accenture
Latvia ATC

Test Automation Engineering Fundamentals: Java

Module 17: Unit Testing

Module Objectives

accenture
Latvia ATC

- At the end of this module, you will be able to:
  - Define the concepts of unit testing
  - Describe the purpose and benefits of unit testing
  - Identify the general principles of test conditions
  - Apply and utilize the principles of test conditions
  - Explain the JUnit framework
  - Perform unit testing using the JUnit framework

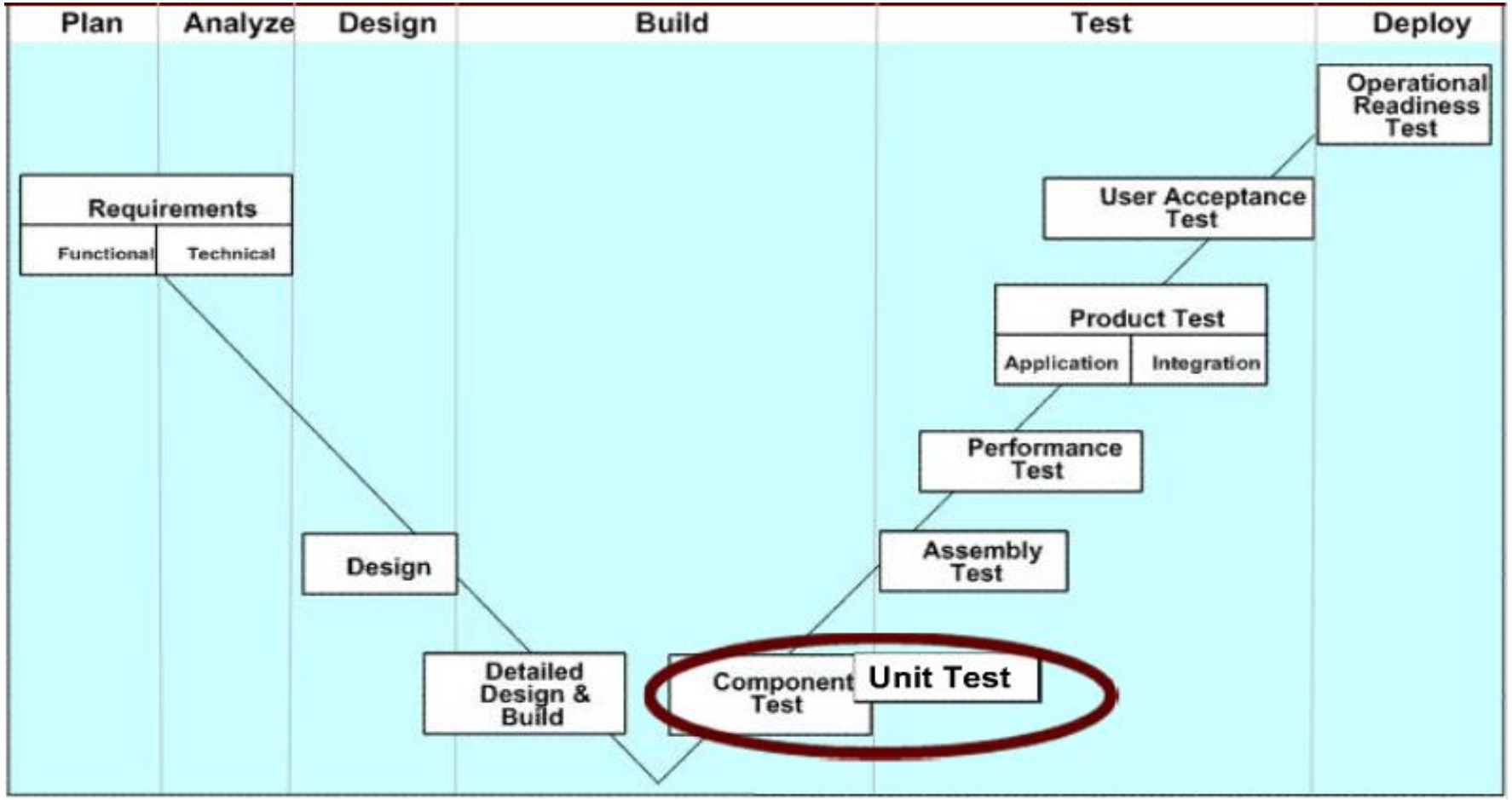
Overview of Unit Testing Concepts

accenture
Latvia ATC

- A unit is the smallest testable part of an application. In Java, and other object-oriented languages, a unit is a method.
- Testing is a way of evaluating software, to determine if requirements and expectations are met, and to detect errors.
- Unit testing validates individual units of source code to check if they are working properly.
- Unit testing is usually done by the developer, during the building phase of the application.
- Although the nature of unit testing may have similarities with debugging, they must not be confused with each other.

