

# Test Driven Development

in Java with Daniel Hinojosa



*The Add*

Book

# 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

“

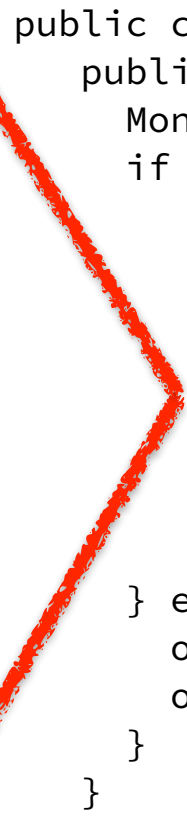
1. Write a failing test.
2. Write code to make it pass.
3. Repeat steps 1 and 2.
4. Along the way, refactor aggressively.
5. When you can't think of any more tests, you must be done.

”

# 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





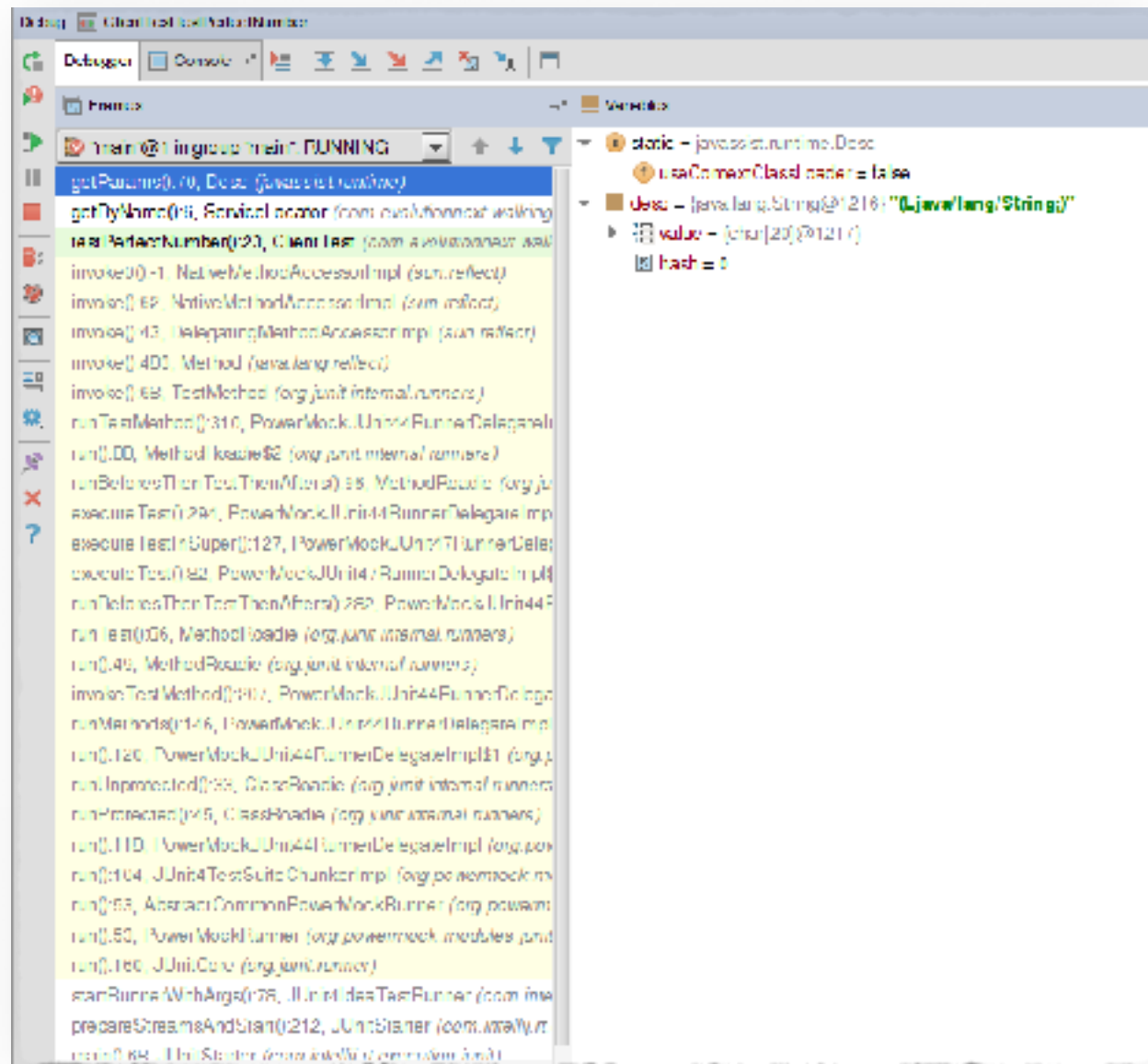
```
public class TaxCode {
    public void applyTaxAndSurcharge(Order order) {
        Money total = order.total();
        if (order.after(LocalDate.of("2001-01-01"))) {
            if (order.customer.getState().equals("FL") ||
                order.customer.getState().equals("NY"))
               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



