**Various Approaches to Software Testing Summary and Reflections Report**

Sorosh Khalili

Department of Computer Science, Southern New Hampshire University

CS-320: Software Test Automation & QA

Dr. Angel Cross

April 17, 2022

**Various Approaches to Software Testing Summary and Reflections Report**

Asides from finding bugs and any other possible flaws in the developed application, perhaps the other purpose of testing is to ensure that the developed software is completely aligned to the software requirements and produces expected results. These are all the required functionalities defined by the customer, and they must be addressed and presented in the final product. To achieve this goal, designed tests to evaluate each function of the application must be precisely aligned and meet the software requirements to ensure it is producing expected and desired results in real-time based on the real data. This is how the developing team measures the quality of the software and whether it meets the customer’s acceptance. All the designed tests for both milestones three and four were completely aligned and covered all the aspects of the software requirements.

One specific example is the following presented snippet of the code that checks to ensure that the contact ID is no longer than ten characters:

@Test

@DisplayName("Contact ID cannot have more than 10 characters")

**void** testContactIDWithMoreThanTenCharacters() {

Contact contact = **new** Contact("FirstName","LastName","PhoneNumbr","Address");

**if**(contact.getUniqueContactID().length() > 10) {

*fail*("Contact ID has more than 10 characters.");

}

}

The designed JUnit tests maintained a high overall quality since they were designed based upon requirements to test every important aspect of the software. To prove this fact, simply executing the JUnit test as the “Coverage As” option would reveal the high overall quality of the designed JUnit test. Conducted Junit test with the “Coverage As” option returns over 80% coverage results.

Simplicity is an important concept in any industry. In the software industry, simplicity is one of the important aspects that separates a good developer from the great one. The capability of producing simple and intuitive code that is easy to follow and covers all the software requirements is considered an important skill that only a great developer has. Although I have yet to reach that level, and there is a lot to learn, I have tried to follow that principle and produce simple and intuitive program code and JUnit tests for it. Within each test case, requirements were individually tested to keep test cases simple and ensure they are implemented properly.

One specific example is the following presented snippet of the code that checks to ensure the application properly creates the new task:

@Test

@DisplayName("Create New Task")

**void** testCreateNewTask() {

Task task = **new** Task("Task Name", "Task Description");

*assertTrue*(task.getTaskName().equals("Task Name"));

*assertTrue*(task.getDescription().equals("Task Description"));

}

Producing simple code not only improves readability but also helps with the reusability of the code. Hence, these lead to code efficiency.

One specific example is the following presented snippets of the code that checks to ensure the task ID and name each less than certain characters. Simply making very small changes to the first test criteria helps to create another test criteria to test another software requirement:

@Test

@DisplayName("Task ID cannot have more than 10 characters")

**void** testTaskIDWithMoreThanTenCharacters() {

Task task = **new** Task("TaskName","Description");

**if**(task.getTaskID().length() > 10) {

*fail*("Contact ID has more than 10 characters.");

}

}

@Test

@DisplayName("Task Name cannot have more than 20 characters")

**void** testTaskNameWithMoreThanTwentyCharacters() {

Task task = **new** Task("TaskName has more than 20 characters","Description");

**if**(task.getTaskName().length() > 20) {

*fail*("Task Name has more than 20 characters.");

}

}

To examine the compliance of the developed application with the gathered functional and non-functional requirements of the business, the developing team must utilize different software testing techniques. The challenge is to ensure that the testing is effective (Hambling et al., 2015). On the other hand, the developed software must address each business’s requirements appropriately because functional and non-functional requirements differ from business to business. software development consists of different stages, and because of their different specificities, each of those stages produces specific outcomes. Because of the unique characteristics of each development stage, errors or bugs results in specific types of defects. According to Hambling et al. (2015), tests should be designed to find as many defects as possible. Therefore, different types of testing techniques require to reveal those specific defects.

Software testing techniques consist of two main classes, each comprising different and special characteristics capable of discovering specific defects. Static and dynamic testing are the two main classes of software testing techniques. The static testing does not require the application’s code execution to discover its defects. The main objective of this testing technique is to avoid mistakes and errors in the early stages of software development to prevent costly defects later because it is much easier and costs less to fix errors in the early stages of the development. According to Hambling et al. (2015), static techniques find the causes of failures rather than the failure itself, which would be found during test execution. There are two main categories within static testing, review, and static analysis.

