ECM2414

A Card Game’s Simulation’s Report

# Design Choices

The first decision to make was the alternative strategy for the players to employ as instructed in the assignment:

Our player strategy is very similar to that of the strategy given, however when the hand is first given to the player it is checked to see which card denomination occurs most in each player’s hand. The card denomination that occurs the most in each player’s hand is taken as the card preference. If there is more than one, then the first card denomination that occurs the most is taken as the card preference.

For example, if player 1 has three 4s in his hand at the start and requires four 4s then this player will try to collect 4s resulting in him to be more likely to win on the majority of occasions. In contrast, if the suggested strategy is chosen then player 1 will discard all of the 4s to obtain 1s which may not even be in the initial pack of cards. Our strategy removes the potential of having a player who can never win because the card they are after is not in the initial pack and reduces the potential of having a player who can never win because there are not enough cards of the denomination that he wants in the initial pack.

To allow the user to run another simulation if he wishes to do so once it has completed one, or alternatively exit the program.

We decided to pause and resume the program using event handling, where one thread is waiting for input from the user while the player threads go through the simulation and produce output.

If the user enters "p" to pause the game, the *wait()* method is called on the player threads, they then stay in the wait state until “r” is entered. Execution then pauses when each player has completed their current turn unless a player has won during these turns. In this case, the simulation will just finish as designed.

This has been implemented with two classes in the *CardGame* class called *KeyInputListenerClass*, which implements the interface *KeyInputListener* and the *KeyInputGenerator* class, which runs as a thread and listens for users input. It adds and removes objects of type *KeyInputListenerClass* to be added and removed.

If “p” is entered by the user during the execution of the simulation then the *KeyInputGenerator* fires a *KeyInputEvent* (detailed in a separate class file). The user will be notified of the pause, and *wait()* is called on the *player* threads. If “r” is entered while the game is paused, *KeyInputGenerator* will again fire a *KeyInputEvent* upon which all waiting Player threads are notified and they continue their execution.

For a number of members we had to decide whether to make them static or not. Whereas this was straight-forward for most of them, some required more thought. For example the value storing the strategy employed by the players: We decided that all players should follow the same strategy throughout the game. But with a different implementation this may not be the case, and each instance of the players could have their own strategy, so it's a non-static member of the player class.

Another decision in terms of both design and efficiency is the usage of switch statements and when to break cases. At two points we’ve used this kind of constructions (within the *getCardToBeDiscarded()* in *Player* and the end of the *run()*-method of the *CardGame*) we have arranged the case statements so there is no need to break as the second one may and need to be carried out after the first one, etc.

## Data Structures

For the *Cards* stored in our *CardDeck* class we have decided to use the a *LinkedList* data structure. This gives us all the functions we require, with low complexities for the most used functions (*add()* and *remove()*). *Get()* may have a higher complexity, compared to alternative structures, but as we only need to access the top and bottom cards of the deck, this is less relevant.

The methods we use are

* *add()* to add more cards to the bottom of the deck
* *remove()* to retrieve the card at the top of the deck and remove it.
* *isEmpty()* to check whether there are any cards in the deck
* *size()* in order to create an array which we use to write to the deck*n*\_output.txt files.

For the *Cards* stored in the player’s hand we decided to use an *ArrayList* as we require fast random access. The complexity to get an item from the array is much lower with only 1 comparison needed to retrieve the required element, compared to alternatives.

To work out how many times a certain card’s face is apparent in a player’s hand (for the custom strategy) we’ve used a *HashMap*. This gives us key-value pairs within which we store the cards’ face values and the number of appearances, respectively. We then choose to keep the card face value *key* with the largest *value*.

## Interactivity/Menu

For the user interactions we’ve decided to wrap the *keyboardInputs* and requesting console-outputs in do-while loops, in case the user’s input is invalid. This saves the user time when making mistakes as the app doesn’t just terminate or needed to be restarted.  
This is also makes sure that we have no ‘impossible’ values to deal with in the code later on, which may be cause for errors or exceptions.

All user input is passed through *toLowerCase()* to ensure that the indented operation is executed, even if, for example, the user is in caps-lock mode by accident.

As all user input is temporarily saved in the String *input*, we even treat single-character inputs as strings to avoid using another variable. This also allows for expansion to use word as commands, instead of single letters.

The pause/resume functionality is run within a separate thread as the player threads and the main thread output data to the terminal and files constantly.

This ensures that the keyboard input will still be read while the program is paused.

# Thread Safety

In order to make the programme thread-safe we have ensured that all methods accessed by multiple threads are *synchronized*.

Whenever there were different implementations of similar functions available, we have tried to assess thread safety appropriately when choosing one over the other, and have also taken into consideration the CPU and memory overhead this may involve.

For example, as described more thoroughly below, we've decided to use a *HashMap* over a *HashTable* because thread-safety was not a main concern at the point of implementation.

# Performance Issues

We’ve decided to choose a *HashMap* over a *HashTable* for the evaluation of a player’s hand when employing the second strategy mainly for performance issues. Whereas a *HashTable* is thread-safe and synchronized, this also slows down its performance significantly. Because our *HashMap* is only accessed by a single thread, there is no reason for thread safety and we can use the better performing *HashMap*.

Other examples of performance trade-offs have been explained in the sections above.

# Testing

In order to deliver an appropriate solution, thorough testing is a key task in the development of this and any other simulation.

