CS1632 – Deliverable 2: Unit Testing Ruby Gold Rush

Project: Ruby Gold Rush simulator

(repository link: https://github.com/shs205/D2)

Daesang Yoon (yds725) and Shay Sinha (shs205)

**Description**:

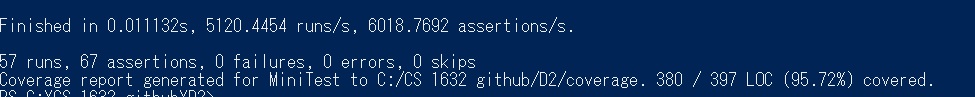
At first, we thought that implementing Ruby Gold Rush simulator was not difficult. Indeed, if we only care about its functionality and printed output results, it is simple to build our simulator. However, the problems occured when we were building our unit test file. We initially implemented our simulator inside only one file gold\_rush.rb. We had a class object constructor, all methods, and main execution function in one single file. This way of implementation turned out to be inefficient.

First, running test file would require running whole simulator program. Every time we run a test file, we also have to run a simulator simultaneously. This is very inconvenient not only for testers, but also for clients in case they want to demonstrate certain software product. However, this problem is easily solved by just changing gold\_rush.rb into main driver program file which includes only main execution function. Then, we create another class file which contains object constructor and all methods for simulator functionality.

Next issue we faced was one of our methods (‘iteration’ method) in game class file was too large. This ‘iteration’ method calls many other methods inside itself and returns no meaningful value. Instead, it prints many outputs as the requirements for the simulator programs describe. The ‘iteration’ method is a big main core function for our simulator. Even though the method runs as intended and prints the result as expected, it is almost impossible for us to test the method. There are some reasons for this problem. You really cannot test a method that does only ‘puts …’ in the end of function. There were no appropriate and easy-to-write Minitest assertions for ‘puts’ or ‘print’.

Another reason is the method itself is very ‘impure’ and an ‘impure’ method is not easy to test. The result of the method depends on other external factors such as calling multiple functions, many loops and printing out into the console. Solution to this problem would be changing ‘impure’ method into ‘pure’ method. We have to improve our implementation by splitting one large method into smaller several methods. Also, we can make methods to be dependent on only inputs or arguments, and make sure methods actually return meaningful value so that we can test with assertions.

Those above were main issues we faced. Other trivial issues such as rubocop formatting & style check were small problems like empty or trailing white spaces in the code. The very important lesson we learned from this deliverable is that ‘red’ phase of TDD is significant for software testing and developing. Especially, writing a test before adding new functionality is very important fact that we should not neglect. Writing whole project before doing any test may provide inefficient implantation and overheads, and you may find yourself in trouble where you may have to write everything up again. It is good for testers or developers to take note of this software testing methodology.



At the end, all unit tests are passed and over 90% statement was covered.