Card Game

Report by Will Harris and Isaac Cheng

Development Log

23/10/2020 – 12:45pm (5 hours) [*Signed: Will, Isaac]*
Started with Will as driver and Isaac as navigator. Created *CardGameTest.java* to test methods *validateNumPlayersInput*, *validatePackInput*, and *importPack*, which will be used to handle input for number of players and input for the pack of cards respectively, then import the pack of cards from the text file if it is valid. Created *CardGame.java*, and added input for number of players and location of pack to load in the main method, with auxiliary methods *validateNumPlayersInput* and *validatePackInput* to handle these inputs, and *importPack* to use the pack input. Ensured code passed unit tests.

Swapped roles (Isaac as driver, Will as navigator), then added test methods to *CardGameTest*.java for *dealCards*, which will deal cards to players, *countFrequencies*, which will check if pack of cards is guaranteed to be winnable, and *genHashMap*, which will generate a HashMap of value-frequency pairs of cards in the game. Developed methods *dealCards, countFrequencies*, and *genHashMap* ensuring code for these methods passed unit tests. Created *Card.java* with setter and getter methods for card values and holders. Created *CardDeck.java* with methods for deck transactions.

25/10/2020 – 12:30pm (3 hours 30 minutes) [*Signed: Will, Isaac]*
Started with Isaac as driver and Will as navigator. Created *PlayerTest.java* to test a threaded class which will handle actions of the player, with a method *play* which will continually draw and discard cards until a winner is found. Created threaded class *Player.java* with method *play*, and ensured the code passed the unit test.

Swapped roles (Will as driver, Isaac as navigator). Added a test method in *PlayerTest.java* for a new method *isWinner*, which will perform checks for a win condition and declare the winner of the game. Created a new method *isWinner* in *Player.java*, and ensured the code passed the unit test.

30/10/2020 – 12:30pm (3 hours 30 minutes) [*Signed: Will, Isaac]*
Started with Isaac as driver and Will as navigator. Added test methods in *PlayerTest.java* for *strategy*, which will help the player decide to do with a newly drawn card depending on cards in their hand, and auxiliary methods *keepCard*, *removeCard*, and *discardCard* to manipulate the cards depending on the scenario. Added a test method for *chooseDiscard*, which will be an auxiliary method to *discardCard* to help it choose which random card from the hand to discard. Created the method *strategy* and its auxiliary methods *keepCard*, *removeCard*, and *discardCard*, as well as *chooseDiscard*, and ensured the code passed the unit tests for these methods.

Swapped roles (Will as driver, Isaac as navigator). Added test methods in *PlayerTest.java* for *createFile*, which will create an output file for each player's actions, *writeToFile*, which will write to these files when a game action has occurred, and *viewArray*, which will display the cards in the hand or deck of a player. Created methods *createFile*, *writeToFile*, and *viewArray* in *Player.java*, ensuring that the code passed unit tests.

6/11/2020 – 1:00pm (4 hours) [*Signed: Will, Isaac]*
Started with Isaac as driver and Will as navigator. Reviewed all methods in existing classes, optimising them for efficiency in terms of speed and storage by removing redundant code and redesigning some decision algorithms. Created a test suite to run our different test classes.

Design Analysis

To help with the decomposition of the aspects of the game, we created two simple classes, *Card.java,* and *CardDeck.java*, which will handle the individual cards in the game and decks of each player in the game, respectively. These two basic classes are essentially driven by each player, meaning the breakdown of these objects into two classes will represent the abstraction of the players' relationship with the cards and decks.

Card.java creates card objects so that after importing the pack of cards from the text file, the cards can have their values checked. This makes it easier later in the game for checks to be performed, which will decide what each player does with the hands they draw in their hand. *CardDeck.java* creates card deck objects, which makes it easier to manage which cards are in every player's deck, and transferring ownership of cards between players. It should be noted that in the decks (and hands for that matter), the indices go from top to bottom, where the 0^{th} card is the top, and the final card in the index is the bottom.

