Unit Testing Conventions

Introduction

Our unit tests are written using the testing framework JUnit version 4. (Currently Migrating to JUnit version 5)

When writing unit tests you should make sure that they

- are short and test only one small part (behavior) of a method (a method usually has more than one test)
- have descriptive names that describe the behavior they are testing
- cover edge cases (see example below)
- cover exceptional cases (what happens if you pass in null?)
- do not access the filesystem, database or network (these components are usually mocked and tested in integration tests)

Take for example this method

```
public static Boolean isInNormalNumericRange(Float value, ConceptNumeric concept) {
    if (concept.getHiNormal() == null || concept.getLowNormal() == null) {
        return false;
    }
    return (value <= concept.getHiNormal() && value >= concept.getLowNormal());
}
```

It lets the user know if the given value is in the range of values (high and low) set in the given concept. How does the method behave when you pass in a concept that has no high or low range set? It will return false. How does the method behave when you pass in a value that is within the range of the given concept? It will return true.

How do we test these behaviors? The following shows 2 example tests that pass in a Concept with different values set for the range by its setters setHiNormal and setLowNormal. Finally we assert on the output of the method we are testing to ensure the result is what we expect. This shows that we try to think of the methods we test as black boxes where we tweak the input to exercise a certain path in the code and assert on its output.

```
@Test
public void isInNormalNumericRange_shouldReturnFalseIfHiNormalIsNull() {
    ConceptNumeric concept = new ConceptNumeric();
    concept.setHiNormal(null);
    assertFalse(OpenmrsUtil.isInNormalNumericRange(5.67f, concept));
}

@Test
public void isInNormalNumericRange_shouldReturnTrueIfValueIsInRange() {
    ConceptNumeric concept = new ConceptNumeric();
    concept.setHiNormal(10.34d);
    concept.setLowNormal(3.67d);
    assertTrue(OpenmrsUtil.isInNormalNumericRange(5.67f, concept));
}
```

Make sure to also test edge cases like in this example when the value is equal to the low or high bound of the range. You can see all of the test cases for isInNormalNumericRange at https://github.com/openmrs/openmrs-core/blob/master/api/src/test/java/org/openmrs/util/OpenmrsUtilTest.java.

The following sections describe our requirements and the new style/convention we follow.

Requirements

We encourage you to use Test Driven Development but you are free to use the development methodology you want to use.

The only thing we do not negotiate is tests. We want to write code to save lives. It is out of question that new contributions come without tests.
These are the basic requirements:

- If committing new code, the unit tests for that code should be committed at the same time
  - when you create a pull request on Github you should make sure that your tests passed (Travis CI will report back if they did not) and you should also make sure you did not decrease code coverage (coveralls.io will report back on this)
- Do not write tests for private methods as these are internals/implementation details that can quickly change. **Test public methods only.** Think of them as the contract that we agree on and promise to keep. The tests should consider the methods they test as a black box:
  - example public int add(int a, int b): you pass in two values and expect the method to return you the sum, assert (check) that the return value is as you expected
- Every unit test must use an assertion (Assert.assert___ methods to test output...not println statements!) that checks whether the expectations you have in this test are met. A test without assertions passes, which could hide that a method does not return what you expected. So add assertions.
- Look at the coverage and ensure that you tested all possible cases (branches in an if/else or switch/case), parameter's and states (what if you call the method with null, an empty List,...)
- If fixing a bug, write a failing test proving the bug, then fix the code to make the test pass.
- Every database-based test must be accompanied by a in-memory dbunit test dataset, described here.

**Conventions**

First start by importing the code templates we provide with the OpenMRS core repository (Eclipse or IntelliJ) which will make it easy for you to insert tests according to our conventions and save you from typing a lot.

Then read on the next sections on how to name test methods, what to test and have a look at the example code.

---

We want to help newcomers with simple rules with good explanations as to why we think these rules (be it naming or testing) are good. These rules also help a community of many to write code in a similar way which helps when reading code someone else wrote. However, the conventions should only be seen as a guideline not as a law and they will also evolve. What we think of as a good approach today might not be so good tomorrow. Please speak up, give us your input, help us to improve the way we work 😊

**How do I name my test?**

We follow a naming approach described by Dan North as **Behavior Driven Development**, he states: "What to call your test is easy – it’s a sentence describing the next behaviour in which you are interested."

We name our test methods in this scheme:

```java
shouldDoThisGivenThat()
```

where the keyword `should` is followed by a [camel cased sentence](https://en.wikipedia.org/wiki/CamelCase) of what you expect the method you are testing to do or return.

