ITSS SOFTWARE DEVELOPMENT
Lab 08 – Unit Testing

Lecturer: NGUYEN Thi Thu Trang, trangntt@soict.hust.edu.vn

1. SUBMISSION GUIDELINE

You are required to push all your work to the valid GitHub repository complying with the naming convention:

"<MSTeamName>-<StudentID>.<StudentName>".

For this lab, you have to turn in your work twice before the following deadlines:

- **Right after class**: Push all the work you have done during class time to Github.
- **10 PM the day after the class**: Create a branch named "***release/lab08***" in your GitHub repository and push the full submission for this lab, including in-class tasks and homework assignments, to this branch.

2. IN-CLASS ASSIGNMENT

In this section, we will get familiar with the software construction process and try ourselves with unit testing for the Case Study.

The first three subsections would give you an overview about unit testing, test-driven development, and JUNIT. After that, you will practice them in the last subsection 2.4. You would need Excel (to design test cases), JUNIT5 (already included in Eclipse IDE), Oracle JDK 11, and then import the given sample project[1].

You are asked to work individually for this section, and then put all your design of unit test (Excel file) to a directory, namely "UnitTest", and put the codes in "Programming" directiory. After that, push your commit to your individual repository before the announced deadline.

[1] https://github.com/trangntt-for-student/AIMS

2.1. UNIT TESTING

Testing plays a crucial role in software development and helps to determine whether a property of the program holds or not.

A well-tested program will have tests for every individual module (where a module is a method or a class) that it contains. A test that tests an individual module, in isolation if possible, is called a unit test. Testing modules in isolation leads to much easier debugging. When a unit test for a module fails, you can be more confident that the bug is found in that module, rather than anywhere in the program[2].

Thus, unit testing is neither suitable for testing complicated user interface nor the interaction among great modules/subsystems.

2.2. TEST DRIVEN DEVELOPMENT (TDD) [3]

Test-driven development (TDD), which is rooted in extreme programming, is all about satisfying your team that the code works as expected for a behavior or use case. Instead of aiming for the optimum solution in the first pass, the code and tests are iteratively built together one use case at a time. Development teams use TDD as part of many coding disciplines to ensure test coverage, improve code quality, set the groundwork for their delivery pipeline, and support continuous delivery.

[2] http://web.mit.edu/6.031/www/sp17/classes/03-testing/#automated_unit_testing_with_junit
[3] https://www.ibm.com/garage/method/practices/code/practice_test_driven_development/

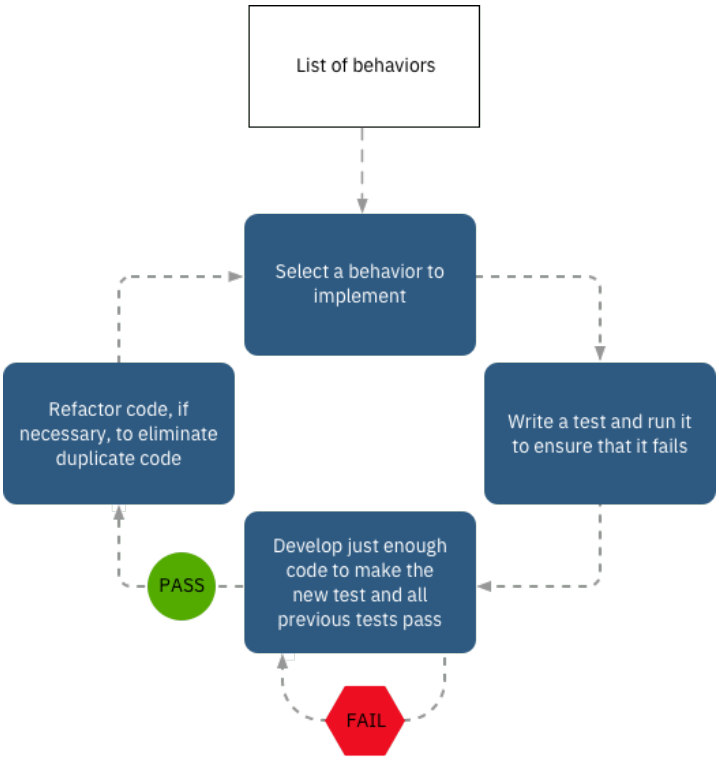


Figure 1- Red/Green/Refactor cycle

In the article *TDD - What it is and what it is not*[4], Andrea Koutifaris describes this cycle:

- Red phase: You write an automated test for a behavior that you're about to implement. Based on the user requirement, you decide how you can write a test that uses a piece of code as if it were implemented. This is a good opportunity to think about the externals of the code, without getting distracted by actually filling in the implementation. Think about what the interface should look like? What behaviours should a caller of that interface expect? Is the interface clean and consumable?
- Green phase: You write production code, but only enough production code to pass the test. You don't write algorithms, and you don't think about performance. You can duplicate code and even violate best practices. By addressing the simplest tasks, your code is less prone to errors and you avoid winding up with a mix of code: some tested (your minimalist functions) and some untested (other parts that are needed later).
- Refactor phase: You change the code so that it becomes better. At a minimum, you remove code duplication. Removing duplication generally leads to abstraction. Your specific code become more general. The unit tests

[4] https://medium.freecodecamp.org/test-driven-development-what-it-is-and-what-it-is-not-41fa6bca02a2

provide a safety net which supports the refactoring, because they make sure that the behavior stays the same and nothing breaks. In general, tests should not need to be changed during the refactor stage.

TDD drives the code development, and every line of code has an associated test case, so unit testing is integrated into the practice. Unit testing is repeatedly done on the code until each unit functions per the requirements, eliminating the need for you to write more unit test cases.

You can get started with TDD by following these steps:

1. Think about the behaviors that your implementation requires. Select a behavior to implement.
2. Write a test that validates the behavior. The test case must fail.
3. Add only enough code to make the new test case and all previous test cases pass.
4. Refactor the code to eliminate duplicate code, if necessary.
5. Select the next requirement and repeat steps 1 - 4.

2.3. UNIT TESTING WITH JUNIT5

2.3.1. Automated Unit Testing with JUnit[5]

JUnit is a widely-adopted Java unit testing library, and we will use it heavily in this course. A JUnit unit test is written as a method preceded by the annotation @Test. A unit test method typically contains one or more calls to the module being tested, and then checks the results using assertion methods like assertEquals, assertTrue, and assertFalse.

For example, the tests we chose for Math.max() above might look like this when implemented for JUnit:

```
@Test
public void testALessThanB() {
    assertEquals(2, Math.max(1, 2));
}
@Test
public void testBothEqual() {
    assertEquals(9, Math.max(9, 9));
}
@Test
public void testAGreaterThanB() {
```

[5] http://web.mit.edu/6.031/www/sp17/classes/03-testing/#automated_unit_testing_with_junit

```
    assertEquals(-5. Math.max(-5, -6));
}
```

Note that the order of the parameters to assertEquals is important. The first parameter should be the expected result, usually a constant, that the test wants to see. The second parameter is the actual result, what the code actually does. If you switch them around, then JUnit will produce a confusing error message when the test fails. All the assertions supported by JUnit follow this order consistently: expected first, actual second.

If an assertion in a test method fails, then that test method returns immediately, and JUnit records a failure for that test. A test class can contain any number of @Test methods, which are run independently when you run the test class with JUnit. Even if one test method fails, the others will still be run.

