**7-2 Project Two**

Robert A. Rust III

Department of COCE-CS, Southern New Hampshire University

CS-320: Software Testing, Automation, and Quality Assurance

Professor Karl Lewis, Dr. Eng.

June 23, 2024

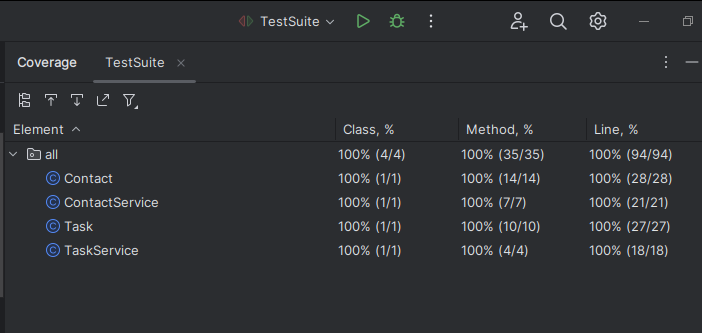
**Unit Testing Approach**

The testing approach used was directly contrived from the software requirements. Given each software requirement, the software tests to ensure that requirement is met. For example, one software requirement is “[the] task service shall be able to add tasks with a unique ID.” A couple tests can be created based on this single requirement. One is a test to ensure that the TaskService is able to add tasks to some form of storage. Along with this, to test if the task was successfully added, the test must ensure that it was retrieved successfully. In addition, there must be a test to address the “unique” requirement. The add task test must also ensure that two tasks with identical IDs cannot be added.

In the test for adding a task, first, new TaskService and Task objects are created. Then, the task is added to some form of storage via the TaskService. The form of storage is not relevant, it could be any kind of database, in memory in the form of any data structure, or in a text file. We rely on the fact that the form of storage is itself tested and works to its own specifications. Once the task has been added, we must ensure that it has actually been added if the TaskService behaves as if it has been added. We do this by ensuring that the Task object in storage is not null (it has been stored), and that its fields are what we expect. Then, we test some invalid ways of adding tasks, such as adding a null task, or a task that already exists.

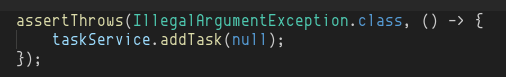
**Effectiveness of JUnit Tests**

Test coverage is the percentage of all lines of code that are executed by the testing functions. If there are multiple different paths that can be taken in a function (conditional statements), just because the function was called in the tests doesn’t mean that each line of code in the function has been tested. Getting one-hundred percent coverage in a large and complicated codebase is usually an unrealistic goal. It is a good metric to use to ensure that you’re testing as much of the code as possible. One-hundred percent test coverage, however, does not mean that the code is free of errors. In the case of the ContactService and TaskService project, I was able to achieve one-hundred percent code coverage.

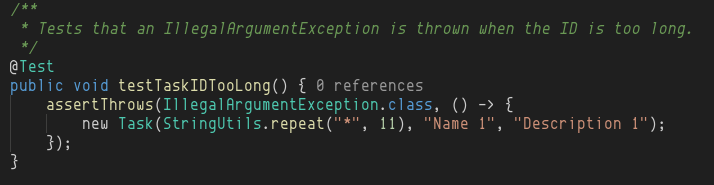
This was an achievable goal as the entire project was only a couple hundred lines of code.

**Ensuring Technically Sound Code**

To ensure technically sound code, testing can be used to validate that the functionality is what we expect. It is especially important to pass in unexpected inputs to functions and ensure that they handle errors gracefully. I did this in my tests by passing in null arguments to functions to ensure that the function does not attempt to dereference a null pointer if an invalid argument is accidentally passed.



This check guarantees that passing null arguments throws an error, forcing developers to handle such cases. Another way to verify that code is technically sound is to test edge cases. Edge cases are inputs that are on the *boundaries* of the domain of the function. For example, the software specification for the Task ID field was that it cannot be more than 10 characters. Passing a string that is 11 characters long checks for off-by-one errors in the code.



