Table of Contents

List of Tables

List of Codes

Questions and Solutions

1 Question 1

Q: As we are utilizing JUnit to implement test cases for our course project, you already know that @Test annotation is used to declare a method as a test method. In many test cases that you are currently making or made for Projects, you may have found that a pre-defined or post-defined environment/condition would be necessary to implement a successful test cases. Investigate and explain which annotations can be used to pre-define and post-define test case environments. *(10 points)*

Solution

In software testing with JUnit, it is often necessary to create a controlled testing environment to ensure the reliability and consistency of the outcome of tests. While the @Test annotation is used to indicate methods intended for testing, certain scenarios require the running of certain setup or teardown operations before and after these testing routines. JUnit addresses this need by supporting a set of annotations that allow developers to set up and tear down the test environment for the test cases beforehand and after the tests.

Core elements relevant to these fields include:

1. @BeforeEach:

This annotation is used to denote an action that is performed before executing every test method. It is especially useful for object initialization or restoring common states to ensure all tests are started in an unpolluted beginning. For example, in the case of testing a class that is used to simulate an IoT device, recreating the device before starting each test might be more desirable to avoid mixing states from other tests.

2. @AfterEach:

Methods with @AfterEach annotation are run after every test method. This is useful for doing cleanup operations, such as closing access to a file or closing connections to a service, to maintain test isolation and avoid resource leaks.

3. @BeforeAll:

This annotation means that there is a specific method which must be executed once before executing even one of the test methods in the class. It is often used for expensive setup operations, such as setting up database connections or servers, which need not be done for every test.

4. @AfterAll:

Similar to @BeforeAll, this annotation is used with methods intended to run once after all test methods have finished. It is usually used for deallocating global resources or doing final cleanup tasks.

Following is an example that illustrates usage of these marks in an ordinary JUnit test class:

```
public class PlugSimTests {
    private PlugSim plug;

    @BeforeEach
    void setUp() {
        plug = new PlugSim("testPlug"); // Initialize before each test
    }

    @AfterEach
    void tearDown() {
        plug = null; // Cleanup after each test
    }

    @Test
    void testSwitchOn() {
        plug.switchOn();
        assertTrue(plug.isOn());
    }
}
```

Code 1: Example usage in course projects.

In Code 1:

1. The setUp() function, prefixed with @BeforeEach, ensures that an object of type PlugSim is created anew before every single test.
2. The tearDown() method, annotated with @AfterEach, cleans up the object after each test.
3. Specifically, testSwitchOn() is completely focused upon the function in question, with, at the same time, careful regulation of the surrounding context both before and after applying it.

By strategically leveraging these annotations, developers can build robust and long-lasting test suites. Not only does this practice align with industry leaders' definitions of best practice in automation software testing, but also with the essential tenets of test-driven development as brought to light in course projects.

2 Question 2

Q: Consider our members application discussed in Lecture 20-24 *(15 points)*

2.1 Sub-question 1

Q: After the user clicks the checkbox to add/remove a member to/from a group, we wait until the server backend replies to update the state and update the view. What would happen if it took a while for the server backend to reply and the user clicks the same checkbox multiple times impatiently?

Solution

In interactive applications, such as the membership management application framework, grouping addition or exclusion operations are typically accomplished by asynchronous communication with a distant server. When the user checks or unchecks a checkbox in order to change group membership, the application makes a RESTful request, then waits for the response from the server before updating its local state and the UI. Even so, if the response from the server is slow, and the user quickly clicks the checkbox multiple times in succession, then many problems are likely to arise that are detrimental to both user experience as well as to the correctness of the application's state.

The major challenges likely to arise in such circumstances include:

1. Race Conditions and System State Inconsistencies:

Multiple successive clicks may result in many asynchronous requests sent to the server before receiving responses. Because responses can become disordered, the final state of the app may not reflect the actual action of the user. For instance, if someone clicks to add a member and then quickly clicks to delete the same member, then receives response to "add" request after receiving response to "delete" request, then the member will be displayed as added by the user interface even though the last action by the user is deleting that member.

