Software Testing Report

11/03/2025

Introduction

Software testing is a vital part of code development as it prevents errors, ensures security measures are implemented correctly, helps identify issues etc. Software testing is essential for quality and reliability in code. In my report I explore key concepts such as data-driven testing, testability, preconditions and postconditions. Through my assigned lab worksheet I apply these ideas to real scenarios to identify potential pitfalls and improve test coverage.

Data-Driven Testing

Parameterized Tests

Verifying the Test Cases

```
@Test
public void testAdd() {
Calculator c = new Calculator(3, 4);
int result = c.add();
assertEquals(result, 7, "result should be 7");
}
//Adding zero to a number should always * give us the same number
back

@ParameterizedTest
@ValueSource( ints = { -99, -1, 0, 1, 2, 101, 337 })
void addZeroHasNoEffect(int num) {
      Calculator c = new Calculator(num, 0); int result = c.add();
      int result = c.add();
      assertEquals(result, num, "result should be same as num");
}
```

A. Run the tests in your IDE or editor to confirm that all those ints are used – how can you tell?

After running the above code from addZeroHasNoEffect to see if all of the ints in @ValueSource are being used, I can see in the test terminal I have all ticks and when hovering other I can see the numbers displayed as expected (-99, -1, 0, 1, 2 101, 337)

b. Try changing them and/or adding to the list.

```
@ParameterizedTest
@ValueSource(ints = { -99, -1, 0, 1, 2, 101, 337, 500, 3, -11 })
void addZeroHasNoEffect(int num) {
      Calculator c = new Calculator(num, 0);
      int result = c.add();
      assertEquals(num, result, "result should be same as num");
```

c. addZeroHasNoEffect is a single method. But (based on the material from lectures and the textbooks) is it also a single test case? If not, how many test cases does it comprise?

The addZeroHasNoEffect(int num) method uses the @ParameterizedTest and @ValueSource which allows multiple integer values to be tested automatically. As each integer is tested separately, that means seven test cases are executed. In IntelliJ IDE you can confirm this through the test report, which logs individual results for each input. Since each integer is tested separately, seven test cases are executed. In IntelliJ, this can be confirmed through the test report, which logs individual results for each input.

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.MethodSource;
import org.junit.jupiter.params.provider.Arguments;
import java.util.stream.Stream;

public class CalculatorTest {
@Test
public void testAdd() {
      Calculator c = new Calculator(3, 4);
      int result = c.add();
      assertEquals(7, result, "result should be 7");
}

//Adding zero to a number should always give us the same number
back.
//This test is parameterized, so it runs once for each int
provided.

@ParameterizedTest
@MethodSource("additionTestCasesProvider")

//Test Calculator.add() with multiple test values and expected
values.
```

```
* @param num1 First test value to be passed to Calculator.add()
* @param num1 Second test value to be passed to Calculator.add()
* @param expectedResult The expected result for Calculator.add()

void tableOfTests(int num1, int num2, int expectedResult) {
      Calculator c = new Calculator(num1, num2);
      int result = c.add();
      assertEquals(expectedResult, result, "result should be same
as as expected result");
}

static Stream<Arguments> additionTestCasesProvider() {
      return Stream.of(

//the arguments are:
//num1, num2, and expected result.

Arguments.arguments(1, 2, 3),
Arguments.arguments(3, 7, 10),
Arguments.arguments(3, 7, 11),
Arguments.arguments(99, 1, 100)
);
}
}
```

a. How many test cases would you say tableOfTests and additionTestCasesProvider comprise?

The tableOfTests and additionTestCasesProvider methods comprise 4 test cases. Both call to Arguements.arguements(num1, num2, expected result), as there are 4 arguments, 4 individual tests will run.

b. Read through the JUnit documentation on writing parameterized tests. (You might find the baeldung.com "Guide to JUnit 5 Parameterized Tests" by Ali Dehghani helpful as well.) If you have time, try experimenting with the different features it describes.

After reading through Junit documentation and the Baeldung guide I have found a variety of features that make parameterized tests versatile. Here are some of the features I experimented with for this report:

1. @ValueScore it you to pass simple values to your parameterized tests. You can use this when testing with one argument.

2. @CsvSource you can pass multiple arguments in a CSV format. This is useful when you need to pass several parameters in a single test case.

3. @EnumSource it allows you to run a test for each value of a enum type.

4. @MethodSource As shown in a previous example, this is used to provide more complex or customized test data. This allows more flexibility in the types of arguments that can be passed.

c. In Java, enum types are used to represent types that can take on values from only a distinct set. By convention, the values are given names in ALL CAPS with underscores to separate words (also called "SCREAMING_SNAKE_CASE").

For instance,

