Unit Tests

Tests Enable the -ilities, Because Tests Enable *Change*

- Fear is the path to the dark side. Tests remove the fear.
  - When you have tests, you are not afraid of changing and improving the system. Architecture is a second order effect.
- The three most important things about clean tests: readability, readability and readability. Test are as important as production code.
  - In a project using TDD, typically 40-50% of all code is test code.
  - If the tests are not easy to change as the system evolves → bad tests slow you down → you throw the tests away → you are afraid of changing the code → the code begins to rot → the project fails.
- "If you don't keep your tests clean, you will lose them. And without them, you lose the very thing that keeps your production code flexible. Yes, you read that correctly. It is *unit tests* that keep your code flexible, maintainable, and reusable. The reason is simple. If you have tests, you do not fear making changes to the code."

Clean Code ch9 p123

http://blog.objectmentor.com/articles/2007/10/20/architecture-is-a-second-order-effect

Tests Are a Specification of the System's Behaviour

- Test names should be sentences that provide the reason *why* some code/test was written.
- When a test fails, there are three options:
  1. Production code is broken → the test must not be changed.
  2. The test does not reflect recent changes in the production code → the test needs to be changed.
  3. The test is obsolete → the test needs to be removed.
- Unless the intent of the test is clear, you will not know what is wrong and what you should do when the test fails.
- The tests should be decoupled from the implementation. There should be no 1:1 relation between the tests and the implementation methods and classes.

http://dannorth.net/introducing-bdd
http://techblog.daveastels.com/2005/07/05/a-new-look-at-test-driven-development/
http://techblog.daveastels.com/files/BDD_Intro.pdf

"xUnit Style"

```
void testNull() {
  shouldFail(RuntimeException) {
    stack.push(null)
  }
  assertTrue stack.empty
}
```

"BDD Style"

```
scenario "null is pushed onto empty stack", {
  given "an empty stack",{
    stack = new Stack()
  }

  when "null is pushed", {
    pushnull = {
      stack.push(null)
    }
  }

  then "an exception should be thrown", {
    ensureThrows(RuntimeException){
      pushnull()
    }
  }

  and "then the stack should still be empty", {
    stack.empty.shouldBe true
  }
}
```


http://enfranchisedmind.com/blog/posts/behavior-driven-development/

33 specifications (including 2 pending) executed successfully

Story: empty stack

scenario null is pushed onto empty stack
given an empty stack
when null is pushed
then an exception should be thrown
then the stack should still be empty

scenario pop is called on empty stack
given an empty stack
when pop is called
then an exception should be thrown
then the stack should still be empty

Story: single value stack

scenario pop is called on stack with one value
given an empty stack with one pushed value
when pop is called
then that object should be returned
then the stack should be empty

scenario stack with one value is not empty
given an empty stack with one pushed value
then the stack should not be empty

scenario peek is called
given a stack containing an item
when peek is called
then it should provide the value of the most \
recent pushed value
then the stack should not be empty
then calling pop should also return the peeked \
value which is the same as the original \
pushed value
then the stack should be empty
then an example pending [PENDING]

...

http://enfranchisedmind.com/blog/posts/behavior-driven-development/

"One Assert per Test"

- Each test should test only one concept/behaviour.
  - Makes it possible to give intention-revealing names for each test, so that the reason why that test failed will be obvious.
- Arrange-Act-Assert
  - Not: Arrange-Act-Assert-Assert-Act-Assert-Assert...
    (which some are tempted to do in slow integration tests)
    - Don't do many things in one test. When there are many things that can cause a test to fail, you will not know quickly that what caused the test to fail. Keep tests without side-effects.

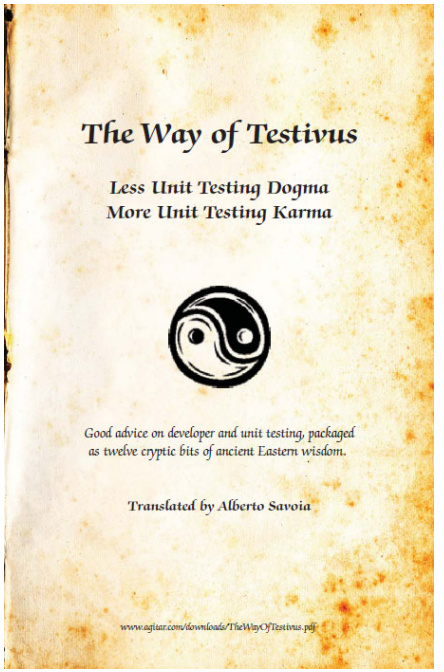
Clean Code ch9 p130
http://blog.astrumfutura.com/archives/388-Unit-Testing-One-Test,-One-Assertion-Why-It-Works.html
http://www.jbrains.ca/permalink/239
http://www.infoq.com/presentations/integration-tests-scam

