Cover Page:

50/50 split plz thx much apprciated

Development Log

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Date | Time | Duration | Driver | Observer | Joshua Sign | Thomas Sign |
| 8/11/22 | 1:00 pm | 2 hr | Thomas | Joshua |  |  |
| 10/11/22 | 11:45 am | 1.5 hr | Joshua | Thomas |  |  |
| 11/11/22 | 1:30 pm | 1.5 hr | Thomas | Joshua |  |  |
| 12/11/22 | 12:00pm | 2 hr | Joshua | Thomas |  |  |
| 13/11/22 | 2:00 pm | 1.5 hr | Joshua | Thomas |  |  |
| 15/11/22 | 1:30 pm | 2 hr | Thomas | Joshua |  |  |
| 19/11/22 | 2:00pm | 3.5 hr | n/a | n/a |  |  |

Design Choices

We do not have any known performance issues.

Table

Description automatically generatedWe made a thread-safe card class. The denomination attribute stores the integer value of the card. To make it thread-safe we implemented the synchronized keyword in the getDenomination method.

Table

Description automatically generatedWe decided to make a BasicWrite class which allows for us to re-use the same code for any class to write to a file, preventing redundant (duplicate) code. The out attribute stores the BufferedWriter to allow this to output to a file. This is used by both the Player and CardDeck classes to output to a file.

Chart

Description automatically generated with medium confidenceWe decided to make a custom event for when the game ends i.e. when a Player has won. It extends the built-in exception class.

Diagram

Description automatically generated

We decided to make an interface listener for the EndGameEvent to alert all Players and Decks when the game is over. This prevent one player continuing to continue to play one a player has declared themselves the victor. It includes the eventOccured method to show what classes do when the game ends

Diagram

Description automatically generated

We designed the CardDeck class to implement the EndGameEventListener. The cards attribute stores an ArrayList of cards, as it can vary in size and needs to act as a queue. The output attribute stores a BasicWrite object so can write to a file. The deckName attribute stores the name of the deck to be used in its output. To allow for full functionality we created various methods. Each deck acts a queues which means when the respective methods of addCard and removeCard are called it will always draw from the front of the cards ArrayList and always places the discarded card to the end of the cards ArrayList. The final method excluding getters and setters is the eventOccured method which overrides from the interface. This then makes the decks write to a file with the current contents of the deck.

A picture containing diagram

Description automatically generated

We made a thread safe player class. This class extends the Thread class and implements the EndGameEventListener interface. The playerName attribute stores the name of the player to be used in its output. The number attribute stores the player number, used for their desired card. The leftDeck and rightDeck attributes store the CardDecks that the player pick up and discard too respectively. The output attribute stores the BasicWrite object used to write to its file. The selectDiscardCard method chooses a random card from the player’s hand which is not of their preferred number and returns the index of said chosen card. The randomness comes from shuffling an ArrayList of possible candidates (i.e. not preferred number) and choosing first. The addCard method adds a card into the player’s hand if it is not full. The removeCard method removes and returns the card from the given index. The checkWin method returns true if a player has won, false otherwise. The overwritten method eventOccured causes the player to output the final messeges to its respective output file when someone has won the game. The overwritten run method creates a gameplay loop where the player checks if they have won, creating an EndGameEvent if they have, if they have not won they check if someone else has won then continues with the game. They draw and discard a card atomically and write their actions to their output file.

Letter

Description automatically generated with low confidence

We made an executable class to set up and run the game. The numPlayers attribute stores the number of players in the game to assist with Player/CardDeck creation and card dealing. The mainDeck attribute stores the complete deck of cards from the card file. The players/decks attributes are ArrayLists which store the Player/CardDeck objects respectively, as they could be of any length. These are static so when a player wins they can easily notify all EndGameEventListeners (all players and decks). The winner attribute stores the name of the winning player and is static so all players can access. The setPlayerDecks functions assigns the players with the decks they should pick up from/discard to. The dealCard method gets the card from the top of the complete deck. The dealCards method deals the entire deck between the players first and then the decks. The getNumPlayers method takes the user input for the number of players of the game.The getDeck method takes the user input and validates the given file is a valid deck for the number of players. The main method gets the number of players and deck from the user by the aforementioned methods before creating the correct number of players and decks, shuffling the deck, dealing out the cards and starting the game (by starting each of the player threads. We chose to shuffle the deck to increase the randomness of the game and reduce its predictability.

Testing Design Choices

We used the Junit 4.13.2 framework for testing. We wrote the following tests:

TestBasicWrite

* testBasicWriteConstructor
  + Creates a BasicWrite object.
  + Checks that an object has been created (i.e. is not null).
  + This tests that the BaiscWrite constructor creates a new object successfully.
* testWriteToFile
  + Creates a BasicWrite object and gives it text to write to a file.
  + Checks that the result of reading from this file is equal to what was given to write to the file.
  + This tests that the object can write to a file successfully.

