Software Engineering COMP1035

Lecture 11

Test Driven Development (TDD)

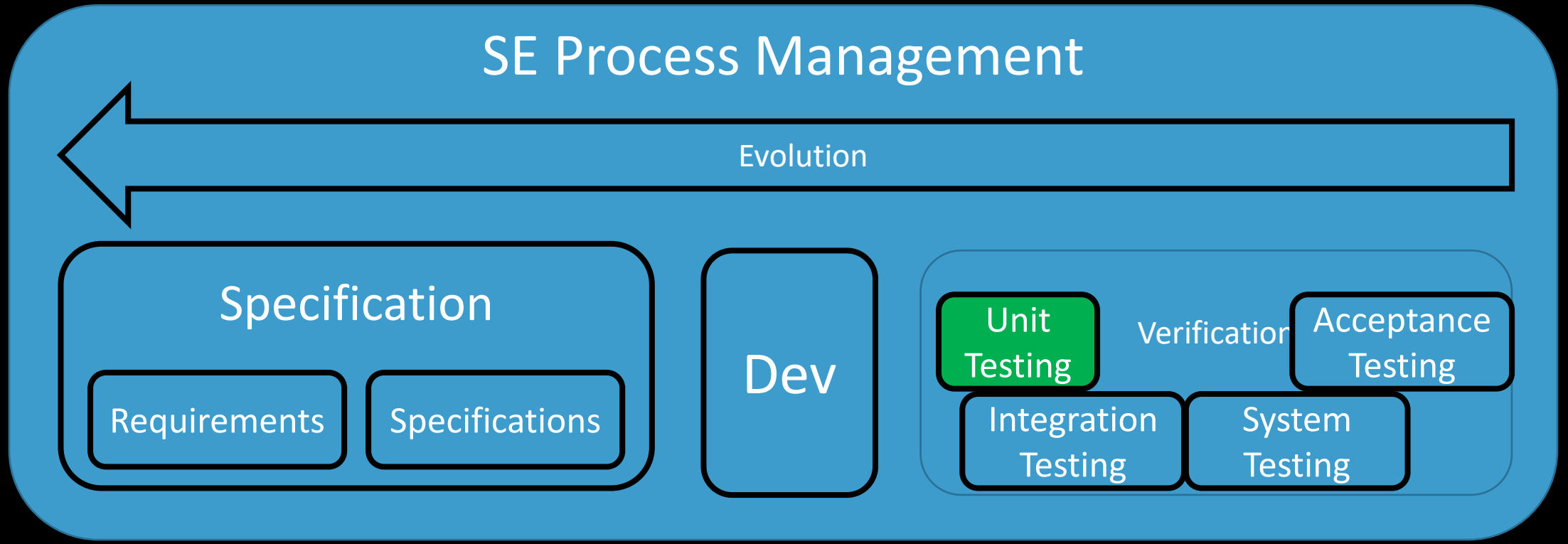
University of Nottingham
UK | CHINA | MALAYSIA

Outline

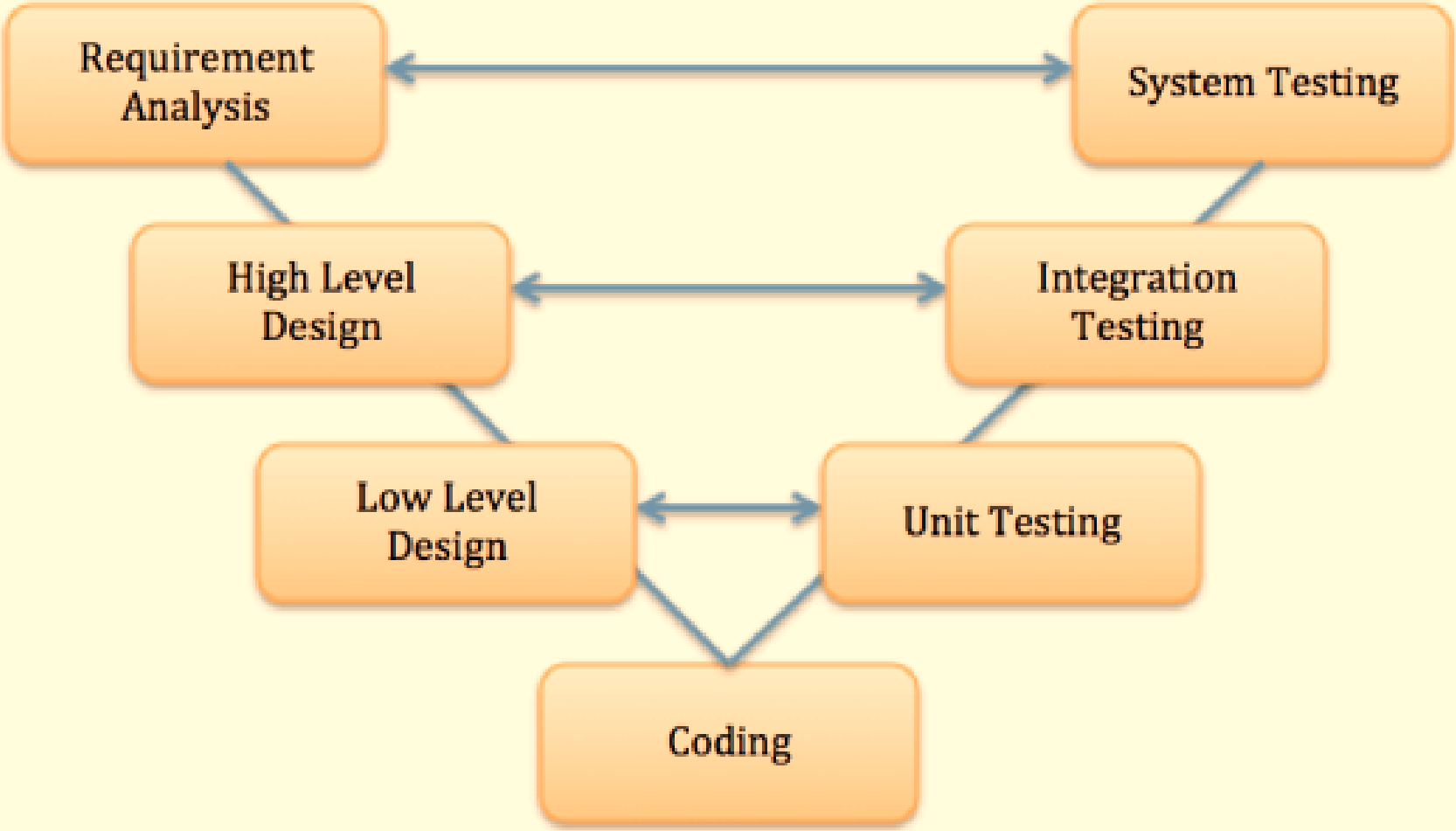
- Testing Methods
- Testing Processes
- Test Driven Development
- Junit
- Testing Metrics

