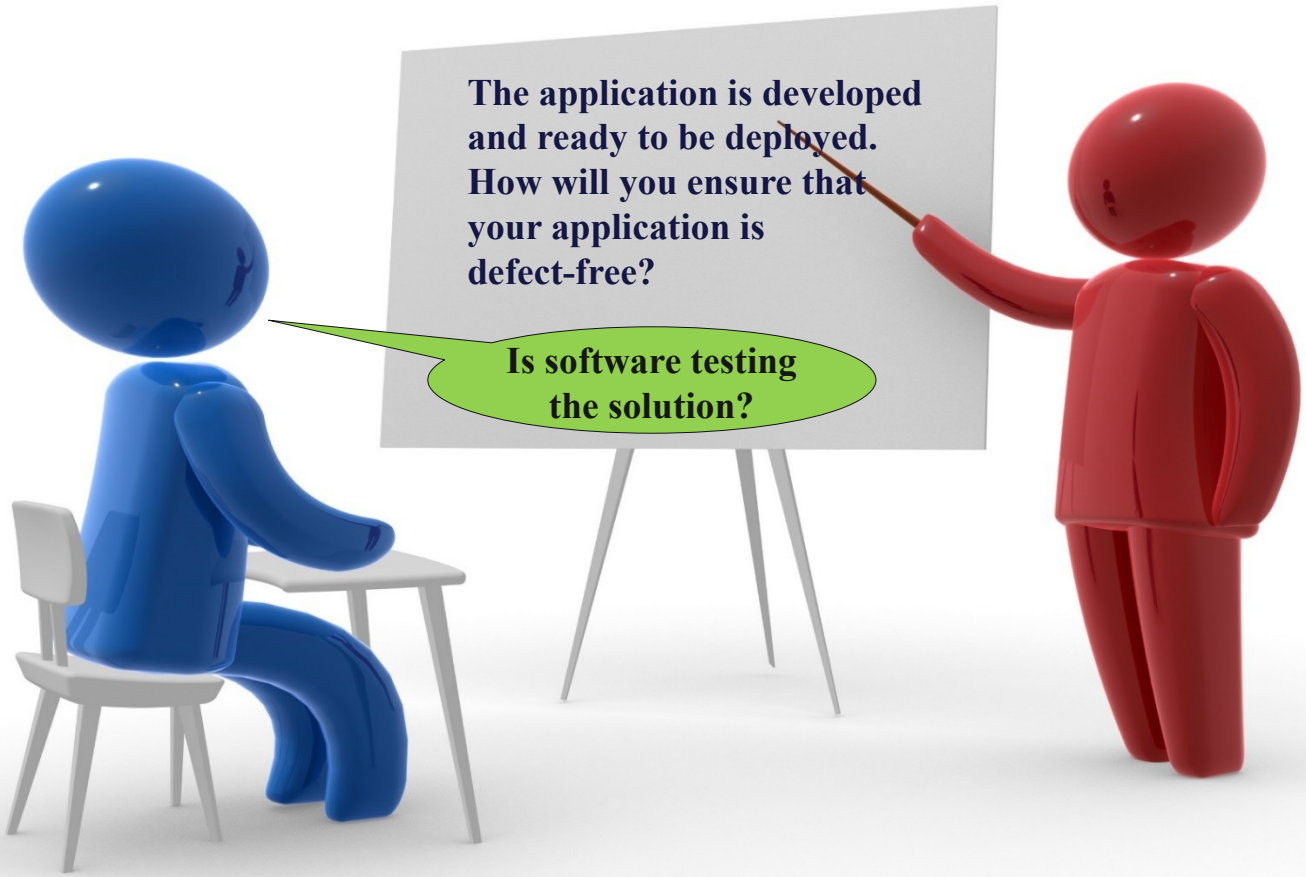
Testing JAVA Applications Using JUnit

Aesthetic: Dr. Tarkeshwar Barua

Introducing Software Testing



Aesthetic: Tarkeshwar Barua

Introducing Software Testing (Contd.)

Software testing plays a vital role in the development of defect-free software.

Software Testing Life Cycle (STLC)

- Is devised to test software comprehensively.
- Serves as a general guideline for testing an application and can vary according to the testing strategy being used.

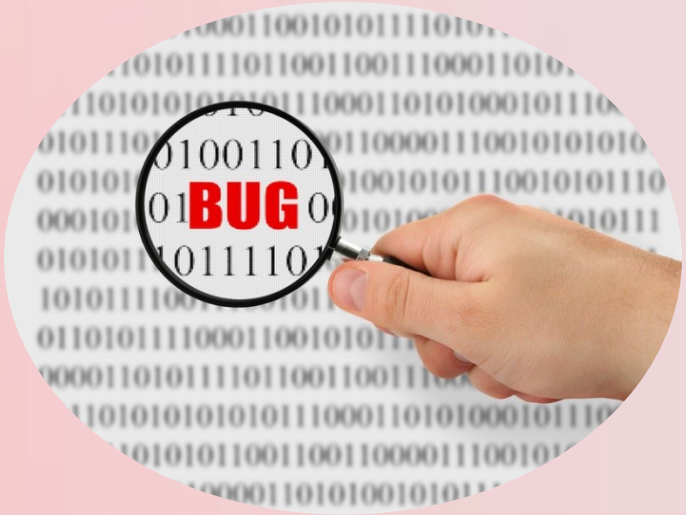
Testing strategy refers to the type of testing approach being used.

