UNIT IV – TESTING AND QUALITY IN SOFTWARE [9 hours]

Types of Testing: Unit, Integration, System, Automated Testing using JUnit or PyTest, Introduction to Selenium (UI testing basics), Code Quality Tools: SonarLint/SonarQube basics , Importance of Testing in Real Projects

---

AUTOMATED TESTING

The most significant part of Software testing is Automation testing. It uses specific tools to automate manual design test cases without any human interference. Software testing is the process in which a developer ensures that the actual output of the software matches with the desired output by providing some test inputs to the software. Software testing is an important step because if performed properly, it can help the developer to find bugs in the software in very less amount of time. Software testing can be divided into two classes, Manual testing and Automated testing.

Automated testing is the execution of your tests using a script instead of a human. Automation testing is the best way to enhance the efficiency, productivity, and coverage of Software testing. It is used to re-run the test scenarios, which were executed manually, quickly, and repeatedly. In other words, we can say that whenever we are testing an application by using some tools, it is known as automation testing.

We will go for automation testing when various releases or several regression cycles go on the application or software. We cannot write the test script or perform the automation testing without understanding the programming language.

Advantages of Automated Testing:

Automated Testing has the following advantages:

1. Automated testing improves the coverage of testing as automated execution of test cases is faster than manual execution.
2. Automated testing reduces the dependability of testing on the availability of the test Engineers.
3. Automated testing provides round the clock coverage as automated tests can be run all time in 24*7 environment.
4. Automated testing takes far less resources in execution as compared to manual testing.
5. It helps to train the test engineers to increase their knowledge by producing a repository of different tests.
6. It helps in testing which is not possible without automation such as reliability testing, stress testing, load and performance testing.
7. It includes all other activities like selecting the right product build, generating the right test data and analyzing the results.

8. It acts as a test data generator and produces maximum test data to cover a large number of input and expected output for result comparison.
9. As with automated testing, test engineers have free time and can focus on other creative tasks.

Disadvantages of Automated Testing:

Automated Testing has the following disadvantages:

1. Automated testing is much more expensive than manual testing.
2. It also becomes inconvenient and burdensome to decide who would automate and who would train.
3. It is limited to some organisations as many organisations do not prefer test automation.
4. Automated testing would also require additionally trained and skilled people.
5. Automated testing only removes the mechanical execution of the testing process, but creation of test cases still requires testing professionals.

Automated Testing using JUnit:

JUnit is an open-source testing framework that plays a crucial role in Java development by allowing developers to write and run repeatable tests. It helps ensure that your code functions correctly by verifying that individual units of code behave as expected. Developed by Kent Beck and Erich Gamma, JUnit has become a standard in Java testing, part of the broader xUnit family of testing frameworks.

JUnit is a framework designed to facilitate unit testing in Java. It allows developers to write test cases for individual functionalities in their code and run these tests to check if the actual outcomes match the expected results. When tests fail, JUnit provides detailed feedback, making it easier for developers to debug their code.

Need of JUnit

- **Automation:** JUnit supports automated testing, allowing developers to quickly verify that recent changes haven't disrupted existing functionality. JUnit tests can be integrated into build tools like Maven or Gradle and CI/CD pipelines.
- **Early Bug Detection:** Unit tests help catch bugs early in the development process, reducing the likelihood of introducing new errors as code evolves.
- **Maintainability:** JUnit ensures that changes, bug fixes, or new features don't inadvertently break other parts of the system. This helps maintain the overall quality of the codebase.
- **Ease of Use:** With simple annotations and methods, JUnit is easy to set up and use, making it accessible even for developers who are new to testing.

Basic Concepts in JUnit

1. Test Case: A test case is a single unit test written to check a specific piece of code. In JUnit, a test case is typically a method within a class.

2. Annotations:

@Test: Marks a method as a test case.

@Before: Executes the code before each test method.

@After: Executes the code after each test method.

@BeforeClass and @AfterClass: Executed once before and after all tests in a class, used for setup and cleanup.

3. Assertions: JUnit provides several assertions like assertEquals(), assertTrue(), and assertNotNull() to verify that the expected results match the actual results.

Example 1:

Java Code:

```
public class Calculator
{
	public int add(int a, int b)
	{
		return a + b;
	}
}
```

Test Code:

```
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
public class CalculatorTest
{
	@Test
	public void testAddition()
	{
		Calculator calculator = new Calculator();
		int result = calculator.add(2, 3);
		Assertions.assertEquals(5, result);
	}
}
```

Example 2:

Java Code:

```
public class TemperatureConverter
{
	public double celsiusToFahrenheit(double celsius)
```

```
        {
                return (celsius * 9/5) + 32;
        }
        public double fahrenheitToCelsius(double fahrenheit)
        {
                return (fahrenheit - 32) * 5/9;
        }
}
```

Test Code:

```
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
public class TemperatureConverterTest
{

        @Test
        void testCelsiusToFahrenheit()
        {
                TemperatureConverter converter = new TemperatureConverter();
                double result = converter.celsiusToFahrenheit(0);
                Assertions.assertEquals(32, result);
        }
        @Test
        void testFahrenheitToCelsius()
        {
                TemperatureConverter converter = new TemperatureConverter();
                double result = converter.fahrenheitToCelsius(32);
                Assertions.assertEquals(0, result);
        }
}
```

Advantages of JUnit

- Automation: Automates tests, allowing for repeated execution, especially useful in CI/CD environments.
- Early Bug Detection: Helps identify bugs early, making debugging more manageable.
- Test-Driven Development (TDD): Supports TDD, encouraging developers to write tests before implementation.

