Testing is a benchmark for how to measure success. There are many ways to achieve success in developing software that rely on strategies and testing suites. In Java we use JUnit tests to write unit tests, which are pieces of code that test pieces of code (typically a function) to understand how the code is progressing. Sometimes the development of tests is done before writing the application, which may be beneficial due to benchmarking the progress your application development is making by how the unit tests are doing.

For me JUnit tests were straightforward, the modules in the testing suite were purposefully named and understanding the documentation was easy. Learning the suite took some time, and understanding the setup was the hardest part; I have limited exposure to Java. Videos provided by our professor were easy to understand, but I was on my own when experiencing Java packages. The suite in JUnit has naming conventions that are straightforward and gaining an understanding of their purpose was done through context.

Once I was able to write one unit test, aiming for coverage became easy. I had an idea of what a unit test should be like from instruction in this course. The texts provided gave insight into the goals of testing and what unit tests should be looking for. Bringing about the success in practice was based on the learning that I had done earlier in the term. The text was deep in knowledge and provided a depth of learning for me.

In writing code to align with requirements, I would try to translate the text of the requirements into a piece of code, like having a requirement that an id could only be created once. I knew that there was an API for the goal of setting variables once and found the atomic library in Java.util. Once I had that code written, testing it took some thinking. What I produced was creating two instances of the objects using the service class and writing a getter for the id. It would then be easy to write a unit test that could test the functionality of the ids.

My unit tests were effective because for every function in the classes that were to be tested there were unit tests for that piece of code. This was a strategy I employed in every assignment and project. I had learned this from reading a discussion post that Professor Luo had replied to. His insight and instruction led me to be able to write tests that could be beneficial in the development process. I found these tests to be helpful, every change in code could be tested by my suite of unit tests. I wish I could have automated this process, but those ideas were just introduced to me.

The comparative analysis to highlight in software development is the use of unit test. A unit test is a small piece of software that can analyze the outputs of applications. This is a useful tool because the application is broken down into small chunks that are reliably tested to understand if edge cases are respected, logic is correct, and bugs are avoided. Reliable testing is achieved by many other methods and strategies.

There are several strategies you can use in unit testing to develop great apps. One way is to think of the application as a black box, in which you as a tester do not know how the code is written and where the outputs are sourced from. This allows for testers to eliminate biases in their tests. Along with other strategies in the chosen methodology in engineering the application, a tester can successfully work in a development team to ensure the application is up to standard and requirements are met.

Test driven development focuses on creating test cases so that you can understand the outputs before writing any code. From the requirements you directly write unit tests that are going to be used for testing. The written tests could allow for developers to understand what their code needs to do to be able to be accepted. The acceptance testing is a crucial step in the software development process and is also a part of automating pipelines too. Automating is another strategy in creating applications using tests.

Automation is a new development and has a specific engineering group called DevOps. These automation and quality engineers are responsible for writing tests and creating tools that will be used to drive acceptance testing and other operations like logging and reporting in IT. Automation takes a large amount of overhead but relies on a solid suite of testing methods to effectively produce applications.

Behavior driven development uses domain specific languages to create applications that are modeled to understand collaborators’ wishes in the uses of the application. In this methodology tests are removed from any coding and instead of writing tests in testing suites first a domain specific language is used. A domain specific language is typically any written language understood by the engineering group. This allows for a collaborative environment; the code can be talked about because it is written in English for requirements and tests.  
 Using these strategies and behaving as a software tester, I assumed that all code was broken and that outputs could fall through the cracks if unit tests were not written well. I used test driven development to write my unit tests in some assignments not only because it was a methodology, but I spent the most time writing these tests. If I worked on the tests first, I found I was effective at writing the application.

Bias exists when you test your own code because you feel like the work you did is perfect. It could also exist when you are exposed to the code prior to writing any tests. If I knew that there were logics that had holes, I could avoid it by writing a unit test that could still pass that piece of code. The function could fall through the cracks because I knew how the code was written.

Testing software is an extremely disciplined task. The responsibility of finding bugs and ensuring correct outputs is not a vital task but may be the most important part of development. There are many ways to tests so you must find the strategy that works for you. From there you must work on writing the best tests and avoid piecing together logic in tests that could fail to catch mistakes because you feel like your code is good enough. A test is a valuable tool in developing the best possible code.