Software Testing Life Cycle

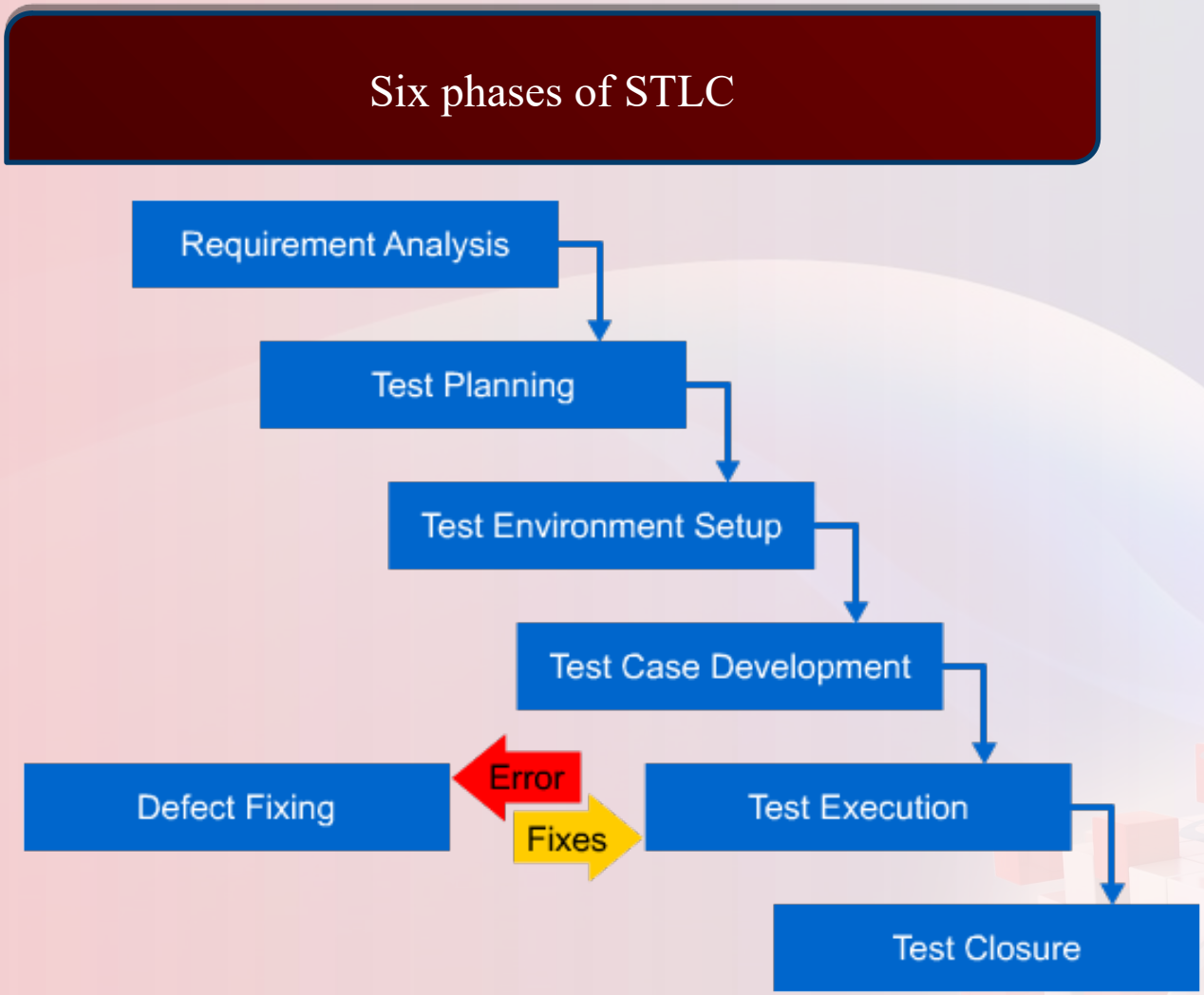
STLC is a systematic and progressive approach used for testing software to enhance its quality.

The software should be tested with the intent of finding every possible bug introduced in the code during the software development.

STLC helps in ensuring the delivery of quality software to users with the minimum risk of failure and errors.



Software Testing Life Cycle (Contd.)



Use the animation, STLC, to understand the concept of STLC.

Software Testing Life Cycle (Contd.)

Requirement analysis

- Is the initial step of the testing process.
- Focuses on understanding the requirement laid down in the requirement specification document.

Test planning

- Defines the objectives of the testing.
- Emphasizes on achieving an optimum balance between the specified requirements and the available resources.

Software Testing Life Cycle (Contd.)

Test environment setup

- Involves deciding the hardware and software requirements.
- Includes setting up the necessary tools and infrastructure required for convenient and smooth execution of the tests.

Test case development

- Emphasizes on developing the test.
- Involves writing step-by-step instructions for the test conditions that are identified in the initial stages of STLC.

- A Unit Test Case is a part of code, which ensures that another part of code (method) works as expected.
- To achieve the desired results quickly, a test framework is required. JUnit is a perfect unit test framework for Java programming language.

