**Cover Page:**

Final mark allocation: 50:50

**Development Log:**

Key:

Developer 1 (Dev 1) – 160221 Developer 2 (Dev 2) – 193960 Observer – O Driver – D

  Date

Time

Duration

Role(s)

Brief description of what was accomplished.

Candidate Number Signatures

  Monday October 18th

2021

3:30-4:30pm

1 hour

D = Dev 1

O= Dev 2

Discussing design/direction, setting up initial classes and files.

160221

193960

  Tuesday October 19th 2021

4:30-6:30pm

2 hours

First half:

D = Dev 2

O = Dev 1

Second half

D = Dev 1

O = Dev 2

Created file reading methods and some object methods.

160221

193960

  Friday October 22nd 2021

6:00-7:30pm

1 hour and 30 mins

D = Dev 2

O = Dev 1

Creating more methods and setting up atomic actions

160221

193960

  Monday October 25th 2021

5:00-6:30pm

1 hour and 30 mins

D = Dev 1

O = Dev 2

Bug fixing and reducing overhead. Began multithreading.

160221

193960

  Monday November 1st 2021

6:00-8:00pm

2 hours

D = Dev 1

O = Dev 2

Some streamlining/restructuring. of existing code.

160221

193960

  Tuesday November 2nd 2021

4:30-6:30pm

2 hours

First half:

D = Dev 1

O = Dev 2

Second half:

D = Dev 2

O = Dev 1

Added file writing, bug fixing, checked coded against the specification.

160221

193960

  Friday November 5th 2021

9:00-10:00pm

1 hour

D = Dev 1

O = Dev 2

Bug fixing, and discussion on remaining tasks to tackle, and code commenting.

160221

193960

  Sunday November 7th 2021

5:30-7:30pm

2 hours

D = Dev 2

O = Dev 1

Running, testing, and fixing remaining bugs and code commenting.

160221

193960

  Tuesday November 9th 2021

4:30-6:30pm

2 hours

First half:

D = Dev 2

O = Dev 1

Second half:

D = Dev 1

O = Dev 2

Finish report, final bug fixes. Last review of the project to make any last-minute changes. Submission.

160221

193960

**Design Choices:**

**Handling an invalid total number of pebbles inputted:**

We initially considered handling ensuring that bags contained enough pebbles for the game in the code. The idea behind this is that when users are entering files, they are just entering a distribution of weights to create the probability they would like to have for their game. Since pebbles are drawn at random, the only way to increase the likelihood of a certain weight is to increase the number of pebbles of that weight in a bag.

Hence, we thought about implementing this by taking this distribution and duplicating it until the total pebbles were greater than 11 to make it valid for one player, and then to replicate this new distribution for as many players as the user entered to ensure the game would be playable for whatever number of players the user entered as well regardless of the original amount of pebbles taken from the flies. This would all be accomplished while maintaining the probability distribution that the user entered as the number of pebbles is corrected through duplication, so the frequency of a certain pebble weight would never change.

We eventually decided not to handle this for the user as it was pretty similar to the idea of handling for the user entering files that contained a distribution of pebble weights that would make it so that a hand could never hold a weight of 100 where both would situations would require the code to handle user inputs that were correct syntactically but still caused issues for a perfect running of the game. Therefore for continuity’s sake, we chose to return to the user that they did not have enough pebbles in one of their entered files.

**Handling distributions of pebbles where 100 is impossible to make:**

We chose not to handle distributions of pebble weights that make a hand reaching 100 impossible and just let the code run. This is because it will often be virtually impossible to account for all the permutations of the total weight in a hand of the total distribution of pebble weights across all three bags. So while it would be feasible to create methods to catch certain constraints (such as checking whether there were pebbles in the bags that make it impossible to get 100 with them in your hand), we found that for it to make enough of a difference that we wanted to implement it, the method would have to brute-force its way through permutations which became completely infeasible as the number of player/number of pebbles increases or to use approximations which could still become infeasible and was also quite complicated to implement.

**Use of Singleton Pattern for PebbleGame game state:**

Initially, we were unable to access the instance of a PebbleGame from the static main method. This led us to create another instance of the PebbleGame within the already created instance of PebbleGame in the method runPlayers; we then had to create instances of players within an instance of a player array within our instance of PebbleGame. This was very inefficient and used more overhead. By implementing PebbleGame as a Singleton Pattern, we only needed to create one instance of PebbleGame to solve the problem and run the program.

**The design choice of atomic Draw and Discard method:**

Initially, we had one method for drawing and discarding. The idea was to make the bags atomic memory spaces so that concurrency could occur within a single method while keeping it thread-safe. However, we decided that as the specification required of us to create a game for only a specified number of three bags that did not change, we decided it would be simpler to code individual synchronized methods that still preserve the atomic action of drawing and discarding for each of the bags to create a more straightforward solution than what would have been required otherwise using our original method. In doing this, we tried to comply more closely with what was required of us rather than making something more complex than what the user (specification) asked for.

**Placement of the run() method for threads:**

Initially, we wanted run() to exist in the PebbleGame class; however, we found it confusing and not developer-friendly as the threads tend to act as players. We wanted to model this view of the game from a “player’s” perspective. It also required a more complex solution to pass players into threads, so creating a specialized PlayerThread class to help run the game made it far easier to understand code.

