“Test”-Driven Development?

Test (těst) n.

- A procedure for critical **evaluation**; a means of **determining** the **presence, quality, or truth** of something; a **trial**: a test of one's eyesight; subjecting a hypothesis to a test; a test of an athlete's endurance

- A series of questions, problems, or physical responses designed to **determine** knowledge, intelligence, or ability

- A basis for **evaluation** or **judgment**: "A test of democratic government is how Congress and the president work together" (Haynes Johnson)
Agenda

- Re-Defining TDD
- Which Tests to Write
  - ...and which ones *not* to write
- Defining a “Good” Unit Test
- Testing Qualities and Code Qualities
- TDD as Test-Driven *Design*
- Breaking Dependencies in Testing
- The Role of Patterns in TDD

Re-Defining TDD
Impediments

- TDD = “Write the Tests First”?
- Test First is a technique, TDD is a paradigm
- At first, developers writing tests (first) seems:
  1. Nonsensical. You cannot “test” something you don’t have
  2. Wasteful. Developers will be doing double the work
  3. Cumbersome. The team will slow down
  4. Unsustainable. The test suite will have to be maintained
- The first three questions can be answered by re-defining what TDD is
- The fourth issue is about what and how we test

Example Problem

- We’re tracking the amortization of a “Fixed Asset”
- Asset has an initial value, and a term for writing it off (“useful Life”)
- The calculation is simple: value/term or “Straight Line”

```
<table>
<thead>
<tr>
<th>Asset</th>
</tr>
</thead>
<tbody>
<tr>
<td>-value : double</td>
</tr>
<tr>
<td>-term : int</td>
</tr>
<tr>
<td>ctor Asset(double, int)</td>
</tr>
<tr>
<td>+getWriteOff(year); double</td>
</tr>
</tbody>
</table>
```

- How do we document this?

```
Functional Spec 1.0
Blah blah blah
Thus and so
Unless this or that
But always finally
```
How can we determine that this is still up to date, if we return six months later?

---

**Functional Spec 3.2**

Blah blah blah
Thus and so
Unless this or that
But always finally

---

```java
public class AssetTest {
    private Random any = new Random();

    @Test public void testAssetFirstYearStraightLineCalculation() {
        double anyValue = any.nextDouble();
        int anyTerm = any.nextInt();
        int yearToWriteOff = 1;

        double straightLineResult = anyValue / anyTerm;

        Asset testAsset = new Asset(anyValue, anyTerm);

        assertEquals(straightLineResult, testAsset.getWriteOff(yearToWriteOff), .02);
    }
}
```
Nonsensical? Wasteful?

- ... in TDD, we do not write “tests”
- They only look like tests
- In TDD, we are conducting an *analysis* to create the *functional specification*
  - ...which you were going to do anyway, right? 😊
  - ...and which will be *automatically verifiable* at any point in the future
- When we’re done, these specifications can *also* serve as regressions tests for refactoring
- Double the work? Nope
- *Multiple values from a single effort*
Asset, A New Requirement

- A year goes by, and the federal government issues a new regulation
- Assets with an initial value of more than $70,000 will amortize as $70,000
- What tests do you write?

New Specification (JUnit)

```java
@Test
public void testAssetsAreCappedAtAMaximum(){
    double cappedValue = 70001.00;
    double uncappedValue = 70000.00;
    int term = any.nextInt();
    int yearToWriteOff = 1;

    Asset cappedAsset = new Asset(cappedValue, term);
    Asset uncappedAsset = new Asset(uncappedValue, term);

    assertEquals(70000.00 / term,
                 cappedAsset.getWriteOff(yearToWriteOff), .02);
    assertEquals(uncappedValue / term,
                 uncappedAsset.getWriteOff(yearToWriteOff), .02);
}
```

Quality Issue: Magic Numbers!
Quality Issue: Redundancy!
Better Specification (JUnit)

```java
@Test public void testAssetsAreCAPPEDAtAMAXimum(){
    double cappedValue = Asset.CAP + 1;
    double uncappedValue = Asset.CAP;
    int term = any.nextInt();
    int yearToWriteOff = 1;

    Asset cappedAsset = new Asset(cappedValue, term);
    Asset uncappedAsset = new Asset(uncappedValue, term);

    assertEquals(Asset.CAP / term,
                 cappedAsset.getWriteOff(yearToWriteOff), .02);
    assertEquals(uncappedValue / term,
                 uncappedAsset.getWriteOff(yearToWriteOff), .02);
}

If this is a Spec, should it not show the current cap?
```

Specifications Must Be Complete (JUnit)

```java
@Test public void specifyAssetConstants(){
    assertEquals(70000.00, Asset.CAP, .02);
}
```
Other Tests?

