Design Choices and Reasoning

Class Diagrams:

As seen in figure 1 below, we decided to have four different classes (Player, Card, CardDeck and CardGame). With the 3 classes, Player Card and CardDeck, having a composition relationship to the CardGame class. This is because without an instance of CardGame there wouldn't be any players, cards, or decks.

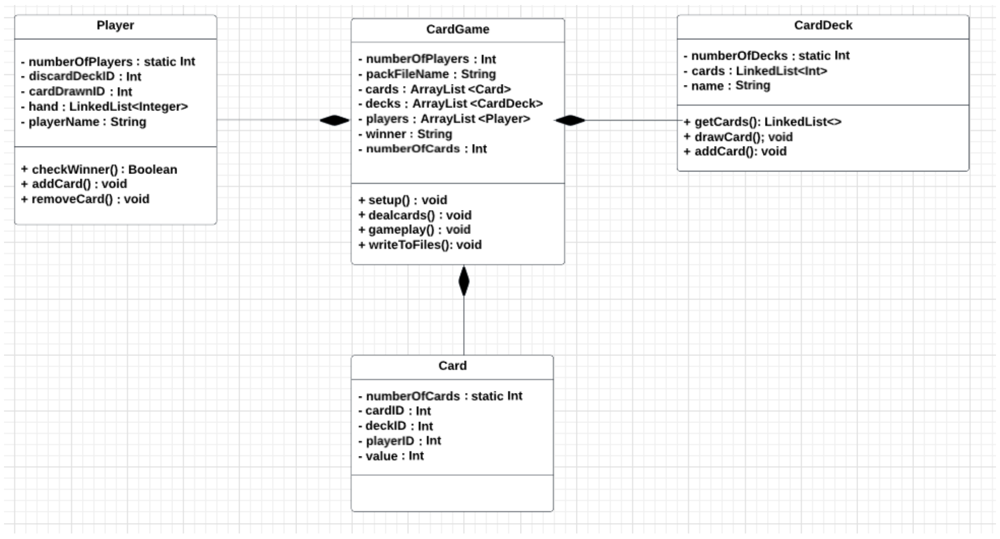

Figure 1 UML Class Diagram (Not including simple methods, such as getters, setters, constructors…)

CardGame

Attributes

We decided to store all the objects, of players, classes and decks, in an ArrayList<Object> inside of CardGame, as it is easy to iterate through, as well as to access objects via their index within the ArrayList. Moreover, we decided to use this, instead of a list, as we can add players, cards, or decks to the ArrayList using for loops.

Methods

The setup() method will take in user input of, the number of players, and the location of the pack file, and check whether both are valid, and matching (e.g., a 4-player game, should have a pack containing 32 cards), this method should not return until this has been done. After this, we can call dealCards() to deal out the cards from the pack to the players in a round-robin fashion, then deal to the deck in the same fashion, as per the rules.

Methods such as gameplay(), and writeToFile() will be used to play out the game of cards, writing to the appropriate output files within the output directory, and once the game is finished, and a winner has been established, the game ends, and appropriate messages are written to all files.

CardDeck (Thread-safe)

Attributes

Inside of the CardDeck class, used to represent a deck of cards, we used a LinkedList to store the **values** of the cards within a certain deck, this was because we can utilise the pop() method when taking a card from the top of a deck, and simply add cards to the end of the LinkedList when putting a card on top of a deck.

Methods

The drawCard() method will use pop() to take what is at the front of the LinkedList, simulating a player drawing a card from the top of the deck, as specified in the specification.

Card (Thread-safe)

Attributes

Card will only hold its value, and where it is currently, (in a player's hand, if so which player, or in a deck, if so which deck). This is to avoid having repeated data stored in multiple places.

Methods

Only simple methods will be stored inside of the card class, such as getters and setters.

Player(Thread-safe)

Attributes

We decided to store the cards in a player's hand as a LinkedList<Integer>, where the integers hold the **values** of the cards in their hand. This is because it will make it much easier to go through the cards to check if a player has a winning hand or not. It will also hold the discardDeckID, as each player will be discarding to a different deck, and it depends on the number of players in the game, for example in a 4-player game player 4 discards to deck 1, whereas in a 5-player game, player 4 would discard to deck 5.

Methods

As we decided to store the cards in a player's hand this way, checking whether they currently have a winning hand shall be easy, using checkwin() we iterate through checking if the cards are all the same value, and they must have only 4 cards in their hand. The removeCard() method will have to implement the game-playing strategy as described in the specification, then randomly choose a card to discard.

Performance Issues

In order to try and reduce overhead and remove any performance issues, we decided to centrally store our objects inside of our CardGame executable class and pass by references using their index inside of the ArrayList<>. This is also to try and reduce repeating data.

An example of a performance issues we had, was that when multithreading, we noticed player 1 always had the first action of the game, so we decided to shuffle the players using collections.shuffle to assure that the order of thread execution was random, and that there was still one thread for each player, created using a for loop.

Furthermore, when multithreading, we needed to assure that a player could not draw from a deck which was empty, in order to make sure it was all thread-safe, we called for threads to wait when they try and draw from an empty deck and notified them when a card was added. Also, once the game was over, we called for all threads to be interrupted, to stop them immediately once a player has a winning hand, without continuing to draw and discard cards.