# Software Development Coursework

## Production Code

The purpose of this project is to develop a card game in Java with the usage of threads to synchronously Our design choice for the project we decided to create simple classes for the cards, the player and the card decks. We decided to implement a few design patterns for example, Threading pattern, where each player is threaded so they all play their actions concurrently; this is used in the Player class.

### Card

The purpose of the Card class is to instantiate and initialise the cards and allow the program to view the value of the card, which is particularly important for checking if the players have a matching set of cards to win the game.

The value of the card is private final static variable as the value of the card is not changed once it has been instantiated and it is retrieved from the getValue() method.

### CardDeck

The purpose of the CardDeck class is to manage the functions of how the deck would operate in the game. The deck would add a card to the bottom of its pile and remove a card from the top of its pile when the player on its right draws a card and the player on its left would removes a card.

Each deck has a deck number that cannot be changed once the deck has been created and the deck of cards is represented as a queue, this allowed us to easily add cards and remove cards in the right order as a queue follows the First-In First-Out (FIFO) principle unlike a stack which follows the First-In Last-Out (FILO) principle. Making an array that would remove the first index and the last index would’ve involved more time.

The deck class also writes to the deck file all the cards currently in the deck when a player makes an action. This helps log the events of the CardDeck class to help with testing.

### CardGame

The CardGame class handles the main functionality of the game, where it handles the user input for the number of players in the game and the pack file they’re going to use, setting up the game readied to be played and then plays the game out until a player has won. The CardGame class utilises all the classes in this program except the Card class as the Card class is already used for the CardDeck class and Player class.

The design of the playGame() function is relatively simple and small as the main bulk of the actions done in the game are involving the player, which is handled by the Player class. The playGame class checks if each player has won the game, and if not, picks each player to start their turn.

### FileEditor

The purpose of the FileEditor class is to manage all the methods and process that deal with file management, whether that is creating a file, writing to the desired file and reading from the designated file.

The reason why we have created the fileEditor class instead of writing the players’ data in the Player class or writing the card deck data in the CardDeck class is because it removes the need of repetition and redundancies of re-writing the same function in different classes. This also allowed implementation to be much easier for example, creating new methods that involved writing or reading data was much easier as we only needed to call the functions from the FileEditor class.

### Pack

The purpose of the Pack class is to create the card packs from reading the pack file that is selected from the user input. It also checks if the created pack is a valid pack by checking if the pack for the game has 8n cards, where n is the number of players that are going to play in the game.

The design choice of this class being abstract is because the important methods for the pack are handled in CardGame for the packOfCards variable. This class was only used for file reading for pack and the validation of the pack.

### Player

The purpose of the Player class is to handle all the methods in a thread-safe manner for how the players would play the game. The player would add a card to their hand and remove a card of their choosing when its their turn in the game. The card they added would be taken from the top of the deck from the left of the player and the card they removed would be added to the bottom of the deck on their right. The game should continue until one of the players has a hand filled with cards of matching values.

The threading is handled in this class as this makes it much easier to understand and the players in the game were the only feature that was required to act synchronously. Each player has a respective identifier that cannot be changed once they’re constructed, and each is assigned a deck to their left and right. The player could choose which card they would want to retrieve as typically the player would discard a card that is not equivalent to their first card. Also, to keep the program simple, we made the value of the first card that the player receives in the game the value that the player would hunt down and collect to win the game.

The getAllCards() function was made so that methods that required all the cards in the player’s hands could be checked, which made writing the tests for the Player class much simpler. The Player class would also write to the respective player file all the actions and data for each turn of the player.

### Known Issues

From running the code, we have encountered no major issues in our program.

One issue that was found is that packs cannot be created, they only created manually and can only be read from for the game.

## Testing Development

For the tests on our project, we used JUnit Framework 5.0, where JUnit Jupiter was used for writing and executing test cases. We followed a Code-First development approach for our project where we

### testcard

The testcard class is used to validate whether that a card can be created and retain its value. Beforehand, the testcard class creates a test number for the card that is going to be tested, and a card is instantiated with the test number.

The test “testCardNumber” compares if the value of the card that has been instantiated matches with the original test number used for the card. If the numbers match, then the test passes, otherwise, the test fails and a failure message is displayed.

### testCardDeck

This class validates that when a CardDeck it correctly stores its corresponding identifier and that it stores and can return its list of 4 cards.

Testing Methods

getDeckNumber()

This testing method generates a random number and then creates a new CardDeck using it, then it asserts that the getDeckNumber() method returns the same value as it was created with.

setTestCard()

This testing method generates a random list of 4 integers and creates a CardDeck with 4 cards of different values in it. It then checks that the getAllCards() method returns all of the expected card values from the CardDeck.

### testCardGame

This is the test class for checking the methods in CardGame. It tests that a game can be setup correctly, checks what happens if a game is won instantly after distribution of the pack, tests the GameWon methods. It also tests the validation of read in pack files.