- Simple Syntax: Easy-to-use annotations like @Test, @Before, and @After simplify writing and maintaining tests.
- Support for Various Tests: Suitable for unit, integration, and acceptance testing.
- Rich Assertion Library: Offers various assertions to validate test results.

Disadvantages of JUnit

- Java-Specific: Designed specifically for Java applications; not suitable for other programming languages.
- Limited to Unit Testing: Cannot test user interfaces or complex database interactions directly.
- Complex Testing Setup: Advanced scenarios may require additional libraries like Mockito for mocking.
- Time-Consuming for Legacy Code: Creating tests for large, untested codebases can be time-consuming.

Automated Testing using PyTest:

Python-based automation testing frameworks like pytest provide a structured and efficient way to write test cases, making test automation more manageable and scalable. pytest is the most popular testing framework for python. Using pytest you can test anything from basic python scripts to databases, APIs and UIs. Though pytest is mainly used for API testing, in this article we'll cover only the basics of pytest.

Installation: Install pytest from PyPI using the command,

pip install pytest

Use: The pytest test runner is called using the following command in project source,

py.test

pytest looks for test files in all the locations inside the project directory. Any file with a name starting with "test_" or ending with "_test" is considered a test file in the pytest terminology. Let's create a file "test_file1.py" in the folder "tests" as our test file. Creating test methods: pytest supports the test methods written in the unittest framework, but the pytest framework provides easier syntax to write tests. See the code below to understand the test method syntax of the pytest framework.

Example: area.py

```
import math
def rectangle_area(length: float, width: float) -> float:
    if length <= 0 or width <= 0:
        raise ValueError("Length and width must be positive numbers.")
    return length * width
def circle_area(radius: float) -> float:
    if radius <= 0:
```

```
            raise ValueError("Radius must be a positive number.")
    return math.pi * radius ** 2
def triangle_area(base: float, height: float) -> float:
    if base <= 0 or height <= 0:
            raise ValueError("Base and height must be positive numbers.")
    return 0.5 * base * height
```

Test code: test_area.py

```
import math
import pytest
from area import rectangle_area, circle_area, triangle_area
def test_rectangle_area():
    assert rectangle_area(5, 4) == 20
def test_rectangle_invalid():
    with pytest.raises(ValueError):
            rectangle_area(-5, 4)
# Circle Tests
def test_circle_area():
    assert circle_area(3) == pytest.approx(math.pi * 9)
def test_circle_invalid():
    with pytest.raises(ValueError):
            circle_area(0)
# Triangle Tests
def test_triangle_area():
    assert triangle_area(10, 5) == 25
def test_triangle_invalid():
    with pytest.raises(ValueError):
            triangle_area(-10, 5)
```

Output:

```
================== test session starts ==================
collected 6 items
test_area.py ..... [100%]
================== 6 passed ==================
```

Example 2: even_odd.py

```
def check_even_odd(number):
    if number % 2 == 0:
```

```
        return "Even"
    else:
        return "Odd"
```

Test code: test_even_odd.py

```
import pytest
from even_odd import check_even_odd
def test_even_number():
    assert check_even_odd(4) == "Even"
def test_odd_number():
    assert check_even_odd(7) == "Odd"
def test_zero():
    assert check_even_odd(0) == "Even"
```

Advantages of Pytest:

- **Extensions:** It can be made more functional by a large community of third-party plugins. This enables developers to alter their testing procedures to suit their requirements.
- **Robust Features:** Pytest boasts an extensive feature set that includes robust assertion introspection, parameterization, and fixtures. Developers can create flexible and manageable test code with these features.
- **Integration:** Pytest easily interfaces with Jenkins and Travis CI continuous integration tools, as well as with other testing frameworks like Unittest and Nose.

Disadvantages of Pytest

- **Learning Curve:** While Pytest is simple to use, developers who are new to the framework may find it difficult to keep up with its vast feature set. It could take some initial study to fully comprehend and utilize all of Pytest's features.
- **Effort Required for Migration:** It could take some work to convert test suites from other frameworks to Pytest, especially if the tests mostly rely on framework-specific features or standards.
- **Dependency on External Libraries:** Certain sophisticated capabilities, such as fixtures and parameterization, depend on external libraries. These libraries offer capabilities to Pytest, but they also introduce new dependencies that might need to be handled.