Where We Are In the Process

Keeping Track of SE Module



Keeping Track of SE Module



Different kinds of Test Approaches

- Acceptance testing
- Black box testing
- Compatibility testing
- Conformance testing
- Functional testing
- Integration testing
- Load testing
- Regression Testing
- Smoke testing
- System testing
- Unit testing
- White box testing

Testing Methods

- White Box Testing
  - A procedure to derive and/or select test cases based on an analysis of the **internal structure** of a component or system.
  - Clear-box testing, Code-based testing, Logic-driven testing, etc.
- Black Box Testing
  - A procedure to derive the test cases based on the **functionality** of the application and not considering the internal structure of the system.
  - Specification-Based Testing

White Box Testing

- Include the following **code coverage** criteria:
  - **Statement Coverage**: Ensure that each code statement is executed once.
  - **Branch Coverage** or **Node Testing**: Coverage of each code branch.
  - **Compound Condition Coverage**: For multiple conditions test each condition with multiple paths and combination of the different path to reach that condition.
  - **Basis Path Testing**: Each independent path in the code is taken for testing.
  - **Data Flow Testing** (**DFT**): each data variable is tracked and its use is verified.
  - **Path Testing**: All possible paths through the code are defined and covered.
  - **Loop Testing**: Independent and dependent code loops and values are tested by this approach.

White Box Testing

- To ensure:
  - That all independent paths within a module have been executed at least once.
  - All logical decisions verified on their true and false values.
  - All loops executed at their boundaries and within their operational bounds.
- To discover the following types of bugs:
  - Logical error
  - Design error
  - Syntax error
- For large systems, it is not always possible…

Black Box Testing

- Techniques:
  - **Equivalence Partitioning**: input values to the system or application are divided into different classes or groups based on its similarity in the outcome.
  - **Boundary Value Analysis**: focus on the values at boundaries
  - **Decision Table Testing**: use decision table to prepare a set of test cases for logical relationships
  - **State Transition Testing**: The state of the system changes depending upon the conditions or events. Test the different states of the system.
  - **Error Guessing**: experience-based testing, use experience to guess the error-prone areas.
  - **Comparison Testing**: different independent versions of the same software are used to compare to each other for testing.

Black Box Testing

- Advantages:
  - The tester does not need to have a technical background.
  - Testing can start once the development of the project/application is done. Both the testers and developers work independently.
  - It is more effective for large and complex applications.
- Disadvantages:
  - Without any technical or programming knowledge, there are chances of ignoring possible conditions of the scenario to be tested.
  - Complete Test Coverage is not possible for large and complex projects.

WBT V.S BBT

White Box Testing	Black Box Testing
It is a testing method having knowledge about the actual code and internal structure of the application.	It is a testing method without having knowledge about the actual code or internal structure of the application.
This type of testing is performed at a lower level of testing such as Unit Testing, Integration Testing.	This is a higher level testing such as functional testing.
It concentrates on the actual code – program and its syntax.	It concentrates on the functionality of the system under test.
White Box testing requires Design documents with data flow diagrams, flowcharts etc.	Black box testing requires Requirement specification to test.
White box testing is done by Developers or testers with programming knowledge.	Black box testing is done by the testers.

Testing Processes

- Manual Testing
  - The process of testing the software manually.
  - The skills, knowledge, and experience of the testers play an important role.
- Automation Testing
  - Use automation tools for executing test cases.

Manual Testing V.S Automation Testing

Manual Testing	Automation Testing
Test cases are executed manually	Test cases are executed with the help of tools
Not efficient, nor reliable	More efficient and reliable
The accuracy cannot be guaranteed	The accuracy can be guaranteed
Difficult to test the application on different OS	Easy to test the application on different OS
It is beneficial to check ease for accessing the application	Unable to check usability or accessibility

Manual Testing V.S Automation Testing

- Scenarios where we should consider Automation Testing
  - Areas where we have to do frequent testing.
  - Test cases with the possibility of human making mistakes
  - Test cases that need to be tested with different browsers and different environments
- Scenarios where we should consider Manual Testing
  - Areas of application which change frequently.
  - A newly designed test and the one that is not executed manually
  - Usability or accessibility test


Test-Driven Development

Test-Driven Development

- Test Driven Development (TDD) is software development approach in which test cases are developed to specify and validate what the code will do.
- In simple terms, **test cases** for each functionality are created and **tested first** and if the **test fails** then the **new code** is **written** in order to pass the test and making code simple and bug-free.
- White box testing or black box testing?
- It can be done at different levels.
  - Unit is the most common, but it can be used for integration etc.

The Three Rules of TDD (Robert C. Martin)

- You are not allowed to write any production code unless it is to make a failing unit test pass. (**Create a unit test that fails.**)
- You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures. (**Write the unit tests that are complied and sufficiently good enough.**)
- You are not allowed to write any more production code than is sufficient to pass the one failing unit test. (**Write the production codes that pass the unit tests.**)