Review (informal review, walkthrough, technical review, inspection) is utilized to verify and inspect the design, code, test plans, and test cases for the system are aligned with the business requirement document. Static analysis (data flow, control flow, data structure) is the requirement and design examination to identify defects that can potentially cause failures in the system. The static analysis aims to discover defects within the source code of the software and its models. From the static testing perspective, the review (informal review) was the only testing technique utilized for each of the milestones because only business requirement documents were studied to determine and verify that the design, code, test plans, and test cases for the system are aligned with the specifications described in the business requirements document.

On the other hand, dynamic testing requires the application’s code execution to discover its defects. The main objective of this testing technique is to evaluate the application’s behavior dynamically to determine if it is producing the expected results based on the dynamic inputs during runtime. There are three main categories within dynamic testing, specification-based (black-box), structure-based (white-box), and experience-based. Specification-based (black-box) testing derives its test cases from the system model or specification. This testing technique utilizes business requirements document (functional and non-functional) analysis to create test cases regardless of internal system structure. The specification-based testing objective is to identify defects related to each given specification. Structure-based (white-box) testing derives its test cases directly from the system structure or component. This testing technique utilizes internal system design to create test cases. Experience-based testing is based on deriving test cases from the tester’s experience with similar systems and general experience of testing (Hambling et al., 2015). From the dynamic testing perspective, the specification-based was the only testing technique utilized for each of the milestones because only system specifications were tested to determine and verify that the designed system meets the specifications described in the business requirements document.

Testing must be an essential and inseparable, or simply put, an integral part of any software project, regardless of its scale and importance. Software testing (especially the quality-designed test) is important because its existence adds value to the software project and increases the software project's success significantly. On the other hand, the absence of software testing reduces the quality of the software and can lead to severe and catastrophic results. There are countless examples of software failures due to the lack of quality-designed software testing, or even worst, the absence of software testing completely that some have resulted in devastating consequences beyond repair. There are many benefits to software testing, such as bug prevention, development cost reduction, performance improvement, and customer satisfaction. And performing software testing, especially early testing, can reveal issues such as flaws in architecture and design, inaccurate or inoperative functionality, vulnerabilities caused by security flaws, and scalability problems.

Software projects are built based on the business requirements document presented to the developing team. The developing team then utilizes requirement elicitation to list all the functional and non-functional requirements of the software project to design it. Software testers utilize static testing (review, and static analysis) to examine the documents (models, design architecture, specification requirements, and codes) to ensure no error is introduced in this stage of the development. After developing the software code, software testers use dynamic testing (specification-based [black-box], structure-based [white-box], and experience-based) to determine how well the developed software aligned with the business requirements document.

Nearly every software development team tests their developing products, but still delivered software carries some defects. Although producing high-quality software is the responsibility of each developing team member, software testers become the last line of defense against any possible errors and defect present in the developing code. Following object-oriented programming and modularity design is one of the fundamental principles in programming. as a result, each portion of the software code can be responsible to achieve a goal and perform one or multiple tasks, but they may rely on other parts of the code (other written functions within the program code) to fulfill their task. This complexity and interrelationships within the developing code can lead to defects and failures if the related parts of the code produce and provide incorrect data to each other. The coverage and effectiveness of the tests cases in the testing process are important. Because of the complex nature of the software code and its interrelationship or dependency on other parts of the code, software testers must carefully pay attention to all the details even the small ones and be cautious that the created automated test does not miss testing of any part of the software.

One specific example is the following presented snippets of the code. Regardless of adding or updating a contact using the contact service, the program must produce correct and expected results. The software tester must ensure that the default constructor and setter functions check to ensure all the fields and inputs follow the guidelines presented in the business requirements document.

// Contact class Constructor

**public** Contact(String givenName, String surName, String phoneNumber, String physicalAddress) {

// Generate a unique ID and set its appropriate variable.

**this**.uniqueContactID = String.*valueOf*(*uniqueID*.getAndIncrement());

// The null or blank first name will be set to "NULL".

**if** (givenName == **null** || givenName.isBlank()) {

**this**.givenName = "NULL";

}

// It shortens longer than 10 characters' first names to the first 10 characters.

