**Project – Cover Page**

Abigail Hinton – 720025014 50%

Giulia Elizabeth Brown – 720047038 50%

**AGREED DIVISION IS 50:50**

**Development Log**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Date | Time | Duration | G. Brown | A. Hinton | Signature |
| 31/10 | 10:00 | 2h | Design discussion | Design discussion | 720047038  720025014 |
| 06/11 | 15:30 | 2h | Driver | Navigator | 720047038  720025014 |
| 08/11 | 12:30 | 1.5h | Navigator | Driver | 720047038  720025014 |
| 11/11 | 14:00 | 3.5h | Driver | Navigator | 720047038  720025014 |
| 12/11 | 14:00 | 2h | Navigator | Driver | 720047038  720025014 |
| 15/11 | 10:00 | 3h | Driver | Navigator | 720047038  720025014 |
| 18/11 | 14:00 | 3h | Navigator | Driver | 720047038  720025014 |
| 21/11 | 12:30 | 4h | Driver | Navigator | 720047038  720025014 |
| 22/11 | 17:00 | 2.5h | Navigator | Driver | 720047038  720025014 |
| 23/11 | 14:00 | 1.5h | Driver | Navigator | 720047038  720025014 |
| 28/11 | 12:30 | 4h | Completing report | Completing report | 720047038  720025014 |

**Production Code Design**

The code we have written simulates a multi-threaded card game where an executable main method receives from the terminal the number of players that will be involved in the game and the location of a valid file containing all the necessary cards to distribute to the players and to the decks. Once the cards have been delt the game can begin. At this point the main method will start all the player threads concurrently.

**Signal classes**

We have created three signal classes that dictate the game, and make sure it proceeds correctly: *WinnerSignal*, *ReadyToPlaySignal*, and *StartGameSignal*.

A *WinnerSignal* object is used to notify the main thread when there is a winner. Meanwhile, the main thread will be waiting on this object.

A *ReadyToPlaySignal* object contains a counter that initially is equal to the number of players, and then it is decreased every time a player calls its method. When the counter is zero, it means that all the players are ready to play, and the signaling object notifies the main thread. This object is important because it makes sure that all players check their initial hand before properly starting the game. And if at this point in the game a player wins from their initial hand, then it will add their index to the *winnerSignal* object, so that the main thread can interrupt all the other non-winning player threads before they have drawn or discarded any cards.

Lastly, a *StartGameSignal* object comes in use when there are no initial winners. In fact, when a player thread is ready to play, it will then wait on this object. When the main thread sees there are no initial winners, then it calls the *startGameSignal* object, and induces it to notify all waiting threads, i.e. all the player threads. At this point, each player thread starts their independent path of drawing and discarding cards until they have won, or until they are told another player has won.

**Other classes**

The rest of the code consists of four main classes: *Card*, *CardDeck*, *Player*, and *Game*; and of an executable class: *CardGame*.

The *Card* class contains a *cardValue* as an attribute, a constructor to assign the value of the attribute, and the methods needed to manipulate the object. An important method is the *isPreferredCardValue* which returns true if the card drawn by the player is their preferred denomination. This makes it possible for a player to sort their initial hand, and to decide whether to discard a card or not.

The *CardDeck* class contains a *cardDeck* and a *cardDeckIndex* as attributes, two constructors that assign the respective values, and various getter and setter methods that are used to add and remove cards from the deck, and return the size.

*Card* and *CardDeck* classes all need to have synchronized methods to enable them to be thread-safe. In fact the “synchronized” keyword makes sure that methods are accessed only by one thread at a time. This is crucial for preventing data conflicts in multithreaded environments. In our code, synchronized blocks players from accessing and modifying card decks and the signal object simultaneously.

*Player* is a class that extends Thread, and it is crucial since it contains all the necessary methods to implement the player logic.

The player logic is a method that is called by run(), and its main job is to check whether the player has won, and otherwise draw and discard cards. This method also deals with writing to the player output file. And finally, player also contains the end player logic method, which simply sets a flag attribute to true so that the thread can stop correctly. This way, when the threads are stopped and the game ends every player will have exactly 4 cards.

The *Game* class takes in the number of players and a list of cards and initializes all the necessary attributes to start the threads. It then has to deal with the different signals correctly, and finally terminate the threads.

The *CardGame* class just deals with the input data and reads and saves the content of the given file path. Then it starts a new game.

For a clearer view of the roles of these signaling objects, and the way they act in the game, below are attached two sequence diagrams that portray two main situations: when there is an initial winner, and when the winner comes in later on in the game.