- What about Asset.Cap / 2? Asset.Cap +10,000.00?

- Theoretically no end to this
  - Though QA may have standards for adequate scenarios

- In TDD the two tests:
  - At Cap + 1 : calculated against the cap
  - At Cap: calculated normally

- ...define the boundary for behavior

- The other tests would be duplication, and would pass and fail together
Other, Other Tests?

- Should we test that the algorithm for capping does not:
  - Format the hard disk?
  - Block port 5388?
  - Start a new thread that incrementally allocates all the memory of the system?
  - Etc...

- In TDD we test for *errors* not *malice*
- QA may, in fact, want to test some of these scenarios
- TDD does not replace QA

Sustainability

- TDD adoption often goes like this:
  - **Initial adoption**: Team is un-familiar with the process, slows down, experiments, resolves problems
  - **Competence**: Team “gets it” starts clicking, speeds up and gains confidence
  - **High Function**: Team becomes extremely productive, produces quality code in a predictable way, QA backlog shrinks, happy-happy joy-joy
  - **Struggle**: A new requirement causes multiple tests to fail which must be repaired. The number of test fails increases by iteration
  - **Collapse**: Test suite becomes un-maintainable, tests get disabled, TDD collapses under its own weight
Interpretation

- Java One Conference, Jim Coplien:
- "... one of the things we see in a lot of projects is that projects go south on about their 3rd sprint, and they crash and burn because they cannot go any further."
- Is this because TDD is not sustainable?
- Maybe it’s because the developers are not writing good TDD tests
- Good QA tests are not always good TDD tests
- What are good TDD tests?

Defining a “Good” TDD Test

“virtue”
A Good TDD “test” has three characteristics

1. The test will fail reliably for an expected reason

2. The test will never fail for any other reason

3. There is no other test that will fail for this same reason

Must Fail

- A test must fail when the behavior it covers is wrong
- A test that can never fail is worse than no test at all:
  - We think we have safety we don’t have
  - We think the system is working when it’s not
  - We think our spec is accurate when it is not
- In test-first we always fail the test before we implement the solution code, to ensure it can fail
- When testing legacy code, you must force a failure
- Right now this is not verifiable after the fact, but we’re working on this
Must Not Fail

- A test that fails for a reason other than intended will mislead the team
  - We try to solve problems we don’t have
  - We miss the problems we do have
  - The test is wasting our time

No Other Failure

- A test should fail alone
- Two tests failing for the same reason is bad...
- ...and becomes 3
- ...then 13
- ...then 30
- ...then they all get disabled
Remember Our Initial Test (JUnit)

```java
public class AssetTest {
    private Random any = new Random();

    @Test public void testAssetFirstYearStraightLineCalculation() {
        double anyValue = any.nextDouble();
        int anyTerm = any.nextInt();
        int yearToWriteOff = 1;

        double straightLineResult = anyValue / anyTerm;

        Asset testAsset = new Asset(anyValue, anyTerm);
        assertEquals(straightLineResult,
                     testAsset.getWriteOff(yearToWriteOff), .02);
    }
}
```
Making it Compile, and \textit{Fail} (Java)

\begin{verbatim}
public class Asset {
    public Asset (double value, int term){

    public double getWriteOff(int year){
        return 0.0;
    }
}
\end{verbatim}

Making it Pass (Java)

\begin{verbatim}
public class Asset {
    private double myValue;
    private int myTerm;

    public Asset (double value, int term){
        this.myValue = value;
        this.myTerm = term;
    }

