Test Driven Development

in Scala with Daniel Hinojosa

The Addison
Test-Driven Development
By Example
Kent Beck

The Process

“

1. Quickly add a test.
2. Run all tests and see the new one fail.
3. Make a little change.
4. Run all tests and see them all succeed.
5. Refactor to remove duplication.

”

Kent Beck – Test Driven Development By Example 2003

“

1. Write a failing test.
2. Write code to make it pass.
3. Repeat steps 1 and 2.
4. Along the way, refactor aggressively.
5. When you can't think of any more tests, you must be done.

”

Neal Ford – Evolutionary Architecture and Emergent Design 2009

Benefits

- Promotes design decisions up front
- Allows you and your team to understand your code
- Model the API the way you want it to look
- Means of communicating an API before implementation
- Avoids Technical Debt
- Can be used with any programming language

```
class TaxCode(...) {
   def applyTaxAndSurcharge(order:Order):Order {
     val total = order.total();
     if (order.after(LocalDate.of(“2001-01-01”))) {
         if (order.customer.getState().equals(“FL") ||
            order.customer.getState().equals("NY"))
            order.applySurcharge(Money(10))
            order.applyTax(Money(total *
              TaxWS.findTaxRate(order.customer.getState()))
         else
            order.applyTax(Money(30))
         }
     } else {
       order.applyTax(0)
       order.applySurcharge(0)
     }
     order
   }
}
```

"Game"ifying Development

Consider each fail test a challenge

Less Debugging!

Disadvantages

- People find it difficult and unintuitive at first
- Requires team investment
- Many do not see the advantages until it is too late

Bob Martin's Three TDD Laws

“

You may not write production code until you have written a failing unit test.

”

Bob Martin – Clean Code 2008

“

You may not write more of a unit test than is sufficient to fail, and not compiling is failing.

”

Bob Martin – Clean Code 2008

“

You may not write more production code than is sufficient to pass the currently failing test.

”

Bob Martin – Clean Code 2008

Adopting TDD

Adopting TDD As An Individual

- Practice Makes Perfect
- TDD every new method, class, or function
- Contribute to an open source project for fun
- Use testing when learning a new language!

* You should learn a new language every year anyway

Adopting TDD As A Team

- For Green Field Projects
  - Start Immediately!
  - Prevent Technical Debt
- For Brown Field Projects
  - Adopt "Testing Thursdays or Fridays" if affordable
  - Powermock in Java if necessary*

* Powermock in Java, or any class manipulation utility is usually a sign of bad design

Measuring and Monitoring TDD

- Use Code Coverage to check that code is being tested
  - Cobertura
  - Emma
  - Jacoco
- Employ Pair Programming
- Employee Code Review
  - Non-TDD Code is easy to spot

Levels of Testing

Load
Unit
Integration
Security
QA
UAT
System

Load
Unit
Integration
• Fast with Instant Feedback
• Bug Replication
• Isolated, Repeatable
• Done by the Developers, not QA

Unit
Integration
System
• Testing already unit-tested (TDD) tests with each other
• Testing separate team developed components
• Done by Developers, Perhaps other Software Testers

- Can the system as a whole handle extreme demand?
- Typically can employ Load Testing Software like Apache JMeter or https://artillery.io/
- Done typically by Developers, Testers, QA

- How does the system test as a whole?
- Does it meet the entire criteria including UI
- Generally done by QA or other testing teams

Unit

UAT

System

- Is the system secure?
- Can it be hacked or compromised
- No holds barred testing
- Results in creation of system tests and Unit Test to plug up
- Done by developers, technical managers, testers, or QA

Load
Unit
Integration
Security
QA
UAT
System
Bug!

Load
Unit
Integration
Security
QA
UAT
System
Duplicated Bug!
Bug!

Infrastructure Changes Required

- TDD Adherence among all developers
- Continuous Integration Server:
  - Performs unit testing every hour
  - Breaks the build if agreed testing metrics are not met

