Automation Testing

Creating Dynamic and Parameterized Tests

A Day in the Life of an Automation Test Engineer

Marcelo had done his tasks for default methods to add new methods, interfaces, and class implementations.

Now, Marcelo has to understand the Dynamic Tests, Parameterized Tests, creation of both tests, which makes tests more powerful and easier to maintain.

This lesson will help Marcelo to implement these modules for his project.

Learning Objectives

By the end of this lesson, you will be able to:

- Explain what are Dynamic and Parameterized Tests
- Illustrate how to create the tests with the help of examples
- Analyze the importance of both tests
- Identify the applications that comes under Dynamic and Parameterized Tests

What Are Dynamic Tests?

What Are Dynamic Tests?

- Dynamic testing is a new programming model introduced in JUnit 5.
- It is useful to create tests that cannot be defined at compile time (loaded via an external resource) or to create tests that cannot be expressed easily via @Parameterized Test.
- A Dynamic Test is a test generated during runtime.
- These tests are generated by a factory method annotated with the @TestFactory annotation.
- A @TestFactory method must return a Stream, Collection, Iterables, or Iterator of Dynamic Test instances.

What Are Dynamic Tests?

Dynamic Test via @TestFactory:

Execute @BeforeEach
Execute @TestFactory
Execute dynamic test 1
Execute dynamic test 2
Execute dynamic test 3
...
Execute @AfterEach

Creating a Dynamic Test

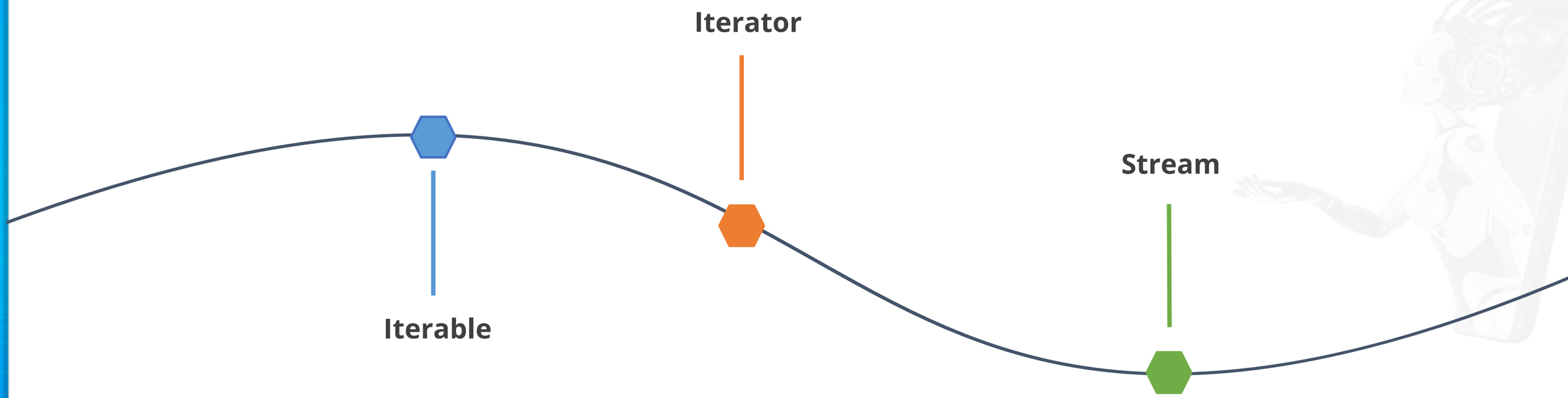
Creating a Dynamic Test

The @TestFactory method tells JUnit that this is a factory for creating Dynamic Tests. Here is an illustration of creating a Dynamic Test:

```
@TestFactory
Collection<DynamicTest> dynamicTestsWithCollection()
 {
    return Arrays.asList(
    DynamicTest.dynamicTest("Add test",
    () -> assertEquals(2, Math.addExact(1, 1))),
    DynamicTest.dynamicTest("Multiply Test",
    () -> assertEquals(4, Math.multiplyExact(2, 2))));
 };
```

