

# Testing Legacy Code and Avoid Mocking

Daniel Hinojosa

@dhinojosa

<http://www.evolutionnext.com>



# About me

Independent Consultant,  
Programmer. Trainer, Author

Just loves doing this kind of  
stuff.

(Sometimes comes with  
beard, sometimes not)

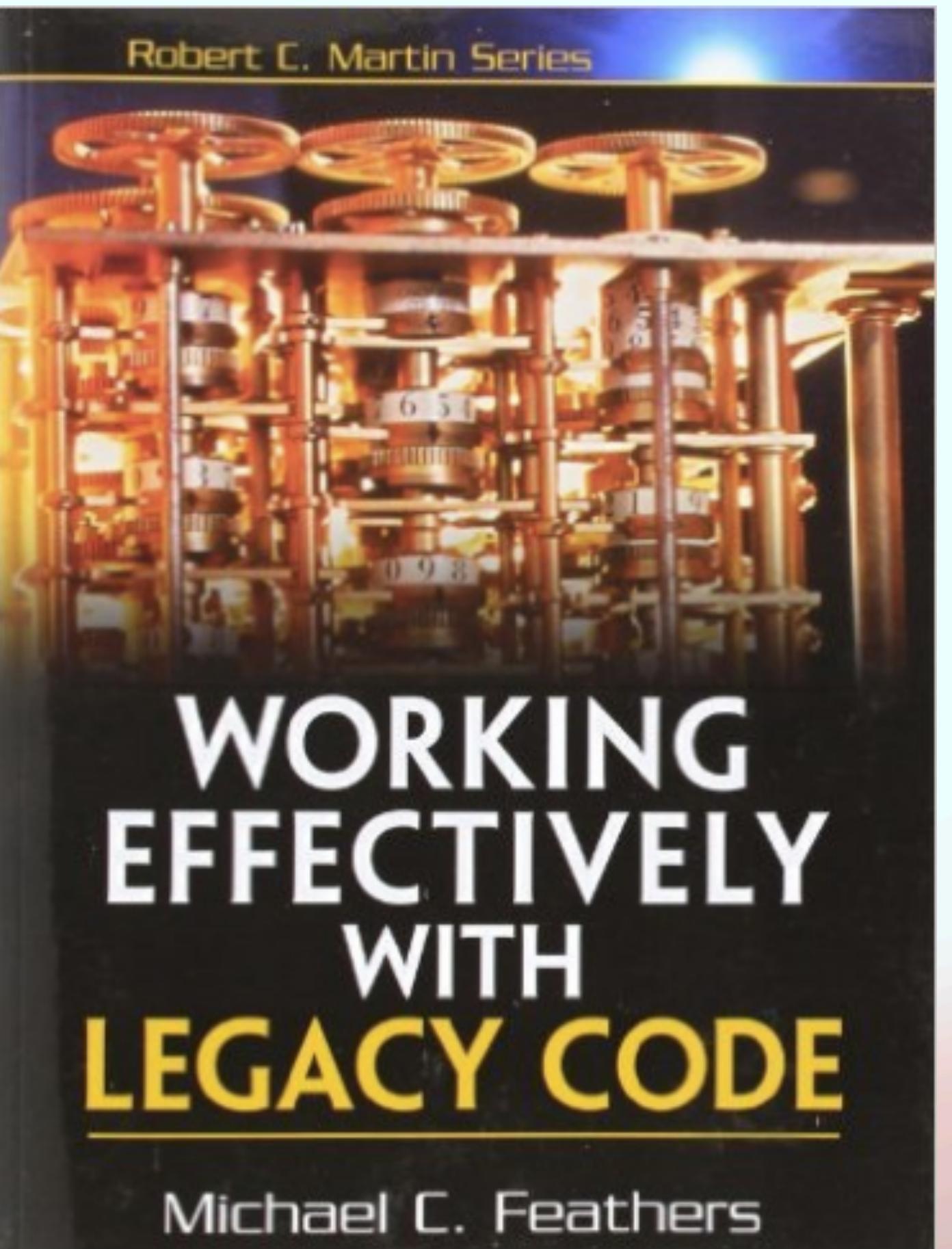


# Agenda

Discuss some of the techniques to get tests around  
some of this undesirable code all while preserving  
signatures.