Bamboo

go®

Travis CI

What to test?

```
#include <iostream>
using namespace std;
void Shop(){
    cout << "The item store is not finished yet\n" << endl;
    int Mainmenu();
    Mainmenu();

    }

void Character(){

    int PlayerHp, PlayerStr, PlayerMana;

    cout << "Your character has:\n\n";

    cout << PlayerHp << " Health" << endl;
    cout << PlayerStr << " Strength" << endl;
    cout << PlayerMana << " Mana" << endl;

    cout << endl;

    int Mainmenu();
    Mainmenu();
    }

void Inventory(){

    cout << "Your inventory contains:\n" << endl;

    int Mainmenu();
    Mainmenu();

    }


void Wilderness(){
    cout << "You search the jungle and find:\n" << endl;

    string Creatures[10];
    Creatures[0] = "1";
    Creatures[1] = "2";
    Creatures[2] = "3";
    Creatures[3] = "4";
    cout << Creatures << " Is really dangerous " << endl;

    int Mainmenu();
    Mainmenu();
    }
void Mainmenu(){
    string choice;

    cout << "1: Attack creature" << endl;
    cout << "2: Buy equipment" << endl;
    cout << "3: Inventory" << endl;
    cout << "4: Stats" << endl;

        // invalid option returns to this
        InvalidOption:

    cin >> choice;
    if (choice == "1"){
                Wilderness ();

    else if (choice == "2"){
                Store();

    else if (choice == "3"){
                Inventory();

    else if (choice == "4"){
                Character();
                }

    else{
        cout << "You have not chosen a valid option" << endl;
        goto InvalidOption;

    }


int main()
{
    int PlayerHp = 100, PlayerStr = 20, PlayerMana = 50;
    Mainmenu ();
}
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
// Exercise the symbol table mechanism of the compiler in an effort to
// take a unreasonable about of time compiling

#define PTR(T) T*
#define CONST(T) T const
#define FUNC(NAME,RTYPE,ARG) RTYPE NAME(ARG)
#define PFFN(NAME,RTYPE,ARG) FUNC(*(*NAME),RTYPE,ARG)

// Forward declaration of repeated OO method pointers
typedef void* (*cctor_ptr)(void*This, void*This0,...);
typedef void* (*dtor_ptr)(void*This);

// Assumes three var-args: sizeof(payload type), cctor, dtor
void* default_ctor(void*This, ...){
  // null in variadic bits
  va_list list;
  int*this=(int*)This;
  int sizeva_arg(list,int);
  cctor_ptr cctor=va_arg(list,cctor_ptr);
  dtor_ptr dtorva_arg(list,dtor_ptr);
  va_end(list);
  // process
  if (*this) this = malloc(size);
  if (this){
    memset(this,size,0);
    /* various dodges to install the cctor and dtor in the write places */
  }
  return this;
}
// Copies the payload from that to this;
void* default_cctor(void*restrict This, void* restrict That, ...){
  // null in variadic bits
  va_list list;
  va_start(list,That);
  int sizeva_arg(list,int);
  cctor_ptr cctor=va_arg(list,cctor_ptr);
  dtor_ptr dtorva_arg(list,dtor_ptr);
  va_end(list);
  // process
  if (*This) this = malloc(size);
  if (this){
    memcpy(This,That,size);
    /* various dodges to install the cctor and dtor in the write places */
  }
  return This;
}
// Assumes that his was allocated with malloc, does not use varargs
void* default_dtor(void*This, ...){
  free(This);
  return NULL;
};

#define DECLARE_STRUCT(N) struct SAMNAME_##N
#define TYPEDEF_ACCESSOR(N,T) typedef struct SAMNAME_##N*(*PFFN##N##_ptr,CONST(PTR(T)),PTR(CONST(struct SAMNAME_##N)));
#define TYPEDEF_STRUCT(N,T) typedef struct SAMNAME_##N (*PFFN##T)(); cctor_ptr cctor; dtor_ptr dtor; PFFN##N##_ptr FFN##N;} SAMN
#define OO_STRUCT(N,T) DECLARE_STRUCT(N); TYPEDEF_ACCESSOR(N,T); TYPEDEF_STRUCT(N,T)

OO_STRUCT(1,char);
OO_STRUCT(2,int);
OO_STRUCT(3,double*);
OO_STRUCT(4,S3);
OO_STRUCT(5,S4);
OO_STRUCT(6,S5);
OO_STRUCT(7,S6);
OO_STRUCT(8,S7);

#define SUBSCOPE(A) { \
    S1*SAM1=default_ctor(NULL,sizeof(char),default_cctor,default_dtor); \
    S2 SAM2; default_ctor(&SAM2,sizeof(int),default_cctor,default_dtor); \
    S3*SAM3=default_ctor(NULL,sizeof(double*),default_cctor,default_dtor); \
    S4 SAM4; default_ctor(&SAM4,sizeof(S4),default_cctor,default_dtor); \
    S5*SAM5=default_ctor(NULL,sizeof(S5),default_cctor,default_dtor); \
    S6*SAM6=default_ctor(NULL,sizeof(S7),default_cctor,default_dtor); \
    \
#define SUBSCOPE2(A,B) { \
    S1*SAM1=default_ctor(NULL,sizeof(S4),default_cctor,default_dtor); \
    S4 SAM2; default_ctor(&SAM2,sizeof(S4),default_cctor,default_dtor); \
    SUBSCOPE(A); SUBSCOPE(B);                                           \
    \
#define SUBSCOPE3(A,B,C) { \
    S1*SAM1=default_ctor(NULL,sizeof(double*),default_cctor,default_dtor); \
    S2 SAM2; default_ctor(&SAM2,sizeof(int),default_cctor,default_dtor); \
    S4*SAM3=default_ctor(NULL,sizeof(S4),default_cctor,default_dtor); \
    SUBSCOPE2(A,B); SUBSCOPE3(B,C) SUBSCOPE2(A,B); \
    \
#define SUBSCOPE4(A,B,C,D) { \
    S1*SAM1=default_ctor(NULL,sizeof(char),default_cctor,default_dtor); \
    S2*SAM2=default_ctor(NULL,sizeof(int),default_cctor,default_dtor); \
    S3*SAM3=default_ctor(NULL,sizeof(double*),default_cctor,default_dtor); \
    S4 SAM4; default_ctor(&SAM4,sizeof(S3),default_cctor,default_dtor); \
    SUBSCOPE4(A,B,C) SUBSCOPE4(B,A,D) SUBSCOPE3(A,C,D) SUBSCOPE3(B,C,D); \
    \
#define SUBSCOPE5(A,B,C,D,E) { \
    S1*SAM1=default_ctor(NULL,sizeof(S4),default_cctor,default_dtor); \
    S2*SAM2=default_ctor(NULL,sizeof(S5),default_cctor,default_dtor); \
    S3*SAM3=default_ctor(NULL,sizeof(S6),default_cctor,default_dtor); \
    SUBSCOPE2(A,B,C,D) SUBSCOPE3(A,B,C,E) SUBSCOPE3(A,B,C,E); \
    SUBSCOPE2(A,C,D,E) SUBSCOPE2(B,C,D,E); \
    \
#define SUBSCOPE7(A,B,C,D,E,F) { \
    S1 SAM1; default_ctor(&SAM1,sizeof(S4),default_cctor,default_dtor); \
    S4 SAM2; default_ctor(&SAM2,sizeof(S5),default_cctor,default_dtor); \
    S6*SAM6=default_ctor(NULL,sizeof(S7),default_cctor,default_dtor); \
    SUBSCOPE3(A,B,C,D,E) SUBSCOPE3(A,B,C,D,F) SUBSCOPE3(A,B,C,E,F); \
    SUBSCOPE3(A,B,D,E,F) SUBSCOPE3(A,C,D,E,F) SUBSCOPE3(B,C,D,E,F); \
  }

  int main(){
    SUBSCOPE7(A,B,C,D,E,F);
  }

  #include <math.h>
  #include <stdio.h>

  // Some random bits we'll need
  #define TWOPOWER(n) (1<<(32-1-n))
  #define TWO(n) (2*(n))
  #define NPLUSONE(n) ((n)+1)
  #define TWONPLUSONE(n) NPLUSONE(TWO(n))
  #define FACT(n) (tgamma(NPLUSONE(n)))

  // The Euler series:
  //                                  2^(2n) * (n!)^2      z^(2n+1)
  // atan(z) = \sum_n=0^\infty  ------------------ * ----------------
  //                                    (2n+1)!        (1 + z^2)^(n+1)
  #define TERMEULER(n,z) (pow(2,TWO(n))*                          \
              FACT(n)*FACT(n)*                                    \
              pow(z),TWONPLUSONE(n))/                             \
```

Any program can be a long series of functions and statements

But we don’t do that.

- We break things up into logical sections
- We also want to reuse those sections
- Those things must form a cohesive unit
- Single Responsibility Principle

We test the logical sections

What not to test?

What not to test?

- The `main` method, this is where wiring takes place
- GUIs
  - There are varying testing frameworks to test GUIs
    - Selenium Web Driver (Web, JavaScript)
    - Fest-Swing (Java Swing Applications)
    - TestFX (Java FX Applications)
- Code in other dependencies, should be doing their own testing*
- What about getters and setters? Yes*

* Strong Opinion

ScalaTest

- Scala Testing Framework
- Started by Bill Venners
- Varying Testing Formats
- Varying Matchers

What is Code Coverage?

- Analyzes what production code has been covered by Unit Testing
- Line Coverage tests which line have been covered by tests
- Branch Coverage tests if all conditions have been tested in your `if`, `else`, `else`, `while`, `do`
- Cyclomatic Complexity
  - Algorithm to determine code complexity
  - Aim for 5 (even if the max should be 10)

How does code coverage work?

- Code that is compiled is then instrumented at byte code level
- Detects when a line of code is process by thread

Example JVM Coverage Report

SCoverage generated at Sun Dec 08 11:16:27 GMT 2013

Lines of code:	169	Files:	8	Classes:	8	Methods:	9
Lines per file	21.125	Packages:	1	Clases per package:	8.00	Methods per class:	1.13
Total statements:	89	Invoked statements:	67	Total branches:	12	Invoked branches:	9
Statement coverage:	75.28 %			Branch coverage:	75.00 %		

Class	Source file	Lines	Methods	Statements	Invoked	Coverage	Branches	Invoked	Coverage
ClientActor	ClientActor.scala	34	2	21	15	71.43 %	4	3	75.00 %
CreditEngine	CreditEngine.scala	13	1	7	7	100.00 %	2	2	100.00 %
InstrumentLoader	InstrumentLoader.scala	24	1	6	5	83.33 %	2	1	50.00 %
OrderEngine	OrderEngine.scala	22	1	14	14	100.00 %	0	0	0.00 %
Platform	Platform.scala	14	1	14	0	0.00 %	0	0	0.00 %
PriceEngine	PriceEngine.scala	31	1	10	10	100.00 %	2	2	100.00 %
QuoteCache	QuoteCache.scala	16	1	8	8	100.00 %	0	0	0.00 %
RandomQuoteGenerator	QuoteGenerator.scala	15	1	9	8	88.89 %	2	1	50.00 %

Build Tools

What is a build tool?

