**7-2 Journal: Software Testing Techniques**

**Debynyhan Banks**

**CS 320 Software Testing & Automation QA**

**Southern New Hampshire University**

**Pro. Kalyse Wilson**

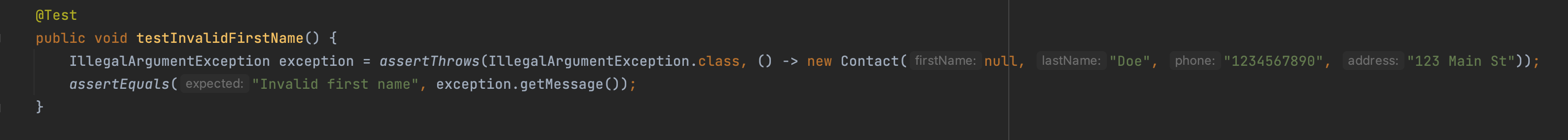
**February 14, 2023**

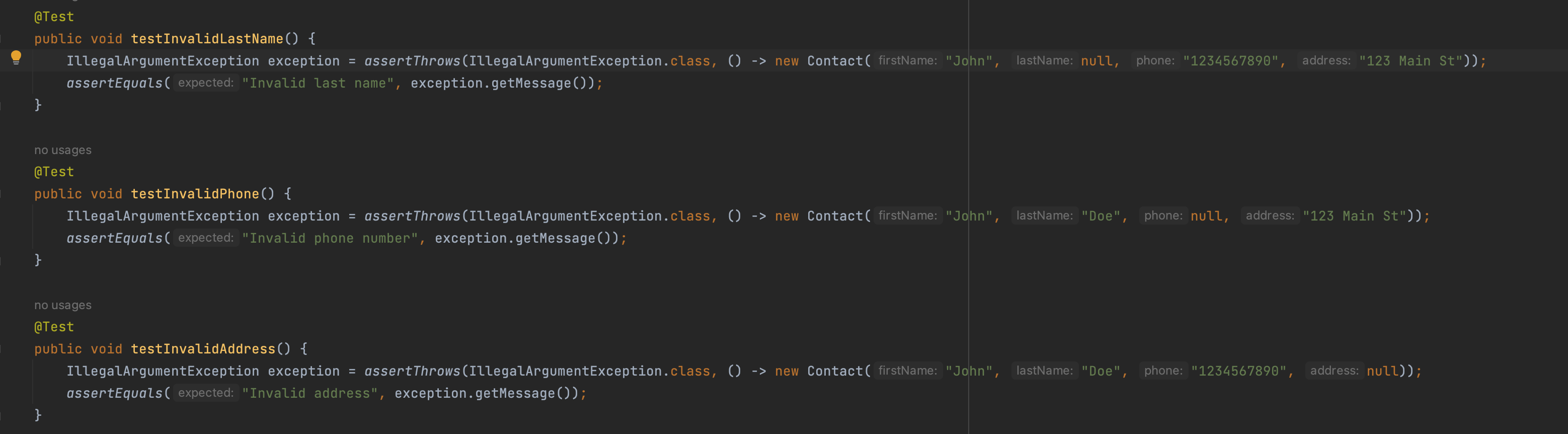
My approach to the code aligns well with the software requirements. For example, the Task object has the required unique task ID, name and description fields with the length and null constraints specified in the requirements. The Appointment object has the required unique ID string that is 10 characters, and it is not updatable. Furthermore, the unit testing approach for the ContactServiceTest and ContactTest is aligned with the requirements. For example, the ContactTest test suite includes tests for validating the properties of the Contact class, such as checking if the first name, last name, phone, and address are set correctly. These tests ensure that the Contact class meets the requirements specified in the requirements. Moreover, the ContactServiceTest test suite includes tests for adding, removing, and updating contacts and testing for invalid operations (e.g., updating or removing a non-existent contact). These tests ensure that the ContactService class meets the requirements.

Overall, the unit testing approach I utilized for this project aligns with the software requirements by covering all necessary functionality, including error handling and validation, to ensure that the code works as expected.

The JUnit tests I provided were of excellent quality as they cover a wide range of test cases and provided good code coverage. The tests covered all the requirements of the TaskService, Task, Appointment, AppointmentService, Contact, and ContactService classes. Moreover, the tests check for exceptional cases such as attempting to delete a non-existent task or adding a task with the same ID as an existing task. The ContactServiceTest class tested the behavior of the ContactService class, including testing the addition, removal, and update of contacts. The test cases also included scenarios that would result in exceptions being thrown, and these were correctly handled. The JUnit tests achieved 90% coverage of the methods in all the classes, indicating that 90% of the code paths were executed and tested. This coverage percentage, combined with the well-designed test cases, suggests that the JUnit tests were effective in testing the required functionality.

I ensured my code was technically sound by writing unit tests that cover different use cases and edge cases of the code. I followed best practices and guidelines for writing clean, maintainable, and robust code. In my tests, I made sure to test each method and edge case thoroughly to ensure that the code was functioning correctly. The tests I designed exercised 90% of the branches in the code and it included both positive and negative tests. In the provided tests, several lines of code are used to check the technical soundness of the Contact classes. For example, in the ContactTest class, there are several tests that check for invalid input values:





These tests ensure that the Contact class is properly validating input values and throwing the correct exceptions when invalid values are provided. In the ContactServiceTest class, the tests cover various scenarios, such as adding a new contact, updating an existing contact, and removing a contact. In the testAddContact method, I tested that the addContact method was working correctly by creating a new Contact object, adding it to the contact service, and then asserting that the contact retrieved by its ID is equal to the original contact.