    public double getWriteOff(int year){
        return this.myValue / this.myTerm;
    }
}
\end{verbatim}
This Second Test Fails, At First (JUnit)

```java
@Test
public void testAssetsAreCappedAtAMaximum(){
    double cappedValue = Asset.CAP + 1;
    double uncappedValue = Asset.CAP;
    int term = any.nextInt();
    int yearToWriteOff = 1;

    Asset cappedAsset = new Asset(cappedValue, term);
    Asset uncappedAsset = new Asset(uncappedValue, term);

    assertEquals(Asset.CAP / term,
                 cappedAsset.getWriteOff(yearToWriteOff), .02);
    assertEquals(uncappedValue / term,
                 uncappedAsset.getWriteOff(yearToWriteOff), .02);
}
```

Implementing the Cap... (Java)

```java
public class Asset {
    public static final double CAP = 70000.00;

    private double myValue;
    private int myTerm;

    public Asset (double value, int term){
        this.myValue = value;
        this.myTerm = term;
    }

    public double getWriteOff(int year){
        return Math.min(this.myValue, Asset.CAP) /
                this.myTerm;
    }
}
```

What happens to the first test when we make this change?
public class AssetTest {
    private Random any = new Random();

    @Test public void testAssetFirstYearStraightLineCalculation() {
        double anyValue = any.nextDouble();
        int anyTerm = any.nextInt();
        int yearToWriteOff = 1;

        double straightLineResult = anyValue / anyTerm;

        Asset testAsset = new Asset(anyValue, anyTerm);

        assertEquals(straightLineResult,
                     testAsset.getWriteOff(yearToWriteOff), .02);
    }
}

Information About the Design

- This is a simple bit of code, but even so...
- The making the second test pass causes the first test to fail randomly. This means they are coupled...
- ...but they aren’t. One test has no reference to the other
- The coupling has to be in the production code
- This tells us to consider other designs
One Possible Solution

- If we pulled the “Capping Behavior” into its own class, then we could test it on its own
- This would be a design change
- This would also improve the cohesion of Asset, and make AssetCap reusable

Implement the Relationship (Java)

```java
public class Asset {
    private double myValue;
    private int myTerm;
    private AssetCap myAssetCap;

    public Asset (double value, int term) {
        this.myAssetCap = new AssetCap();
        this.myValue = value;
        this.myTerm = term;
    }

    public double getWriteOff(int IgnoreYear) {
        return myAssetCap.capValue(this.myValue)/this.myTerm;
    }
}
```
However!

- The test for AssetCap can only fail if AssetCap does not work
- The test for Asset can fail if Asset fails, OR if AssetCap fails
- We don’t want that, and it’s because we’ve created a dependency
Mocking

- Breaking dependencies for testing is a big subject, and it a major aspect of developer training
- We’ll look at one example, using a hand-crafted Mock object in place of the dependency
- We’ll also look at one example of injecting the Mock at runtime, for testing only
- There are many, many other ways to do all of this

Step One, Extract Interface

- This is a relatively simple refactor
- Most tools will do it for you
- It does point out the advantage of designing to Interfaces in general
You can write this yourself, or have it generated by a tool.

```
class MockAssetCap implements AssetCap {
    public double capValue(double aValue) {
        return aValue;
    }
}
```
Step 3, Inject the Dependency at Runtime (Java)

```java
public class Asset {
    private double myValue;
    private int myTerm;
    private AssetCap myAssetCap;

    public Asset (double value, int term) {
        this(new AssetCapImpl(), value, term);
    }

    public Asset (AssetCap assetCap, double value, int term) {
        this.myAssetCap = assetCap;
        this.myValue = value;
        this.myTerm = term;
    }

    public double getWriteOff(int IgnoreYear) {
        return this.myAssetCap.capValue(this.myValue) / this.myTerm;
    }
}
```

Remember, this is only one way of doing this...

An Often-Missed Issue...

- We know that Asset works
- We know that AssetCap works
- We have 100% coverage... or do we?