# "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.

”

# **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

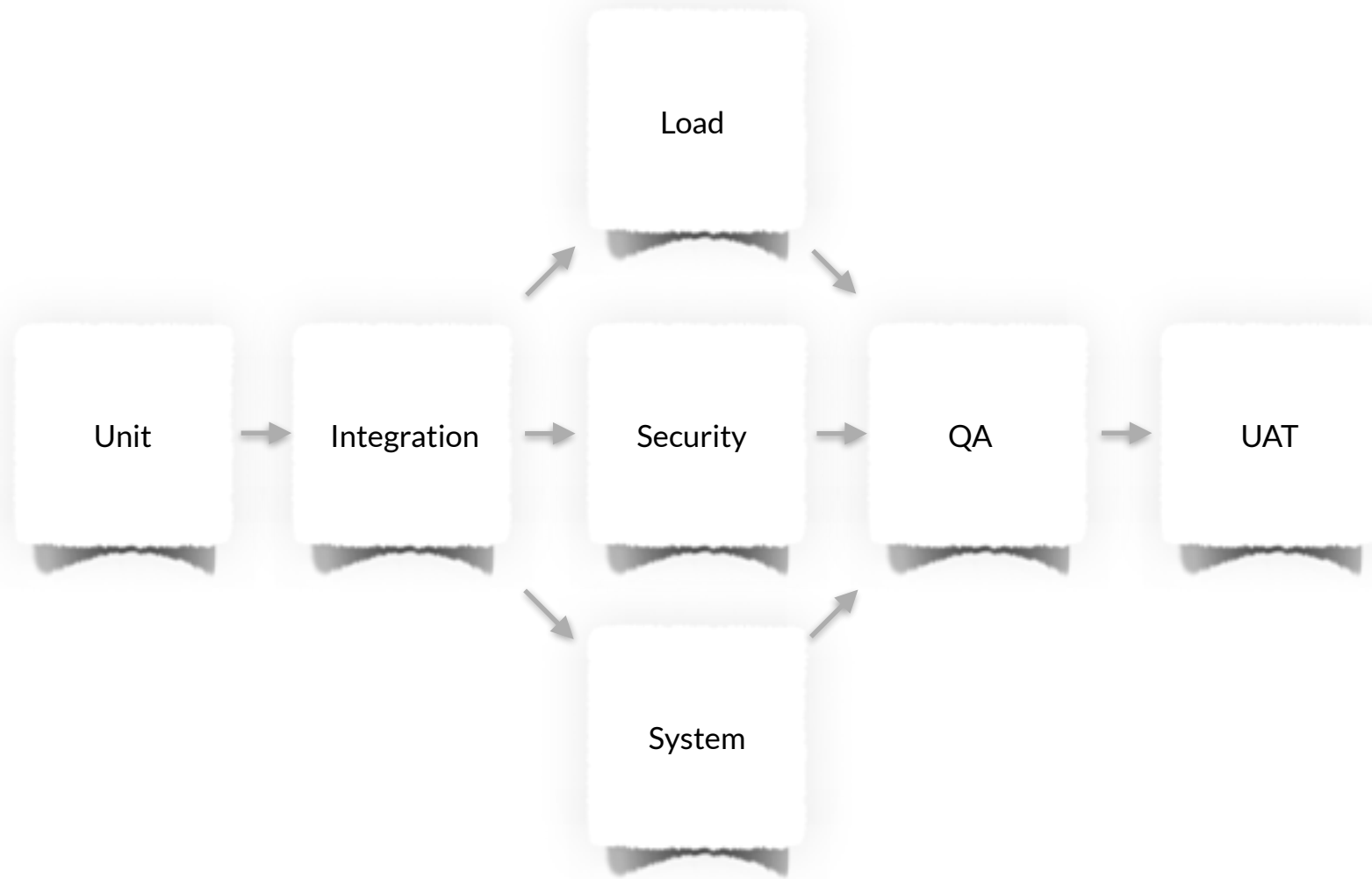
- For Green Field Projects
  - Start Immediately!
  - Prevent Technical Debt
- For Brown Field Projects
  - Adopt "Testing Thursdays or Fridays" if affordable
  - Powermock in Java if necessary\*

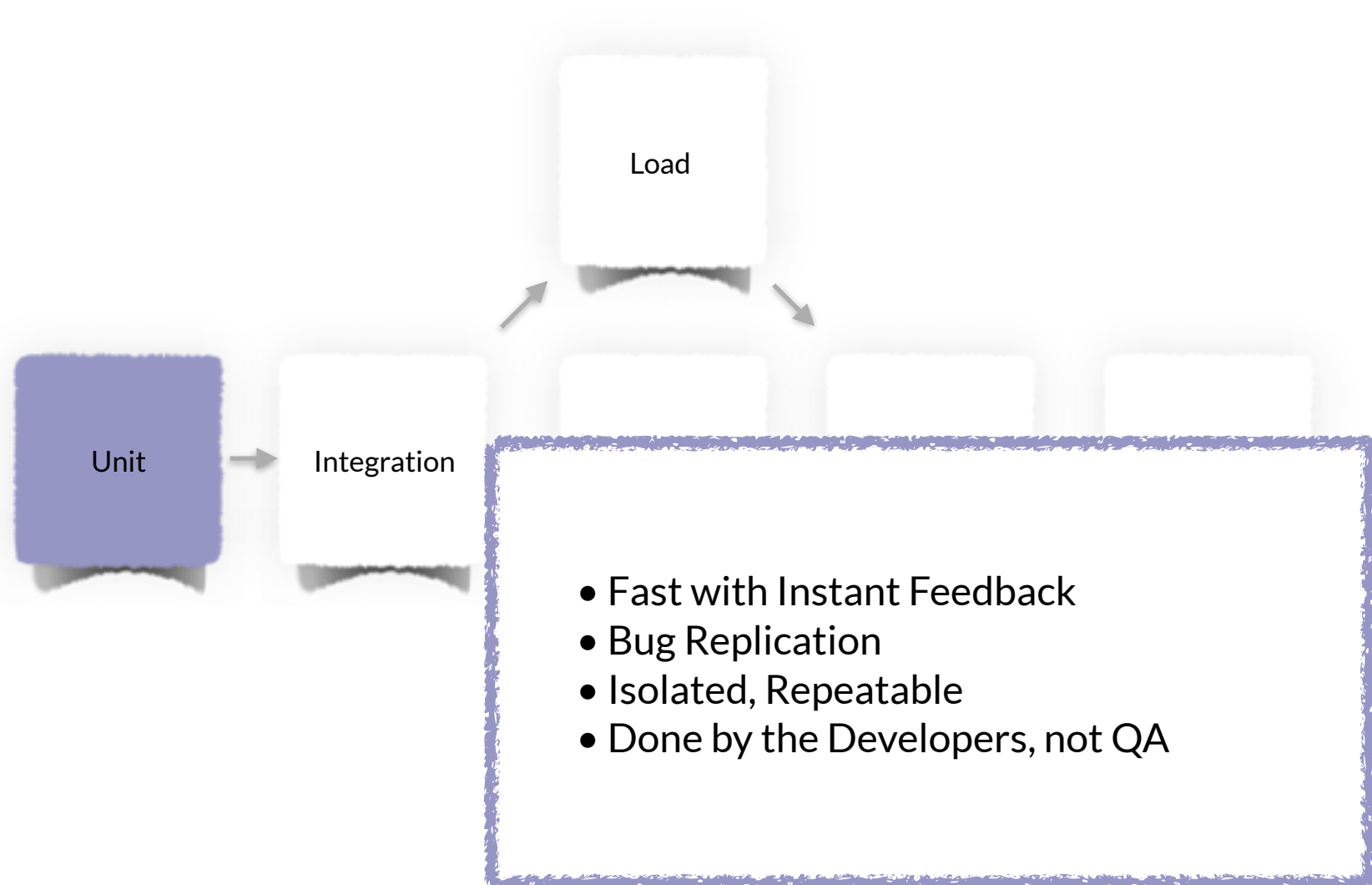
\* Powermock in Java, or any class manipulation utility is usually a sign of bad design

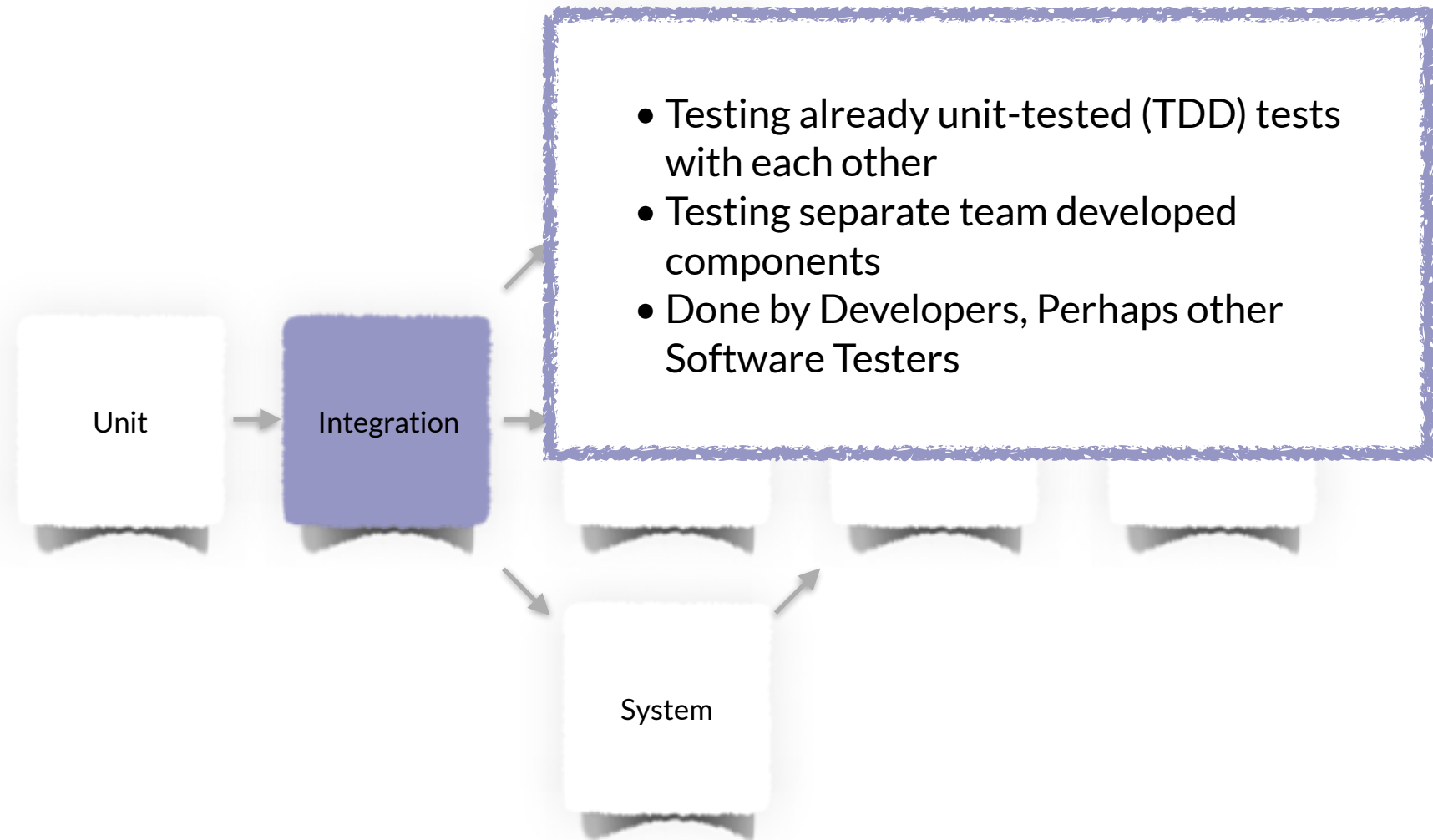
# Measuring and Monitoring TDD

- 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

- 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

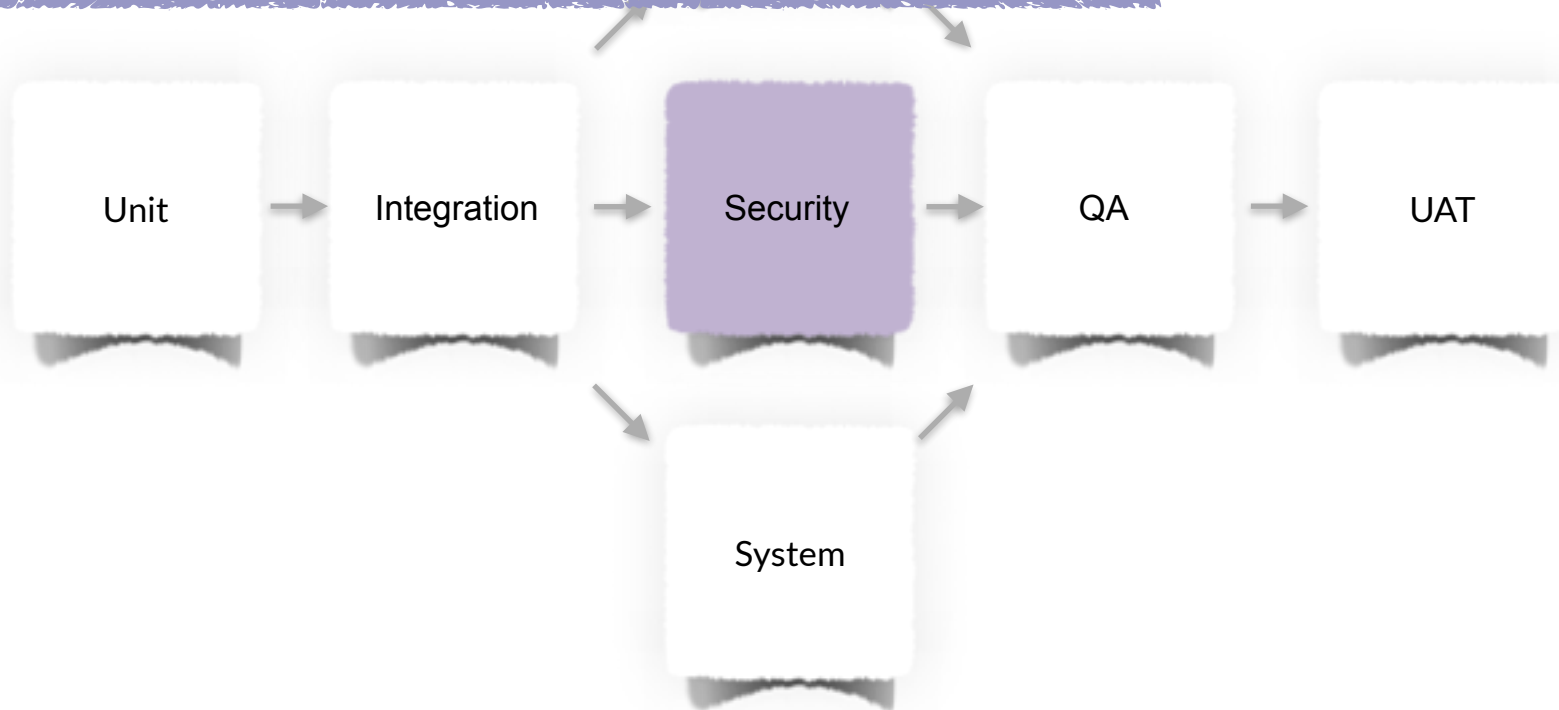
Unit

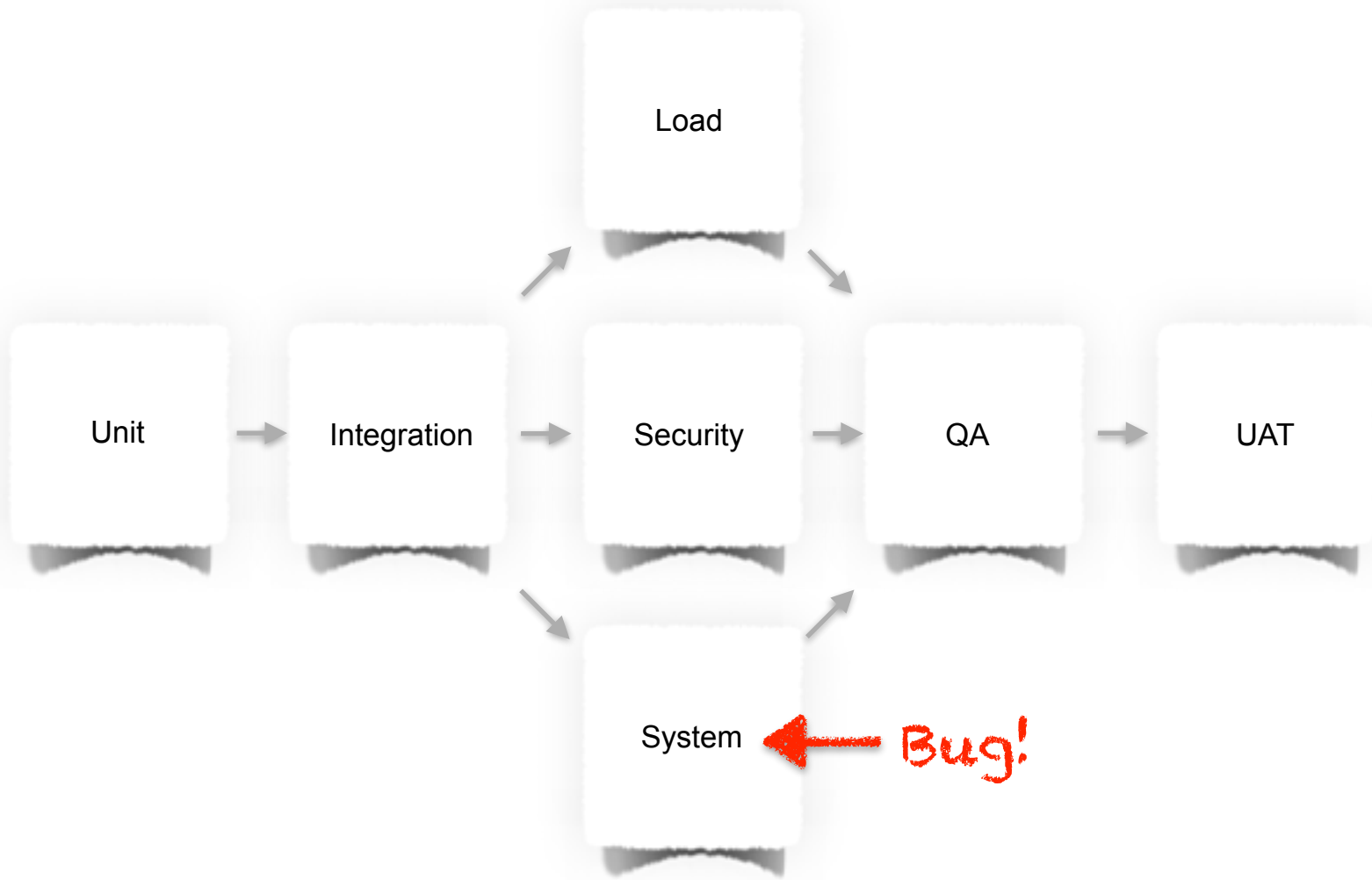
UAT

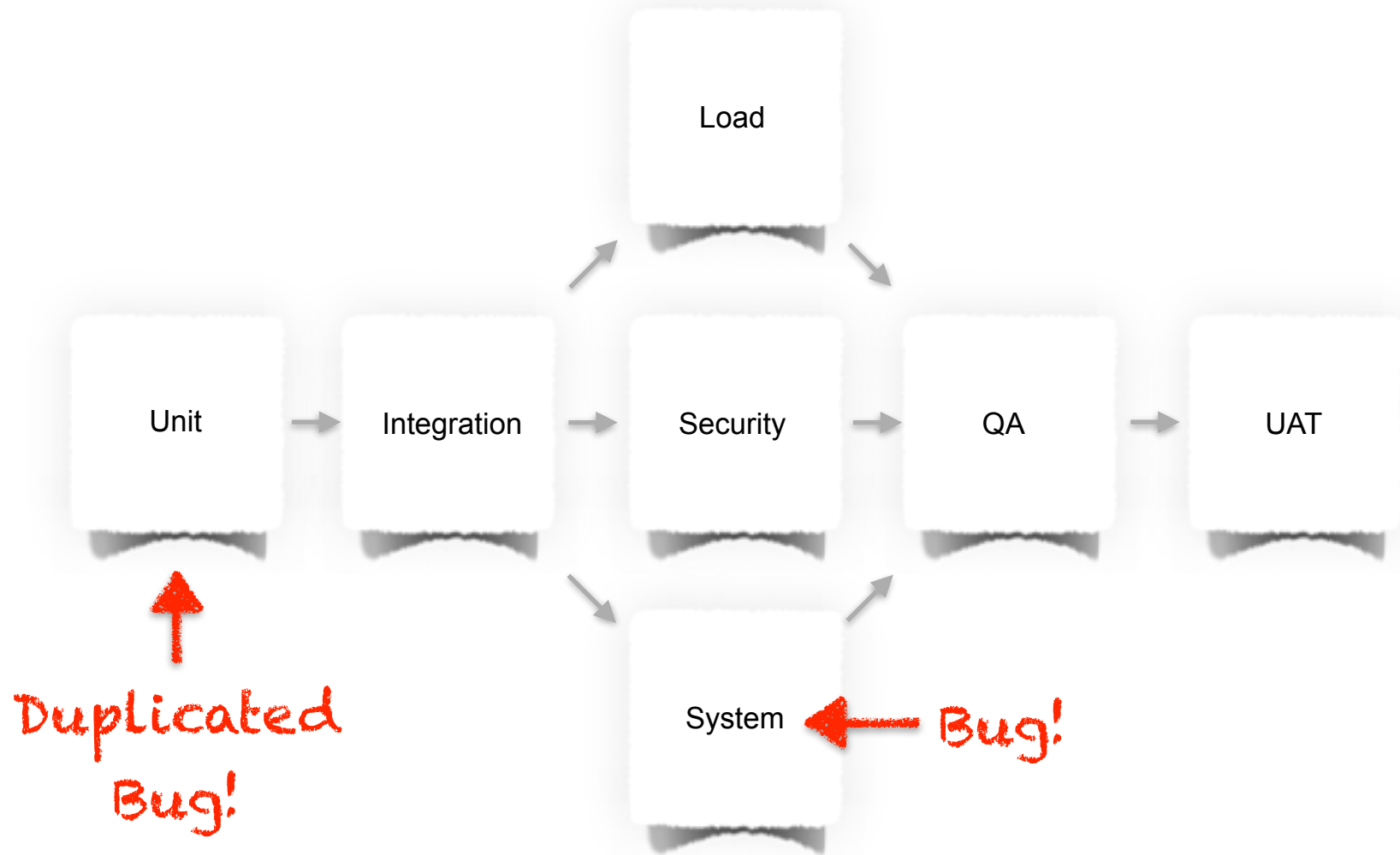
System

The diagram illustrates the relationship between different levels of testing. At the top, a large white box with a blue border contains three bullet points about system testing. Below this box, a row of seven white boxes is shown, with the first labeled 'Unit' and the last labeled 'UAT'. Below the row of boxes, a blue box labeled 'System' is positioned. Two arrows point from the 'System' box up towards the 'Unit' and 'UAT' boxes, indicating that system testing encompasses the testing of individual units and the entire 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







# 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



**What to test?**

[illegible]

Any program can be a long series of functions and statements

But we don't do that.





# Production

# Test

[illegible][illegible]