Software Testing Life Cycle (Contd.)

Test execution

- Involves executing the developed test cases under the given test environment to verify the test results.
- Has a small cycle that executes till the time all results are successful.

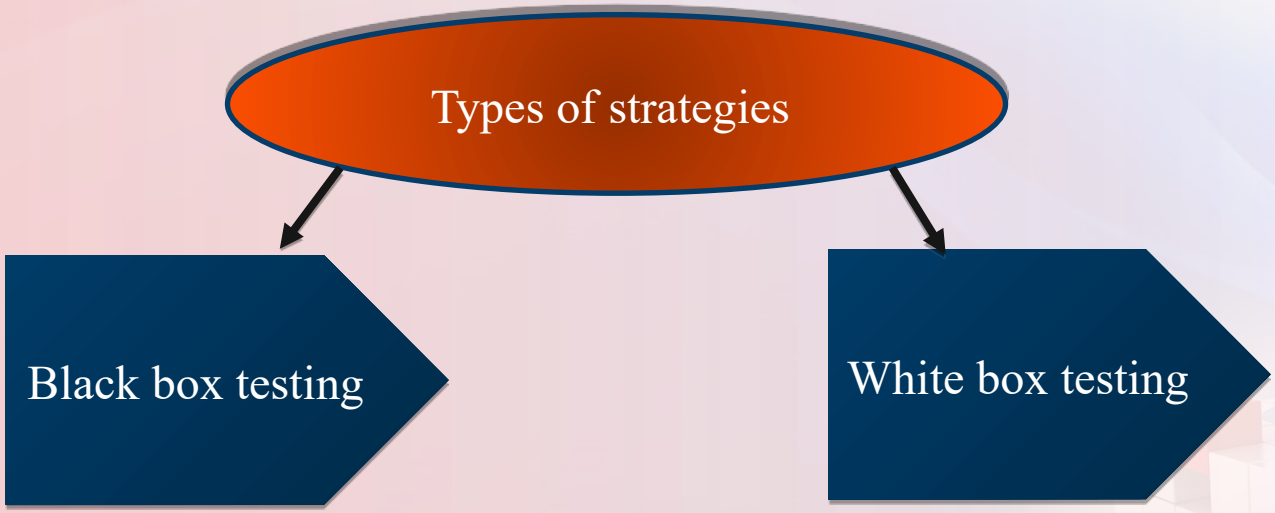
Test closure

- Includes matching the expected results of the test cases with the desired objectives.
- Involves generating test reports specifying the issues identified, issues rectified, test results summary, and test metrics.

Different Types of Testing

Software testing constitutes different types of testing strategies and levels.

Testing strategy is the approach followed to test software.



Different Types of Testing (Contd.)

