Code Design

When analysing the requirements of the game, we thought it was logical to model each element of the game as an object, using the object-oriented programming paradigm. This is because the Java programming language works well for implementing a well-designed, object-oriented system. Therefore, we created the *Player*, *Card*, and *CardDeck* classes to model objects in the game, and an executable *CardGame* class that runs the simulation after reading from standard input and instantiating the aforementioned classes. We also wrote our own exceptions to enforce the constraints of the game. To better document our design, a UML diagram is shown in *Figure 1* to represent each of the classes and how they interact with each other.

The *Card* class is used to model a single card in the game. It is a very simple class, with only one attribute: the cards *value*. In line with the specification, its only constraint is that *value* must be a positive integer. If a negative/zero integer is passed into the *Card* constructor, a custom *InvalidCardException* is thrown. As part of our design choice, we chose to omit getter/setter methods for this class. This is because we didn't need them (Cards only need to be instantiated once, and we made the attributes package-private), and having redundant methods wastes storage. We also needed to keep the *Card* class thread-safe.

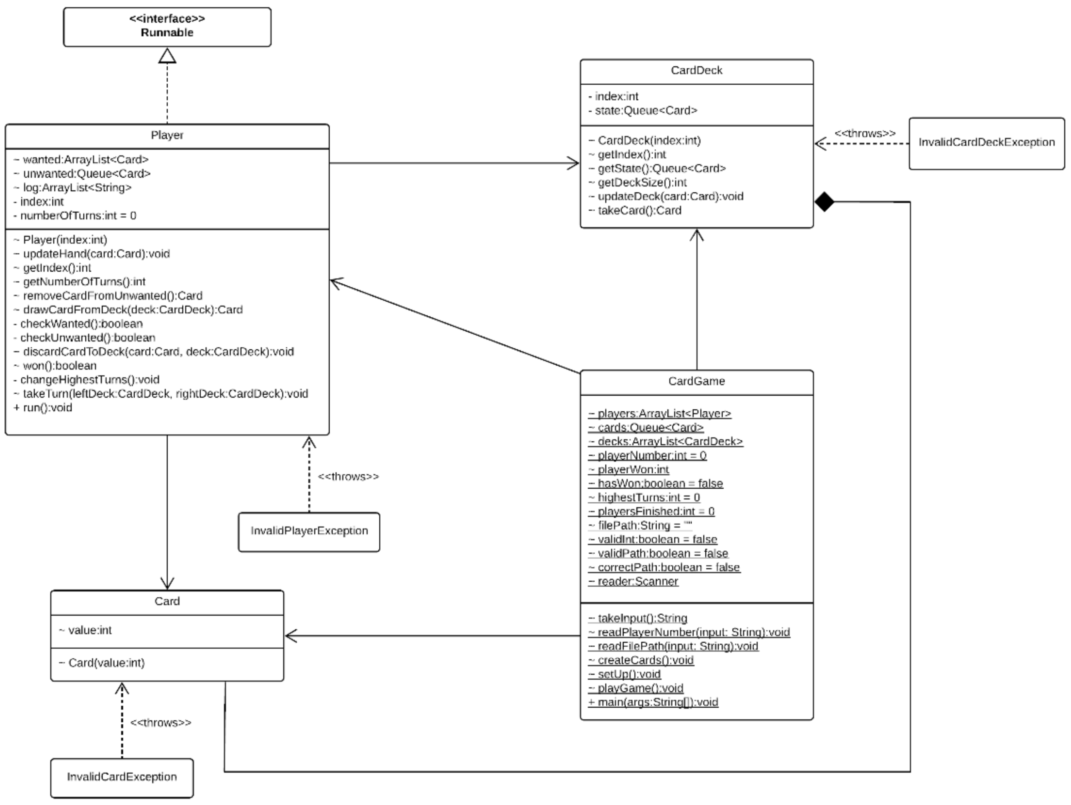
The *CardDeck* class models a single deck in the game. Each deck instance has two attributes: *index* and *state* (contents). We chose to represent *index* with an int, and *state* with a Queue of *Card* objects. We chose these types as they allowed us to enforce the constraints of the game. Decks must be numbered 1 to n, so an int is used and a custom *InvalidCardDeckException* is thrown if a negative/zero integer is passed into the constructor. Decks exhibit a first-in first-out (FIFO) principle in the game, because players draw from the top of a deck (head of the queue) and discard cards to the bottom of a deck (back of the queue). We chose to write simple package-private getter methods *getIndex*(), *getState*(), and *getDeckSize*(), as *Player* objects will need to access the private attributes when playing the game. This also keeps our class well-encapsulated. Along with these, we wrote two synchronised methods *updateDeck*(*Card card*) and *takeCard*(). These are essentially equivalent to the standard queueing and dequeuing methods provided in the Java *Queue* interface, and are synchronised to prevent any race conditions or issues with stale data. Another design choice was to import only Queue and LinkedList from *java.util* instead of importing the entire package (*java.util.**), to save on storage.

