Mary Peterson

8/2/2024

Journal 5 – Software Testing Techniques

In the three modules of this course, I used both unit testing and static testing, methodologies under white-box testing. Static testing involves thoroughly examining the source code against specifications to detect defects, which helped me identify and fix issues after JUnit test failures. During class method development, I aligned implementation logic with the documentation specifications and reviewed them while creating unit tests to find potential discrepancies. A failed JUnit test would prompt a static code review to identify the underlying issue, such as using an incorrect attribute within a logic branch.

I also realized that merely confirming an exception is thrown when anticipated is inadequate; catching the correct exception type is crucial. For instance, while testing a constructor expecting an exception due to invalid input, the wrong exception was thrown. Therefore, in Module 5's assignment, I assessed the exception type to ensure it aligned with expectations, preventing false positives. Integrating both testing methods deepened my understanding of the code and enhanced its quality, ensuring compliance with both functional and non-functional requirements.

Modules 3 and 4 had minimal system testing and no integration testing, while Module 5 included a few system tests. Each assignment required creating a base class and a service class to interact with it, allowing for system-level testing through the service class. This was partially done in Module 5 but not in Modules 3 and 4. Integration testing, which evaluates the entire application by coordinating all subsystems, was not conducted, even though it would overlap with system tests.

Additionally, all JUnit tests were manually run, without automation. Ideally, a service could automate JUnit tests after each build in Eclipse, which is crucial for larger applications. No security vulnerability scanning was performed either. While it is important to scan libraries and components for security issues, such as those in the SPRING framework or Java components, the limited use of external libraries, databases, or shared components made security testing less of a priority at this stage.

Automated testing is essential for large-scale applications and should be incorporated into a continuous integration pipeline, particularly when implementing continuous delivery practices. These tests can be executed at multiple phases, including during the build process, pre-deployment, and post-deployment. In my projects, I generally perform Unit tests and System tests during the build phase, while Integration tests are conducted prior to deployment. Automated testing facilitates the timely identification and resolution of issues, thereby preserving the integrity of the application throughout its lifecycle.

Unit testing is consistently advantageous, as it mitigates the risk of minor issues developing into significant complications. Unit tests enhance code quality by identifying overlooked logic branches, such as the handling of null values, and validating specifications, such as enforcing a maximum string length of 50 characters. As the codebase expands and components become more interdependent, the significance of System and Integration tests escalates. Furthermore, Security testing is critical, particularly in contexts involving data. Although my current projects do not utilize databases or user input—rendering countermeasures against vulnerabilities such as SQL injection or buffer overflows unnecessary at this stage—security considerations will become increasingly important as the system evolves.