ECM2414 COVER PAGE

Lucia Adams and Jessie McColm

November 2022

Mark allocation

Lucia Adams	[redacted]	50%
Jessie McColm	[redacted]	50%

Design Document

We started our design process by considering the various classes we would need and their respective methods and attributes. We then considered the inputs and output types of these methods, considering how we could test these as part of unit testing. This included considering which methods should be private and public and therefore could be accessed from different objects.

Card

We chose to store the cards as objects opposed to purely integer values, as this makes it easier to distinguish between two cards that have the same value but are different items within the game. We compare whether objects are the same as opposed to integer values.

Pack

We decided to implement a class Pack that would deal with: reading in the pack file; checking its validity; and making a list of card objects (the pack) for the game. We wanted to encapsulate these similar functions into one separate space. We have a private checkFileValidity method only ever called within pack methods. Having checkFileValidity as a separate method opposed to within the readPackFile decomposes the code into logical sections and also results in the ability to test whether a pack is valid separately from creating the card list. When a pack is created we set it to a default value of being invalid. We then pass in a file to read and set its validity based on whether the file is valid or not. If valid, it stores a list of card objects, otherwise it prompts the user to resubmit a file. We have used BufferedReader to read the files as it is synchronised and can be read from multiple threads. Despite not using this feature, we thought it was good to use due to the emphasis on this application being thread safe.

Deck

We decided on implementing a Deck class as it had several distinct functions from Player, and we believed that it shouldn't be a thread/ implement runnable. A deck is constructed with a given deck ID (these decks are created sequentially in cardGame via a loop that runs up to the number of players inputted). In the constructor we have chosen to create the filename of the file that deck will write to at the end of the game. We created this here as it only needs the deck ID and is unique for each deck object. The deck is structured in such a way that removing a card pulls it from the start of the array list containing all of the deck's cards, and that adding a card adds it to the end of the list.
We have a method endGame that writes the final contents of the Deck to an output file. It made sense to contain this within Deck as it only needs information pertaining to a deck object. We have implemented several methods to report on the size of the deck to maintain clear control on the size of the deck throughout, which is used heavily in Player's pickAndDrop method. We store the list of cards in deck as an arrayList as it fluctuates in size throughout the game.

Player

The add method is used to initially deal out the cards to the players opposed to picking up a card. We implement picking up and dropping a card in one go via the synchronised pickUpAndDrop method, in order to maintain the atomic nature of the function which was detailed in the specification. This function is synchronised as we need to ensure thread safety by preventing two players attempting to work with the same deck simultaneously, which could lead to unwanted behaviour. In pickUpAndDrop we check if the deck we pick up from is not empty before we attempt to pick up from it, in order to prevent errors where we try to pick up a card but no cards exist. We also check that the deck that we drop to doesn't have too many cards (more than 4) so the decks don't get too unbalanced (i.e. one deck has 6 cards). We cannot use the idea of only dropping if the deck isn't full (i.e. has less than 4 cards), as when the game starts we wouldn't be able to drop to any decks.
We have an attribute hasWon to track whether the player is the winner of the game, which is set to true after running the checkVictory method. The win and loss methods handle writing the corresponding messages depending on the game result to the respective player files. EndGame is similar to the corresponding Deck method, but for the Player's file, and is called in both win and loss to write the full message to the output file. This helps reduce repeated code.
We also have a volatile attribute called alive which is used to determine when to end the thread. It is set to true at the start, and false by either winning the game in checkVictory, or by terminating the process with the method kill. It is set to volatile to ensure this is permanently checked. The run method makes the player pick up and drop cards while the player attribute alive is true. It runs checkVictory throughout before pickAndDrop, so that a player can't attempt to drop a card once they have already won (it also ensures that if a player wins at the start of the game, the game ends immediately).

CardGame

We use a scanner to take in user input for the number of players and pack file, which is then validated. We have loops to allow the user to re-enter inputs if they were deemed invalid.
We also have a method to deal cards to the players and decks in the specified order. We have a runGame method that starts the threads on all player objects and is polling the player objects to check for a win. Once a player has won, determined by calling getHasWon, the polling loop is exited and we end the other players threads with our kill method. We then call win on the winner and loss on the other players, thus outputting their results to the file. We decided to use a polling method as we wanted to end the game as soon as a player had won and cardGame isn't a thread object.

Development

We followed a pair programming methodology, switching the observer and driver after every break. Our version control was done using git and github to push and pull versions of the code as we switched between these roles on our local devices. We used a github branch to start implementing the polling initially until we were happy with its functionality, at which point we merged it into main.

Testing Document

Framework

We wrote our tests using the BlueJ IDE, so they use the default framework for that platform, which is JUnit 5.

Developing tests

We wrote tests throughout our development, starting with testing the basic functions of our classes (especially those functions that did not call any other functions).
Then we focussed on writing tests for all of the functions present in our classes, ending with testing the functions in the CardGame class. Due to some methods in that class requiring user input from the command line, we decided, instead of directly testing them, we would thoroughly test the methods that they use in tests for our supporting classes.

