ECM24214 Software Development Pair Programming Coursework

Cover page (1)

We would like an equal 50:50 split, for Mati and Lai, 144211(Mati) 197348(Lai)

Development log (2)

Date	Time	Duration(hr)	Driver	Navigator	Candidate Signature	Notes
13/10/2021	14:45	1.00	-	-	144211(Mati) 197348(Lai)	First meeting for the general project direction and meeting arrangement
15/10/2021	14:00	2.00	-	-	144211(Mati) 197348(Lai)	Analysis of project requirements
16/10/2021	13:00	2.00	-	-	144211(Mati) 197348(Lai)	Initial project design and test planning and discussion
17/10/2021	13:00	1.00	Mati	Lai	144211(Mati) 197348(Lai)	Made Bag superclass
17/10/2021	14:00	1.00	Lai	Mati	144211(Mati) 197348(Lai)	Made Black & White Bag inherited class
20/10/2021	13:00	1.00	Mati	Lai	144211(Mati) 197348(Lai)	Beginning of PebbleGame class and Player class
20/10/2021	14:00	1.00	Lai	Mati	144211(Mati) 197348(Lai)	Start implementing draw and discard methods
22/10/2021	14:00	1.00	Mati	Lai	144211(Mati) 197348(Lai)	Implemented Run method
22/10/2021	15:00	1.00	Lai	Mati	144211(Mati) 197348(Lai)	Implemented main
23/10/2021	14:00	1.00	Mati	Lai	144211(Mati) 197348(Lai)	Cleaned code,made better use of exceptions
23/10/2021	15:00	1.00	Lai	Mati	144211(Mati) 197348(Lai)	Improved and formalised annotations and doc strings
24/10/2021	13:00	1.00	Mati	Lai	144211(Mati) 197348(Lai)	Improved structure of the program & formalised annotations and doc strings
25/10/2021	13:00	1.00	Lai	Mati	144211(Mati) 197348(Lai)	Fixed problem with reading file
25/10/2021	14:00	1.00	Mati	Lai	144211(Mati) 197348(Lai)	Tested all other methods other of Black & white bag
26/10/2021	14:00	1.00	Lai	Mati	144211(Mati) 197348(Lai)	Tested player class methods
27/10/2021	13:00	1.00	Lai	Mati	144211(Mati) 197348(Lai)	Tested pebble class methods
27/10/2021	14:00	2.00	Mati	Lai	144211(Mati) 197348(Lai)	Tested threading for 1 player
28/10/2021	13:30	1.00	Lai	Mati	144211(Mati) 197348(Lai)	Applying player log functionality
28/10/2021	14:30	1.00	Mati	Lai	144211(Mati) 197348(Lai)	Tested threading for more than one player
29/10/2021	13:00	1.00	Lai	Mati	144211(Mati) 197348(Lai)	fixed draw method and user input validation
29/10/2021	14:00	1.00	Mati	Lai	144211(Mati) 197348(Lai)	Fixed file writing and exception handling, tested program for 50+ users

30/10/2021	13:00	1.25	Mati	Lai	144211(Mati) 197348(Lai)	Cleaned code and further tested threading issues
30/10/2021	14:15	1.25	Lai	Mati	144211(Mati) 197348(Lai)	testing behavior of the write method, and threading
31/10/2021	13:00	1.00	Lai	Mati	144211(Mati) 197348(Lai)	Created junit test and IDE env
31/10/2021	14:00	1.00	Mati	Lai	144211(Mati) 197348(Lai)	Provided testing functionality for methods in Black Bag
03/11/2021	13:00	1.00	Lai	Mati	144211(Mati) 197348(Lai)	Fixed threading stop issues
03/11/2021	14:00	1.00	Mati	Lai	144211(Mati) 197348(Lai)	Fixed file write delay issues and cleaned code
05/11/2021	13:00	1.50	Mati	Lai	144211(Mati) 197348(Lai)	Finished unit testing for bags
05/11/2021	14:30	1.50	Lai	Mati	144211(Mati) 197348(Lai)	Started unit testing for player
06/11/2021	13:00	1.25	Lai	Mati	144211(Mati) 197348(Lai)	Planned testing for pebble game and created write-to-file test
06/11/2021	14:15	1.25	Mati	Lai	144211(Mati) 197348(Lai)	Finished unit testing for pebble game
07/11/2021	13:00	1.25	Mati	Lai	144211(Mati) 197348(Lai)	Wrote development part of report and tested program
07/11/2021	14:15	1.25	Lai	Mati	144211(Mati) 197348(Lai)	Attempt to make TestSuite in java
11/11/2021	13:00	3	-	-	144211(Mati) 197348(Lai)	Making jars preparing submission