Overview of Unit Testing Concepts (cont.)

accenture
Latvia ATC



Overview of Unit Testing Concepts (cont.)

accenture
Latvia ATC

- **Purpose of Unit Testing**
  - Ensures that code is consistent with the detailed design
  - Ensures that the program can execute normally
  - Early detection of bugs
  - Prepares for future test phases

- **Unit Test should cover**
  - Code
  - Branches
  - Paths
  - Cycles

Overview of Unit Testing Concepts (cont.)

accenture
Latvia ATC

Scope of Unit Testing

- For newly generated/developed code
  - All units/components of the code
- For changed/modified code
  - All the affected units/components of the code along with the units/components that were directly changed

Overview of Unit Testing Concepts (cont.)

accenture
Latvia ATC

- The technique usually used for unit testing is white-box testing.
- White-box testing is a testing strategy wherein the internal workings of a unit/component are studied to create test conditions.
- The test environment should be, as much as possible, similar to the real environment.

Overview of Unit Testing Concepts (cont.)

accenture
Latvia ATC

- **Benefits of Unit Testing**
  - Facilitates Change
  - Simplifies Integration
  - Provides living documentation
  - Serves as a design

- **Unit testing Phases**
  - Detailed Design Phase
  - Coding Phase
  - Unit Test Phase

Overview of Unit Testing Concepts (cont.)

accenture
Latvia ATC

- **Unit Testing Deliverables**
  - Test Conditions and Expected Results
  - Test Cycle Control Sheet
  - Test Data
  - Test Drivers / Media Programs Document
  - Test Environment Document

Overview of Unit Testing Concepts (cont.)

accenture
Latvia ATC

Deliverables	Detailed Design Phase	Coding Phase	Unit Test Phase
Test Conditions and Expected Results	Created	Updated	Updated
Test Cycle Control Sheet	Created	Updated	Updated
Test Data Document	Created	Updated	Updated
Test Driver / Media Program Document	Created	Updated	Updated
Test Actual Result			Created
Test Environment Document	Created	Updated	Updated

Overview of Unit Testing Concepts (cont.)

accenture
Latvia ATC

- **Limitations of Unit Testing**
  - Checks only the functionality of units; unable to identify system-wide issues
  - Exposes only the presence of errors, not their absence
  - Combinatorial problems require significant amount of test code to be written to cover all combinations
  - Needs rigorous sense of discipline throughout the software development process
  - More effective when used in conjunction with other software testing activities

Writing Test Conditions

accenture
Latvia ATC

- A test condition is a statement that defines a constraint that must be satisfied by the program being tested.

- **Test Condition General Principles**
  - Branch Coverage
  - Condition Coverage
  - Loop Coverage
  - Interface Coverage
  - Logic Path Coverage

Writing Test Conditions (cont.)

accenture
Latvia ATC

Branch Coverage

- Test conditions should be developed to cover all logic branches in the function modules. This ensures that no executable statements are missed.

- Some conditions that may belong in this category are
  - IF-ELSE-ENDIF
  - CASE-WHEN-ENDCASE
  - WHILE-ENDWHILE

Writing Test Conditions (cont.)

accenture
Latvia ATC

Condition Coverage

- Test conditions should be developed to make sure that every possible value of every judgment can be executed at least once.
- Test conditions that are used are boundary conditions usually for conditions that allow ranges.
- IF-ELSE conditions and its combinations belong in this category.

Writing Test Conditions (cont.)

accenture
Latvia ATC

Loop Coverage

- Test conditions should be developed to cover all possible cases, including the loop being executed only once, or multiple times, or executed maximum times, or even the loop never being executed.

- Loop statements such as the FOR, WHILE, DO-WHILE belong in this category.

Writing Test Conditions (cont.)

accenture
Latvia ATC

- **Interface Coverage**
  - It should be ensured that the interface for individual program units can work correctly
- **Logic Path Coverage**
  - All logic paths should be tested during the unit test
- **Other Types of Coverage**
  - Exception Handling Test
  - Argument Type Cast Test
  - Comparison Computing Test
  - Local Variable Usage Test

Using JUnit as a Testing Tool

accenture
Latvia ATC

- JUnit is an open source Java testing framework shipped with Eclipse. It is an instance of the xUnit architecture for unit testing frameworks.

