CS-320 7-2 Assignment: Project Two Submission

Cade Bray

Cade.Bray@snhu.edu

Southern New Hampshire University

Angelo Luo, M.S

December 21st, 2024

As a software engineer for Grand Stand Systems I used a test driven development approach. Having knowledge of the inner workings of a system and designing tests around that knowledge is considered white box testing (NIST - White Box Testing, 2011). This contrasts with black box testing where there is no assumption to internal mechanisms (NIST - Black Box Testing, 2017). Test driven development allows an engineering group to align directly with the software requirements before any code is written. By determining the needs of the product and forming a unit test around them a development team can produce code that is purpose built and minimal.

While Black box testing was not used during this project it would be used in acquiring requirements and as a final testing iteration. Black box testing in the final iterations can be used to test driver expectations. This would include testing the input and output of a driver's functionality without considering the internal workings. This is primarily focused on systems that a user would directly interact with instead of system connection points. Black box testing might include non-functional testing which targets usability and specific user-oriented outcomes. Black box testing can be useful in products that have multiple public releases for functionality because regression testing focuses on existing functionality and ensuring continuity through code releases.

During the development phase I worked on unit testing and production code for three products including the Contact, Task, and Appointment class and drivers. My approach to building deliverables was methodical and repeatable. Using test-driven development my team identified the client's product requirements. Once these requirements were identified and approved by the product owner we moved to the next step, building test cases. These test cases

were used as minimal passing criteria for any production code built. Test cases were used for both the class and driver code to ensure minimal and desired outcomes.

The unit tests built included the happy path and sad path for potential outcomes. The happy path is defined as expected outcomes for predictable user entries. While the sad path is unexpected outcomes for unpredictable user entries. Tests for the happy path might include ensuring that a recently constructed class object returns the correct class name field previously given as seen in figure one. While a test for the sad path might include edge case testing seen in figure two (Morgan, Samaroo, Thompson, & Williams, 2019). Edge cases are given values that are usually unlikely but ensure a system can handle the maximum or minimum desired characters or one character past the maximum for an expected failure. Focusing too much on the happy path can result in a product that can contain potential security flaws for accepting values not previously expected.

Figure 1, Positive unit test creating an object and testing its return value for a function.

```
@Test
@DisplayName("Testing getAppointment_id() with positive test case.")
void getAppointment_id() {
  Appointment appointment = new Appointment("1234567890", new Date(), "Test");

  assertEquals("1234567890", appointment.getAppointment_id());
}
```

Figure 2, Negative test case that shows a previous date being assigned for an expected future date.

```
@Test
@DisplayName("Testing setDate() with Negative test case, Past Date.")
void setDatePast() {

   Date pastDate = new Date();
   // Set the date to be 1 day in the past
   pastDate.setTime(pastDate.getTime() - 1000 * 60 * 60 * 24);

   Appointment appointment = new Appointment("1234567890", new Date(), "Test");
   assertThrows(IllegalArgumentException.class, () -> {
      appointment.setDate(pastDate);
   });
}
```

Testing coverage is another concern that was taken into consideration during my unit testing. Ensuring that the collective of all tests reaches all lines of code is the desirable ratio that the metric test coverage conveys (Coverage measurement in model-based testing, 2024). During this development I was able to achieve 100% test coverage in all three class and driver code bases. Gathering test coverage ratios can be tedious but there are many tools that help gather this information such as IntelliJ IDEA Code Coverage tool that is built directly into the IDE (JetBrains, n.d.). While these tools can be time savers for development it should be noted that they can be misleading, and any deliverables should be evaluated by an individual of the testing team.

My experience with Junit testing has been extremely valuable. Using test driven development and tools such as Junit has allowed me to deliver purpose-built code that operates at a higher standard. I can ensure that my code is technically sound by not only testing both the

happy and sad path but testing for expected points of failure as seen in figure three. By testing for expected points of failure I can ensure that the products I deliver are within defined parameters. In addition, my technical soundness can be illustrated in the code's efficiency. Code efficiency is calculated typically in Big O notation. Throughout all three platforms, all code was O(n) or better in terms of time complexity.

Figure 3, Test case that ensures assigning a 'null' argument results in an error.

```
@Test
@DisplayName("Testing setAppointment_id() with null variable.")
void setAppointment_idNull() {
  assertThrows(IllegalArgumentException.class, () -> {
    new Appointment(null, new Date(), "Test");
  });
}
```

The mindset I've adopted while working on this project can only be described as testing as a focal point. While working as a software tester on this project I employed caution by ensuring that code coverage met a desirable level, all requirements had a happy path test constructed, all requirements had a sad path test constructed including edge case testing, and I worked to view the project from a security standpoint. It's important to appreciate the complexity and interrelationships of the code because it allows a tester to identify software requirements that are more technical, build more comprehensive test cases, and identify issues in the requirements acquisition stage before development time is spent (Morgan, Samaroo, Thompson, & Williams, 2019). This is illustrated in my code because while not rigidly defined the product indirectly called for a search function to be generated and tested to search through

the array list as seen in figure four. This functionality needs to be documented and tested rigorously in addition to client product requirements.

Figure 4, search engine for objects in an array list.

```
public Appointment findAppointment(String id) {
  for (Appointment appointment : appointments) {
    if (appointment.getAppointment_id().equals(id)) {
      return appointment;
    }
  }
  throw new IllegalArgumentException("Appointment not found");
}
```

I limited my bias throughout the project by setting aside my assumptions that code will work as intended because of its simplicity and created a rigorous test case to ensure coverage. Using strict methodology helps eliminate bias and focus on a larger scope objective and the framework for this is the seven phases of Software Testing Lifecycle (STLC) (Morgan, Samaroo, Thompson, & Williams, 2019) Finally, discipline is crucial for a software tester. Without discipline to seek out code coverage and test requirement coverage a software tester might cut corners and make assumptions on outcomes of code. If the project is multistage, it's important for a software tester to focus on how releases connect with each other and code can be reused across the entire project's life instead of that specific release to avoid accruing technical debt.

References

Garousi, V., Keleş, A. B., Balaman, Y., Mermer, A., & Güler, Z. Ö. (2024, May 27-31). Coverage measurement in model-based testing of web applications: Tool support and an industrial experience report. *IEEE Xplore*, 37-43. doi:10.1109/ICSTW60967.2024.00019

JetBrains. (n.d.). *Code coverage*. Retrieved from Intellij IDEA : https://www.jetbrains.com/help/idea/code-coverage.html

Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing : An istqb-bcs certified tester foundation guide* (4th ed.). (B. Hambling, Ed.) BCS Learning & Development Limited.

National Institute of Standard and Technology. (2011, September). *Computer Security Resource Center - Glossary - White Box Testing*. Retrieved from Information Technology Laboratory: https://csrc.nist.gov/glossary/term/white_box_testing

National Institute of Standard and Technology. (2017, June). *Computer Security Resource Center - Glossary - Black Box Testing*. Retrieved from Information Technology Laboratory: https://csrc.nist.gov/glossary/term/black_box_testing