Design of program (3)

PebbleGame

Pebble game is the main class of the program and contains the main, the class has 4 attributes, the games list of all player threads, a list of all bags and 2 game state variables called game_won and launch.

A list of threads and bags is necessary to access the objects created within PebbleGame, game_won alerts all player threads within PebbleGame that they should stop and launch tells players that they should start running as all threads have been made.

This was done so that all threads start at the same time and finish within each other as much as possible, the program will witness threads drawing and discarding after the win condition has been declared, this has been avoided with a System.exit(-1) command. The issue only seems to arise after player thread counts of around 250.

The pebble game asks the user for inputs like the number of players, and the files of which the black bags should be created from.

Inputs are managed by our own exceptions and make the user reenter an input till our conditions are satisfied.

Player

Player acts as a thread and runs the code, its attributes are just to do with player id, this increases incrementally with every new player thread made, 0,1,2,3…

The Player's run function first enters a while loop that breaks when launch is true, so all threads start as close to each other as possible, an initial condition is reached where 10 pebbles are drawn and then the main functionality is reached.

The player discards and then draws with a win condition being checked between each, this operation is done in a while loop will won_game is true, discard and win, this is because we made a while loop with all of them together, breaks need to be added so the player does not draw or discard after a player has already won, however, some threads slip past anyway, this doesn't often occur before thread counts of 250.

At thread counts over 750 a player log error occurs that will be elaborated later.

Bag

The superclass bag provides base functionality for the subclasses of white and black bags. They have attributes of ArrayList Atomic integer pebbles(used for ease of access and lack of flexibility associated with arrays), its bag id and its linked bag.

The Black bag has a draw function that is atomic, we thought of making that functionality inside the player but chose to do it within the bag classes to make it more centralised and clear.

The White bag has a simple method to update its list of pebbles and the action of a player discarding to another bag doesn't need to be “synchronized” as it only concerts one bag and one player. The action is atomic through its variables and existence is player.run(). White bag throws an error to ensure its pebbles are initiated properly.

Pebbles was initially an AtomicInteger however we realised we needed to send data on the bags that came with it, we used an Object[] to get the code to work and didn't change it later, however, pebbles should be its own class, containing an Atomic integer, and linked black and white bags, we would have done this but we would have to change much code and there were other parts of the program that didn't work well, and while casting may take more processing time, the program worked.

Exceptions

We have exceptions created for:
BagEmptyException: if bag interacted with is empty
BagFileFormatException: for any issue arising to the file not being compatible with the program
FileException: derived from reading a file, indicated the file doesn't exist
InsufficientPebblesException: Indicated the format of the file is correct but the amount of pebbles doesn't fulfil 11*player_num