2. Discrepancy in Backend Status and User Interface:

Accordingly, with backend authentication enforced by the application before updating the user interface, there is also a risk that the interface will slow down or even flicker as it processes every response. Such events have the ability to confuse users, who might have trouble understanding the absence of instant effect from their activity in the interface.

3. Potential Server Overuse:

Repeated and successive questions can put undue stress on the backend server, especially if each question spawns database transactions or other processes that consume significant resources.

To overcome these challenges, several strategies can be utilized:

4. Disabling User Interface Elements for Pending Inquiries:

A robust approach is to disable the checkbox (or related UI controls) while a request is pending, preventing further user interaction until the server has responded. In Java-based desktop applications using Swing, this can be accomplished by setting the checkbox's enabled state:

```java
import javax.swing.*;
import java.awt.event.*;

public class MemberCheckboxHandler {
    private JCheckBox memberCheckbox;
    private boolean isPending = false;

    public MemberCheckboxHandler(JCheckBox checkbox) {
        this.memberCheckbox = checkbox;
        this.memberCheckbox.addItemListener(new ItemListener() {
            @Override
            public void itemStateChanged(ItemEvent e) {
                if (!isPending) {
                    isPending = true;
                    memberCheckbox.setEnabled(false); // Disable checkbox
                    // Simulate sending request to server
                    new Thread(() -> {
                        try {
                            // Simulate network delay
                            Thread.sleep(2000);
                            // Update state based on server response
                        } catch (InterruptedException ex) {
                            ex.printStackTrace();
                        } finally {
                            SwingUtilities.invokeLater(() -> {
                                isPending = false;
                                memberCheckbox.setEnabled(true); // Re-enable checkbox
                            });
                        }
                    }).start();
                }
            }
        });
    }
}
```

Code 2: Java Code for setting a checkbox's state.

In Code 2, the checkbox is switched off as soon as a change is detected and will stay in such state until the end of the mock response from the server. Such behavior ensures that at no point is there either greater than one request in pending for processing, thus ensuring state correctness and overall system performance.

5. Decreasing or Managing User Engagements:
Another method is to use debouncing or throttling in the event handler, which ensures that only one request is executed in an allotted timeframe, regardless of how often the user is interacting with the checkbox.

6. Cautious Optimistic User Interface Improvements:
Some applications choose to apply immediate updates to the user interface, a technique known as optimistic updating, then undoing the update in case of an error from the server. This approach has the advantage of making the interface more responsive; however, it requires careful handling to avoid inconsistencies, especially when multiple operations are applied in a sequence.

In conclusion, delayed server responses combined with high volumes of user interactions with user interface elements can lead to race conditions, invalid application states, and overall diminished user experience. Strategies that include actions such as deactivating user interface elements pending requests, strict state handling policies, and possibly applying debouncing methods to user behavior can help address these issues, as discussed in lectures and showcased by the example code in Java.

2.2 Sub-question 2

Q: To address the potential issues in 2.1, someone proposes to disable the checkbox somehow after the user clicks it and enable it again once the RESTful response to get all groups arrives. List all events involved in implementing this solution.

Solution

To solve the problems that come with multiple selections of checkboxes within the wait period for a server response in the Members app, a widely supported and effective approach is the immediate disablement of the checkbox upon the client's click. The checkbox should only be re-enabled after receiving the RESTful response, as indicated by the refreshed list of groups. This approach ensures consistency within the graphical user interface while maintaining the integrity of the backend.

Implementation of the solution requires a set of predetermined procedures:

1. User Engagement Activities:
The process is initiated when the user checks or unchecks the checkbox to include or exclude a member from a group. Usually, this particular event is noted by an event listener attached to the checkbox element.

2. UI Disable Event:

When the application selects an element, the checkbox is instantly disabled. In doing so, extra user activity during the completion of the current operation is precluded, preventing the possibility of subsequent requests and race conditions.

3. RESTful Request Event:
The application makes a RESTful request (for example, a POST or a DELETE) to the back-end server, requesting the modification of the group membership. The request is made asynchronously, often by a background thread for applications written in Java.