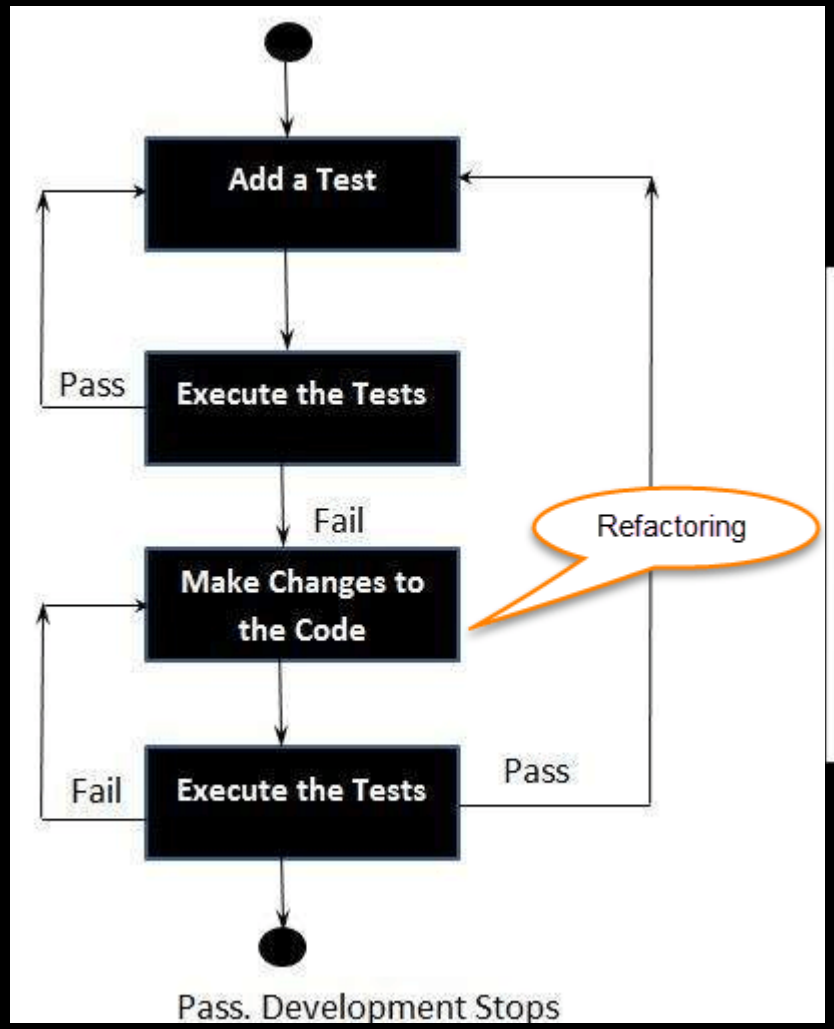
The Three Rules of TDD (Robert C. Martin)

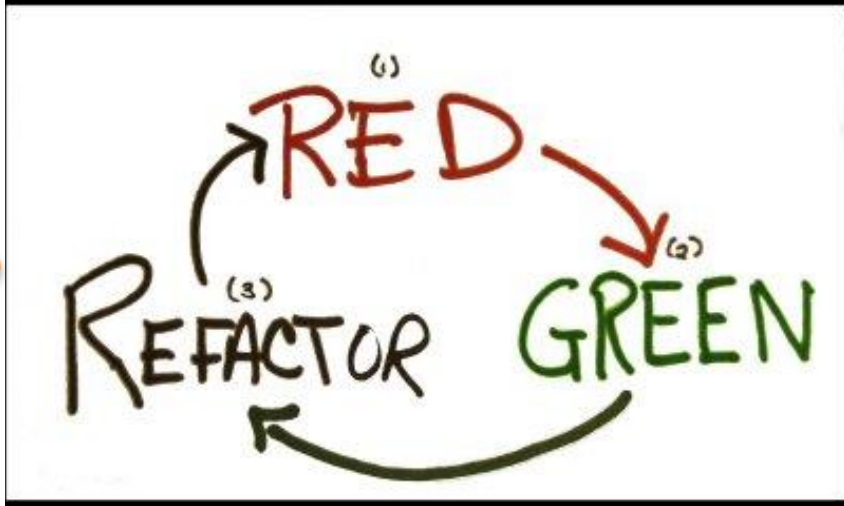
(1) Begin by writing a unit test for the functionality that you intend to write.
-- Red light from unit testing.

(2) You can't write very much of that unit test. As soon as the unit test code fails to compile, or fails an assertion, you must stop and write production code.
-- Green light from the unit testing.

(3) You can only write the production code that makes the test compile or pass, and no more
-- Refactor

In everything we do, whether writing tests, writing production code, or refactoring, we keep the system executing at all times.





TDD - Advantages

- It integrates aspects of Specification and Coding and Testing
- It does as much specification as it is testing
- It also belongs to parts of documentation!
- Makes you think about how code is used, before you build it.
- It means you ‘plan’ code before you ‘write’ code.
- It means you code to the big picture, not the current function.
- Also: When you make a change
  - it checks you haven't broken something from earlier
  - you know things that *still* pass tests, even after e.g. reconstruction.

TDD V.S Traditional Testing

- With traditional testing, a successful test finds one or more defects. It is same as TDD. When a test fails, you have made progress because you know that you need to resolve the problem.
- TDD ensures that your system actually meets requirements defined for it. It helps to build your confidence about your system.
- In TDD more focus is on production code that verifies whether testing will work properly. In traditional testing, more focus is on test case design.
- In TDD, you can achieve 100% coverage test. Every single line of code is tested, unlike traditional testing.

Demo – TDD With JUnit

JUnit

- A framework for writing tests for java code
  rather than just writing test functions within your code
- First-class support from many popular IDEs:
  IDEA, Eclipse, NetBeans, and Visual Studio Code.
- It is added to the project and code is written in separate files.
- This means that your software code is not full of testing code, which would make it messier and more confusing
  and harder for someone else to read / maintain.
- We will use Junit 5
  JUnit 5 = JUnit Platform + JUnit Jupiter (Junit-5 engine) + JUnit Vintage (Junit-4-engine)

General Approach

- Create an empty class (with stub methods)
- Write the tests first (calling those stubs)
- Run Test - IMPORTANT! Check that the tests fail
  - Write code for the first test
  - “just enough code to pass the test”
- Only move on when the tests for that code pass

Demo

- Given hourly wage, calculate month wage
  - Assume 160 working hours / month
  - Hourly wage is 50
  - Total Monthly wage = 8000

Red

```
import org.junit.jupiter.api.Test;

public class MonthlySalaryCalculatorTest {
    @Test
    void
testCalculateMonthSalary_GivenHourlySalary50_Return8000(){
        // Arrange
        SalaryCalculator sc = new SalaryCalculator();

        // Act
        double monthlySalary = sc.GetMonthlySalary(50);

        // Assert
        Assertions.assertEquals(8000, monthlySalary);
    }
}
```

Red to Green

```
public class SalaryCalculator {
    public double GetMonthlySalary(int i) {
        return 0;
    }
}
```

