ECM2414 Pair Programming Coursework

**Thomas Newbold** – 71000126

**Steven Jangcan** – 710042102

Weighting 50:50

Development Log

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Date | Time | Duration (hh:mm) | Roles | |
| 71000126 | 710042102 |
| 08/11/2022 | 11:30 | 01:00 | Observer | Driver |
| 08/11/2022 | 12:30 | 01:00 | Driver | Observer |
| 12/11/2022 | 10:00 | 01:30 | Driver | Observer |
| 12/11/2022 | 11:30 | 01:30 | Observer | Driver |
| 12/11/2022 | 01:00 | 01:00 | Driver | Observer |
| 12/11/2022 | 02:00 | 01:00 | Observer | Driver |
| 15/11/2022 | 11:30 | 01:00 | Observer | Driver |
| 15/11/2022 | 12:30 | 01:00 | Driver | Observer |
| 15/11/2022 | 02:30 | 00:30 | Observer | Driver |
| 18/11/2022 | 01:30 | 01:00 | Driver | Observer |
| 18/11/2022 | 02:30 | 01:00 | Observer | Driver |
| 18/11/2022 | 03:30 | 00:30 | Driver | Observer |
| 20/11/2022 | 04:30 | 01:00 | Driver | Observer |
| 22/11/2022 | 11:30 | 01:00 | Observer | Driver |
| 22/11/2022 | 12:30 | 01:00 | Driver | Observer |

Design

The overall (high-level) design for the system can be seen in this UML diagram *(Fig. 1, left)*. Most classes are used by the CardGame class, within its main loop.

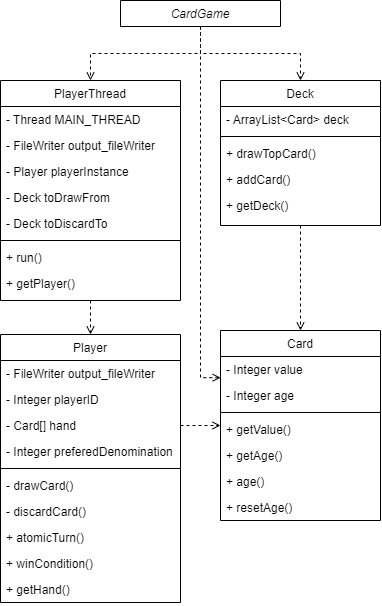


Figure 1: UML Class Diagram

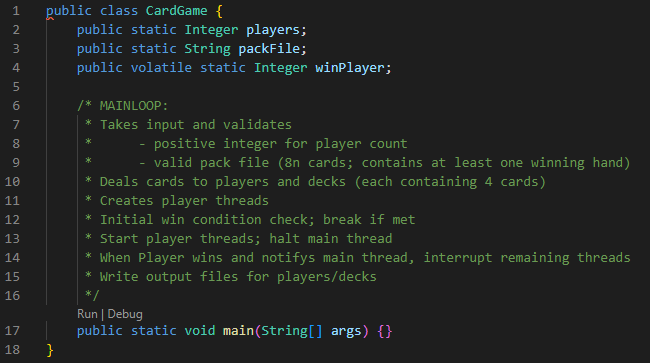


Figure 2: CardGame class outline

The main CardGame class *(Fig. 2, shown above)* deals with input validation, dealing the cards, player thread creation, and initialising the associated output files.

Provided below are the outlines for supporting classes, with docstrings describing each function to be implemented. Instance variables have been confined by using private variables and getter methods.