2.3.2. Documenting Your Testing Strategy[6]

Let consider a function that reverses the end of a string.

```
/**
 * Reverses the end of a string.
 *
 * For example:
 *   reverseEnd("Hello, world", 5)
 *   returns "Hellodlrow ,"
 *
 * With start == 0, reverses the entire text.
 * With start == text.length(), reverses nothing.
 *
 * @param text    non-null String that will have
 *                its end reversed
 * @param start   the index at which the
 *                remainder of the input is
 *                reversed, requires 0 <=
 *                start <= text.length()
 * @return input text with the substring from
 *               start to the end of the string
 *               reversed
 */
static String reverseEnd(String text, int start)
```

[6] http://web.mit.edu/6.031/www/sp17/classes/03-testing/#automated_unit_testing_with_junit

For example, at the top of the class, we can document the testing strategy we worked on in the partitioning exercises above. The strategy also addresses some boundary values we did not consider before.

```
/*
 * Testing strategy
 *
 * Partition the inputs as follows:
 * text.length(): 0, 1, > 1
 * start:         0, 1, 1 < start < text.length(),
 *                text.length() - 1, text.length()
 * text.length()-start: 0, 1, even > 1, odd > 1
 *
 * Include even- and odd-length reversals because
 * only odd has a middle element that doesn't move.
 *
 * Exhaustive Cartesian coverage of partitions.
 */
```

Each test method should also need a comment above it saying how its test case was chosen, i.e. which parts of the partitions it covers:

```
// covers test.length() = 0,
//        start = 0 = text.length(),
//        text.length()-start = 0
@Test public void testEmpty() {
    assertEquals("", reverseEnd("", 0));
}
```

2.3.3. Using annotations[7]

JUnit 5	Description
@Test	The annotated method is a test method.
@BeforeAll	The annotated (static) method will be executed once before any @Test method in the current class.
@BeforeEach	The annotated method will be executed before each @Test method in the current class.
@AfterEach	The annotated method will be executed after each @Test method in the current class.

[7] https://developer.ibm.com/tutorials/j-introducing-junit5-part1-jupiter-api/

JUnit 5	Description
@AfterAll	The annotated (static) method will be executed once after all @Test methods in the current class.
@Disabled	The annotated method will not be executed (it will be skipped), but reported as such.

Figure 2-Important Annotations in JUNIT5

2.4. UNIT TESTING AS PART OF TDD

2.4.1. Test case specification

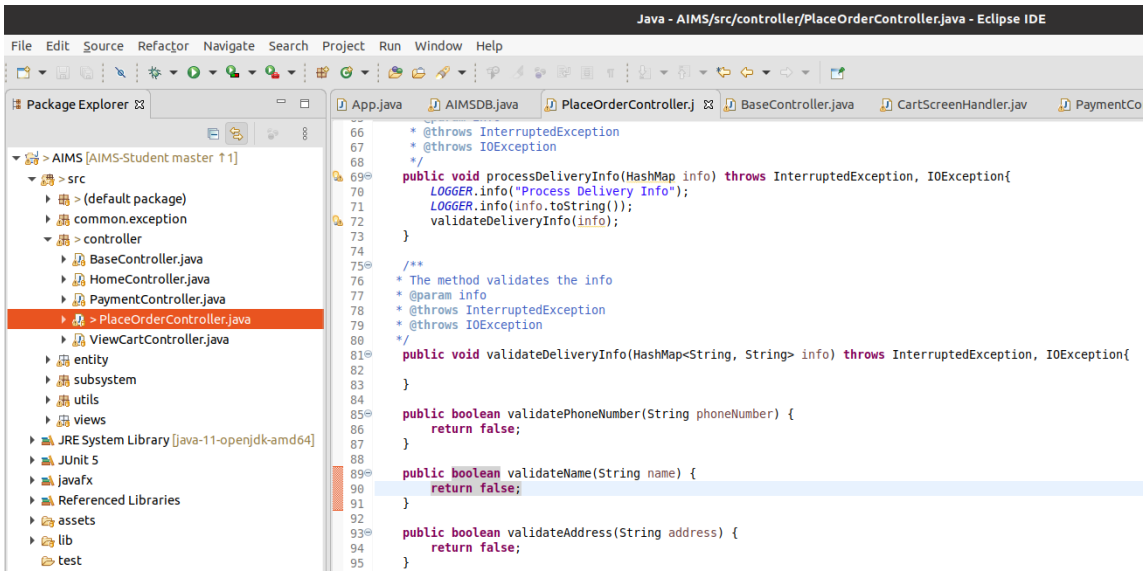
In this part, we would design test cases for the method `validateDeliveryInfo()` in class `PlaceOrderController`.