**Testing Methods**

resetGame()

This testing method is run before each test and sets the GameWon variable to false and clears out the list of players.

testIsGameWonInitial()

This checks that when the game is started the GameWon value is set to false.

testSetGameWon()

This test uses the SetGameWon() method to make the GameWon variable true then asserts that the variable is true to make sure that when a player wins the game they can set this variable to notify other players that the game is over.

testSetupGame()

This test generates a mock pack of 32 cards to be used with 4 players, it then sets this pack as the variable PackOfCards in CardGame allowing the setupGame() method to run. The setupGame() function runs trying to make a game with 4 players which involves distributing the cards out. Then this asserts that 4 players have been created and that they all have been given 4 cards.

testPlayGameWinningCondition()

This method checks that if a player starts with 4 cards of the same denomination, when the playGame() method is called it instantly realises that there is a winner and does not start the thread for each of the players. First it creates a player with a winning hand, then it sets the GameWon variable to true, before running the playGame() method. Then it asserts that the game is won after this.

testInvalidPackFile()

This test attempts to read in a non-existent pack file and the asserts that an exception is thrown to do with the file not existing.

### testCardGameSuite

This class uses the Junit Suite annotation to give an easy location to run all of the tests from in one go. It uses @SelectClasses to choose which test classes should be run so that you can change or run specific tests easily.

### testFileEditor

The testFileEditor class ensures that the functions and methods in FileEditor work as intended.

The test “testFileCreation” creates a file using a string name for the test file and checks the file to confirm that the file does exist. If the file exists, then the test passes, otherwise the test fails if the file does not exist in the project folder and displays a failure message.

The test “testFileWriting” checks both the writeFile method and the readFile method in the FileEditor class by creating the test file, then writes to the file with the test contents and validates if the test contents exist by reading the file and compares if the contents retrieved from the file match the test contents. If the contents don’t match however, the test fails, and a failure message is displayed.

### testPack

The testPack checks to see if generating the card pack and validating the card pack in the Pack class works as intended. Before all the tests, a pack file is generated using the FileEditor class, and then a selection of 24 cards with random values is generated (supposing the test is going to generate a pack for a game of 3 players).

The test “readTestPack” reads the pack file using the name of the file and stores the cards in a queue called fileCards. Another queue called testPack takes the contents of the original cardsPack queue and then both fileCards and testPack are converted into arrays. The test finally compares if both contents of the arrays match. If the contents match, the test passes, otherwise, the test fails, and a failure message is displayed.

The test “validateTestPack” creates two packs (as type Queue<Card>) to be tested. The first pack receives 8 cards, and the second pack receives 4 cards. The test then confirms if the first pack has a multiple of 8 cards, whilst the second pack does not have multiple of 8 cards in the pack. If one of the tests fail, then the entire test method fails, and the corresponding failure message is displayed.

### testPlayer

The testPlayer class ensures that all the methods for the Player class works as intended. Beforehand, a test player number is randomly generated as well as a placebo left deck and right deck are instantiated with the deck numbers 1 and 2 respectively. This is to make creating the player simple.

The test “testPlayerNumber” tests if the player number is recorded properly. It checks this by creating another test player number, then creates a new player using the test number. The test then compares if the player number matches with the test number. If so, the test passes, otherwise, the test fails, and a failure message is displayed.

The test “testPlayerCard” creates a new card with a random value and creates a new player using the already made player number. Then it adds the new card to the new player’s hand and retrieves the card to check if the card that the player has received matches with the original card. If the card matches, then the test passes, otherwise, the test fails, and a failure message is displayed.

The test “multiplePlayerCards” is like the testPlayerCard test but instead it adds multiple cards with random values to the player and all the cards added are also added to an array. The test retrieves all the cards and converts it to an array to see if it equates to the original array. If not, the test fails, and a failure message is displayed.

The test “playerWinsTest” creates 3 different players with different player numbers, and they’re given different sets of cards. Player 1 receives 4 cards of value 1, Player 2 receives 4 cards of value 5 and Player 3 receives 4 random cards of different values. The test then confirms if that Player 1 and Player 2 should win, and Player 3 shouldn’t. If a single test fails, the entire test method fails, and the corresponding message is displayed.

The test “playerTakesTurn” creates a temporary left deck and right deck and inserts random cards into the decks. A new player is created with these temporary decks and begins their turn, drawing from the left deck and dropping one of their cards into the right deck. The test then checks if the player’s hand has 4 cards remaining. If not, then the test fails and a failure message is displayed.

### Conclusion

The JUnit tests had been done to ensure the reliability of the project. The unit tests that were written had covered 100% of the main classes for the card game, 82% of the methods used in the card game and 56% of the lines that were written in the code. According to all the tests in Card Game, 17/17 of the tests passed.

## Development Log

A screenshot of a document

Description automatically generated