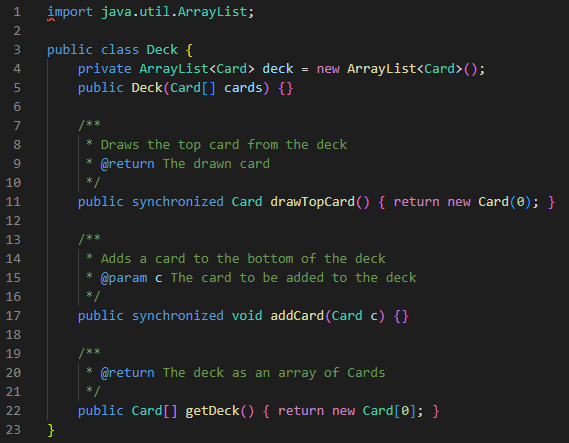


Figure 3: Deck class outline

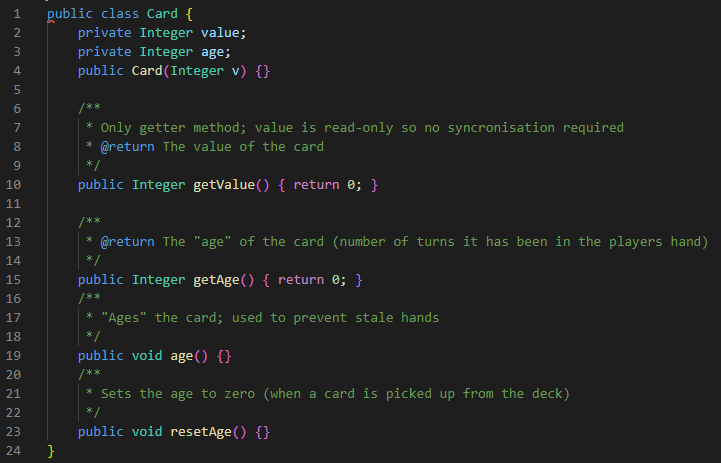


Figure 4: Card class outline

The Deck class *(Fig. 3, above, left)* has been synchronised, as it is likely that different player threads will be adding/removing cards simultaneously. This is the only data structure that will be accessed by multiple threads (at the same time; instances of Card will be passed between threads but never modified during this time). The Card class *(Fig. 4, above, right)* simply holds a value, and an “age” (which increments each turn a card spends in a player’s hand) – this value is used when choosing a card to discard, and prevents stale hands.

The Player class *(Fig. 5, left)* defines drawCard and discardCard methods, which will be contained within the atomicTurn method. This effectively makes the drawing/discarding of a card an atomic action, so outside of this, the Player’s hand will always have a constant number of cards. It also contains a method to check whether the player’s current hand is a winning hand.

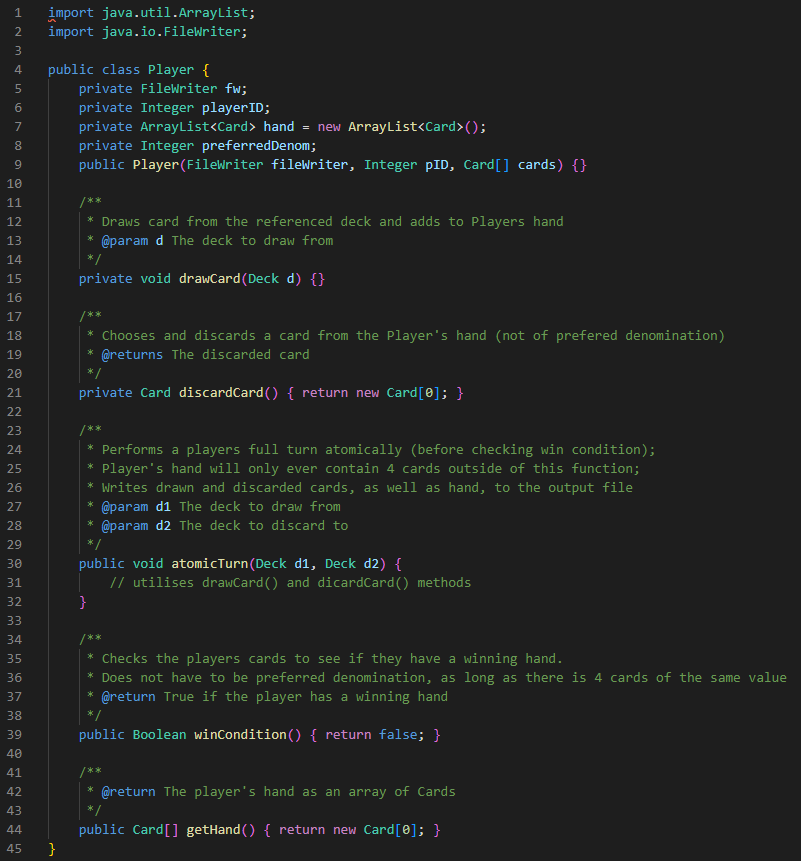


Figure 5: Player class outline

Each instance also contains a FileWriter instance – this is to allow player actions to be documented in an output file.

