TDD Standard Work

Author: Rakesh Kumar Sharma

This document assumes you have familiarity with writing JUnit test cases. If you don’t, the following tutorials are recommended:

1. JUnit - <https://www.tutorialspoint.com/junit/index.htm>
2. Hamcrest - <https://code.google.com/archive/p/hamcrest/wikis/Tutorial.wiki>
3. Mockito - <https://www.baeldung.com/mockito-series>
4. PowerMock - <http://www.baeldung.com/intro-to-powermock>
5. MockMVC - <https://www.petrikainulainen.net/programming/spring-framework/unit-testing-of-spring-mvc-controllers-normal-controllers/>

In addition, there are some common practices we will be following:

1. Within a Java module, the primary source folder is src/main/java. All tests should be placed in a separate source folder, src/test/java. Gradle uses this source folder by convention, so following this standard avoids having to customize build scripts.
2. Each class in src/main/java will have a corresponding class in src/test/java, with an identical package. The test class should be prefixed or suffixed with “Test”. For example:
   1. Source Folder: src/main/java Package: com.discovercard.cardmembersvcs.promotions.bo Class: PromotionsBO.java
   2. Source Folder: src/test/java Package: com.discovercard.cardmembersvcs.promotions.bo Class: TestPromotionsBO.java
3. Tests should be written with as few dependencies as possible. This includes testing frameworks. If it is possible to write a test without PowerMock, it should be written without PowerMock.
4. Tests for controllers should all use MockMVC.
5. Code reviews and demos should not be approved/accepted without unit tests. Incremental development will make this easier. If you are having trouble with this, try breaking stories down into smaller components.
6. Keep the following best practices in mind, and read up on other JUnit best practices. (See references at the bottom of this document.)
   1. Tests should run quickly, as they will run frequently.
   2. Tests should be meaningful and sufficient. A controller test was recently observed that was only asserting the pageName in the model was correct, when the test was added because of other new variables being added to the model. A test like that has little value. A better approach would have been to create an expected model, and a single assertion that compares the entire model object.
   3. Use appropriate assertion methods. For example, assertEquals(expected,actual) is generally better than assertTrue(expected==actual).

# TDD

The core concept of TDD is that development is performed iteratively with a specific loop that includes writing tests prior to writing code. A simplified way to view this iterative process is the following set of steps:

1. Add a test.
2. Run all tests. The new test should fail.
3. Write the code.
4. Run all tests. The new test should pass. If it doesn’t, go back to step 3.
5. Refactor.
6. Repeat.

The remainder of this document will illustrate how to apply this to common practical scenarios.

# Scenario #1: Adding a new method.

Let’s walk through, step by step, how to write a new method using TDD. We’ll make this new method something simple, formatting a date.

**Step 1:** Stub out the method you want to create. We will call this the “method under test.”

**public** **static** String formatDate(Date dateToFormat) {

**return** "";

}

**Step 2:** Stub out the test within your test class.

@Test

**public** **void** testFormatDate() {

}

**Step 3:** Fill in your test method.

@Test

**public** **void** testFormatDate() {

// Using a calendar object to initialize the date as the Date

// constructors are deprecated.

Calendar calendar = Calendar.*getInstance*();

calendar.setTimeInMillis(0);

calendar.set(2017, 00, 01, 00, 00, 00); //January 1st 2017 at midnight.

Date dateToTest = calendar.getTime();

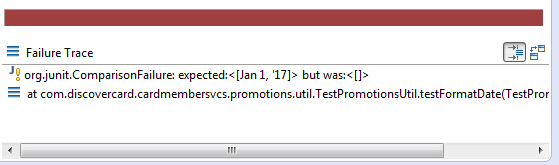
String testResultString = PromotionsUtil.*formatDate*(dateToTest);

//Business requirement is to display Mon DD, 'YY

*assertEquals*("Jan 1, '17", testResultString);

