CS 1632 - DELIVERABLE 6

Members: Christian Boni

Project: Writing a New App from Scratch using TDD

Summary

For this final deliverable I had the freedom of choosing the topic of the project. From the (non-exhaustive) list of possible project ideas in the deliverable description, I made the decision to go with the “Writing a new app from scratch using TDD” idea. I chose this as the topic for my project for several reasons.

The first of which is that Test Driven Development (TDD) is a relatively simple way to create software which is maintainable. This is because the TDD paradigm implicitly creates testable and extensible code. Another reason for choosing TDD is that development automatically creates a unit test suite for the software. During the development stage, many relevant unit tests are created. In turn, this creates a unit test suite which is immediately available for use if any existing code needs to be modified. Furthermore, since a higher number of *relevant* tests are correlated with a higher quality software, we can assure that our code has fewer defects and thus spend a less amount of time debugging.

Finally, the last reason I chose the topic for this project actually dealt with the “Writing a new app from scratch” aspect of the description. During lectures, Professor Laboon would occasionally bring up a variety of programming languages and their corresponding unit testing frameworks. After hearing about all this unknown technology for the first time, I became motivated to learn them. As a result, I chose to teach myself a new language (Ruby) and its corresponding unit test framework (RSpec) for the development and test of this project.

Development

The application that was developed for this deliverable is a simulation of a queue that keeps track of orders. This application could be used for a variety of purposes but the main focus of my implementation was geared toward the food industry. As a result this application will be simulating food orders for places such as a restaurant or fast-food establishment.

Since this program is simulating a queue of orders, it is obvious that I needed to use a queue as the primary data structure. Here I had two options, I could either use Ruby's predefined array with its own push, pop, shift, unshift methods or I could implement a queue on my own. I chose the latter as I wanted to really learn the language and gain an understanding on Ruby's methodology.

Next, I started on the architecture of the application. I came up with a structure that contained several classes. Nodes would be used to build the data structure of the Queue. Additionally, Orders would be the type of data stored in each Node of the Queue. Finally, OrderUp would encapsulate a Queue an act as the overall Order-Queue interface. However, I realized that I would need several driver programs to illustrate how the application would operate. As a result I added two drivers, OrderDriver and OrderUpDriver. Since I was applying the methods of TDD I started coding only the bare minimum scaffolding for these classes, such as defining instance variables and method signatures.

Test

After the bare minimum architecture was complete, I moved onto creating the unit test suite for the application. As I was writing each test case of the unit tests I made sure to follow the **Red-Green-Refactor Loop** methodology of TDD. Because this application has multiple pieces which all depend on each other, I started writing unit tests at the lowest level and worked my way up to the top. As a result, the order of the classes that I wrote unit tests for was: Order -> Node -> Queue -> OrderUp. This made sense to me since Nodes make up a Queue and the Queue is the primary data structure being accessed and modified in OrderUp. Additionally, Order was tested first as it is the form of data being stored in the Node. Lastly, I made the decision to not test the drivers as I felt this was irrelevant and unnecessary.

Issues

As Ruby is a “duck” typed language I had a hard time deciding whether to incorporate nil checks when testing and developing this application. Due to duck typing in Ruby a variable is not constrained to remain a specific type. As a result, not only would I have to add nil checks but I would also have to add type checks when setting various variables (mainly instance variables). After debating about what to do, I came to the conclusion on only adding nil checks on several instance variables. I felt that due to Ruby’s nature it would be considered wrong and irrelevant to add type checks and I made sure to only add nil checks when absolutely necessary. It is important to note that these concerns arose during the creation of the unit tests when thinking about base, edge and corner cases.

During the **Red-Green-Refactor Loop** of the TDD process it often was difficult to anticipate every base, edge, corner, and error case. A number of times after completing both the unit tests and the application’s code, I actually stumbled upon a test case that I had forgotten about. As a result, I would have to go back and add additional test cases for that particular functionality. During the TDD process something I had run into a quite a few times which was actually more of a benefit than a drawback, was the test suite catching bugs and defects in code for the application.