The *Player* class provides the blueprints for an arbitrary player in the game. We chose to implement the *Runnable* interface rather than inheriting from the *Thread* class, because implementing an interface is far less expensive than inheriting from an entire class, and Java doesn't support multiple inheritance. Each *Player* instance has five attributes: *index*, *numberOfTurns*, *wanted*, *unwanted* and *log*. The *index* and *numberOfTurns* are both of type int. The *index* attribute is used in the constructor and its purpose is to identify different instances of *Player* objects, whilst *numberOfTurns* keeps track of the total number of turns taken by a given *Player* instance. We chose to include this *numberOfTurns* attribute so that we could implement a system in which all players take the same number of turns by the end of the game, regardless of who won. This aligns with the constraints of the game, as it ensures all decks contain 4 cards by the end of the game. The *log* attribute is an ArrayList of *String* objects, used to keep track of player behaviour by storing strings describing the actions taken throughout the game. An ArrayList was used over an Array because the duration of any one game is unpredictable (and dependent on OS/CPU architecture), so using a fixed-size array would be inappropriate. An integral aspect of our system design was the decision to represent the hands of players using two list types rather than one: a *wanted* ArrayList of *Card* objects to represent preferred cards held by the player, and an *unwanted* Queue of *Card* objects to represent any other cards held by the player. Representing a players hand in this way is what allowed us to implement an effective game-play strategy that also met the constraints of the specification. For example, using a Queue to represent unwanted cards allowed us to easily enforce the restriction that players must not hold non-preferred denomination cards indefinitely (preventing the game from stagnating).

The Player class features lots of helper methods for getting/checking/updating attributes, and three synchronised methods *drawCardFromDeck(CardDeck deck)*, *discardCardToDeck(Card card, CardDeck deck)* and *changeHighestTurns()*. These synchronised methods are used in the implementation of *takeTurn()*, another synchronised method that actually carries out the turn of the player by drawing/discarding from appropriate decks, and updating the *CardGame.highestTurns* attribute (used to ensure decks finish with 4 cards in each). We synchronised these methods to ensure we were able to make a players turn a single atomic action. This prevents any possible race conditions caused by players attempting to draw from and discard to a given *CardDeck* instance simultaneously. Our *run*() method implements the abstract *run*() method from the *Runnable* interface, and gets called when we construct threads with *Player* objects and call *start*(). Again to meet the specification constraints (and avoid *IndexOutOfBoundsException*), it utilises a while loop that ensures players only take a turn if they are able to draw from their left deck (i.e. if it's non-empty). Within this loop, a synchronised block checks if the player has won after taking their turn, breaking from the loop if they have. Finally, the current highest number of turns (stored in *CardGame.highestTurns*) is updated. After this loop has exited (i.e. after the game has been won), *log* is updated and any more necessary turns are taken so that all players have taken the same number of turns. Lastly, the *log* and information about the deck to its left are written to two files (so each player writes their own log and the decks log who has the same index, e.g. player1 writes player1_output.txt and deck1_output.txt). The design choice of storing log messages in an ArrayList of Strings throughout the game and writing them to files after the game has finished was intended to avoid lots of expensive I/O operations. This way we only have to write to a file once (after the game is over), rather than dynamically writing to a file throughout the game.

