Lab Report for SQAT

# Lab01: JUnit 4 for Unit Test

## Tasks

### Task 01: Terminology Illustration

Please illustrate the following Terminology about Unit Test:

* + - 1. What’s test suite?

*A test suite is a collection of test cases.*

* + - 1. What’s test case?

A test case tests the response of a single method to a particular set of inputs.

* + - 1. What’s unit test?

A unit test is a test of the smallest element of code you can sensibly test, usually a single class.

* + - 1. What’s test fixture?

A test fixture is the environment in which a test is run. A new fixture is set up before each test case is executed, and torn down afterwards.

Example: if you are testing a database client, the fixture might place the database server in a standard initial state, ready for the client to connect.

* + - 1. **Annotations**. JUnit 4.0 uses **annotations** rather than special names for setting up, tearing down and testing. Please list the mainly used **annotations** in JUnit 4.0.

@before, @after, @Ignore, @Test

* + - 1. **Assert** is very important for Making Tests. Please read and run the test codes, CalculatorTest and TriangleTest, in *LectureCodes/src/lec01*, and illustrate why? That is, if we don’t use Assert functions, what will happen in our testing code?

Assert functions simplify our testing codes greatly. If we do not use Assert functions, we have to write if statements by ourselves to check whether the testing results are expected.

* + - 1. **Assert Statement**. There are two forms of the assert statement. Please illustrate what they are.

There are two forms of the assert statement:

– assert boolean condition ;

– assert boolean condition : error message ;

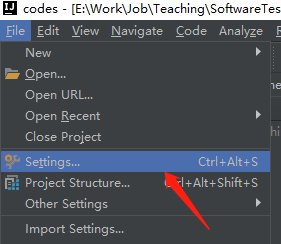
Both forms throw an AssertionFailedError if the boolean condition is false. The second form, with an explicit error message, is seldom necessary.

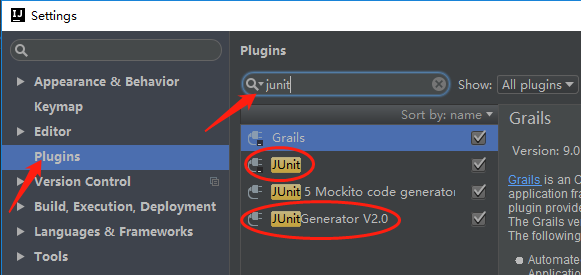
### Task 02: Set up JUnit in Eclipse / IDEA

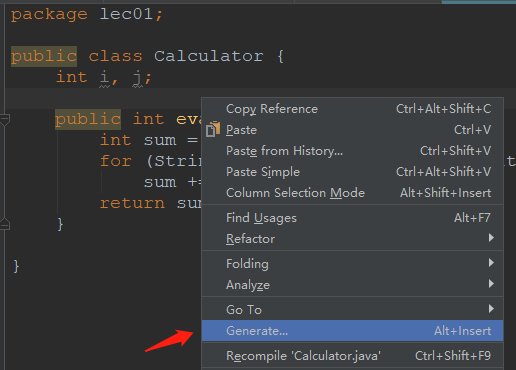
Set up JUnit 4.0 in Eclipse / IDEA, and record your setting procedure by text description and screen snapshot. Please check your setting by running the test codes, CalculatorTest and TriangleTest in LectureCodes/src/lec01.

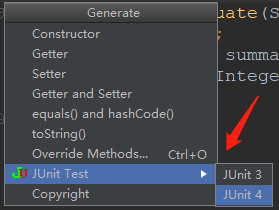
*In the following, we only give the answer in IDEA. For Eclipse users, I strongly suggest you using IDEA.*

*For ease of using JUnit 4, we need install both JUnit 4.0 and JUnitGenerator 2.0. Actually, IDEA has integrated JUnit 4 since 2017. The following snapshots show how to check if JUnit 4 is installed in your IDEA and how to install and use JUnit 4.*