The PlayerThread class *(Fig. 6, left)* will encapsulate an instance of the Player class, and implement the threaded behaviour, calling atomic turn method and checking the win condition, notifying the main thread when this is met.

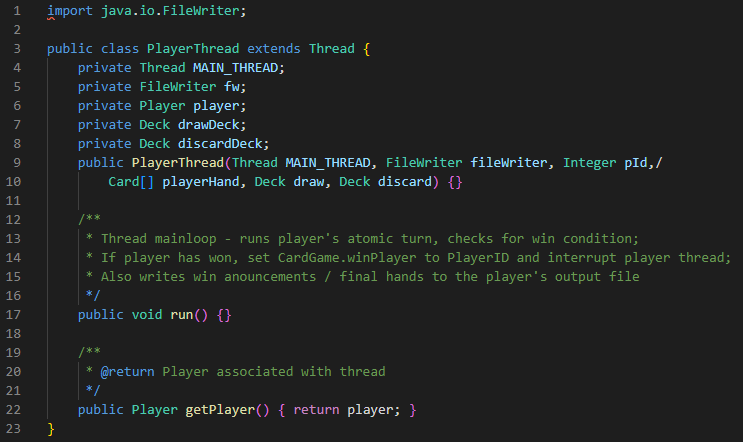


Figure 6: PlayerThread class outline

Testing

For our tests, we will be using JUnit 4 (specifically 4.13.2).

Each of the supporting classes has a range of unit tests written for them to test their methods and behaviours. CardGameTestSuite class acts as a wrapper for the rest of the tests, running all classes.

For Card, this is simply getters and setters – the constructors are tested within the getter test. Certain duplicates have also been introduced to double-check for any unexpected values.

Similarly for Deck, however this class also encapsulates drawing/discarding functionality which must be tested. As cards are always drawn from the top and discarded to the bottom, it is easy to check the value of card outputted, and the state of the residual deck – this is all deterministic behaviour.

For Player, we set up some mock decks/hands with known card values, as this creates somewhat predictable behaviour (except for card discarding, which has a level of randomness – this is addressed in a separate test which uses the card’s “age” attribute) and allows us to test the atomicTurn() method by asserting the number of cards in each deck/the players hand after the turn, and any cards of preferred denomination which should be retained *(see Fig. 7, below)*.

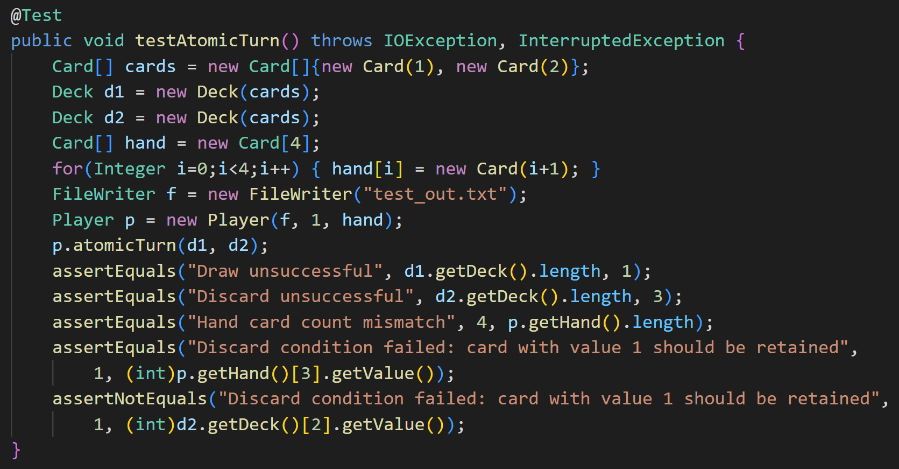


Figure 8: testAtomicTurn()



Figure 7: testAtomicTurn\_discardByAge()

In a variation on this test, the ages of cards within a deck/players hand are manipulated such that a known card is discarded each time *(see Fig. 8, left)*. Also, some cases contain assertions to check if preferred denominations of card (matching the player’s id) are retained.