**When testing exceptions**

Lets say you want to write a test that asserts that

```java
public Patient savePatient(Patient patient)
throws a NullPointerException when called with null.
```

Possible

```java
shouldThrowNullPointerExceptionGivenNull or shouldFailIfGivenNullAsPatient
```

Better

```java
shouldFailGivenNull
```

Reasons:

- in the test method itself you will assert the type of exception that this method throws there is no need to duplicate that in the method name
- the method you are testing only has one parameter so there is no need to specify which parameter was passed in as null

**When testing success or failure cases**

**Method**

```java
public User authenticate(String username, String password)
```

authenticates a user if the given user and password are correct.

These are the test method names testing

```java
• success: shouldAuthenticateGivenUsernameAndPassword()
```
How do I check (assert) my code works?

You ensure your code behaves as you intended it to by checking that the method you test either returns what you expect or checking that an object is changed in the way you expect.

To do that you will use what is called an assertion.

Assertions are checks that ensure what you expect is actually happening. For example in

```java
assertTrue(isAuthenticated(user));
```

the assertion assertTrue from JUnit will make sure that the call to isAuthenticated will evaluate to true for the given user. At least one assertion has to be present in every one of your tests. The testing framework JUnit already comes with a set of assertions which we extend with the assertion library hamcrest. Hamcrest helps make these checks read more natural and gives better outputs in case the tests fail. You will most likely choose a Hamcrest assertion when working with collections like Lists or Maps, Strings or Objects. Hamcrest also lets you combine matchers.

Hamcrest has a very valuable short tutorial showing you why it exists and how you can get started using it http://hamcrest.org/JavaHamcrest/tutorial. Just note that the tutorial uses the latest Junit version 5 and OpenMRS still relies on the version 4. Once you are comfortable with the basics you can find all the matchers you can use at http://hamcrest.org/JavaHamcrest/javadoc/2.2/

Use the assertThat from Hamcrest instead of the one coming from Junit as the Junit one is deprecated! For details see: https://github.com/junit-team/junit4/blob/master/doc/ReleaseNotes4.13.md#pull-request-1150-deprecate-assertassertthat

The following shows a test using Hamcrest matchers

```java
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.hasItems;
import static org.hamcrest.Matchers.is;
import org.junit.Test;

@Test
public void parse_shouldParseMappingFiles() throws IOException {
    Document config = buildOnValidConfigXml()
        .withTextNode("mappingFiles", "
            ReportDesign.hbm.xml ReportDesign.hbm.xml
            ReportRequest.hbm.xml"
        )
        .build();

    Module module = parser.parse(writeConfigXmlToFile(config));

    assertThat(module.getMappingFiles().size(), is(3));
    assertThat(module.getMappingFiles(), hasItems("ReportDesign.hbm.xml", "ReportRequest.hbm.xml"));
}
```

The test is checking that if we parse an XML configuration for an OpenMRS module we will get a module that has all of the mapping files set. We do that by looking at the getMappingFiles() which gives us a List<String>. We use the org.hamcrest.MatcherAssert.assertThat combined with matchers like is and hasItems. HasItems makes sure that the list contains these 2 items. Hamcrest provides many more matchers that you can find at http://hamcrest.org /JavaHamcrest/javadoc/2.2/

How do I test my code throws an exception?

Always use the assertThrows that was added in JUnit 4.13. Do not use the ExpectedException Rule or the expected field in the "@Test" annotation! Both will not be available in JUnit 5 to which we are currently migrating

![TRUNK-5813 - Replace all ExpectedException Rules and the @Test(expected..) annotation! For details on how to use assertThrows see https://github.com/openmrss/openmrss-core/](https://github.com/openmrss/openmrss-core/)

For details on how to use assertThrows see https://github.com/junit-team/junit4/wiki/Exception-testing#using-assertthrows-method

Let's say you wanted to test that the method Order.isDiscontinued throws an ApiException in case the dateStopped is after the autoExpireDate as in the code below

```java
failure: shouldNotAuthenticateGivenUsernameAndIncorrectPassword()
```

see how this reads nicely about what behavior is expected and the case of failure is simply shown by Not and the parameter that caused the failure right here highlighted with the word Incorrect.
public boolean isDiscontinued(Date aCheckDate) {
    if (dateStopped != null && autoExpireDate != null && dateStopped.after(autoExpireDate)) {
        throw new APIException("Order.error.invalidDateStoppedAndAutoExpireDate", (Object[]) null);
    }
    if (getVoided()) {
        return false;
    }
    Date checkDate = aCheckDate == null ? new Date() : aCheckDate;
    if (!isActivated(checkDate) || dateStopped == null) {
        return false;
    }
    return checkDate.after(dateStopped);
}

This is how you can ensure an APIException is thrown using the JUnit assertThrows method.

assertThrows does return the Exception so you can assert on its message if you want. In this case it might not not be absolutely necessary since isDiscontinued only throws one APIException. So if our test passes due to assertThrows making sure an APIException was thrown we know we hit the exact code path we wanted to test. You will find occasions where a method throws more than one Exception of the same type like the APIException. In such cases its better to also assert on at least parts of the message. This makes sure your tests cover different cases. Otherwise you might end up with 2 tests passing because the expected Exception is thrown but since you did not assert on the message it could well be the same in both tests.