```
public enum Weekday {
MON, TUE, WED, THU, FRI, SAT, SUN
}
```

or

```
public enum Color {
RED, ORANGE, YELLOW, BLUE, GREEN, INDIGO, VIOLET
}
```

Suppose we need to run a test which should be passed each Weekday in turn. Which JUnit annotation should we use for this?

The JUnit annotation that should be used in this instance is @EnumSource, This annotation provides the values from an enum to the test method.

d. If you have used testing frameworks in languages other than Java – how do they compare with JUnit? Do they offer facilities for creating data-driven tests? Are these more or less convenient than the way things are done with JUnit?

Other testing frameworks in languages also provide facilities for creating data-driven tests, but the syntax and approach will likely be different. Junits tests are typically more powerful when dealing with large applications and complex testing scenarios, however Pythons pytest is preferred by some for smaller projects as it is considered more succinct and flexible, with less boilerplate code than JUnit .

Preconditions and Postconditions

Scenario: Removing a Unit from a Database

A student database includes tables for students, course units, and enrollments. When a course unit is deleted, all associated enrollments must also be removed. The existing method:

```
//remove a unit from the system
void removeUnit(String unitCode) {
       units.removeRecord(unitCode);
}
```

a. What preconditions do you think there should be for calling removeUnit()?

1. Unit must exist.he unit code must correspond to an existing course unit in the database

2. Student enrollments. If the design requires that all enrollments for the unit be removed when the unit is deleted, the system should check that the unit has associated enrollments, as they must be deleted as well

3. Database connection. There should be an active connection to the database because if there is no database connection, the removal operation won't work

4. Permission check. The user calling the removeUnit method must have permissions to delete units from the system.

b. What postconditions should hold after it is called?

1. **Unit is removed from the database**. The unit with the given unitCode should be removed from the table or data structure and there should be no record of the unit remaining in the system

2. **Associated enrollments are removed**. Any records in the enrollments table that are associated with the removed unit should also be deleted

3. **Database consistency**. After the removal, the database should remain consistent

4. **No references to the deleted unit**. No other objects or systems should reference the removed unit. If the unit is being referenced anywhere else in the system, it should be handled or updated to show that the unit no longer exists

c. Does the scenario give rise to any system invariants?

Yes the following invariants are likely:

1. **Unit consistency**

2. **Enrollments consistency**

d. Can you identify any problems with the code? Describe what defects, failures and erroneous states might exist as a consequence.

Yes, there are some potential issues with the code and the scenario:

1. **No deletion of related enrollments**: The current method only removes the unit itself, but it does not account for the related enrollments in the enrollments table

2. **No error handling**: The code does not check if the unit exists before attempting to remove it

3. **Lack of atomicity**: If the removal of the unit and enrollments are not handled atomically there could be an issue where the unit is deleted, but the associated enrollments are not, or vice versa

4. **Permission and access checks**: The code doesn't include permission checks for who can call the removeUnit method. Meaning unauthorised users might be able to delete units from the system.

Testability Analysis

Testing Usability in a Flight Booking System

a. The flight booking system should be easy for travel agents to use.

Yes the booking system should be easy for travel agents to use. To ensure the booking system is easy to use its important to carry out analysis such as:

Usability testing - This can be used to determine how easy travel agents find the booking system whilst carrying out their usual tasks in real world scenarios. Data such as time to complete a task, error rates and user satisfaction can help evaluate usability.

Task completion rates - One way to test this is by determining if travel agents can complete their tasks efficiently and as fast if not faster than on previous software

Accessibility - Testing the system's accessibility for users with disabilities

b. The int String.indexOf(char ch) method should return a -1 if ch does not appear in the receiver string, or the index at which it appears, if it does

To test this requirement, you would write test cases that cover a variety of conditions such as:

- When the character appears at the start of the string
- When the character appears at the end of the string
- When the character appears multiple times
- When the character does not appear in the string

c. Internet-aware Toast-O-Matic toasters should have a mean time between failure of 6 months.

Testing should be carried out regarding the lifespan and reliability of the toaster under normal usage conditions.

Failure Rate Measurement: To test this, you need to track the failures of the toaster over time under normal usage and observe how often it fails. The goal is to measure whether the toaster fails on average every 6 months.

Simulation of Failure Modes: If the toaster's failures are not easily observed in norma; use, simulate stress testing or simulate failure scenarios

Conclusion

I examined data-driven testing in JUnit and explored preconditions, postconditions, and testability challenges. These exercises highlight the importance of structured test cases, database integrity and understanding testing limitations. By applying these principles, software systems can be made more reliable and resilient.