intelliware
software development

Delivery matters.™

Writing Tests : Best Practices and Legacy Code

Theory and Examples

Presented by Michael Pickard

intelliware
software development

Basic Guidelines

A.A.A.

Common test design pattern is:

- **A**rrange
  - Instantiating the class being tested.
  - Setting up required variables, etc.
- **A**ct
  - Execute the code under test.
- **A**ssert
  - Confirm the expected result matches the actual.
  - Verify method calls if necessary.

```
@Test
public void can_add_two_numbers_together() {
    // Arrange
    int firstNumber = 2;
    int secondNumber = 3;
    Calculator fixture = new Calculator();

    // Act
    int result = fixture.add(firstNumber, secondNumber);

    // Assert
    Assert.assertEquals(5, result);
}
```

Overall Goals

Tests should be:

- Maintainable
  - Tests should have minimal duplication.
  - Avoid testing multiple things in a single test.
  - Avoid testing private/protected methods directly. {*Will be addressed in a later slide*}
- Trustworthy
  - Does not repeat the logic from the code.
  - Every test should be repeatable 100% of the time.
  - Should never rely on order of execution of the tests.
  - Avoid elements that change such as `new Date()`, `Random`, Threads, etc.
  - Avoid using a real database or file system.
- Readable
  - Follow the A.A.A. pattern as much as possible.
  - Follow proper naming conventions.
  - Proper use of set up and clean up methods.
  - Maintain visibility of values (well named constants can be helpful).

Simple Best Practices

- A test should only do one thing (Just like a method should!).
  - Regular cases and error cases should not be in the same test.
- Use blatantly descriptive names.
  - Makes it very easy to see what fails (and why) without looking at the code itself.
- Refer to the object under test by a specific common name .
  - E.g. **fixture** or **target**
- Elements that are used in all tests should be set up in a test initializer method.
- Tests should **never** rely on the order of execution!
- Tests should not change the global state.
  - If they do (some legacy code may make it so there is no choice) make sure that global state is reverted at the end of the test (potentially in a test clean up method).
- Try to check for exceptions when they occur instead of generically.
  - `Try/catch` with `Assert.fail()` is usually better than `@Test(expected = SomeException.class)`

Example tests that can be improved

Test #1

```
@Test
public void test1()
{
    String[] countries = { "Canada", "AUSTRALIA", "spain", "Chad", "JaPaN" };
    Sorter sorter = new Sorter();
    CaseModifier caseModifier = new CaseModifier();
    String[] expected = { "AUSTRALIA", "CANADA", "CHAD", "JAPAN", "SPAIN" };

    String[] result = sorter.sort(caseModifier.toUpper(countries));

    Assert.assertArrayEquals(expected, result);
}
```

What was wrong?

```
@Test
public void test1()
{
    String[] countries = { "Canada", "AUSTRALIA", "spain", "Chad", "JaPaN" };
    Sorter sorter = new Sorter();
    CaseModifier caseModifier = new CaseModifier();
    String[] expected = { "AUSTRALIA", "CANADA", "CHAD", "JAPAN", "SPAIN" };

    String[] result = sorter.sort(caseModifier.toUpper(countries));

    Assert.assertArrayEquals(expected, result);
}
```

- The test name tells us nothing.
- What are we actually testing? The sorter or the case modifier?
- Are we actually trying to test two things at once?

Test #2:

```
@Test (expected = ArgumentException.class)
public void converter_converts_all_values_to_roman_numerals()
{
    RomanNumeralConverter fixture = new RomanNumeralConverter();

    Assert.assertEquals("I", fixture.toRomanNumeral(1));
    Assert.assertEquals("V", fixture.toRomanNumeral(5));
    Assert.assertEquals("X", fixture.toRomanNumeral(10));
    Assert.assertEquals("L", fixture.toRomanNumeral(50));
    Assert.assertEquals("C", fixture.toRomanNumeral(100));
    Assert.assertEquals("D", fixture.toRomanNumeral(500));
    Assert.assertEquals("M", fixture.toRomanNumeral(1000));
    Assert.assertEquals("CDXLIV", fixture.toRomanNumeral(444));
    Assert.assertEquals("CMXCIX", fixture.toRomanNumeral(999));
    Assert.assertEquals("MMMDCCCLXXXVIII", fixture.toRomanNumeral(3888));
    fixture.toRomanNumeral(0);
    fixture.toRomanNumeral(-1);
}
```