Our final class, the executable *CardGame* class, makes use of the above classes to run a simulation of the game. To do this, it has methods for reading and processing user input, instantiating the corresponding objects and spinning up the appropriate number of threads to simulate the game. Here we made the design choice to encapsulate these procedures, rather than writing them all in the body of the *main* method. This ensures that we can write unit tests for testing each element of the simulation, allowing for a more robust program to be developed. On reading input, we check the validity of the data provided, printing relevant errors when invalid input is given. For example, checking that the number of players is a positive integer, checking file paths are valid, and that pack contents comply with the constraints outlined in the specification.

Figure 1:



Test Design

In order to develop a robust program, we followed thorough *test-driven development* practises throughout our software development lifecycle. By writing tests before implementing the actual production code, we were able to minimise bugs, increase reliability, and eliminate any testing bias. Moreover, writing tests gave a clearer idea of how exactly we'd go about implementing the production code, greatly simpliying our development process and maximising our efficiency as a pair. This principle of *test-driven development* was an essential component of our system design, as it established program stability and effective case handling.

The first choice we needed to make with respect to testing design was which version of the *JUnit* framework to use. Whilst both allow us to write unit tests, we found the differences between the *3.x* and *4.x* distributions were minimal (with respect to the task at hand), the only key changes being syntax-related. For this reason, we chose to use the latest of the two frameworks: *JUnit 4.x.*

To measure the code coverage of our unit tests, we chose to use the *IntelliJ IDEA code coverage runner.* Here we had other choices such as *JaCoCo* or *EMMA*, but found that IntelliJ provided the simplest user interface and was the most convenient (built-in to our IDE of choice).

Before writing any tests, we thought carefully about the high-level design of our system (see *Figure 1* in *Code Design*). This ensured that we knew exactly how the classes were supposed to act, both in terms of structure and behaviour/relations with other classes. Subsequently, writing appropriate tests for each of the classes came with relative ease.

We set out to cover as much code as possible with our tests, with the intention of writing production code that was as stable as possible, and able to pass all test-cases. Although we aimed for a high method coverage and were able to write unit tests for the large majority, we had no hard target for code coverage. This is because we knew some aspects of the simulation would be very difficult to test, as they exhibit unpredictable behaviour and rely on console input.

For example, testing I/O operations such as console input or file output was particularly difficult. Testing thread behaviour also proved difficult, as there are lots of possible paths a thread can take through the system, depending on game conditions (such as pack contents). We also knew we wouldn't be able to test the *main* method of the *CardGame* class.

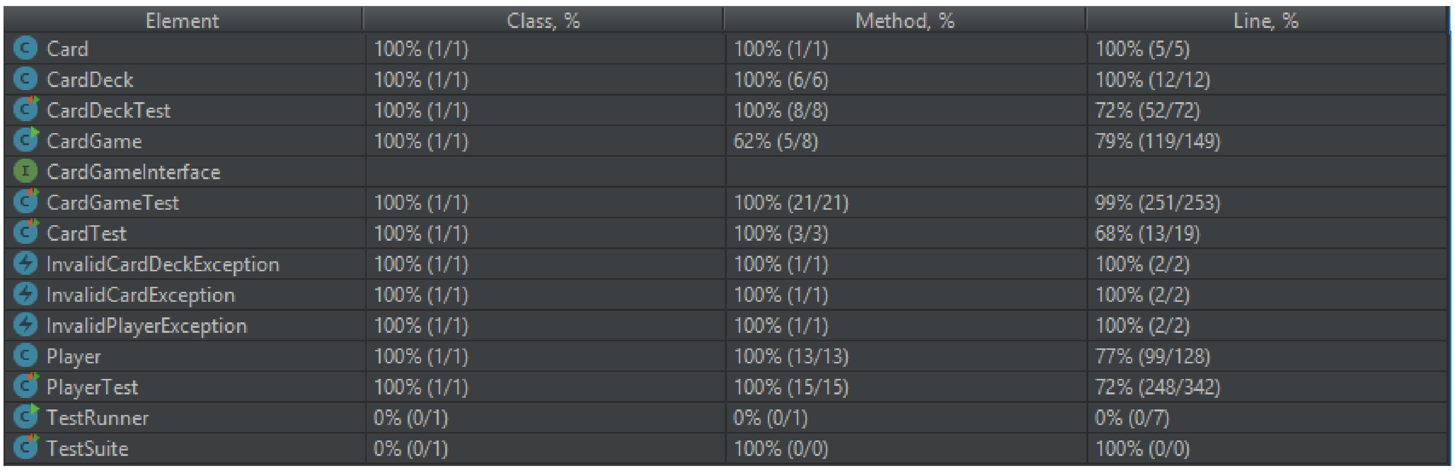
Despite not being able to test the aforementioned methods, we still attempted to maximise the code coverage for the sake of developing a robust system and testing as much as possible. To do this, at the design stage we made the decision to encapsulate the *CardGame* class, by breaking the simulation down into a series of method calls, such that minimal computation is carried out in the *main* method. This way we could thoroughly test each constituent part of the simulation. For example, in order to better test I/O, we isolated the user input into two smaller methods, *readPlayerNumber(String input)* and *readFilePath(String input)*, and called these from the *main* method (rather than implementing them both in the method body of *main*).

