The Cover Page

Split of the Marks

The final marks should be split 50:50 between 700018370 and 700050709.

Development Log

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Date | Time | Duration | Driver | Observer |
| Tue 19/10 | 11:00 | 2 hours | 700018370 | 700050709 |
| Wed 20/10 | 14:00 | 1 hour 30 mins | 700018370 | 700050709 |
| Thu 21/10 | 14:00 | 1 hour | 700018370 | 700050709 |
|  |  |  |  |  |
|  |  |  |  |  |

Design Choices of Production Code

Firstly, we decided to represent both black bags and white bags in our game using ArrayLists. Since an integer could easily represent a pebble, the bags only needed to contain a list of integers. And as the list would need to be resizable, ArrayLists were considered the most straightforward way of modelling a bag. Additionally, since black bags must be paired with corresponding white bags, we created separate ArrayLists of both black bags and white bags themselves. This allowed for a white bag and a black bag to be linked by the same index.

To convert the CSV files into black bags, which were ArrayLists in our case, a method was used first to convert the contents of a CSV file into strings and have the commas removed. The strings were then converted into integers, which were added into an ArrayList. There was a requirement for a minimum of 11 pebbles per black bag; the method counted the number of integers in the CSV file and threw an exception if that number was not met. Since the number of players gets requested at the start of each game, we stored this value within the main class for easy access. This was then used to tell the method how many times to copy the contents of the CSV file into a black bag, as this, combined with the exception, ensured that there were at least 11 pebbles per player in each black bag.

To represent the player's collection of pebbles, an ArrayList was made an attribute for the nested player-class. To select 10 pebbles for each player, the Random class was used to select one of the three black bags pseudo-randomly. The class was then used to randomly select integers up to the size of that black bag, with those integers being used as indices to determine which pebbles were to be removed from the black bag and added to the player's pebbles.

Since the game's objective was to have a collection of pebbles with a total weight of 100, an integer representing the total weight was attributed to each player. Each time a pebble gets added to a player's collection, we made sure also to update the total weight so that the figure was always up to date.

To allow the game to be playable as a multiplayer, an integer named "turn" was made an attribute of the main class to store the current player's turn. We created an array of threads so that there was one for each player. The benefit of multithreading here was to improve response time so that the user interface can update and respond to each player whilst the background task is running. A new thread was run for each turn, and the threads were made to start and join so that they could work concurrently. After all, players have had their turn, the method calls upon itself at the end to continue playing, and players can have subsequent turns.

So that a game would finish immediately if a player won, a Boolean value named "finished" was attributed to the main class. This Boolean would act as a series of barriers, such that if the game were finished, the following code would no longer continue running, and the game would effectively end. Within each thread created for each player's turn, a method was run that would form the basis of how the game would go on to play. For the game to end immediately, should a player be fortunate enough to have winning pebbles on the initial draw, the first part of the method was designed to display the winning message and update the Boolean to true.

However, in the more likely scenario that no player has immediately won, the method goes on to display which player's turn it is, along with their collection of pebbles and their total weight. This information is provided to aid the player in choosing which pebble would be best to discard, for we decided to offer the player that choice. Since all other steps in the simulated game had to occur randomly, it was deemed necessary to allow player involvement in this part of the game. When the player enters the pebble's weight they wish to remove, it calls upon a method that removes the first instance of that integer in the player's ArrayList of pebbles and deducts it from the player's total weight.

The method then adds a random pebble from a randomly selected black bag, similar to how the initial draw of 10 pebbles worked. But in this case, the black bag is first checked to see whether its ArrayList is empty. If it is, the contents of the corresponding white bags are added to the black bag, and then the contents of the white bag are cleared, all to enact the white bag getting emptied into the black bag. The randomly chosen index of the black bag from which a pebble is drawn is stored as a player attribute so that it can be used to discard a pebble into the correct white bag before the index gets updated again. Finally, the method checks again at the end of the player's turn whether they have won so that if they have, the Boolean "finished" can be updated accordingly, and the game can be easily ended.

To allow the program to be exited at any stage if "E" was received at any input, "finished" would be set to true, just like if a player had won. The program was structured so that if "finished" were set to true at any point, the condition to allow any subsequent code to be run would not be met, and thus the program would end.

To make sure that drawing and discarding were an atomic action, we made use of AtomicIntegers. Since thread contention was low in our game, it was the preferred way of achieving an atomic action ahead of the synchronized mechanism, which would have come at the cost of worse performance.

Finally, to ensure that the game has a strictly positive number of players when asked for the number playing, any input that is not an integer greater than zero will return the same question repeatedly until a valid answer is provided. For pebble weights to be strictly positive integers, the method that reads the CSV files was also made to check that each integer is not zero or less. Otherwise, an exception is thrown to notify the user.

