# Multithreaded Card Game

710048792 & 710005281

50:50

| Date | Time | Duration | 710048792 | 710005281 | Signed | Signed |
|---|---|---|---|---|---|---|
| 7/11 | 11:30 - 12:30 | 1h | Observer | Driver | 710048792 | 710005281 |
| 8/11 | 10:30 - 12:30 | 2h | Driver | Observer | 710048792 | 710005281 |
| 8/11 | 13:30 - 15:30 | 2h | Observer | Driver | 710048792 | 710005281 |
| 9/11 | 09:00 - 10:00 | 1h | Driver | Observer | 710048792 | 710005281 |
| 10/11 | 14:35 - 15:35 | 1h | Observer | Driver | 710048792 | 710005281 |
| 12/11 | 12:00 - 15:00 | 3h | Driver | Observer | 710048792 | 710005281 |
| 13/11 | 10:00 - 12:30 | 2.5h | Observer | Driver | 710048792 | 710005281 |
| 15/11 | 16:00 - 17:30 | 1.5h | Driver | Observer | 710048792 | 710005281 |
| 16/11 | 15:00 - 17:00 | 2h | Observer | Driver | 710048792 | 710005281 |
| 18/11 | 10:00 - 11:30 | 1.5h | Driver | Observer | 710048792 | 710005281 |
| 20/11 | 13:00 - 15:30 | 2.5h | Observer | Driver | 710048792 | 710005281 |

Table 1: Development Log

# 1 Design Choices

## 1.1 Requirement Analysis

Our first phase of production was to analyse the requirements provided in order to understand the constraints we have to follow. We also attempted to predict the possible problems we may face in order to plan around them. We predicted that deadlocking would be one such issue. Since players are interacting with multiple decks, we could easily end up in a situation where every player is waiting on a deck which is locked by another player. Similarly, we predicted we could end up in a live lock situation, where one player is accessing its neighbouring decks so much that no other player can access them.

Furthermore, we observed that turns should be atomic. In other words, a player should either take a turn, or not take a turn. If a player were to be interrupted mid turn, we could end up with a situation where a card is lost in "limbo" as a player picks up a card but gets stopped before they can either discard it to a deck or add it to their hand.

## 1.2 Design

When it came to designing our game, we decided that first we wanted to get it working sequentially, one turn at a time. This meant that we could easily distinguish between bugs that were related to threading and bugs that were related to the core game logic. We also ensured that we strictly followed object oriented principles, especially that of encapsulation. If we made sure that each class had limited interactions with each other we could reduce the number of chances for accessing possible old or stale data from another class. However, there are some instances where we need to share information between classes. For example, the `Player` class must have a way of checking if someone else has already won. We decided the best way to do this was by accessing the current running `CardGame` instance through a static class variable `currentGame`. This does have the drawback that only one `CardGame` class can be running at a time (which is a reasonable assumption), and also makes our code simpler as we do not have to tie each Player instance with a specific `CardGame` instance.

During the planning process we discovered that the decks were the biggest bottleneck in the game running smoothly. Every deck needs to regularly provide its neighbouring players with an opportunity to discard as well as an opportunity to pick up. For our decks we decided to use a queue, since this matches a deck's typical first-in first-out nature. We realised that if we locked a deck when a player was picking up a card this would prevent another thread from discarding to the same deck at the same time. However, as long as there is more than one card in a deck, players should theoretically be able to pick up from and discard to the same deck simultaneously without accessing the same data. We just needed to use two separate locks, one for the head and one for the tail. This feature is provided in a built-in java class called a `LinkedBlockingQueue`.[1] With a rough outline of how our code was going to function, we drew up a basic class diagram with a rough overview of potential methods and attributes.