Unit Tests By Example

Plain old java object testing

The getters and setters do not have to be tested unless they are doing something more than getting/setting an internal variable

Like for example

Patient.getIdentifiers() which returns an empty set if the field identifiers is null, see test PatientTest

Service layer testing

All methods in the ___Service classes should be testing with proper input, improper input, and improper data in between.

See EncounterServiceTest.java http:-dev.openmrs.org-browser-openmrs-trunk-test-api-org-openmrs-test-api-EncounterServiceTest.java

Database layer testing

Most database methods are testing in the process of testing the aforementioned Service Layer Testing. However, some database layer methods are not exposed through Service layer methods. These methods need to be tested.

See EncounterDAOTest.java http:-dev.openmrs.org-browser-openmrs-trunk-test-api-org-openmrs-test-api-db-EncounterDAOTest.java
Editor testing

All set and get methods in the editor require testing.
See DrugEditorTest.java http:-dev.openmrs.org-browser-openmrs-trunk-test-api-org-openmrs-test-propertyeditor-DrugEditorTest.java

See DrugEditor.java http:-dev.openmrs.org-browser-openmrs-trunk-src-api-org-openmrs-propertyeditor-DrugEditor.java

Validator testing

Make use of JUnits Before to setup variables (like for example encounter) that you need in multiple tests so that your tests ideally only contain a few changes to the variables that make this test special. Also assert the exceptions you throw using JUnits assertThrows. Note that you will find existing tests that use the ExpectedException Rule but that one is deprecated so please do not copy this approach.

EncounterValidator.java
EncounterValidatorTest.java

See Also

- See the Spring Testing Documentation for background on what the Spring framework provides OpenMRS in terms of testing.
- Look at the source code for OpenMRS core for examples of how to test each layer. By convention, code is kept in folders called src/main, and related tests are kept next to it in src/test.
- Read the first 4 sections of Dan North’s excellent introduction to BDD

(Optional) Background on previous styles of testing

We have been changing the way we write our tests twice in the past which you can see in the code of the OpenMRS core and maybe also some OpenMRS modules.

It started with tests looking like this

```java
@Test
@Verifies(value = "return null low and high if accessors are not called", method = "DoubleRange ()")

public void DoubleRange_shouldReturnNullLowAndHighIfAccessorsAreNotCalled() {
    DoubleRange dr = new DoubleRange();
    assertNull(dr.getHigh());
    assertNull(dr.getLow());
}
```

with the org.openmrs.test.Verifies annotation

And moved on to

```java
/**
 * @see org.openmrs.web.WebUtil#getContextPath()
 * @verifies should return empty string if webappname is null
 */
@Test
public void getContexPath_shouldReturnEmptyStringWhenWebAppNameIsNull() throws Exception {
    WebConstants.WEBAPP_NAME = null;
    assertEquals("", WebUtil.getContextPath());
}
```

where verifies was an annotation in the javadoc stating what was tested.

Both styles had in common that you were writing a sentence about what case you were trying to test using the should annotation in an interface or above a concrete method.
/**
 * Gets a allergy matching the given allergyId
 * @param allergyId of allergy to return
 * @return the allergy matching the given allergyId
 * @should return allergy given valid allergyId
 * @should return null if no object found with given allergyId
 */
public Allergy getAllergy(Integer allergyId);

and then use the IDE plugin - Generate Test Case Plugin to generate the test boilerplate (javadoc, test annotation and test method with no test implemented) in a test class.

The reason why these styles came into existence were, back then

- we were focused on the question of "how to deal with a huge pile of existing API code with no tests" and not thinking about how best to write code going forwards
- we were completely unfamiliar with Test Driven Development (TDD) best practices and test coverage tools

Going forward we are discouraging new contributions from using these old styles for the following reasons:

- it violates Test Driven Development since the plugin forces you to write the should annotation sentence (of what you want to test) in an existing interface or a concrete class and not the test class. This means you have to write code before writing a test. Writing tests first naturally leads to code that is easy to test.
- the plugin is not maintained and was not always working (URL needed to install it was broken, might not work on all IDE versions/types). This style is repeating the same sentence in 3 places (AT should on interface or class, AT verifies in javadoc, method name in the test class) in slightly different styles which without the plugin is a waste of precious time. We do want developers to contribute their time and solve problems and not get carpal tunnel syndrome 😏
- the OpenMRS core on Github and also modules are integrated with the continuous integration provider Travis CI so that whenever a new contribution of code comes in as a pull request the tests are run and a test coverage report is sent back to Github (via coveralls.io). So we know if your contribution is well tested

If you want to read the community discussion we had about changing the style see https://talk.openmrs.org/t/rethink-unit-test-style-to-remove-duplication/10200