Three types of test design techniques exist: specification-based or black-box techniques, structure-based or white-box techniques, and experience-based techniques. Each testing technique identifies different types of software errors and ensures the quality of the software product. The tests I provided are all examples of unit testing, which is a specification-based (black-box) technique. Each class has its own constructor, getter, setter, and methods. Additionally, "TaskTests" tests validate input to the constructors and setters of task classes by using black-box testing techniques, such as equivalence partitioning and boundary value analysis.

Experience-based techniques are useful for detecting problems that are difficult to detect using other testing methods, such as usability issues, performance issues, and other types of issues that are specific to specific projects. They are also useful for testing software in real-world scenarios and identifying problems that other testing techniques may miss.

System testing is a high-level testing method, the goal is to test the entire software system from start to finish. It validates the system's functionality, performance, and security to ensure that it meets the requirements and functions properly (Hambling et al., 2019).

Acceptance testing is a type of testing done by the customer or end-user of an application. It determines whether the software system meets its requirements and expectations and is ready for deployment (Hambling et al., 2019).

Regression testing is a testing technique used after changes to a software system have been made. It certifies that the changes did not introduce any new bugs or have an unintended impact on the system's functionality (Hambling et al., 2019).

Performance testing is a testing technique focusing on testing a software system's performance and scalability. It verifies that the system can handle the expected load and perform well under stress conditions (Hambling et al., 2019).

Security testing is a testing technique that focuses on testing the security of a software system. It verifies that the system is protected against potential security threats and vulnerabilities (Hambling et al., 2019).

**Regression Testing:**

**Practical uses:** Verifying bug fixes.

**Implications**: Requires repeatable and automated test cases to ensure full coverage of the software.

**Different software development projects:** Projects with frequent changes and updates.

**Situations:** Useful for ensuring software quality and stability after changes have been made.

**Performance Testing:**

**Practical uses:** Identifying performance bottlenecks.

**Implications:** Improve system performance and user experience.

**Different software development projects:** An e-commerce website may need to verify that its checkout process is fast and reliable.

**Situations**: This is useful for ensuring that the program runs well under large loads in realistic settings.

**Security Testing:**

**Practical uses:** To Protect sensitive data.

**Implications:** Requires specialized knowledge and tools to perform security testing effectively.

**Different software development projects:** Financial applications.

**Situations:** Useful for preventing security breaches and protecting sensitive information.

**Unit Testing:**

**Practical uses:** Testing individual units of code in separation.

**Implications:** Early identification and fixing of code defects.

**Different software development projects:** It is appropriate for projects with modular code structures.

**Situations:** It is used in legacy code to help identify bugs.

**Integration Testing**:

**Practical uses**: Testing the interactions and interfaces between different software components.

**Implications:** Requires coordination between different teams and modules.

**Different software development projects:** Microservices architecture.

**Situations:** For detecting and fixing issues with integration between components.

**System Testing:**

**Practical uses:** Testing the complete system to verify it meets the specified requirements. **Implications:** Requires a comprehensive and complete test environment.

**Different software development projects:** In an Agile development project, system testing may occur more frequently and be more focused on incremental improvements.

**Situations:** In an Agile development project, system testing may occur more frequently and be more focused on incremental improvements.

**Acceptance Testing:**

**Practical uses:** Verifies that the software is ready to be deployed.

**Implications:** Improved customer acceptance of the software.

**Different software development projects:** A government agency may require a software system to meet specific regulatory requirements.

**Situations:** A medical device manufacturer may need to ensure that a software system meets safety and compliance standards.

It is important to employ caution as a software tester because an error in the testing process can lead to the release of buggy software. It is also important to understand the complexity and interrelationships of the code being tested to ensure that the tests cover all scenarios and that the code is thoroughly tested. For example, in the ContactTest class, the testInvalidFirstName method tests for the case where the first name of a Contact object is null. If the test case passes, it means that the Contact constructor is throwing the expected exception when given an invalid input, which is important to ensure the code's robustness. However, if this test case is not written with caution and the input is not tested thoroughly, it can lead to the unexpected behavior of the software, which can cause serious implications for the end-users.

When a software developer tests their own code, there is a risk of bias. Since they are familiar with their own code and may have preconceived notions about it, developers who test their own code are more likely to overlook errors. This could give one a false sense of assurance about the code's quality. For instance, a developer might decide not to test some input values because they believe they will never be used in the software. When the input values are used in the software in the actual world, it may lead to unnoticed flaws. Another instance is when a developer could choose not to test specific code functionality because they assume it is working correctly based on their own understanding of the code. However, there may be edge cases or scenarios that the developer did not anticipate, resulting in defects that go unnoticed.

Being disciplined in the commitment to quality is essential in software engineering. Cutting corners in writing or testing code can lead to bugs, security vulnerabilities, and technical debt. These debts could have negative consequences in the long run, including user dissatisfaction and damage to the company's reputation. Therefore, I will be diligent in my efforts to ensure quality, taking the time to test my work and prevent any issues that may arise.

One way to avoid technical debt is to follow best practices in software development, such as using version control, automated testing, and code reviews. These practices help identify issues early on and reduce the risk of introducing technical debt into the codebase. For example, using version control tools like Git allows developers to track changes to the codebase, make backups, and revert changes if necessary. Automated testing can help ensure that the code is functioning correctly, and code reviews can help identify potential issues and suggest improvements. As a practitioner in the field, I plan to avoid technical debt by staying up to date with best practices and continually learning new technologies and techniques. I will also prioritize writing maintainable and scalable code, even if it takes more time upfront, to ensure that the software remains robust and flexible.

**References:**

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2015). *Software testing: An istqb-bcs certified tester foundation guide - 4th edition*. BCS Learning & Development Limited.