Rose St.Clair

12/13/2024

CS-320

Summary:

The software requirements were defined in advance, specifying the expected actions under certain conditions or scenarios. For instance, in contact submission, one requirement states: "The contact object must include a required firstName field of type String, with a maximum length of 10 characters. The firstName field cannot be null."  This requirement was met with the code: if (firstName == null || firstName.isEmpty()) {

this.firstName = "NULL"

else if (firstName.length() > 10) {

this.firstName = firstName.substring(0, 10)

else {

this.firstName = firstName;

}

This example demonstrates how the given requirements align with the code, and when tested, the requirement is satisfied through tests for both null values and length: \*@Test\*

\*@DisplayName\*("Contact First Name cannot have more than 10 characters")

void testContactFirstNameWithMoreThanTenCharacters() {

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

if (contact.getFirstName().length() > 10) {

\*fail\*("First Name has more than 10 characters.");

}

}

}

\*@Test\*

\*@DisplayName\*("Contact First Name shall not be null")

void testContactFirstNameNotNull() {

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

\*assertNotNull\*(contact.getFirstName(), "First name was null.");

} Each test is run using JUnit and can either pass, result in an error, or fail. The tests ensure the requirements are met by adding, updating, deleting, and modifying data as specified. In both programs, all tests were executed successfully without any errors. Although complete code coverage does not guarantee that no errors will occur, it does help establish confidence in the program's functionality (Testing Code Coverage in Eclipse, n.d.).

To me, being technically sound means having the ability to write code that is both practical and efficient. This involves using clear formatting, adding comments, and keeping the code simple and organized. I also believe that comments are essential for explaining the functionality of the code to others, such as outlining the purpose of constructors or grouping code into sections for improved readability and usability. For instance, in the top section of Task.java, the constructors for the ID, name, and description are placed before the getters, along with an explanation of how the IDs are generated: public class Task {

private final String taskID;

private String taskName;

private String taskDesc;

private static AtomicLong \*idGenerator\* = new AtomicLong();

// CONSTRUCTORS

/\*

\* Constructor takes task ID, task name, and task description as parameters.

\* All parameters are checked if null or empty. If either exists, the field is filled

\* with the phrase "NULL" to protect data integrity as a placeholder. Task ID is truncated

\* to a maximum of 10 characters, task name to 20 characters, and description to 50 characters.

\*

\*/

public Task(String taskName, String taskDesc) {

// TASKID

// Task ID is generated when the constructor is called. It is set as a final variable and has

// no other getter or setter so there should be no way to change it.

// The idGenerator is static to prevent duplicates across all tasks.

this.taskID = String.\*valueOf\*(\*idGenerator\*.getAndIncrement());

if (taskName == null || taskName.isEmpty()) {

this.taskName = "NULL";

} else if (taskName.length() > 20) {

this.taskName = taskName.substring(0, 20);

} else {

this.taskName = taskName;

}

if (taskDesc == null || taskDesc.isEmpty()) {

this.taskDesc = "NULL";

} else if (taskDesc.length() > 50) {

this.taskDesc = taskDesc.substring(0, 50);

} else {

this.taskDesc = taskDesc;

}

} This code effectively meets the required specifications while also addressing potential issues in case the input is invalid. This is further supported by the implementation of the maximum length constraint for the name in another section of the Task.class code: if (taskName == null || taskName.isEmpty()) {

this.taskName = "NULL";

} else if (taskName.length() > 20) {

this.taskName = taskName.substring(0, 20);

} else {

this.taskName = taskName;

} Code efficiency is generally associated with resource usage, where efficient code consumes fewer resources to achieve its purpose compared to inefficient code (Bentley, 1981). An example of efficiency in the tests written and executed is that the code execution and test completion times are measured in hundredths or thousandths of a second, which serves as a strong indicator of efficient code.

Each milestone's objects are tested individually to verify that the code functions correctly and as expected. The results are displayed in both the console for testing purposes and in JUnit as a pass, fail, or error. One example of a test technique used in the milestones is the Equivalence Partitioning technique, which ensures that input conditions fall within a specified range (Hambling et al., 2015, pp. 87-88). For instance, this technique can be applied to validate the appointment description character limit of 50: public void setAppointmentDesc(String appointmentDesc) {

if (appointmentDesc == null || appointmentDesc.isEmpty()) {

this.appointmentDesc = "NULL";

} else if (appointmentDesc.length() > 50) {

this.appointmentDesc = appointmentDesc.substring(0, 50);

} else {

this.appointmentDesc = appointmentDesc;

}

} This example demonstrates the description limit of 50 characters, after which any additional characters are truncated. The Equivalence Partitioning (EP) technique is often combined with Boundary Value Analysis to test within a specified range, as well as the extreme ends of that range (Rungta, 2020).

However, there were several software testing techniques not used in the milestones. One such technique is Decision Table testing, which is primarily used to define functions, the conditions under which they operate, and the actions triggered by testing different combinations of these conditions (Hambling et al., 2015). Another technique not applied in the milestones is State Transition testing, which involves testing the behavior of an Application Under Test (AUT) as it transitions between different states (Rungta, 2020).

The Equivalence Partitioning technique was used extensively in the milestones to test character limits, such as those for descriptions, IDs, and names. These limits help ensure that data fits properly within fields, like tables, or avoid issues in situations where no limit exists, such as when posting on social media.