- **JUnit features include:**
  - Assertions for testing expected results
  - Test fixtures for sharing common test data
  - Test suites for easily organizing and running tests
  - Graphical and textual test runners

Using JUnit as a Testing Tool (cont.)

accenture
Latvia ATC

- JUnit helps to write code faster while increasing code quality.
- JUnit tests check their own results and provide immediate feedback.
- JUnit tests increase the stability of software.
- JUnit is elegantly simple.
- JUnit tests are developer tests.
- JUnit tests are written in Java.

Using JUnit as a Testing Tool (cont.)

accenture
Latvia ATC

- A test case is a set of inputs, execution conditions, and expected results, which is used to evaluate a program's compliance towards a specified requirement.

- **Tips for creating test cases:**
  - Test results should be quick and easy to determine and understand.
  - Tests should be self explanatory.
  - Tests should be valid, unless intentionally testing for errors and/or error-recovery.

Using JUnit as a Testing Tool (cont.)

accenture
Latvia ATC

This is a sample code of a method that adds two numbers, *num1* and *num2*.

```
public int add(int num1, int num2)
        {
                return num1 + num2;
        }
```

```
public void TestAdd()
{
  if(add(3,5)==8)
  {
    System.out.println("Success !");
 }
 else
 {
    System.out.println("Oops. Failed!");
 }
}
```

This is a sample unit test case that verifies the result of the method on the top figure.

Using JUnit as a Testing Tool (cont.)

accenture
Latvia ATC

- In JUnit, unit test cases are created by using assert methods.

- Assert methods are built in the JUnit framework and are used to determine the compliance of source code to the requirements.

- Commonly used assert methods are:
  - assertTrue(boolean),
  - assertTrue(String, boolean)
  - assertEquals(Object, Object)
  - assertNull(Object)
  - fail(String)

Using JUnit as a Testing Tool (cont.)

accenture
Latvia ATC

This is a generic test case, which prints out a message depending on the outcome of the AddTwoNumbers method.

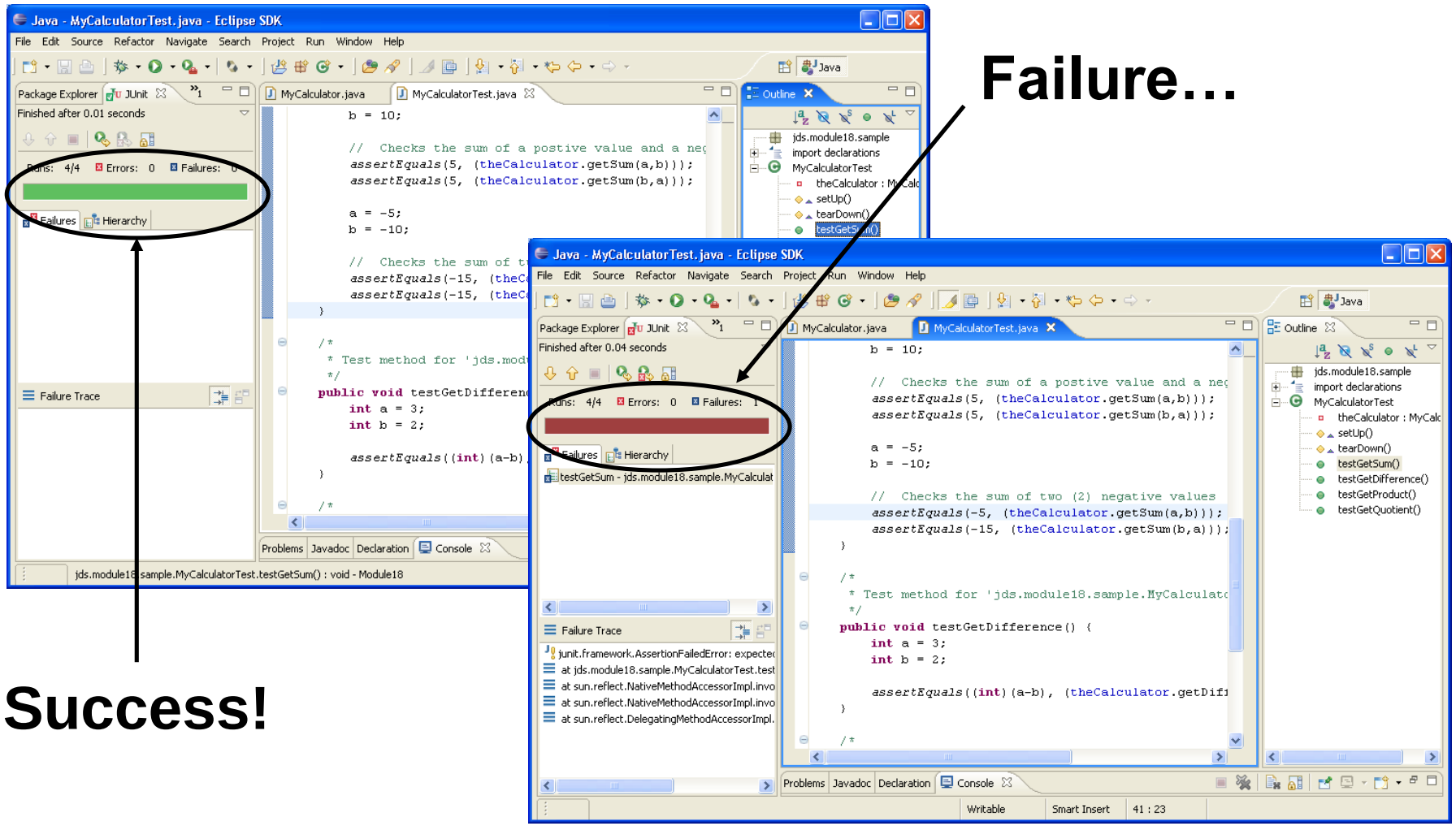
```
public void TestAdd()
{
  if(add(3,5)==8)
   {
     System.out.println("Success !");
  }
  else
  {
     System.out.println("Oops. Failed!");
  }
}
```

```
public void TestAdd()
{
  assertEquals(8, add( 3 , 5 ));
}
```