4. Server Response Event:
After the server has finished processing the request, a response is sent back to the client. The response can include either the updated list of groups or an acknowledgment of the current state.

5. Model and UI Update Event:
When the server response is received, the application updates its internal model to reflect the new group membership and then updates the user interface in a corresponding way.

6. UI Re-enable Event:
Finally, the checkbox is re-enabled, allowing the user to interact with it again. This ensures that the user interface remains responsive and accurately reflects the state of the backend.

Summary Table of Events

Step	Event	Component/Layer	Description
1	User clicks checkbox	View/UI	User initiates add/remove action
2	Checkbox disabled	Controller/View	UI is updated to prevent further input
3	RESTful request sent	Controller/Backend	Asynchronous request updates group membership on server
4	Server response received	Backend/Controller	Server processes request and sends response
5	Model/UI updated	Controller/View	Application updates model and refreshes view
6	Checkbox re-enabled	Controller/View	UI is re-enabled for further user interaction

Table 1: Overview of Events.

Java Example: Simulating Checkbox Disable/Enable

The following example using Java Swing demonstrates this sequence of events. When the checkbox is clicked, it is disabled, a request is sent to a simulated server in a background thread, and then the checkbox is re-enabled when the "response" is received:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class CheckboxDisableExample {
    private JFrame frame;
    private JCheckBox memberCheckbox;
    private boolean isPending = false;

    public CheckboxDisableExample() {
        frame = new JFrame("Member Checkbox Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 150);
        frame.setLayout(new FlowLayout());

        memberCheckbox = new JCheckBox("Add/Remove Member");
        memberCheckbox.addItemListener(new ItemListener() {
            @Override
            public void itemStateChanged(ItemEvent e) {
                if (!isPending) {
                    isPending = true;
                    memberCheckbox.setEnabled(false); // Disable checkbox
                    System.out.println("Checkbox clicked. Sending request to
server...");

                    // Simulate server request with a background thread
                    new Thread(() -> {
                        try {
                            Thread.sleep(3000); // Simulate network delay
                            System.out.println("Server response received. Updating
state...");
                        } catch (InterruptedException ex) {
                            ex.printStackTrace();
                        } finally {
                            // Re-enable checkbox on the Event Dispatch Thread
                            SwingUtilities.invokeLater(() -> {
                                isPending = false;
                                memberCheckbox.setEnabled(true);
                                System.out.println("Checkbox re-enabled.");
                            });
                        }
                    }).start();
                } else {
                    System.out.println("Request pending. Please wait...");
                }
            }
        });

        frame.add(memberCheckbox);
        frame.setVisible(true);
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> new CheckboxDisableExample());
    }
}
```

Code 3: Checkbox simulation.

This Code 3 represents the following order:

1. User clicks the checkbox.
2. Checkbox is immediately disabled.
3. An artificial server request, along with a latency, is run in a background thread.
4. After the delay, which represents the server response time, the checkbox becomes available for further user interaction.

In short, the implementation in the members application requires a well-defined sequence of events covering user interactions, user interface functionality, network communication, and model updates. Disabling the checkbox on receiving user interactions and then re-enabling it after receiving a message from the server is part of making the application reliable, consistent, and easy to use, as seen in both course lectures and the previous Java example.

2.3 Sub-question 3

Q: The solution in 2.2 should still be implemented within the MVC pattern. Discuss the changes you need to make for the model, view, and controller. (Note that you don't need to implement it but need to discuss what needs to be added/modified)

Solution

Implementing a solution to deactivate checkboxes as part of the members application's Model-View-Controller architecture requires meticulous alterations to each aspect of the structure to maintain the integrity of the respective responsibilities while keeping the application robust and maintainable. The following discussion identifies the alterations to each member of the MVC trio in the context of the interactions of users and the resulting system responses.

When the user interacts with the application—more specifically, when a checkbox is selected or deselected to add or remove a member from a group—processing is triggered within the Controller. The Controller acts as the liaison between the user interface and the data behind it, immediately realizing that a potentially long-running operation on the server is in progress. To prevent duplicated or conflicting actions, it is necessary for the Controller to record that the particular operation is in a pending state. This is done by updating the Model, which means adding an entry to a collection (like a Set or Map) that tracks all outstanding group-member operations still pending validation from the server.

At this point, the Model takes on a broader role. In the past, the Model has been used to hold core business data—namely, to identify members of some groups.

Now, the Model also keeps up to date with current activities, including pendingOperations. As such, the Model becomes the one definitive point of reference, including both the application's data and the status of the user interface to backend operations. The Model, for example, would implement a method called isPending(String key) to indicate a checkbox should be temporarily disabled.

In parallel, the View focuses upon the presentation aspect. The View requests data from the Model, often by way of the Controller, in an effort to determine the prevailing status of each checkbox. If the Model indicates that action regarding a member and a given group is in progress, the corresponding checkbox is disabled in the View. This measure ensures that the user is given immediate visual acknowledgment that their action is in progress, while also preventing additional manipulation of that checkbox prior to the operation's completion.

The story continues with the sending of a RESTful request from the Controller to the backend server. All such operations are normally done in an asynchronous mode using a background thread in Java, thus avoiding any possible unresponsiveness in the UI. While the request is executed, the Model's pending state maintains the synchronization of the View, keeping the respective checkbox disabled in the meantime.

After the server response—whether it reports success or failure—the Controller steps in again. It updates the Model by removing the completed operation from the list of pending operations. This change triggers the View to render again, thus reenabling the checkbox and reflecting any changes to the group-member relations that may have resulted from the server response.

To illustrate, let us consider the following Java code snippets that capture this story:

Model:

```
public class MembersModel {
    private final Set<String> pendingOperations = new HashSet<>();

    public synchronized void addPendingOperation(String key) {
        pendingOperations.add(key);
    }

    public synchronized void removePendingOperation(String key) {
        pendingOperations.remove(key);
    }

    public synchronized boolean isPending(String key) {
        return pendingOperations.contains(key);
    }
}
```

Code 4: Model.

View:

```
JCheckBox checkbox = new JCheckBox();
checkbox.setSelected(isMemberInGroup(memberName, groupName));
checkbox.setEnabled(!model.isPending(memberName + "-" + groupName));
```

Code 5: View.

Controller:

```
public void onCheckboxClicked(String memberName, String groupName) {
    String key = memberName + "-" + groupName;
    model.addPendingOperation(key);
    view.update();

    new Thread(() -> {
        try {
            // send RESTful request to backend
        } finally {
            model.removePendingOperation(key);
            view.update();
        }
    }).start();
}
```

Code 6: Controller.

This development in the story ensures that each component of the MVC architecture serves its intended purpose:

1. The Model manages both application data and UI-related state.
2. The View accurately reflects the current state of the Model, providing users with quick and accurate feedback.
3. The Controller oversees activity and information allocation, ensuring consistency and flexibility throughout the whole application.

Using this approach, the application is in a consistent state and resistant to impatient users or network delays yet still adheres to Model-View-Controller concepts.

3 Question 3

Q: In our course project, we are utilizing Gradle version 6.9.3 and Spring Boot version 1.5 whereas the up-to-date version of Gradle is version 8.7, and Spring Boot is 3.2.4. Identify the key necessary steps required to perform this upgrade by identifying the key difference between the versions, and what changes you need to make in the 'build.gradle'. Your answer does not need to include any implementation. *(15 points)*

Solution

The upgrade of a Java Spring Boot project from Gradle 6.9.3 and Spring Boot 1.5 to the latest stable versions, that is, Gradle 8.7 and Spring Boot 3.2.4, is a complex process that involves careful planning, understanding of fundamental differences, and corresponding modifications of the build settings. This discussion outlines the key phases of this process, clarifies the underlying reasoning for each phase, provides detailed Java-focused examples with a summary table for the clarity of the process.

Major differences among versions and rationales for modifications

1. Gradle 6.9.3 → 8.7:

The newer Gradle versions introduce stricter deprecation policies, improved task configuration, updated plugin APIs, and require newer Java versions (Java 17+ for Spring Boot 3.x compatibility).

2. Spring Boot 1.5 → 3.2.4:

Spring Boot 3.x requires Java 17 or later, includes improvements from Spring Framework 6, and moves to the Jakarta EE 9 namespace (using jakarta.* instead of javax.*). Many starters, configuration properties, and auto-configuration features have been removed or replaced, and dependency management practices have been updated.

Progressive Clarification of the Improvement Process

1. Assessment and Readiness
   - Start the process by checking the compatibility of project dependencies and plugins with the latest versions.
   - Use Gradle's deprecation scanning tool (./gradlew help --scan) to identify deprecated APIs or plugins.

2. Improve the Gradle wrapper
   - The Gradle wrapper ensures that all developers use the same version of Gradle.
   - Modify the wrapper found in your project directory.