Boundary Value Analysis, as previously mentioned, tests values at the extremes of a given range. When combined with Equivalence Partitioning, this approach is often used in scenarios like password creation, ensuring that the character count and type fall within specific minimum and maximum limits (Rungta, 2020).

Decision Table testing is a technique used to assess combinations of conditions and actions, as illustrated in Example 4.3 of *Software Testing: An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition)* (Hambling et al., 2015, p. 92). In this example, different customer conditions at a supermarket trigger specific actions based on predefined rules.

State Transition testing differs from Decision Table testing in that it focuses on changes between states. Instead of examining combinations of inputs and conditions, it tests how the behavior changes based on transitions between the current and previous states (Hambling et al., 2015, p. 94). This technique can be likened to a button press, where the system remains in one state until the button is pressed, triggering a state change (Hambling et al., 2015, p. 94). It can be applied in scenarios where events are triggered by changes from one state to another.

According to O'Dell (2017), a significant portion of a software developer's time—ranging from 35 to 50 percent—is spent on validating and debugging software. With this in mind, adopting a problem-solving mindset can be highly beneficial. Testing often involves identifying and resolving issues, as well as proactively mitigating potential problems before they arise. This preventative approach is especially valuable in a testing environment.

It’s crucial to treat testing with the same level of importance and efficiency as coding. Viewing debugging as a problem-solving exercise can be an effective learning strategy (O'Dell, 2017), and applying a similar approach to testing can yield the same benefits. Tests are directly tied to the code, and both work together to ensure that the necessary requirements are met. For example, the connection between the first name string in a contact record, the code that enforces character limits, and the test that ensures the code works properly to meet these requirements. Below is the code for handling the first name in Contact.java: if (firstName == null || firstName.isEmpty()) {

this.firstName = "NULL";

// If first name is longer than 10 characters, just grab the first 10 characters

} else if (firstName.length() > 10) {

this.firstName = firstName.substring(0, 10);

} else {

this.firstName = firstName;

} This code is then tested to verify that it functions correctly and meets the specified requirements: \*@Test\*

\*@DisplayName\*("Contact First Name cannot have more than 10 characters")

void testContactFirstNameWithMoreThanTenCharacters() {

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

if (contact.getFirstName().length() > 10) {

\*fail\*("First Name has more than 10 characters.");

}

} When testing your own code, it’s crucial to minimize bias. Being overly confident in the effectiveness of your own code can lead to leniency in cutting corners or neglecting the thoroughness of testing. It’s important to approach your own code with the same level of constructive critique and focus on functionality as you would with someone else’s code.

As noted by Calikli et al. (2010), it is essential to consider failures alongside successes and avoid confirmation bias. Failing to apply the same diligence when testing your own code could result in poor testing practices, subpar code quality, or an overall decline in quality (Calikli et al., 2010).

Discipline in your craft is a vital skill, especially when consistency and efficiency are important, and when adhering to best practices in the field. Cutting corners in coding and testing can lead to unforeseen issues, potentially resulting in wasted time, resources, and the need to redo work (Farias, 2018).

I approach development similarly to how I approach hobbies or self-taught skills. Bad habits can impede the proper learning of a skill, which ultimately affects the quality of work. Cutting corners can hinder the opportunity to master a skill and apply it effectively. I adopt a “measure twice, cut once” mentality, striving to follow best practices in coding and testing, double-checking my work, and ensuring that my first attempt is well thought out. Maintaining a balance of humility and openness to learning can open up many opportunities for growth and improvement.

RESOURCES:

Bentley, J. L. (1981). *Writing Efficient Code.* CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE.

Calikli, G., Aslan, B., & Bener, A. (2010). Confirmation Bias in Software Development and Testing: An Analysis of the Effects of Company Size, Experience and Reasoning Skills - Open Research Online. *Open.ac.uk*. [https://doi.org/http://oro.open.ac.uk/45370/1/PPIG2010.pdf](https://doi.org/http:/oro.open.ac.uk/45370/1/PPIG2010.pdf)

Farias, H. (2018, July 2). *Where you should never cut corners in software development*. InfoWorld. <https://www.infoworld.com/article/3286265/where-you-should-never-cut-corners-in-software-development.html>

Hambling, Brian Morgan, Peter Samaroo, Angelina Thompson, Geoff Williams, Peter. (2015). Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition) - 4.5.1.3 Other Partitions. (pp. 87,88). BCS The Chartered Institute for IT. Retrieved from <https://app.knovel.com/hotlink/pdf/id:kt00UC2J6V/software-testing-an-istqb/other-partitions>

O’Dell, D. H. (2017). *The Debugging Mindset: Understanding the psychology of learning strategies leads to effective problem-solving skills.: Queue: Vol 15, No 1*. Queue. Retrieved from <https://dl.acm.org/doi/10.1145/3055301.3068754>

Rungta, K. (2020). *Software Testing Techniques with Test Case Design Examples*. Guru99.com; Guru99. Retrieved from <https://www.guru99.com/software-testing-techniques.html>

*Testing Code Coverage in Eclipse*. (n.d.). Cornell. Retrieved December 13, 2024 from <https://www.cs.cornell.edu/courses/JavaAndDS/files/codeCoverage.pdf>