This test case performs the same task as above and shows the results with a red or green bar, instead of string messages.

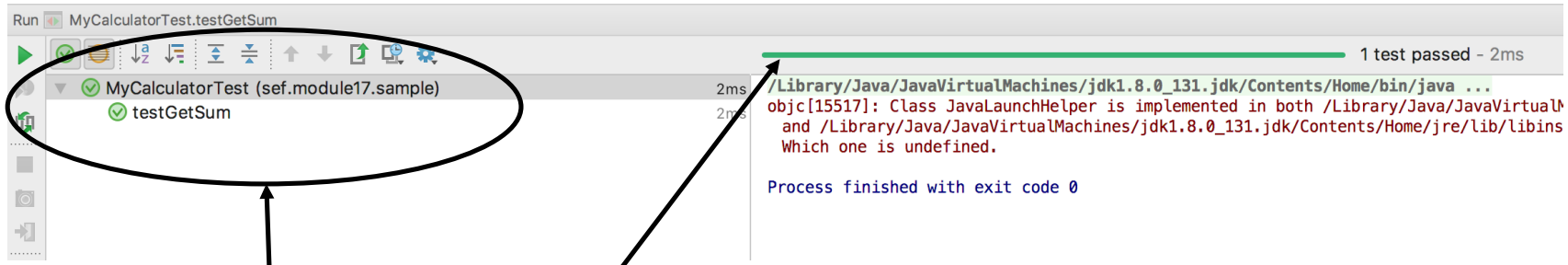
Using JUnit as a Testing Tool (cont.)

accenture
Latvia ATC



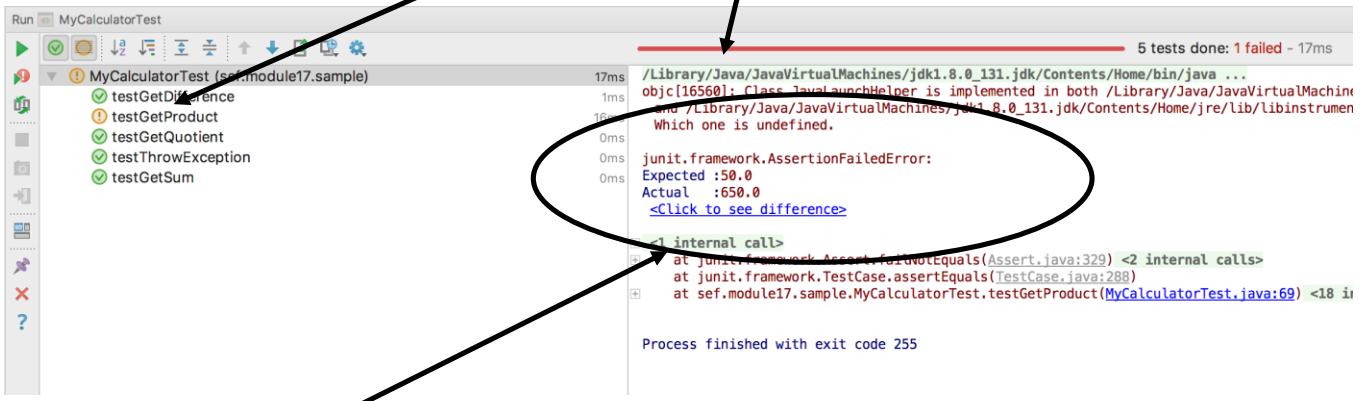
Using JUnit as a Testing Tool (cont.)

accenture
Latvia ATC



Success!