Design Choices of Testing

JUnit 4.13.2 framework was used for testing. The primary purpose of the testing was to ensure that the simulated game met with the properties stated within the controlled assessment specification and to make sure that to prevent unhandled exceptions during user inputs. Each of the tests was broken down by class and by methods in each class. Breaking them down was done to perform unit tests on each of the methods within the classes without any interference.

The tests were designed to follow the ATRIP properties to ensure they were good tests. They are automatic since they do not require user input or files and automatically check that each criterion has been met. This was done with the use of the methods writer, reader and test input. The writer method would create temporary files and write to them, and the reader would read from them for validation. The method testInput would take a string and then convert this into bytes. It would then convert these bytes into a byte input stream, replacing the standard input stream before setting the scanner in the pebble game to the correct input stream. The tests were through since I assured that they tested for the required properties and that each input would catch any thrown errors. The tests met the repeatable property because I made sure they used the same input each time and did not set any checks on values that could change between tests (e.g. their value was created randomly). If there were times that I had to check a value that could change, I would check a range of values. The tests were independent of each other using the two setup methods that would reset all values before a test so that the result of another test wouldn't influence the result of another. Finally, I made sure the testing was professional but included plenty of comments to make it easier to understand, diagnose errors, and maintain both the test file and production code. Also, this was a reason for splitting up the tests for each method and using a method that would set up for each test to make the code more readable.

The file PlayerTest.java was used for the testing of the nested class player. The two methods being tested were to remove a pebble from the player's pebbles and add a new pebble to the player's pebbles. When adding a pebble, I tested a typical case and an invalid case, a negative value. This was done to make sure that it was functioning correctly and would catch any errors. Similarly, by removing a pebble, I tested a standard case and two incorrect cases, removing an item that isn't in the list and removing it from an empty list. I chose to do those since it again tested that pebbles could be removed but caught errors. Despite being very similar, the two incorrect cases were chosen to validate that the total weight variable would be correct and wouldn't erroneously change.

The file PebbleGameTest.java was used for integration testing and to perform unit tests on methods within PebbleGame.java. One of the test methods was on writing to a text file when the player drew a pebble. Two tests were conducted within this method using valid values. They were done to ensure that the correct file was being created and write the correct values to the file. The first test used the X black bag, and the second used the Z black bag to act as a boundary case for each. Also, the second test was done to check that the file correctly appended the text to the file, rather than overwriting it. The following test method was on writing to a text file when the player discards a pebble. Two tests were also carried out here using valid values. In a similar fashion to the previous method, the first test was carried out using the A white bag, and the second used the C white bag such that both boundary cases were tested. Again, the second test was also designed to check that the file appended the text from the first test instead of overwriting it.

Since the program was designed to read the CSV file into a black bag the same number of times as the number of players in the game, another test method was created to ensure that multiple players would result in sufficient pebbles in a black bag. A CSV file with 11 pebbles was used to test three different games, varying in player counts of 1, 4 and 100. After reading the CSV files, the games were checked to see if a dummy black bag would end up with at least 11, 44 and 1100 pebbles, respectively. And as the CSV files could not contain negative numbers or contain fewer than 11 integers, CSV files were created which separately failed those criteria and threw the right exceptions when read within this method.

In order to test the method which draws a pebble from a black bag in the game, another test was designed to apply this method and assert that the player's pebble count increased by one and that the black bag was drawn from decreased its count by one. If both events take place, it can be surmised that the draw worked correctly. Additionally, this method was also created to check that if a black bag were empty, a non-empty white bag would empty its contents into a black bag, and the draw would take place as expected. This was tested by asserting that the white bag was empty after the draw, and the black bag ended with one fewer pebble than the white bag started with (to account for the pebble given to the player). Like this test, another testing method checked that the initial draw took place correctly by asserting that the player's pebble count increased by 10 and that the black bag decreased by the same amount after this draw.

One of our main testing methods was created to test all of the functionality within the playerthread method, which contained much of how the game was to be played for the duration of each player's turn. Conducting these tests would be like testing the game if there was only one player and a single thread. This test started by checking that "E" ends the game by asserting that the Boolean "finished" is triggered by that input. Similarly, the test was made to check that a total pebble weight of 100 from the initial draw ends the game immediately. Aside from setting "finished" to true, it was tested that the player's total weight remained at 100 and their pebble count remained the same, to indicate that the rest of the playerThread method was not triggered and that no further draw or discard took place, since the game had finished. Moreover, it was tested that discarding and drawing a pebble could also be a means to a win.Unlike the other tests that were checking win or end conditions, this test ensured that the playerThread was functioning as expected by inputting a 2 (an arbitrary value that would always be in the players pebbles). Then several checks could be made that the 2 had been removed and a random pebble from the bags had been inserted as expected. The last 2 tests were both done similarly to check that pebbles that were not inside the player's pebbles and that values that weren't integers should be thrown and caught.