Island Escape

Casey Nispel

CS 1699 - FINAL DELIVERABLE

For the final deliverable, I decided to write a new application from scratch using test driven development. Test driven development was something during the semester that I understood in theory but had never actually had to put into practice. In the past I have always written code first and tests second and so I was curious how test driven development worked in practice and how it would change how I write and think about the code that I write.

The application called Island Escape that I created uses a basic text interface to interact with the user, similar to the Coffee Maker Quest. Here, the user appears on a deserted island, and the goal of the game is to craft a boat and escape. There are six locations arranged in a circle and each location has items that the user can pick up and store in an inventory. At each iteration of the game, the user has the option to go forward, backward, look around, craft a new item, view inventory, get help, or exit. The user must craft a torch, then fire, and finally a boat in order to win the game.

I performed functional testing on this application using JUnit and Mockito. I began by writing a list of requirements I needed the application to fulfill. For each requirement that I had created, I wrote a test for that requirement. I then wrote enough code to make the test for that requirement to pass. Once that test and all previous tests were passing, I integrated that code into the rest of the code I had written so that the program would work as expected from the user’s point of view. Once I had written all of the tests, I also ran it through EclEmma to get the code coverage of my tests.

The first problem I encountered was having to ignore my instincts to write code first and test later. It felt unnatural at first to test something that did not yet exist, but I found myself planning out the code more in my head as I decided what each test had to assert than I normally do. When I went to write the actual code then I had a more clear idea of what I needed that portion to do and the actual writing of code went much faster than if I had written it initially.

I also had to think about ways to determine whether code was running correctly or incorrectly. Instead of creating a lot of void methods then like I might usually do, I needed methods to return a value to determine if the method was a success or failure. Some other requirements were checking whether the program was printing out a certain line, so I had to capture the output from the program while I was testing it so that I could check that the output was correct. I also ran into an issue where a requirement was that the program would exit. Since I could not test that the program ended, I created a Boolean to keep track of whether or not the program should continue running that I could then test the status of without having the program actually exit.

Another problem I encountered was attempting to test a method that used a global variable. I ended up passing the variable into the methods where this occurred so that I could create a double of this variable within the method. Though I kept this as a global variable, it would make more sense to change this to just a local variable. This portion made me realize how difficult it is to test programs with global variables and why local variables are preferred. The global variable has too many opportunities to be altered by another method and this is very hard to catch even with testing and pinpoint where the problem is occurring.

While writing the tests I had to determine how I wanted to store information and access it, since it would change how I would have to test certain requirements. This was especially true for the inventory and the items that were found in each location. I considered storing both either locally in a basic array, or creating new classes for the inventory and items and creating new objects to test. In the end this issue came down to how easy each structure would be to test, versus how easy it would be to write into the actual application. I ended up storing these in arrays which, though slightly more complicated to test, ended up being faster to implement and use within the program.

I found that in the end, the code that I wrote ended up being more modular than normal. I had many smaller methods that I knew all worked, and I could just combine them to form a functional application that I then also knew would run error-free. Though writing tests first and code later took longer, I felt I had a deeper understanding of my own code and where issues were bound to occur. If I were to write more tests, I would cover more edge cases, especially in strange input the user could enter. In order to cover these and also get more code coverage, I would test more branches of code, such as attempting to craft a fire both with and without the right objects. This would help find strange errors that do not appear with regular use and normal testing of functional requirements. I would also want to make the global variables all local and test them to make sure they are not being changed anywhere that they are not supposed to. If updating the code in the future, I would also like to make a more robust and object oriented way of storing items and inventory as opposed to just arrays, which would change the current tests but make the items much easier to access logically.

The application Island Escape has seventeen tests at the moment. All seventeen tests pass with no errors and no failures. The area where a majority of problems occurred was within the parseInput method, however since this is a large method that deals with every input the user enters, this is to be expected. One issue to be on the lookout for is the global variable location. Though not causing any errors at the moment, changes to the application in the future could cause issues, especially with testing any methods that use this variable.

The testing suite covers 92.7% of the code at time being. This suite covers all requirements, and either partially or completely all methods. The biggest method bringing down the percentage is the main method, which is not being directly tested. Without the main method, the code has 95.3% coverage. Overall I feel this application is of high quality and is able to be used without crashing or facing unexpected behavior. Due to the passing tests and high code coverage, we can be confident that the application runs as expected and without bugs.

Every requirement has been implemented and runs successfully with no errors. From an end user’s point of view, the application is clear and self-explanatory, prompting the user while still maintaining a minimalist text-based display. The application is responsive and performs effectively. In light of the 100% of tests passing, 92.7% code coverage, and the effective and efficient user experience, I would recommend that this application is ready to be released.

Code Location: github.com/Cln42/1632-Final

Screenshots: 