- Tool that manages a project
- Set of commands that allow to describe build a project
  - Compiling
  - Testing
  - Packaging
  - Cleaning Binaries
  - Deployment
- Any build tool can be used with TDD

SBT

- Official Name: Simple Build Tool
- Scala Build Tool
- Flexible
- Out of the box is simple
- Creating your tasks can be difficult

Lab: Retrieve Project

- For Scala Based Project: http://www.github.com/dhinojosa/scala-tdd

Either clone or download project of your choice:

Lab: Run A Quick Test

- `cd scala-tdd`
- `sbt test`

or

- `cd scala-tdd`
- `sbt`
- `test`

Lab: Run A Quick Test

- `cd scala-tdd`
- `sbt test`

or

- `cd scala-tdd`
- `sbt`
- `test`

Setting up your IDEs

Quick Note About Java IDEs

- Integrated Development Environment
- They are tremendous for full projects
- They are terrible for simple file editing
- Two popular ones for the JVM
  - Eclipse
  - IntelliJ IDEA

Eclipse

- http://www.eclipse.org
- Variant Eclipse versions depending on focus (Spring STS, Jboss Tools, Scala-IDE, etc)
- Open Source
- Pluggable Features
- Most popular IDE among JVM developers

IntelliJ IDEA

- http://www.jetbrains.com/idea
- Community Edition (Free)
- Ultimate Edition (Various Pricing Packages)
- Has more keyboard shortcut bindings than Eclipse
- Pluggable Features
- Easy to Use

Learning Shortcuts for IDEs

Why Keyboard Shortcuts?

- You waste an enormous amount of time using a mouse
- Efficient development requires you know most if not all keyboard shortcuts
- Learn one or two keyboard shortcuts a day (It adds up)
- If you need to perform a task, look up the shortcut until it is committed to memory
- It is essential for successful Test Driven Development

Essential Shortcuts (Windows/Linux)

- `CTRL+S` **- Save**
- `CTRL+C` **- Copy**
- `CTRL+X` **- Cut**
- `CTRL+V` **- Paste**
- `CTRL+Z` **- Undo**
- `CTRL+SHIFT+Z` or `CTRL+R` **- Redo**
- `ALT+TAB` **– Switch Applications**

Essential Shortcuts (Mac OS X)

- ⌘+S **- Save**
- ⌘+C **- Copy**
- ⌘+X **- Cut**
- ⌘+V **- Paste**
- ⌘+Z **– Undo**
- ⌘ + SHIFT(⇧) + Z **– Redo**
- ⌘ + TAB **– Switch Applications**

Essential IntelliJ Shortcuts (Linux/Windows)

- `CTRL + SHIFT + A` – **Keyboard Shortcut Lookup**
- `CTRL + E` - **Recent Files**
- `CTRL + SHIFT + F12` – **Maximize Screen**
- `CTRL + Y` – **Delete Line**
- `SHIFT + ALT + L` – **Format Code**
- `ALT + ENTER` – **Context Help**
- `CTRL + SHIFT+ T` – **Toggle between Test and Class**
- `CTRL + SHIFT + F10` – **Run**

https://www.jetbrains.com/idea/docs/IntelliJIDEA_ReferenceCard_Mac.pdf

Essential IntelliJ Shortcuts (Mac OSX)

- SHIFT(⇧) + ⌘ + A – **Toggle Keyboard Shortcut Help**
- ⌘ + E - **Recent Files**
- ⌘ + SHIFT(⇧) + F – **Toggle Fullscreen**
- ⌘ + DELETE(⌫) – **Delete Line**
- ⌘ + OPTION(⌥) + L – **Format Code**
- OPTION(⌥) + ENTER(⏎) – **Context Help**

https://www.jetbrains.com/idea/docs/IntelliJIDEA_ReferenceCard_Mac.pdf

Editor Can't Toggle?

Consider using split pane

calculator-spec.js — test — ~/Development/danno-test-project

```
import chai,{
  expect,
} from 'chai';
import add,{foo} from '../src/calculator.js';
import chaiAsPromised from 'chai-as-promised'
chai.use(chaiAsPromised)

describe('add', () => {
  describe('addition', () => {
    it('should add two numbers', () => {
      expect(add(2, 3)).to.equal(5);
    });
  });
  describe('negative numbers', () => {
    it('should add consider negative numbers', (
      expect(add(2, -3)).to.equal(-1);
    });
  });
  describe('the hell is going on here', function
    it('should smell great', function() {
      expect(add(5, 10)).to.equal(15);
    });
  });
  describe('obj.blah', () => {
    it('should be mocked as...', () => {
      const baz = {blah : () => 'butt'}
```


```
import chai,{
  expect,
} from 'chai';
import add,{foo} from '../src/calculator.js';
import chaiAsPromised from 'chai-as-promised'
chai.use(chaiAsPromised)

describe('add', () => {
  describe('addition', () => {
    it('should add two numbers', () => {
      expect(add(2, 3)).to.equal(5);
    });
  });
  describe('negative numbers', () => {
    it('should add consider negative numbers', (
      expect(add(2, -3)).to.equal(-1);
    });
  });
  describe('the hell is going on here', function
    it('should smell great', function() {
      expect(add(5, 10)).to.equal(15);
    });
  });
  describe('obj.blah', () => {
    it('should be mocked as...', () => {
      const baz = {blah : () => 'butt'}
```


test/calculator-spec.js 1:1 LF UTF-8 JavaScript master

Atom Keyboard Shortcuts

https://github.com/nwinkler/atom-keyboard-shortcuts

Sublime Keyboard Shortcuts

http://docs.sublimetext.info/en/latest/reference/keyboard_shortcuts_win.html

Sublime Keyboard Shortcuts

http://docs.sublimetext.info/en/latest/reference/keyboard_shortcuts_osx.html

VSCode Keyboard Shortcuts

https://code.visualstudio.com/shortcuts/keyboard-shortcuts-windows.pdf

VSCode Keyboard Shortcuts

https://code.visualstudio.com/shortcuts/keyboard-shortcuts-macos.pdf

ScalaTest Styles

ScalaTest Styles

- Varying Styles
- http://www.scalatest.org/user_guide/selecting_a_style

Assertion Language Libraries

Standard Scala Test Assertions

```
val left = 2
val right = 1
assert(left == right)
```

Standard Scala Test Assertions

```scala
assert(a == b || c >= d)
// Error message: 1 did not equal 2, and 3
// was not greater than or equal to 4

assert(xs.exists(_ == 4))
// Error message: List(1, 2, 3) did not contain 4

assert("hello".startsWith("h") && "goodbye".endsWith("y"))
// Error message:
// "hello" started with "h", but "goodbye" did not end with "y"

assert(num.isInstanceOf[Int])
// Error message: 1.0 was not instance of scala.Int

assert(Some(2).isEmpty)
// Error message: Some(2) was not empty
```

Assertion Responses

```
assert(a == b || c >= d)
// Error message: 1 did not equal 2, and 3
// was not greater than or equal to 4

assert(xs.exists(_ == 4))
// Error message: List(1, 2, 3) did not contain 4

assert("hello".startsWith("h") && "goodbye".endsWith("y"))
// Error message:
// "hello" started with "h", but "goodbye" did not end with "y"

assert(num.isInstanceOf[Int])
// Error message: 1.0 was not instance of scala.Int

assert(Some(2).isEmpty)
// Error message: Some(2) was not empty
```