Testing the win condition was also key, so various concrete winning and losing scenarios are checked in a unit test *(see Fig. 9)*, as well as induced winning states *(testAtomicTurn\_Win() is not documented here for brevity)*.

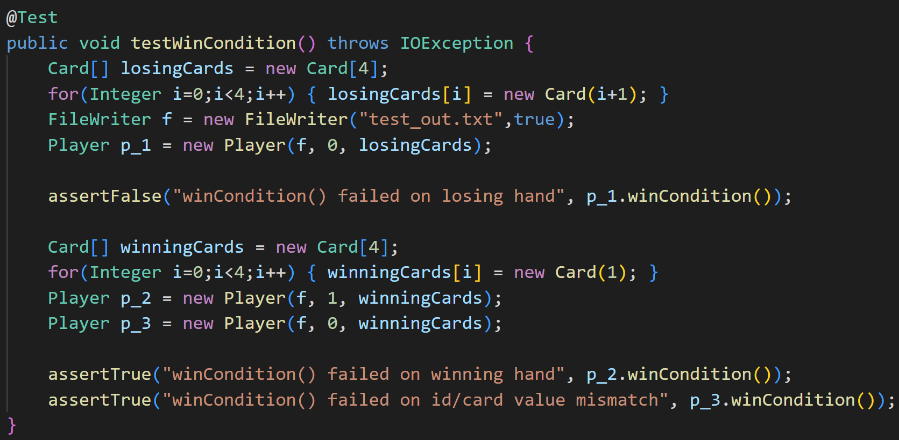


Figure 9: testWinCondition()

Reflection is used to individually test both the drawCard() and discardCard() methods by assertions on the residual hands using a known set of input cards *(Fig. 10 and Fig. 11, below)*.

The PlayerThread tests lead on from these. Again, this class contains a getter method which must have an associated unit test. Outside of that, the condition being checked for the run method is that the contained player’s hand must contain 4 cards throughout, so the thread is run for 500ms, and the size of the player’s hand is checked following that *(see Fig. 12, below)*.

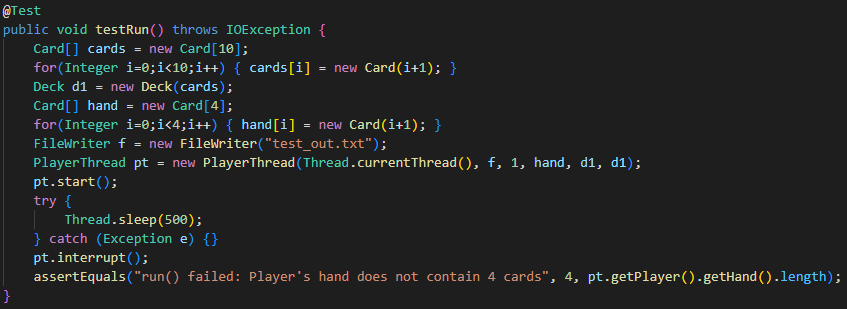


Figure 2: testRun()

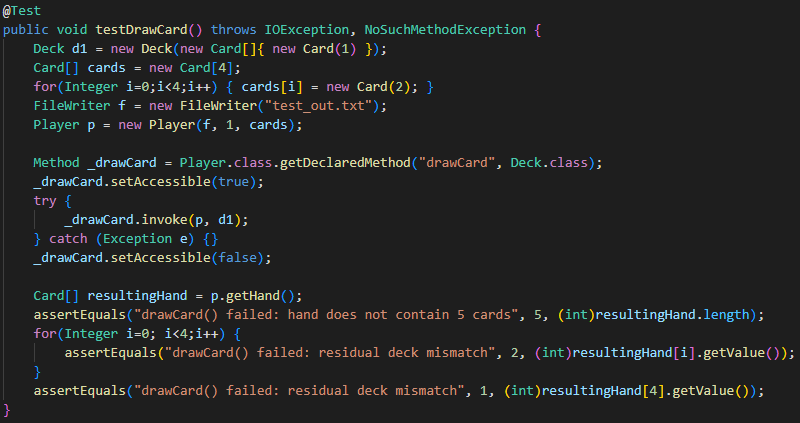


Figure 10: testDrawCard()