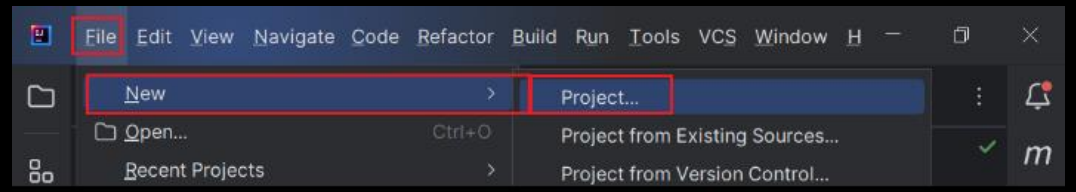
```
public class SalaryCalculator {
    private final int HOURS_IN_MONTH = 160;
    public double GetMonthlySalary(int i) {
        return i*HOURS_IN_MONTH;
    }
}
```

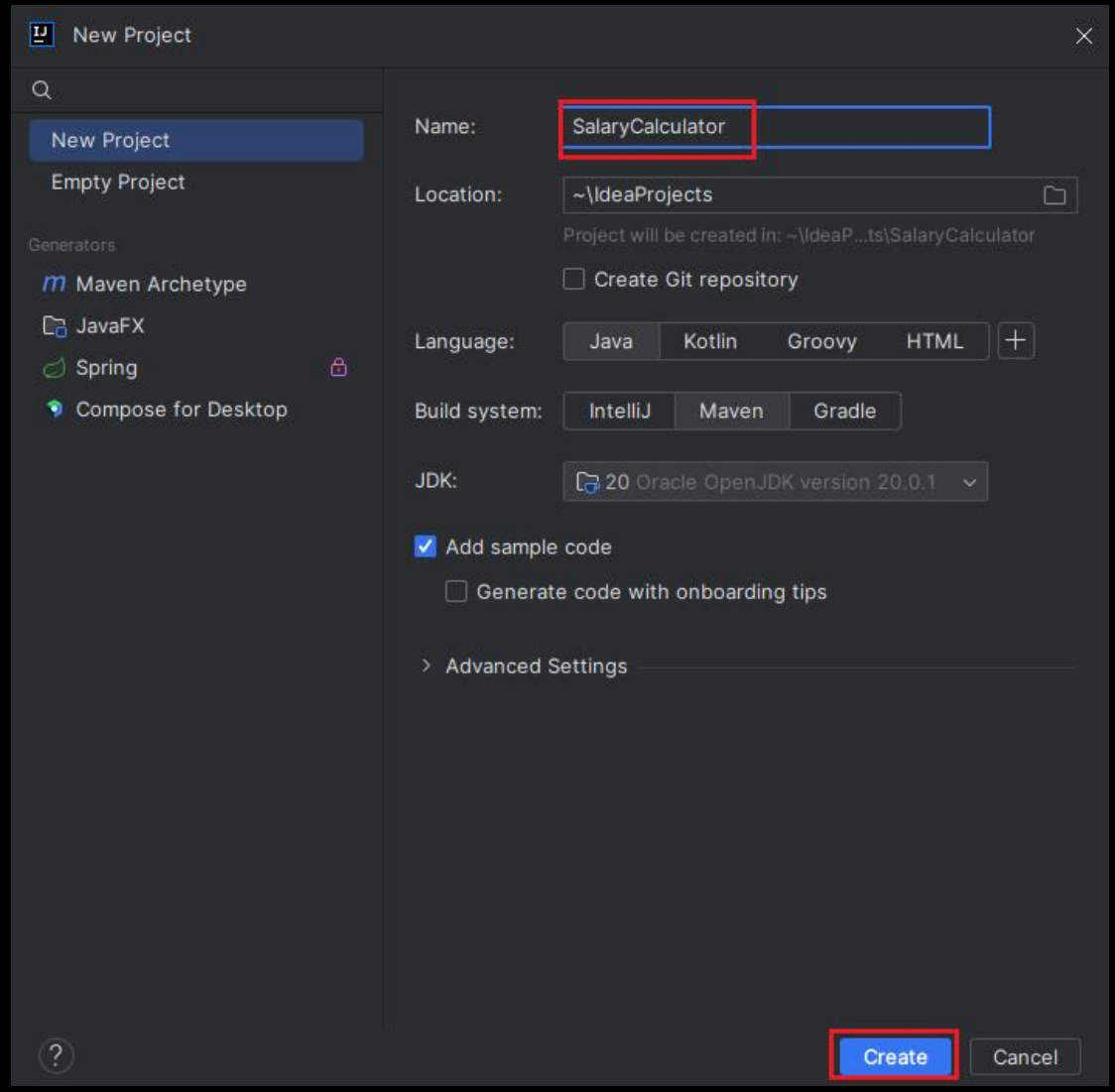
Refactor

```
public class SalaryCalculator {
    private final int HOURS_IN_MONTH = 160;
    public double GetMonthlySalary(double hourlySalary) {
        return hourlySalary*HOURS_IN_MONTH;
    }
}
```