Creating a Dynamic Test

The same test can be modified to return an Iterable, Iterator, or a Stream:



Creating a Dynamic Test

Here is an example of how to write a Dynamic Test with Iterable:

```
@@TestFactory
Iterable<DynamicTest> dynamicTestsWithIterable()
 {
    return Arrays.asList(
    DynamicTest.dynamicTest("Add test",
    () -> assertEquals(2, Math.addExact(1, 1))),
    DynamicTest.dynamicTest("Multiply Test",
    () -> assertEquals(4, Math.multiplyExact(2, 2))));
 };
```

Creating a Dynamic Test

Here is an example of how to write a Dynamic Test with Iteration:

```
@TestFactory
Iterator<DynamicTest> dynamicTestsWithIterator()
 {
    return Arrays.asList(
    DynamicTest.dynamicTest("Add test",
    () -> assertEquals(4, Math.addExact(2, 2))),
    DynamicTest.dynamicTest("Multiply Test",
    () -> assertEquals(7, Math.additionExact(4, 3))))
    .iterator();
 };
```

Creating a Dynamic Test

Here is an example of how to write Dynamic Test with a Stream:

```
@TestFactory
Stream<DynamicTest> dynamicTestsFromIntStream()
 {
    return IntStream.iterate(0, n -> n + 2).limit(10)
    .mapToObj(n -> DynamicTest.dynamicTest("test" + n,
    () -> assertTrue(n % 2 == 0)));
 };
```

What Are Parameterized Tests?

What Are Parameterized Tests?

- JUnit 5, the newest version of JUnit, makes building developer tests easier with the capabilities.
- Parameterized tests are one such feature. Users may use this capability to run a single test method several times with various parameters.
- Parameterized tests make it possible to run a test multiple times with different arguments.
- They are declared just like regular @Test methods but use the @ParameterizedTest annotation instead of the @Test annotation.

What Are Parameterized Tests?

Steps to create Parameterized Tests in Junit 5:

- Declare @ParameterizedTest to the test.
- Declare at least one source (example – @ValueSource) that will provide the arguments for each invocation of test.
- Consume the arguments in the test method.

Creating a Parameterized Test

Here is an example of how to write Parameterized Test:

```
public class Junit5_Parameterized_Test
 {

   This test will run sequentially 5 times with 1 argument each time
   @ParameterizedTest
   @ValueSource(int parameter ={8,4,2,6,10})
   void test_int_arrays(int arg)
  {
   System.out.println("arg => "+arg);
   assertTrue(arg % 2 == 0);
  }

 }
```

Creating a Parameterized Test

Here is an example of Parameterized Test in Maven Dependency:

```
<!-- Parameterized Tests -->
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-params</artifactId>
      <version>${junit.jupiter.version}</version>
      <scope>test</scope>
    </dependency>
```

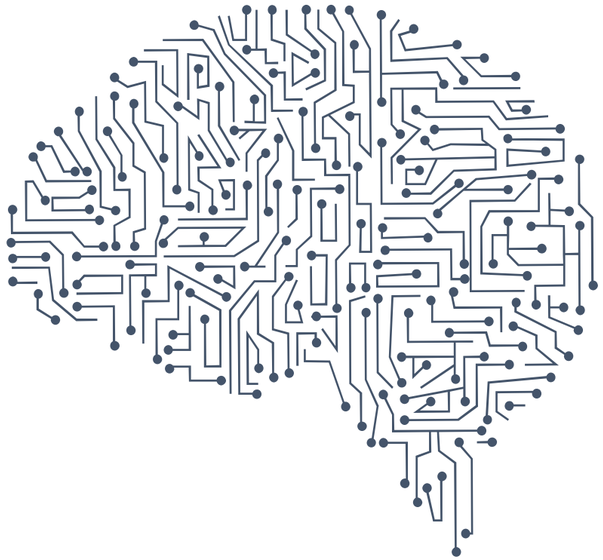
Creating a Parameterized Test

Here is an example of Parameterized Test with different arguments:

```
public class Junit5_ParameterizedTest_Arguments
{

  @ParameterizedTest
  @MethodSource("stringProvider")
  void testWith_MethodSource(String arg)
  {
    System.out.println("testWith_MethodSource(arg) => "+arg);
    assertNotNull(arg);
  }

};
```