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





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



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



```
class SBank {  
    def bar(String i){  
        ...  
    }  
}
```

*Before we Begin...*

# Test Driven Development

We will be using Test Driven Development, although this time....after the fact.

# TDD, a gentle reminder

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.

*What do we want to  
work towards?*

# Single Responsibility Principle (SRP)

Every class should have a single responsibility: It should have a single purpose in the system, and there should be only one reason to change it.

# Liskov Substitution Principle

Objects of subclasses should be substitutable for objects of their superclasses throughout our code.

or

“Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.”

# Command/Query Separation

“A method should be a **command** or a **query**, but not both”

# What is a Command?

A **command** is a method that can modify the state of the object but that doesn't return a value.

# What is a Query?

A **query** is a method that returns a value but that does not modify the object.

# Why can't we violate it?

A method should be responsible for one thing,  
without us have to question to what it does or look to  
see if it has some side effect.

*What should I test  
first?*

# *Effect Sketching*

# Effect Sketching

Effect Sketching draws out what affects a change would happen on your code.

# Interception Point

A point in your program where can detect the effects  
of a particular change

# Effect Sketching

Start with the method that will be affected. This is called the *change point*.

# Which Points to Choose?

Find *interception points* that are closer to your  
*change points*.

# Which Points to Choose?

Given multiple *change points* find the *interception point* where the effects meet.

# *Techniques*

# **Seam Model**

# Seams Defined

A Seam is a place where you can alter behavior in your program without editing in that place.

Michael Feathers - Working Effectively with Legacy Code

# Seam Enabling Points

Every seam has an enabling point, a place where you can make the decision to use one behavior or another.

Michael Feathers - Working Effectively with Legacy Code

# Java Seam Enabling Points

That point (In Java) can be at the classpath.

Michael Feathers - Working Effectively with Legacy Code



# The Problem with Mocking Frameworks

Mocking though useful can get out of hand with multiple setup methods, actions, tear downs, and it's maintenance and lack of readability over time.

# In Java, and OOP you can only minimize mocking

Since the nature of OOP is to have an object with multiple methods, that are interacted in the method you are testing, it is difficult not to use mocking entirely

# Patterns

# Extract Delegate/Class

# Extract Delegate

```
public class Person {  
    private String name;  
    private String areaCode;  
    private String number;  
  
    public String getPhoneString() {  
        return String  
            .format("(%s) %s",  
                areaCode,  
                number);  
    }  
}
```

```
public class PhoneNumber {  
}
```

Step 1: Create a new class with the name you'd like to use.  
Don't add any methods to it yet.

# Extract Delegate

```
public class Person {  
    private String name;  
    private String areaCode;  
    private String number;  
  
    public String getPhoneString() {  
        return String  
            .format("(%s) %s",  
                areaCode,  
                number);  
    }  
}
```

```
public class PhoneNumber {  
    private String areaCode;  
    private String number;  
}
```

Step 2: Create the attributes that need to be moved to the new class

# Extract Delegate

```
public class Person {  
    private String name;  
    private String areaCode;  
    private String number;  
  
    public String getPhoneString() {  
        return String  
            .format("(%s) %s",  
                areaCode,  
                number);  
    }  
}
```

```
public class PhoneNumber {  
    private String areaCode;  
    private String number;  
  
    public String getAreaCode() {...}  
    public String getNumber() {...}  
}
```

Step 3: Create getters in the new delegate to be able to access the values, we will delete them later

# Extract Delegate

```
public class Person {  
    private String name;  
    private String areaCode;  
    private String number;  
    private PhoneNumber phoneNumber;  
  
    public String getPhoneString() {  
        return String  
            .format("(%s) %s",  
                phoneNumber.getAreaCode(),  
                phoneNumber.getNumber());  
    }  
}
```

```
public class PhoneNumber {  
    private String areaCode;  
    private String number;  
  
    public String getAreaCode() {...}  
    public String getNumber() {...}  
}
```

