumps and another of indexes for the 'top' card.

addMoves() iterates through the *piles* ArrayList. It calls *checkMoves()* to find valid moves for each, using a jump given by parameter.

This function is run up to two times. It first checks for moves with a jump of three. If none are found, it searches for adjacent moves.

If only one move is found, it is performed automatically. If multiple are found, *moveList()* is called to allow the user to select one.

At first, the first card in the list was always selected, and many cards were missed out. This was due misuse of the *.length()* function in relation to a 3-D array. While I assumed it would give the length of the array at a specified index, it actually gave the amount of 2-D arrays held.

I created *checkLength()* to iterate through one of the lists and add one every time it sees a value which is not the default.

FR9

I made the *load()* function before the *save()* function to ensure I knew what format the data should be in to load correctly. For testing, I used a manually created text file filled with names and scores.

To display loaded scores in a leaderboard, I created the *Score* class. I had previously tried keeping scores in other formats, but this made sorting difficult.

I implemented saving through *AddScore()*, which collects the user's name and points value. Points are calculated by taking piles remaining from the *MAXDECK* value, making the top score 51 for a deck of 52.

Creativity and Innovation

To expand on the existing features of the project, I made the patience game playable through only the GUI.

I first attempted this using a new class named *GUICard*. This was intended to make the static cards into draggable objects. However, I found it easier to add the events required to the existing *ImageView* objects. I used a new function, *Dragging()*, to assign these when drawing the cards.

Initially, I attempted to ensure cards were not dropped onto themselves by validating via the source image, but this was inaccessible once it was used by an *ImageView*.

In addition, I was setting the target and source events with the same ID and function. Separating *dragging()* into two functions, one for source behaviour and one for target behaviour, resolved both issues.

I added an event to highlight cards when hovered over, along with an event that output a test string once a card was dropped onto another.

With these drag/drop events functioning correctly, I add a parameter to the *drawCards()* function to pass a String value which is assigned to each *ImageView* object as an ID attribute. This String holds the value of the given card.

I also add three attributes to *CardTable*. Two strings to hold card data, and one Boolean to signify whether or not a move has been made.

The *Game* function can intermittently check this Boolean and take the card data from the strings once it becomes true.

I add a new function definition for *move()*. This version takes cards as parameters and translates them into their positions to be used by the standard *move()* function.

I add dealing functionality to the graphical menu via clicking the base card. I also automatically shuffle the deck before the beginning of the game.

A strange error occurs. When the *moved* boolean is set, the function does not recognise it, except when watched through the debugger.

It appears that the while loop is ignored if empty of code. I add a sleep statement to rectify this, however, responsiveness is reduced to stop the program using too much memory.

At first, the click action interfered with the dragging. I placed a check to ensure the action occurs only for the cards with clicking functionality, allowing the standard cards to be dragged unimpeded.

To make the game better suited for its new interactions, I replace the HBox with a FlowPane. This is a good format for dragging.

I add new buttons for quit, leaderboard and auto to perform their respective functions.

To adapt *auto()* for the GUI, I replace *moveList()* with *hintHighlight()*, which blurs all cards except the ones which can currently be moved.

I add a new scene with a text input for saving. This was difficult to achieve in the *cardTable* function so, as it had little relation to any of the other display functions, I moved it into a new *Name* class.

With this complete, every function is present within the GUI version, making the entire project adapted to its new drag/drop format.

What was difficult

At first, I didn't understand how *CardTable()* was used to create the GUI. I attempted to place the *Game* function calls outside of the start function, with only the *CardTable* functions in it. This resulted in the cards being displayed only once the game ended.

I believed the *Application.launch()* function was what drew the cards, when I should have used *cardDisplay()*. Once I realised how to use the threads, I created *showGUI()* to pass the correct cards to *cardDisplay()*.

I also had difficulty storing scores for the leaderboard. I attempted to use an ArrayList, but couldn't create a practical function to sort it while keeping them matched up with the names.

I decided to create a new class to hold both as attributes. This allowed me to sort by points with *Collections.sort()*.

The *Name* class was difficult to implement. Even after I moved the input menu code away from the *cardTable()* code which it was clashing with, I seemed unable to return a name value. This turned out to be due to missing brackets in the *getName()* function calls.

What remains to be done

While in the GUI menu, aspects such as move failed notifications and the leaderboard are still output to the command line. The game could feel more cohesive if all output was within the GUI window.

The current method of receiving input from the *cardTable* relies on a busy-waiting loop. This would work better given a more appropriate function, such as *wait()* and *notify()*.

Mark

All functional requirements are complete and without error, along with in-depth documentation. JavaDoc comments are present on all necessary functions, as well as regular comments to help with understanding the code.

I have surpassed the functional requirements in creating a drag/drop menu which is more intuitive for a card game than the command line.

I would therefore award myself a mark of 75-85%