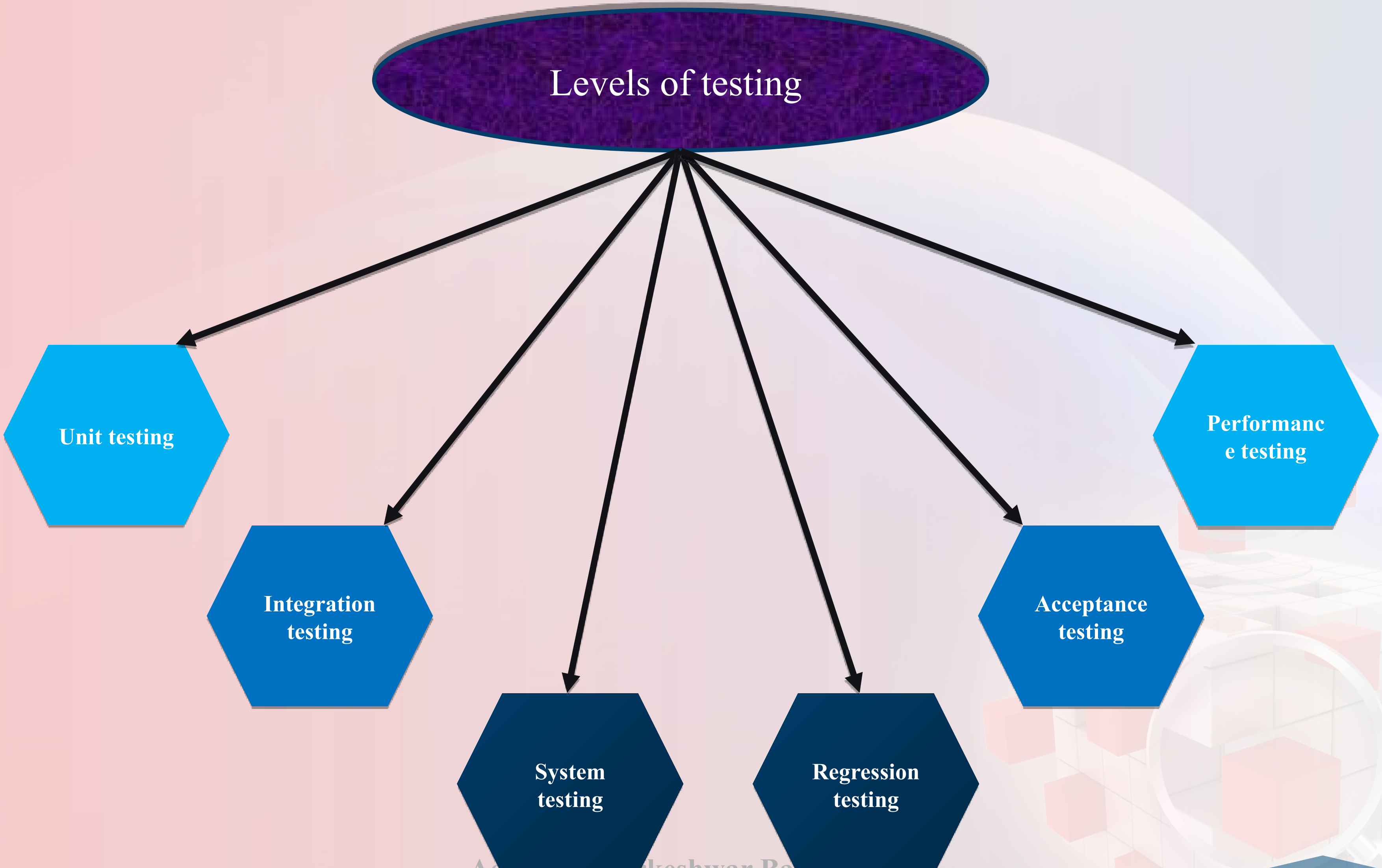
Black box testing

- Tests the software for its overall functionality.
- Does not test each statement and branches of the code.
- Access to the code is not available.
- Verifies if the actual output is same as the defined output.

White box testing

- Tests the software in detail.
- Tests every statement and condition given in the code.
- Requires access of code.
- Involves understanding the code and then testing every statement and condition given in the code.

Different Types of Testing (Contd.)



Different Types of Testing (Contd.)

Unit testing

- Tests the individual units of application, which can be a method of a class, a complete class, a set of inter-related class, or a module.
- Is one of the best development practices to test the smaller unit of code.
- Reduces the chances of encountering errors at a later stage of development and makes the code efficient.
- Ensures that it functions efficiently, individually on each unit of an application.

Integration testing

- Verifies that different units, when combined together, produce the desired result.
- Tests different modules of the given application while integration.
- Examines whether each module is working perfectly along with the other modules of the same application on integration.

Different Types of Testing (Contd.)

System testing

- Tests the complete system for the desired functionality.
- Needs you to combine all the units of an application to build the entire application.
- Tests the reliability of the overall system.
- Uses black box testing approach because in this case, you test the complete functionality of the software by providing certain input and verifying the outcome of that input.

Regression testing

- Refers to testing an application after it has been modified.
- Aims to find out bugs in the code.
- Looks for the effect of the change in other parts of the code.
- Retests the given modules with same or different tests to ensure that changes in a module have not impacted other modules.

Different Types of Testing (Contd.)