Clever Tricks

```
assertDoesNotCompile("val a: String = 1")
```

```
assertTypeError("val a: String = 1")
```

```
assertCompiles("val a: Int = 1")
```

http://www.scalatest.org/user_guide/using_assertions

ScalaTest Matchers

- Set of Domain Specific Language Features
- Bring in trait `Matchers`
- Typical format: `result should be (3)`
- http://www.scalatest.org/user_guide/using_matchers

cyber-dojo.org

the place to practice programming

setup a new practice session

enter a practice session

review a practice session

100% of your donation buys Raspberry Pi computers to help children learn to program

please donate

Group Lab: Caesar Cipher

“Foo” ⇒ **+5** ⇒ “Ktt”

Individual Lab/Homework!
Fizz Buzz

Fizz Buzz Test

The "Fizz-Buzz test" is an interview question designed to help filter out the 99.5% of programming job candidates who can't seem to program their way out of a wet paper bag. The text of the programming assignment is as follows:

> *"Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz"."*

Don't believe it,
it's more like 60%

http://wiki.c2.com/?FizzBuzzTest

Scala Check

Property Based Testing

- Specification-Based Testing
- A *specification* is a definition of a program's behavior in the general case. Preferably, a specification should not deal with concrete examples of program functionality. Instead it should provide an abstract description of the behavior.

Property Based Testing Benefits

- **Better test coverage -** You have larger amount of tests rigorously hitting your Subject Under Tests
- **Harder to miss edge cases -** Edge Cases are built in, since different obscure inputs are generated
- **Wrong tests are simplified -** Given a wide range of testing inputs the reports by ScalaCheck are great simplified using a process call shrinking

Test

Tests are mostly formal and easy to verify, but they don't give us assurance about the overall program behavior

Specs

Specifications give us a more complete picture of the program but they are often informal and difficult to verify formally.

Tests are used to verify specific cases of a specification.

Sample Specification

```scala
import org.scalacheck.Prop.forAll
import org.scalacheck.Properties
```

```scala
object StringSpecification extends Properties("String Specification") {
  property("startsWith") = forAll { (a: String, b: String) =>
    (a + b).startsWith(a)
  }

  property("concatenate") = forAll { (a: String, b: String) =>
    (a + b).length > a.length && (a + b).length > b.length
  }

  property("substring") = forAll { (a: String, b: String, c: String) =>
    (a + b + c).substring(a.length, a.length + b.length) == b
  }
}
```

Specification Results

```
+ String Specification.startsWith: OK, passed 100 tests
+ String Specification.substring: OK, passed 100 tests.
! String Specification.concatenate: Falsified after 0 passed tests.
```

Rapid Response

```
property("this should fail after a while") = forAll {(x:Int) =>
    x.abs >= 0
}
```

```
 ! Failed Specs.this should fail after a while: Falsified after 3 passed tests.
[info] > ARG_0: 0
[info] > ARG_0_ORIGINAL: -2147483648
```

Constraints

```scala
import org.scalacheck.Prop.{BooleanOperators, forAll}

property("this should succeed") = forAll {(x:Int) =>
    x > Int.MinValue ==>
    x.abs >= 0
}
```

Shrinking

```
def brokenReverse[A](xs: List[A]): List[A] = {
    if (xs.length > 3) xs else xs.reverse
}

property(
    """When things need to be reported it will
      |  find the easiest combination""".stripMargin) = forAll{
    xs:List[Int] => xs.length > 0 ==>
        (xs.last == brokenReverse(xs).head)
}
```

```
> ARG_0: List("0", "0", "0", "1")
> ARG_0_ORIGINAL: List("2147483647", "951598451", "-1677906989",
"850298247")
```

Constraint Exhaustion

```
property("exhaustion is a possible result") = forAll
{ (xs:List[Int]) =>
    (xs.length > 3 && xs.length < 6)  ==>
    (xs.last == myReverse(xs).head)
}
```

```
! Failed Specs.exhaustion is a possible result: Gave up
after only 13 passed tests. 501 tests were discarded.
```

Gen

- Generator
- Can produce any Value or Subset of Values
- Many provided
- Can create your own

Out of the Box Gens

```scala
import org.scalacheck.Gen

def alphaChar: Gen[Char]

def alphaStr: Gen[String]

def posNum[T](implicit n: Numeric[T]): Gen[T]

def listOf[T](g: Gen[T]): Gen[List[T]]

def listOfN[T](n: Int, g: Gen[T]): Gen[List[T]]
```

Monadic Custom Generation

```scala
val capitalizedString: Gen[String] = for {
    first <- Gen.alphaUpperChar
    rest <- Gen.listOf(Gen.alphaLowerChar)
} yield (first :: rest).mkString
```

Monadic Case Class Custom Generation

```scala
case class Track(name: String, seconds: Int)

case class Album(name: String, artist: String, tracks: List[Track])

val genTrack: Gen[Track] = {
    for {
        name <- Gen.alphaStr
        seconds <- Gen.choose(1, 8 * 60)
    } yield Track(name, seconds)
}

val genAlbum: Gen[Album] = {
    for {
        name <- Gen.alphaStr
        artist <- Gen.alphaStr
        tracks <- Gen.containerOf[List, Track](genTrack)
    } yield Album(name, artist, tracks)
}

private val genAlbums = Gen.listOf[Album](genAlbum)
```

Arbitrary

```
case class Track(name: String, seconds: Int)

case class Album(name: String, artist: String, tracks:
List[Track])

val genTrack: Gen[Track] = {...}

val genAlbum: Gen[Album] = {...}

implicit val album:Arbitrary[Album] = Arbitrary(genAlbum)

property("check that an album will work") = forAll {a:Album =>

}
```

ScalaTest Integration

```
forAll { (n: Int, d: Int) =>

  whenever (d != 0 && d != Integer.MIN_VALUE
      && n != Integer.MIN_VALUE) {

    val f = new Fraction(n, d)

    if (n < 0 && d < 0 || n > 0 && d > 0)
      f.numer should be > 0
    else if (n != 0)
      f.numer should be < 0
    else
      f.numer should be === 0

    f.denom should be > 0
  }
}
```

Property Based Testing Best Practices

Start Simply

- **Start out simply (like TDD). What is unacceptable?**
- Examples:
  - Tested method should never return a negative integer
  - Output list must always be shorter than the input list

Not a Random Generator

- Since this is used for edge cases it doesn't perform well for random generation

Abandoning Specifications

- Leave the Specification incomplete if it becomes too involved
- There is a high risk of introducing bugs in the test and in the production code if they look alike
- Keep them simple and straightforward

Pit two instances against each other

You can use two or more test cases in the same property and base your specification on the relation between the inputs

```
val propRankTweet = Prop.forAll(genTweet, genTweet) {
   (tweet1,tweet2) =>
     val score1 = rankTweet(tweet1)
     val score2 = rankTweet(tweet2)
     if (tweet1.length <= tweet2.length) score1 >= score2
     else score1 < score2
 }
```

Reference Implementation

- Compare your results with a well established and trusted object!

java.util.HashMap implementation

SUT IntMap implementation

```
def equalMaps(hm: HashMap[Int,Any], im: IntMap[Any]) = {
   im.keys.forall(hm.containsKey) &&
   hm.keySet.containsAll(im.keys) &&
   im.keys.forall(k => im(k) == hm(k))
 }
```

Round Trip Properties

Pit the results against itself and verify if that works

```
property("reverse inverse") = Prop.forAll {
  xs: List[Int] => xs.reverse.reverse == xs
}
```

Day 2: Isolated Testing, Legacy Testing, and Machine Learning

Isolated Testing

Isolation

- The key to unit testing is isolation
- Mock, Stub, Dummy, or Fake dependencies
- Prefer to investigate an interface of the object you wish to inject.