University of Nottingham
UK | CHINA | MALAYSIA

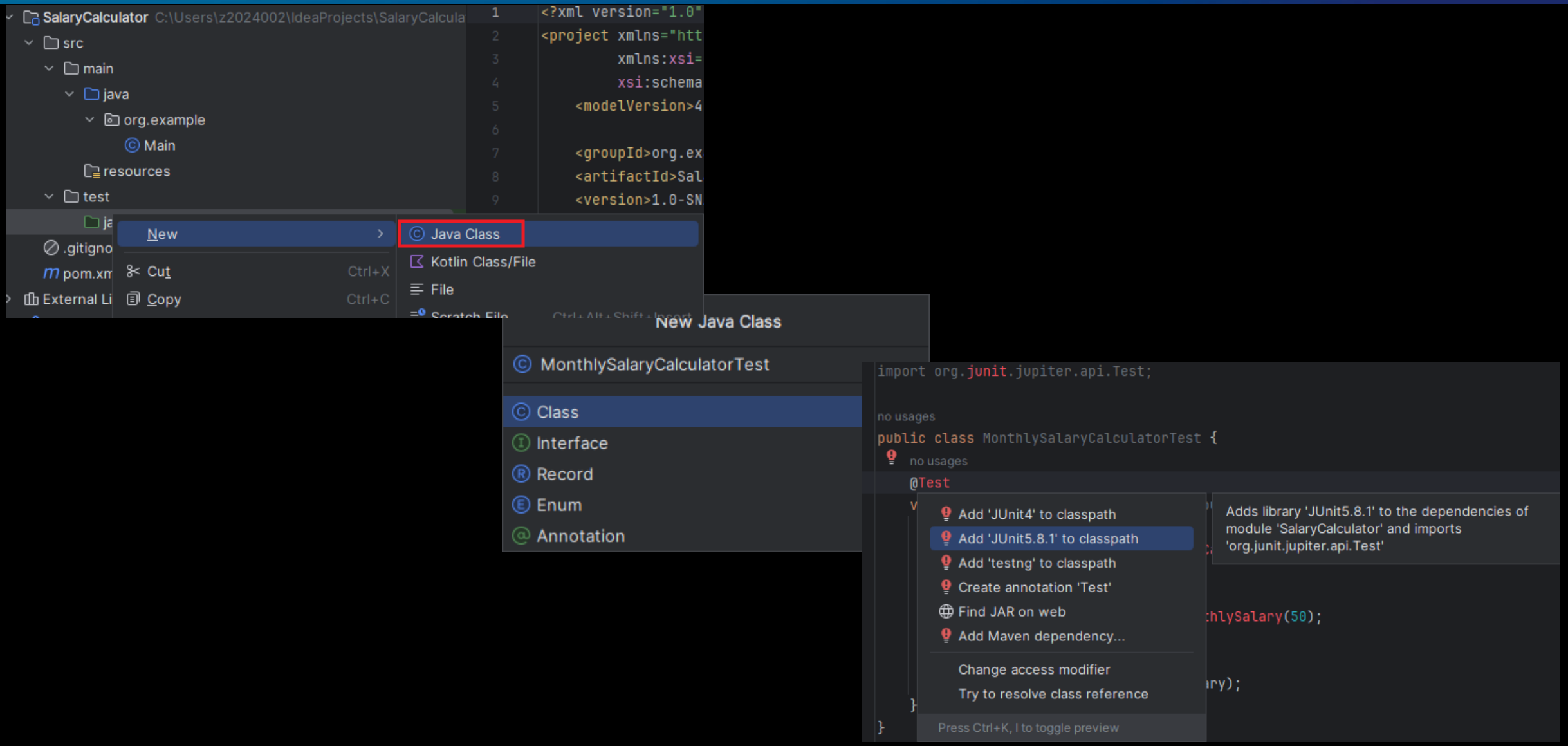
Demo-Recap





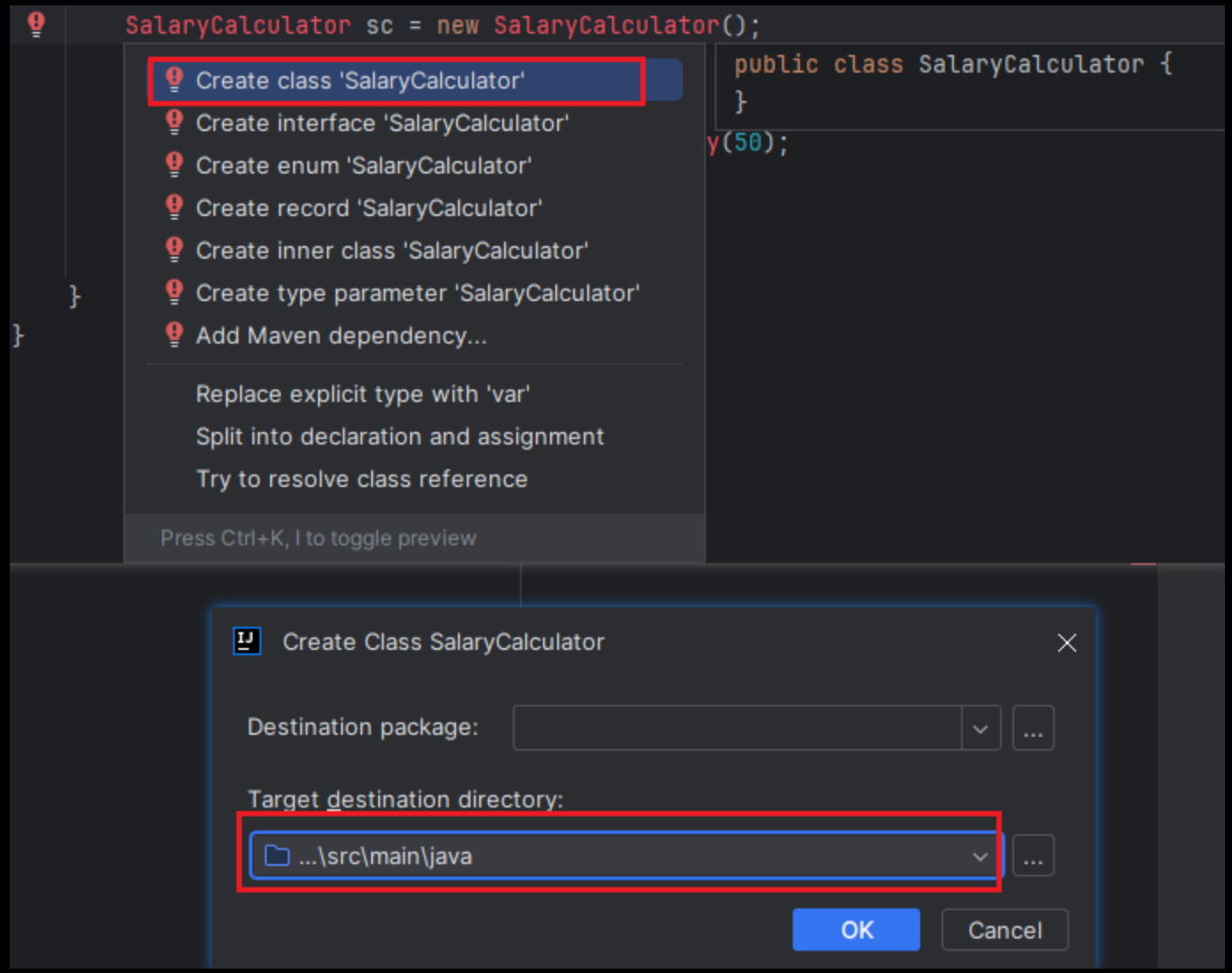
University of Nottingham
UK | CHINA | MALAYSIA