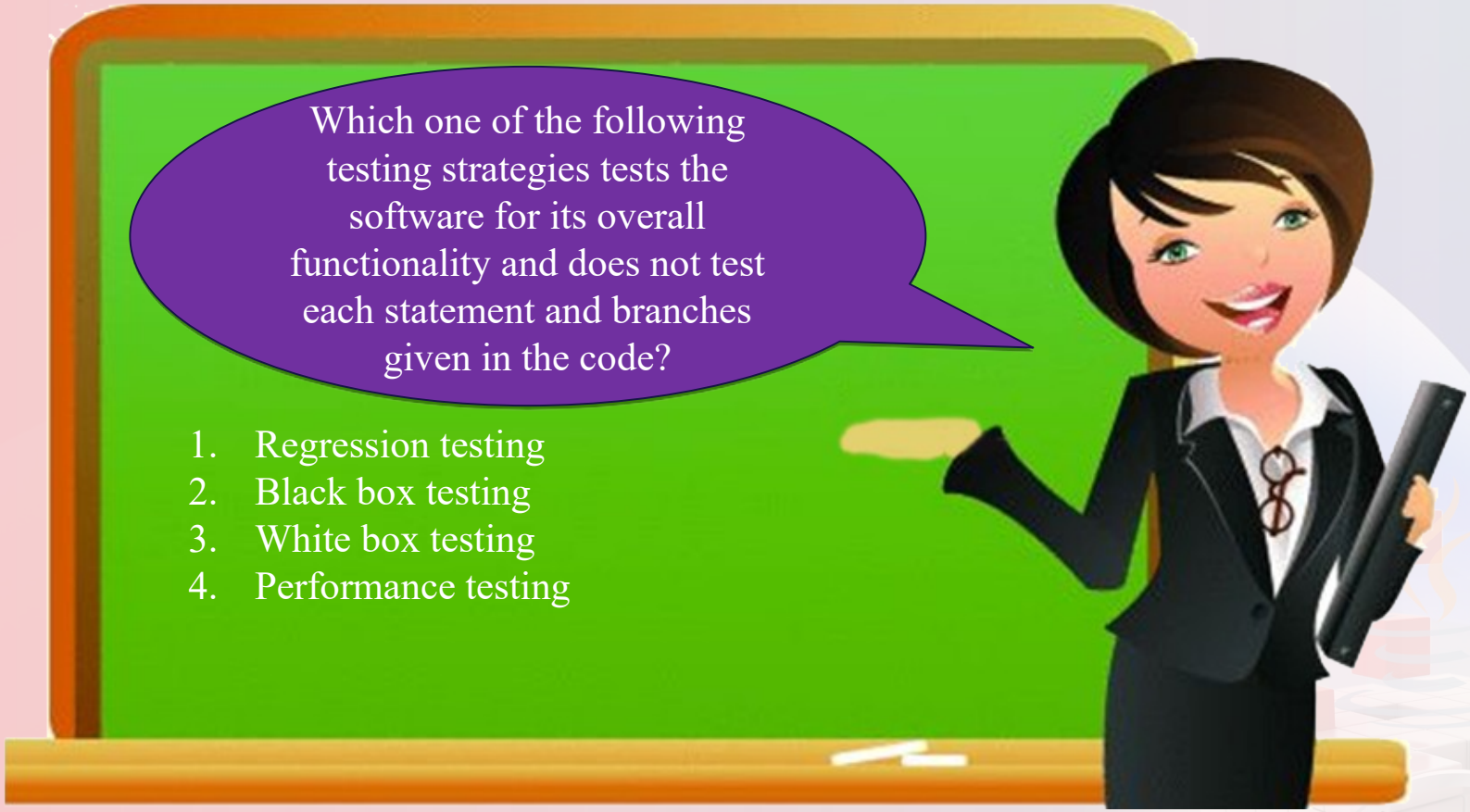
Acceptance testing

- Done at the customer's end for accepting the software.
- Verifies whether the developed application meets the user requirements.
- Performed at the client-site in real-time environment.
- Ensures that the proposed system is considered ready to be deployed at the client-side with all the tools and infrastructure present.

Performance testing

- Checks the response time of the application in a usual scenario as well as in a loaded scenario.
- Tests the entire non-functional performance expected of the system, which could be related to time, speed, security, reliability, and other performance-related issues of the application.

Just a Minute



Answer: 2. Black box testing

Aesthetic: Tarkeshwar Barua

Introducing Unit Testing

Testing small pieces of code, known as units, is known as unit testing.

This unit can be a method, multiple methods of a class, a class, multiple classes, or the modules of an application.

It is done to ensure that the smallest unit of the code is bug-free and reusable.

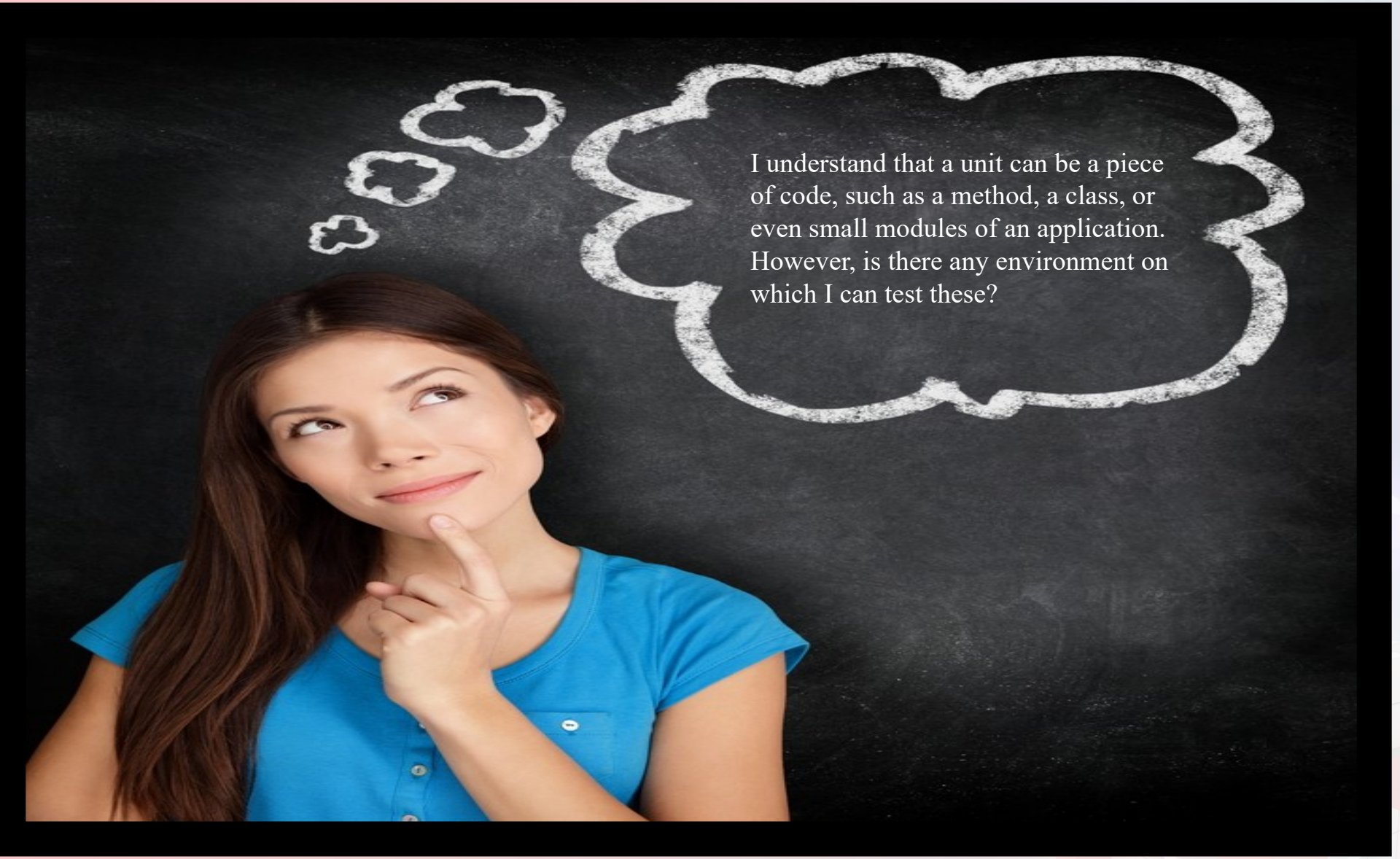
Introducing Unit Testing (Contd.)