Evidence of Dependencies

No evidence of a Dependency

```
def method1() {
      new Employee("Roger", "Moore");
}
```

Note: No equals in the method, this is a Unit

No evidence of a static dependency

```
def method2() {
    val resource =
        ServerInstance.find("/accounts/resource")
    resource.addDeposit(3000.00)
}
```

Note: No equals in the method, this is a Unit

Static Dependency Hard to Control

```
def method3() = {
   resource =
      ServerInstance.find("/accounts/resource")
   resource.addDeposit(3000.00)
   resource
}
```

Full Evidence of a Dependency

```
def method4(resource:Resource) = {
    resource.addDeposit(3000.00)
    resource
}
```

Full Evidence of a Dependency

```
def method4() = {
  new Employee("Sean", "Connery");  //OK
}
```

Full Evidence of a Dependency

```
def method5 ={
  Array(new Employee("Sean", "Connery"),
        new Employee("George", "Lazenby"),
        new Employee("Pierce", "Brosnan"),
        new Employee("Roger", "Moore"),
        new Employee("Timothy", "Dalton"),
                new Employee("Daniel", "Craig"));
}
```

Fakes

“

Objects that actually have working implementations, but usually take some shortcut which makes them not suitable for production (an in memory database is a good example).

”

Martin Fowler - http://martinfowler.com/articles/mocksArentStubs.html

Dummies

“

Objects that are passed around but never actually used. Usually they are just used to fill parameter lists.

”

Martin Fowler - http://martinfowler.com/articles/mocksArentStubs.html

Stubs

“

Stubs provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test. Stubs may also record information about calls, such as an email gateway stub that remembers the messages it 'sent', or maybe only how many messages it 'sent'.

”

Martin Fowler - http://martinfowler.com/articles/mocksArentStubs.html

Mocks

“

Mocks are ... objects pre-programmed with expectations which form a specification of the calls they are expected to receive.

”

Martin Fowler - http://martinfowler.com/articles/mocksArentStubs.html

Functions

- If you need multiple methods from a single object then mock
- If you are needing a test double for one action, use a function
- Potential to minimize the use of mocks & stubs, save time

The thing that you will find is the more you adapt either functions or higher abstractions *the more likely you will not need mocks.*

Mocking Frameworks

EasyMock

- One of the first mocking frameworks
- http://www.easymock.org
- Strict by default

Mockito

- Flexible Mocking Framework
- Most popular of the mocking framework
- Lenient

ScalaMock

- Flexible Mocking Framework
- Most popular of the mocking framework
- Scala Based
- Ability to mock and stub
  - Classes
  - Objects
  - Functions
- Two Flavors
  - Expectations-First Style
  - Record-then-Verify (Mockito Style)

ScalaMock Example

```scala
"drawLine" should "interact with Turtle" in {
  // Create mock Turtle object
  val mockedTurtle = mock[Turtle]

  // Set expectations
  (mockedTurtle.setPosition _).expects(10.0, 10.0)
  (mockedTurtle.forward _).expects(5.0)
  (mockedTurtle.getPosition _).expects().returning(15.0, 10.0)

  // Exercise System Under Test
  drawLine(mockedTurtle, (10.0, 10.0), (15.0, 10.0))
}
```

Discussion: Finding overdue library books.

Best Practices and Advice

Code Reviewer Guide

http://misko.hevery.com/code-reviewers-guide/

Exception Handling

- One Exception can be thrown for multiple reasons
- It is best to check the messages to avoid false positives

AntiPattern: Mocks returning Mocks

- Mocks returning Mocks shows bad form
- Having more than 2 mocks can possibly show bad form
- Shows that a class is multipurpose
- Likely broke the "Single Responsibility Principle"

“

Perhaps a better rule is that we want to test a single concept in each test function. We don’t want long test functions that go testing one miscellaneous thing after another.

”

Bob Martin – Clean Code 2008

“

Every class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility

”

Source: http://en.wikipedia.org/wiki/Single_responsibility_principle

Class Cohesion

- Methods should support most if not all private encapsulated member variables
- A few can go unused in some methods, but should have a very good reason to do so.
- Any private member variables that are not entirely supports by encapsulating variables should be removed or refactored

In VCS We Trust

- Commenting out code that you do think you need anymore is bad form
- If you don't need it, **delete it**
- Trust in your version control system
- Commit green and clean code constantly, so that you can recover it
- Take time or take training to know and understand your VCS very well

Fail Fast

- Definition: A fail-fast system is designed to immediately report at its interface any failure or condition that is likely to lead to failure.
- Do not hide exceptions
- Go wrong fast and upfront or go wrong in production

What about Powermock?

Testing Legacy Code

Robert C. Martin Series
WORKING EFFECTIVELY WITH LEGACY CODE
Michael C. Feathers

```
class Service1 {
  def bar(int i) {
    foo.bar(i + 9)
  }
}
```

```
class Service2 {
  def bar(int i) {
    foo.bar(i + 3)
  }
}
```

```
class Foo {
  def bar(int i) {
    ...
  }
}
```

Legacy Code Techniques

Parameterize Method

Parameterize Method

```
class Foo {
   def bar():Unit {
       val bar = new Bar();
       bar.baz();
       bar.qux();
   }
}
```

Step 1: Identify the method that you want to replace

Parameterize Method

```
class Foo {
   def bar():Unit {
       val bar = new Bar();
       bar.baz();
       bar.qux();
   }
   def bar():Unit {
       val bar = new Bar();
       bar.baz();
       bar.qux();
   }
}
```

Step 1.1: Make a copy of the method

Parameterize Method

```
class Foo {
   def bar():Unit {
       val bar = new Bar()
       bar.baz()
       bar.qux()
   }
   def bar(b:Bar):Unit {
       val bar = new Bar()
       bar.baz()
       bar.qux()
   }
}
```

Step 2.1: Add a parameter to the method for the object whose creation you are going to replace.

Parameterize Method

```
class Foo {
   def bar():Unit {
       val bar = new Bar()
       bar.baz()
       bar.qux()
   }
   def bar(b:Bar):Unit {
       val bar =
       bar.baz()
       bar.qux()
   }
}
```

Step 2.2: Remove the object creation

Parameterize Method

```
class Foo {
   def bar():Unit {
       val bar = new Bar()
       bar.baz()
       bar.qux()
   }
   def bar(b:Bar):Unit {
       val bar = b
       bar.baz()
       bar.qux()
   }
}
```

Step 2.3: Add an assignment from the parameter to the variable that holds the object

Parameterize Method

```
class Foo {
    def bar():Unit {

    }
    def bar(b:Bar):Unit {
        val bar = b
        bar.baz()
        bar.qux()
    }
}
```

Step 3.1: Delete the body of the original method

Parameterize Method

```
class Foo {
    def bar():Unit {
        bar(new Bar())
    }
    def bar(b:Bar):Unit {
        val bar = b
        bar.baz()
        bar.qux()
    }
}
```

Step 3.1: Make a call to the parameterized method, using the object creation expression for the original object.