**else** **if**(givenName.length() > 10) {

**this**.givenName = givenName.substring(0,10);

}

**else** {

**this**.givenName = givenName;

}

// The null or blank last name will be set to "NULL".

**if** (surName == **null** || surName.isBlank()) {

**this**.surName = "NULL";

}

// It shortens longer than 10 characters' last names to the first 10 characters.

**else** **if**(surName.length() > 10) {

**this**.surName = surName.substring(0,10);

}

**else** {

**this**.surName = surName;

}

// The null, blank, or longer than 10 digits phone numbers will be set to "0123456789".

**if** (phoneNumber == **null** || phoneNumber.isBlank() || phoneNumber.length() != 10) {

**this**.phoneNumber = "0123456789";

}

**else** {

**this**.phoneNumber = phoneNumber;

}

// The null or blank address will be set to "NULL".

**if** (physicalAddress == **null** || physicalAddress.isBlank()) {

**this**.physicalAddress = "NULL";

}

// It shortens longer than 30 characters' addresses to the first 30 characters.

**else** **if**(physicalAddress.length() > 30) {

**this**.physicalAddress = physicalAddress.substring(0,30);

}

**else** {

**this**.physicalAddress = physicalAddress;

}

}

The program must also perform the same check as the default constructor within setter functions to ensure all the fields and inputs follow the guidelines presented in the business requirements document.

// Setters

**public** **void** setGivenName(String givenName) {

**if** (givenName == **null** || givenName.isBlank()) {

**this**.givenName = "NULL";

}

**else** **if**(givenName.length() > 10) {

**this**.givenName = givenName.substring(0,10);

}

**else** {

**this**.givenName = givenName;

}

}

**public** **void** setSurName(String surName) {

**if** (surName == **null** || surName.isBlank()) {

**this**.surName = "NULL";

}

**else** **if**(surName.length() > 10) {

**this**.surName = surName.substring(0,10);

}

**else** {

**this**.surName = surName;

}

}

**public** **void** setPhoneNumber(String phoneNumber) {

**if** (phoneNumber == **null** || phoneNumber.isBlank() || phoneNumber.length() != 10) {

**this**.phoneNumber = "0123456789";

}

**else** {

**this**.phoneNumber = phoneNumber;

}

}

**public** **void** setPhysicalAddress(String physicalAddress) {

**if** (physicalAddress == **null** || physicalAddress.isBlank()) {

**this**.physicalAddress = "NULL";

}

**else** **if**(physicalAddress.length() > 30) {

**this**.physicalAddress = physicalAddress.substring(0,30);

}

**else** {

**this**.physicalAddress = physicalAddress;

}

}

The following presented snippets of the test case code verifies that adding or updating functionalities correctly adds or updates the contact’s first name field:

@Test

@DisplayName("Test to Update First Name.")

@Order(1)

**void** testUpdateFirstName() {

ContactService service = **new** ContactService();

service.addContact("James", "Bond", "0123456789", "123 Dream Big Lane");

service.updateGivenName("Jacob", "0");

//service.displayContactList();

*assertEquals*("Jacob",service.getContact("0").getGivenName(), "First name was not updated.");

}

The best result final goal in software development is to develop and deliver a final product that exceeds customer satisfaction. The software developer must be open to the fact that developed code has always contained errors and defects. This means that software developers should welcome any errors and defects because it reduces the deficiency and increases the quality of the software. Following and developing test cases based on the business requirements document is a good example that can prevent any software developer’s bias.

Although measuring the negative results of an error or a very small mistake can put into perspective why it is important to test a product or procedure before deployment, it cannot stress enough the severe and catastrophic consequences that the affected entity must deal with. Due to the nature of the error, sometimes it may be hard to determine the actual cost of the damage but regardless of the cost, conducting extensive quality testing that discovers errors in the first place can significantly reduce severe consequences. Although errors are inevitable, performing testing thoroughly can help to discover and fix them. Time, budget, and development team members (manpower) are the most important assets and resources in any software development project. Testing early and often from the very early stages of the development process can lead to finding and fixing errors and defects before they become big and severe. Consequently, this approach saves previously pointed out resources.

**References**

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2015). *Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition).* BCS The Chartered Institute for IT. Retrieved from   
https://app.knovel.com/hotlink/toc/id:kpSTAIST01/software-testing-an-istqb/software-testing-an-istqb