```
./gradlew wrapper --gradle-version 8.7
```

   - In gradle/wrapper/gradle-wrapper.properties, verify the distribution URL:

```
distributionUrl=https\://services.gradle.org/distributions/gradle-8.7-
bin.zip
```

   - Ensure your development environment uses Java 17 or newer, as required by Spring Boot 3.x.

3. Re-evaluate spring boot and related dependencies

- In the build.gradle file, update the Spring Boot plugin and dependency versions:

```
plugins {
    id 'org.springframework.boot' version '3.2.4'
    id 'io.spring.dependency-management' version '1.1.4'
    id 'java'
}
```

- Update all Spring-related dependencies to versions compatible with Spring Boot 3.2.4.
- Replace all previous javax.* imports in your Java code with their equivalent jakarta.* counterparts so that your application is Jakarta EE 9 compliant.

4. Modify build.gradle Configuration
   - Set Java Version Compatibility

Gradle 8.x and Spring Boot 3.x require Java 17+:

```
java {
    sourceCompatibility = JavaVersion.VERSION_17
    targetCompatibility = JavaVersion.VERSION_17
}
```

- Define the Primary Class.

The process for determining the main application category has been updated:

```
springBoot {
    mainClass = 'com.example.Application'
}
```

- Reevaluate Dependency Claims

Make sure all the dependencies match the new version of Spring Boot. For example:

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
    testImplementation     'org.springframework.boot:spring-boot-starter-
test'
    // Other dependencies...
}
```

- Remove Deprecated or Incompatible Plugins/Settings

Remove or modify plugins and configuration settings that are not compatible with Gradle version 8.x anymore. An example of these is the deprecated build cache configurations or customized bootJar configurations.

5. Evaluate and Differentiate

After making these changes, run your build process and test.

```
./gradlew build
```

- Fix any errors related to dependency conflicts, deprecated application programming interfaces, or plugin incompatibilities.
- Refer to the [Spring Boot 3.0 Migration Guide] and the [Gradle 8.x Upgrade Guide] for troubleshooting help

Summary Table: Main Changes in build.gradle

Area	Old (Gradle 6.9.3, Spring Boot 1.5)	New (Gradle 8.7, Spring Boot 3.2.4)
Plugin Version	org.springframework.boot' version '1.5x'	org.springframework.boot' version '3.2.4'
Java Version	sourceCompatibility = 1.8	sourceCompatibility = 17
Main Class Config	Custom or Implicit	springBoot { mainClass = '…'}
Dependency Versions	Old, possibly javax.*	Updated, use jakarta.* where required
Deprecated APIs	Allowed	Must be removed/replaced
Build Cache Settings	removeUnusedEntriesAfterDays	Use new cache retention configuration

Table 2: Gradle version differences.

Java Illustration: Main application class

The principal with the newest versions of Spring Boot and Gradle, that first application class must be kept in compliance, but the imports must be in the newly required Jakarta namespace.

```java
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Code 7: Java's class for Main application (example).

In summary, upgrading from Gradle 6.9.3 and Spring Boot 1.5 to Gradle 8.7 and Spring Boot 3.2.4 is a meticulous process that encompasses build tool considerations, dependency management, Java versioning specifications, and source code alignment. Systematically updating the Gradle wrapper, adjusting the build.gradle

settings, and verifying that all Java source files and dependencies meet the updated requirements can improve the project's ability to take advantage of the better performance, increased security, and better maintainability offered by the latest frameworks.