## 1.3 Implementation

Our implementation phase was quite straightforward due to our detailed planning and foresight. Thanks to our class diagram, we managed to quickly produce a minimum viable product. We were able to implement a sequential, non-threaded, version of our game without much difficulty. Through our planning, we knew to use `put()` and `poll()` when adding to and removing from decks. This is because these functions will block the current thread (indefinitely or for a set period of time) until the operation can succeed.[2] This blocking behaviour allows a player to wait for another player to add a card to a deck if it is empty. We also tried to make our code as efficient as possible by reducing our functions complexity as much as possible. For example in our dealing function we chose to use modulo

[1] `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedBlockingQueue.html`
[2] `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html`

arithmetic to avoid nesting multiple for loops. Once we were happy with the sequential version, and had ironed out a few game logic related bugs, we could move on to making our game threaded.

To make our sequential game threaded, we implemented a run function so that our `Player` thread could implement the `Runnable` interface. This run function checks that a player has not already won, then they take their turn. If they win, they update a flag in the main class in order to notify the other players. Once the main thread has created a thread for each player, it then starts all of them. The main thread will then check if each thread has terminated, and wait for it to finish if not. Once every thread has terminated, our main thread then creates the final logs. Thanks to our careful planning and anticipation of the issues we may face we managed to make our code from completely sequential to threaded and thread-safe within 20 minutes (much to our surprise).

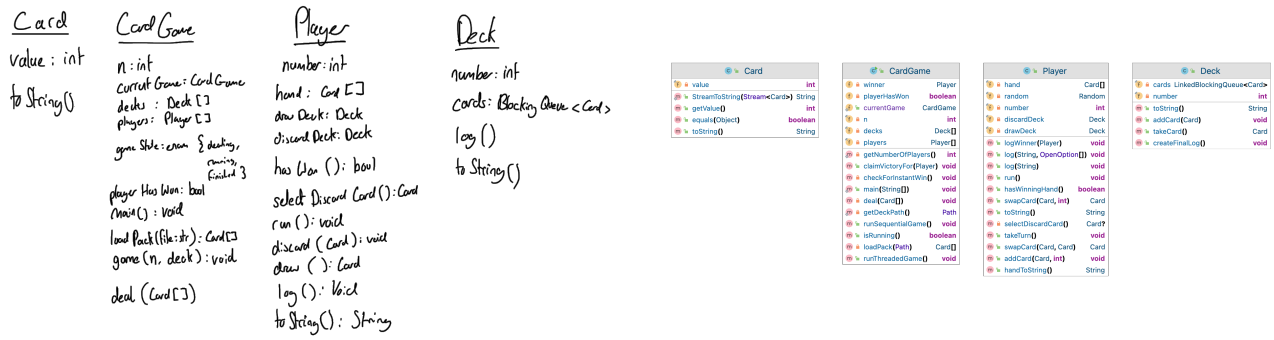


Figure 1: Hand Drawn Planning Document & Final Class Diagram

Our final class diagram looks a lot like our planned one, give or take a few extra methods that were created for better readability and reusability (see figure 1).

## 1.4 Known Performance Issues

After some basic profiling we've identified a few areas where our game is inefficient. Currently our logging solution writes to a file every time we log an event. Since IO operations take a very long time this significantly slows down each of our threads. This performance hit is significantly noticeable since there are multiple logs per player turn and multiple threads logging player actions. A better solution would be to use a `BufferedWritter` to save logs to a temporary buffer before appending them to a file. This way we could have a high number of writes with a low number of actual IO operations, leading to better performance. However, actually implementing this solution may overcomplicate what should be a simple logging file for a card game. We also chose not to implement a BufferedWriter as we wanted to have the most up to date logs as possible in the event of a crash.

Additionally, it can be noticed from the output files that often one or two players may have more turns than another. This can be seen from the fact that they have longer output files compared to other players. Unfortunately, not much can be done to solve this issue as it is due to how resources are allocated to threads. We could implement checks so that each player has the same number of turns, but this would essentially eliminate any performance benefit of threading since players would essentially be running synchronously, just across different threads. Therefore we would have the high cost of creating threads without any of the benefits of parallel execution. This issue is not significant enough to warrant any changes to our design. Through our testing we've only noticed an approximately 10% variance in the number of turns each player has, which we have determined is an acceptable range for the scope of this project.

# 2 Tests

When testing our program, we decided to use the JUnit 5.x framework as it provided an efficient way to structure our tests using the fundamental building blocks of unit testing. We separated our tests according to the class that they are primarily testing.