Advantages of unit testing

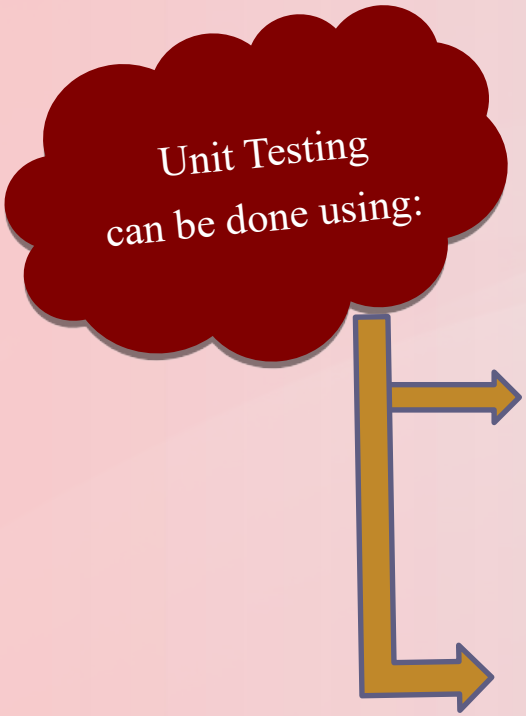
- It ensures that the individual units are reliable and error-free.
- It allows you to find the errors/issues in the code.
- It reduces the impact and the cost that might be huge when the errors occurs at a later stage.

Use the animation, Unit_Testing, to understand the concept of unit testing.

Exploring IDEs for Unit Testing



Exploring IDEs for Unit Testing (Contd.)