Step 4: Create a delegate instance in the original class and route the values to the new class

# Extract Delegate

```
public class Person {  
    private String name;  
    private PhoneNumber phoneNumber;  
  
    public String getPhoneString() {  
        return String  
            .format("(%s) %s",  
                phoneNumber.getAreaCode(),  
                phoneNumber.getNumber());  
    }  
}
```

```
public class PhoneNumber {  
    private String areaCode;  
    private String number;  
  
    public String getAreaCode() {...}  
    public String getNumber() {...}  
}
```

Step 5: Remove the private member variables since we have rerouted the field storage onto another class

# Extract Delegate

```
public class Person {  
    private String name;  
    private PhoneNumber phoneNumber;  
  
    public String getPhoneString() {  
        return formatted();  
    }  
  
    public String formatted() {  
        return String  
            .format("(%s) %s",  
                phoneNumber.getAreaCode(),  
                phoneNumber.getNumber());  
    }  
}
```

```
public class PhoneNumber {  
    private String areaCode;  
    private String number;  
  
    public String getAreaCode() {...}  
    public String getNumber() {...}  
}
```

Step 6: Extract methods for anything using the get methods of another class. This is called the “Feature Envy” smell and we want to rid ourselves of it.

# Extract Delegate

```
public class Person {  
    private String name;  
    private PhoneNumber phoneNumber;  
  
    public String getPhoneString() {  
        return phoneNumber.formatted();  
    }  
}
```

Step 7: Move the method related to the new class to the new class

```
public class PhoneNumber {  
    private String areaCode;  
    private String number;  
  
    public String getAreaCode() {...}  
    public String getNumber() {...}  
    public String formatted() {  
        return String  
            .format("(%s) %s",  
                   getAreaCode(),  
                   getNumber());  
    }  
}
```

# Extract Delegate

```
public class Person {  
    private String name;  
    private PhoneNumber phoneNumber;  
  
    public String getPhoneString() {  
        return phoneNumber.formatted();  
    }  
}
```

Step 8: Inline to the member variables

```
public class PhoneNumber {  
    private String areaCode;  
    private String number;  
  
    public String getAreaCode() {...}  
    public String getNumber() {...}  
    public String formatted() {  
        return String  
            .format("(%s) %s",  
                areaCode, number);  
    }  
}
```

# Extract Delegate

```
public class Person {  
    private String name;  
    private PhoneNumber phoneNumber;  
  
    public String getPhoneString() {  
        return phoneNumber.formatted();  
    }  
}
```

```
public class PhoneNumber {  
    private String areaCode;  
    private String number;  
  
    public String formatted() {  
        return String  
            .format("(%s) %s",  
                areaCode, number);  
    }  
}
```

Step 9: Delete the getters in the delegate

# Extract Delegate

```
public class Person {  
    private String name;  
    private PhoneNumber phoneNumber;  
  
    public String getPhoneString() {  
        return phoneNumber.formatted();  
    }  
}
```

```
public class PhoneNumber {  
    private String areaCode;  
    private String number;  
  
    public String formatted() {  
        return String  
            .format("(%s) %s",  
                areaCode, number);  
    }  
}
```

Step 10: As a side effect you do have a man-in-middle in the Person class; you can leave it or run analysis and possibly re-route it to the delegate

# Extract Delegate

```
public class Person {  
    private String name;  
    private PhoneNumber phoneNumber;  
  
    public String getPhoneString() {  
        return phoneNumber.formatted();  
    }  
}
```

```
public class PhoneNumber {  
    private String areaCode;  
    private String number;  
  
    public String formatted() {  
        return String  
            .format("(%s) %s",  
                areaCode, number);  
    }  
}
```

Step 11: As another side-effect you have definitely “lightened the load”, smaller class, smaller testing, smaller maintenance.  
Everything in its place

# Extract Delegate

```
public class Person {  
    private String name;  
    private PhoneNumber phoneNumber;  
  
    public String getName();  
    public PhoneNumber getPhoneNumber()  
}
```

```
public class PhoneNumber {  
    private String areaCode;  
    private String number;  
  
    public String formatted() {  
        return String  
            .format("(%s) %s",  
                areaCode, number);  
    }  
}
```

Step 11: You can just get rid of the `PhoneString` method if you'd like. It really is a part of the phone number. Clients can call it if they need it

# Extract Interface

# Extract Interface

```
public interface IAccount {  
}
```

Step 1: Create a new interface with the name you'd like to use. Don't add any methods to it yet.

