Our solution was initially developed based on a UML diagram designed in the first meeting. Our code, however, gradually diverged from this first design as we continued development. Our final solution consists of 4 classes: Player, Deck, Card and CardGame.

## Card

A *Card* is a mostly-static object that holds an integer face value retrievable by other classes. The class *Player* has an attribute, *cards*, of type *ArrayList<Card>* to hold its cards. Each *Deck* contains a very similar object to hold its *Cards*, called *cardsInDeck*.

## Deck

The Deck class contains a lot of the core thread safety code, as in order to avoid concurrent modification, we have to ensure that each deck can only be accessed by one *Player* at a time. *Deck*’s two main methods are *addCard* and *removeCard*, used by the *Player* class to discard cards and pick up new cards. Another key part of this class is its *ReentrantLock* attribute, *deckLock*. Before a Player draws or discards a card, we obtain a lock on the deck object using *deckLock.tryLock()*. We decided to use this lock instead of synchronised methods because the implementation is more flexible and it automatically supports a fairness policy.

## Player

This class defines the strategy adopted by the players in playing the game. *Player* implements *Runnable*, and each Player runs as a thread, independent of the other *Players*. The *run* method of this class defines how each player acts and interacts – an abstract description of how it works is given below.

1. Check if the player has started with a winning hand, and run the game-finishing code if it has.
2. While the game is still running, have turns.
3. When the above while loop exits, increment the number of finished players held in the CardGame object.
4. Wait until notified by CardGame – CardGame is busy counting the maximum number of turns had by any player.
5. Continue playing until the number of turns had is equal to the number described in step 4.
6. Increment the count of finished players held in the CardGame object.

The concept for the design of the *run* method is that there are two distinct stages in its running, each enclosed within a while loop. The first while loop loops while the Boolean *running* in the CardGame object is true. This loop runs while the first stage of the game is still running, when no players have won yet and they are trying to win. Then, the players *wait()*, and then enter into their next stage – a while loop with the condition that turnsHad < turnsAllowed. This is where the players are catching up on the number of turns they are allowed.

We use *wait()* on the Player object to wait for the CardGame object to find the largest number of turns had by any player. We made this design choice because the CardGame object should only count the number of turns had by a player, when the player is not having turns (and therefore not changing its number of turns had).

A player has a turn by calling *haveTurn()*, which works as follows:

1. If the player’s left deck does not have any cards, then exit.
2. If we have an unsuccessful attempt at accessing the lock, then another player is currently modifying the deck, so exit.
3. Draw a card from the left deck
4. Discard a card to the right deck, holding onto cards of the player’s preferred denomination.
5. Check whether we have a winning deck. If we do, inform the CardGame object.

## CardGame

This class is used to set up the game and keep track of its players and decks. This involves asking the user for input on the number of players and location of the pack file, and then creating objects used in the running of the game, such as an ArrayList of players, *players*. We do some exception handling here, and the reason for using the general Exception class rather than subtypes is that there are so many possible invalid packs that can be passed in as pack files to the program, that it is simpler just to output a general “try again” method with a description of a valid pack.

The CardGame is designed to not be involved in the interaction between players – so that the players can independently play the game, until they have all finished. We achieve this with the following pattern of communication between CardGame and Player.

1. CardGame creates the Player threads, and starts them.
2. CardGame does not get involved with the playing of the game, and waits for the Players to finish, with an empty-body while loop.
3. When all the players have finished, CardGame tells the players how many turns they are allowed before they are finished, using the variable *turnsAllowed*.
4. CardGame starts the players again by calling *notify()* on them.

## Ensuring thread safety

We used three different methods for keeping the card game thread safe, these were;

Atomic variables, these were used within the CardGame class for the finishedPlayers (AtomicInteger) and the gameRunning (AtomicBoolean) variables. An atomic vaiable can only be updated by compare-and-swap operations. This makes so when multiple threads attempt to update the same value one of them wins and updates it. This functionality allows us to make sure that the finishedPlayers and gameRunning variables are never updated at the same time by different threads.

ReentrantLocks, in our program we used a Reentrantlock on each deck to make sure that only one player can hold the lock and so access it at a time. The ReentrantLock was used for this instead of a synchronised method because it is unstructured; therefore a block structure is not required. This allows other threads to access parts of the deck object to start their computation while still blocking them from parts where changes are made to avoid concurrent changes occurring.

Synchronised methods/blocks, our program made use of the synchronised keyword for the ‘incrementFinishedPlayers’ method. This is because we only want one thread to be able to run this method at any one time, therefore making the method synchronised means that any other thread trying to access it has to wait till the lock has been released. We also used the synchronised keyword for small blocks of code including within the ‘playGame’ method where it is used to notify each player individually that they can stop waiting and continue with their computation.