```

import static org.junit.Assert.*;

import org.junit.runner.RunWith;
import org.junit.runners.model.FrameworkField;
import org.mockito.Mockito;

public class DynamicMockerTest {

    public static int[] data() {
        return new int[]{ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 };
    }

    @RunWith(MockitoJUnitRunner.class)
    void setUpAndRunTests() throws Exception {
        MyClass tester = new MyClass();
        int n = data().length;
        assertEquals("data()",
            Mockito.mock(MyClass.class).getMyInteger(n), data());
    }

    // Class to be tested
    class MyClass {
        public int getMyInteger(int i, int j) {
            return i + j;
        }
    }
}

```

[illegible][illegible]

## 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)
- What about getters and setters? Yes\*

\* Strong Opinion

# **Testing Libraries On The JVM**

# JUnit

- Kent Beck
- Original Testing Framework on the JVM
- Most popular
- Plugins easy available for Eclipse and IntelliJ

# TestNG

- Developed by Cedric Beust
- Multiple Features for Testing
  - Groups (also now available in JUnit)
  - Providers
  - Ordered Testing
  - JUnit Integration

# Cucumber JVM

- Specification Testing Framework
- Used to primarily to test features demanded by stakeholder
- Non-Programmer Readable Style Tests
- Can be used for Unit Testing

# Testing Libraries In Node/JS



# Mocha

- Mocha.js
- Node.js and Web Based Testing
- Asynchronous Capable

# Jasmine

- Behavior Driven Development Based Framework
- Node.js and Web Based Testing
- Handy Testing Utilities

# **Code Coverage Libraries On The JVM**

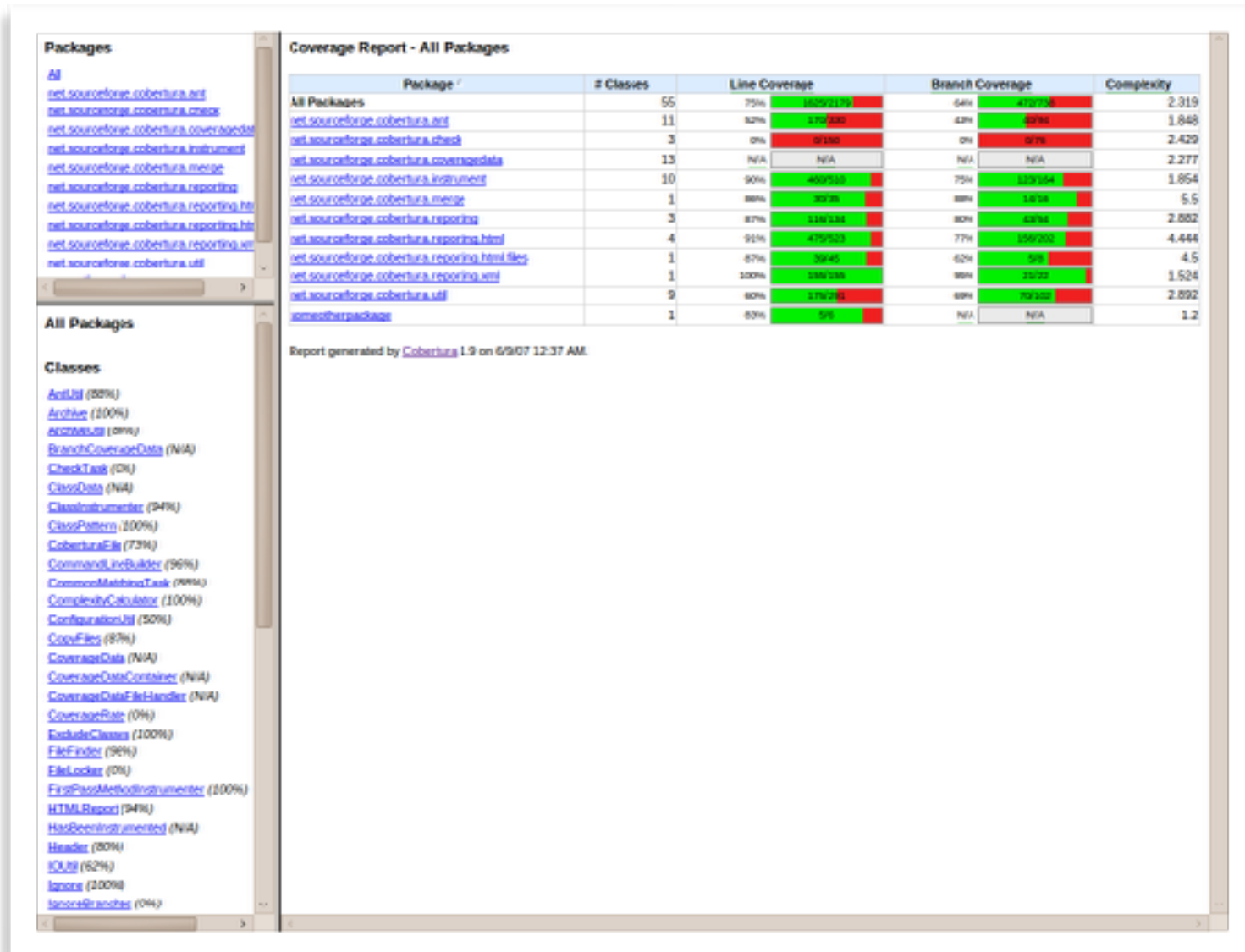
# What is Code Coverage?

- 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



# Cobertura

- Open Source Code Coverage
- Spanish/Portuguese for "Coverage"
- Easy Use

# Emma

- Open Source Coverage Tool
- Easy Use



# JaCoCo

- Another Open Source Coverage Tool
- Easy Use
- Excellent Java 8 Use

# SCCM

- Open Source
- Scala based Code Coverage for Scala programs

# **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

# **Build Tools On The JVM**

# Ant

- First Popular JVM Build Tool
- XML Based
- Developed By James Duncan Davidson
- Flexible
- Contains Multiple Tasks
- Disadvantage: Lots of XML and too much configuration!

# Maven

- An Apache Open Source project
- Development began in 2001
- Grew out of unwieldy Ant build files for other Apache projects
- Has gone through many iterations
- Current version is Maven 3.x
- Multiple Plugins

# Gradle

- Open Source
- Groovy based Build Tool
- Feature Rich and Friendly
- Contains same features as Maven
- Allow scriptability
- Multiple Plugins



# Buildr

- Open Source
- Ruby Based Build Tool
- Small following
- Flexible

# SBT

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


# Leinigen

- Clojure Based Build Tool
- Open Source


# Lab: Retrieve Project

- For Java Based Project:  
<http://www.github.com/dhinojosa/java-tdd>

Either clone or  
download project  
of your choice:

**Clone with SSH**  Use HTTPS

Use an SSH key and passphrase from account.

`git@github.com:dhinojosa/java-tdd.git` 

[Open in Desktop](#) [Download ZIP](#)

# Lab: Run A Quick Test

- For Java Based Project:
  - `cd java-tdd`
  - `mvn 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
- Full fledged Node/JS IDEs are since editors are usually used
  - WebStorm

\* NetBeans still has some Mind Share, Not Sure it is Popular

# 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 Eclipse Shortcuts

## (Linux/Windows)

- **CTRL + SHIFT + L – Toggle Keyboard Shortcut Help**
- **CTRL + E - Recent Files**
- **CTRL + M – Toggle Fullscreen**
- **CTRL + D – Delete Line**
- **SHIFT + CTRL + F – Format Code**
- **CTRL + 1 – Context Help**



# Essential Eclipse Shortcuts (Mac OSX)

- ⌘ + SHIFT(⇧) + L – **Toggle Keyboard Shortcut Help**
- ⌘ + E – **Recent Files**
- **CTRL** + M – **Toggle Fullscreen**
- ⌘ + D – **Delete Line**
- ⌘ + SHIFT(⇧) + F – **Format Code**
- ⌘ + 1 – **Context Help**

# MoreUnit for Eclipse

- Eclipse Plugin that allows you to:
  - Switch between Test and Production Easily
  - Run Tests
- CTRL + J – Switch to Test/Production Code
- CTRL + R – Run the Test





# Installing MoreUnit for Eclipse

- **Help > Eclipse Marketplace...**
- Search for **MoreUnit**
- Click Install
- Accept License Agreement
- Restart Eclipse if necessary



# 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](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**



# Editor Can't Toggle?

## Consider using split pane



The screenshot shows a code editor window titled "calculator-spec.js — test — ~/Development/danno-test-project". The editor is in split-pane mode, displaying two identical copies of the file "calculator-spec.js". The left pane shows the file content with line numbers 1 through 26. The right pane shows the same content. The code is as follows:

```
1 import chai, {
2   expect,
3 } from 'chai';
4 import add, {foo} from '../src/calculator.js';
5 import chaiAsPromised from 'chai-as-promised'
6 chai.use(chaiAsPromised)
7
8 describe('add', () => {
9   describe('addition', () => {
10     it('should add two numbers', () => {
11       expect(add(2, 3)).to.equal(5);
12     });
13   });
14   describe('negative numbers', () => {
15     it('should add consider negative numbers', () => {
16       expect(add(2, -3)).to.equal(-1);
17     });
18   });
19   describe('the hell is going on here', function() {
20     it('should smell great', function() {
21       expect(add(5, 10)).to.equal(15);
22     });
23   });
24   describe('obj.blah', () => {
25     it('should be mocked as...', () => {
26       const haz = {blah: () => 'hutt'}
```

The status bar at the bottom indicates "test/calculator-spec.js:1:1" and "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](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](http://docs.sublimetext.info/en/latest/reference/keyboard_shortcuts_osx.html)

# **Assertion Language Libraries**



# Standard JUnit

- `assertEquals(expected, actual)`
- `assertEquals(message, expected, actual)`
- `assertNotEquals(unexpected, actual)`
- `assertNotEquals(message, unexpected, actual)`
- `assertNull(object)`
- `assertNull(message, object)`
- `assertNotNull(object)`
- `assertNotNull(message, object)`

# Standard JUnit

- `assertSame(expected, actual)`
- `assertSame(message, expected, actual)`
- `assertTrue(boolean)`
- `assertTrue(message, boolean)`
- `assertArrayEquals(expectedArray, actualArray)`
- `assertArrayEquals(message, expectedArray,  
actualArray)`
- `fail()`
- `fail(message)`

# Standard TestNG

- `assertEquals(actual, expected)`
- `assertEquals(actual, expected, message)`
- `assertNotEquals(actual, unexpected)`
- `assertNotEquals(actual, unexpected, message)`
- `assertNull(object)`
- `assertNull(message, object)`

# Standard TestNG

- `assertSame(actual, expected)`
- `assertSame(actual, expected, message)`
- `assertTrue(boolean)`
- `assertTrue(message, boolean)`
- `assertEquals(actualArray, expectedArray)`
- `assertEquals(actualArray, expectedArray, message)`
- `fail()`
- `fail(message)`

# Hamcrest

- Fluent Assertion Language
- Uses Matchers to Assertions
- Allows Essentially Wildcard Matching using Matchers

# Simple Hamcrest Matchers

- `assertThat(cheese, is(equalTo(smelly)))`
- `assertThat("", isEmptyString())`
- `assertThat((String)null, isEmptyOrNullString())`
- `assertThat(cheese, is(not(equalTo(smelly))))`
- `assertThat(cheese, isNullValue())`
- `assertThat(cheese, is(notNullValue()))`

# Descriptive Testing with Hamcrest

- **describedAs**("a big decimal equal to %0",  
equalTo(myBigDecimal),  
myBigDecimal.toString())

# Comparison Hamcrest Matchers

- `assertThat(1, comparesEqualTo(1))`
- `assertThat(2, greaterThan(1))`
- `assertThat(1, greaterThanOrEqualTo(1))`
- `assertThat(1, lessThan(2))`
- `assertThat(1, lessThanOrEqualTo(1))`
- `assertThat("Foo", equalToIgnoringCase("F00"))`
- `assertThat(" my\tfoo bar ",  
equalToIgnoringWhiteSpace(" my foo bar"))`



# Instance Hamcrest Matchers

- `assertThat(cheese, is(  
                  instanceOf(Cheddar.class)))`
- `assertThat(cheese, isA(Cheddar.class))`
- `assertThat(cheese, is(sameInstance(cheddar)))`
- `assertThat(cheese, is(theInstance(cheddar)))`
- `assertThat(Integer.class,  
          is(typeCompatibleWith(Number.class)))`
- `assertThat(cheddar, is(any(Cheese.class)))`

# Array Hamcrest Matchers

- `assertThat("foo", isIn(new String[]{"bar", "foo"}))`
- `assertThat(new Integer[]{1,2,3},`
- `is(array(equalTo(1), equalTo(2), equalTo(3))))`
- `assertThat(new Integer[]{1,2,3}, hasItemInArray(equalTo(3)))`
- `assertThat(new String[] {"foo", "bar"},`
- `hasItemInArray(startsWith("ba")))`
- `assertThat(new String[]{"foo", "bar"},`
- `contains(Arrays.asList(equalTo("foo"), equalTo("bar"))))`
- `assertThat(new String[]{"foo", "bar"},`
- `containsInAnyOrder("bar", "foo"))`
- `assertThat(new String[]{"foo", "bar"},`
- `arrayWithSize(equalTo(2)))`
- `assertThat(new String[0], emptyArray())`

# Collection Hamcrest Matchers

- `assertThat(Arrays.asList("bar", "baz"), everyItem(startsWith("ba")))`
- `assertThat(Arrays.asList("foo", "bar"), hasItem("bar"))`
- `assertThat(Arrays.asList("foo", "bar"), hasItem(startsWith("ba")))`
- `assertThat(Arrays.asList("foo", "bar", "baz"), hasItems("baz", "foo"))`
- `assertThat(Arrays.asList("foo", "bar", "baz"), hasItems(endsWith("z"), endsWith("o")))`
- `assertThat(Arrays.asList("foo", "bar"), hasSize(equalTo(2)))`
- `assertThat(new ArrayList<String>(), is(empty()))`

# Map Hamcrest Matchers

- `assertThat(myMap, hasKey(equalTo("bar")))`
- `assertThat(myMap, hasKey("bar"))`
- `assertThat(myMap, hasValue("foo"))`
- `assertThat(myMap, hasValue(equalTo("foo")))`

# Floating Point Hamcrest Matchers

- `assertThat(1.03, is(closeTo(1.0, 0.03)))`
- `assertThat(new BigDecimal("1.03"),  
is(closeTo(new BigDecimal("1.0"),  
new BigDecimal("0.03"))))`

# String Hamcrest Matchers

- `assertThat("myStringOfNote", containsString("ring"))`
- `assertThat("myStringOfNote", startsWith("my"))`
- `assertThat("myStringOfNote", endsWith("Note"))`

# Property Hamcrest Matchers

- `assertThat(myBean, hasProperty("foo"))`
- `assertThat(myBean, hasProperty("foo",  
equalTo("bar"))`
- `assertThat(myBean,  
samePropertyValuesAs(myExpectedBean))`

# Wildcard Matchers

- `assertThat("myValue", allOf(startsWith("my"), containsString("Val")))`
- `assertThat("myValue", anyOf(startsWith("foo"), containsString("Val")))`
- `assertThat("fab", both(containsString("a"))  
                                  .and(containsString("b")))`
- `assertThat("fan",  
              either(containsString("a"))  
              .and(containsString("b")))`
- `assertThat(cheese, is(anything()))`



# Fest Assert

- Fluent Assertions based on type.
- Filtering based on type provided
- Very little need to look up matchers since IDE will provide guidance
- Supplanted by AssertJ

# AssertJ

- Update to Fest Assert
- Contains assertions for Guava, Joda-Time, Swing
- `import static org.assertj.core.api.Assertions.*;`

# cyber-dojō.org

the place to practice programming



**setup a new practice session**

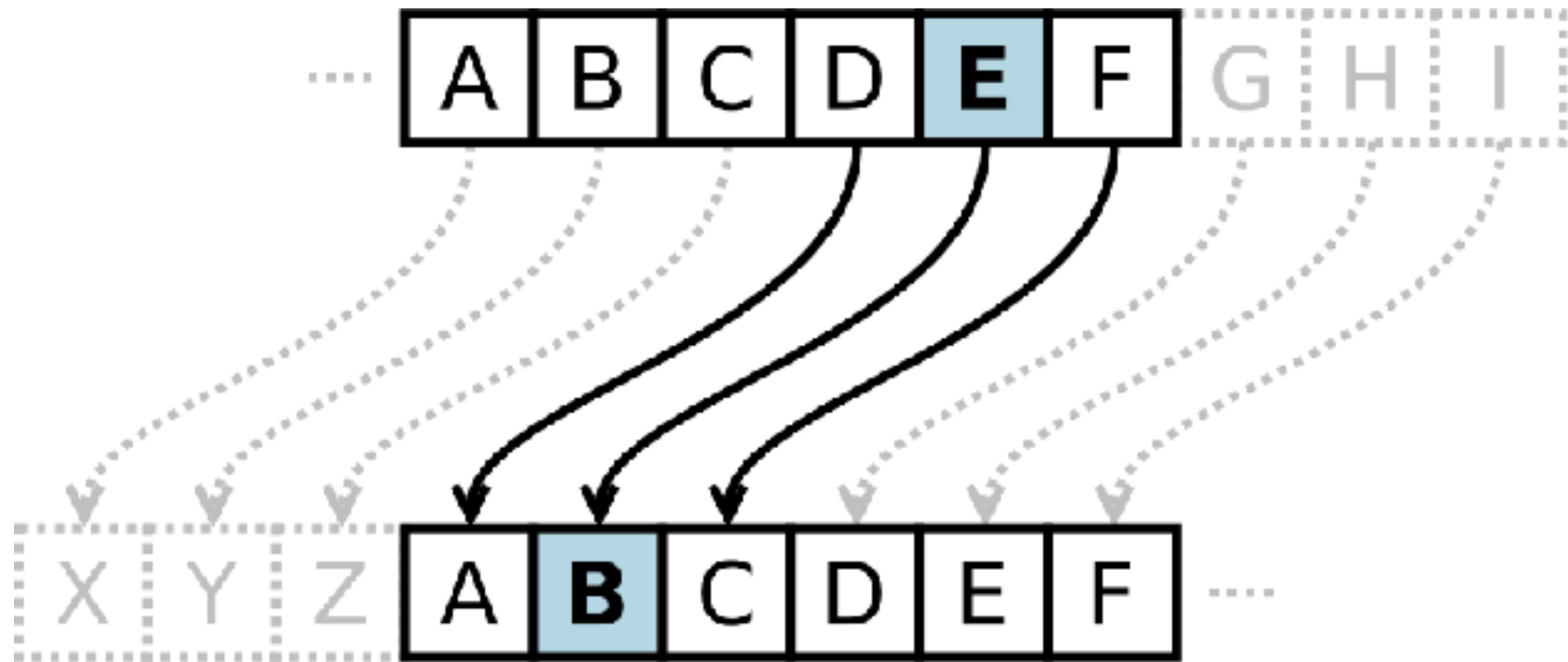
**enter a practice session**

**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”  $\Rightarrow$  +5  $\Rightarrow$  “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>

# **Day 2: Isolated Testing**



# **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

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



# No evidence of a static dependency