## 2.1 TestUtilities

We created a `TestUtilities` class that holds methods and functions that we used throughout multiple tests to avoid code duplication. For example, we have a public `clean()` method that deletes all the current deck or player output files that end with `'\_output.txt'`. This is useful in combination with JUnit's `@BeforeEach`, so that we know the current test logs are not influenced by a previous one. We use `TestUtilities` to further improve how we test logs by writing functions that compare the contents of files to each other.

Another issue we encountered in our overall testing was that we could not access private methods or private object fields. This was a significant problem since we wanted our code to remain as encapsulated as possible, but at the same time we also wanted to test our private function. Therefore, we created functions within our `TestUtilities` that allowed us to do so. We made use of Java reflector to create a `getPrivateField()` function, which takes in an object and a string of the field that we want to access. Likewise, the `runPrivateMethod()` takes in an object and its private method with parameters for the method, and returns the result. These have both proven incredibly useful throughout our testing and have helped us catch a few bugs in private functions.

## 2.2 CardGame

The initial thing we tackled when creating the `CardGame` itself was ensuring that any invalid inputs were thrown straight away. We tested this using parameterised tests provided by JUnit as it allows us to execute the same test multiple times with varying arguments that are specified in the value source and then passed through to the test method as a variable. We first tested for invalid player number inputs by using negative numbers or zero. Then we tested for a variety of invalid deck paths.

One issue we encountered quite early on in our programming was how we handled the `deal()` method. The requirements specified that the cards must be dealt to the players first, and then dealt to the respective player decks both in a round-robin fashion. However, an issue we encountered that complicated this was the fact that we hold the cards in the player decks in a queue rather than in a list as it would aid us in the deck's future use cases. This meant that we had to handle the dealing to players and to player decks differently in our `deal()` method, therefore making it an important factor to test. Furthermore, the players' hands are constantly changing as the game goes on, so it was necessary to only check the expected hand against their initial hand, rather than one of their hands during the game. In `dealTest()`, we used a premade initial pack which allowed us to preemptively predict which cards should be placed in the players hands and decks, and the position they should be placed in. We used the `assertEquals()` overloading method provided by JUnit when testing the player's initial hands, which allowed us to compare the expected value we already calculated with the actual value of the cards in each hand produced in the code under the test. Similarly, when it came to the player decks we used `assertTrue()`, but this time compared the expected deck log with the actual log outcome. As we had created this test simultaneously with the `deal()` method itself, we discovered immediately that the player hands were dealt correctly but the player decks weren't quite right. Thankfully, after some tweaking, all the tests were passed.

Another factor we considered when creating our tests was the bias element involved with using premade decks for our testing. However, using an entirely random deck meant that typically there weren't enough cards of the same value for any player to win, particularly because each player only looks for cards that match its value. Therefore, we wanted to find a happy medium by creating a deck generator method that would generate a random deck where each player also has the potential to win. We decided that this would make our tests much more fair because it would entirely remove

the bias that comes from self made decks. We achieved this by creating a deck in stages. Firstly, for each player we added 4 cards of its value to the deck - this meant that there were enough of each player's desired card in the pack for each of them to win. Then, for the second half of the deck, we added random card values using `random()`. Finally, we shuffled the cards in the deck using `Collections.shuffle()` in order to randomise the order that these cards are currently in. Randomly generated decks also provided us an avenue to stress test our project. As part of our own testing we ran our game overnight, playing thousands of games back to back, to see if we could by chance find a scenario where there was a crash or an exception was thrown. Consequently, we successfully removed the bias in our `limitTest()` and were able to test the game with a randomly generated yet valid card pack, greatly improving the scope and quality of our tests, whilst eliminating the task of having to manually create decks that have the potential to win for our tests.

We wanted to test how the logging works once a game has finished. We did this by comparing the final logs for player hands and decks to files that we created ourselves that consist of what the final logs should actually be. When actually executing the game, we decided to run it sequentially rather than executing the threaded version as the outcome is determinate, giving us confidence that our pre-made final log files are correct.