Demo-Recap



University of Nottingham
UK | CHINA | MALAYSIA

Demo-Recap



Demo-Recap

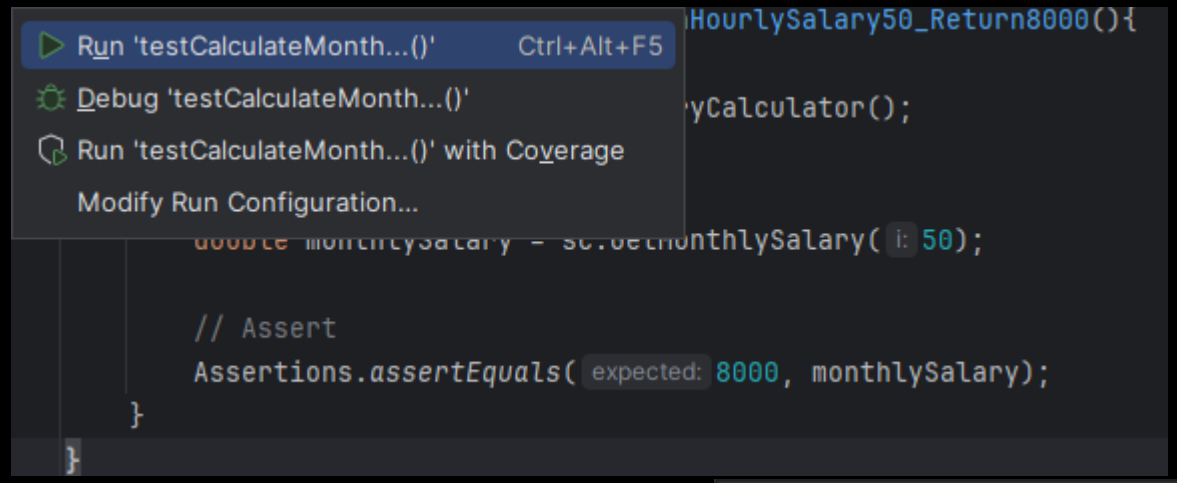
```
public class MonthlySalaryCalculatorTest {
    @Test
    void testCalculateMonthSalary_GivenHourlySalary50_Return8000(){
        // Arrange
        SalaryCalculator sc = new SalaryCalculator();

        // Act
        double monthlySalary = sc.GetMonthlySalary(50);

        // Assert
        Assert.AreEqual(8000, m
    }
}
```

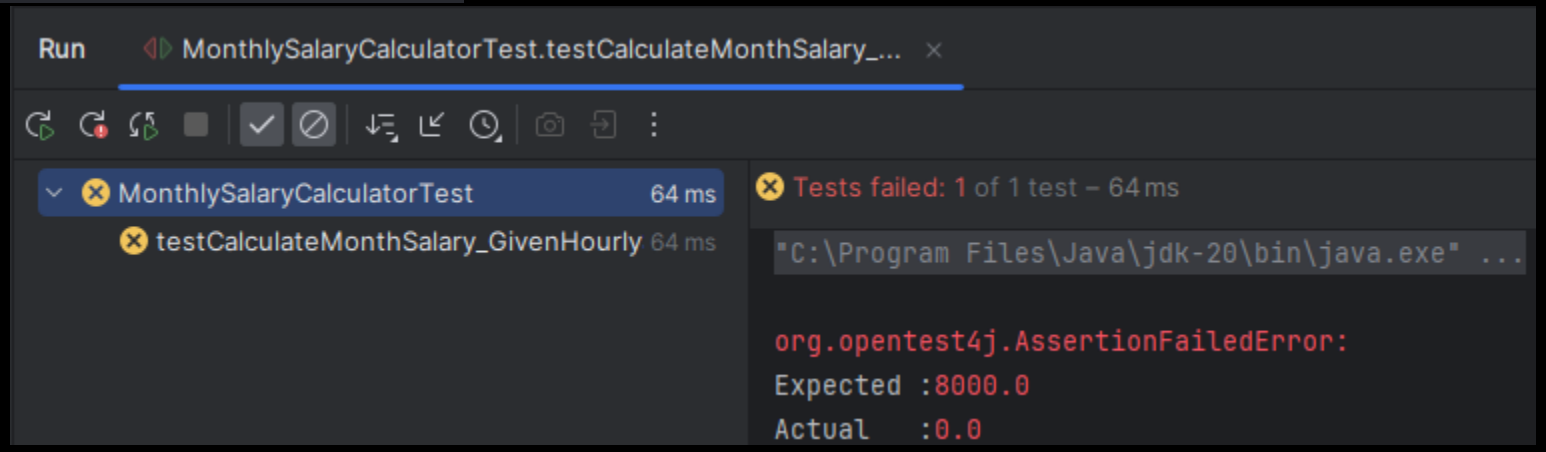
```
Create method 'GetMonthlySalary' in 'SalaryCalculator'
Rename reference
Split into declaration and assignment
Press Ctrl+K, I to toggle preview
```

```
// SalaryCalculator.java
public double GetMonthlySalary(int i) {
    return 0;
}
```

```
public class SalaryCalculator {
    1 usage
    public double GetMonthlySalary(int i) {
        return 0;
    }
}
```

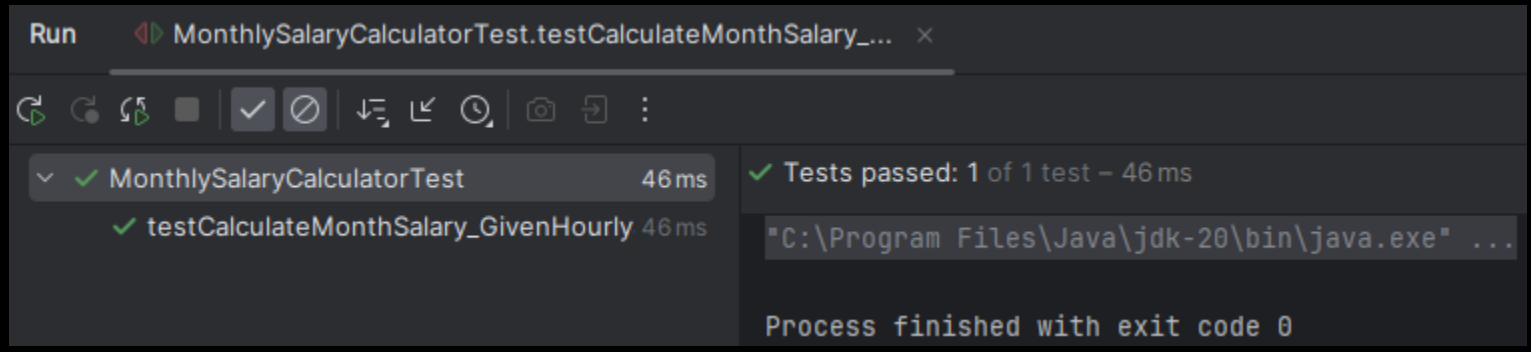
Demo-Recap


Run 'testCalculateMonth...()' Ctrl+Alt+F5
Debug 'testCalculateMonth...()'
Run 'testCalculateMonth...()' with Coverage
Modify Run Configuration...
HourlySalary50_Return8000(){
yCalculator();
onthlySalary(i: 50);
// Assert
Assertions.assertEquals(expected: 8000, monthlySalary);
}
}