```
public void method2() {  
    Resource resource =  
        ServerInstance.find("/accounts/resource");  
    resource.addDeposit(3000.00);  
}
```



# Static Dependency Hard to Control

```
public Resource method3() {  
    Resource resource =  
        ServerInstance.find("/accounts/resource");  
    resource.addDeposit(3000.00);  
    return resource;  
}
```



# Full Evidence of a Dependency

```
public Resource method4(Resource resource)
{
    resource.addDeposit(3000.00);
    return resource;
}
```



# Full Evidence of a Dependency

```
public Employee method4() {  
    return new Employee("Sean",  
"Connery"); //OK  
}
```





# Full Evidence of a Dependency

```
public List<Employee> method5() {  
    return Arrays.asList(  
        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).

”

# Dummies

“

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

”

# 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'.

”

# Mocks

“

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

”

# Java 8 Functions

- Lambda Expressions are available now on Java 8
- Potential to minimize the use of mocks & stubs, save time

# Java 8 Functions

“

A functional interface is any interface that contains only one abstract method. (A functional interface may contain one or more default methods or static methods.) Because a functional interface contains only one abstract method, you can omit the name of that method when you implement it.

”

Functional Interface Definition - <https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>

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

# JMock

- Mocking Framework
- No current release since 2012

# Mockito

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

# **Instructor Led: Dice! Dice! Baby!**



# **Group Lab: Finding overdue library books.**

# **Best Practices and Advice**

# Code Reviewer Guide

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



**Miško Hevery**  
The Testability Explorer Blog





# 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.

”

“

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](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 supported 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

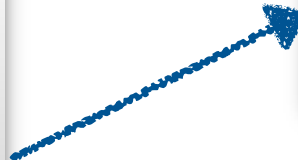
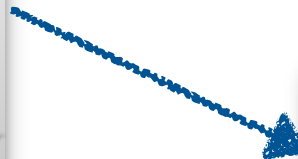
```
class Service1 {  
    def bar(int i) {  
        foo.bar(i + 9)  
    }  
}
```



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



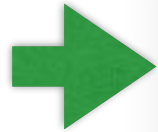
```
class Foo {  
    def bar(String s) {  
        ...  
    }  
}
```



# **Legacy Code Techniques**

# **Parameterize Method**

# Parameterize Method



```
public class Foo {  
    public void bar() {  
        Bar bar = new Bar();  
        bar.baz();  
        bar.qux();  
    }  
}
```

Step 1: Identify the method that you want to replace

# Parameterize Method

```
public class Foo {  
    public void bar() {  
        Bar bar = new Bar();  
        bar.baz();  
        bar.qux();  
    }  
  
    public void bar() {  
        Bar bar = new Bar();  
        bar.baz();  
        bar.qux();  
    }  
}
```

Step 1.1: Make a copy of the method

# Parameterize Method

```
public class Foo {  
    public void bar() {  
        Bar bar = new Bar();  
        bar.baz();  
        bar.qux();  
    }  
  
    public void bar(Bar b) {  
        Bar 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

```
public class Foo {  
    public void bar() {  
        Bar bar = new Bar();  
        bar.baz();  
        bar.qux();  
    }  
  
    public void bar(Bar b) {  
        Bar bar =  
        bar.baz();  
        bar.qux();  
    }  
}
```

Step 2.2: Remove the object creation

# Parameterize Method

```
public class Foo {  
    public void bar() {  
        Bar bar = new Bar();  
        bar.baz();  
        bar.qux();  
    }  
  
    public void bar(Bar b) {  
        Bar bar = b;  
        bar.baz();  
        bar.qux();  
    }  
}
```

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

# Parameterize Method