At the start of the game, the program asks for the user to input the number of players in the *main* method. It uses the *validateNumPlayersInput* method to check if an integer greater than two is inputted. We created a custom exception, *IllegalNumPlayersSizeException*, which is thrown when the input is a valid integer, but it is smaller than two, as there must be at least two players in the card game. When a valid integer is not inputted, *NumberFormatException* is thrown, as the input would not be able to be parsed as an integer. Then, the game iterates this number of times to create a player object and a deck object for that player. Creating deck objects for each player makes it easier to decompose problems later when it comes to drawing and discarding cards.

In the main method, it also asks the user to input the file name for the pack of cards to be used. When loading the pack of cards in the *importPack* method, the program checks that the values in the text file are all positive integers, and that there are enough cards for the number of players. The pack of cards is implemented as an ArrayList because this allows us to dynamically add and remove cards from the pack as cards are transferred to players' decks.

In *dealCards*, the method keeps track of the size of players' hands through the getter method *getHandSize* to ensure that it continues dealing directly to their hands until every player has four cards. After that, it sequentially adds cards to players' decks with the *addToDeck* method.

The *genHashMap* method is used to generate a HashMap of key-value pairs of the cards in play, where the key represents the value of the card, and the value represents the number of these cards in the game. This is used in the *countFrequencies* method, which iterates through the players, and analyses their chances of winning based on the number of cards with their preferred value. If the player has fewer than four cards of their preferred value, but there are at least four cards with the same value, then a winner is not guaranteed from the game, and it may stagnate, but the game will continue running because there is still a chance of a winner. The game will not continue running if there are not at least four cards with the same value, because it would be impossible for a winner to be found.

We created a class *Player.java* to handle the decisions of each player in the game, as this helps to segregate decisions made by the game (like a referee managing the game), and the players.

To draw a card from the player's deck, the method *draw* takes the last card from the top of the deck on their left, removes it from the deck, and then discards a card to the bottom of the deck on their right. After every draw, the player's hand is printed using the method *viewArray*, which looks at the player object and iterates over the hand to check each value.

Immediately drawing a new card, the method *strategy* looks at the player's hand of cards (excluding the card which has just been drawn), and randomly discards one of the cards which is not their preferred value.

Auxiliary functions *keepCard*, *removeCard*, and *discardCard* are used for the player to perform the appropriate actions according to the game scenario. The *keepCard* method takes the card, player number, and player array as an argument, then adds the card to the player object's hand. The *removeCard* method takes the card, player number, and player array as an argument, then removes the card from the player object's hand. The *discardCard* method takes the card, player number, and player array as an argument, then discards the card to the deck of the next player.

To ensure that this method discards the card randomly, an additional auxiliary function *chooseDiscard* is used. It uses the built-in *nextInt* function, which returns a pseudorandom integer value between 0 and 3 inclusive, meaning the newly drawn card will not ever be removed. If the card at this index does not have a value which is preferred by the player, then it will return that index to the *discardCard* method.

To track the activities of players as actions in the game occur, we made a method *createFile*, which creates an output .txt file for every player in the game in the format *player[NUMBER]_output.txt*, and for every deck of each player in the format *deck[NUMBER]_output.txt*. The additional method *writeToFile* is used to write strings into the text file on new lines after an action in the game is made, taking in an argument *delim* which helps it decide whether it should write to a player output file or a deck output file, and an argument *writeString*, which is the description of the game action which has just occurred. When events occur, it will call *writeToFile* with the relevant *delim* and *writeString* so that the method is used with the appropriate context.

Will method *isWinner* is used to check whether a player has the appropriate cards in their hand to win the game. As there may be multiple players with a complete hand at the same time due to the threading, the method checks if a winner has been found before overwriting the winning player's integer. If a winner has been found, then it stops players from continuing to draw and discard cards in the *run* method.

We encountered no performance issues in our production code; all the code was executed in a timely manner on our two Windows 10 systems which we used for development.

Test Design Analysis

For our testing, we decided to use JUnit 4.12, as this seems to have the most support, due to having been used for a longer period. We took a test-driven development approach, and we sought to abide by best practices for testing, as outlined by JUnit themselves (https://junit.org/junit4/faq.html#best_1).