Run
MonthlySalaryCalculatorTest.testCalculateMonthSalary_...
MonthlySalaryCalculatorTest 64 ms
testCalculateMonthSalary_GivenHourly 64 ms
Tests failed: 1 of 1 test – 64 ms
"C:\Program Files\Java\jdk-20\bin\java.exe" ...
org.opentest4j.AssertionFailedError:
Expected :8000.0
Actual :0.0

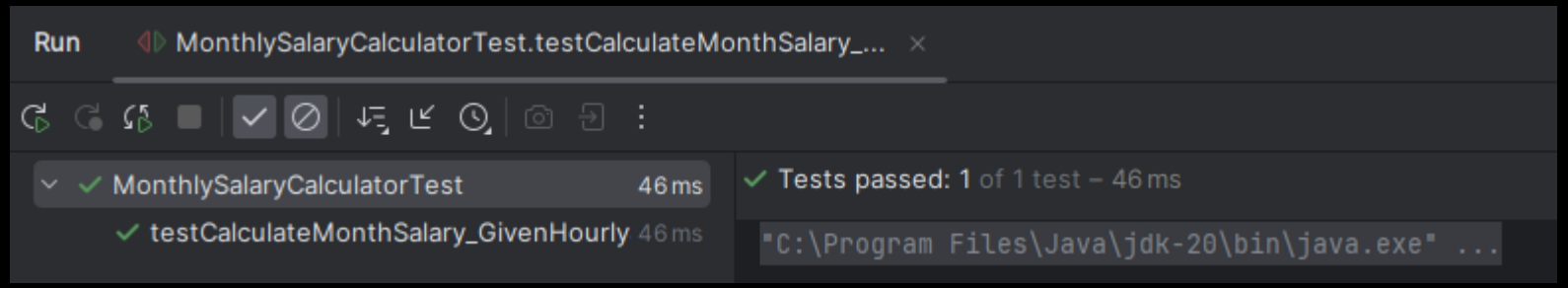
Demo-Recap

```java
public class SalaryCalculator {
    1 usage
    public double GetMonthlySalary(int i) {
        return i*160;
    }
}
```



Demo-Recap

```java
public class SalaryCalculator {
    1 usage
    private final int HOURS_IN_MONTH = 160;
    1 usage
    public double GetMonthlySalary(double hourlySalary) {
        return hourlySalary*HOURS_IN_MONTH;
    }

}
```



JUnit5 Annotations

https://junit.org/junit5/docs/current/user-guide/#overview

@Test - Denotes that a method is a test method

@BeforeEach - Denotes that the annotated method should be executed before each @Test, @RepeatedTest, @ParameterizedTest, or @TestFactory method in the current class (**execute regularly**)

@AfterEach - Denotes that the annotated method should be executed after each @Test, @RepeatedTest, @ParameterizedTest, or @TestFactory method in the current class (**execute regularly**)

@BeforeAll - Denotes that the annotated method should be executed before all @Test, @RepeatedTest, @ParameterizedTest, and @TestFactory methods in the current class (**execute once**)

@AfterAll - Denotes that the annotated method should be executed after all @Test, @RepeatedTest, @ParameterizedTest, and @TestFactory methods in the current class (**execute once**)

@Disabled - Used to disable a test class or test method

```
class StandardTests {

    @BeforeAll
    static void initAll() {
    }

    @BeforeEach
    void init() {
    }

    @Test
    void succeedingTest() {
    }

    @Test
    void failingTest() {
        fail("a failing test");
    }

    @Test
    @Disabled("for demonstration purposes")
    void skippedTest() {
        // not executed
    }

    @Test
    void abortedTest() {
        assumeTrue("abc".contains("Z"));
        fail("test should have been aborted");
    }

    @AfterEach
    void tearDown() {
    }

    @AfterAll
    static void tearDownAll() {
    }

}
```

Testing Metrics

- **Code Coverage** - to verify the extent to which the code has been executed.
- **Test Coverage** – to monitor the number of tests that have been executed.

Code Coverage (Statement Coverage)

- Ensure that each code statement is executed once.

Source code

```
Prints (int a, int b) {                          ------------  Prin
tsum is a function
    int result = a+ b;
    If (result> 0)
        Print ("Positive", result)
    Else
        Print ("Negative", result)
    }                                            -----------   End o
f the source code
```

```
1  Prints (int a, int b) {
2  int result = a+ b;
3  If (result> 0)
4      Print ("Positive", result)
5  Else
6      Print ("Negative", result)
7  }
```

Senario1 : a=3,b=9
Number of executed statements 5.
Number of total statements 7.
Statement coverage = 5/7.

```
1  Prints (int a, int b) {
2  int result = a+ b;
3  If (result> 0)
4      Print ("Positive", result)
5  Else
6      Print ("Negative", result)
7  }
```

Senario2 : a=-3,b=-9
Number of executed statements 6.
Number of total statements 7.
Statement coverage = 6/7.

Senario1 & Senario2
Statement coverage = 7/7=100%.

Code Coverage (Condition Coverage)

- Ensure that each Boolean sub-expression is executed once.

Source code

```
Prints (int a, int b) {                          ------------  Prin
tsum is a function
    int result = a+ b;
    If (result> 0)
        Print ("Positive", result)
    Else
        Print ("Negative", result)
    }                                            -----------   End o
f the source code
```

```
1  Prints (int a, int b) {
2  int result = a+ b;
3  If (result> 0)
4      Print ("Positive", result)
5  Else
6      Print ("Negative", result)
7  }
```

Senario1 : a=3,b=9
The first condition result>0 is fulfilled.
Condition coverage = 1/2.

```
1  Prints (int a, int b) {
2  int result = a+ b;
3  If (result> 0)
4      Print ("Positive", result)
5  Else
6      Print ("Negative", result)
7  }
```

Senario2 : a=-3,b=-9
The second condition result<0 is fulfilled.
Condition coverage = 1/2.