TestCard

* testCardConstructor
  + Creates a Card object.
  + Checks that an object has been created (i.e. is not null).
  + Checks that the denomination matches given.
  + This tests that the Card constructor creates a new card successfully, with the correct attributes.
* testDenomination
  + Creates a new card object and assigns it a denomination.
  + Checks that the assigned denomination is equal to what it was given.
  + This tests that a Card stores the correct denomination.

TestCardDeck

* testCardDeckConstructor
  + Creates a CardDeck.
  + Checks that a deck has been created (i.e. is not null).
  + Checks that the deck has created an ArrayList for Cards.
  + This tests that the CardDeck constructor creates a new CardDeck successfully with the correct attributes, creating the data structure to store Cards in the process.
* testAddCard
  + Adds cards to a deck.
  + Checks that they have been added in the correct order by comparing their denominations with the ones given.
  + This tests that Cards are successfully added to the bottom of the deck.
* testRemoveCard
  + Adds cards to a deck.
  + Checks that the removed card was the one first added to the deck.
  + Checks that the length of the deck has decreased (i.e. the card has been removed from the deck).
  + This tests that cards are always drawn from the top of the deck.
* testEventOccured
  + Adds cards to a decks then triggers an EndGameEvent.
  + Checks that the correct message is written to the correct file.
  + This tests for the correct output when a player wins the game.

TestCardGame

* testSetPlayerDecks
  + Creates two players and decks.
  + Checks that the players have been assigned the correct decks.
  + This tests that each player is assigned the correct deck on their left and right.
* testDealCard
  + Creates a main deck.
  + Checks that it returns the top card from the deck.
  + Checks that the length of the deck has decreased (i.e. the card has been removed from the deck).
  + This tests that cards are successfully removed from the top of the main deck.
* testDealCardException
  + Creates empty main deck.
  + Checks throws exception when tries to deal from empty deck.
  + This tests that an exception is thrown if trying to get a card from the main deck if the main deck is empty.
* testDealCards
  + Creates players and decks, then deals the cards.
  + Checks that each players’ hand doesn’t conatin any null values (i.e. contains 4 cards)
  + Checks that each deck has four cards in it.
  + This tests that four cards are dealt to all players and decks from the main deck.

TestPlayer

* testPlayerConstructor
  + Creates a player object.
  + Checks that the player object is not null.
  + Checks that the player’s number is the given number.
  + This tests that the Player constructor creates a new player successfully, with the correct attributes.
* testAddCard
  + Creates a player and adds cards to their hand.
  + Checks that the cards are successfully added to their hand.
  + Checks that after a card is removed a new card can be added.
  + This tests that a card can be successfully added into a player’s hand if their hand is not full.
* testAddCardException
  + Creates a player and tries to add five cards to their hand.
  + Checks that a HandFullException is thrown.
  + This tests that an exception is thrown if try to add a card to a player’s hand when their hand is full.
* testRemoveCard
  + Creates a player and fills its hand with cards.
  + Checks that the card removed is equal to the expected card.
  + Checks the array in the position of the removed card is null.
  + This tests that the card at the given index is removed from the player.
* testRemoveCardException
  + Creates a player.
  + Checks that an IndexOutOfBoundsException is thrown.
  + This tests that an exception is thrown if try to remove a card from an invalid index.
* testSelectDiscardCard
  + Creates a player and adds three cards of the preferred value and one of unpreferred value.
  + Checks that the unpreferred value card is selected for discard.
  + This tests that a card which is not of the player’s preferred denomination is selected to be discarded. We cannot test the randomness of the selected card.
* testCheckWin
  + Creates a player with a winning/non winning hand.
  + Checks that returns true if player wins.
  + Checks that returns false if player hasn’t won.
  + Checks that returns false if player’s hand is not full.
  + This tests that a player can be successfully identified as having won or not. It also makes sure that no error occurs if a player’s hand is not full.
* testEventOccured
  + Creates two players, one of which has won the game.
  + Checks that the file output is correct for the player who has won.
  + Checks that the file output is correct for the player who has not won.
  + This tests that the correct information is outputted to a file when the event occurs.
* testRun
  + Creates a player and scenario so the player wins after one draw.
  + Checks that the deck the player picked up from has decreased in size (i.e. player has successfully drawn from it)
  + Checks that the deck the player put down onto has increased in size (i.e. player has successfully discarded a card onto it)
  + Checks that the player output to the file are as expected.
  + This tests that the player outputs the correct information on game start, during play and on game end. Also checks that the player picks up and discards to the correct decks.