### Task 03: Build your first JUnit testing function

Please add the following method “int sub(int i, int j)” into the class [Cal](src/Lab01/task03/Cal.java), and then add a new testing method in [CalTest](src/Lab01/task03/CalTest.java) class by referring to its testAdd method.

public int sub(int i, int j) {

return i - j;

}

Please refer to the answer: [ANSWER](src/Lab01/task03/answer)

### Task 04: Test your own code with JUnit 4.0

Basic Employee Compensation Problem. For each week, hourly employees are paid a standard wage per hour for the first 40 hours worked, 1.5 times their wage for each hour after the first 40 hours, and 2 times their wage for each hour worked on Sundays and Holidays. Table 1 gives some test cases of this.

Please write a Java class, WageCalculator, to solve the wage problem in the following and a test class WageCalculatorTest to test your code by using the test cases in Table 1.

Table 1 Test Cases for Basic Employee Compensation Problem

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input** | | | **Expected** | **Testing Result** |
| StandardHours | HolidayHours | HourlyWage |
| 40 | 0 | $20 | $800 | Pass |
| 45 | 0 | $20 | $950 | Pass |
| 48 | 8 | $20 | $1280 | **Fail**  Actual: $1360 |

N/A now.

# Lab02: JUnit 5 for Unit Test

**Note: in this labwork, all of the related codes are in Labwork/Lab02/src.**

## Target

* To know about **Maven** and Maven-based Project
* To understand how to **add dependencies** by Maven
* To understand **Parameterized Test**
* To understand **Timeout Testing** and **Exception Test**

## Tools

* IDE: [Eclipse](https://www.eclipse.org/) / [IntelliJ IDEA](file:///E:\Work\Job\Teaching\SoftwareTest(overseas)\Experiments\Lab1\IntelliJ%20IDEA) / any IDE you’d like to use
* Programming Language: Java

## Tasks

### Task 01: Config JUnit 5 in a Maven Project

All things are difficult before they become easy! In this subtask, you almost do nothing except for configuring JUnit 5. However, it might be a little challenge! I spent a lot of time!

IntelliJ IDEA supports running tests on the JUnit Platform since version 2016.2. However, only IDEA 2017.3 or newer versions of IDEA will download the following JARs automatically based on the API version used in the project: *junit-platform-launcher*, *junit-jupiter-engine*, and *junit-vintage-engine*. In addition, [Parameterized Tests](https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests), which are very important in JUnit testing, are not automatically supported by any version of IDEA. You need to add a *dependency* on the *junit-jupiter-params* artifact.

In this subtask, I provided you a *well-configured* Maven project, which added all of the JUnit 5 *dependencies* we’ll use in the following tasks. Please do the following subtasks to test if your JUnit 5 packages are well set in your IDE. In the following, I suppose you use IDEA, instead of Eclipse.

Find the ***settings.xml*** of your Maven, which lies in your Maven install path,like %IDEA\_INSTALL\_PATH%/plugins/maven/lib/maven3/conf/settings.xml. **Revise <mirrors> as follows**:

<mirrors>

<mirror>

<id>nexus-aliyun</id>

<mirrorOf>\*</mirrorOf>

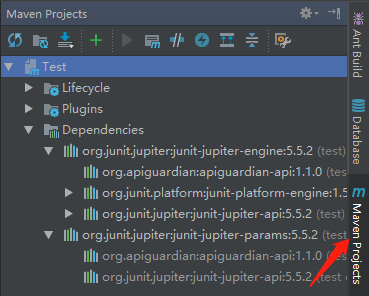
<name>Nexus aliyun</name>

<url>http://maven.aliyun.com/nexus/content/groups/public</url>

</mirror>

</mirrors>

Restart IDEA and check if the JUnit 5 dependencies in your Maven project are well-installed. If well-installed, there will be no any error tips as follows:



To test if your JUnit 5 can well work, please run the test method, testAppleColor, in *task01.AppleTest*. If you get the following result, congratulations, you make it!



Now, please illustrate where we add the JUnit 5 dependencies in this project? Why we manage these dependencies by maven, rather than manually? Please give your answer in detail.