What was wrong?

```
@Test (expected = ArgumentException.class)
public void converter_converts_all_values_to_roman_numerals()
{
    RomanNumeralConverter fixture = new RomanNumeralConverter();

    Assert.assertEquals("I", fixture.toRomanNumeral(1));
    Assert.assertEquals("V", fixture.toRomanNumeral(5));
    Assert.assertEquals("X", fixture.toRomanNumeral(10));
    Assert.assertEquals("L", fixture.toRomanNumeral(50));
    Assert.assertEquals("C", fixture.toRomanNumeral(100));
    Assert.assertEquals("D", fixture.toRomanNumeral(500));
    Assert.assertEquals("M", fixture.toRomanNumeral(1000));
    Assert.assertEquals("CDXLIV", fixture.toRomanNumeral(444));
    Assert.assertEquals("CMXCIX", fixture.toRomanNumeral(999));
    Assert.assertEquals("MMMDCCCLXXXVIII", fixture.toRomanNumeral(3888));
    fixture.toRomanNumeral(0);
    fixture.toRomanNumeral(-1);
}
```

- Too many asserts. Failing fast will always only show us one error even if there are multiple.
- Where does the exception happen?

Test #3

```
public class TestSumCalculator {

    SumCalculator fixture = SumCalculator.getInstance();

    @Test
    public void initial_sum_calculated_correctly()
    {
        int result = fixture.calculateSum(50);

        Assert.assertEquals(50, result);
    }

    @Test
    public void subsequent_sum_calculated_correctly()
    {
        int result = fixture.calculateSum(100);

        Assert.assertEquals(150, result);
    }
}
```

What was wrong?

```
public class TestSumCalculator {

    SumCalculator fixture = SumCalculator.getInstance();

    @Test
    public void initial_sum_calculated_correctly()
    {
        int result = fixture.calculateSum(50);

        Assert.assertEquals(50, result);
    }

    @Test
    public void subsequent_sum_calculated_correctly()
    {
        int result = fixture.calculateSum(100);

        Assert.assertEquals(150, result);
    }
}
```

- Tests will fail if not run in the specified order.
- We're relying on a state change after the first test for our assert in the second.

Test #4

```
public class AuthorDateFormatter {
    public String format(String first, String last, Date published) {
        return last.toUpperCase() + ", " + first.toUpperCase() + " : "
                + DateFormat.getInstance().format(published);
    }
}
```

```
@Test
public void correct_author_date_formatting()
{
    String last = "Last";
    String first = "First";
    AuthorDateFormatter fixture = new AuthorDateFormatter();

    String result = fixture.format(first, last, new Date());
    String expected = last.toUpperCase() + ", " + first.toUpperCase() + " : "
            + DateFormat.getInstance().format(new Date());

    Assert.assertEquals(expected, result);
}
```

What was wrong?

```
public class AuthorDateFormatter {
    public String format(String first, String last, Date published) {
        return last.toUpperCase() + ", " + first.toUpperCase() + " : "
         + DateFormat.getInstance().format(published);
    }
}

@Test
public void correct_author_date_formatting()
{
    String last = "Last";
    String first = "First";
    AuthorDateFormatter fixture = new AuthorDateFormatter();

    String result = fixture.format(first, last, new Date());
    String expected = last.toUpperCase() + ", " + first.toUpperCase() + " : "
     + DateFormat.getInstance().format(new Date());

    Assert.assertEquals(expected, result);
}
```

- Duplication of production code.
- Use of variable element “Date”.

What about private methods?

J.B. Rainsberger states in '*JUnit Recipes – Practical Methods for Programmer Testing*':

"If you want to write a test for a private method, the design may be telling you that the method does something more interesting than merely helping out the rest of the class's public interface. Whatever that helper method does, it is complex enough to warrant its own test, so perhaps what you really have is a method that belongs to another class – a collaborator of the first."

"Moreover, by applying this refactoring, you have taken a class that had (at least) two independent responsibilities and split it into two classes, each with its own responsibility. This supports the Single Responsibility Principle of OO programming."[1]

Or Michael Feathers in '*Working Effectively with Legacy Code*':

"If you have the urge to test a private method, the method shouldn't be private; if making the method public bothers you, chances are, it is because it is part of a separate responsibility: it should be on another class"[2]

1. Rainsberger, J.B. JUnit Recipes: Practical Methods for Programmer Testing. 2004. Manning Publications.
2. Feathers, Michael C. Working Effectively with Legacy Code. 2004. Prentice Hall.

Example:

This is what we really want to be testing in this class.