Good Unit Tests Are FIRST

- **F**ast – If you hesitate to run the tests after a simple one-liner change, your tests are far too slow. We need to be able to run hundreds or thousands of tests per second.
- **I**solated – Tests isolate failures. Each test class name and test method name, together with the text of the assertion, should state exactly what is wrong and where.
- **R**epeatable – Test must have no side-effects. Repeatable tests do not depend on external services or resources.
- **S**elf-validating – Tests either pass or fail. You should not need to read a log file to decide whether a test passed or failed.
- **T**imely – Tests are written at the right time, immediately before the code that makes the tests pass.

http://agileinaflash.blogspot.com/2009/02/first.html
http://blog.objectmentor.com/articles/2007/08/02/which-came-first
Clean Code ch9 p132

The Way of Testivus



- If you write code, write tests.
- Don't get stuck on unit testing dogma.
- Embrace unit testing karma.
- Think of code and test as one.
- The test is more important than the unit.
- The best time to test is when the code is fresh.
- Tests not run waste away.
- An imperfect test today is better than a perfect test someday.
- An ugly test is better than no test.
- Sometimes, the test justifies the means.
- Only fools use no tools.
- Good tests fail.

http://www.agitar.com/downloads/TheWayOfTestivus.pdf
http://www.artima.com/weblogs/viewpost.jsp?thread=203994

Code Coverage

- Code coverage tells you when something is *not* tested. It will not tell when something *is* tested enough.
- Go for 100% coverage.
  - If you start with the first line of code you write, it's possible when using TDD. It's "easy" to get 90-95%.
- Don't go for 100% coverage.
  - Use your head.
- How many code paths are there in the following method?

```
public static String foo(Object obj) {
    return obj.toString();
}
```

http://www.artima.com/weblogs/viewpost.jsp?thread=204677
http://homepage.mac.com/hey.you/lessons.html
http://www.ibm.com/developerworks/java/library/j-cq01316/

Test Doubles

- When you want to test code that depends on something that is too difficult or slow to use in a test environment, or you need to verify the interaction with the component, swap in a test double for the dependency.
  - Different test doubles: Dummy, Stub, Mock, Spy, Fake.
- Mocks: state verification vs. behavior verification
- Mock roles, not objects.
- Dependency injection is useful for inserting test doubles into the system-under-test.

http://martinfowler.com/articles/mocksArentStubs.html
http://www.mockobjects.com/files/mockrolesnotobjects.pdf
http://www.infoq.com/news/2008/08/Mock-Roles-Pryce-and-Freeman
http://code.google.com/testing/TotT-2008-06-12.pdf
http://googletesting.blogspot.com/2008/06/tott-friends-you-can-depend-on.html
http://xunitpatterns.com/Mocks,%20Fakes,%20Stubs%20and%20Dummies.html
http://hamletdarcy.blogspot.com/2007/10/mocks-and-stubs-arent-spies.html

A **Dummy** passes bogus input values around to satisfy an API.

```
Item item = new Item(ITEM_NAME);
ShoppingCart cart = new ShoppingCart();
cart.add(item, QUANTITY);
assertEquals(QUANTITY, cart.getItem(ITEM_NAME));
```

A **Stub** overrides the real object and returns hard-coded values. Testing with stubs only is state-based testing; you exercise the system and then assert that the system is in an expected state.

```
ItemPricer pricer = new ItemPricer(){
  public BigDecimal getPrice(String name){
    return PRICE;
  }
};
ShoppingCart cart = new ShoppingCart(pricer);
cart.add(dummyItem, QUANTITY);
assertEquals(QUANTITY*PRICE, cart.getCost(ITEM_NAME));
```


http://code.google.com/testing/TotT-2008-06-12.pdf

A **Mock** can return values, but it also cares about the way its methods are called ("strict mocks" care about the order of method calls, whereas "lenient mocks" do not.) Testing with mocks is interaction-based testing; you set expectations on the mock, and the mock verifies the expectations as it is exercised. This example uses JMock to generate the mock (EasyMock is similar):

```
Mockery ctx = new Mockery();
final ItemPricer pricer = ctx.mock(ItemPricer.class);
ctx.checking(new Expectations() {{
  one (pricer).getPrice(ITEM_NAME);
               will(returnValue(PRICE));
}});
ShoppingCart cart = new ShoppingCart(pricer);
cart.add(dummyItem, QUANTITY);
cart.getCost(ITEM_NAME);
ctx.assertIsSatisfied();
```


http://code.google.com/testing/TotT-2008-06-12.pdf

A **Spy** serves the same purpose as a mock: returning values and recording calls to its methods. However, tests with spies are state-based rather than interaction-based, so the tests look more like stub style tests.