**Handling of starvation between player threads:**

Initially, we faced a minor starvation issue which was rectified by creating a separate loop to start threads from that used to create them. That decreased the overhead between starting threads which reduced starvation. The limitations of this solution came when the program had a very high number of players or a pebble weight distribution which has a very high probability of creating 100 or from both, which could cause players to win and end the game before another player even got to draw their hands let alone perform their first draw and discard. This led to the creation of a few unique situations that still led to starvation. We considered a more complex method to reduce starvation:

This involved using a counter in the while loop to count how many times a player has drawn and discarded; we then would have a limit to this counter with an if statement where when the player has drawn/discarded “its fair share of pebbles,” it would call a PebbleGame method. This method would utilize a priority queue of threads where pebbles are once they have gone “too many times” are moved to the back of. It would also use the join()/yield() methods to make a thread wait to go until after the current last player in the priority queue that it will now queue behind has been sent to the back of the queue. This would stop any thread from getting too much an unequal amount of draw/discards, reducing starvation. A limitation would be that with this arbitrary counter, what could be considered fair would change depending on the number of players. We also found a solution for this by creating the limit for the counter using an algorithm calculated off the number of players input to play the game.

However, we still decided not to implement this solution to starvation. The reason is that although it would somewhat prevent players from winning before others had drawn their hands. It was still possible for with a vast number of players/extremely high probability of making 100 pebble distributions that a player would win with their first hand drawn, without drawing/discarding a single pebble and hence the same limitation may only have been reduced slightly, and very much is still a problem. Therefore we decided that all the extra overhead required and the additional complexity added to the code would, in most cases, still outweigh the benefits of this more controlled method of avoiding starvation. Since we both deduced it was most likely that/that all the cases we observed of starvation occurred in the extreme cases that are still a limitation of this more complex solution. Hence, we chose not to implement it after having a few discussions on how we would deal with starvation and how much it would be affecting the game.

**Multiple winners:**

We found that multiple players could win at once when performing unit tests on the run() method. In finding this, we realized the existence of a limitation to the boolean lock method we used for ending the game for all players once a player won. We found a small timeslice wherein threads that did not win may still be running through their run methods after a winning player has triggered the boolean lock. As a result, more players could succeed in this time slice with a distribution of pebbles that creates a very high probability of winning. We considered reducing the likelihood of this occurring by adding in more if statements to if the boolean lock had been triggered or not yet throughout the run() method. However, we eventually decided against it as we decided that the likelihood of multiple winners occurring was already extremely low to the point of needing distributions that almost guaranteed wins for it to happen frequently. It was not worth adding in the extra overhead for the systems by introducing more overhead in many additional comparison operations.

**Test-Based Design Choices**

We have decided to use JUnit version: 4.13.1 Library as there is a lot of support we could find on this particular version.The code coverage is maximised to the largest amount to ensure code integrity and correctness of our pebble Game program;This leaves only a few methods that are not tested,they are as follows: start\_game() in PebbleGame and run() in the PlayerThread class.The reason why we left out those particular methods from our unit testing is because they don't return anything or have any parameters which would make it very hard to test in JUnit wich any of the assertions that JUnit provides.For example start\_game(); only accepts inputs via console input so we would of had to somehow integrate JUnit to enter values into console line and then extract the outcomes which is a complex thing to do.

We have quickly realised that testing the methods in a specific order is the most effective way of unit testing the whole program.This is due to certain methods relying on dependencies on other methods so if we know a dependency works correctly and is error free then we can use the dependency like its a known to be working method in testing the method on the outer layer.A classic example of this would be when a method calls another method to fetch and process something from that method to use it then within itself. Therefore,while testing we had to keep this in mind when picking appropriate test data and also the order in which we test the methods.

Each test method features @Test which allows JUnit4 to execute each test method and unit test the component of the pebble Game.We also decided to also use @Before method which is run before each test method is run and inside that method we keep tack of the amount of tests executed so the used can then see how many tests have been run and see out the tests run how many of the tests have been successful/failed.@After a method that executes after each test method that keeps track of the just executed method and checks for things like failed asserts and if there's a failed assert it will add that to a counter that will help the user,at the end,check the errors and their quantity.

Each test method contains test data, assertion and conditions that has to be met which verifies the data has been manipulated ocrrectly.To ensure we have maintained a somewhat consistent way of unit testing,In each method we applied the BICEP strategies into our unit testing to help ensure :good code methodology, testing and structure.This will minimise the possibility of missing potential code bugs while we were making the code and unit tests for it.We also had to make sure certain methods return the right exceptions with specific data input so we had to use “AssertExeption” to help us check that.Another way we tried to follow a systematic approach is that we tried to make the tests be setup in such a way so it's easy to change the data and see how each method bencaves when incorrect data is passed/we create errors in the methods.This was done by having them assigned first thing at the top of each test method so its developer friendly and easy to interpret the input data.

The player class mostly consists for getters and setters so We decided it would be a good idea to implement JUnit assert that compares input and outputs of getters and by doing so it confirms the setters also work since getters return the value that has just been set.for a getter and setter methods involving purely integers we have used the JUint’s ”assertEquals” which helps to verify expected values and values that are being output by the method being analysed.

At the end of the testing we decided to add one more method test: @AfterClass which executes after all tests have been run.This method looks at the static variable such as a array list contain all the failed test information,the number of tests done and the number of tests passed/failed which is then displayed inside a console that helps the used debug the unit testing when some @Test methods fail certain test requirements.