**5-2 Journal: Software Testing Techniques**

Christine R. Emerson

Department of Computer Science, Southern New Hampshire University

CS-320: Software Test Automations

Dr. Toledo Lopez

March 27, 2023

* **What were the software testing techniques that you employed for each of the milestones? Describe their characteristics using specific details.**

The modules in this course have been tested with a combination of white-box and black-box approaches, including system, unit, and static testing. When creating an application or software, black-box, and white-box testing should almost always be combined. They analyze your software from different perspectives to better find any issues before it is released. By comparing code to specifications, static testing identifies bugs in the code. After one of my JUnit tests failed, I often used this technique to locate and fix the offending code. Creating the Contact, Task, and Appointment classes before anything else was necessary to meet the client's requirements. Each of which had service classes, ContactService, TaskService, and AppointmentService. Each class was tested against the specifications with J-Unit tests to ensure its functionality met the customer's requirements. Each class was tested using J-Unit directly. Before testing could be performed, the code had to be written according to the specifications.

For example, in the Task.java and TaskService files, requirements needed to be met, such as the ability to add tasks with a unique task Id that cannot be null, the length cannot be longer than ten characters, and it cannot be updated. A task object must have a required name String field no longer than twenty characters, not null, and updateable. The task object shall have a required description of no longer than 50 characters and shouldn't be null and be updatable. In addition, tasks can be deleted according to the task Id. For example, in the *Task.java* file:

Text

Description automatically generated

A line of code like the one above ensures that the taskId, taskName, and taskDescription do not fall within the parameters of being null or invalid length. If they do, they will throw an IllegalArgumentException, which was hardcoded in the method. The assertTrue method from the assertions API built into Java was chosen to test for the thrown during JUnit testing. It is the idea that if the class is properly constructed, there will be no exceptions, and the methods will work when passed the correct parameters is true. For example:

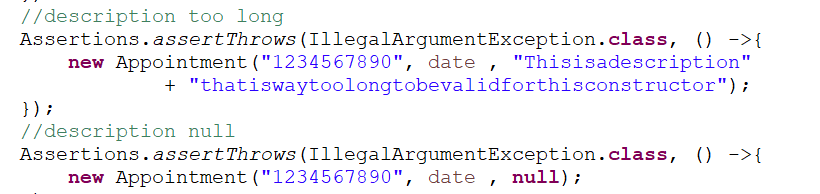
***ContactTest.java***

Text

Description automatically generated

Furthermore, there was a method that tested whether an exception was thrown when incorrect data was passed. An exception should be thrown if the method works correctly. The tests were also run with coverage to ensure that all J Unit tests covered as much code as possible. For example:

***AppointmentTest.java***



In addition, testing the functionality of updating objects using assertTrue, set, and get and the functionality of deleting objects by adding new objects and using a method to delete the contact by getting the object by the Id, which all tests were successful in each class-tested for this test. For example:

***ContactServiceTest.java***

Text

Description automatically generated with medium confidence

The simple act of throwing an exception when necessary is not enough. It is also crucial to catch the correct and expected exceptions.

* **What are the other software testing techniques that you did not use for the milestones? Describe their characteristics using specific details.**

There were no integration tests performed. Integration testing would test the entire application, bringing all the systems together for testing. In this case, the system and integration tests would be the same. I could, in theory, validate that the TaskService, AppointmentService, and any other classes all work together well at their integration layer – I have not. In addition, I did no automated testing – all my JUnit tests were manually run. In theory, I could have had a service that ran the JUnit tests after each build-event in the IDE. This is more important for large applications. There was also no security scanning performed. Libraries and components need to be scanned for security vulnerabilities, e.g., SPRING or even the Java components themselves. In this case, I am not extensively using any libraries, databases, or shared components so security testing could wait.

* **For each of the techniques you discussed, explain the practical uses and implications for different software development projects and situations.**

**Automatic unit** testing isolates codes to be tested independently in a test environment by replicating them, revealing dependencies between code units. Automatic Testing relies heavily on unit testing tools and frameworks. Automated tests should be built into the continuous integration pipeline when continuous delivery is being used - especially for large applications. Automated tests can be run at any stage of the development process, from development to deployment. Most of my projects run Unit tests, System tests at build time, and Integration tests just before deployment.

**Unit Testing** ensures that every unit works properly and can cause problems if not performed, significantly affecting the program's performance. Code can also be hardened by unit tests to prevent missed logic branches, like what if the value is null, and missed specifications, like how long a string can be.

**Static Testing** **checks software application defects without executing the code. Therefore, it is easier to identify and solve errors at an early stage of development with static Testing. It also helps find errors that Dynamic Testing may not see.**

**System Testing**is always essential to ensure the final product meets all client specifications, especially where data is concerned. Comparatively, **Integration Testing** ensures that connected units collaborate effectively. The more interconnected the components are within the code base, the more critical integration testing becomes.

**Security Testing** identifies vulnerabilities and risks associated with applications, systems, and networks. During security testing, potential threats to the system are assessed. The system's security is tested to identify potential security threats by performing both positive and negative tests. Identifying the dangers in the system and measuring its potential vulnerabilities is the primary goal of security testing so that the threats can be encountered, and the system doesn't stop functioning or cannot be exploited.