We created test classes for major classes, and created test methods to test each method in these classes where necessary. Tests were written before the code, which helped us to get an idea of how we should write the code to be concise and fit for purpose. Test-first programming was practised, only writing new code when an automated test was failing.

When we designed our tests, we focused on passing the Right BICEP tests. This included checking that the boundary results were right, that the boundary conditions were correct, and that we could force error conditions to occur. We also considered the ATRIP acronym to check that we had properties of good tests; our tests were designed to be automatic, thorough, repeatable, independent of other tests and the environment, and professional.

A pragmatic approach was taken in that tests were not created for everything; we simply tested everything which could reasonably break. Where possible, we adapted our design so that testing would be possible. For example, when seeking to take inputs from the user, we took the input from the main method, and then processed them in auxiliary methods which took the input as an argument. This ensured that we could test different inputs, and check whether they were being validated correctly in the auxiliary methods. Getters and setters are examples of methods which we chose not to test, as these are simple methods with no logic to test. Hence, we did not have a test class for *Card.java* or *CardDeck.java*, as these classes consist of only getter and setter methods.

The benefit of our test-driven approach was that the code developed was less likely to include bugs, as they would have been exposed through the unit tests created beforehand. We had very few problems with the functionality of our code, which left us with more time to optimise our code for efficiency.

Creating the tests for the *CardGame.java* class were easier because the scenarios were generally simpler, and hence easier to break down.

At the start of the game, the game asks the user to input the number of players in the game, which was then validated with the function *validateNumPlayersInput*. We broke this down into four scenarios; the user can make a valid input of a positive integer greater than 1, an invalid input of an integer which is too small, an invalid input of an integer which is negative, or an invalid input of a string (which cannot be parsed as an integer).

In the first case, we used *assertEquals* to check whether our number (which is taken in as a string, like most inputs in Java) was correctly parsed as an integer, returning an integer number of players. For the next two cases, we checked whether the input was caught by our custom exception, *IllegalNumPlayersSizeException*, using *assertTrue*. For the final test case, the *NumberFormatException* was expected, as the string input cannot be parsed as an integer, so we checked that this exception was thrown by using the *fail* method of JUnit4.

For the pack of cards to be imported, the user must input the file name, and this is validated in *validatePackInput*. It is important that this is validated correctly, as the pack of cards is essential to the running of the game. We created a test case and the *fail* method to check that when an invalid file name is given, it throws the exception *FileNotFoundException*.

The method *testImportPack* tests whether the game correctly parses a file for a pack of cards. We created a test pack of cards, *testCardPack.txt*, and designed it for a three-player game, containing values of only 4. This method checks the properties of the ArrayList which results from parsing the

text file; it should have 8 * 3 = 24 lines, with every line containing ‘4’. If this is not the case, it will fail the *assertFalse* statement at the end of the test, as *notIdentical* will be set to true.

testGenHashMap tests the *genHashMap* method by inputting a test card pack which contains sixteen cards of value 2, and calling *genHashMap* on that card pack. It uses *assertEquals* to check whether the frequency of value 2 in the resultant dictionary is equal to 16 as expected.

testCountFrequencies uses the same card pack to check whether the game would be played when used for a two-player game, using *assertTrue* to test this.

The tests for the *Player.java* class were more complex, because the scenarios often involved a lot of decision making. However, this did also mean that it was more important that the tests for this were designed thoroughly, as it was important for edge cases to be discovered and accounted for when designing the code.

Throughout the unit test, we needed to use mock objects. This is because the real objects would be complicated to set up, and would have behaviour which would be difficult to trigger due to the random aspect of discarding cards. These mock objects enabled us to test methods from the *Player.java* class, which would otherwise not be the case due to the nature of the class involving decisions related to the player. Every test method in this unit test involved the use of mock objects.