**Ensuring Efficient Code**

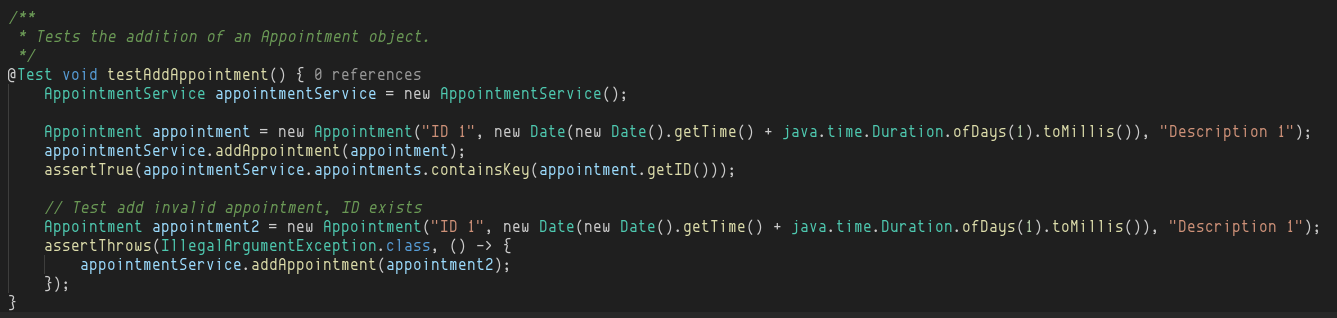
Using testing to validate that code runs at an acceptable efficiency standard helps protect against a poor user experience once the product launches. To test efficiency, rigorous tests should be formulated which push the code to its limits. If a software specification is that the code could hold up to 10,000 tasks, tests should be run at this 10,000-task limit. Performance profiling can be done within the tests such as checking runtime by saving the current time before the function call and after the function call and verifying that the difference is within an acceptable range. There are also performance profiling tools that can be used to check for efficiency such as JProfiler which can be used to pin down performance bottlenecks. There were no specific performance constraints in the software specification for the Contact or Task projects, but some reasonable estimates can be made and tested for.

**Software Testing Techniques**

**Techniques Employed**

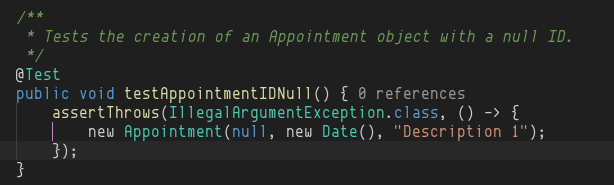
***Functional Testing***

The software specification came with functional requirements to meet. Functional testing tests that the functional software requirements are met. Functional software requirements are requirements that define the functions the system should perform for the end-user.

This code snippet tests the functionality of the system. The functional requirement was that the user must be able to create appointments.

***Non-Functional Testing***

Non-functional testing was used to ensure that the system behaved as expected and invalid inputs could not be passed to functions. For example, in each checkpoint, the IDs must be unique, not null, and not updatable. This non-functional requirement was tested to ensure that the software throws an error if a user (or developer) attempts to initialize an object with a null ID.

Due to Java’s static type checking and code analysis by modern IDEs, tests were not needed to ensure that the ID could not be changed, as the Java keyword `final` ensures that the value cannot be changed after initialization. This non-functional test ensures that the code is secure and reliable. It is non-functional because users could still add appointments if this were not tested, but it validates that the code is robust and performs well when faced with unpredictable end-users. It also keeps the code maintainable and scalable, as it notifies developers when they do something incorrectly. Reusable tests mean the test can be written once and reused for the lifetime of the software system (so long as the requirements do not change).

