Software Development (ECM2414)

Programming Report

2021

Cover Page

Mark Allocation:

50:50

Production Log

The following is the development log for the project, we have included the date, time and duration of each session and the role that each member took in that session. The final two columns show the role we took identified and signed with by the candidate number at the top of the column.

Date	Time	Duration	105950's Role	154026's Role	Signed By
25/10/2021	12:00	120	OO Design	OO Design	105950 and 154026
27/10/2020	14:00	90	Driver	Observer	105950 and 154026
27/10/2020	15:30	90	Observer	Driver	105950 and 154026
30/10/2020	11:00	90	Observer	Driver	105950 and 154026
30/10/2020	12:30	110	Driver	Observer	105950 and 154026
01/11/2020	15:00	120	Observer	Driver	105950 and 154026
02/11/2020	20:00	70	Driver	Observer	105950 and 154026
02/11/2020	21:10	70	Observer	Driver	105950 and 154026
04/11/2020	12:00	90	Driver	Observer	105950 and 154026
04/11/2020	13:30	90	Observer	Driver	105950 and 154026
05/11/2020	09:30	120	Driver	Observer	105950 and 154026
05/11/2020	11:30	120	Observer	Driver	105950 and 154026
06/11/2020	13:30	90	Observer	Driver	105950 and 154026
06/11/2020	15:00	75	Driver	Observer	105950 and 154026
08/11/2020	20:00	105	Driver	Observer	105950 and 154026
08/11/2020	22:45	70	Observer	Observer	105950 and 154026
09/11/2020	10:00	120	Observer	Driver	105950 and 154026
09/11/2020	20:00	120	Driver	Observer	105950 and 154026
10/11/2020	12:00	60	Driver	Observer	105950 and 154026
10/11/2020	14:00	180	Final Checks	Final Checks	105950 and 154026

Production Code Design Choices

Pebble Game

The class PebbleGame is what made up the majority of our code and is where we implemented the threadable player class. In the main method we started off by initialising all important objects, beginning with the user interface. This projects the title message, detailing what inputs are required and then allowing the user to input the number of players, as well as the locations of the three files used for the weights of the rocks. From here our main method creates all the bags, players, and finally the threads which all get started so the game can play out.

We made our Player class threadable by using “implements Runnable” as we felt it worked best in the structure of the game. We could focus all of the games elements into the run() method, making it easier to read. The player is initialised with an index so we can see what number they are and it also creates an output file which is where all of their draws and discards are read to. Inside run() we call the beginGame() method which gives the player 10 rocks randomly from any black bag, it did not need to be synchronized as it does not reference any bags itself. A while loop in run() calls hasWon() to check if the players total weights adds to 100, if not then the inside of the loop can run, this felt the easiest way to ensure the player continued to play if they did not hit the target weight.

In the loop the player first calls discardRock(), this needed to be synchronized as it references and calls elements of a white bag which should only be able to be accessed by one thread at a time. The method removes a random rock from the players hand and then decreases the size of the players rocks by 1. It also adds the removed rock to the white bag which our bag handler pointed to next, and finally, it writes the discard message to the player’s file.

Next, we call addRock() and pass drawRock() as the argument because it returns a rock once called. DrawRock() needed to be synchronized as it calls black bags as well as white bags which means it needed to be thread safe. Inside the method it selects a random black bag and checks that it isn’t empty, if it is then it calls the drain white bag method from its assigned white bag. Otherwise, it tries getting a random rock from the black bag, removing it from the bag and updating which white bag should be discard to next with the bag handler class. It writes the draw rock message to the players file and finally returns the rock. We added error handling to make it clearer if our code wasn’t working, giving us a reason as to why.

AddRock() was a simple method and did not need to be synchronized. It created a new list of rocks, one larger than the current list, and added all the current rocks to the list. It then added the new rock to the end and set the original rocks list to the new rocks list. It also calls the output file handler to write the new list of rocks into the players output file.

These actions continue to loop over and over until hasWon() returns true. Once this happens, “Player X: Has Won” is outputted to show which player won the game and then System.exit(0) is called, stopping all of the other threads from continuing to play the game.

Bag

The Bag class is what both BlackBag and WhiteBag extend from and contains the method addRock which is very important for the whole game to work. AddRock() creates a new list of rocks, longer than the original by 1, and adds all of the original rocks to the list. It then sets the last place in the list to be the new rock and then updates the original rocks list with the new one. The class also contains a getter method for the rocks list as it is a private attribute.

Bag Handler

We created the Bag Handler class to allow us to point to the next white bag to discard to when we draw a rock from a black bag. It has an attribute nextDiscard which is a white bag and two methods, a getter for this bag and a method which is a setter for the nextDiscard. The setter is contained in the drawRock() method in PebbleGame, it needed to be synchronized so that two threads can’t change it at the same time. We decided that this class will be initialised for each player so they all have a bag handler object which they can use to update where next to discard their next rock to. We believed this was the best way to ensure that all the rocks are allocated to the correct white bag.