We created mock objects for a player with the method *playerSetUp*, mock objects for a deck with *deckSetUp*, mock objects for an array of players with *playerArrSetUp*, and mock objects for an array of decks with *deckArrSetUp*. All these methods take place before the tests are performed, as the tests would not work without them; this is denoted with the *@Before* annotation above these methods. The teardown of the mock objects occurs after all tests have been completed, as denoted by the *@After* annotation above the *teardown* method. This is necessary as a basic optimisation technique because the mock objects serve no purpose after the unit testing is complete. By setting all the objects to null, it makes them eligible for the automatic garbage collection which occurs in Java, hence saving memory.

We created several tests for the cards in the game to check that the cards are being manipulated and transferred to and from players correctly. The method *testAddToHand* creates a mock object card of value 7, and then adds this card to the hand of a mock player object. As the player object from *playerSetUp* was created to start with four cards in the hand, the size of the player’s hand after adding the card should have increased from four to five, and the final card in the player’s hand ArrayList should have value 7. The test uses two *assertEquals* statements to check that this is true.

The method *testRemFromHand* works in a similar way. It removes the first card from the mock player object’s hand, then checks that the hand size decreased from four to three.

Meanwhile, *testDrawValue* creates a mock deck1, then adds cards to this deck with values 1, 2, 3, and 4 iteratively. This means that the first card in the hand should have a value of 1. During the iteration, when the card with value 1 is being added, *getValue* is used to set 1 as the expected value, and this is tested against the deck’s first card value using *assertEquals*.

Testing the *chooseDiscard* method was more difficult, as it involves randomly picking a non-preferred value from the player’s hand to discard. However, this card must be a non-preferred value, so we created *testChooseDiscard* to create a mock player2, and add three cards with values 2, and one card with value 1. Then, *chooseDiscard* is run to select the index to remove, and the card is removed from the hand. At the end, the test iterates through the mock hand, and checks if all cards have value 2 using *assertTrue*.

The method *testDiscardCard* involves testing two logic paths. It tests that the discarded card is transferred from one player’s hand to the top of the deck of the player on their right, so player1’s

card should go to player2's deck. The first block creates a mock object player2, and discards a card of value 15, which should go to the top (last index) of player3's deck. It tests that this is automatically done by transferring it to the (n+1)'s deck if the nth player is not the last player in the game, using *getValue* to get the value of the fifth card in player3's deck ArrayList, and compare it with value 15 in *assertEquals*.

When the nth player is the last player, it should loop back to discard the card to the first player's deck, which is tested in the second block of the test. In a mock three player game, it discards a card of value 17 from player3, and checks that this moves the card to the top of player1's deck using *getValue* to get the fifth value of player1's deck ArrayList, and compare it with value 17 in with *assertEquals.*

testKeepCard works similarly to *testAddToHand*, as keeping the newly drawn card should provide the same result as adding a card to the hand. It creates a mock player2, then calls the *keepCard* method to keep a mock card of value 11. It checks that it was kept in the hand correctly by using *assertEquals* to compare the mock card to the fifth card in player2's hand.

testRemoveCard creates a mock player3, then gets the first card in its hand, and removes it by calling *removeCard* on that first card. This means that the size of player3's hand should have decreased from four to three, which the test checks by using *getHandSize* to check its new hand size, and compare it with the value 3 in *assertEquals*.

Our final test, *testIsWinner*, creates a mock player2, then adds four cards with value 2 to its hand iteratively, and calls the *isWinner* method on player2. This method tests that the correct winner is being declared by checking if *isWinner* successfully sets the *complete* Boolean value to true, and if it checks that the winner value is set to player2.

To check the effectiveness of our testing, we ran each unit test with code coverage in our IDE, as code coverage is often a good metric of thorough testing. *CardGameTest.java* covered 71% of methods, whilst *PlayerTest.java* covered 68% of methods, indicating that the classes were tested to a high standard.

After we had finished designing the unit test classes, we added it to a test suite, *TestSuite.java*, which can be used to load a database in memory, run all test cases, and then unload the database from memory after the suite has completed the testing. This saves time, and makes it easier to run all tests if new tests are added to the unit test classes in the future.