```
public class Foo {  
    public void bar() {  
  
    }  
  
    public void bar(Bar b) {  
        Bar bar = b;  
        bar.baz();  
        bar.qux();  
    }  
}
```

Step 3.1: Delete the body of the original method

# Parameterize Method

```
public class Foo {  
    public void bar() {  
        bar(new Bar());  
    }  
  
    public void bar(Bar b) {  
        Bar 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

```
public class Foo {  
    public void bar() {  
        bar(new Bar());  
    }  
  
    public void bar(Bar b) {  
        Bar bar = b;  
        bar.baz();  
        bar.qux();  
    }  
}
```

Addendum: The new method is now testable

# Parameterize Method

```
public class Foo {  
    public void bar() {  
        bar(new Bar());  
    }  
  
    public void bar(Bar bar) {  
        bar.baz();  
        bar.qux();  
    }  
}
```

Addendum: After your test, perform refactoring

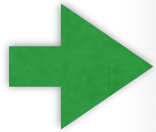
# **Parameterize Method to Overcome Static Calls**

# **Parameterize Method to Overcome Static Calls**





# Parameterize Method



```
public class Foo {  
    public void bar(int a, int b) {  
        Resource r = Server.getResource("account")  
        r.send(r.process(a, b) * 10);  
    }  
}
```

Step 1: Identify the method that you want to replace

# Parameterize Method

```
public class Foo {  
    public void bar(int a, int b) {  
        Resource r = Server.getResource("account")  
        r.send(r.process(a, b) * 10);  
    }  
  
    public void bar(int a, int b) {  
        Resource r = Server.getResource("account")  
        r.send(r.process(a, b) * 10);  
    }  
}
```

Step 1.1: Make a copy of the method

# Parameterize Method

```
public class Foo {  
    public void bar(int a, int b) {  
        Resource r = Server.getResource("account")  
        r.send(r.process(a, b) * 10);  
    }  
  
    public void bar(int a, int b,  
                    Function<String, Resource> f){  
        Resource r = Server.getResource("account")  
        r.send(r.process(a, b) * 10);  
    }  
}
```

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

# Parameterize Method

```
public class Foo {  
    public void bar(int a, int b) {  
        Resource r = Server.getResource("account")  
        r.send(r.process(a, b) * 10);  
    }  
  
    public void bar(int a, int b,  
                    Function<String, Resource> f){  
        Resource r = f("account")  
        r.send(r.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

```
public class Foo {  
    public void bar(int a, int b) {  
  
    }  
  
    public void bar(int a, int b,  
                    Function<String, Resource> f){  
        Resource r = f("account")  
        r.send(r.process(a, b) * 10);  
    }  
}
```

Step 3.1: Delete the body of the original method

# Parameterize Method

```
public class Foo {  
    public void bar(int a, int b) {  
        bar(a, b,  
            s -> Server.getResource(s)  
        )  
    }  
  
    public void bar(int a, int b,  
                    Function<String, Resource> f){  
        Resource r = f("account")  
        r.send(r.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.

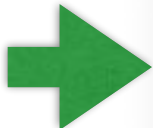
# **Parameterize Constructor**



# 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



```
public class Foo {  
    public Foo() {  
        Bar bar = new Bar();  
        bar.init();  
    }  
  
    public baz() {  
        bar.qux();  
    }  
}
```

Step 1: Identify the constructor that you want to parameterize

# Parameterize Constructor

```
public class Foo {  
    public Foo() {  
        Bar bar = new Bar();  
        bar.init();  
    }  
}
```

```
public Foo() {  
    Bar bar = new Bar();  
    bar.init();  
}
```

```
    public baz() {  
        bar.qux();  
    }  
}
```

Step 1.1: Make a copy of it.

# Parameterize Constructor

```
public class Foo {  
    public Foo() {  
        Bar bar = new Bar();  
        bar.init();  
    }  
  
    public Foo(Bar b) {  
        Bar bar = new Bar();  
        bar.init();  
    }  
  
    public baz() {  
        bar.qux();  
    }  
}
```

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

# Parameterize Constructor

```
public class Foo {  
    public Foo() {  
        Bar bar = new Bar();  
        bar.init();  
    }  
  
    public Foo(Bar b) {  
        Bar bar = b  
        bar.init();  
    }  
  
    public baz() {  
        bar.qux();  
    }  
}
```

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

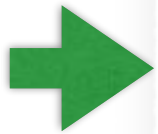
# Parameterize Constructor

```
public class Foo {  
    public Foo() {  
        this(new Bar());  
    }  
  
    public Foo(Bar b) {  
        Bar bar = b  
        bar.init();  
    }  
  
    public baz() {  
        bar.qux();  
    }  
}
```

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

**Parameterize Constructor  
with Lambdas to overcome  
static calls**

# Parameterize Constructor



```
public class Foo {  
    public void bar(int a, int b) {  
        Resource r = Server.getResource("account")  
        r.send(r.process(a, b) * 10);  
    }  
}
```

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



# Parameterize Constructor

```
public class Foo {  
    public Foo() {}  
    public Foo() {}  
    public void bar(int a, int b) {  
        Resource r = Server.getResource("account")  
        r.send(r.process(a, b) * 10);  
    }  
}
```

Step 1.1: Make a copy of the constructor  
(in this case there was no explicit constructor, so I  
made two to preserve signatures)

# Parameterize Constructor

```
public class Foo {  
    public Foo() {}  
    public Foo(Function<String, Resource> f) {}  
    public void bar(int a, int b) {  
        Resource r = Server.getResource("account")  
        r.send(r.process(a, b) * 10);  
    }  
}
```

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

# Parameterize Constructor

```
public class Foo {  
    private Function<String, Resource> f;  
    public Foo() {}  
    public Foo(Function<String, Resource> f) {}  
    public void bar(int a, int b) {  
        Resource r = Server.getResource("account")  
        r.send(r.process(a, b) * 10);  
    }  
}
```

Step 2.2: Create a member variable to store the function

# Parameterize Constructor

```
public class Foo {  
    private Function<String, Resource> f;  
    public Foo() {}  
    public Foo(Function<String, Resource> f) {  
        this.f = f;  
    }  
    public void bar(int a, int b) {  
        Resource r = Server.getResource("account")  
        r.send(r.process(a, b) * 10);  
    }  
}
```

Step 2.2: Add an assignment from the parameter to the member variable in the copied constructor

# Parameterize Constructor

```
public class Foo {  
    private Function<String, Resource> f;  
    public Foo() {  
        this(s -> Server.getResource(s))  
    }  
    public Foo(Function<String, Resource> f) {  
        this.f = f;  
    }  
    public void bar(int a, int b) {  
        Resource r = Server.getResource("account")  
        r.send(r.process(a, b) * 10);  
    }  
}
```

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

# Parameterize Constructor

```
public class Foo {  
    private Function<String, Resource> f;  
    public Foo() {  
        this(s -> Server.getResource(s))  
    }  
    public Foo(Function<String, Resource> f) {  
        this.f = f;  
    }  
    public void bar(int a, int b) {  
        Resource r = f("account")  
        r.send(r.process(a, b) * 10);  
    }  
}
```

Step 4: Replace the static with your function

# Parameterize Constructor

```
public class Foo {  
    private Function<String, Resource> f;  
    public Foo() {  
        this(Server::getResource)  
    }  
    public Foo(Function<String, Resource> f) {  
        this.f = f;  
    }  
    public void bar(int a, int b) {  
        Resource r = f("account")  
        r.send(r.process(a, b) * 10);  
    }  
}
```

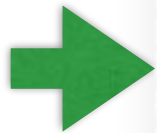
Addendum: test your method with TDD, refactor

# **Sprout Class & Method**



# **Sprout Method**

# Sprout Method



```
public class Foo {  
    int i;  
    public void bar {  
        doSomething1;  
        doSomething2;  
        doSomething3;  
    }  
}
```

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

# Sprout Method

```
public class Foo {  
    int i;  
    public void 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

```
public class Foo {  
    int i;  
    public void 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

```
public class Foo {  
    int i;  
    public void bar {  
        //float f = baz(i)  
        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

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

```
public class FooTest {  
    @Test  
    public void testBaz {  
        Foo foo = new Foo();  
        assertEquals(  
            19.0f, foo.baz(10))  
    }  
}
```

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

# Sprout Method

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

```
public class FooTest {  
    @Test  
    public void testBaz {  
        Foo foo = new Foo();  
        assertEquals(  
            19.0f, foo.baz(10))  
    }  
}
```

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

# **Sprout Class**



# Sprout Class