Please create a new Maven project as follows and add the JUnit 5 dependencies where you found in the previous step and copy them to your new project. And also, copy the two classes, Apple and AppleTest, to your new project. Finally, try to run testAppleColor in AppleTest again and snapshot the result.

### Task 02: Timeout Testing and Exception Test

Both *Timeout Testing* and *Exception Test* are very important. The class, *task02.Calculator*, has two methods, *squareRoot* and *divide*. In the following, we have 2 subtasks to perform *Timeout Testing* and *Exception Test* on the above 2 methods. Please do them carefully.

***Timeout Testing***. In 2 scenarios, we need to use Timeout Testing. ***First***, it can be used for ***performance test*** to ensure the method is returned within a reasonable time. ***Second***, it is advisable to use Timeout Testing when we write test cases with external dependencies, which never give a 100% certainty that they will be available while executing the test cases.

JUnit 5 provides us with two ***Timeout Assertions*** (see [here](https://howtodoinjava.com/junit5/junit-5-assertions-examples/#assertTimeout-assertTimeoutPreemptively)), which are used to test long running tasks. If given task inside testcase takes more than specified duration, then test will fail.

**Subtask 01**: The method, *squareRoot*, is expected to be returned within 1 second. However, due to having an endless loop, *squareRoot* will never finish. Please choose a right assertion to make a Timeout Testing for *squareRoot*. Give your answer in the testSquareRoot method in task02.CalculatorTest.

***Exception Test***. Some test cases are used to test the execution of the supplied Executable throws an exception of the *expectedType* and returns the exception.

**Subtask 02**: Dividing by zero will lead to an *ArithmeticException*. It means the *divide* method in task02.Calculator has a bug if it does not throw an *ArithmeticException* in this scenario. Please write a test method to check if the *divide* method can throw an *ArithmeticException* when the *divisor* parameter is zero. Give your answer in the testDivide method in task02.CalculatorTest.

Please refer to the answer: [ANSWER](src/Lab02/task02/answer)

### Task 03: Parameterized Test

[This video](https://youtu.be/srJ91NRpT_w) introduced the roman numeral problem. We provide its implementation in *task03.RomanNumeral* and its corresponding test class in *task03.RomanNumeralTest*. The method, *singleDigit*, tries to check the 7 roman numerals, I, V, X, L, C, D, and M, can be correctly mapped by *singleDigit* to their corresponding Arabic numbers, 1, 5, 10, 50, 100, 500, and 1000. However, the testing codes seem very duplicated and repetitive. Thus, it’s very difficult to perform large-scale test cases in this way. Fortunately, we can greatly simplify the testing codes by using ***Parameterized Test*** provided by JUnit 5. Please refer to user guide [here](https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests) to rewrite the test method *singleDigit* by using *Parameterized Test* in *task03.RomanNumeralParamTest*.

Please refer to the answer: [ANSWER](src/Lab02/task03/answer)

# Lab03: Testing Principles

## Exercise 01: failure, fault, or error

Having a certain terminology helps testers to explain the problems they have with a program or in their software. Below is a small conversation. Fill each of the caps with: failure, fault, or error.

**Mark**: Hey, Jane, I just observed a (1) \_ \_ \_ \_ \_ \_ in our software: if the user has multiple surnames, our software doesn’t allow them to sign in.

**Jane**: Oh, that’s awful. Let me debug the code so that I can find the (2) \_ \_ \_ \_ \_ \_.

*(a few minutes later)*

**Jane**: Mark, I found it! It was my (3) \_ \_ \_ \_ \_ \_. I programmed that part, but never thought of this case.

**Mark**: No worries, Jane! Thanks for fixing it!

Your Answer is ?

1. Failure, the user notices the system/program behaving incorrectly.
2. Fault, this is a problem in the code, that is causing a failure in this case.
3. Error, the human mistake that created the fault.