The method `validateDeliveryInfo()` requires customer's name, phone number, and address as the input.

#	Parameter	Conditions
1	Name	- Only letters (a-z and A-Z) are allowed - Must not null
2	Phone	- Only numbers (0-9) are allowed - Must have 10-character length - Start with 0
3	address	- Only letters (a-z and A-Z) are allowed - Must not null

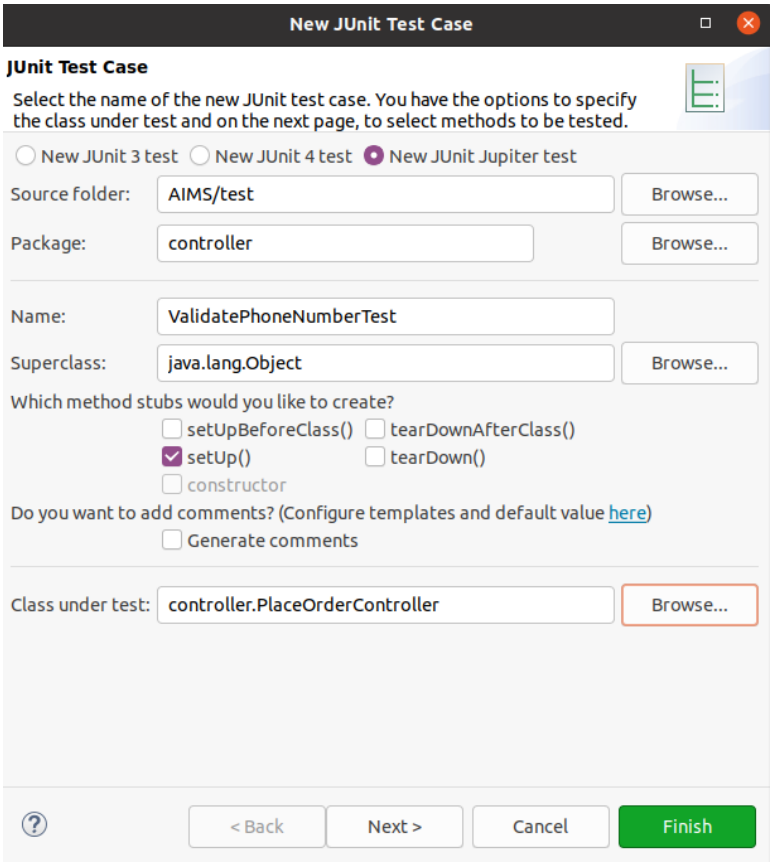
Based on SRS and SDD, we design unit tests and develop the method `validateDeliveryInfo()` by applying TDD.

We divide the method `validateDeliveryInfo()` into 3 three methods: `validateAdress()`, `validateName()` ,and `validatePhoneNumber()`. Initially, all these methods are empty.