Failure...



Error message

Using JUnit as a Testing Tool (cont.)

accenture
Latvia ATC

To create a JUnit Test case:

1. Code the component to be tested
2. Create a corresponding JUnit test case by right-clicking the component's source code class
   - In the pop-up menu choose "New" then choose "JUnit Test Case"
3. In the JUnit dialog box:
   i. Change the package name to a specific project package standard by appending "test." to the package name
   ii. Select the "setUp" and "tearDown" methods and click "Next"
   iii. Choose the methods that will be included in the JUnit test case and click the "Finish" button

Using JUnit as a Testing Tool (cont.)

accenture
Latvia ATC

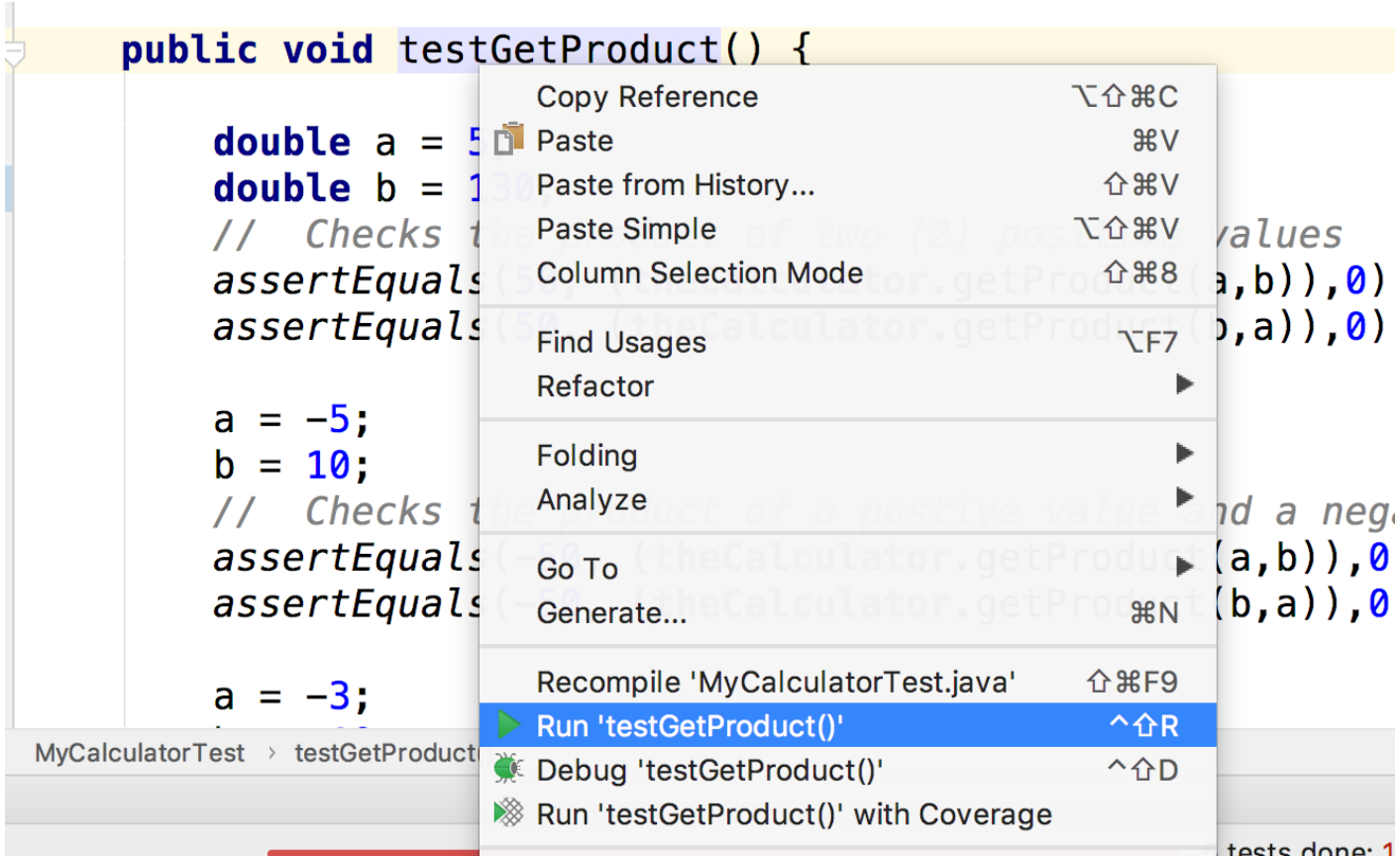
- In the JUnit Test Case class, make sure that you have imported all the needed packages to be used, including the package name of the component to be tested.

