ECM1410 Cover Page

|  |  |  |
| --- | --- | --- |
| Student ID | 710011124 | 700032368 |
| Weight | 50% | 50% |

# Design Choices

Diagram

Description automatically generatedThe overall design for the program is shown in the UML diagram to the left. All classes are used in the main function of the CardGame class.

The CardGame class deals with creating the players and decks, dealing out the cards (validation the inputs given) and starting the players threads.

The CardGame() method takes inputs from the user specifying the pack file and also the number of players. This input is read using a BufferedReader (bReader), validating the number of player given by the user is done within this method, however by passing the bReader object into the Pack() method of the Pack class, we were able create a new Pack instance, while keeping the implementation of checking for the validity of the pack file within the Pack Class.

Once the CardGame() method has a valid number of players and an instance of a Pack, it them runs the dealCards() method – this method creates the players (when a player is created their left and right decks are passed in as variables to the creator method), decks, and deals the cards to specification (storing the players in the players : ArrayList<Player> attribute). Finally the CardGame methods checks whether any of the players already have a winning hand and the runs threads for each player.

When designing this program, we decided that a passive way to avoid stagnation (i.e. cards remaining in the same place indefinitely, and the game not ending) was to store the cards in queues. Expanding on this we decided that the Player and Pack classes should inherit from the CardDeck class, because all three entities have the same basic storing/drawing and discarding of cards.

In order to make a player’s go atomic, we implemented a static AtomicInteger `winner` that stored the winning player’s number and is checked before every player starts their go – if `winner` has been set then a player has won, and all player will stop their next go, if they are in the middle of a go then this finishes, and they are stopped when they next try to play. This also provides a simple way for dealing with two players winning at the same time – because setting the value of `winner` is an atomic action, when a player “wins” they try to set `winner` to their number, if this has already been set then they still lose (i.e. first to change `winner` wins the game).

When designing the program to be thread safe we initially make the Player.playGo() method static, this was to insure the atomic requirement of a player’s go. This however led to only one player’s thread playing at any one time, which we felt defeated the point of multithreading. So insure the thread safety of taking and discarding cards to the deck, we synchronized the takeCardFromTop() and placeCardOnBottom() methods – thus allowing only one player to access a deck at the same time, but multiple player to player at the same time.

We decided to implement Runnable in the Player class. Originally the Player.playGo() method was void, but we changed it to return a Boolean – `true` if the player should play another go, `false` if one of the players has won. This allowed a very simple while loop to implement the running of a player go in the player’s .run() method.

In the validation of the pack file, we initially had a counter that incremented when a new line was reached and invalidated the pack when to many lines where being read (according to the number of player). This led to clunk looking code and would only provide noticeable performance improvements in the case that a pack file is exceptionally larger. Thus, assuming the pack file only contain number (invalidation occurs while reading the file if a line doesn’t have a number in it) the whole pack file is parsed into the pack’s cards queue, and then the length of this queue is checked against the expected number of cards.