Repository Link: https://github.com/Shahriar-0/Software-Testing-Course-Projects-S2025
Latest Commit Hash: 7b5f84dea9a09cc0df81a87223a96ba9ada704e9

Project 1

1. `Assume` Methods and `assumeTrue`

In software testing, particularly with JUnit, the **Assume** methods, including **assumeTrue**, are used to manage test execution based on certain preconditions. These methods allow tests to be skipped rather than failing outright when certain assumptions are not met.

The `assumeTrue` method is designed to validate a given condition and proceed with the test only if that condition is true. If the condition evaluates to false, the test is aborted, and it is not marked as a failure; instead, it is skipped. This is particularly useful in scenarios where a test may not be applicable under certain conditions, such as specific environment settings or configurations.

Method Signatures

The `assumeTrue` method has several overloaded versions:

```
public static void assumeTrue(boolean assumption) throws TestAbortedException
```

```
public static void assumeTrue(boolean assumption, Supplier<String> messageSupplier) throws TestAbortedException
```

```
public static void assumeTrue(boolean assumption, String message) throws TestAbortedException
```

```
public static void assumeTrue(BooleanSupplier assumptionSupplier) throws TestAbortedException
```

```
public static void assumeTrue(BooleanSupplier assumptionSupplier, String message) throws TestAbortedException
```

```
public static void assumeTrue(BooleanSupplier assumptionSupplier, Supplier<String> messageSupplier) throws TestAbortedException
```

These variations allow for flexibility in how assumptions are checked and how messages are reported if an assumption fails.

Practical Usage

Here's how wemight typically use `assumeTrue` in a test:

```
@Test
void testOnDev() {
    System.setProperty("ENV", "DEV");
    Assumptions.assumeTrue("DEV".equals(System.getProperty("ENV")));
    // Remainder of the test will proceed if the assumption is true
}
```

If the environment variable does not match "DEV", the execution of this test will be aborted without marking it as a failure.

Differences Between Assertions and Assumptions

There is a difference to **assertions** and **assumptions**:

a. **Assertions** (e.g., `assertEquals`, `assertTrue`) are used to validate conditions that must hold true for the test to be considered successful. If an assertion fails, the test fails.

b. **Assumptions** (like `assumeTrue`) allow tests to skip execution when certain conditions are not met. The test does not fail; it simply does not run further if the assumption is false.

Citations:

- https://www.lambdatest.com/automation-testing-advisor/selenium/methods/org.junit.Assume.assumeTrue
- https://howtodoinjava.com/junit5/junit-5-assumptions-examples/
- https://www.javaguides.net/2021/12/junit-assumptions-assumetrue-example.html
- https://stackoverflow.com/questions/44628483/assume-vs-assert-in-junit-tests/44628970

2. Using Unit Tests to Ensure Correctness of Multi-Threaded Code

Unit testing multi-threaded code presents unique challenges due to the non-deterministic nature of concurrent execution. **We can't use (unit) testing to *prove* our code thread safe but that doesn't mean we can't test such code at all.**

Some challenges that hinder our testing is:

a. **Non-Determinism**: The primary challenge in testing multi-threaded code is that the order of execution can vary, leading to different outcomes even with the same input. This can cause race conditions, deadlocks, and other synchronization issues.

b. **Race Conditions**: These occur when multiple threads access shared resources simultaneously, leading to unpredictable results. For instance, if two threads attempt to add reviews at the same time, one may overwrite the other's changes.

c. **Deadlocks**: This happens when two or more threads are waiting indefinitely for resources held by each other, causing the application to hang.

Solutions:

To effectively unit test multi-threaded code like `ReviewService`, we can consider the following strategies: (although none of them would solve the problem for us)

- **Isolation of Concurrency Logic**: Separate the business logic from concurrency management. This allows us to test the core functionality without involving threads directly. For example, we could test `addReview` in isolation by mocking dependencies like `Database` and `UserService`.
- **Use of Synchronization Primitives**: Tools like `CountDownLatch`, `Semaphore`, or `CyclicBarrier` can help manage thread execution flow during tests. For instance, we can use a `CountDownLatch` to ensure that all threads reach a certain point before proceeding.
- **Stress Testing**: Run tests under high concurrency conditions to reveal potential issues that may not appear under normal loads. This involves executing multiple threads that perform operations on shared resources simultaneously.
- **Timeouts and Retries**: Implement timeouts for tests to avoid hanging indefinitely due to deadlocks or other blocking operations. This ensures that tests fail gracefully rather than causing prolonged waits.