Concerns

Going forward with methods of testing this application I strongly believe that the addition of Property-Based testing would have a significant impact on the quality of this application. Several deliverables ago, I chose to perform property based tests on Java’s Arrays.sort() method. During this project I discovered many of the benefits of applying the Property-Based testing paradigm. Property-Based testing moves to higher level of abstraction and allowing the test cases to test the expected properties of the behavior against the observed properties of the behavior. Which allows the computer to do the tedious work of generating all the various test cases. As I found in that deliverable, Property-Based testing was a very effective way of testing the Arrays.sort() function. The Queue is also a data structure, which operates much like an Array. For that reason, there would be an endless amount of test cases to test the Queue. In the future, I could apply my knowledge of Property-Based testing to this application when testing the Queue to better improve the quality of the application.

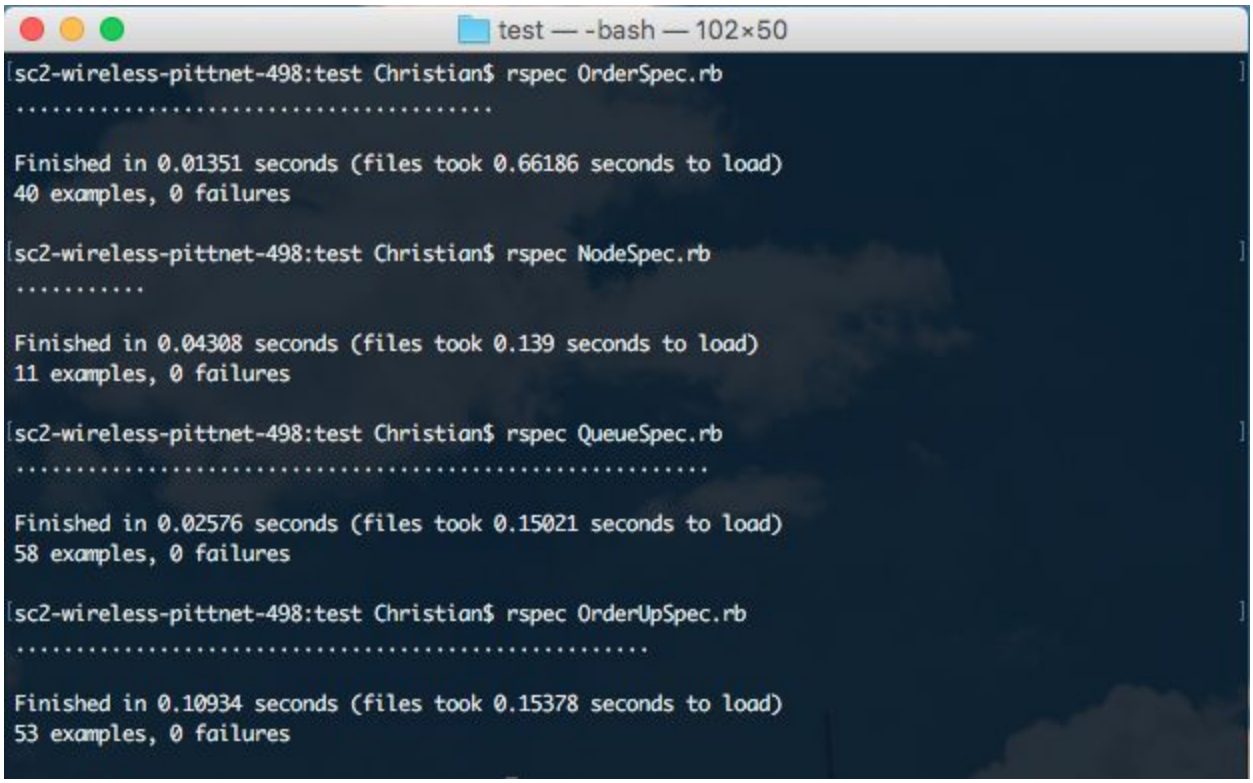
Quality Assessment

The Order simulation has been developed using the Test Driven Development (TDD) methodology. As a result, nearly all of the code that has been written for the application has been previously tested for bugs and defects. More importantly, by using the TDD paradigm an entire unit test suite was implicitly created for the application. Referring to the screenshots pictured below, all of the unit tests for the application have passed. Additionally, there is a **99.46%** code coverage of the unit test suite for the application. As for my recommendation on whether the application is ready for release or not, I would not advise for immediate release. Despite **99.46%** code coverage and all of the unit tests passing, I still think further testing of the application should be done using the Property-Based testing methodology which was mentioned above.

Source Code

https://github.com/thisbechristian/quality-assurance/tree/master/deliverables/d6

Unit Tests (RSpec)



Code Coverage (SimpleCov)