When writing unit tests, the idea is to test how methods (and constructors) react when given different inputs. So to make our tests most effective, we ensured that all edge cases were tested. For example, in *PlayerTest* we have a *testEdgeCasePlayer()* test that instantiates a *Player* object with an *index* attribute of 1, as this is the minimum *index* a player is allowed to be assigned according to the specification (positive integer). We also wrote tests to check our custom exceptions were being thrown where appropriate, examples being the *testNegativePlayer()*, *testZeroPlayer()*, *testNegativeCard()*, *testZeroCard()*, *testNegativeCardDeck()*, and *testZeroCardDeck()* tests. These tests check to see if the corresponding custom exception is thrown when invalid input is passed through the constructors of our classes, ensuring that they work as intended.

Beyond the design stage, we continually ran unit tests throughout the implementation and development of production code. This helped maintain a robust system that could endure a range of different test-cases. For example, if we made a slight change to the implementation, we would check the corresponding test method and ensure that it still covered all edge and corner cases. When developing the production code itself, methods were tested further in the form of debugging: utilising a trial-and-error approach to better understand the behaviour of methods. Whilst this isn't anywhere near as robust as writing an entire test suite, it still contributed to securing code stability and played a key role in our approach to *test-driven development*.

Thanks to our rigorous and diligent use of *test-driven development*, our tests were able to cover a large majority of our production code, as shown in *Figure 2*.

Figure 2:



Element	Class, %	Method, %	Line, %
Card	100% (1/1)	100% (1/1)	100% (5/5)
CardDeck	100% (1/1)	100% (6/6)	100% (12/12)
CardDeckTest	100% (1/1)	100% (8/8)	72% (52/72)
CardGame	100% (1/1)	62% (5/8)	79% (119/149)
CardGameInterface			
CardGameTest	100% (1/1)	100% (21/21)	99% (251/253)
CardTest	100% (1/1)	100% (3/3)	68% (13/19)
InvalidCardDeckException	100% (1/1)	100% (1/1)	100% (2/2)
InvalidCardException	100% (1/1)	100% (1/1)	100% (2/2)
InvalidPlayerException	100% (1/1)	100% (1/1)	100% (2/2)
Player	100% (1/1)	100% (13/13)	77% (99/128)
PlayerTest	100% (1/1)	100% (15/15)	72% (248/342)
TestRunner	0% (0/1)	0% (0/1)	0% (0/7)
TestSuite	0% (0/1)	100% (0/0)	100% (0/0)

As part of our final performance assessment, we found the following known bugs:

- The specification states *'there should only be one player declaring it has won for any single game'.* However, when testing our program on multiple different computers we found one case where our program would declare that two players had won, printing both to standard output (terminal). Thankfully this bug is very rare, as we only encountered it once after running the program over 50 times on the same machine. The actual gameplay and output files are still accurate and behave as they should (only one output file will declare that it has won). This is likely due to unexpected thread behaviour, and differences in OS/CPU architecture. On reflection, next time we will spend more time in the design stage ensuring that we account for these differences, to minimise the dependencies of our multi-threaded application.