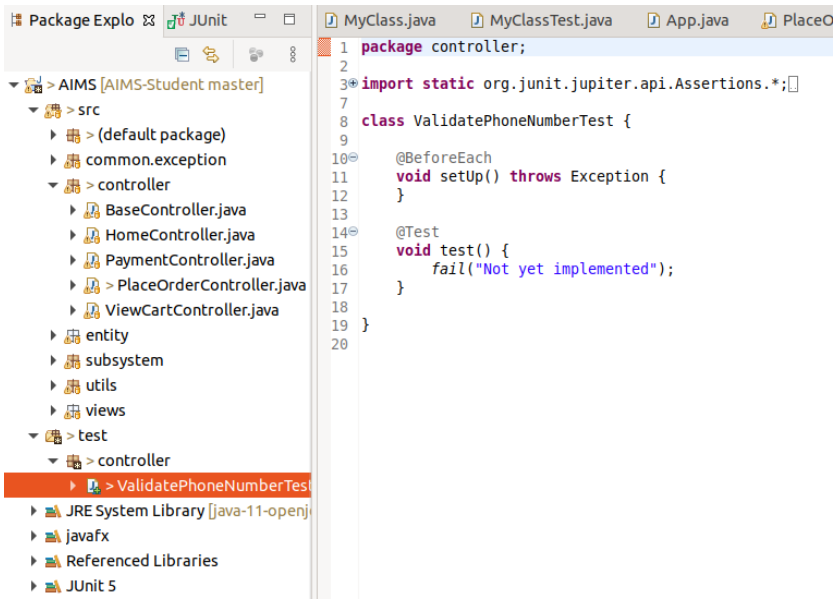
2.4.2. Creating unit test class

To illustrate, we create a new class for the testing the method `validatePhoneNumber()`.



You may be asked to import JUNIT5 library as follows.

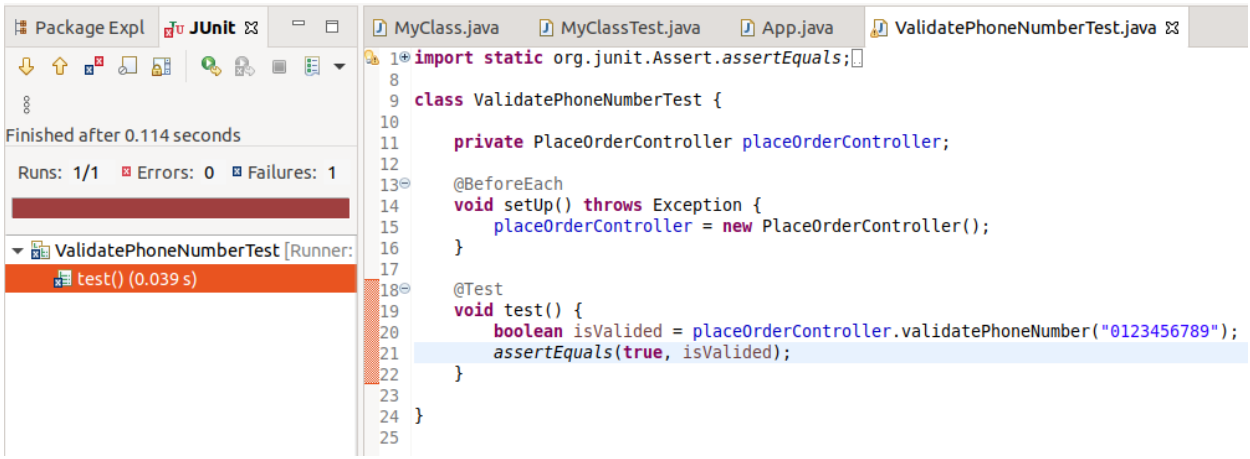
The result is shown as follows.