Rock

The Rock class has an attribute of weight, a constructor and two methods. The constructor only needs an integer as a parameter and simply sets it to the rocks weight attribute. GetWeight() is a getter for the value of weight and the toString() method is to returns the rocks weight. We felt that using a class to represent a Rock was necessary as it maintained the object-oriented principles we were following when designing this system.

Black Bag

The BlackBag class has attributes, assignedWhiteBag, rocks, and number. The constructor takes in three parameters which are used to set all three attributes. It also makes sure the white bag is also linked to that black bag by using the assignBlackBag method in WhiteBag. RefillBag() is used for adding rocks to the black bag and works like all of our other add rock methods. RemoveRock() also works like all of our methods that discard a rock from a list and uses the index of the rock to do so. There are three getter methods for retrieving the bag number, the assignedWhiteBag and finally the rocks list. Lastly there is a toString() method for formatting the output when a black bag is called.

White Bag

The WhiteBag class has attributes assignedBlackBag, rocks and number. The constructor just sets the list of rocks to an empty list. AddToWhiteBag() takes in a rock and adds it to the bags list of rocks, it is synchronized as we don’t want multiple threads trying to change the length of rocks at the same time. DrainWhiteBag() just empties out the whole bag into it’s assigned black bag once the black bag is empty and then updates the rocks list to be an empty list. AssignBlackBag() just makes the WhiteBag have the same number as the black bag and assigns it the black bag to point to. There is also a getter for the rocks list and finally a toString() method to display the WhiteBags nicely when referenced.

Output File Handler

This class is used for creating output files for every player that is playing in the pebble game. It only has filename and playerName as attributes. The constructor takes in the player’s name and creates a file with the same name and ends with “_output.txt”. We also made it so that if there is already a file with the same name it will add a “_v” and the number of the version to avoid duplicate files. The writeMessageToFile() method simply takes in a string and writes it to the file related to the player. WriteDrawnRockMessage() and writeDiscardRockMessage() are both used to display when a player is drawing or discarding a rock. They take in the rock weight and bag number and format the string which is then passed to writeToFile(). Finally, writeAllRocksMessage() just takes in a string of the rocks the player is holding and formats a phrase that again is sent to writeToFile().

User Interface

The UserInterface class is for taking the input in from the user at the beginning of the game. It creates a scanner and has a getter method that takes the next line and returns the input as a string. There is a checker for if the user has tried to exit the program with an E and if so will exit the program for them. It has a method to display the title message, display an error message and ask for the file locations for bag locations. Finally there is a method called askNumPlayers() which checks that the user inputs a non-negative, integer and it outputs error messages until the user actually does so correctly.

Test Code Design Choices

Planning the Tests

When planning our project, we decided that we would focus our testing on unit testing due to how it allows us to test individual blocks of code in terms of correctness at the function level, as our platform is designed to follow object-oriented principles this means tests are carried out at the method level. We have also used integration testing to ensure that the different objects are able to work together to form the final system. We chose to include these types of tests as it is common that unexpected behaviour may occur during these tests hence, we believed they were important to be included.

To create these tests, we have used the Junit5 framework through its standalone console library. This was to allow us to carry out tests I our standard IDE Visual Studio Code where we can then use the "Test Runner for Java" extension to control and manage the test files. We chose to use this framework because Junit allows you to write test cases while developing which therefore helps to detect issues early on, due to how the framework in collaboration with Visual Studio Code allows the programmer to swiftly run the tests we were able to always ensure the tests continued to pass wen the code was modified and before it was pushed to the GitHub.

System Testing

Once the system was complete, we designed several test input files to allow us to test the entire system in a number of scenarios to ensure that the system responds in the way we expect and want it to. We felt it was important to carry out these tests as well as our unit and integration testing to ensure that the system matches the specification and to expose any bugs which would have not been already thrown.

Firstly, we tested using different inputs for the number of players. To do this initially we tested entering inputs which were to be expected (non-zero positive integers) to check that the system excepts the inputs we want, next we tested entering 0 as one test case and 1 as the other to check the values on the boundaries and the finally we tested using an erroneous input where we expect the code to reject our input. We tested using a negative integer and then a character and finally a string to ensure the system returns an acceptable error message.

The next stage of the system we tested was entering the file location for the rock weights. We first tested entering file paths which followed the specified rules set out by the system to ensure that the system performs as expected. The next form of file which we tested was wen the file path does not exist; this test was to ensure that the system to throws an error which is handled appropriately. Finally, we tested when the files specified by the user do not contain rock weights in the correct format, these tests included when the files contain non positive values, not enough weights for the number of players specified and when there are empty spaces and elements in the file. The tests on these files were to ensure that the system does not accept them and produces an appropriate error message response. We felt it was very important to carry out these tests as they assure that the system is going to have the correct starting data and if there was an issue or bug here it would have a degenerating affect on the rest of the system.

