Test Design Choice and Reasoning

We decided to use JUnit 5.9.1, in order to test our project.

Design

For our test we have had to create multiple examples of a game of cards, with varying numbers of players (from a 4-players game, all the way up to a 10-player game). We tested these games with several packs, some of which are invalid, and the rest with valid number of cards holding non-zero positive digits. We also created games where we know the outcome, for example a game where a certain player will win with their initial hand, but also games where a certain player will always win, due to the gameplaying strategy, throughout the production of our tests, we also creating games which never end, due to it being impossible for any player to win, to assure that in those cases our game does not end.

CardGame

Firstly, our game should be setting up the game, using the user's inputs, for the number of players, and a file containing the pack of cards, to be used in a game. Hence, it is necessary that we test whether this works as expected, inside of setupTest(), that our programme now knows, the number of players, and the corresponding pack, and that they are both valid.

After this, our game shall need to deal the cards out to all the players, then all of the decks, this shall be tested inside of 2 tests called dealCardsToPlayersTest() and dealCardsToDeckTest(). Via one of our test games.

Next, our game should carry out all the necessary processes for the game to take place, with players drawing and discarding cards simultaneously, from the correct decks, this can be done via our gameplayTest(). Moreover, we shall need to test the thread safety of this, assuring that while the game has been written in a multithreaded fashion, it must be thread safe, meaning if a player draws from a deck until it has no more cards left, the game must no longer allow that player to draw more cards, until another player has added cards to that deck. This can be done in several ways, such as we create a player and a deck, both with n cards, and ask the player to draw from that deck n + 1 times, thus a player should only have 2n cards, not 2n+1.

Once the game is over, our checkWinnerTest() method must stop all of the threads and output the correct messages to all files. This can happen at 2 different points throughout a game, for example when a player has been dealt a winning hand at the start or after having drawn and discarded, one or many more cards throughout the gameplay process. Both of these examples will be tested. We can do this with our example games, where a player wins immediately, or when a known player will always win, and asserting that what was written in the output files, is what was expected. Note that this will also test that our methods to output messages to files also works how we want it to.

Due to the fact, that this class will be working with treads, we must assure that these threads are working as expected. For example, we must assure that in games with different numbers of players, there are different numbers of threads being created, one for each player. Moreover, we must also assure that our methods are thread safe, and that these threads are running lightweight processes as expected.

Player

At the start of the game, cards must be dealt, from the pack to all players in a round-robin fashion, we shall need to create a test which makes sure that this is done accordingly. This can be done within a test called addCardTest(). We can use one of our pack that we created, and check the contents of a player hand, using Player.getHand() and make sure that it is what we would expect from our pack of cards.

Our player class, we will need to test our checkWin() method, to make sure that we can calculate whether a player has a winning hand, and if so, the correct procedure occurs, this shall be done in our checkWinTest(). We can do this via simulating a game where we know the outcome, such as a player having a winning hand as their initial hand, assuring that this is the result our programme reaches, this test also tests that we can obtain a player's hand.

We will also need to test, that our gameplay functions from our CardGame class, are interacting with our player classes as expected (when a thread from CardGame, asks a player to draw a card, it draws from the right deck, as well as when it asks a player to discard a card, a card is discarded, following the gameplaying strategy, e.g., player 1 cannot discard a card with the value 1). This shall be done in drawCardTest(), addCardTest().

This class, as per the specification, must be thread safe, therefore while a certain player is drawing and discarding cards, other players can be doing the same simultaneously, this must be tested. Moreover, in a fair game, it must be tested that not always the same player wins, this can be tested by running one of our example games, where multiple players have a chance of winning, and checking that, through the operating system being in control of when threads are being scheduled that not always the same player wins.

Card

Our card class shall be a very simple class to test, as it is small class, with very simple methods, we shall need to test, that at all times a card knows where it is in the game, such as in whose hand, or in which deck, this shall be tested in our dealCardsForDeckTest(), and our dealCardsForPlayersTest() tests, both of which are methods found in CardGameTest. These tests will take in a given pack, where we can work out what the starting hands for all players are, and make sure that in the output files, our project dealt the players the correct hands. As well as all of the getters, setters, and that we can correctly instantiate an instance of the Card class, with the correct values loaded from the pack, this shall be done inside of our loadPackFromFileTest() found in CardGameTest. We can do this by, loading a valid pack, and making sure that the correct number of cards have been instantiated, and with the corresponding attributes.

Card class, as per the specification must be thread-safe, therefore even though actions can happen simultaneously, a card can only be in a certain place at one time, for example a card should always be inside either a player's hand or inside of a deck, never both.

CardDeck

At the start of the game, when we have dealt the cards to the players, we must now deal the cards to the deck, in the same round-robin fashion that we did with the players. Therefore, we must test that this round-robin fashion has been administered correctly, this can be done in a dealCardsForDeckTest() method. Given that we can create our own pack of cards, we can find out where all those cards will end up, and using our outputDeckToFile() method we can see the contents of all of our decks, this assuring all the cards are in the right deck/player's hand

Moreover, this class must adhere to several of the rules, given in the specification, such as when a player tries to draw a card from the deck, that they draw cards from the top. Similarly, this class must assure that when a card is discarded from a player's hand to a given deck, the card goes to the bottom of the deck. These rules shall need to be tested inside of drawCardTest() and discardCardTest(). This can be done, by using a simulated game where we know what cards are in the decks and player's hands at the start of the game, and reading through the output files, to make sure when that a deck is putting the cards it's been given in the correct places, and player's are taking cards from the correct places, inside of our LinkedList<Integer>, taking from the start of the list, and adding to the end.

Exception handling

The specification demands for our code to be robust, therefore throughout our code we will need to deal with several kinds of exceptions, excluding when there are multiple players who have won the game. When setting up the game, we are expected to take inputs from the user. Therefore, we must test, several different exceptions the user could throw. For giving the number of players, for example not giving us a positive integer value (such as a string, float, negative number, numbers which exceed the limit for an int or 0). Or when being asked for the location of the pack to load, giving us a non-existing path, or an invalid pack (such as a pack with too many numbers, a pack with negative numbers for the values of cards, or a pack containing non-integer values).

Moreover, throughout our code, we are expected to be writing to files, therefore we must always be handling exceptions where the path of the file we are trying to write to does not exist or is not correct. This would occur when the user has not changed the GLOBALPATH inside of the executable CardGame, or inside of our test suite.

As we are working with threads and will be using methods such as wait(), and interrupt(). Therefore, we will need to be able to handle exceptions such as when we try to ask a thread to wait, when it has been interrupted.