
MAHARISHI UNIVERSITY of MANAGEMENT
Engaging the Managing Intelligence of Nature
Computer Science Department
CS401 Modern Programming Practices (MPP)
Professor Paul Corazza

Lecture 10:Best Programming Practices with Java 8

Living Life in Accord with Natural Law

REVIEW: TDD and Unit Testing

The Test-Driven Development (TDD) Paradigm
(Optional Module #1)

Executive Summary

Best practice during code development:

1. Always unit test your code
2. Even better – as you develop your code, develop unit tests for the code

The Test-Driven Development (TDD) Paradigm *(Optional Module #1) (cont.)*

1. The TDD paradigm says that the best testing strategy is to develop tests as part of the implementation process. Some even say that test code for a method or a class should be written before the actual code for the method or class is written. (TDD originated in the Extreme Programming development process, but is used these days regardless of the process used.)

2. TDD follows these steps
   a. First the developer writes an (initially failing) automated test case that defines a desired improvement or new function
   b. Next he produces the minimum amount of code to pass that test
   c. Finally, he refactors the new code to acceptable standards.

The Test-Driven Development (TDD) Paradigm *(Optional Module #1) (cont.)*

3. Benefits of TDD (See the Wikipedia article)
   A. Studies
      a. A 2005 study found that using TDD meant writing more tests and, in turn, programmers who wrote more tests tended to be more productive.
      b. Madeyski (Springer, 2010) provided an empirical evidence (involving 200 developers) that the TDD approach led to bettter OO design and more thorough and effective testing of a code base than the Test-Last approach.
      c. Surveys of developers (many examples of this) reveal that developers find significantly less need to debug code when they use TDD.
      d. Müller and Padberg ("About the Return on Investment of Test-Driven Development" – pdf available online) showed that, while it is true that more code is required with TDD, the total code implementation time is often shorter.

The Test-Driven Development (TDD) Paradigm *(Optional Module #1) (cont.)*

B. *Better handling of requirements.* Since coding is closely related to testing, developer tends to think more effectively about how the code will be used – the result is that requirements are met with at a higher success rate.

C. *Catch defects early* – this leads to a reduction in overall cost

D. *Modular, flexible, extensible code.* TDD leads to more modularized, flexible, and extensible code because developers think of the software in terms of small units that can be written and tested independently and integrated together later

4. Example (see package lesson10.lecture.tdd)

Best Practices When Unit-Testing

(Optional Module #1)

- Separate common set up logic into support methods.
- Keep each test focused on only the results necessary to do the necessary validation..
- Treat your test code with the same respect as your production code. It also must work correctly for both positive and negative cases, last a long time, and be readable and maintainable. In particular, tests should not depend on data or other aspects of the system that are likely to change (example: relying on the number of records in a table as part of a test)

Things to avoid

- Having test cases depend on system state manipulated from previously executed test cases.
- Dependencies between test cases. A test suite where test cases are dependent upon each other is brittle and complex. Execution order should not be presumed.
- Building "all-knowing oracles." An oracle that inspects more than necessary is more expensive and brittle over time.
- Slow running tests.

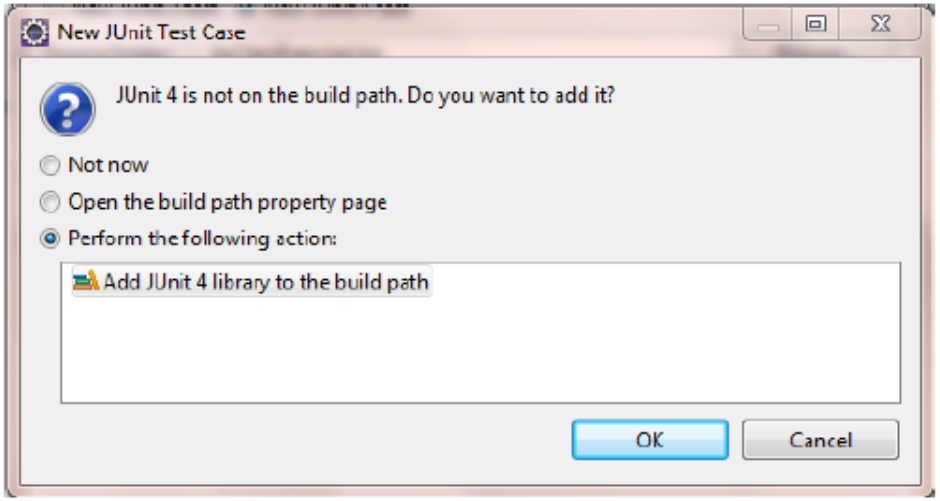
Unit-Testing Tools *(Optional Module #1)*