### 2.3 Player

When testing various functionalities related to player hands, specifically when a hand has won or when a card is discarded from a hand, we found that it was useful to have some player decks and results handy to compare against when testing. Therefore, we created a `hasWonGenerator`, which stored an array of various player hands, and then stored an array of boolean values detailing whether the hand has won or not. These two arrays were then returned as a stream of arguments. Furthermore, we created a `selectDiscardCardGenerator` that also stores an array of player hands, but this time also stores an array of integer values of the card that the player should discard that were then also returned as a stream of arguments. Therefore, in our future tests we were able to parameterise them with the arguments from these functions in order to test the discarding and winning functionalities of player hands.

When a player discards a card, it randomly selects a card from its hand unless the card is the one it desires. However, this makes it particularly difficult to test as we cannot predict which card it will choose, limiting our ability to test its behaviours surrounding the discarding functionality. Since each game was not deterministic, it was hard to tell when our program diverged from its intended behaviour. Therefore, we decided to seed the players' random choices by using the player number as the seed when the player is discarding a card. This essentially allows us to predict a player's random choices, based just on their number.

Another element of the player class we needed to test were the logs. A player can both create a log for itself, and can then append this log as its hand changes throughout the game. We tested both these elements of logging by using `assertTrue`. We start by asserting whether the outputs are a file, and then asserting whether the length of the file is larger than zero if it has just been created, and then asserting whether the length of the file increases as more lines are added to the log.

It was important that we tested the functionality of a player swapping out a card in its hand, as we use overloading polymorphism in order to allow the player to swap the card either by its object or its index. We therefore tested this by using `assertEquals` and ensuring that the `swapCard()` function returns the right card regardless of whether it takes in the card itself or the card index.

### 2.4 Deck

We chose to use the `@BeforeEach` annotation when testing the decks in order to first create a deck and then add some cards to it before each test. Moreover, we used the `@AfterEach` annotation to clean the files by calling the `clean()` method from our `TestUtilities` class. These annotations aided us when testing things like adding and taking cards from a pre-existing deck. When testing the adding

of cards to a deck we assert that the deck log has the same cards as the ones that were added in the `@BeforeEach`. When testing how a deck takes a card, we assert that the value of the card added in the `beforeEach` method is the same as the card taken - we do this four times for each card in the deck. We further assert that the output log file once all cards have been taken reflects that there are no cards in the deck.

## 2.5 Card

We found that there were limited things that we could test about our `Card` class due to the fact that it mainly stores information about the card rather than doing much data manipulation, which primarily happens to a card when it moves from a player's hand to a player's deck or vice versa. It essentially acts as a record class.

Something that we did test was the functionality of the `equals()` boolean return function, and we did this by using a combination of `assertEquals` and `assertNotEquals` in order to determine whether cards are stored properly. We created cards with integer values and then compared them to a variety of correct and incorrect values, and also null values and strings in order to ensure that they are only considered equal when they have the same value. It is important that our equals function works since there may be multiple cards of the same value in a pack, meaning we can't just check that the object references are equals.

## 2.6 Coverage

| Element | Class, % | Method, % | Line, % |
|---|---|---|---|
| all | 100% (12/12) | 91% (68/74) | 80% (308/382) |
| Card | 100% (1/1) | 100% (5/5) | 100% (13/13) |
| CardGame | 100% (3/3) | 78% (11/14) | 72% (81/111) |
| Deck | 100% (1/1) | 100% (5/5) | 83% (10/12) |
| Player | 100% (1/1) | 100% (13/13) | 90% (50/55) |

Figure 2: Test Coverage

Looking at the test coverage of our code, it is very good as we covered 100% of classes, 91% of methods and 80% of lines (see figure 2). However this does not necessarily guarantee that it all works, simply that it meets the requirements of our tests. The only aspects of code that our tests haven't covered are interrupt exceptions, IO exceptions and the static functions in `CardGame` such as the main, and asking for the player number or deck. This is because the best and easiest way to test these is by doing it manually and running the whole project. We did this using a black-box style of testing by inputting a variety of values for player number, and many different decks to see what the outcome was.