- To execute the test cases:

Using JUnit as a Testing Tool (cont.)

accenture
Latvia ATC

To execute the test cases:



Using JUnit as a Testing Tool (cont.)

accenture
Latvia ATC

- In the “setUp” method, it is a good practice to initialize all variables that are to be used in the tests.

- The developer can also make use of the “tearDown” method, to explicitly remove/destroy all variables created and used by the test case(s).

Using JUnit as a Testing Tool (cont.)

accenture
Latvia ATC

Sample code:

package sef.module18.sample;

```
public class MyCalculator {
    public int getSum(int a, int b){
            return a + b;
    }
    public int getDifference(int a, int b){
            return a - b;
    }
    public double getProduct(double a, double b){
            return a * b;
    }
    public double getQuotient(double a, double b){
            return a / b;
    }
}
```

Using JUnit as a Testing Tool (cont.)

accenture
Latvia ATC

JUnit Test Case

package sef.module18.sample;

```
import junit.framework.TestCase;

public class MyCalculatorTest extends TestCase {

    private MyCalculator theCalculator;

    protected void setUp() throws Exception {
            super.setUp();
            //     Initialize variables to be used here
            theCalculator = new MyCalculator();
    }

    protected void tearDown() throws Exception {
            super.tearDown();
    }
```

Using JUnit as a Testing Tool (cont.)

accenture
Latvia ATC

```
public void testGetSum() {

        int a = 5;
        int b = 10;
        // Checks the sum of two (2) positive values
        assertEquals(15, (theCalculator.getSum(a,b)));
        assertEquals(15, (theCalculator.getSum(b,a)));

        a = -5;
        b = 10;
        // Checks the sum of a postive value and a negative value
        assertEquals(5, (theCalculator.getSum(a,b)));
        assertEquals(5, (theCalculator.getSum(b,a)));

        a = -5;
        b = -10;
        // Checks the sum of two (2) negative values
        assertEquals(-15, (theCalculator.getSum(a,b)));
        assertEquals(-15, (theCalculator.getSum(b,a)));
}
```

Refer to the MyCalculator.getSum(int, int) sample code.

Using JUnit as a Testing Tool (cont.)

accenture
Latvia ATC

```
public void testGetDifference() {
        int a = 3;
        int b = 2;
        // Checks if b is lesser than a
        assertEquals(1, (theCalculator.getDifference(a,b)));
        // Checks if b is greater than a
        assertEquals(-1, (theCalculator.getDifference(b,a)));

        a = 5;
        b = -4;
        // Checks if negative value is subtracted from positive value
        assertEquals(9, (theCalculator.getDifference(a,b)));
        // Checks if positive value is subtracted from negative value
        assertEquals(-9, (theCalculator.getDifference(b,a)));

        a = -10;
        b = -5;
        // Checks if negative value is subtracted from negative value: a > b
        assertEquals(-5, (theCalculator.getDifference(a,b)));
        // Checks if negative value is subtracted from negative value: b < a
        assertEquals(5, (theCalculator.getDifference(b,a)));
}
```

Refer to MyCalculator.get(Difference (int, int) sample code.

Using JUnit as a Testing Tool (cont.)

accenture
Latvia ATC

```
public void testGetProduct() {

        double a = 5;
        double b = 10;
        // Checks the product of two (2) positive values
        assertEquals(50, (theCalculator.getProduct(a,b)),0);
        assertEquals(50, (theCalculator.getProduct(b,a)),0);

        a = -5;
        b = 10;
        // Checks the product of a positive value and a negative value
        assertEquals(-50, (theCalculator.getProduct(a,b)),0);
        assertEquals(-50, (theCalculator.getProduct(b,a)),0);

        a = -3;
        b = -10;
        // Checks the product of two (2) negative values
        assertEquals(30, (theCalculator.getProduct(a,b)),0);
        assertEquals(30, (theCalculator.getProduct(b,a)),0);

}
```

Refer to MyCalculator.getProduct (double, double) sample code.

