ECM2414 Software Development Continuous Assessment – Card Game

Joey Griffiths and Alexander Cairns

Final mark allocation

Student	Joey	Alexander
Weight	50%	50%

Development log

Date	Time	Duration	J Role	A Role	J Sig.	A Sig.
31/10/2022	14:00	02:15	Observer	Driver	710022765	710033804
03/11/2022	15:15	01:45	Driver	Observer	710022765	710033804
09/11/2022	14:30	02:30	Observer	Driver	710022765	710033804
10/11/2022	14:30	01:00	Driver	Observer	710022765	710033804
11/11/2022	15:00	03:00	Observer	Driver	710022765	710033804
16/11/2022	15:30	02:30	Driver	Observer	710022765	710033804
19/11/2022	15:00	02:00	Observer	Driver	710022765	710033804
22/11/2022	14:00	02:15	Driver	Observer	710022765	710033804
23/11/2022	11:30	02:30	Observer	Driver	710022765	710033804

Design Choices

- PlayerThread class that implements Runnable, with each player running their own instructions for playing the game in parallel.
  This was done so that the program runs more efficiently and to simulate how the game would be played in real life, with each player thinking and acting independently.
  - Each player stores a list of Card objects to represent the player's hand; it has methods to draw cards from the CardDeck object (which represents the deck to the players left) and similarly to discard cards to the deck on the right of the player. The references to these decks are given to the player when it is instantiated. The hand is stored as an ArrayList<Card> as opposed to an array as this allows elements to be dynamically added and removed from the hand as the program runs. The PlayerThread also stores an ArrayList<PlayerThread> of the other players in the game, so that they can alert each other if they win the game using PlayerThread.endgame(winnerId).
  - The PlayerThread run() method runs when the Thread containing it has its start() method called in CardGame. When run() is called, the player first records its initial state to its unique output file player[id]_output.txt. It then checks whether it has won the game initially using checkWin(), and if it has, it iterates through each player in the game, including itself, and tells them to execute their endgame() method, which records the players final hand to their output file, and has a different message

depending on whether that player is the winner. If not, it records that it has been notified by the winner, that the winner has won. The winning player also outputs to the console "player [myId] wins". If the game does not end initially, each player will run a loop every 50ms (using Thread.sleep(50) at the end of the loop), executing the players move and recording it to the output file. The move consists of a draw and a discard, using the drawCard() and discard() methods respectively, which interact with the left and right decks. The player must choose a random card to discard, so calls the chooseDiscard() method, which creates a copy of the hand list, filtering out cards that are of the player's preferred value, and then randomly selects a card from this list to return. After drawing and discarding, the player checks if it has won using checkWin() and if it has, tells every player in the game, including itself, to run their endGame() method.
The Thread.sleep(50) is added at the end of each loop to insure that every thread has time to complete its loop before they all move on to the next iteration, because otherwise this would cause some players to draw or discard more than others, leading to errors, as decks can run out of cards when they should always have 4 after every move.
    - The checkWin() method checks whether every card in the player's hand has the same value by finding the first Card's value (using getValue()) and iterating over the rest of the Cards in the hand, counting how many have the same value. If 3 more have this value, the method returns true, otherwise, returns false.
    - The recordInit(), recordMove(), and recordEnding() methods use the FileWriter outputWriter to write strings containing information about the player to its respective output file. These strings are assembled using the methods getInitString(), getMoveString() and getEndingString(). recordEnding() also tells the player's left deck to record its final state.
    - endgame() simply changes the ended variable to true, meaning the loop in run() stops, and then calls recordEnding(). This method is synchronized so that if two players win at the same time, the outputs are recorded one after the other, avoiding synchronisation issues. The attribute ending is volatile for this same reason.
- Card class with each card containing an attribute to store its numerical value.
  This was done instead of just using integers so that multiple cards with the same value are still unique objects. When the Card is instantiated it throws an error if the given value is less than 0.
- CardDeck class representing a deck with four cards which are changed throughout the course of the game.
  Each deck stores its cards in a queue of Card objects, as cards are always added to a deck at the bottom and always taken from the top, giving it a first-in-first-out rule of operations.
    - addCard() is called by the player who has this deck to their left, and it adds the Card they specify to the back of the queue. drawCard() is called by the player who has this deck to their right, and it removes a Card from the head of the queue and returns this to the player
    - The player to the deck's right will call writeResult() when the game ends, which uses getOutputString to assemble a string of the message to record to the deck's output file, deck[id]_output.txt, and writes this string to the file using a FileWriter.
- CardGame class where the game is run, containing the main() method, and other methods that perform the setup of the game and one that begins it.