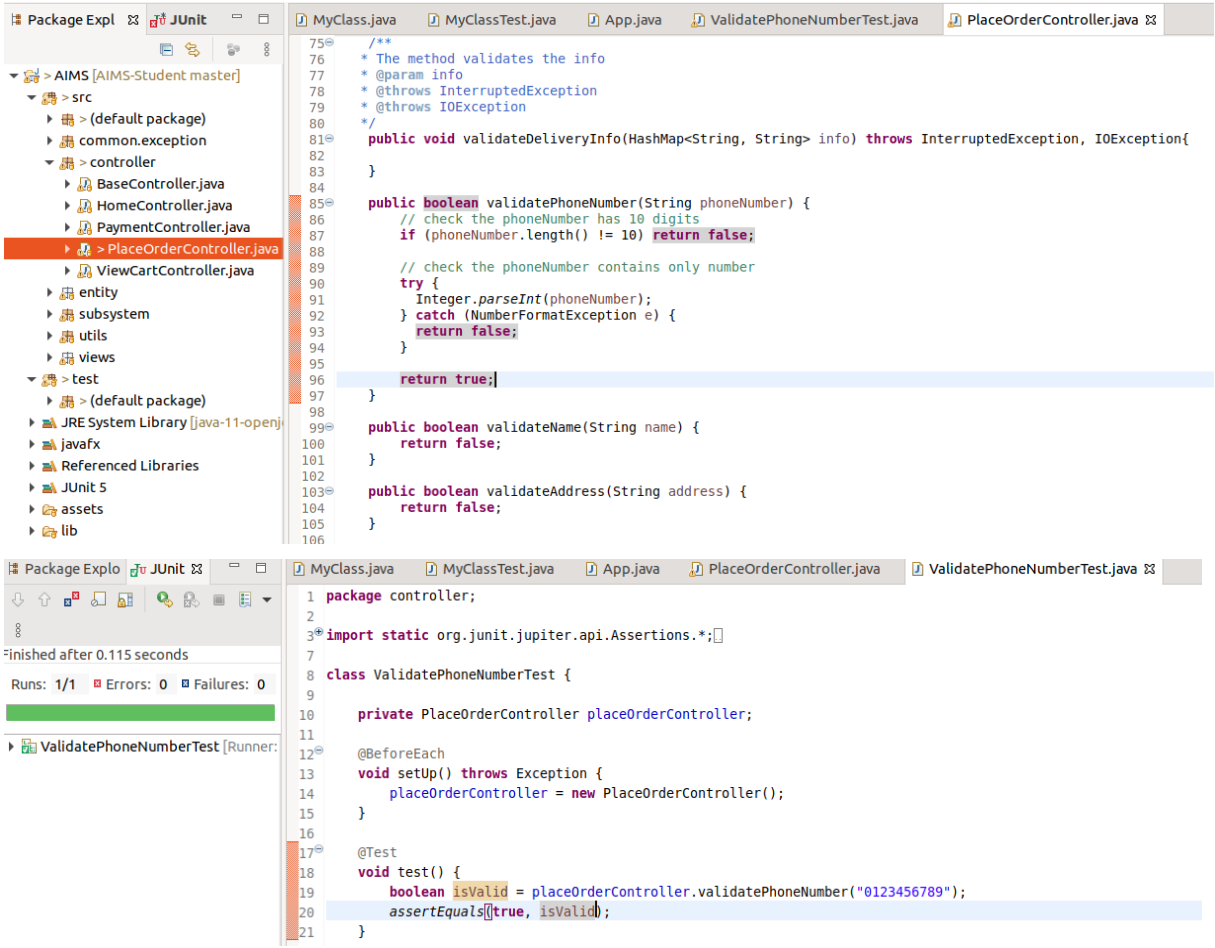
Obviously, if we run the test now, the test will be failed since we have not implemented anything yet.

2.4.3. Development with TDD

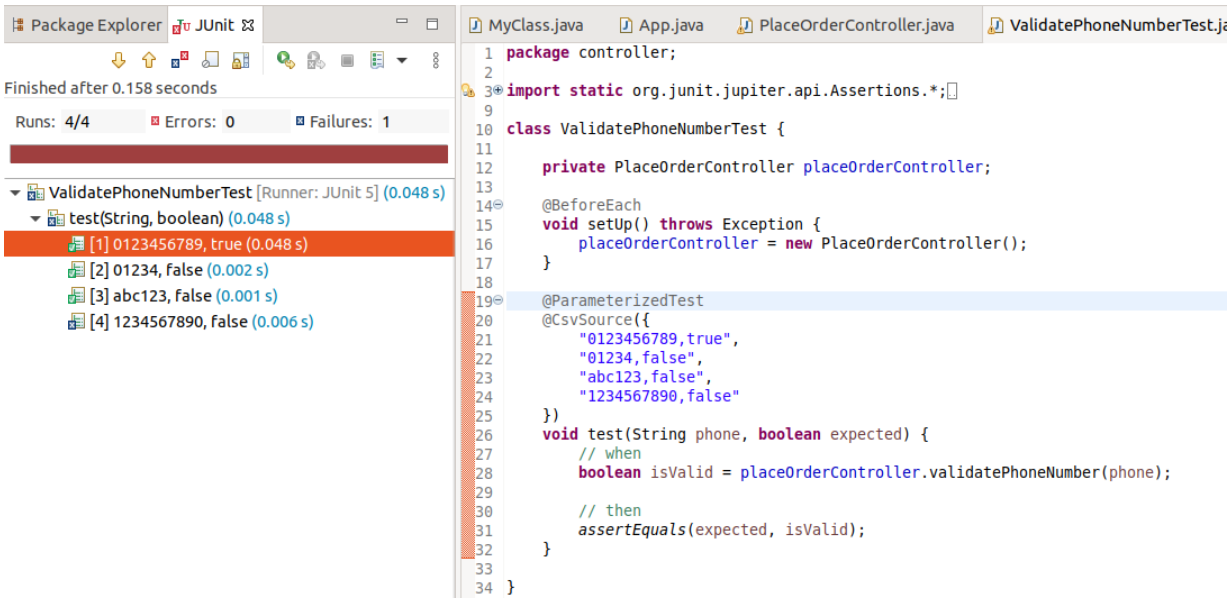
Now, if we create an object of `PlaceOrderController` class and then try testing the method `validatePhoneNumber()` with a valid phone number, e.g., 0123456789, the test will be failed since the method return false by default.