## Exercise 02: testing principle

Kelly, a very experienced software tester, visits Books!, a social network focused on matching people based on books they read. Users do not report bugs so often; Books! developers have strong testing practices in place. However, users do say that the software is not really delivering what it promises. What testing principle applies to this problem?

Your Answer is ?

*The absence-of-errors fallacy. While the software does not have a lot of bugs, it is not giving the users what they want. In this case the verification was good, but they need work on the validation.*

## Exercise 03: testing principle

Suzanne, a junior software testing, just joined a very large online payment company. As a first task, Suzanne analyzed their past two years of bug reports. Suzanne observes that more than 50% of bugs have been happening in the ‘International payments’ module.

Suzanne then promises her manager that she will design test cases that will completely cover the ‘International payments’ module, and thus, find all the bugs.

Which of the following testing principles might explain why this is not possible?

1. Pesticide paradox.
2. Exhaustive testing.
3. Test early.
4. Defect clustering.

Your Answer is ?

*2. Exhaustive testing.*

## Exercise 04： only unit testing?

John strongly believes in unit testing. In fact, this is the only type of testing he actually does at any project he’s in. All the testing principles below, but one, might help in convincing John that he should also focus on different types of testing.

Which of the following is the least related related to help John in moving away from his ‘only unit testing’ approach?

1. Pesticide paradox.
2. Tests are context-dependent.
3. Absence-of-errors fallacy.
4. Test early.

Your Answer is ?

Test early, although an important principle, is definitely not related to the problem of only doing unit tests. All others help people in understanding that variation, different types of testing, is important.

# Lab04: Specification-Based Testing

Note: all of the following 7 exercises can be found [here](https://mordeky.github.io/SQAT/chapters/testing-techniques/specification-based-testing.html#exercises). You just need to report your answer in the corresponding subsection.

## Exercise 01

A group of inputs that all make a method behave the same way.

## Exercise 02

We use the concept of equivalence partitioning to determine which tests can be removed. According to equivalence partitioning we only need to test one test case in a certain partition.

We can group the tests cases in their partitions:

* Divisible by 3 and 5: T1, T2
* Divisible by just 3 (not by 5): T4
* Divisible by just 5 (not by 3): T5
* Not divisible by 3 or 5: T3

Only the partition where the number is divisible by both 3 and 5 has two tests. Therefore we can only remove T1 or T2.

## Exercise 03

Following the category partition method:

1. Two parameters: key and value
2. The execution of the program does not depend on the value; it always inserts it into the map. We can define different characteristics of the key:
   * The key can already be present in the map or not.
   * The key can be null.
3. The requirements did not give a lot of parameters and/or characteristics, so we do not have to add constraints.
4. The combinations are each of the possibilities for the key with any value, as the programs execution does not depend on the value. We end up with three partitions:
   * New key
   * Existing key
   * null key

## Exercise 04

We go over each given partition and identify whether it is a valid partition:

1. Valid partition: Invalid numbers, which are too small.
2. Valid partition: Valid numbers
3. Invalid partition: Contains some valid numbers, but the range is too small to cover the whole partition.
4. Invalid partition: Same reason as number 3.
5. Valid partition: Invalid numbers, that are too large.
6. Invalid partition: Contains both valid and invalid letters (the C is included in the domain).
7. Valid partition: Valid letters.
8. Valid partition: Invalid letters, past the range of valid letters.

We have the following valid partitions: 1, 2, 5, 7, 8.

## Exercise 05

* P1: Element not present in the set
* P2: Element already present in the set
* P3: NULL element.

The specification clearly makes the three different cases of the correct answer explicit.

## Exercise 06

Option 4 is the incorrect one. This is a functional based technique. No need for source code.

## Exercise 07

Possible actions:

1. We should treat pattern size 'empty' as exceptional, and thus, test it just once.
2. We should constrain the options in the 'occurrences in a single line' category to happen only if 'occurrences in the file' are either exactly one or more than one. % It does not make sense to have none occurrences in a file and one pattern in a line.
3. We should treat 'pattern is improperly quoted' as exceptional, and thus, test it just once.