Importance of Dynamic Tests

Importance of Dynamic Tests

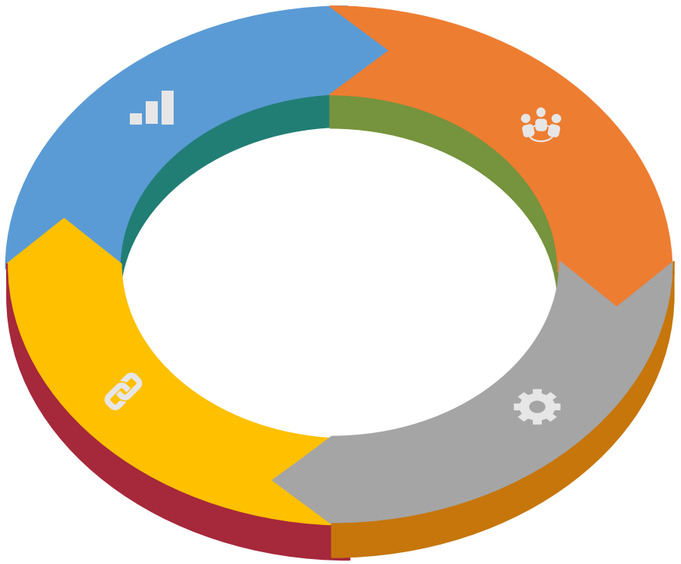


- It is an effective method for measuring the impact of various environmental stresses on the software product, such as hardware, networks, and more.
- Another main reason for implementing dynamic testing is that it helps the team find errors and defects in the software.
- It is executed to check the functional behavior of the software.
- Most importantly, it helps the team validate the overall performance of the software.

Levels of Dynamic Tests

Levels of Dynamic Tests

These are the levels of Dynamic Tests:



Unit Testing

Integration Testing

System Testing

Acceptance Testing

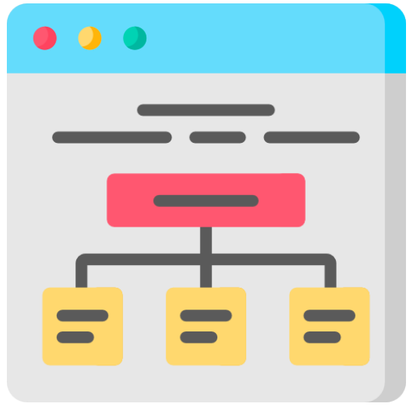
Unit Testing

The first level of testing is called unit testing. It is essential for the verification of the code produced by the individual programmers, and is typically done by the programmer of the module.

Unit Testing Techniques

- **Black Box Testing:** Used with the user interface, input and output are tested
- **White Box Testing:** Used to test each one of those functions behavior is tested
- **Gray Box Testing:** Used to execute tests, risks, and assessment methods

Integration Testing



Integration testing is the next level of dynamic testing. Several unit-tested modules are integrated into distinct subsystems, which are then tested. The goal here is to see if the modules can be properly integrated.