It's all private though!

```
public class TransactionLogger {

    FileWriter writer;

    public TransactionLogger() throws IOException
    {
        writer = new FileWriter(new File("log.txt"));
    }

    public void log(Transaction transaction) throws IOException
    {
        writer.write("Transaction: " + transaction.getId());
        writer.write("Store: " + transaction.getStore());
        writer.write("Date: " + formatDate(transaction.getDate()));
        writer.write("Price: " + formatMoney(transaction.getPrice()));
        writer.write("Tax: " + formatMoney(transaction.getTax()));
    }

    private String formatMoney(int money)
    {
        int dollars = money / 100;
        int cents = money % 100;
        String centsString = (cents < 10 ? "0" : "") + cents;
        return "$" + dollars + "." + centsString;
    }

    private String formatDate(Date date)
    {
        return new SimpleDateFormat("MM/dd/yyyy").format(date);
    }
}
```

What we really want is…:

```
public class TransactionLogger
{
    FileWriter writer;
    IFormatter formatter;

    public TransactionLogger(IFormatter formatter) throws IOException
    {
        writer = new FileWriter(new File("log.txt"));
        this.formatter = formatter;
    }

    public void log(Transaction transaction) throws IOException
    {
        writer.write("Transaction: " + transaction.getId());
        writer.write("Store: " + transaction.getStore());
        writer.write("Date: " + formatter.formatDate(transaction.getDate()));
        writer.write("Price: " + formatter.formatMoney(transaction.getPrice()));
        writer.write("Tax: " + formatter.formatMoney(transaction.getTax()));
    }
}
```

… And:

```
public interface IFormatter {
    String formatMoney(int money);
    String formatDate(Date date);
}

public class StandardFormatter implements IFormatter
{

    @Override
    public String formatMoney(int money)
    {
        int dollars = money / 100;
        int cents = money % 100;
        String centsString = (cents < 10 ? "0" : "") + cents;
        return "$" + dollars + "." + centsString;
    }

    @Override
    public String formatDate(Date date)
    {
        return new SimpleDateFormat("MM/dd/yyyy").format(date);
    }

}
```

This class is very testable:

- Two public methods.
- No complex dependencies.

As an added bonus, we can now swap in any kind of `IFormatter`.

For instance, if we want to support different jurisdictions where the date or currency formats are different.

intelliware
software development

Dealing with Legacy Code

Where to start

- Legacy code is often a pseudonym for “Code with minimal testing”. ☺
- Our assumption is that the code works.
- Much of the code is very difficult to write tests for (potentially impossible in it’s current state!).

- Unfortunately we have NO safety net now so any refactoring we do has the potential to break things. ☹
- Start with any points in the code that are straightforward to add tests for.
- Follow with the safest (typically the most simple and straightforward) refactor and then add a test for that.
- Repeat as often as required.

Some Options:

- Break a large complex method into multiple easily testable methods (or even separate classes).
- Break a large complex class (with multiple responsibilities) into multiple single responsibility classes.
- Replace concrete objects with abstract ones.
- Use polymorphism to isolate units of testable work.
- Minimize the number of dependencies.
- Minimize constructor responsibilities (pass in variables rather than instantiate, and do minimal work in the constructor).

Don’t be afraid to step away from a chunk of code that is too difficult to work with for now. You can always come back to it later when you have built up more of a testing safety net.

Example:

```
public class FileExtensionManager
{
    public boolean hasValidExtension(String file)
    {
        boolean result = SomeProblemDependency.doSomething();   // Database access?  File access?  etc..
        return result;
    }
}
```

```
public class MyClass
{
    public boolean isValidFilename(String filename)
    {
        FileExtensionManager manager = new FileExtensionManager();
        return manager.hasValidExtension(filename) && filename.length() > 5;
    }
}
```

```
public class TestMyClass
{
    @Test
    public void is_valid_filename_returns_false_for_short_names()
    {
        MyClass fixture = new MyClass();
        Assert.assertFalse(fixture.isValidFilename("a.txt"));
    }
}
```

For our testing, we want *manager.hasValidExtension* to return what we choose. For this first test, we are only interested in a short name; we want *manager.hasValidExtension* to return true.

Option #1 (Extract & Override):

Extract the problematic code to a new method.

Override the method in a subclass of the class under test.

