Cover Page

50:50 split between
Jeroen Mijer, student number = 700046371, candidate number is 014777

Alexander Robertson, student number = 700034546, candidate number is 185742

Development Log

Driver	Observer	Duration	Date & Time	Signatures
Jeroen Mijer	Alexander Robertson	2 hr	04.11	014777 185742
Alexander Robertson	Jeroen Mijer	2 hr	05.11.2021	014777 185742
Jeroen Mijer	Alexander Robertson	5 hr	06.11.2021	014777 185742
Alexander Robertson	Jeroen Mijer	7 hr	07.11.2021	014777 185742
Jeroen Mijer	Alexander Robertson	8 hr	08.11.2021	014777 185742
Alexander Robertson	Jeroen Mijer	4 hr	09.11.2021	014777 185742
Jeroen Mijer	Alexander Robertson	10 hr	10.11.2021	014777 185742

We created three classes. The first is the PebbleGame class, which the game runs in, similar to what a typical Main class would be. The second is the Player class, which implements the threads and is a nested class inside the PebbleGame class, as is specified in the CA. The third is the InputValidation class. This was put in to validate the inputs that the user put in and get the system up and running.

The PebbleGame class was responsible for providing the framework that the Player class ran in. This is because most actions were taken by the players themselves who were “playing” the PebbleGame. Therefore the PebbleGame class consisted of three arraylists. 1 containing the bags, 1 containing the players and 1 containing the threads. This allowed for the Player class to be able to access all the information needed such as the contents of the bags and the threads should they need to access information about the threads. By keeping this information in the PebbleGame class all of the players and the threads that are running them have access to the same information and as a result will mean that the players are all involved in a fair game, not a game where different players drew the same pebble from the same bag. The PebbleGame class is also responsible for dealing with user inputs, this includes the initial user inputs of the number of players, the files and if the user inputs ‘E’ at any point. It validates the inputs using the InputValidation class, however having the values of the user inputs in PebbleGame makes the access to the user input information much simpler to implement and reduces unwanted bugs.

The InputValidation class consists of methods that validate user input, the methods return a boolean and if it is false then the PebbleGame class will ask the user to put in valid input, this continues until there is a valid input or E is input. There are also two methods that are responsible for file creation and clearing the files from the previous game. Finally there is one method that is called whenever reading a user input to check if it was ‘E’ in which case it shuts down the system. These last three methods are in this class in order to decrease the amount of code reused, since these methods are called multiple times. It also helps to organise and encapsulate the code which helps with debugging and creating desired functionality.

The Player class is the largest of the two in terms of functionality. Each player implements a thread per the CA specification. The PebbleGame class creates as many instances of a Player as the user desires. This is useful because it allows for a simple and safe implementation of storing, accessing and editing the information of each Player. The Player does the initial draw after which the thread of the player starts. This means that players may start drawing and discarding before all players have drawn all their initial pebbles. However changing this makes debugging as well as the implementation significantly harder. We made the decision not to

change because there are more important issues to focus on and this is because even with ‘generous’ pebbles that are designed to make winning easier, containing a relatively large number of pebbles having a value of 10 or lots of pebbles that add up to 10 such as (5+5 or 4+6) it usually takes players at least a couple thousand draws to win the game, with the example files we were given usually in the tens of thousands of draws. It may also take a long time to win the game because of our implementation where the pebble is randomly selected from the bag when drawn but when a pebble is discarded it is always the first pebble. This was done because implementation and debugging is incredibly easy as a result and it shouldn’t matter too much since the pebble draw is always uniformly random, but it could be a potential reason.

The threads then continue to run the draw and discard methods, to simulate the drawing and discarding of pebbles until a player has either won, at which point the game stops or if the user interrupts the program by putting in E.

The draw and discard methods are synchronized in order to make it more thread safe, because otherwise multiple threads may try to access the same bag at the same time. Each bag was implemented as an ArrayList of Integers. This is because it is a lot more efficient to use an ArrayList of Integers stored in the PebbleGame application than having a file for each bag and constantly reading and writing to said file. Since the user does not care about the pebbles in each bag, since their goal is to win the game and as a result only need to know whether to draw or discard and the pebbles in their hand and their sum, the values of the bags do not need to be stored after the game ends. In terms of debugging, using a debugger such as the one available on VScode is very effective in being able to view the values of each bag (ArrayList<Integer>) during runtime. It also returns an int in order to allow the next discard to go to the correct bag. The correct int is always passed since the method that makes a thread draw and discard a pebble is synchronized meaning that every time a pebble draws every time a thread executes run it immediately executes discard, meaning that the value is not changed and the int is thread safe.

We also made sure to check weight in the draw method to ensure that no winning hand was discarded. It also ensures that once a Player has a winning hand no other Player can then draw a winning hand and cause a dispute over which player won.

By keeping the two methods draw and discard thread safe we made the entire application thread safe, since these are the only two methods that try to access information that is available to all threads. The issues can occur when 2 Players or Threads implementing them try to change a bag at the same time, which can lead to different.

Testing

For this project we planned to use the JUnit 4 framework. After hours upon hours of research and attempting, we could not integrate JUnit with our code, as when compiling the JUnit file, nonsensical errors were repeatedly thrown up in both of our separate development environments. These errors were related to the importing of packages required for JUnit tests.

We were extremely time limited, and we deeply regret not having had the opportunity to further try and use JUnit testing.

At the beginning of the project, we did in fact plan out the structure of our code in a collaborative session, where we decided what we thought at the time would be all of the required methods, and the structure of the code which we described in the section above. In that session we also considered how to write tests before implementation, however problems with JUnit unfortunately scuppered those plans.

We wrote tests concurrently to test methods as they were written, but most often used the Debugger; Adding variables to the watchlist, in addition to setting breakpoints, was found to be much more beneficial than the tests we wrote. This is because we were able to see a live snapshot of the environment's state, and the contents of separate threads in runtime, so bugs were easier to solve. Moreover the consistent use of printStackTrace enabled us to see the location of bugs very simply, and from there we were able to use the debugger to methodically sweep through the code and discover the exact points at which bugs were being generated. This led to simple fixing of bugs.