**Southern New Hampshire University**

Joshua Wozny

[joshua.wozny@snhu.edu](mailto:joshua.wozny@snhu.edu)

June 7, 2023

**CS-320**

**Project Two: Testing Reflection**

SUMMARY

Developing software requires identifying and implementing system requirements. Through requirements gathering, we’ve identified that the application requires contact, task, and appointment entities and their associated service to manage each. Unit tests were used to verify that each entity and service behaves as expected with all required features.

JUnit is an unit testing framework for Java that can be used to integrate testing into the development process – especially when combined with an IDE and a build system like Maven. JUnit was used to ensure that all requirements have been met within our implementation of the following entities and service: Entity, EntityService,Contact, ContactService, Task, TaskService, Appointment, and AppointmentService. Unit Tests were re-run after all changes to the code to ensure that no breaking changes were made. Generally, the unit tests used in this project have used White Box testing, since the inner workings of the software being tested are known (Boni Garcia, 2017).

Test-driven development (TDD) was also used to some extent when developing the Appointment class. By first creating a failing unit test, and then adding the minimum code to pass the test, it was assured that the class would meet the requirements without any scope creep. With this approach the code was refactored as each additional test was created, and additional code added, ensuring clean and efficient code. Alongside this approach, I also used property-based testing for each of the entities, by selecting edge cases that should pass or fail, rather than just selecting cases that are most likely to arise under normal use of the application. Using these approaches we can feel confident that the requirements of the application hold true under normal and edge cases (Improve your code with the property-based testing | Promyze, n.d.).

Each property of the Entities, contact, task, and appointment was tested to ensure that valid strings were allowed, and that invalid strings were disallowed. This was accomplished through the use of comparisons (id==null, task.description.length()>50, appointment.appointmentDate\ .before(newDate()!=true) to throw exceptions for values outside of the expected range. With 100% code coverage I am confident that these services function as intended.

I used best practices to ensure technically sound code, using inheritance for Contacts, Tasks and Appointments by extending inheriting from Entity, an abstract class used to reuse code for each class that requires an Id. This helped standardized and simplified how Ids are treated. I also used an EntityService class to standardize how each classes Service manages itself by providing an API for adding, updating, and deleting it’s objects. Regression testing was conducted to ensure that each change (through the addition of new classes or through refactoring) did not break previously functional code. Our unit tests performed integration tests between each entity and the EntityService API through their individual service. Integration testing ensures that individually unit tested components interact with each other as expected (Boni Garcia, 2017).

Using the correct data structures helped to make sure that the code is efficient, using HashMap to store contacts and tasks in their associated services. Using the HashMap structure made the Services easier to implement and makes keeping unique values for Ids straightforward since the Id was used as the key of the HashMap, preventing non-unique values within each service.

Using clean coding practices, good commenting, and self-documenting code through naming conventions, the current code base is in good shape and ready to be added to and get integrated into a larger application. As more code is developed, additional refactoring may be needed to ensure that the application can continue to be extended, and will be easy to understood by other developers. The unit tests will also aid other developers to understand the software requirements as they clearly outline what these classes are designed to do .

Other testing techniques which should be considered to ensure that the implementation is error free and meets client expectations. Static testing with a thorough code review by another, more experienced, developer is a great way to ensure best practices are being used. End-To-End testing using Black Box testing verifies that the entire system as a whole behaves how we expect it to, regardless of the internal implementation (15 testing methods all developers should know | CircleCI, n.d.).

REFLECTION

A significant driver of test development is evaluating risk, “a factor that could result in future negative consequences, usually expressed as impact and likelihood” (Hambling, Brian, et al, 2019, pp 158). Good testing design requires identifying risks, their likelihood, and potential costs. This requires caution and discipline, and careful examination of the interrelationships within the code under test while understanding its complexity to ensure proper test coverage and risk mitigation. (Hambling, Brian, et al, 2019).