- Is there a change that we could accidentally make in the future, that would break the system even though all tests would still pass?
public class Asset {
    private double myValue;
    private int myTerm;
    private AssetCap myAssetCap;

    public Asset (double value, int term) {
        this(new AssetCapImpl(), value, term);
    }

    public Asset (AssetCap assetCap, double value, int term) {
        this.myAssetCap = assetCap;
        this.myValue = value;
        this.myTerm = term;
    }

    public double getWriteOff(int IgnoreYear) {
        return this.myValue / this.myTerm;
    }
}

...and you’ll still show 100% code coverage too!

//Pseudocode
public ReallyImportantClass {
    public void GoshAwfulImportantBehavior(int param) {
        //Lots of complicated stuff that has no return
    }

    public int Foo() {
        int i;
        i++; i++; i++; i++; i++; i++; i++; i++; i++; i++; i++; i++; i++;
        i++; i++; i++; i++; i++; i++; i++; i++; i++; i++; i++; i++; i++;
        i++; i++; i++; i++; i++; i++; i++; i++; i++; i++; i++; i++; i++;
        i++; i++; i++; i++; i++; i++; i++; i++; i++; i++; i++; i++; i++;
        i++; i++; i++; i++; i++; i++; i++; i++; i++; i++; i++; i++; i++;
        return i;
    }

    *http://www.sustainabledd.com
    (Thanks to Paddy Healey for the example)
Missing Test

- Our tests to this point are about behavioral boundaries
- The other kind of test we often need are tests of proper workflow
- Mocks can help us here too, if we create them properly

- Be careful not to overdo this

- You don’t want more coupling between your tests and production code than you need to form a complete specification

A Slightly Smarter Mock (Java)

class MockAssetCap implements AssetCap {
    private boolean gotCalled = false;

    public double capValue(double aValue) {
        gotCalled = true;
        return aValue;
    }

    public boolean didGetCalled() { return gotCalled; }
}
A Work-Flow Test (JUnit)

```java
@Test
public void testAssetUsesAssetCap(){
    MockAssetCap mock = new MockAssetCap();
    Asset testAsset = new Asset(mock, any.nextDouble(), any.nextInt());
    testAsset.getWriteOff(1);
    assertTrue(mock.didGetCalled());
}
```

Conclusions, So Far

- In TDD, good tests are unique
- In TDD, the suite is written as a complete spec
  - All behavioral boundaries, workflows, and constants
- The “D” in TDD is about “Driven” but also indicates a role in helping to see alternative “Designs”
- ...good design is also what patterns are about!
The most widely known version today appears in Stephen Hawking's 1988 book *A Brief History of Time*, which begins with an anecdote about an encounter between a scientist and an old lady:

A well-known scientist (some say it was Bertrand Russell) once gave a public lecture on astronomy. He described how the Earth orbits around the sun and how the sun, in turn, orbits around the centre of a vast collection of stars called our galaxy.

At the end of the lecture, a little old lady at the back of the room got up and said: "What you have told us is rubbish. The world is really a flat plate supported on the back of a giant tortoise."

The scientist gave a superior smile before replying, "What is the tortoise standing on?"

"You're very clever, young man, very clever," said the old lady. "But it's turtles all the way down."
The Decorator Pattern

The Decorator Pattern In Action

Easily Testable

Dependant on Delegation

"Turtles all the way down"
The Decorator Pattern, Tested

A Mock Turtle

Decorator Example Code (Java)

```java
public interface StringProcess {
    public abstract void ProcessString(String s);
}

public abstract class StringDecorator implements StringProcess {
    private StringProcess myNextProcess;
    public StringDecorator(StringProcess aNextProcess) {
        myNextProcess = aNextProcess;
    }
    public void ProcessString(String s) {
        String decoratedString = decorateString(s);
        myNextProcess.ProcessString(decoratedString);
    }
    protected abstract String decorateString(String s);
}

public class UpperDecorator extends StringDecorator {
    public UpperDecorator(StringProcess aNextProcess) {
        super(aNextProcess);
    }
    protected String decorateString(String s) {
        return s.toUpperCase();
    }
}
```
import org.easymock.*;

public class UpperDecoratorTest {

    private UpperDecorator testDecorator;
    private MockControl control;
    private StringProcess mockConcreteProcess;

