Unit Testing Conventions

Introduction

Our unit tests are written using the testing framework JUnit version 4

- Unit tests should be very short and test only one small part (behavior) of a method
- The unit test name should be descriptive enough to describe the entire test

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

Unit Test 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.

Unit Test 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 😊

Naming Test Methods

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:

shouldDoThisGivenThat()

where the keyword should is followed by a camel cased sentence of what you expect the method you are testing to do or return.

Example Should Sentences

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
shouldThrowNullPointerExceptionGivenNull or shouldFailIfGivenNullAsPatient

Better

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 null

### Testing Success/Failure Parameters

**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

- success: shouldAuthenticateGivenUsernameAndPassword
- 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 highlighted with IncorrectPassword

### Assertions

Assertions are checks that what you expect to happen is happening. At least one 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. It helps make these checks read more natural and gives better outputs in case the tests fail.

### Example Code

#### 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

```java
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 JUnit's `@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 JUnit's `@Rule` and `ExpectedException`.

EncounterValidator.java

EncounterValidatorTest.java
Related Frameworks/Projects

- [http://junit.org/](http://junit.org/) - framework to write Java tests
- [http://hamcrest.org/JavaHamcrest/](http://hamcrest.org/JavaHamcrest/) - assertions to better express the intent of our Java tests
- JBehave - Behavior driven testing
- DbUnit - JUnit extension for setting and checking DBMS state
- Eclemma - Eclipse code coverage plugin
- Selenium - Web app testing tool
- DWR - Direct Web Remoting

See Also

- See the [Spring Testing Documentation](http://www.springframework.org/) for background on what the Spring framework provides OpenMRS in terms of testing.
- Look at the source code for [OpenMRS core](http://www.openmrs.org/) 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
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Verifies {
    String value() default "default verify value";
    String method() default "default verify method";
}

public void DoubleRange_shouldReturnNullLowAndHighIfAccessorsAreNot Called() { 
    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
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Verifies {
    String value() default "default verify value";
    String method() default "default verify method";
}

public void getgetContextPath_shouldReturnEmptyStringWhenWebAppNameIsNull() throws Exception {
    WebConstants.WEBAPP_NAME = null;
    Assert.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

```java
/**
 * 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](https://www.redhat.com/) 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