A grid of graph paper with writing on it

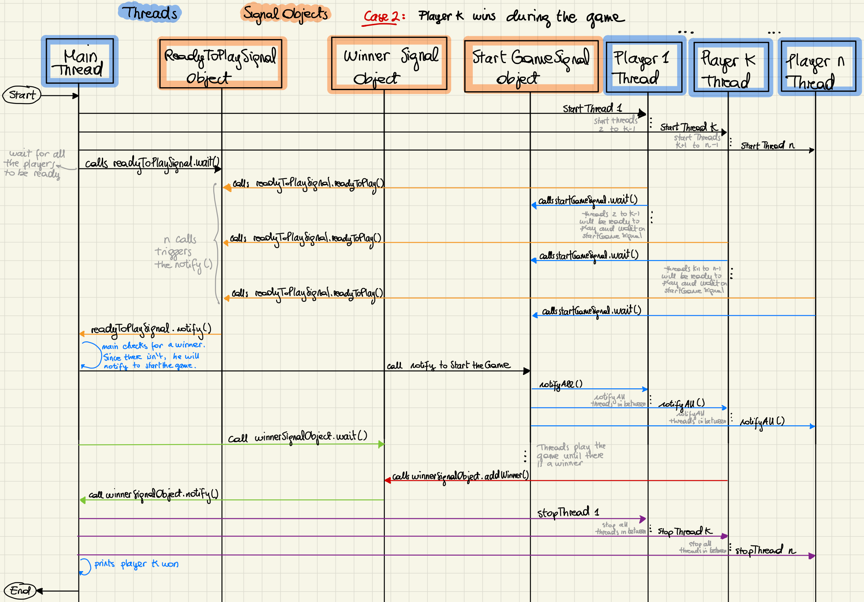
Description automatically generated

UML Diagram of the project

A black background with white dots

Description automatically generated





**Testing Code Design**

We have chosen to use the JUnit4.x framework for our tests. This has been done in-line for each method we have created, that we use for our implementation. To be able to test certain methods, we have had to create getter methods to be able to retrieve some outputs, therefore we have not tested these new methods that are not being used in the actual implementation.

**TestCard.java**

testGetCardValue() just checks that once a Card is created with a certain card value, that getCardValue() returns the same value.

testIsPreferredCardValue() is similar to above, that a new card object is created and when isPreferredCardValue is called and given a player index, then it checks if this value is the same or different to the preferred denomination.

**TestCardDeck.java**

testGetCardDeckIndex(), testSetCard(), testGetCard(), testGetCardDeck(), testSetFirstCard(), testSetLastCard(), testGetFirstCard(), testGetLastCard(), testRemoveFirstCard(), testRemoveLastCard(), are all testing the getter and setter methods, and therefore they are all similar in their structure. They all set up a card deck, and then check the relative results coming from each method. For example, testSetFirstCard() creates an empty card deck, sets 2 cards, and then sets a first card. By getting the first card, it checks that it gives the expected result.

testGetCardDeckSize() creates an empty card deck and checks it is empty, and then sets three cards. It asserts that the new size of this deck is 3.

testToString() creates a new CardDeck and checks that the toString for this deck is an empty string (“”), and then adds 3 cards to the deck, and checks that the new toString is correct for the indexes of the 3 cards inside the deck.

**TestWinnerSignal.java**

testGetWinnersListSize() creates a new empty WinnerSignal object and checks it is empty. It then adds 3 winners to the list and checks that the new size of the list is 3.

testGetWinnerList() creates a new WinnerSignal with three elements and checks that the first, second and third elements respectively are as expected. We do this separately, as we cannot just check that getWinnersList() returns the correct value due to it returning a location in memory.

testAddWinner() creates a new WinnerSignal with three elements and asserts the size of this list to be 3. It then adds another winner using the addWinner() method, and then checks that size of the winners list has increased to 4 and the toString returns the correct value.

testGetWinner() also creates a new WinnerSignal with three elements and then checks that the getWinner() returns only the first element in the list. We have to ensure it only returns one value, even if multiple values are added to ensure that only one player can win, even when more than one gets the correct cards at the same time.

testToString() creates a new empty WinnerSignal and checks that toString returns a string with no values, and then it adds 3 winners, and then checks that the new toString contains these new winners.