The main method instantiates a Scanner for user input, and then simply calls all these methods in order: getNoPlayers(), which uses user input to get the number of players in the game, getPack(), which assembles an ArrayList<Card> pack, using a pack file specified by user input, createDecks(), createPlayers() and dealOutCards(), which set up all the objects for the game, and startPlaying(), which creates Thread objects containing each PlayerThread, and then starts all these threads.

Testing Choices

- We chose to use Junit 4 (4.13.2) for our tests, as this is what we had covered in the lectures and so it was most familiar to us. We made a Junit Suite class and a TestRunner class so that we could run tests together in the command line. The TestRunner main() method uses JUnitCore to run all the tests, then prints out any failures that occur and whether the test was successful or not.
- Each class in the program has its own testing class (TestCard, TestCardDeck, TestCardGame, and TestPlayerThread), each having methods to test every necessary method in the class.
- TestCard only has one method, testCard(), due to the simplicity of the Card class. It tests whether the Card can output the correct value that it was instantiated with, and then makes sure that if it is given a negative value, the program throws an error.
- TestCardDeck has several methods to test each method in the CardDeck class:
    - testAddCard() creates a deck and adds several cards to it. It then draws these cards to assert that their values are what they should be, following a FIFO structure
    - testDrawCard() tests that a deck can draw 4 cards, where 4 were added, and then makes sure that the program throws an error if an attempt is made to draw a 5th card that doesn't exist.
    - testOutputString creates a deck with a set of cards and tests whether the output string it assembles is formatted correctly and contains accurate values.
- TestPlayerThread has methods to test the methods in the PlayerThread class:
    - testAddCard() asserts that a newly instantiated player has a hand of size 0, and then when a card is added to its hand, has a hand of size 1.
    - testGetInitString() creates a player with a set of cards in its hand, and tests whether the output string for the initial recording has the correct formatting and values.
    - testGetMoveString() does a similar test to testGetInitString(), but for the output string of a player's move.
    - testGetEndingString() does a similar test to testGetInitString() and testGetMoveString(), but for the output string of a player when the game ends, making sure that the string is different if the player in question is the winner or not.
    - testDrawCard() sets up a CardDeck to be the player's left deck with 4 cards. It then gets the player to draw each card, making sure the cards are drawn in the correct order according to a FIFO structure. It will then try to draw a 5th card, making sure that an error is thrown, as that card will not exist in the deck.
    - testDiscard() sets up a player with 4 cards, and then calls discard() for each card. It then calls rightDeck.drawCard() to test if the value of the cards in the deck to the player's right match those that the player discarded, and that they are in the correct order following the FIFO structure.
    - testCheckWin() sets up two players, one with a hand of 4 of the same card and one with 4 mixed cards. It calls checkWin() on each player and makes sure that the

    player with the 4 identical cards returns true, implying a win, and that the player with the 4 mixed cards returns false, implying no win.
  - testChooseDiscard sets up a player with 4 mixed cards, one of which is of the player's preferred value. It then performs chooseDiscard() 20 times, each time testing that the returned card is not of the preferred value (the player should hold on to this card every move). It repeats this using another player with a different preferred value, making sure that the cards returned never matched the preferred value.
- TestCardGame tests the methods in the CardGame class, which are used in CardGame's main() method to run the game. The setup for this testing class involves explicitly creating a pack of Cards, as the test cannot involve user input to retrieve a pack file. The setup also includes a badPack that will causes errors when used.
  - testCreateDecks() calls the createDecks() method with 4 players, and checks that the decks attribute is of size 4 once created.
  - testCreatePlayers() creates decks, followed by 4 players using createPlayers(), testing that the players attribute has a size of 4 once finished.
  - testDealOutCards() creates 4 decks and 4 players, then calls dealOutCards() to populate them using the badPack, testing that an error is thrown. It will then try again with the valid pack, checking that each player and deck starts with 0 cards, and contains 4 after dealOutCards() is called.
  - testStartPlaying() initialises 50 games with 4 players each, all using the valid pack. It calls startPlaying() for each of these games, and should terminate smoothly if each game finishes without breaking. This is essentially an integration test rather than a unit test.