Using JUnit as a Testing Tool (cont.)

accenture
Latvia ATC

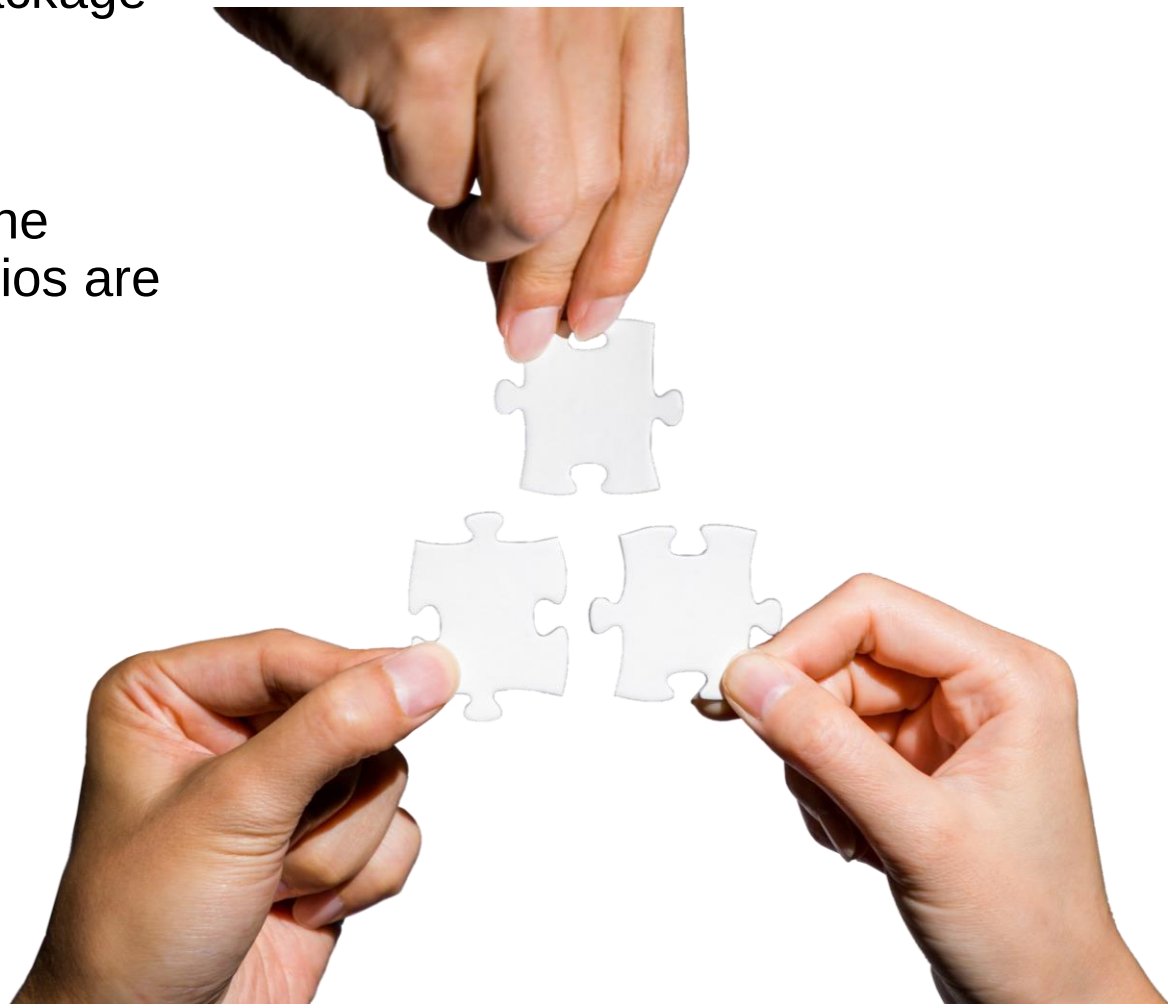
```
public void testGetQuotient() {

        double a = 10;   double b = 5;
        // Checks if b is lesser than a
        assertEquals(2, (theCalculator.getQuotient(a,b)),0);
        // Checks if b is greater than a
        assertEquals(0.5, (theCalculator.getQuotient(b,a)),0);

        a = 9;   b = -3;
        // Checks if positive value is divided with negative value
        assertEquals(-3, (theCalculator.getQuotient(a,b)),0);
        // Checks if negative value is divided with positive value
        assertEquals(-0.33, (theCalculator.getQuotient(b,a)),0.007);

        a = -10; b = -5;
        // Checks if negative value is divided with negative value: a > b
        assertEquals(2, (theCalculator.getQuotient(a,b)),0);
        // Checks if negative value is divided with negative value: b < a
        assertEquals(0.5, (theCalculator.getQuotient(b,a)),0);

        }
}
```

Refer to MyCalculator.getQuotient (double, double) sample code.