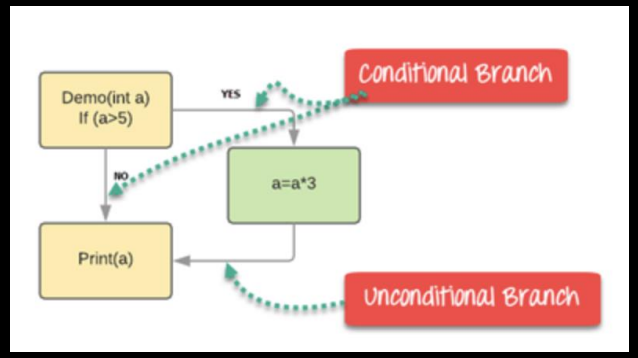
Senario1 & Senario2
Condition coverage = 2/2=100%.

Code Coverage (Branch Coverage)

- Ensure that each branch of each control structure is executed once.

Source code

```
Demo(int a) {
     If (a> 5)
        a=a*3
     Print (a)
    }
```



Senario1 : a=6
The condition (a>5) is fulfilled.
path coverage = 2/3=67%.

Senario2 : a=3.
The condition (a>5) is not met.
path coverage = 1/3=33%.

Senario1 & Senario2
Path coverage = 3/3=100%.

Testing Coverage

- Simple case:
  - If there are 100 test cases and 30 of those are executed, test coverage = 30/100
- Associated with requirement:
  - If there are 10 requirements and 100 tests created – when these 100 tests target all of the 10 requirements and don't leave out any, test coverage = 10/10
  - When only 80 of the created tests are executed and target only 6 of the requirements, test coverage = 6/10
  - When only 90 tests relating to 8 requirements are executed and the rest of them are not, test coverage = 8/10

Good Practices for TDD

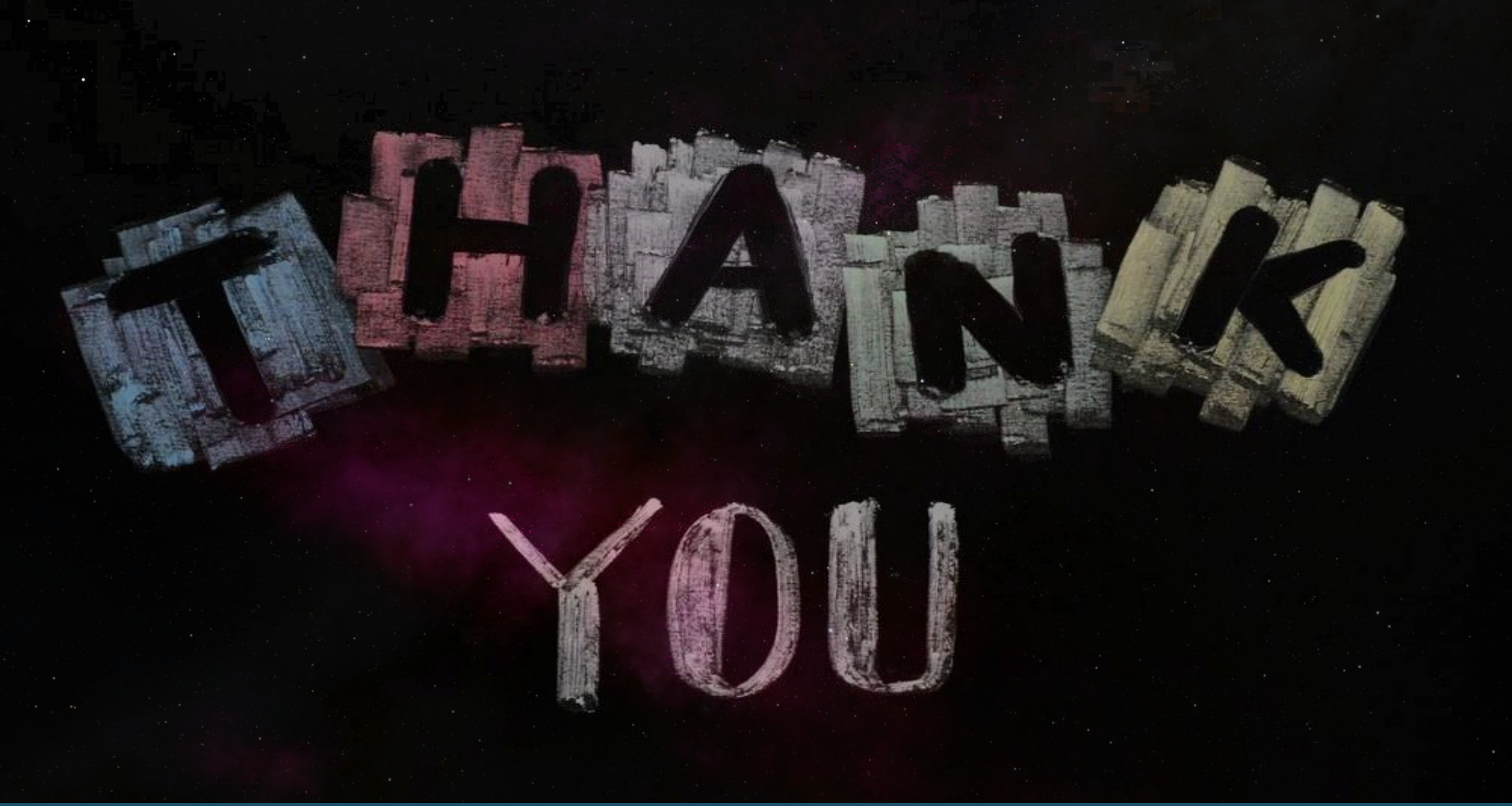
- Use descriptive names for test methods.
- Separate the implementation from the testing code.
- Place test classes in the same package as implementation.
- Write the simplest code to pass the test.
- Write assertions first (check certain conditions to be true)
- Minimize assertions in each test.
- Do not introduce dependencies between tests.
- Tests should run fast.

Unit Testing In Other Language

- These concepts exist for all sorts of languages
  - Python – PyUnit
  - C++ - CppTest (and many many more)
  - easy to do in Visual Studio etc
- https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

Summary

- Testing Methods
  - White Box Testing, Black Box Testing
- Testing Processes
  - Automation, manual
- Test Driven Development
- Junit
- Testing Metrics
  - Code Coverage, Test Coverage


THANK
YOU