Set up the variables to meet the tests needs.

```
public class MyClass
{
    public boolean isValidFilename(String filename)
    {
        return isValidExtension(filename) && filename.length() > 5;
    }

    protected boolean isValidExtension(String filename)
    {
        FileExtensionManager manager = new FileExtensionManager();
        return manager.hasValidExtension(filename);
    }
}
```

```
public class TestableMyClass extends MyClass
{
    public boolean validExtension;

    @Override
    protected boolean isValidExtension(String filename) {
        return validExtension;
    }
}
```

```
public class TestMyClass
{
    @Test
    public void is_valid_filename_returns_false_for_short_names()
    {
        TestableMyClass fixture = new TestableMyClass();
        fixture.validExtension = true;
        Assert.assertFalse(fixture.isValidFilename("a.txt"));
    }
}
```

Option #2 (Test Double):

Two constructors. One that instantiates a `FileExtensionManager` and one that accepts one as a parameter.

Create a subclass of the problematic dependency that we can set up as our tests require.

```
public class MyClass
{
    FileExtensionManager extManager;

    public MyClass() {
        this(new FileExtensionManager());
    }

    public MyClass(FileExtensionManager extManager) {
        this.extManager = extManager;
    }

    public boolean isValidFilename(String filename)
    {
        return extManager.hasValidExtension(filename) && filename.length() > 5;
    }
}
```

```
public class FakeExtensionManager extends FileExtensionManager
{
    public boolean validExtension;

    @Override
    public boolean hasValidExtension(String file) {
        return validExtension;
    }

}
```

Continued…

Set up our fake extension manager to meet the test's needs, and pass it into the constructor of our class under test.


```
public class TestMyClass
{
    @Test
    public void is_valid_filename_returns_false_for_short_names()
    {
        FakeExtensionManager extManager = new FakeExtensionManager();
        extManager.validExtension = true;
        MyClass fixture = new MyClass(extManager);
        Assert.assertFalse(fixture.isValidFilename("a.txt"));
    }
}
```

Option #2b (Extract Interface):

Two constructors. One that instantiates a FileExtensionManager and one that accepts an IExtensionManager as a parameter.

Extract an interface for the problematic class.

Implement the interface so that we can set up as our tests require.

```
public class MyClass
{
    IExtensionManager extManager;

    public MyClass() {
        this(new FileExtensionManager());
    }

    public MyClass(IExtensionManager extManager) {
        this.extManager = extManager;
    }

    public boolean isValidFilename(String filename)
    {
        return extManager.hasValidExtension(filename) && filename.length() > 5;
    }
}
```

```
public interface IExtensionManager
{
    boolean hasValidExtension(String file);
}
```

```
public class FakeExtensionManager implements IExtensionManager
{
    public boolean validExtension;

    public boolean hasValidExtension(String file) {
        return validExtension;
    }

}
```

Continued…

Set up our fake extension manager to meet the test's needs, and pass it into the constructor of our class under test.


```
public class TestMyClass
{
    @Test
    public void is_valid_filename_returns_false_for_short_names()
    {
        FakeExtensionManager extManager = new FakeExtensionManager();
        extManager.validExtension = true;
        MyClass fixture = new MyClass(extManager);
        Assert.assertFalse(fixture.isValidFilename("a.txt"));
    }
}
```

Option #3 (Adapter):

The constructor takes in an `IExtensionManagerAdapter`.

Extract an interface for the adapter class.

Implement the actual adapter to wrap the `FileExtensionManager` class; delegating to it.

```
public class MyClass
{
    private IExtensionManagerAdapter extManager;

    public MyClass(IExtensionManagerAdapter extManager) {
        this.extManager = extManager;
    }

    public boolean isValidFilename(String filename) {
        return extManager.hasValidFileExtension(filename) && filename.length() > 5;
    }
}
```

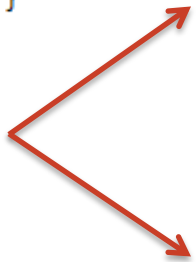
```
public interface IExtensionManagerAdapter
{
    boolean hasValidFileExtension(String filename);
}
```

```
public class ExtensionManagerAdapter implements IExtensionManagerAdapter
{
    private FileExtensionManager extManager;

    public ExtensionManagerAdapter() {
        this.extManager = new FileExtensionManager();
    }

    public boolean hasValidFileExtension(String filename) {
        return extManager.hasValidFileExtension(filename);
    }
}
```