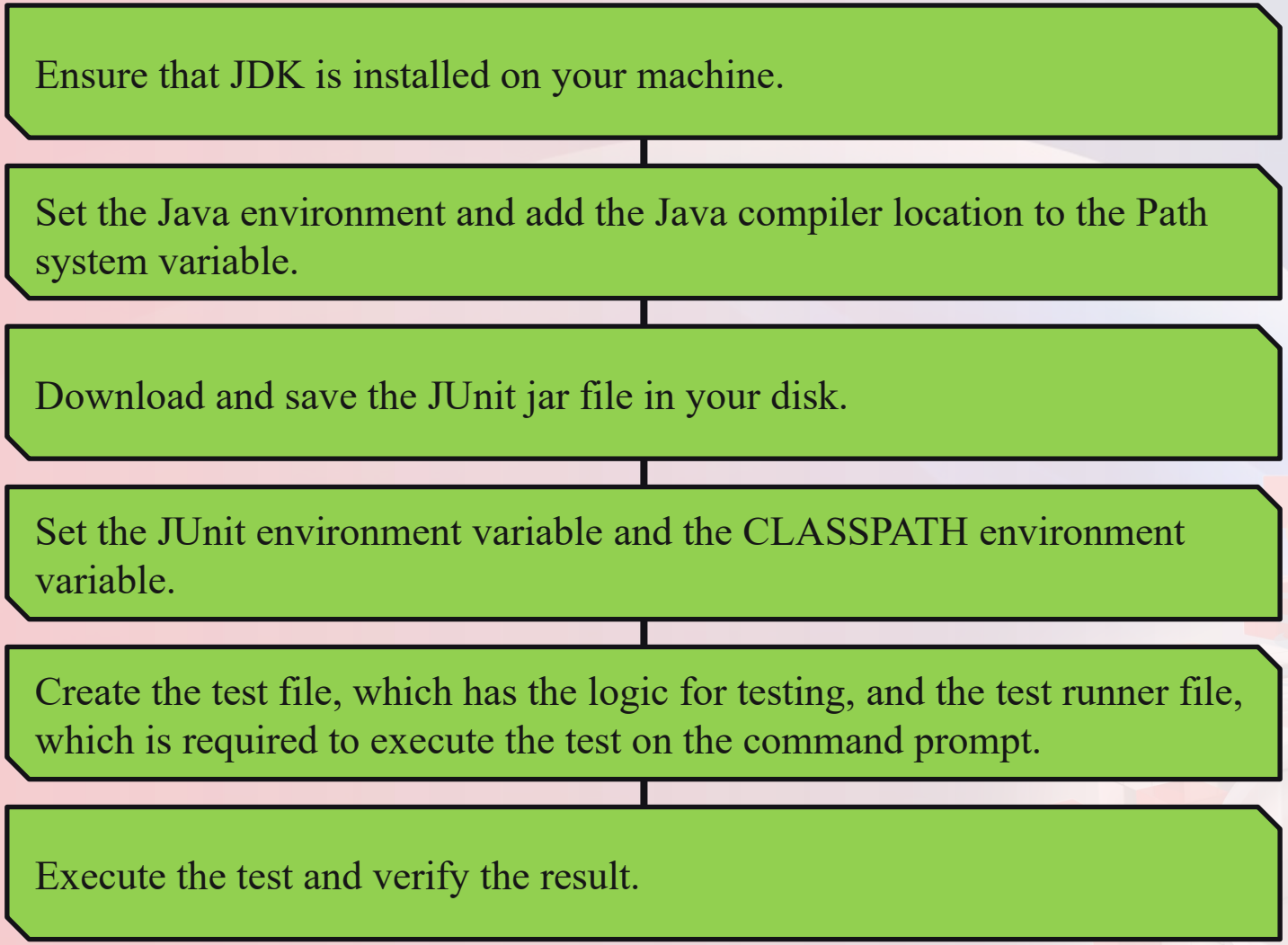


- **Command prompt**
  This is a complex and tedious procedure. You need to install JDK and set the Java environment.

- **IDEs, such as NetBeans and Eclipse**
  This is a simple process. These IDEs automatically set up the desired environment and download the necessary jar files required to execute unit test code.

Exploring IDEs for Unit Testing (Contd.)

- To use the JUnit framework at command line, you need to perform the following steps:



Exploring IDEs for Unit Testing (Contd.)

Is an open-source IDE for Java that provides source-code-related support.

DE for Java

Supports in-built tools such as compiler, debugger, version controller, and profiler.

Supports various types of application such as console based, mobile based, and Web based.

Supports various types of application

Runs on Windows, Mac OS X, GNU/Linux, and Solaris provided JVM is installed on it. It comes with an in-built support for SQL, MySQL, and Oracle drivers.

Runs on cross-platform machines

Supports various plugins

Comes with support for various plugins, which makes NetBeans an extensible IDE.

Aesthetic: Tarkeshwar Barua

Exploring IDEs for Unit Testing (Contd.)

Is open-source IDE sponsored by IBM, but is quite easier to set up and configure.

E for Java

Has in-built compiler and interpreter.

Integrated with in-built tools

Supports various types of application, such as console based, mobile based, and Web based.

Supports various types of applications

Runs on Windows, Mac OS X, GNU/Linux, and Solaris. These machines must have JVM installed on them.

Runs on cross-platform machines

In-built JDBC driver support

Has in-built JDBC driver support. It also has an automatic build feature, which automatically compiles the changes in the code.

Aesthetic: Tarkeshwar Barua

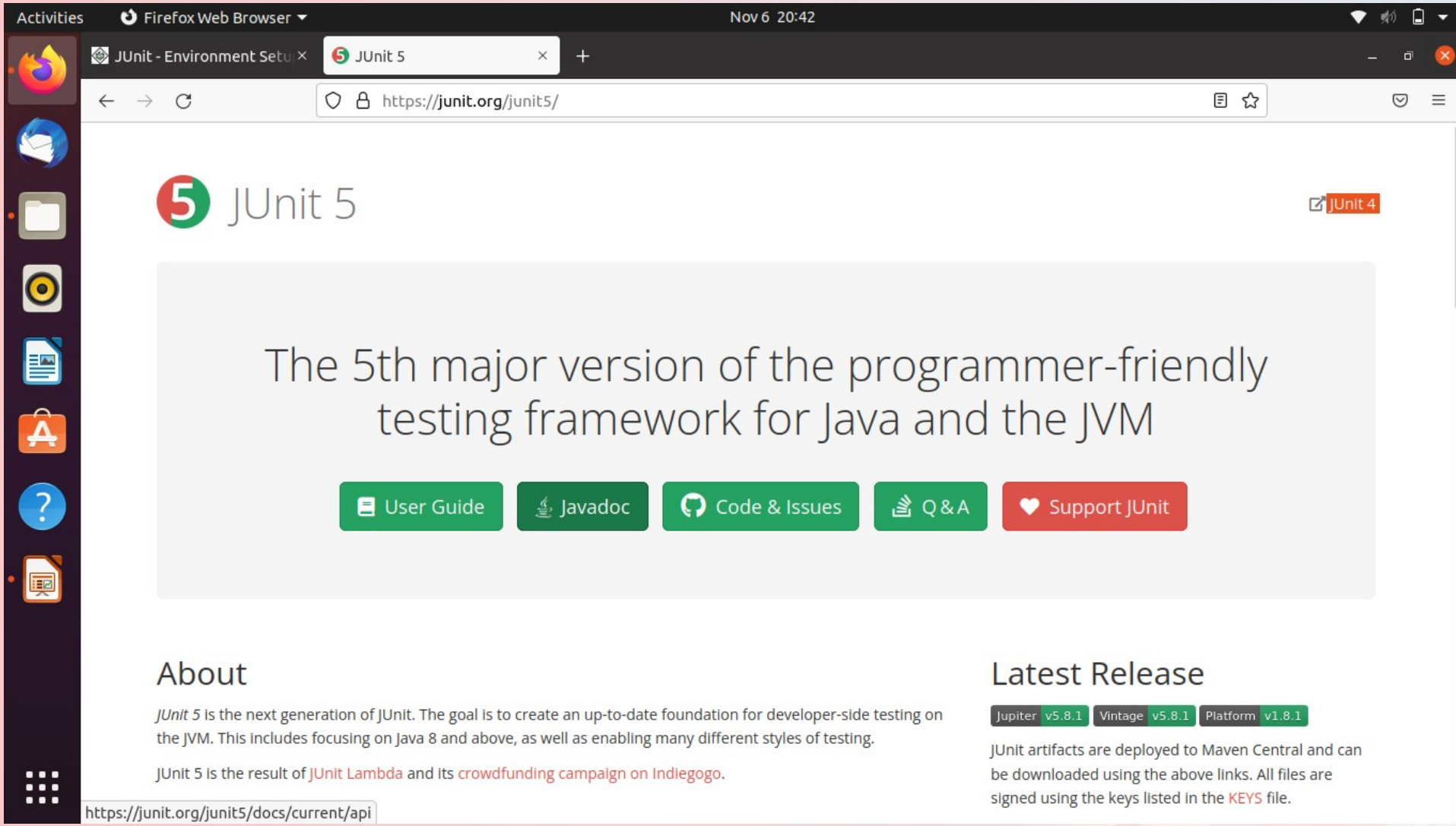
What is JUnit?