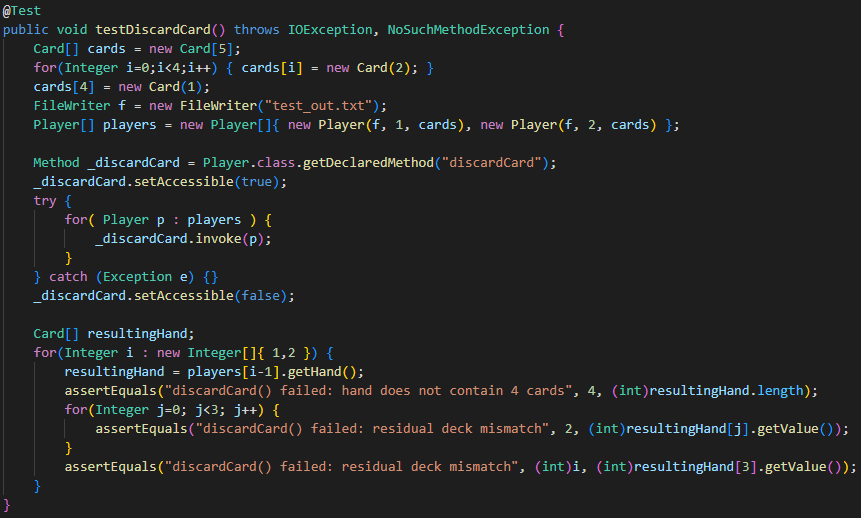


Figure 1: testDiscardCard()

The Player strategy is externally tested by a unit test here which runs the thread for a further 500ms (or until interrupted) using a deck containing only a winning hand – the resulting state of the program is such that a winning condition for the player is effectively guaranteed, and this is checked using assertions as usual *(see Fig. 13, left)*.

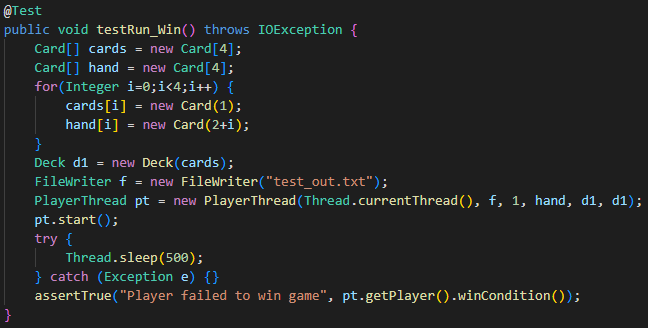


Figure 13: testRun\_Win()

We believe that the code coverage for our supporting classes is effectively 100%.

As CardGame only holds the main code, there are no methods to test per-se. However, we can run the main loop, and check the residual static variables, as this will verify a significant proportion of the code. By doing this, we also have a chance to check the input verification: the player count input is tested on letters, negative, zero and positive values; the pack filepath input is tested on non-existent, as well as valid/invalid files *(see Fig. 14, right)*.

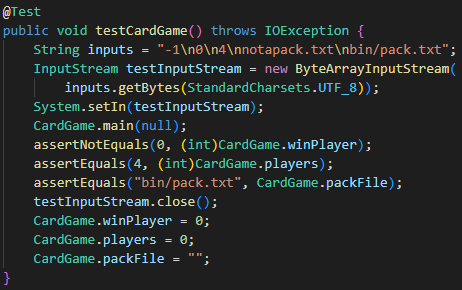


Figure 14: testCardGame()