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, and the method calls upon itself at the end so that the game can 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 weight of the pebble they wish to remove, it calls upon a method that removes the first instance of that integer in 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 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 main purpose of the testing was to ensure that the simulated game meet 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 were broken down by class and by methods in each class. This was done to be able to perform unit tests on each of the methods within the classes.

The file PlayerTest.java was used for the testing of the nested class, player.