Parameterize Method

```
class Foo {
   def bar():Unit {
       bar(new Bar())
   }
   def bar(b:Bar):Unit {
       val bar = b
       bar.baz()
       bar.qux()
   }
}
```

Addendum: The new method is now testable

Parameterize Method

```
class Foo {
   def bar():Unit {
       bar(new Bar())
   }
   def bar(b:Bar):Unit {
       b.baz()
       b.qux()
   }
}
```

Addendum: After your test, perform refactoring

Parameterize Method to Overcome Static Calls

Parameterize Method to Overcome Static Calls

Parameterize Method

```
class Foo {
   def bar(a:Int, b:Int):Unit = {
       val resource = Server.getResource("account")
       resource.send(resource.process(a, b) * 10)
   }
}
```

Step 1: Identify the method that you want to replace

Parameterize Method

```
class Foo {
   def bar(a:Int, b:Int):Unit = {
       val resource = Server.getResource("account")
       resource.send(resource.process(a, b) * 10)
   }

   def bar(a:Int, b:Int):Unit = {
       val resource = Server.getResource("account")
       resource.send(resource.process(a, b) * 10)
   }
}
```

Step 1.1: Make a copy of the method

Parameterize Method

```
class Foo {
   def bar(a:Int, b:Int):Unit = {
       val resource = Server.getResource("account")
       resource.send(resource.process(a, b) * 10)
   }

   def bar(a:Int, b:Int, f: String => Resource):Unit = {
       val resource = Server.getResource("account")
       resource.send(resource.process(a, b) * 10)
   }
}
```

Step 2.1: Add a functional interface to the method for the static you are going to replace.

Parameterize Method

```
class Foo {
   def bar(a:Int, b:Int):Unit = {
       val resource = Server.getResource("account")
       resource.send(resource.process(a, b) * 10)
   }

   def bar(a:Int, b:Int, f: String => Resource):Unit = {
       val resource = f("account")
       resource.send(resource.process(a, b) * 10)
   }
}
```

Step 2.3: Add an assignment from the parameter to the variable that holds what the function will return, including the static's former parameters.

Parameterize Method

```
class Foo {
   def bar(a:Int, b:Int):Unit = {

   }

   def bar(a:Int, b:Int, f: String => Resource):Unit = {
       val resource = f("account")
       resource.send(resource.process(a, b) * 10)
   }
}
```

Step 3.1: Delete the body of the original method

Parameterize Method

```
class Foo {
   def bar(a:Int, b:Int):Unit = {
       bar(a, b, s => Server.getResource(s))
   }

   def bar(a:Int, b:Int, f: String => Resource):Unit = {
       val resource = f("account")
       resource.send(resource.process(a, b) * 10)
   }
}
```

Step 3.2: Make a call to the parameterized method, using the object creation expression for the original object.

Caveat

If the function is something where you don't expect much of a different behavior to change you may opt for a *"Parameterize Constructor"* with the function instead.

Parameterize Constructor

Parameterize Constructor

- **Constructors are not a part of the interface!**
- **It is false to think that a change in the constructor will change your interface.**

Parameterize Constructor

```
class Foo() {
  val bar = new Bar()
  bar.init()

  def baz() {
    bar.qux()
  }
}
```

Step 1: Identify the constructor that you want to parameterize

Parameterize Constructor

```
class Foo() {
  val bar = new Bar()
  bar.init()

  def this() = this()

  def baz() {
    bar.qux()
  }
}
```

Step 1.1: Make a copy of the primary constructor while calling the primary constructor

Parameterize Constructor

```
class Foo(b:Bar) {
  val bar = new Bar()
  bar.init()

  def this() = this()

  def baz() {
    bar.qux()
  }
}
```

Step 2.1: Add a parameter to the primary constructor for the object whose creation you are going to replace

Parameterize Constructor

```
class Foo(b:Bar) {
  val bar = b
  bar.init()

  def this() = this()

  def baz() {
    bar.qux()
  }
}
```

Step 2.2: Remove the object creation and add an assignment from the parameter to the instance variable for the object.

Parameterize Constructor

```
class Foo(b:Bar) {
  val bar = b
  bar.init()

  def this() = this(new Bar())

  def baz() {
    bar.qux()
  }
}
```

Step 3: Remove the body of the old constructor and replace it with a call to the new constructor

Parameterize Constructor

```
class Foo(bar:Bar) {
  bar.init()

  def this() = this(new Bar())

  def baz() {
    bar.qux()
  }
}
```

Step 4: Refactor

Parameterize Constructor with Lambdas to overcome static calls

Parameterize Constructor

```
class Foo (x:String) {
   def bar(int a, int b) {
     val resource = Server.getResource("account")
     resource.send(resource.process(a, b) * 10);
   }
}
```

Step 1: Identify the static method that you want to parameterize in a constructor

Parameterize Constructor

```
class Foo(x:String) {
  def this(x:String) = this(x)
  def bar(a:Int, b:Int) {
    val resource = Server.getResource("account")
    resource.send(resource.process(a, b) * 10)
  }
}
```

Step 1.1: Make a copy of the constructor

Parameterize Constructor

```
class Foo(x:String, f: String => Resource) {
  def this(x:String) = this(x)
  def bar(a:Int, b:Int) {
    val resource = Server.getResource("account")
    resource.send(resource.process(a, b) * 10)
  }
}
```

Step 2.1: Add a parameter to the original constructor for the functional interface that matches your static's signature

Parameterize Constructor

```
class Foo(x:String, f: String => Resource) {
  def this(x:String) = this(x, s => Server.getResource(s))
  def bar(a:Int, b:Int) {
    val resource = Server.getResource("account")
    resource.send(resource.process(a, b) * 10)
  }
}
```

Step 3: Call the primary constructor in the secondary constructor

Parameterize Constructor

```
class Foo(x:String, f: String => Resource) {
  def this(x:String) = this(x, s => Server.getResource(s))
  def bar(a:Int, b:Int) {
    val resource = f("account")
    resource.send(resource.process(a, b) * 10)
  }
}
```

Step 4: Replace the static with your function

Sprout Class & Method

Sprout Method

Sprout Method

```
class Foo (i:Int) {
   def bar() {
       doSomething1
       doSomething2
       doSomething3
   }
}
```

Step 1: Identify where you need to make your code change.

Sprout Method

```
class Foo (i:Int) {
   def bar() {
       // baz()
       doSomething1
       doSomething2
       doSomething3
   }
}
```

Step 2: If the change can be formulated as a single sequence of statements in one place in a method, write down a call for a new method that will do the work involved and then comment it out.

Sprout Method

```
class Foo (i:Int) {
   def bar() {
       // baz(i)
       doSomething1
       doSomething2
       doSomething3
   }
}
```

Step 3: Determine what local variables you need from the source method, and make them arguments to the call.

Sprout Method

```
class Foo (i:Int) {
   def bar() {
       // val f:Float = baz()
       doSomething1
       doSomething2
       doSomething3
   }
}
```

Step 4: Determine whether the sprouted method will need to return values to source method. If so, change the call so that its return value is assigned to a variable.