- **Deterministic Testing Tools**: Consider using libraries like `vmlens` or `jcstress`, which help make multi-threaded tests deterministic by exploring all possible thread interleavings during test execution. This can uncover subtle concurrency bugs that might otherwise go unnoticed.

Citations:

- https://www.reddit.com/r/java/comments/jpde37/examples_of_good_github_repos_illustrating/
- https://www.qodo.ai/question/how-should-i-unit-test-multithreaded-code/
- https://www.goatly.net/post/writing-multi-threaded-unit-tests/
- https://dzone.com/articles/a-new-way-to-junit-test-your-multithreaded-java-co
- https://www.javacodegeeks.com/2012/09/5-tips-for-unit-testing-threaded-code.html
- https://www.lambdatest.com/automation-testing-advisor/selenium/methods/org.junit.Assume.assumeTrue

3. Using Printing for Testing

Using the console for printing output in unit tests, as shown in the provided code example, has several drawbacks when compared to using assertions. Below are the key issues associated with this practice.

Lack of Automation in Test Results

- **Manual Verification**: When using `System.out.println`, developers must manually check the console output to verify test results. This manual process can lead to human error, as it's easy to overlook discrepancies between expected and actual values, and besides it would be really tedious and time-consuming since it’s not automatic.
- **Non-Deterministic Outputs**: Console outputs do not provide a clear pass/fail indication. If the output does not match expectations, the test does not fail automatically, which can lead to false confidence in the code's correctness.

Poor Integration with Testing Frameworks

- **No Integration with Test Reports**: Most testing frameworks (like JUnit) generate reports based on assertions. Console outputs are not captured in these reports, making it difficult to track test results over time or in continuous integration environments.

- **Incompatibility with Test Runners**: Many IDEs and CI/CD tools expect tests to use assertions for reporting results. Tests that rely on console output may not be recognized as valid tests, causing them to be ignored or misreported.

Difficulty in Debugging

- **Limited Contextual Information**: Console output lacks the structured context that assertions provide. Assertions typically include detailed error messages that specify what was expected versus what was received, aiding in debugging.
- **No Stack Trace on Failure**: When an assertion fails, it often provides a stack trace that helps identify where the failure occurred. Console outputs do not offer this information, making it harder to diagnose issues quickly.

Code Example Comparison

Here's a comparison of the original console-based test and a proper assertion-based test:

Console Output Test

```
@Test
public void testA() {
    Integer result = new SomeClass().firstMethod();
      System.out.println("Expected result is 10. Actual result is " +
result);
}
```

Assertion-Based Test

```
@Test
public void testA() {
    Integer result = new SomeClass().firstMethod();
    assertEquals("Expected result is 10", Integer.valueOf(10), result);
}
```

In the assertion-based test:

The use of `assertEquals` automatically checks if `result` matches the expected value (10). If the assertion fails, it provides a clear message indicating what was expected and what was actually received, along with a stack trace for easier debugging.

Citations:

- https://stackoverflow.com/questions/11209639/can-i-write-into-the-console-in-a-unit-test-if-yes-why-doesnt-the-console-win
- https://www.danielhall.io/towards-better-unit-testing

4. Problems and Solutions for the Provided Tests

TestB

Problem:

In the `testB` method, the expectation (which is btw not a valid java syntax) is that an exception will be thrown when `AnotherClass.process(badInput)` is called with `badInput` set to 0. However, there is no verification in place to ensure that `AnotherClass.process(int)` actually throws an exception for this input. This could lead to false positives if the method does not behave as expected.

Solution:

Verifying that `AnotherClass.process(badInput)` is designed to throw an exception when `badInput` is 0. We can do that by using `assertThrows`.

TestCalculator

Problem:

In the `TestCalculator` class, the `fixture` instance of `Calculator` is shared across multiple test methods. If the state of `fixture` is not reset before each test, it can lead to incorrect results due to state leakage from previous tests. For example, if `testAccumulate()` runs before `testSubsequentAccumulate()`, the initial value may not be what is expected.

Solution:

To address this issue, we can implement a setup method that resets the state of `fixture` before each test runs. We can achieve this by:

Using the `@BeforeEach` annotation (in JUnit 5) or a similar setup method in our testing framework to ensure that `fixture` is initialized or reset to a known state before each test case is executed. This will help maintain test isolation and reliability.