Tool	Description
Cactus	A JUnit extension for testing Java EE and web applications. Cactus tests are executed inside the Java EE/web container.
GrandTestAuto	Comprehensive Java software test tool not related to xUnit series of tools
Jtest	Commercial unit test tool that provides test generation and execution with code coverage and runtime error detection
JUnit 4.0	Standard unit test tool; more flexible with release of version 4.0
JUnitEE	A variant of JUnit that provides JEE testing
Mockito	Unit testing with mock objects
TestNG	Actually a multi-purpose testing framework, which means its tests can include unit tests, functional tests, and integration tests. Further, it has facilities to create even no-functional tests (as loading tests, timed tests). It uses Annotations since first version and is a framework more powerful and easy to use than the most used testing tool in Java: JUnit

Review: Setting Up JUnit 4.0

(Optional Module #1)

- Right click the default package where your Hello.java class is, and create a JUnit Test Case TestHello.java by clicking New→JUnit Test Case. (After we introduce the package concept, we will put the tests in a separate package.)
- Name it TestHello and click finish. You will see the following popup window.



- Select "Perform the following action" → Add JUnit 4 library to the build path → OK
- Then you will see that JUnit 4 has been added to the Java Build Path by right clicking your project name → properties → Java Build Path.

Review: Setting Up JUnit 4.0

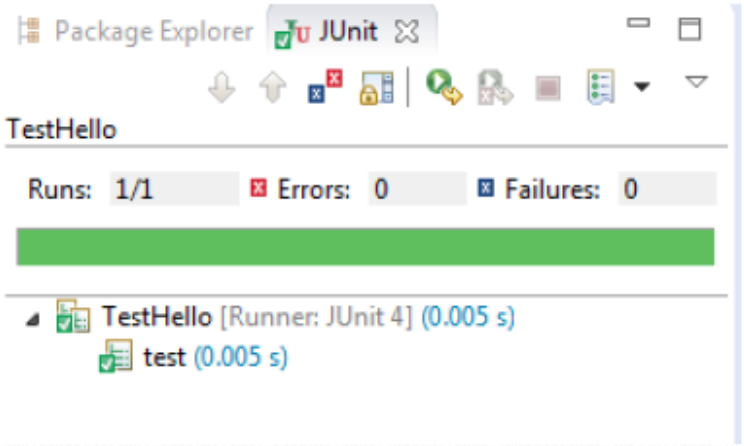
(Optional Module #1) (cont.)

- Create a testHello method in TestHello.java (see below) and remove the method test() that has been provided by default:

```
@Test
public void testHello() {
    assertEquals("Hello", Hello.hello());
}
```

If you have compiler error, you can hover over to the workspace where you have error and Eclipse will give you hints of what to do. (In this case, Add import 'org.junit.Assert.assertEquals'.)

- Run your JUnit Test Case by right clicking the empty space of the file → Run As → JUnit Test. If you see the result like below, it means you have successfully created your first unit test. ☺



Review: Setting Up JUnit 4.0

(Optional Module #1) (cont.)

- Methods that are annotated with @Test can be unit-tested with JUnit.
- Three most commonly used methods:

  assertTrue(String comment, boolean test)
  assertFalse(String comment, boolean test)
  assertEquals(String comment, Object ob1, Object ob2)

  In each case, when test fails, comment is displayed (so it should be used to say what the expected value was).
- Demo: TDD Example Using JUnit 4.0 – see lesson10.lecture.tddxxx

Main Point 1

Unit testing, in conjunction with Test-Driven Development, make it possible to steer a mistake-free course as programming code is developed. The self-referral mechanism of anticipating logic errors in unit tests, developed as the main code is developed, is analogous to the mechanism that leads awareness to be established in its self-referral basis; action from such an established awareness tends to make fewer mistakes.