Sprout Method

```
class Foo (i:Int) {
   def bar() {
       // val f:Float = baz()
       doSomething1
       doSomething2
       doSomething3
   }
}
```

```
class FooTest extends FunSpec {
  test("calculation should be 19
       given 10") {
     val foo = new Foo();
     foo.baz(10) should be (19f)
   }
}
```

Step 5: Develop the sprout method using test-driven development

Sprout Method

```
class Foo (i:Int) {
   def bar() {
       val f:Float = baz()
       doSomething1
       doSomething2
       doSomething3
   }
}
```

```
class FooTest extends FunSpec {
  test("calculation should be 19
       given 10") {
     val foo = new Foo();
     foo.baz(10) should be (19f)
   }
}
```

Step 5: Remove the comment in the source method to enable the call.

Sprout Class

Sprout Class

```
class Foo (i:Int) {
   def bar() {
       doSomething1
       doSomething2
       doSomething3
   }
}
```

Step 1: Identify where you need to make your code change.

Sprout Class

```
class Foo (i:Int) {
   def bar() {
       doSomething1
       doSomething2
       doSomething3
   }
}
```

Step 2.1: If the change can be formulated as a single sequence of statements in one place in a method, think of a good name for a class that could do that work.

Sprout Class

```
class Foo (i:Int) {
   def bar() {
       // val other = new Other()
       // other.baz()
       doSomething1
       doSomething2
       doSomething3
   }
}
```

Step 2.2: Write code that would create an object of that class in that place, and call a method in it that will do the work that you need to do; then comment those lines out.

Sprout Class

```
class Other(i:Int)
  def baz(){}
}
```

```
class Foo (i:Int) {
   def bar() {
       // val other = new Other(i)
       // other.baz()
       doSomething1
       doSomething2
       doSomething3
   }
}
```

Step 3: Determine what local variables you need from the source method, and make them arguments to the classes' constructor.

Sprout Class

```
class Other(i:Int)
  def baz():Float = {}
}
```

```
class Foo (i:Int) {
   def bar() {
       // val other = new Other(i)
       // other.baz()
       doSomething1
       doSomething2
       doSomething3
   }
}
```

Step 4.1: Determine whether the sprouted class will need to return values to the source method. If so, provide a method in the class that will supply those values,

Sprout Class

```
class Other(i:Int)
  def baz():Float = {}
}
```

```
class Foo (i:Int) {
   def bar() {
       // val other = new Other(i)
       // val f = other.baz()
       doSomething1
       doSomething2
       doSomething3
   }
}
```

Step 4.2: Add a call in the source method to receive those values.

Sprout Class

```
class Other(i:Int)
  def baz():Float = {}
}
```

```
class OtherTest extends FunSpec{
   test("baz needs...") {
      ...
   }
}
```

Step 5: Develop the sprout class test first (TDD)

Sprout Class

```
public class Other {
  public Other(int i) {}
  public float baz(){}
}
```

```
public class Foo {
   int i;
   public void bar {
       Other other = new Other(i);
       float f = other.baz();
       doSomething1;
       doSomething2;
       doSomething3;
   }
}
```

Step 6: Remove the comments

Wrap Class & Method

Wrap Method

Wrap Method (v1)

```
class Foo {
  def deposit(amt:Int):Unit  = ...
}
```

1. Identify a method you need to change.

Wrap Method (v1)

```
class Foo {
  def deposit(amt:Int):Unit  = ...
  protected def depositTx(amt:Int):Unit  = ...
}
```

Step 2: Rename the method and then create a new method with the same name and signature as the old method.

Wrap Method (v1)

```
class Foo {
  def deposit(amt:Int):Unit  = depositTx(amt)
  protected def depositTx(amt:Int):Unit  = ...
}
```

Step 3: Place a call to the old method in the new method

Wrap Method (v1)

```
class Foo {
  def deposit(amt:Int):Unit  = {
    audit(amt, "Depositing")
    depositTx(amt)
    audit(amt, "Deposited")
  }
  protected def depositTx(amt:Int):Unit  = ...
  protected def audit(amt:Int, msg:String):Unit = ...
}
```

Step 4: Develop a method for the new feature, test first, and call it from the new method

Wrap Method (v2)

Wrap Method (v2)

```
class Foo {
  def deposit(amt:Int):Unit  = ...
}
```

1. Identify a method you need to change.

Wrap Method (v2)

```
class Foo {
  def deposit(amt:Int):Unit  = ...
  protected def audit(amt:Int, msg:String):Unit = ...
}
```

Step 2: Develop a new method for it using test-driven development.

Wrap Method (v2)

```
class Foo {
  def deposit(amt:Int):Unit  = ...
  protected def audit(amt:Int, msg:String):Unit = ...
  def auditedDeposit(amt:Int): Unit = {
    audit(amt, "Depositing")
    deposit(amt)
    audit(amt, "Deposited")
  }
}
```

Step 3: Create another method that calls the new method and the old method.

Wrap Method (v2)

```
class Foo {
  def deposit(amt:Int):Unit  = ...
  protected def audit(amt:Int, msg:String):Unit = ...
  def auditedDeposit(amt:Int): Unit = {
    audit(amt, "Depositing")
    deposit(amt)
    audit(amt, "Deposited")
  }
}
```

Addendum: You now have a choice, regular or audited

Wrap Class

Wrap Class

```
class Foo {
  def deposit(amt:Int):Unit  = ...
}
```

1. Identify a method you need to change.

Wrap Class

```scala
trait Foo {
  def deposit(amt: Int): Unit
}
```

```scala
class FooImpl extends Foo {
  def deposit(amt: Int): Unit = ...
}
```

2. Extract an trait from a class

Wrap Class

```
class Foo {
  def deposit(amt:Int):Unit  = ...
}
```

```
class FooLogger(foo:Foo) extends Foo {
  override def deposit(amt: Int): Unit = ???
}
```

```
class FooImpl extends Foo {
  def deposit(amt: Int): Unit = ...
}
```

3. Create a class that accepts the interface you are going to wrap as a constructor argument

Wrap Class

```
class FooLogger(foo:Foo) extends Foo {
  def audit(amt:Int, msg:String):Unit = ...
  override def deposit(amt: Int): Unit = ...
}
```

```
import org.scalatest.{FunSpec, Matchers}

class FooLoggerTest extends FunSpec with Matchers {
   ...
}
```

4. Create a new method using TDD that does the new work and behavior

Wrap Class

```
class FooLogger(foo:Foo) extends Foo {
  def audit(amt:Int, msg:String):Unit = ...
  override def deposit(amt: Int): Unit = {
    audit(amt, "Depositing")
    deposit(amt)
    audit(amt, "Deposited")
  }
}
```

5. Ensure that you call the old methods on the old class

Wrap Class

FooLogger
foo Foo
FooLogger(Foo)
audit(int, String) void
deposit(int) void
1
1
Foo
deposit(int) void
FooImpl
deposit(int) void

Introduce Instance Delegator

Introduce Instance Delegator

```
object Foo {
  def bar (x:Int, y:String):String = ...
}
```

Step 1: Identify a static/object method that is problematic to use in a test.

Introduce Instance Delegator

```
class Foo {
  def inBar(x:Int, y:String):String = ...
}

object Foo {
  def bar (x:Int, y:String):String = ...
}
```

Step 2: Create an instance method for the static/object method on the class, preserving signatures

Introduce Instance Delegator

```
class Foo {
  def inBar(x:Int, y:String):String = Foo.bar(x, y)
}

object Foo {
  def bar (x:Int, y:String):String = ...
}
```

Step 3: Make the instance method delegate to the static/object method.