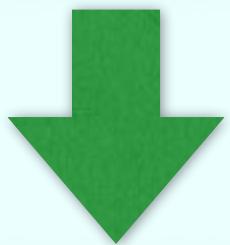
# Extract Interface

```
public interface IAccount {  
}  
  
public class Account implements IAccount {  
}
```

Step 2: Make the class that you are extracting from implement the interface. This can't break anything because the interface doesn't have any methods. But it is good to compile and run your test just to verify that.

# Extract Interface

```
public Account merge(Account account1, Account account2)
```



```
public IAccount merge(IAccount account1, IAccount account2)
```

Step 3: Change the place where you want to use the object so that it uses the interface rather than the original class.

# Extract Interface

```
public interface IAccount {  
    public void deposit(float amt);  
    public void withdrawal(float amt);  
}
```

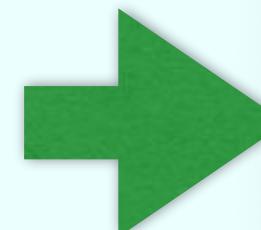
Compile the system and introduce a new method declaration on the interface for each method use that the compiler reports as an error.

# Couple of notes about it...

- Great way to try a new implementation
- Important when someone wants a “change”

# Parameterize Method

# Parameterize Method



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

Extremely Powerful Concept:  
Duplicating Code for Refactoring

Step 1.1: Make a copy of the method

# Parameterize Method

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

Step 3.1: Delete the body of the original method

# Parameterize Method

```
public class Bank {  
    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.apply("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

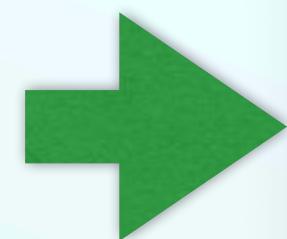
If the function is something where you don't expect much of a different behavior to change you may opt for a "*Parameterize Constructor*" with the function instead.

# 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 Bank {  
    public Bank() {  
        Bar bar = new Bar();  
        bar.init();  
    }  
  
    public baz() {  
        bar.qux();  
    }  
}
```

Step 1: Identify the constructor that you want to parameterize

# Parameterize Constructor

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

Step 1.1: Make a copy of it.

# Parameterize Constructor

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

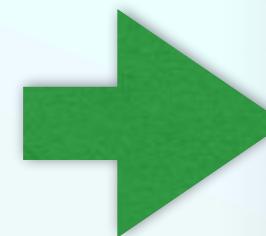
# Parameterize Constructor

```
public class Bank {  
  
    @Deprecated(  
        "Use Bank(Bar) instead")  
    public Bank() {  
        this(new Bar());  
    }  
  
    public Bank(Bar b) {  
        Bar bar = b  
        bar.init();  
    }  
  
    public baz() {  
        bar.qux();  
    }  
}
```