Once we had carried out tests on the initial user inputs of the system, we next tested the overall performance of the system when correct entries are given. We carried out several tests using different input files and number of players to check that the system ran as expected and completed finding a winner. When using a input file with larger numbers the majority of numbers are greater than 25) it is unlikely to find a winner and rather run infinitely so we tested that the system was able to do this by allowing it to run for ten minutes before terminating it.

Unit and Integration Testing

Testing Bag.java

The Bag.java file contains the Bag class which is a parent class for both the Black Bag class and White Bag class. We first tested the constructor for the class by creating a new Bag object using the objects constructor, and then checking the created object has the correct class instance and correct initial attributes. Next, we tested the addRock method by creating a new bag, adding a rock to it using the method and finally comparing if the bag's rocks were now including this new rock. We did this because adding a rock to a bag is a key part of the game and if this were to fail, then the entire platform would not work as we would have wanted and expected. Finally, we tested the getRocks method to ensure it returns the objects rocks correctly. We did this by simply creating a new bag, checking the getRocks method returns an empty array of rocks as we would expect and then adding a rock to the bag and checking that the method now returns an array with the rock in. This was important to test as it ensured that the method always returned the most up to date version of rocks.

Testing BagHandler.java

The BagHandler.java file contains the BagHandler class which is used to control where next to discard a rock to. We first tested that the getNextDiscardBag method returns the expected result, being a WhiteBag. We decided to carry this test out in this way to ensure that the returned object would be of the right instance to contain any methods which the object would need to impliment. Next, we tested the updateNextDiscard method to ensure that it changed the objects nextDiscard attribute to the one given as a parameter. We tested that when you update the nextDiscard to a new WhiteBag that new WhiteBag is then returned when you call the getNextDiscardBag. This was using integration testing as it also tested that both methods worked together to return the most recent value for the attribute.

Testing BlackBag.java

The BlackBag.java file contain the BlackBag class which is a child class of the Bag class. It is used to hold all the rocks for a BlackBag and the majority of other classes use it's methods hence it is crucial to ensure they are thoroughly tested. The first test we created as testing the constructor of the object to check each Black Bag is instantiated correctly. We checked that when a Black Bag is created it of the correct class type, identifying number, assigned white bag and rocks. For the purpose of testing, we let the rocks just be an empty array of Rocks. The next test we created ensured that the refillBag method, which adds a new rock to the object's array of rocks was correct. We wrote the test so that we attempt to add a new rock then check that the rock is added successfully. Then we wrote a test the removeRock method, ensuring that removing a rock from the object's array works as expected. We checked that when you add a rock to an objects array of rocks, the length of the rocks array reacts as expected (decrements by one). The final three tests in the BagHandler tests are checking that the getter functions return the values we are expecting them to. We decide it was important to test that this was operating as we wanted to ensure that we didn't have any incorrect or stale date being returned by the methods.

Testing the Player Class

The Player class is a nested class inside of the PebbleGame.java file where the main method is stored. The player class is used to create the player objects which each thread runs, therefore its methods are primarily private methods hence when it came to testing, we had some issued but then decided to use Java Reflection to allow us to test the private methods. The first test we created tests the players constructor method, we tested that when you create a new player the constructor creates a player name and output file handler as we would expect. The next test we created tested the hasWon function, this test ensure that the method only returns true when the players rocks add up to a combined weight of 100. This is a crucial test as it ensures that the game only stops when a player actually has the correct rocks.

Testing Rock.java

The Rock.java file is used to store the Rock class which represents a rock or pebble in the game, these rocks are solely used to hold a rock's weight. Hence when it came to testing there were only two areas we needed to test. Firstly, that when the constructor method is run it creates a rock with the expected rock weight, we tested this with valid rocks however did not need to test any boundary or erroneous weights as they are handled when the platform reads from the file. The next test we designed and created tested that the get weight method returns the correct weight for the given rock object, we did this by creating a rock with a weight x, then ensuring that when we call the function x is returned. This was important to check as if this method produced erroneous data, then it would affect if the player has won or not and producing further issues late down the line of development.

Testing WhiteBag.java

The WhiteBag.java file is used to store the WhiteBag class which represents a white bag where rocks are discarded to by a player and then where BlackBags are refilled by when they become empty. The first test we created tests the constructor method of the WhiteBag class, we did this by creating a new WhiteBag object and then checking that the array of rocks stored in the White bag is empty as we expect. Next we tested that you are able to adda rock to the white bag, this is important to test as it is used when a player discards a rock it needs to be added to a white bag so it can later be added back into a black bag. The next method we designed tests for is the drain white bag method which loops through all the rocks in the white bag and adds them to the rocks of the asigned black bag. We tested this by creating mock objects for the black bag and white bag and then draining the rocks from the white bag into the black bag and then checking that the black bag has the expected rock moved over from the white bag. The final test tat we ran on the White Bag class was to check that the assign black bag method correctly assigns a black bag to the attribute. This was import for us to test as if the wrong bag was asigned then we would not have the correct white bags filling the correct black bags and therefore may cause certain black bags never to be refilled.