Introduce Instance Delegator

```
class Nom {
  def chomp(i:Int, j:Int, status:String): String =
      Foo.bar(i + j, status)
}
```

Step 4: Find locations where the static/object call was made.

Introduce Instance Delegator

```
class Nom (foo:Foo){
  def this() = this(new Foo)
  def chomp(i:Int, j:Int, status:String): String =
     foo.inBar(i + j, status)
}
```

Step 5. Use ParameterizeMethod or ParameterizeConstructor to exchange it for the instance call.

Introduce Instance Delegator

```
class Nom (f: (Int, String) => String){
  def this() = this(new Foo().inBar)
  def chomp(i:Int, j:Int, status:String): String =
       f(i + j, status)
}
```

Step 5.1. Also remember using functions is always a possibility

Passing Null

Passing Null

- It’s important to get the test out as soon as possible.
- If you need to create a test and the subject under test (SUT) is hard to construct *pass null!*

Subclass and Override Method

Subclass and Override Method

```
class InterestRateCalc {
    private def callWebServiceForInterestRate():Float = ...
    def calculate:Float = {
      ...
      callWebServiceForInterestRate()
      ...
    }
}
```

Step 1: Identify the method dependencies that are complex, gather those that are giving you problems

Subclass and Override Method

```
class InterestRateCalc {
    private def callWebServiceForInterestRate():Float = ...
    def calculate:Float = {
      ...
      callWebServiceForInterestRate()
      ...
    }
}
```

Step 2: Make each method overridable

Subclass and Override Method

```
class InterestRateCalc {
    protected def callWebServiceForInterestRate():Float = ...
    def calculate:Float = {
      ...
      callWebServiceForInterestRate()
      ...
    }
}
```

Step 3: Adjust the visibility

Subclass and Override Method

```
class TestableInterestRateCalc extends InterestRateCalc {
    //Fake Interest Rate
    override def callWebServiceForInterestRate():Float = 40.0f
}
```

Step 4: Create a subclass that overrides the methods. Verify that you are able to build it in your test harness.

Sensing Variables

Sensing Variables

- A publicly accessible public variable used to determine if a certain behavior is triggered.
- It is then used in a test to verify and used to test other regressions.
- Once long methods have been broken up the tests and sensing variable can be removed.
- Great for the long “inverted pyramid” code

Sensing Variables

```
class TaxCode(...) {
   def applyTaxAndSurcharge(order:Order):Order {
     val total = order.total();
     if (order.after(LocalDate.of(“2001-01-01”))) {
         if (order.customer.getState().equals(“FL") ||
            order.customer.getState().equals("NY"))
            order.applySurcharge(Money(10))
            order.applyTax(Money(total *
              TaxWS.findTaxRate(order.customer.getState())
         else
            order.applyTax(Money(30))
         }
     } else {
       order.applyTax(0)
       order.applySurcharge(0)
     }
     order
   }
}
```

Sensing Variables

```
class TaxCode(...) {
   var nyFLTaxApplied = false
   def applyTaxAndSurcharge(order:Order):Order {
     val total = order.total();
     if (order.after(LocalDate.of("2001-01-01"))) {
         if (order.customer.getState().equals("FL") ||
            order.customer.getState().equals("NY"))
            order.applySurcharge(Money(10))
            order.applyTax(Money(total *
              TaxWS.findTaxRate(order.customer.getState()))
            nyFLTaxApplied = true
         else
            order.applyTax(Money(30))
         }
     } else {
       order.applyTax(0)
       order.applySurcharge(0)
     }
     order
   }
}
```

Sensing Variables

```
class TaxCode(...) {
   var nyFLTaxApplied = false
   def applyTaxAndSurcharge(order:Order):Order {
     val total = order.total();
     if (order.after(LocalDate.of("2001-01-01"))) {
```

```
public class TaxCodeTest extends FunSpec with Matchers {
  it("should apply a tax if from NY or FL") {
     val fakeOrder = ...
     val taxCode = new TaxCode()
     taxCode.applyTaxAndSurcharge(fakeOrder);
     taxCode.nyFLTaxApplied should be (true)
  }
}
```

```
       order.applySurcharge(0)
     }
     order
   }
}
```

Machine Learning/TMOG Unit Testing

Machine Learning Testing

- Some of the "laws" we touched upon are hard to do
- While some things are hard to manage, **keeping the principles in mind like start simple should carry us through**
- TMOG/Spark requires a local Instance to run and uses the amount of processors that are available on your machine

Cloning TMOG for ML TDD

- Clone TransmogrifAI from https://github.com/salesforce/TransmogrifAI

Lab 1: Creating Outliers

- Get into a team of 2 or more, the more brains, the better
- In TMOG, create a test, and in the test use a test generator to create sample data with some outliers
- Create a matrix of two features of your choosing and one target, e.g. height, hand-size, shirt size. Ensure that some observations are outliers, e.g. large height, and hand size or very small.
- Create functionality that would remove the outliers within TMOG

Lab 2: Refactoring a Test

- Get into a team of 2 or more, the more brains, the better
- Open `core/src/test/scala/com/salesforce/op/stages/impl/insights/RecordInsightsLOCOTest.scala`
- Go to the test at line 205
- This test is huge, given everything that you learned in this class can this be simplified? Discuss and implement

Lab 3: Finding Dates

- Get into a team of 2 or more, the more brains, the better
- Create a sample data that would mimic dates with different formats, some ideas:
  - "2013-03-01 13:00:00" (ISO-8601)
  - "2/12/1983 1:00PM"
  - "12/2/1982 16:00"
  - "3-14-98 1 AM"
- Using either a standard function format extract anything that is a date and convert into an appropriate date type that you use at work.

The naysayers

"It takes a lot time"

"In the beginning, it does, but anything worthwhile takes time. It is a great practice, and the initial time investment up front will provide faster code maintenance later."

"Mocking is kind of an inane practice"

"There should only be two mocks or less used per test. If there are more, you may have to reevaluate your design"

"Seems I am going to spend my whole life testing!"

"Test your core. Test what you believe is critical to the project. Test also what you believe will have a negative impact if given the wrong input"

"Testing in general sucks when my boss asks me to make a change"

"Don't change your class that you worked so hard on. Respect your code. There are many design patterns that you can use to change the behavior of your code. Look up Adapter Pattern, Decorator Pattern, and the Strategy Pattern"

"We weren't taught that way"

"Agreed, but we are doing more than Hello World apps now. We are also paid to maintain what we create."

Quotes from the Pros

“

As a programmer, do you deserve to feel confident?" (Can you sleep at night knowing your code works)

”

Kent Beck, Is TDD Dead? You Tube Video Series

“

The primary benefit of TDD is self testing code

”

Kent Beck, Is TDD Dead? You Tube Video Series

“

Testing extends what the compiler does, check against your domain to ensure what you are doing is accurate.

”

Daniel Hinojosa -- Yes, I am quoting myself

“

I know this sounds strident and unilateral, but given the record I don’t think surgeons should have to defend hand-washing, and I don’t think programmers should have to defend TDD.

”

Robert Martin, The Clean Coder: A Code of Conduct for Professional Programmers 2011

“

To me, legacy code is simply code without tests.

”

Michael Feathers, Working Effectively with Legacy Code 2004

Recap

Recap

- Test First.. Always
- Have an editor **and** an IDE of choice, and learn its keymap very well
- Learn your version control very well.
- Speed in TDD is key.
- Perhaps a mock can be replaced by a function!
- "Game"ify your development with testing