Step 3: Consider  
using @Deprecated  
Annotation

# Parameterize Constructor with Lambdas to overcome static calls

# Parameterize Constructor



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

Step 4: Replace the static with your function

# Parameterize Constructor

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

Addendum: test your method with TDD, refactor

# Parameterize Constructor

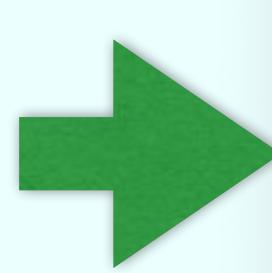
```
public class Bank {  
    private Function<String, Resource> f;  
    @Deprecated("Use Bank(Function) instead")  
    public Bank() {  
        this(Server::getResource)  
    }  
    public Bank(Function<String, Resource> f) {  
        this.f = f;  
    }  
    public void bar(int a, int b) { 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

Addendum: Consider @Deprecated if the old method is not going to be used

# Sprout Class & Method

# **Sprout Method**

# Sprout Method



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

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

# Sprout Method

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

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

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

# Sprout Method

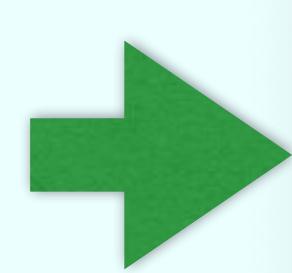
```
public class Bank {  
    int i;  
    public void baz {...}  
    public void bar {  
        float f = baz(i)  
        doSomething1;  
        doSomething2;  
        doSomething3;  
    }  
}
```

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

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

# Sprout Class

# Sprout Class



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

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

# Sprout Class

```
public class Bank {  
    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 Bank {  
    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 Bank {  
    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 Bank {  
    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 Bank {  
    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(){}  
}
```

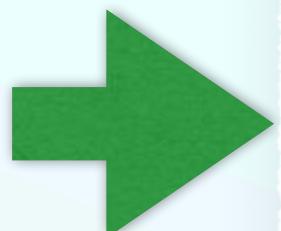
```
public class Bank {  
    int i;  
    public void bar {  
        Other other = new Other(i);  
        float f = other.baz();  
        doSomething1;  
        doSomething2;  
        doSomething3;  
    }  
}
```

Step 6: Remove the comments

# Wrap Class & Method

# **Wrap Method**

# Wrap Method (v1)



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

1. Identify a method you need to change.

# Wrap Method (v1)

```
public class Bank {  
    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 Bank {  
    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

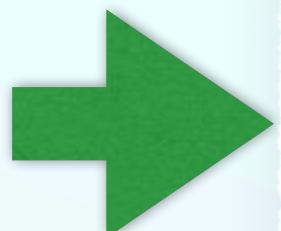
# Wrap Method (v1)

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

1. Identify a method you need to change.

# Wrap Method (v2)

```
public class Bank {  
    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 Bank {  
    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)

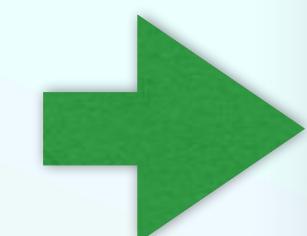
```
public class Bank {  
    public void deposit(int amt) {...}  
    protected void audit(int amt, String msg) {...}  
    public void auditedDeposit(int amt) {  
        audit(amt, "Depositing");  
        deposit(amt);  
        audit(amt, "Deposited");  
    }  
}
```

Addendum: You now have a choice,  
regular or audited

# Wrap Class

# Wrap Class

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



1. Identify a method you need to change.

# Wrap Class

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

```
public class BankImpl implements Bank {  
    public void deposit(int amt) {...}  
}
```

2. Extract an interface from a class

# Wrap Class

```
public interface Bank {  
    public void deposit(int amt);  
  