}

**Step 4:** Run the test. It will fail.



**Step 5:** Fill in the method under test.

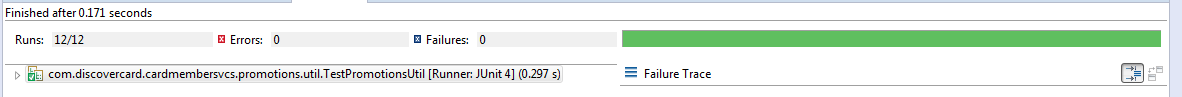
**private** **static final** SimpleDateFormat *USER\_FRIENDLY\_DATE\_FORMAT* = **new** SimpleDateFormat("MMM d, ''yy");

**public** **static** String formatDate(Date dateToFormat) {

**return** *USER\_FRIENDLY\_DATE\_FORMAT*.format(dateToFormat);

}

**Step 6:** Run the tests again. They should now pass. If they don’t, refine the method under test until they do.



Now pick the next requirement you want to work on and repeat the process. Let’s use error handling as an example of how to proceed. What do we do when the input date is null? Let’s say the business requirement is to display a friendly error in that case.

**Step 1:** Add another test. You could also add on to an existing test, but since we are adding a new negative scenario it’s better to make a distinct test.

@Test

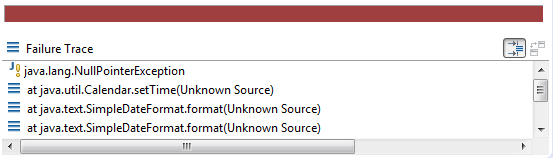
**public** **void** testFormatDate\_negativeTestNullInput() {

String testResultString = PromotionsUtil.*formatDate*(**null**);

*assertEquals*("Unknown Date", testResultString);

}

**Step 2:** Run the test, it will fail.



**Step 3:** Modify the method under test to pass the new test case, while not impacting the old test case.

**private** **static final** SimpleDateFormat *USER\_FRIENDLY\_DATE\_FORMAT* = **new** SimpleDateFormat("MMM d, ''yy");

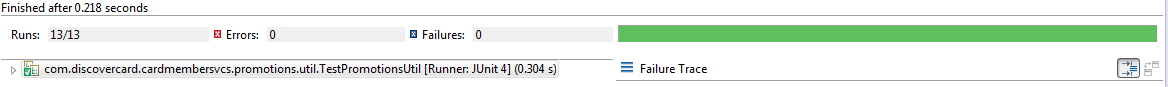
**public** **static** String formatDate(Date dateToFormat) {

**if** (dateToFormat==**null**) **return** "Unknown Date";

**return** *USER\_FRIENDLY\_DATE\_FORMAT*.format(dateToFormat);

}

**Step 4:** Run all tests, they should now pass.



**Step 5:** Refactor. Did you notice we could have used a constant for the friendly message?

**private** **static final** SimpleDateFormat *USER\_FRIENDLY\_DATE\_FORMAT* = **new** SimpleDateFormat("MMM d, ''yy");

**private** **static final** String *USER\_FRIENDLY\_ERROR* = "Unknown Date";

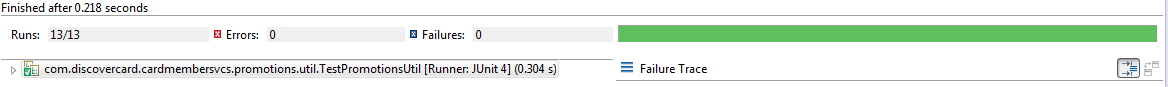
**public** **static** String formatDate(Date dateToFormat) {

**if** (dateToFormat==**null**) **return** *USER\_FRIENDLY\_ERROR*;

**return** *USER\_FRIENDLY\_DATE\_FORMAT*.format(dateToFormat);

}

**Step 6:** Run all tests again, they should still pass.



You should notice this is very similar to how you normally code. The difference is instead of writing a method and testing it by just looking at the output visually, we are now writing code to test it.

This accomplishes a few things:

1. Following this pattern naturally creates a robust high-coverage test suite.
2. Following this pattern encourages writing code that is JUnit-friendly.
3. Reduces development cost by delivering higher quality code to QA. *Every defect caught by a unit test is one that never gets logged in ALM.*
4. Reduces development cost by making ongoing maintenance faster and easier. *Would you rather maintain an application with 0% test coverage or 90% test coverage?*

# Scenario #2: Modifying an existing method without an existing test.

When modifying an existing method that has no tests, it would be ideal to start by creating a robust set of positive and negative tests prior to making your changes. That is not always practical.