Running the program

At 100 players the program is perfectly stable, takes about 10s* seconds to reach the win condition and file logs are of equal size.
At 250 players the program is also stable, while threads take 7.5s* This does little to disrupt the program. Player logs are also quite equal.
At 500 players the program is also stable, while threads take 5s* This does little to disrupt the program. Player logs are also quite equal.
At 750 players the program is also stable, while threads take 3s* This does little to disrupt the program. Player logs get quite wary at this point, with 1% of threads doing 10x the work, and 2% doing none at all.
At 1,000 threads this is made more exaggerated.
At 750+ an error also occurs where the threads don't stop and while a winner is declared. And has to manually be stopped.
The program uses 100% of the CPU indicating there are little resources wasted as cores do not switch, all 24 cores of CPU at 100% on my tests.
player6 has drawn a 1 from bag x
player6 hand is 51, 55, 46, 56, 13, 17, 24, 1, 25, 1
player6 has discarded a 55 to bag a
player6 hand is 51, 46, 56, 13, 17, 1, 25, 1
Sometimes a player will randomly discard 2 (55,24) pebbles, as to why we do not know, happens more often as the time the program runs increases.
Threads sometimes throw indexes out of bounds with random when drawing a bag.(rarely)

Design of testing (4)

The Junit 5.7.0 framework is used to test every aspect of the production code.
There are three testing classes created to test the functionality of the code to see if it matches all the requirements in the project specification. Mock objects are used throughout all the unit tests as this significantly reduces testing and debugging time, negating the need for prompting user input, and computer resources.

The testing for the white bag class is used to test if the bag can store pebbles properly with the correct data type when the game assigns pebbles to a white bag. Also, testing to see if the correct exception is thrown.
We chose to create a testAddBag() method to test the only method addBag() in the WhiteBag class. This test set up a black and white bag as mock objects beforehand. This is better than testing by running the entire game, as the test goal is to only test the white bag storage feature. This test uses the addBag() method to add pebbles to the mock white bag object. An assertion is then used to see if the correct amount of pebbles are stored properly, and an assertion is to check for the correct data type, with an expectation of both to be true. This is because addBag() only deals with updating its list of pebbles every time a player discards a pebble to it. The second assertion makes sure the BagEmptyException when the list of pebbles is initialised properly. The last test case uses a try-catch statement to see if adding an integer -5 to the bag throws an error, which is expected to be error-free despite it should not exist in the pebble input file in the first place.
There is nothing to teardown to reset variables in the white bag testing environment as the test is completed by the end of the testAddBag().

The testing for the black bag class is used to check if its methods validate pebble file properly, refills pebbles from a correct white bag, and returns pebble properly to a player, with exception handling in place.

We created testings for all the methods in the BlackBag class. They are testInitialiseBag(), testReadBagFile(), testFillBag(), and testDrawBag(), which corresponds to initialiseBag(), readBagFile(), fillBag(), and drawBag() methods in the BlackBag class.
A black bag mock object is used throughout all the tests in this black bag unit test because most tests do not alter its content, which makes it suitable for it to be used for the last test case, testDrawBag(), which alters it by calling the drawBag() method.

testInitialiseBag() contains multiple assertions that use invalid pebble input files to simulate exceptions that could potentially occur. For example, the pebble input file might not be in a CSV file format; the pebble file has zeros in it; the file has negative numbers, or it does not have enough pebbles. The assertions on these simulated situations ensure the correct exception is thrown and handled properly during the running of the game. If the testing for the mentioned scenarios above is not tested thoroughly, any errors that are not displayed during the process would result in the game not working.

Similarly, testReadBagFile() is used to test if the method recognises invalid file location input from the user-side, before initialising bags. The assertion here uses a string containing random words to simulate invalid user input.
Both the testInitialiseBag() and testReadBagFile() are two of the most important pebble file testing that ensures the game is able to handle all kinds of user induced exceptions and respond in a way that the project specification intended to.

The testFillBag() is used to test if the fillbag() method in the black bag class performs pebbles transfer from its corresponding white bag properly. The assertion here compares whether the pebbles in the black bag are the same as the ones in the white bag after the method is performed.
This ensures the contents including the number of pebbles and data types are transferred properly in the middle of the game for it to run smoothly.

The testDrawBag() is used to test the drawBag() method in the black bag class. The black bag mock object is used by the pebble variable, temp, which simulates a pebble drawn by a player in the game. Multiple assertions are used to ensure a random pebble with a correct data type is returned by the method, and the appropriate exception is thrown when the black bag is empty.
The layout of the test makes sure the draw bag functionality works according to the requirement by testing it to return a valid random pebble, and throw an exception whenever the bag is empty.