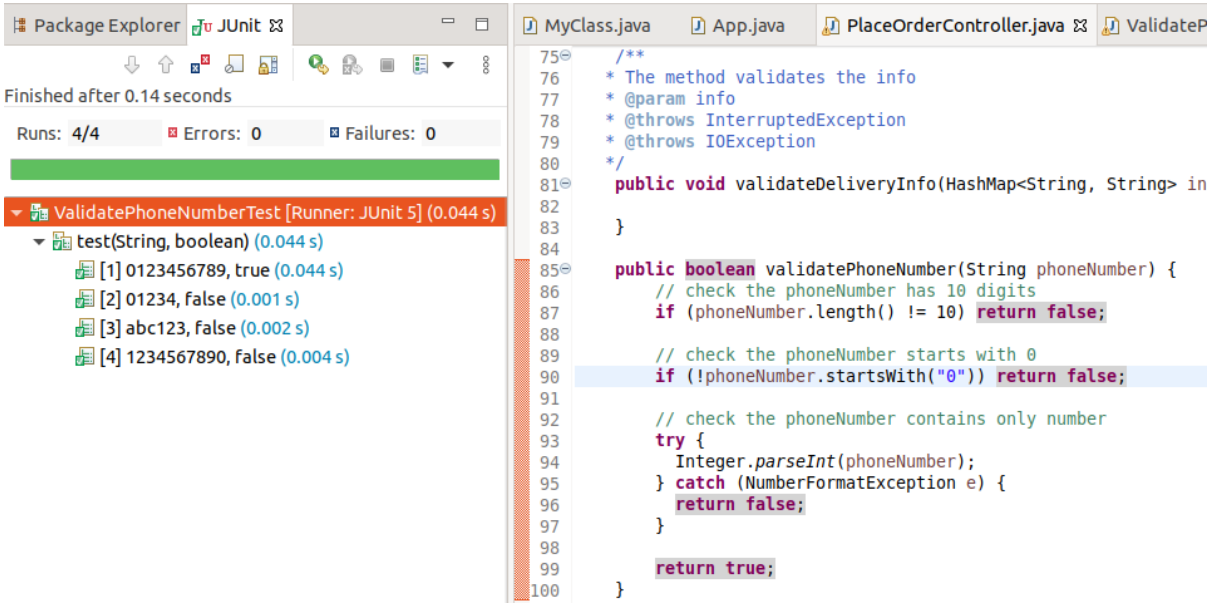
Thus, we need to go back to `PlaceOrderController` class to implement `validatePhoneNumber()` method. After that, the test case for this method should pass.



We can put a list of input-expected output pairs to test at once by using annotations @Parameterized and @CsvSource. After that, run the test again.