Let’s say we need to add a new field into the model for display on the /5percent page in the Promotions application and let’s pretend no JUnit tests exist. I could add a test that only tests my change. Again, this is not ideal. If you can create the full tests, do it. If that is not practical, creating a partial test is better than nothing.

**Step 1:** Create a test that just tests the basics of the existing method.

@Test

**public** **void** test5Percent() **throws** Exception {

//Note: acctInfo is created in the setUp method.

mockMvc.perform(*get*("/5percent").requestAttr("acctInfo", acctInfo).requestAttr("org.springframework.security.web.csrf.CsrfToken", csrfToken)).andExpect(*status*().isOk()).andExpect(*view*().name("masterTemplate")).andExpect(*model*().attribute("pageName", "fivepercent"));

}

**Step 2:** Add a condition to the test for your requirement. Let’s say we are adding the card member’s name, so we can display “Hello <name>!” on the page. (Highlighted portion is the addition.)

@Test

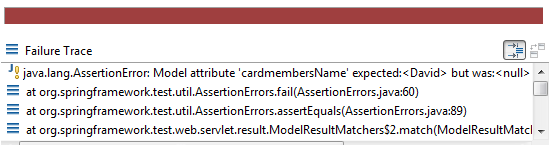
**public** **void** test5Percent() **throws** Exception {

// Note: acctInfo is created in the setUp method.

mockMvc.perform(*get*("/5percent").requestAttr("acctInfo", acctInfo).requestAttr("org.springframework.security.web.csrf.CsrfToken", csrfToken)).andExpect(*status*().isOk()).andExpect(*view*().name("masterTemplate")).andExpect(*model*().attribute("pageName", "fivepercent")).andExpect(*model*().attribute("cardmembersName", "David"));

}

**Step 3:** Run the test, it should fail.



**Step 4:** Modify the method under test. (Highlighted portion is the addition.)

@RequestMapping(value = "/my5percent", method = RequestMethod.***GET***)

**public** ModelAndView myFivePercent(HttpServletRequest request, HttpServletResponse response) {

**if** (logger.isDebugEnabled())

logger.debug("MyFivePercentController::myFivePercent Enter");

ModelAndView mav = **new** ModelAndView(***MASTER\_TEMPLATE***);

mav.addObject(***PAGE\_NAME***, ***MY\_FIVE\_PERCENT***);

AccountInfo acctInfo = (AccountInfo) request.getAttribute(***ACCT\_INFO***);

String cardMemberName = PromotionsUtil.*getCardMemberName*(acctInfo);

mav.addObject("cardmembersName", cardmembersName);

mav.addObject("logicalDate", ***LOGICAL\_DATE\_FORMAT***.format(DateUtil.*getCurrentCalendar*().getTime()));

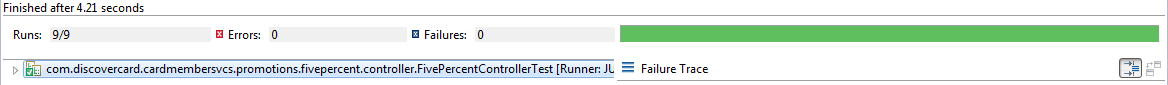
**if** (logger.isDebugEnabled())

logger.debug("MyFivePercentController::myFivePercent Exit");

**return** mav;

}

**Step 5:** Run the tests, they should now pass.



# Further Reading

<https://www.tutorialspoint.com/software_testing_dictionary/test_driven_development.htm>

<https://www.telerik.com/blogs/30-days-tdd-day-one-what-is-tdd>

<https://technologyconversations.com/2013/12/24/test-driven-development-tdd-best-practices-using-java-examples-2/>

<https://www.javaworld.com/article/2076265/testing-debugging/junit-best-practices.html>

<http://www.kyleblaney.com/junit-best-practices/>

<https://www.youtube.com/watch?v=s9vt6UJiHg4&t=218s>