Example of Integration Testing

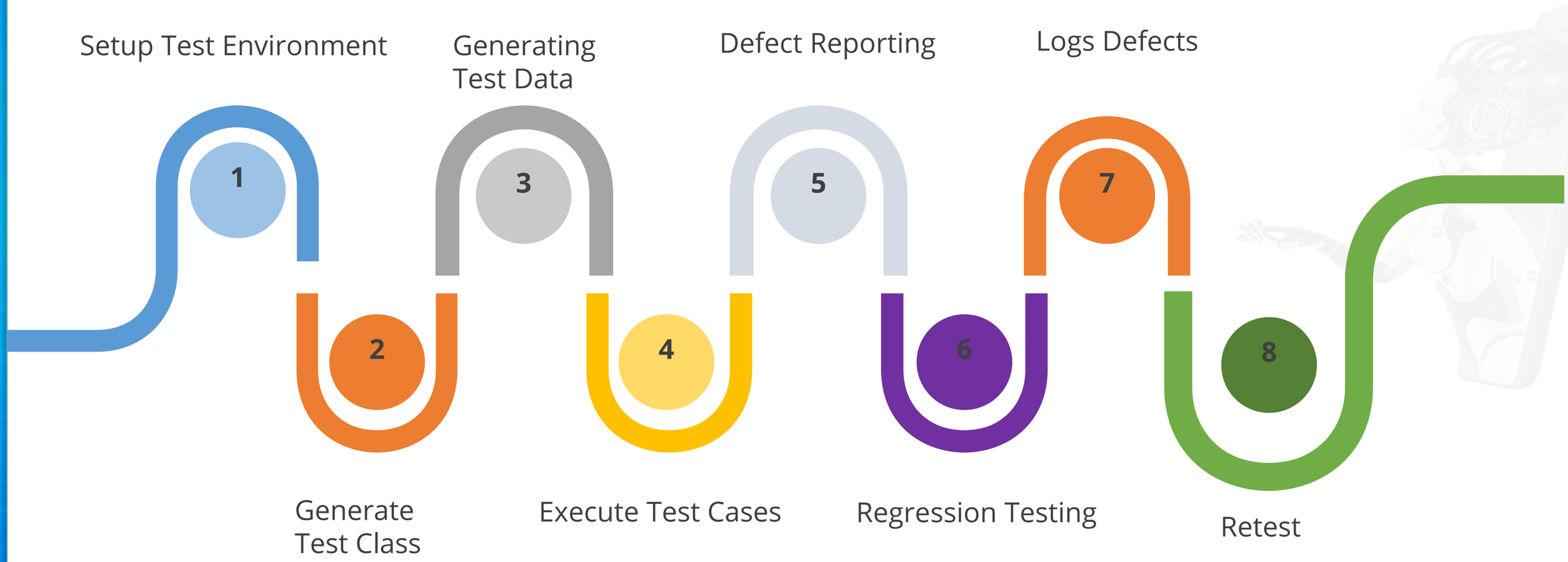
```
import static org.junit.Assert.assertEquals;
@Category(IntegrationTest.class)
public class IntegrationTestSampleTest
 {

    private JUnitAssertEqualsServiceExample junitAssertEqualsServiceSample;
    private ServiceObject serviceObject;
    @Before
    public void setData()
    {
      serviceObject = new ServiceObject();
      junitAssertEqualsServiceSample = new JUnitAssertEqualsServiceExample();
      junitAssertEqualsServiceSample.initiateMetaData(serviceObject);
    }
 }
```

System Testing

Here the entire software system is tested. The reference document for this process is the requirements document, and the goal is to see if the software meets its requirements.

Steps of System Testing

System Testing is performed by the following steps:



Acceptance Testing

Acceptance testing is often performed with realistic data of the client to demonstrate that the software is working satisfactorily. It can be performed as where as software is to eventually function.

Types of Acceptance Testing

- **Alpha and Beta Testing**
- **Regulation Acceptance Testing**
- **Contract Acceptance Testing**
- **Black Box Testing**

Importance of Parameterized Tests

Importance of Parameterized Tests

- Parameterized tests are an excellent approach to creating and running several test cases with only variation being the data.
- They are capable of validating code behavior for a wide range of values, including boundary situations.
- Parameterizing tests can improve code coverage and offer assurance, that the code is performing as planned.

Key Takeaways

- Dynamic Tests are used to create tests that cannot be defined at compile time (loaded via an external resource).
- Parameterized tests are one such feature. Users may use this capability to run a single test method several times with various parameters.
- Unit testing is essential for verification of the code produced by the individual programmers.
- Parameterizing tests can improve code coverage and offer assurance, that the code is performing as planned.