- JUnit promotes the idea of **"first testing then coding"**, which emphasizes on setting up the test data for a piece of code that can be tested first and then implemented.
- This approach is like **"test a little, code a little, test a little, code a little."**
- It increases the productivity of the programmer and the stability of program code, which in turn reduces the stress on the programmer and the time spent on debugging.

Local Environment Setup

- JDK 1.5 and above(Recommanded 1.8 JDK)
- c:\> java --version
- $ java --version
- machine:~ joseph$ java --version
- JAVA_HOME to C:\Program Files\Java\jdk1.8.0_101
- export JAVA_HOME = /usr/local/java-current
- export JAVA_HOME = /Library/Java/Home
- C:\Program Files\Java\jdk1.8.0_101\bin in the Path.
- export PATH = $PATH:$JAVA_HOME/bin/
- Mac not required any path settings

Local Environment Setup



What is in Junit?

- https://github.com/downloads/junit-team/junit/junit-4.10.jar
- https://junit.org/junit5/docs/current/user-guide/
- **writing tests, extension authors, and engine authors as well as build tool and IDE vendors.**
- **Unit 5 is composed of several different modules from three different sub-projects.**
- **JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage**

JUnit Platform

- It serves as a foundation for launching testing frameworks on the JVM.
- It also defines the TestEngine API for developing a testing framework that runs on the platform.
- The platform provides a Console Launcher to launch the platform from the command line and a JUnit 4 based Runner for running any TestEngine on the platform in a JUnit 4 based environment.
- First-class support for the JUnit Platform also exists in popular IDEs (see IntelliJ IDEA, Eclipse, NetBeans, and Visual Studio Code) and build tools (see Gradle, Maven, and Ant).
-

JUnit Jupiter

- The combination of the new programming model and extension model for writing tests and extensions in JUnit 5.
- The Jupiter sub-project provides a TestEngine for running Jupiter based tests on the platform.

```
<dependency>
    <groupId>org.junit</groupId>
    <artifactId>junit-bom</artifactId>
    <version>5.8.1</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>
```

JUnit Vintage

- TestEngine for running JUnit 3 and JUnit 4 based tests on the platform. It requires JUnit 4.12 or later to be present on the class/module path.

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <scope>test</scope>
</dependency>
```

- testImplementation(platform('org.junit:junit-bom:5.8.1'))
- testImplementation('org.junit.jupiter:junit-jupiter')

JUnit Vintage

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import example.util.Calculator;
import org.junit.jupiter.api.Test;
class MyFirstJUnitJupiterTests {
    private final Calculator calculator = new Calculator();
    @Test
    void addition() {
        assertEquals(2, calculator.add(1, 1));
    }
}
```

Sample Demo Test Case

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class TestJunit {

 @Test
   public void testAdd() {
      String str = "Junit is working fine";
      assertEquals("Junit is working fine",str);
   }
}
```

Test Case Runner

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;
public class TestRunner {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(TestJunit.class);

      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }

      System.out.println(result.wasSuccessful());
   }
}
```

Test Case Execution

C:\JUNIT_WORKSPACE>javac TestJunit.java TestRunner.java

Now run the Test Runner to see the result as follows −

C:\JUNIT_WORKSPACE>java TestRunner

Verify the output.

true

Demo: Creating Test Scenarios

- **Problem Statement**
  - Sandra has created the following code snippet to find the greatest of three numbers:

```
public class Greatest {
    public static int greatest(int a, int b, int c)
    {
        if(a>b && a>c)
        return a;
        else if(b>a && b>c)
        return b;
        else
        return c;
    }
}
```

  - Now, you need to test this class considering the following conditions:
    - Pass the integer values to the method to make the test pass.
    - Pass the negative values to the method to check its outcome.
    - Pass the decimal values to the method to make the test fail.
    - Pass the string values to the method to make the test fail.
  - Create four test scenarios for the preceding conditions.