    @Test public void testUpperDecoration() {
        control = MockControl.createControl(StringProcess.class);
        mockConcreteProcess = (StringProcess) control.getMock();
        testDecorator = new UpperDecorator(mockConcreteProcess);

        mockConcreteProcess.ProcessString("AAA");
        control.replay();
        testDecorator.ProcessString("aaa");
        control.verify();
    }
}

An Empowering Question

- Design is about decisions
- When you:
  - Understand how tdd tests work
  - Understand how dependencies and can be tested with mock objects
  - Let patterns suggest specific techniques for testing
- Then you can ask the question:
  - How Would I Test This?
- ...whether you actually write the test or not
Summary

“10 Things You Might Not Know About Sustainable Test-Driven Development”

Point 1

- A test should be *proven to fail* for a well-understood, narrowly-defined reason. If this is not proven, then it is possible that a test will never fail, and thus the behavior it defines is not guaranteed. When a test fails, it delivers a large part of its value, since it gives us critical information for finding the source of the failure. This is why we write the test to fail, and then change the production code to make it pass -- This proves that the test can fail, and for the reason we intend.
Point 2

- A test should *never* fail for a reason *other* than it was intended to. When a test fails, yet the behavior it was designed to cover is working properly, then the test misleads us as to the source of the failure. Often this is due to unexpected/undesirable coupling in the production code.

Point 3

- No *other test* should fail for the *same reason* that a given test will fail. Multiple test failures for a single cause creates maintenance problems when new features are added to the system, when existing features are removed, or when there are changes for performance, security, or any other reason. Agility requires that we embrace change, and large numbers of test failures due to a single change inhibits this.
Point 4

- Each test should make a unique behavioral distinction in the system that no other test makes. This will ensure that points 1, 2, and 3 are adhered to.

Point 5

- Tests must be written within a well-understood scope. For example, if we are testing that an algorithm properly calculates tax, we do not need to write a test to ensure that the code being tested does not also reset the system clock, or format the disk, or anything else outside the scope of the problem. We test for errors, not malice.
Point 6

- A test tests all those things which are in scope, but are not under the test’s control. Therefore, if we wish to test a single, narrowly-defined thing, then everything else in scope must be brought under the test’s control. We use techniques such as mocking, shunting, endo-testing, dependency injection, etc… to accomplish this. Developers must be well-trained in these techniques if the test suite is to remain maintainable as the project matures.

Point 7

- The role of a code coverage tool in TDD is to verify, after a test is removed, that the system is indeed still fully covered. For example when a test is found to duplicate the distinction of another test, one of them must be removed. If code coverage reveals an uncovered path, then either the test was not actually a duplicate, or there is “dead” (unused, unneeded) code in the system.
Point 8

- The writing of the tests is how we conduct analysis, and thus the resulting test suite becomes the functional specification of the system. It has advantages over traditional specification documents in that the suite is:
  - Written in technical language (code)
  - Executable/verifiable by the computer without human intervention and analysis.
- For the suite to fill this role, it must be extremely readable.

Point 9

- The level of complexity in testing is an indicator of the quality of the design of the code being tested. For example, when tests classes are significantly larger than the classes they test, this can indicate that the tested class does too many things. Or, a test that requires a large number of instances can indicate excessive/unwanted coupling in the system. Therefore, testability is a driver of good design.
Test suites must run **fast** so they can be run **frequently** without excessive cost. Therefore all dependencies on the database, the GUI, the network, etc... that would make tests run slowly must be mocked for testing. These external dependencies should be tested as well, but they can be tested by integration tests that are not run frequently, and are therefore not expected to run as fast.

---

**Resources for Further Investigation**

- **Sustainable Test-Driven Development Book in Progress**
  - [www.sustainabletdd.com](http://www.sustainabletdd.com)

- **Design Patterns:**
  - [www.netobjectivesrepository.com](http://www.netobjectivesrepository.com)

- **Testing, Refactoring, Qualities**
  - [www.netobjectives.com/resources](http://www.netobjectives.com/resources)
Net Objectives Pattern Repository
http://www.netobjectivestest.com/PatternRepository
Register for newsletter and access to resources:
http://www.netobjectives.com/register
*Net Objectives SPC Training in Seattle, Oct 27-30*
www.netobjectives.com/events