***Other Techniques***

A software testing technique that was not employed is usability testing. This involves observing real end-users interacting with the system and noting what they struggle with and the issues they face. This provides valuable feedback which the team can then use to make the software better for whom it matters – the users. Oftentimes, the developers will have a greater understanding of the system and how it should be used, so their opinion on the functionality of the system is biased. Furthermore, stakeholders can view the real application and compare it to their expectations.

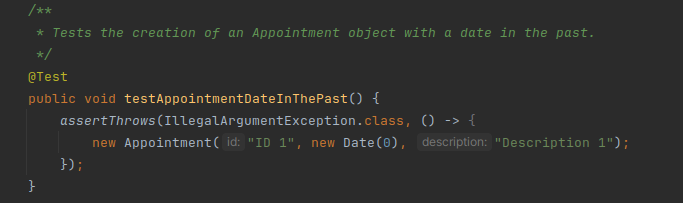
***Uses and Implications of Techniques***

Each of these software testing techniques discussed is a valuable tool that can increase the quality of the software. Functional testing is used to ensure that the system's functional requirements are met as the user expects. Just because the requirements are met, however, does not mean that the software is ready to be shipped. The system must also be easy to use and intuitive, which is where usability testing can be employed. This creates a feedback loop that makes it much easier to ship a good product. Non-functional testing is useful in all situations. It keeps the software maintainable and scalable. After a product ships, it cannot just be ignored by developers. The software will need to be maintained and possibly extended. Dependencies will update, possibly causing breaking changes that need to be fixed. It must perform and be secure to be usable. Overall, each software technique has its benefits and situations where it is best used.

**Mindset**

***Employing Caution***

The right mindset for testing is crucial both for coming up with a successful testing strategy and implementing it. Being cautious about potential risks and issues that may arise is essential in writing good tests. Thinking about all possible inputs and what the expected behavior of the program should be is necessary to understand before writing tests. Software has specifications, but as testers, it can’t be relied upon that people or systems interacting with the software will fully understand it and know how to use it properly. This is why having a destructive mindset when writing tests is helpful. Coming up with ways to break the code in every possible way will only make the code more robust and reliable.

For example, it is not logical to have the ability to create an appointment with a date that is in the past, but a user can accidentally select the wrong date and not notice. The software should expect this possibility and provide useful error messages to guide the user in the right direction in solving their problem. Being cautious about the code and respecting its complexity allowed me to predict this possibility.

***Limiting Bias***

Bias is dangerous in software testing as it can cause large problems with the code to get overlooked. If the tester wrote the code, the tester may be biased towards their code and think it is better or more robust than it is. This can cause problems as the tester may underestimate the ability of certain areas of the code to cause problems. Going into testing, the author of the code may view the code as they did when they wrote it, neglecting to account for other perspectives. This is an example of congruence bias, causing certain types of tests to be overlooked (Shenk, 2018). Another type of bias is confirmation bias. This occurs when testers view passing tests as confirmation that their code is correct and become overconfident in the durability of the code. This can be mitigated by discipline, always thinking of new ways to test the code, and never viewing passing tests as proof that the software is bug-free. To limit the effects of bias, I viewed passing tests only as proof that the code works for the specific case was written as.

***Being Disciplined***

Disipline is fundamental to being a good quality assurance practitioner. Getting lenient in testing the quality of software products even for something that may seem simple can be costly. There have been many cases in the real-world in which seemingly harmless bugs have cost lives. Even if the code being written is not mission-critical or people’s lives aren’t depending on it, it can cause many problems. These problems can cost businesses millions of dollars in the form of downtime, lost data, and other business setbacks. In testing, cutting corners invites biases to take control and make developers feel that their code is ready for production. It is essential to always remain disciplined and rigorous in testing any software.

**References**

Shenk, G. (2018). *The Impact of Cognitive Bias on Software Testing*. <https://www.functionize.com/blog/the-impact-of-cognitive-bias-on-software-testing>