    public class BankLogger implements Bank {  
        public BankLogger(Bank Bank) {...}  
        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 BankLogger implements Bank {  
    public BankLogger(Bank Bank) {...}  
    protected void audit(int amt, String msg) {...}  
    public void deposit(int amt) {}  
}
```

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

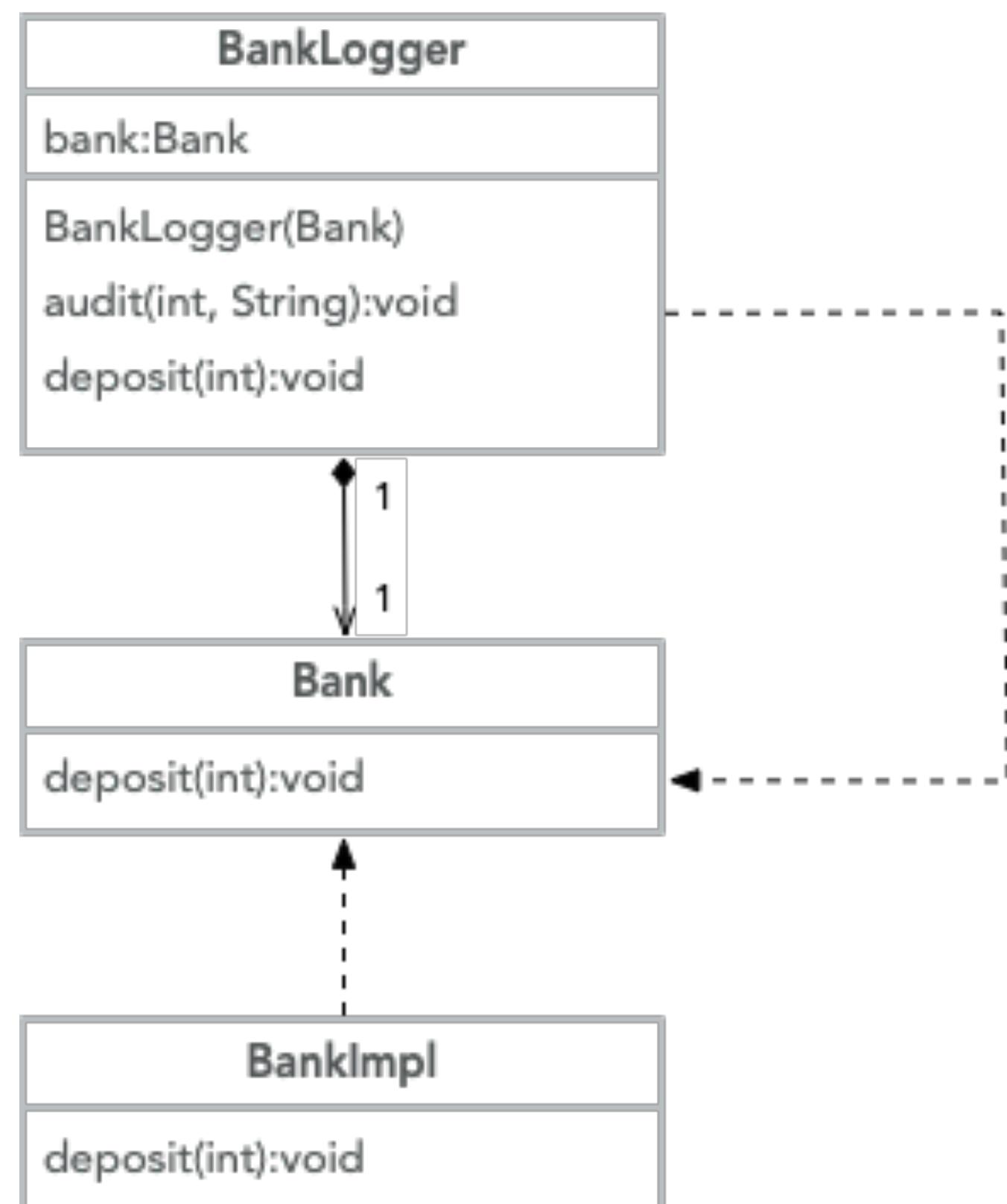
4. Create a new method using TDD that does the new work and behavior

# Wrap Class

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

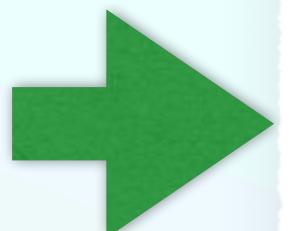
5. Ensure that you call the old methods on the old class

# Wrap Class



# Introduce Instance Delegator

# Introduce Instance Delegator



```
public class Bank {  
    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 Bank {  
    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 Bank {  
    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) {  
        Bank.bar(i + j, status);  
    }  
}
```

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

# Introduce Instance Delegator

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

    @Test
    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(  
                    ));  
        } 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(  
                ));  
        } 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)  
        }  
    }  
}
```

# *Soft Skills*

# **Learn your IDEs well**

# Learn to Jump from Test to Production Quickly

# Testing Thursdays

or you can choose the day

# Scratch Refactoring

# Scratch Refactoring Rules

- Checkout a new branch in version control
- Move stuff around
- **Refactor without tests!**
- **No matter what happens, never, never merge!**