**TestPlayer.java**

This test file mostly does unit testing on the setter and getter methods which include testGetPlayerIndex(), testSetLeftCardDeck(), testSetRightCardDeck(), testAddToSignalObject(), testSetPreferredCardDeck(), testSetNotPreferredCardDeck(),

testSetInitialHandCardDeck(). These tests create a player, pass it a defined cardDeck and check that the respective action for each method returns the correct result.

testSortInitialHandCardDeck() creates a player and passes it a handCardDeck, sorts the hand, and then checks that the preferred and not preferred card decks created contain the correct values, and that the sum of the sizes of both decks results to 4.

testIsPreferredCardDeckFull() instead checks that if a player is given a winning hand with their preferred denomination, after sorting the deck, the preferredCardDeck should be full, and the notPreferredCardDeck empty.

testIsNotPreferredCardDeckAWin() is similar to the test above, but instead it checks that the winning hand is not of a preferred denomination.

testDrawCardFromLeftDeck(), testDiscardCardToRightDeck() test that a card is correctly drawn and discarded respectively. To do so, they are respectively given a leftCardDeck and a rightCardDeck, and then it checks that the correct card is drawn, and the correct card is discarded. And testDrawAndDiscardAction() tests that the two actions work together sequentially.

testCreatePlayerFile(), testDeleteFileContent(), testWriteToPlayerFile(), testWriteToPlayerFileWin(), testWriteToPlayerFileLose() all manipulate files. So they just check that the action of creating, writing and deleting is done as expected.

To be able to test playerLogic it is necessary to create a mock object that replaces the startGameSignal object. An interface that specifies the general behaviour of the StartGameSignal class is required to create such an object. Therefore, the interface IStartGameSignal was created such that it defines two methods to wait and notify. But the mock object, instead of waiting and notifying, it does nothing. Like this the methods result empty, and the player logic can be tested without having any waiting and signaling issues. Without this mock object, the test would not be able to end because it would be stuck waiting for a signal in the playerLogic.

In the case of testPlayerLogic1() this problem doesn’t occur, as we are testing that the player wins with their initial hand. This means that the player will never wait for a signal, so the mock object is not needed.

While in testPlayerLogic2(), as we are checking that the player doesn’t win with their initial hand but wins after one draw and discard action, the mock object is crucial. In fact, by using it, when the player calls to wait on the object, since the respective method in the mock object is empty, it doesn’t actually wait on it, but it just continues the game, allowing for the playerLogic to run and end.

Lastly, testRun() and testStopThread() simply assert that the threads created are alive or not.

**TestGame.java**

testCreateAllPlayersAndDecks() checks that the correct number of players and decks are creates and set into their lists.

testDistributeCards() creates a pack of cards and after distributing them it asserts that they were handed correctly. To test this method we decided to simplify the problem by having only two players and by reducing the number of cards each player should be handed and the number of cards each deck should contain to 1. This way we just created a pack of 4 cards.

testSetCardDecksIntoPlayers() simply checks that all the card decks are assigned correctly to each player.

testCreateAndWriteCardDeckFile() tests that for the created card deck, the output file contains the correct card values within the card deck.

testCreatePlayer(), testCreateCardDeck(), testCreateCardDeckNoIndex() all verify that either a player, a cardDeck with index, or a cardDeck without index are created correctly.

testGetNextCard() checks that given a pack of cards, each getNextCard action returns the expected card.

testPlayGame1() verifies that given two players and a valid pack of cards, when the game starts it works as predicted, i.e. player 1 wins.

testPlayGame2() tests that the game works for a single player, as long as the pack of cards contains a winning hand.

**TestCardGame.java**

In these tests, we would like to check all the ways a file could be invalid, and therefore testIsFileValid() uses external files that we have created. The first file testTrue.txt contains a completely valid file for 3 players, and so this asserts true. The second file testFalseEmpty.txt is a completely empty file and so is invalid, asserting false. The third file testFalseInvalid1.txt contains spaces and characters, and so will assert false, and the final file testFalseInvalid2.txt contains one less number than what is required for 3 players (needs 24 numbers but only has 23), and so will assert false also.

testReadFile() creates a linked list with a packOfCards containing 3 cards to match the file testCardDeck.txt which contains 3 numbers (1, 2, 3) where it checks that the first and last number in this file are 1 and 3 respectively. For testing purposes, we only use 3 cards which is an invalid file in the game, however, it is impractical to test a completely valid file, as this testing can be scaled up for any numbers.

testCreateCard() creates a new object of type Card with card index 1, and asserts that this card is correctly stored with the card index of 1.

We referenced: <https://www.w3schools.com/java/> mostly for file IO issues, and the functionalities of LinkedLists and ArrayLists.

<https://www.codejava.net/testing/how-to-compile-and-run-junit-tests-in-command-line>

to understand how to compile and run junit tests in the command line