Continued…

Create and set up our fake extension manager adapter to meet the test's needs; passing it into the constructor of our class under test.

```
public class FakeExtensionManagerAdapter implements IExtensionManagerAdapter
{
    public boolean validExtension;

    public boolean hasValidFileExtension(String filename) {
        return validExtension;
    }
}
```



```
public class TestMyClass
{
    @Test
    public void is_valid_filename_returns_false_for_short_names()
    {
        IExtensionManagerAdapter extManager = new FakeExtensionManagerAdapter();
        ((FakeExtensionManagerAdapter)extManager).validExtension = true;
        MyClass fixture = new MyClass(extManager);
        Assert.assertFalse(fixture.isValidFilename("a.txt"));
    }
}
```

Option #4 Mocking:

Two constructors. One that instantiates a `FileExtensionManager` and one that accepts one as a parameter.

```
public class MyClass
{
    FileExtensionManager extManager;

    public MyClass() {
        this(new FileExtensionManager());
    }

    public MyClass(FileExtensionManager extManager) {
        this.extManager = extManager;
    }

    public boolean isValidFilename(String filename)
    {
        return extManager.hasValidExtension(filename) && filename.length() > 5;
    }
}
```

Using a mocking framework (in this example 'Mockito', create a mock object and configure it to return what you need.

```
public class TestMyClass
{
    @Test
    public void is_valid_filename_returns_false_for_short_names()
    {
        FileExtensionManager extManager =
                Mockito.mock(FileExtensionManager.class);
        Mockito.when(extManager.hasValidExtension(
                Mockito.anyString())).thenReturn(true);

        MyClass fixture = new MyClass(extManager);
        Assert.assertFalse(fixture.isValidFilename("a.txt"));
    }
}
```

Some more options:

- Use factories for object instantiation as they are easy to mock/stub (and are a good design pattern regardless).
- Use adapter classes to communicate with external systems/libraries (as any change to the external system only needs to be handled in a single place in your code).
- Avoid static methods/classes as much as possible (they are easy to test in their own right, but are difficult to mock when testing other classes).
- Avoid passing around objects when you only need is a few pieces of data from that object (e.g. don’t pass a whole transaction to a formatter class when all you need to format is the date!).

intelliware
software development

Test Suite Best Practices

Some basic guidelines

- Unit tests should run very fast. They should run every time your local code builds.
- Integration (or full End to End) tests can be significantly slower to run, so those should ideally be set up as manual runs on your local machine (auto on the build server though).
- Don’t aim for 100% coverage. 70-80% is typically sufficient.
  - Don’t test every single input. Choose 1 or 2 vanilla cases, plus interesting (or edge) cases. Zero or null are often interesting cases.
- Don’t test the same thing in different tests (duplication in tests is not any better than duplication in code!)
- Don’t be afraid to drop a test that is excessively slow. You’re better off losing a little coverage than having people stop running tests because they take too long. Discuss this with your team first though, as better solutions may exist.

Specific examples

If multiple test classes share common configuration steps, or common data sets, then extract out a parent test class. This can be useful for:

- Instantiating mock/stub objects such as web services.
- Loading a single common set of test data (potentially into a mock database instance).
- Maintaining a common set of constants for easy reference.

Continued…

Use factories for generating stub objects easily (static classes are fine here).

- Useful to have generic versions of your objects (where you don't care about the contents), as well as specific ones where one or more fields have data you are interested in.

```
public class TestUserFactory {

    public static User createGenericUser(String userId)
    {
        return createSpecificUser(userId,
                "First", "Last", "Test Address", "test@email.com");
    }

    private static User createSpecificUser(String userId,
            String firstName, String lastName, String address, String email) {
        User testUser = new User(userId);
        testUser.setFirstName(firstName);
        testUser.setLastName(lastName);
        testUser.setAddress(address);
        testUser.setEmail(email);
        return testUser;
    }
}
```

Continued…

Extract common assertions to separate methods (or into parent test classes) to minimize duplication.

- Once again, generic and specific versions can be useful at times.

```
public class TestParent {

    public void assetUsersAreEqual(User expected, User actual) {
        assertStringValue(expected.getUserId(), actual.getUserId(), "User Id");
        assertStringValue(expected.getFirstName(), actual.getFirstName(), "First Name");
        assertStringValue(expected.getLastName(), actual.getLastName(), "Last Name");
        assertStringValue(expected.getAddress(), actual.getAddress(), "Address");
        assertStringValue(expected.getEmail(), actual.getEmail(), "Email");
    }

    private void assertStringValue(String expected, String actual, String fieldName) {
        Assert.assertEquals("Field: " + fieldName + " did not match.", expected, actual);
    }
}
```

intelliware
software development

Thank You.