The testing for the pebble game class is used to test all the methods that are inside the player nested class and the in-game functionality. Six black and white bag mock objects are created to satisfy the need of all the test cases. The tests testWritePlayerFile(), testDraw(), testDiscard(), and testWin() are created to focus on testing the corresponding writePlayerFile(), draw(), discard(), and win() methods in the player class.

The test discardBagCorrectly() does not deal with testing any methods in the player class. It is rather used to check if a pebble is discarded to a white bag that is corresponding to the last chosen black bag during the running of the game. Instead of running the actual game, a mock player object is made to test this feature, as this way reduces testing time from user input, avoids any potential related errors, and saves computer resources. Just like the game, the player mock object receives pebbles from the draw method, a variable ‘previous_Bag’ would save the corresponding white bag of that black bag, and then it calls the discard method to discard a pebble to a white bag according to the variable ‘previous_Bag’. An assertion is then used to compare the original pebble, with the new pebble that is subsequently added to the desired white bag. It is expected to be true if their values are the same, that would indicate the player discarded properly.

testWritePlayerFile() tests if any actions from a player are recorded properly into its own player log. We decided to create a player mock object, with pebbles added to the player's hand. This can help to simulate an accurate scenario when the writePlayerFile() is used in the game but without the hassle of setting up and running the entire game. Then, the player is then made to call the writePlayerFile() filled with mock parameters. A variable ‘data’ is used to store the string values retrieved from the player log by using a File and a Scanner object. Variables ‘expected_content’ and ‘expected_content_2’ have the expected string values inside the player log after calling the writePlayerFile() method. In the end, these variables are used to compare the first and second lines of the log to see if they match the assertions.
testDraw() tests the draw() method whether it returns a pebble from a black bag properly to the player's hand. It is done by comparing the player’s hand with the list of pebbles in the black, after calling the draw() method. A player mock object is created for this test case. Variable ‘p’ stores the amount of pebbles in the player's hand before draw operation; variable ‘x’ stores the amount of pebbles in the black bag before draw operation. Two assertions are used to check if the amount of pebble in the black bag increased by one and if the amount of pebble in the black bag decreased by one. Another assertion is used to check if the correct assertion is thrown correctly when the black bag is empty. The design for this test case creates a sufficient simulation to ensure that the draw method works on both the player and bag object during the running of the game with appropriate exception handling. However, the random aspect according to the specification is not tested here as that is not the focus of this test case, but rather purely on the drawing aspect of the method.

The test case for discardTest() is almost identical to the testDraw() test case, with the only difference that the assertion checks if the player’s hand decreases by one, and the list of pebbles in a white bag increases by one, after calling the discard method.

winTest() tests if the win() method returns a boolean correctly, it should return true if the player’s hand is a sum of 100, otherwise, return false. In order to run the entire game, we decided to set up a simulated environment. To do so, pebbles with a total weight of 100 are initially added to a player mock object’s hand. Then, an assertion is used to check if it returns true after the mock player called the win() method, which we expected the result to be true. To test the other condition for win() also works, another pebble is added to the mock player’s hand so that the total weight is not equal to 100. The player’s win condition is again checked through assertion, and the result is expected to be false. The design of winTest() simulated and validated all the expected output of win(), which ensures both the boolean

values are returned correctly when checking the win condition of the player during the actual in-game scenario.

Test coverage

We have used the integrated Test coverage tool in the Intellij IDE to see how well our test is written. The result shows 100% coverage across most of the .java files except PebbleGame.java. This is because only all the methods in the player nested class (except the unused getters and setters) and the game simulation are tested, whilst the test coverage tool does not recognise the simulated portion in the PebbleGameTest.

Overall, the result indicates our test cases cover the majority of the production code. This means the pebble game program is robust and reliable in practice.