We have developed a separate program to create random decks to test the application, which lets us specify the deck’s range of values, the amount of cards, the number of card repeats and whether to shuffle it or not. This enables us to test the system more efficiently and allows us to use a wider range of input data.

## Approach

The testing approach taken is the one of *grey box testing*. JUnit 4.0 is used for the *white box testing* of public methods and for the *black box testing* we have used the output files and terminal output generated by the simulation and cross-checked it with the user input and initial pack of cards whether the program does what is required.

## Assessment

|  |  |  |
| --- | --- | --- |
| Test series | Purpose of test series | Test strategy |
| 1 | Test the output of the program at different stages. This also tests the *isDeckValid()* method. | Black box testing |
| 2 | Test output to files; tests *appendToPlayerLog()*. | Black box testing |
| 3 | Testing that methods of classes return the correct results, also done in other tests without using JUNIT. | Unit testing |
| 4 | Testing that the program pauses (when “p” is entered by the user) and that the program resumes correctly (when “r” is entered by the user). Tests the classes and methods used for event handling. | Black box testing and Dynamic testing (debugging) |
| 5 | Testing that the player strategy 1 correctly simulates a game; the same is needed for strategy 2. However this must be shown to result in games finishing more quickly than using strategy 1. This must be tested using a variety of decks, number of players etc. Tests *hasWon()* and *getCardToBeDiscarded()*. | Black box testing |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Description | Input Data | Expected Result | Actual Result |
| 1.1 | Output produced when a user enters different values for k and n. | *“Five”*  *-1*  *0*  *0.7*  *1*  *2*  *“”* | Error  Error  Error  Error  Accepted  Accepted  Error | As expected  As expected  As expected  As expected  As expected  As expected  As expected |
| 1.2 | Output produced when a user enters different values for the strategy number. | *“One”*  *1*  *2*  *3*  *1.4* | Error  Accepted  Accepted  Error  Error | As expected  As expected  As expected  As expected  As expected |
| 1.3 | Output produced when a user enters a filename. | A file that doesn’t exist.  Deck with 2 \* k \* n rows.  Deck with more than 2 \* k \* n rows.  Deck with less than 2 \* k \* n rows.  Deck containing:  One non-negative integer in each row  A negative integer  An empty row  Two numbers in a row (*0 6*)  A zero  A floating point number (*0.5*) | Error  Accepted  Accepted and uses the first 2 \* k \* n rows for the initial pack.  Error  Accepted  Error  Error  Error  Accepted  Error | As expected  As expected  As expected  As expected  As expected  As expected  As expected  As expected  As expected  As expected |
| 1.4 | Check that the simulation runs again if “y” is entered and doesn’t if “n” is entered. At the end of each simulation for both of the questions. | *“y”*  *“n”*  *“N”*  *“Y”*  *“p”*  *“r”*  *“”* | Accepted and re- runs simulation.  Accepted and exits program.  Accepted and exits program.  Accepted and re- runs simulation.  Error  Error  Error | As expected  As expected  As expected  As expected  As expected  As expected  As expected |
| 1.5 | Check that if a Player has k cards of the same value when a hand is dealt, then they win and the simulation ends regardless of the Player’s card preference. | A deck which causes at least one Player to win immediately after their hand has been dealt where their hand contains k cards of the same value which is not equal to the Player’s card preference using either strategy. | The Player that is dealt this hand wins and no other turns commence. | As expected |
| 2.1 | Check that the player output files are populated correctly after each game. For example pick up is output before discard. | Multiple runs with different combinations of initial packs, number of players and number of cards in hand. | The data in all of the Player files are output correctly and in the correct order for all runs. | As expected |
| 3.1 | Tests *saveToFile()* method and addCard() method in the *CardDeck* class. Other methods in this class are tested during other tests.  This also tests that the Deck output files are correctly populated. | A test deck containing two cards (10 and 20). These were added using the *addCard()* method. Then the *getSize()* method was called on this deck where the value was checked using *assertEquals*. If the value returned by *getSize()* when called on this deck returned 2, then the cards have been added correctly. The *saveToFile()* method was then called on this deck. The file was then checked for the correct data. | Saves deck contents to file correctly and the *addCard()* method adds cards correctly. | As expected |
| 4.1 | The program must pause and resume whenever the user enters “p” or “r” respectively. Therefore this test is very important.  The above was checked using the output of the program and the Player thread states shown during debugging after *wait()* and *notifyAll()* are called on these threads. | “*p*”(Multiple times in multiple runs)  “*r*”(Multiple times in multiple runs) | Pauses all threads after their current turn has ceased when not in paused state, otherwise nothing happens.  Program resumes all threads when in paused state otherwise nothing happens. | As expected  As expected |
| 5.1 | To test strategy 1. This test was set up using a program that generated large decks as explained earlier. | Many combinations of different decks, different player numbers (between 1 and 1000), different hand sizes (between 3 and 20) using strategy 1. | All ran without error and output is as expected. Files are updated correctly. | As expected |
| 5.2 | Same as in test 5.1 but to test strategy 2. | Same as in 5.1. | Same as in 5.1. | As expected |

Besides the above, we have run the program and private methods of classes at different stages of development and after development. Due to the lack of further problems with inputs & outputs from that and the tests done previously (e.g. *isDeckValid()*) works as deduced from test 1.3) we assume that the program behaves as expected due to the tests carried out and explained above.