```
public class Foo {  
    int i;  
    public void bar() {  
        doSomething1;  
        doSomething2;  
        doSomething3;  
    }  
}
```

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

# Sprout Class

```
public class Foo {  
    int i;  
    public void 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

```
public class Foo {  
    int i;  
    public void bar() {  
        // Other 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

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

```
public class Foo {  
    int i;  
    public void bar() {  
        // Other 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

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

```
public class Foo {  
    int i;  
    public void bar() {  
        // Other 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

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

# Sprout Class

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

```
public class OtherTest {  
    @Test  
    public void testBaz() {  
        ...  
        assertEquals(...)   
    }  
}
```

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;  
    }  
}
```

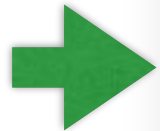
Step 6: Remove the comments



# **Wrap Class & Method**

# **Wrap Method**

# Wrap Method (v1)



```
public class Foo {  
    public void deposit(int amt) {...}  
}
```

1. Identify a method you need to change.

# Wrap Method (v1)

```
public class Foo {  
    protected void depositTx(int amt) {...}  
    public void deposit(int amt) {}  
}
```

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

# Wrap Method (v1)

```
public class Foo {  
    protected void depositTx(int amt) {  
        ...  
    }  
    public void deposit(int amt) {  
        depositTx(amt);  
    }  
}
```

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

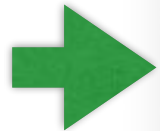
# Wrap Method (v1)

```
public class Foo {  
    public void depositTx(int amt) {...}  
    public void deposit(int amt) {  
        audit(amt, "Depositing");  
        depositTx(amt);  
        audit(amt, "Deposited");  
    }  
    protected void audit(int amt, String msg) {...}  
}
```

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

# **Wrap Method (v2)**

# Wrap Method (v2)



```
public class Foo {  
    public void deposit(int amt) {...}  
}
```

1. Identify a method you need to change.



# Wrap Method (v2)

```
public class Foo {  
    public void deposit(int amt) {...}  
    protected void audit(int amt, String msg) {...}  
}
```

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

# Wrap Method (v2)

```
public class Foo {  
    public void deposit(int amt) {...}  
    protected void audit(int amt, String msg) {...}  
    public void auditedDeposit(int amt) {  
        audit(amt, "Depositing");  
        deposit(amt);  
        audit(amt, "Deposited");  
    }  
}
```

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

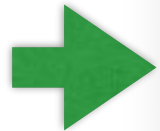
# Wrap Method (v2)

```
public class Foo {  
    public void deposit(int amt) {...}  
    protected void audit(int amt, String msg) {...}  
    public void auditedDeposit(int amt) {  
        audit(amt, "Depositing");  
        deposit(amt);  
        audit(amt, "Deposited");  
    }  
}
```

Addendum: You now have a choice, regular or audited

# **Wrap Class**

# Wrap Class



```
public class Foo {  
    public void deposit(int amt) {...}  
}
```

1. Identify a method you need to change.

# Wrap Class

```
public interface Foo {  
    public void deposit(int amt);  
}
```

```
public class FooImpl implements Foo {  
    public void deposit(int amt) {...}  
}
```

2. Extract an interface from a class

# Wrap Class

```
public interface Foo {  
    public void deposit(int amt);  
  
    public class FooLogger implements Foo {  
        public FooLogger(Foo foo) {...}  
        public void deposit(int amt) {}  
    }  
  
    public void deposit(int amt) {...}  
}
```

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

# Wrap Class

```
public class FooLogger implements Foo {  
    public FooLogger(Foo foo) {...}  
    protected void audit(int amt, String msg) {...}  
    public void deposit(int amt) {}  
}
```

```
public class FooLoggerTest {  
    @Test  
    public void testAudit() {...}  
}
```

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

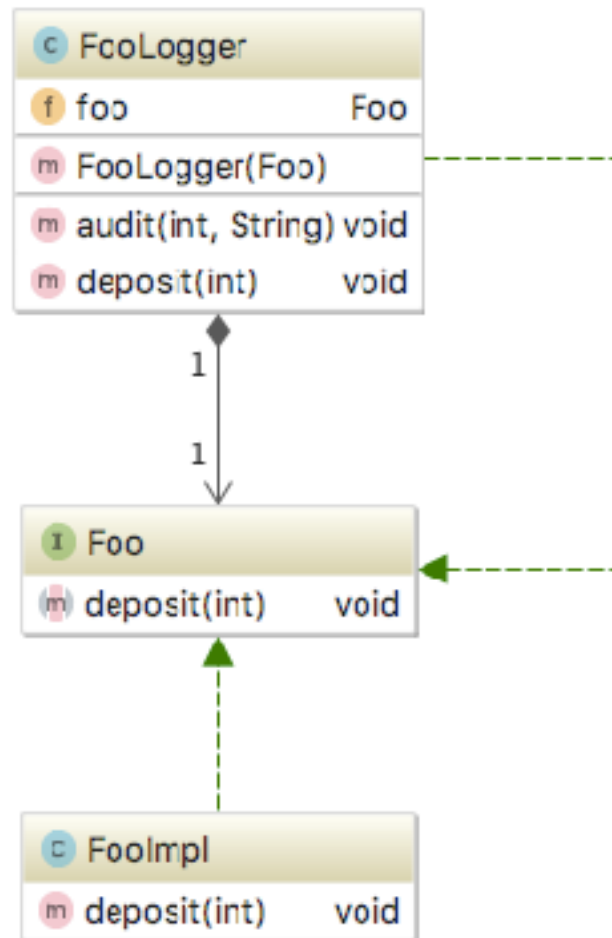


# Wrap Class

```
public class FooLogger implements Foo {  
    private Foo foo;  
    public FooLogger(Foo foo) {this.foo = foo;}  
    protected void audit(int amt, String msg) {...}  
    public void deposit(int amt) {  
        audit(amt, "Depositing");  
        foo.deposit(amt);  
        audit(amt, "Deposited");  
    }  
}
```

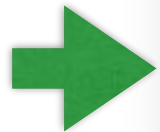
5. Ensure that you call the old methods on the old class

# Wrap Class



# **Introduce Instance Delegator**

# Introduce Instance Delegator



```
public class Foo {  
    public static bar(int x, String y) {...}  
}
```

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

# Introduce Instance Delegator

```
public class Foo {  
    public static void bar(int x, String y) {..}  
    public void inBar(int x, String y) {  
  
    }  
}
```

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

# Introduce Instance Delegator

```
public class Foo {  
    public static void bar(int x, String y) {..}  
    public void inBar(int x, String y) {  
        bar(x, y);  
    }  
}
```

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

# Introduce Instance Delegator

```
public class Nom {  
    public void chomp(int i, int j, String status) {  
        Foo.bar(i + j, status);  
    }  
}
```

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

# Introduce Instance Delegator

```
public class Nom {  
    public Nom() {}  
    public Nom(Foo foo) {this.foo = foo;}  
    public void chomp(int i, int j, String status) {  
        foo.inBar(i + j, status);  
    }  
}
```

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



# 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

```
public class InterestRateCalc {  
    private final float callWS() {..}  
    public float calculate {..}  
}
```

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

# Subclass and Override Method

```
public class InterestRateCalc {  
    private float callWS() {...}  
    public float calculate {...}  
}
```

Step 2: Make each method overridable

# Subclass and Override Method

```
public class InterestRateCalc {  
    protected float callWS() {..}  
    public float calculate {..}  
}
```

Step 3: Adjust the visibility

# Subclass and Override Method

```
public class TestableInterestRateCalc extends InterestRateCalc {  
    protected float callWS() {return 100.0;} //fake  
    public float calculate {..} //testable  
}
```

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

# Subclass and Override Method

```
public class InterestRateCalcTest
    extends InterestRateCalc {
    @Override
    protected float callWS() {return 100.0;} //fake

    public void testCalculate() {
        assertEquals(200, this.calculate());
    }
}
```

Addendum: You can also extend in a test class!



# **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

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

# Sensing Variables

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

# Sensing Variables

```
public class TaxCode {  
    public boolean nyFLTaxApplied = false  
    public void applyTaxAndSurcharge(Order order) {  
        Money total = order.total();
```

```
public class TaxCodeTest {  
    public void testNYFLTaxApplied() {  
        Order fakeOrder = ...;  
        TaxCode taxCode = new TaxCode();  
        taxCode.applyTaxAndSurcharge(fakeOrder);  
        assertTrue(taxCode.nyFLTaxApplied());  
    }  
}
```

```
    } else {  
        order.applyTax(0)  
        order.applySurcharge(0)  
    }  
}  
}
```

# **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