# Lab05: Boundary Testing

Note: all of the following exercises can be found [here](https://mordeky.github.io/SQAT/chapters/testing-techniques/boundary-testing.html#exercises). You just need to report your answer in the corresponding subsection. Please notice the exercise numbers are 01, 03, 05, and 07.

## Exercise 01

The on-point is the value in the conditions: half.

When i equals half the condition is false. Then the off-point makes the condition true and is as close to half as possible. This makes the off-point half - 1.

The in-points are all the points that are smaller than half. Practically they will be from 0, as that is what i starts with.

The out-points are the values that make the condition false: all values equal to or larger than half.

## Exercise 03

The on-point can be read from the condition: 570.

The off-point should make the condition false (the on-point makes it true): 571.

An example of an in-point is 483. Then the condition evaluates to true.

An example of an out-point, where the condition evaluates to false, is 893.

## Exercise 05

An on-point is the (single) number on the boundary. It may or may not make the condition true. The off point is the closest number to the boundary that makes the condition to be evaluated to the opposite of the on point. Given it's an inequality, there's only a single off-point.

## Exercise 07

We should always test the behavior of our program when any expected data actually does not exist (EXISTENCE).

# Lab06: Structural-Based Testing

Note: all of the following exercises can be found [here](https://mordeky.github.io/SQAT/chapters/testing-techniques/structural-testing.html#exercises). You just need to report your answer in the corresponding subsection. Please notice the exercise numbers are 01, 03, 05, and 07.

## Exercise 01

Example of a test suite that achieves 100\%100% line coverage:

@Test

public void removeNullInListTest() {

LinkedList<Integer> list = new LinkedList<>();

list.add(null);

assertTrue(list.remove(null));

}

@Test

public void removeElementInListTest() {

LinkedList<Integer> list = new LinkedList<>();

list.add(7);

assertTrue(list.remove(7));

}

@Test

public void removeElementNotPresentInListTest() {

LinkedList<Integer> list = new LinkedList<>();

assertFalse(list.remove(5))

}

Note that there exists a lot of test suites that achieve 100\%100% line coverage, this is just an example.

You should have 3 tests. At least one test is needed to cover lines 4 and 5 (removeNullInListTest in this case). This test will also cover lines 1-3.

Then a test for lines 9 and 10 is needed (removeElementInListTest). This test also covers lines 6-8.

Finally a third test is needed to cover line 11 (removeElementNotPresentInListTest).

## Exercise 03

Option 1 is the false one.

A minimal test suite that achieves 100\% (either basic or full) condition has the same number of tests as a minimal test suite that achieves 100\% branch coverage. All decisions have just a single branch, so condition coverage doesn't make a difference here. Moreover, a test case that exercises lines 1, 6, 7, 8, 9, 10 achieves around 54\% coverage (6/11).

## Exercise 05

First, we find the pairs of tests that can be used for each of the conditions:

* A: {2, 6}
* B: {1, 3}, {2, 4}, {5, 7}
* C: {5, 6}

For A and C we need the decisions 2, 5 and 6. Then you can choose to add either 4 or 7 to cover condition B.

The possible answers are: {2, 4, 5, 6} or {2, 5, 6, 7}.

## Exercise 07

A lot of input strings give 100% line coverage. A very simple one is "aa". As long as the string is longer than one character and makes the condition in line 9 true, it will give 100% line coverage. For "aa" the expected output is "a".

# Lab07: Model-Based Testing

Note: all of the following exercises can be found [here](https://mordeky.github.io/SQAT/chapters/testing-techniques/model-based-testing.html#exercises). You just need to report your answer in the corresponding subsection. Please notice the exercise numbers are 08, and 09.

## Exercise 08

## Exercise 09

# Lab08: Design by Contracts

Note: all of the following exercises can be found [here](https://mordeky.github.io/SQAT/chapters/testing-techniques/design-by-contracts.html#exercises). You just need to report your answer in the corresponding subsection. Please notice the exercise numbers are 01, and 03.

## Exercise 01

## Exercise 03