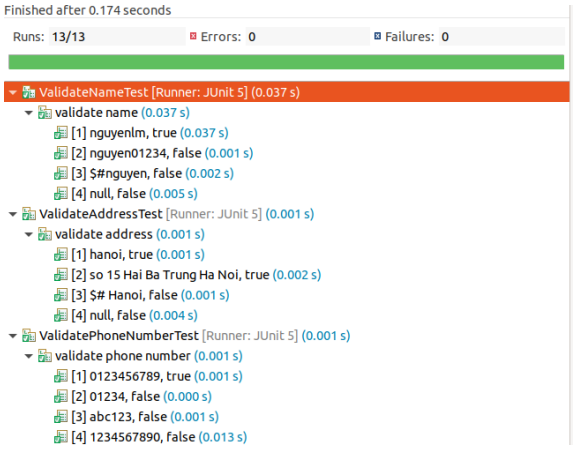
You can see that there is a failed test case: the one with the input-output pair "1234567890,false". We need to go back to the method and add the validation codes for the phone number which must start with 0. At last, we have all passed test cases.



2.4.4. Working with test suite

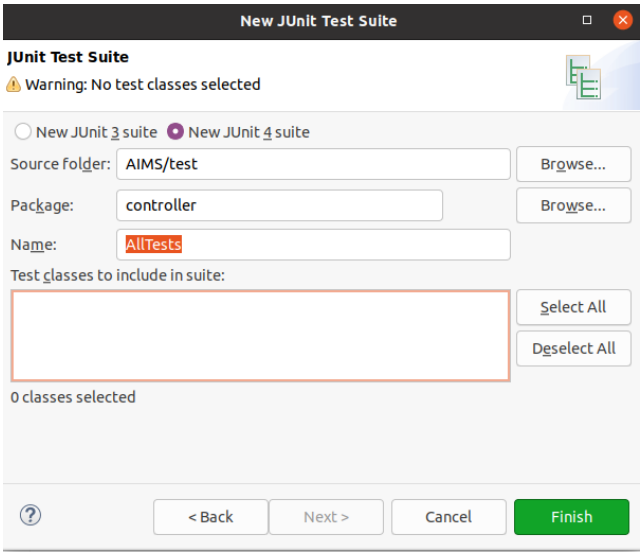
You are asked to implement 2 methods `validateAdress()` and `validateName()` by yourself.

After you finish the task, we would have 3 testing classes. Eclipse can run all the three test cases at once. The result is shown as follows.

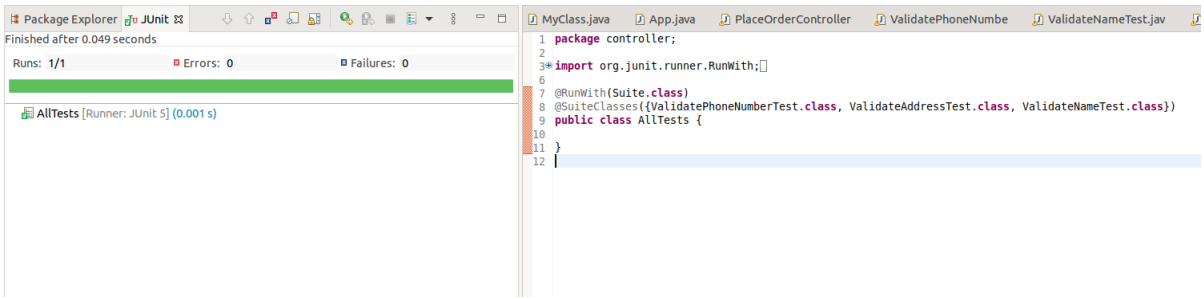


In another hand, these three test cases could make a test suite. A test suite is a collection of test cases related to the same test work.

Right click on the project -> New -> Test Suite



Add all classes that we need to test in the test suite into class AllTest and then run.



2.5. TDD AND UC "PLACE RUSH ORDER"

In this part, you are asked to apply TDD to implement your design for UC "Place Rush Order."