Activity – Part 1

accenture
Latvia ATC

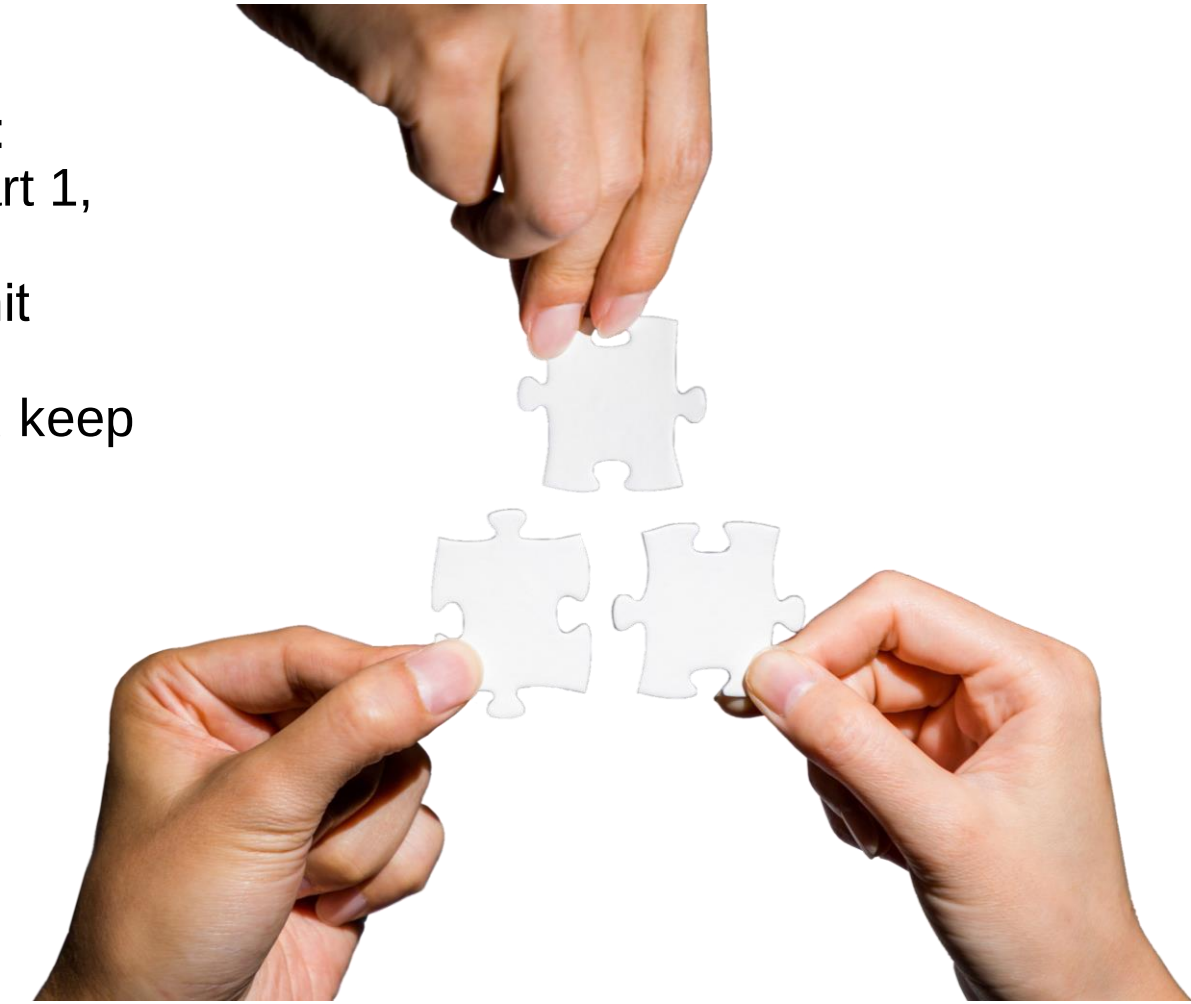
1) Open your SEF workspace in Eclipse.
2) In the package explorer, navigate to the following package **sef.module17.activity.**
3) Open the following class:
   EmployeeJDBC.java
4) Create TCERs for this class. Base your TCERs on the methods written in this class. Ensure that all scenarios are covered.



Activity – Part 2

accenture
Latvia ATC

1) Open your SEF workspace in Eclipse.
2) In the package explorer, navigate to the following package **sef.module17.activity.**
3) Create JUnit Test Cases for **EmployeeJDBC.java**:
   - Based on the TCERs you created in activity part 1, create the test methods.
4) Right-click on the Test class, select 'Run As'->'JUnit Test' and hope for a green bar!
5) If the JUnit test returns a red, check your code and keep working on it until you get a green bar.



Questions and Comments

accenture
Latvia ATC

What questions or comments do you have?