TestPack:

In order to test our pack file validification function, we created many different test functions operating on different pack files we had created.
We tested valid packs for:

- 2, 3 and 100 players
- A valid pack of all 0s (as an edge case)

Alongside multiple packs that were invalid for different reasons such as:

- File being too long (for specified amount of players)
- File being too short (for specified amount of players)
- File including invalid characters
- File including negative integers
- Empty file
- File not existing

We checked that all the invalid files resulted in the validity function returning false.
Since this method was private in the Pack class, we had to use Java Reflection to invoke the method and to change instance variables to allow a given pack file to be tested.
Each method using java reflections is marked as throwing an exception, however we know these private methods and variables we are getting exist within the class so an exception will not occur.

To test that reading a pack file would result in the correct cards being created, we created a test where we read a given pack file and checked that the generated card list had cards of the expected values. We did this by looping through the list, getting the value of each card and comparing it to a list we created that stored the values we expected to see.

We also wrote a script (in java) to generate pack files for us to use in testing. It took in the number of players we wanted to test for, as well as whether we wanted it to have AI bias (including numbers of the players) or not. They were all winnable packs in order to terminate the program, however we did test non winnable packs separately. This was used to generate some pack files used within our junit tests, furthermore to our physical testing that the game was operational. This file has not been supplied in the submission.

TestDeck:

Several tests in Deck require us to set up card objects to provide to the deck, and we often are checking that the card objects returned from various methods are equal to the card objects we expect. In testRemoveCardOrdering we check that the first card added to a deck is also the first card removed from a deck, to ensure that the ordering of adding and removing cards is as we expected. We also check that the ordering of cardList is as we expect in testGetCardListOrdering().
We checked that the output Deck writes to the output file is as expected in testEndGame, by reading the output file after calling endGame and checking that it matches the String we know it should write (as we have set up the deck to be storing cards of known values).

We delete this output file after the tests as it is not needed. This is done within the afterEach teardown method.

There are multiple tests that check methods that deal with a Deck's size such as:

- Deck size too big
- Deck full
- Deck empty

For each of these we call them on a deck that matches the description (has 4 cards so is full, has 0 cards so is empty, or has 5 cards and is too big) and check that all these methods return true. We also call each of them on a deck that doesn't match the description (has 1 card so isn't full, has 1 card so isn't empty, or has 4 cards and isn't too big) and check that the methods return false.

TestCard:

In testCard we only need to test Card's two small functions, so we set up a Card with a given value. We then check that its toString method returns a string that matches that value and the getCardValue method returns the integer value of the card. Storing cards as objects as opposed to just as integer values is overall useful in testing, as we can ensure a card is the right object, as well as just having the right value.

TestPlayer:

In testPlayer, we again have methods to check that getPlayerID of a player returns the ID of a player as expected; that cards are added by addCard in the expected order; and that the expected cards in the expected order are returned by getHand. We also have tests that check that the output to a players file when a player wins or loses is as expected. For example, the string messages in the player file match whether win or loss was called and the state of the player object when that method is called. We check this part by checking that the card values outputted match what we have set. In loss, we also have to test that the playerID who has been declared as the winner (by being passed to the method) is outputted as having won. Ie "player x has informed player y that player x has won".

We also test the pickAndDrop method by checking that the output files match the options we expect to be possible (due to the randomness inherent in the method). We have two tests that test this method, one of which where the player can only drop one known card and the other when the player has the option to drop two different cards. We check that the output line detailing the card picked up and the deck picked up from is as we expect, likewise for the discarded card and deck. Finally we check the line detailing the player's current hand is one of the options we expect. We have a test to check the player doesn't drop a card whose value matches the player's ID, as this shouldn't be possible due to the AI in the game.

We test the kill method by calling the run method after calling the kill method and ensuring that the run method terminates. We test if run terminates by asserting true is true after the run method is called.

Finally we have methods that check the test victory method. We ensure the check victory method is true when a player object is set up to have 4 cards of matching values and that the method is false when the player has :

- Less than 4 cards (ie 3 cards) regardless of whether they are matching or not
- Less than 4 cards of matching values (ie 3 matching cards and 1 non matching card)
- More than 4 cards of matching value (ie 5 cards), as this shouldn't be possible

We have a tearDown method that deletes the player1_output.txt file after each test, so that there aren't unexpected values from previous tests within the file for a new successive test.

TestCardGame

In TestCardGame, we have a test that checks that dealing cards to 2 players and 2 decks works in the intended order, by checking that the cardLists of player1 and deck1 have cards that match the card objects we expect in the correct order. We also have a method to check that these cards, when dealt, aren't equal to something we know they shouldn't be equal to.
Furthermore, we test the runGame method with two short games. One of which has only 1 player and 1 deck, where we check that the last lines in the player output file are as we expect as there is only one player who can win and one winnable hand.
We also test a game with two players, where player1 should win immediately as player1 is initially set up with a winning hand. We check that the only lines outputted to player1's output file, are lines outputted by the win method that detail how player 1 won.We also check the last lines in player2's output, are lines from the loss method, detailing how player 2 has lost. We don't test any longer games due to the randomness inherent in the game, which results in an expected indeterminable length of games.

We have a tearDown method that deletes the player1, player2, deck1 and deck2 files after each test.