Lambdas will be your  
friend



# Delete Unused Code

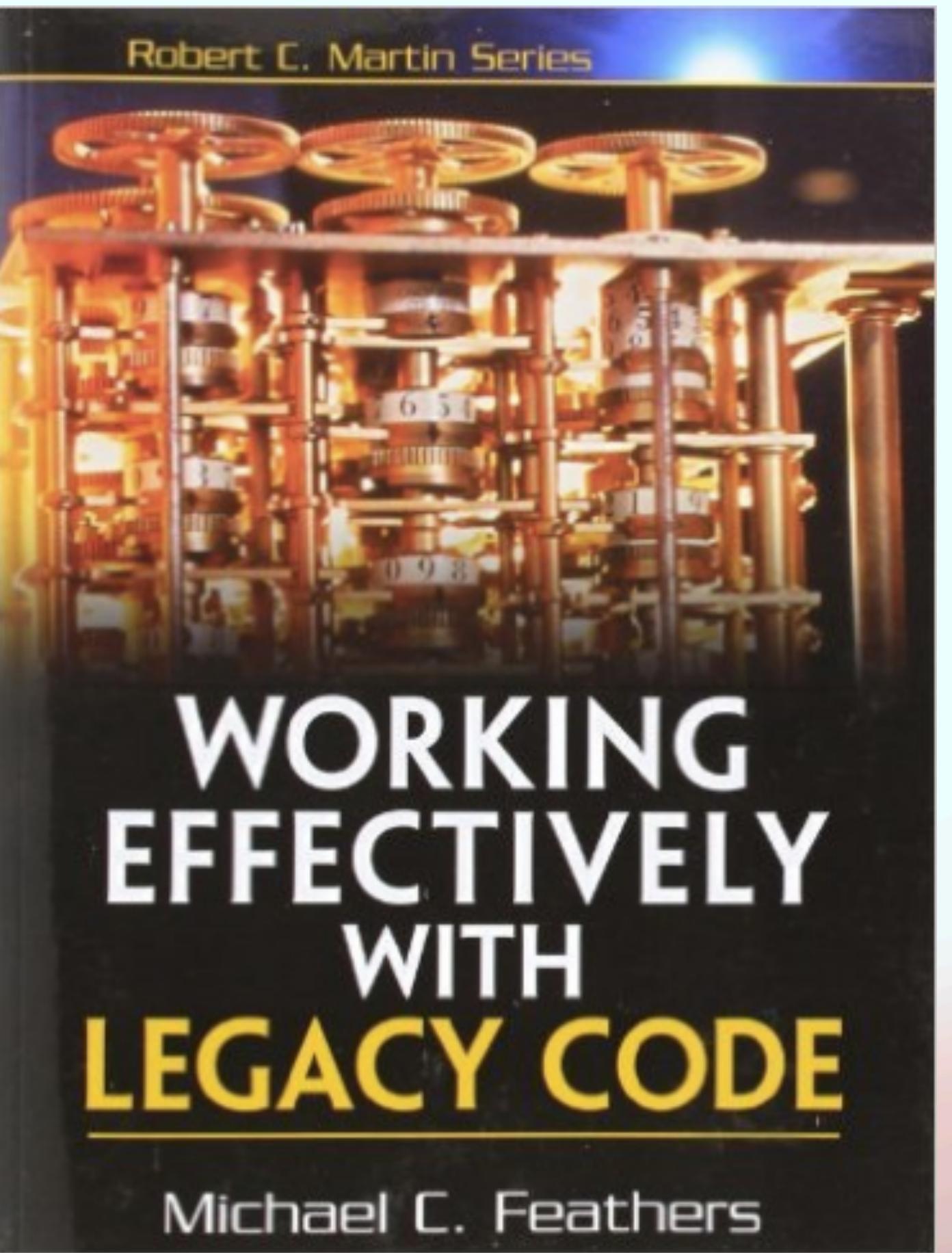
Do you even version control?

# Delete Commented Code

You may never need it

“Programming is the art of  
doing one thing at a time.”

Michael Feathers



# Thank You