Sometimes project manager may find themselves in a situation where a project is behind, over budget, or both and may need to consider cutting corners to get back on track. Often they may believe that testing and code refactoring may be areas that can be trimmed. However, cutting corners often leads to the accumulation of technical debt, which refers to the additional work required to fix or improve code in the future due to shortcuts taken in the present (Technical debt - Wikipedia, n.d.). By investing time and effort in writing clean and well-tested code, developers can ensure that the codebase remains maintainable and easier to understand and modify in the long run. Neglecting this can result in increased development time, decreased productivity, and a higher likelihood of introducing bugs or regressions. This should be carefully considered before these activities are cut.

It is important to minimize bias while developing and implementing a testing strategy. Developers naturally believe that their software adequately meets all requirements and may overlook edge cases or be less deliberate while testing software, while testers believe that every piece of code likely contains s vulnerability, missing feature, or other defect and may desire to extend testing by pursuing test cases ad infinitum rather than making efficient use of their time. Balancing these opposing mindsets is key to creating a useful and efficient testing strategy.

When designing test cases, it's crucial to exercise caution to ensure that the tests cover all possible scenarios and edge cases. Failure to do so may result in overlooking critical issues that could impact the software's functionality, security, or performance. The importance of employing caution and appreciating the complexity and interrelationships of the code during software testing can be described by each of these categories: Caution, Understanding, Integration, Risk, and Regression (Hambling, Brian, et al, 2019).

* Caution in Test Design: When designing test cases, it's crucial to exercise caution to ensure that the tests cover all possible scenarios and edge cases. For example, in a financial application, a tester should exercise caution to test various conditions, such as testing with minimum and maximum values, to ensure accurate calculations and prevent potential financial errors (Improve your code with the property-based testing | Promyze, n.d.).
* Understanding Code Interrelationships: Appreciating the complexity and interrelationships of the code being tested is vital to identify potential areas of impact and ensure comprehensive testing. For instance, if a tester is validating a module that interacts with a database, they should understand the underlying database schema, relationships between tables, and how data manipulation affects other parts of the system to design effective tests (Boni Garcia, 2017).
* Integration Testing: Testing the interaction between different modules or components is critical to validate they work together as intended. In complex systems, various modules may rely on one another, and changes in one module may affect others. For example, in an online shopping application, a tester must ensure that the shopping cart object integrates seamlessly with the payment gateway service to enable successful transactions (15 testing methods all developers should know | CircleCI, n.d.).
* Risk Assessment: Understanding the complexity of the code helps testers assess the risks associated with different parts of the software allowing testers to allocate time and resources for testing appropriately and ensure thorough test coverage. For instance, in a medical software application, a tester must recognize the complexity and sensitivity of models used for diagnosing diseases to prioritize testing to minimize the risk of incorrect diagnoses (Hambling, Brian, et al, 2019).
* Regression Testing: When modifications or new features are added to a codebase, it is crucial to understand the interdependencies and retest previously functioning components. For example, when adding a new feature to a messaging application, testers should rerun tests for message sending, receiving, and notifications to verify that the changes haven't affected these previously implemented features (Boni Garcia ,2017).

In summary, employing caution and understanding the complexity and interrelationships of the code being tested is essential for comprehensive testing, risk mitigation, and maintaining the overall quality and integrity of the software. It helps testers identify critical areas, design effective tests, and minimize the chances of introducing unintended issues.

**References**

15 testing methods all developers should know | CircleCI. (n.d.). Circleci. <https://circleci.com/blog/testing-methods-all-developers-should-know/?utm_source=google>.

Boni Garcia. (2017). Mastering Software Testing with JUnit 5 : A Comprehensive, Hands-on Guide on Unit Testing Framework for Java Programming Language. Packt Publishing.

Improve your code with the property-based testing | Promyze. (n.d.). Promyze. <https://promyze.com/code-improvements-property-based-testing-fast-check/>.

Hambling, Brian, et al. *Software Testing : An ISTQB-BCS Certified Tester Foundation guide - 4th edition*, edited by Brian Hambling, BCS Learning & Development Limited, 2019. *ProQuest Ebook Central*, <https://ebookcentral-proquest-com.ezproxy.snhu.edu/lib/snhu-ebooks/detail.action?docID=5837074>.

Technical debt - Wikipedia. (n.d.). En. https://en.wikipedia.org/wiki/Technical\_debt.