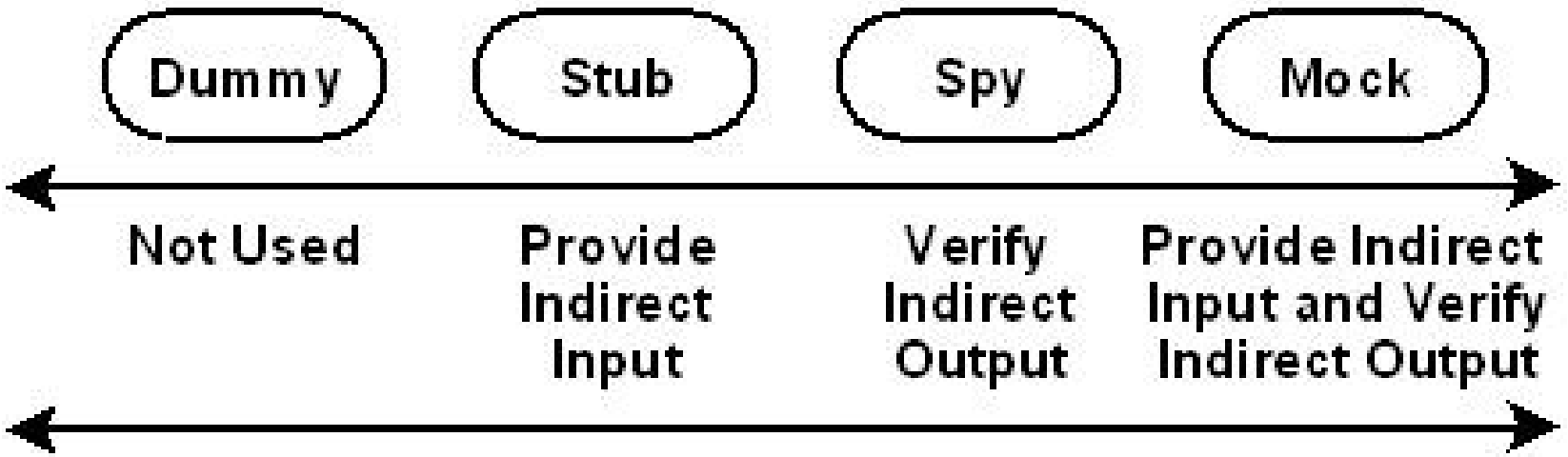
```
TransactionLog log = new TransactionLogSpy();
ShoppingCart cart = new ShoppingCart(log);
cart.add(dummyItem, QUANTITY);
assertEquals(1, logSpy.getNumberOfTransactionsLogged());
assertEquals(QUANTITY*PRICE, log.getTransactionSubTotal(1));
```

A **Fake** swaps out a real implementation with a simpler, fake implementation. The classic example is implementing an in-memory database, or using a fake BigTable.

```
Repository repo = new InMemoryRepository();
ShoppingCart cart = new ShoppingCart(repo);
cart.add(dummyItem, QUANTITY);
assertEquals(1, repo.getTransactions(cart).count());
assertEquals(QUANTITY,
             repo.getById(cart.id()).getQuantity(ITEM_NAME));
```


http://code.google.com/testing/TotT-2008-06-12.pdf

Collaborator Continuum



Simple — Complex

Specifies State — Specifies Behavior

Communicates Intent — Confusing Expectations

Front Door Setup — Back Door Setup

Failures at Assertions — Fails with Stack Trace

Easy to Maintain — Slit Your Wrists

http://hamletdarcy.blogspot.com/2007/10/mocks-and-stubs-arent-spies.html

Mock Frameworks for Java

JMock (http://www.jmock.org/)

- Set expectations declaratively before the test.

Uses a non-conventional, formatter-unfriendly syntax.

EasyMock (http://easymock.org/)

- Record expectations before the test, then replay them.

Mockito (http://code.google.com/p/mockito/)

- Does not really have mocks, but spies.

Verify interactions after the test. No before-test expectations, but only stubbed return values.

PowerMock (http://code.google.com/p/powermock/)

- Enables mocking of static methods, constructors, final classes and methods, private methods etc.

Course Material

- *Clean Code* chapter 9: Unit Tests
- http://blog.objectmentor.com/articles/2007/10/20/architecture-is-a-second-order-effect
- http://dannorth.net/introducing-bdd
- http://techblog.daveastels.com/files/BDD_Intro.pdf
- http://agileinaflash.blogspot.com/2009/02/first.html
- The Way of Testivus
  http://www.artima.com/weblogs/viewpost.jsp?thread=203994
- Testivus on Test Coverage
  http://www.artima.com/weblogs/viewpost.jsp?thread=204677
- http://martinfowler.com/articles/mocksArentStubs.html
- http://www.infoq.com/presentations/Mock-Objects-Nat-Pryce-Steve-Freeman
  http://www.infoq.com/news/2008/08/Mock-Roles-Pryce-and-Freeman
- http://code.google.com/testing/TotT-2008-06-12.pdf
  http://googletesting.blogspot.com/2008/06/tott-friends-you-can-depend-on.html
- http://xunitpatterns.com/Mocks,%20Fakes,%20Stubs%20and%20Dummies.html