

Working Effectively with Legacy Code

Michael Feathers

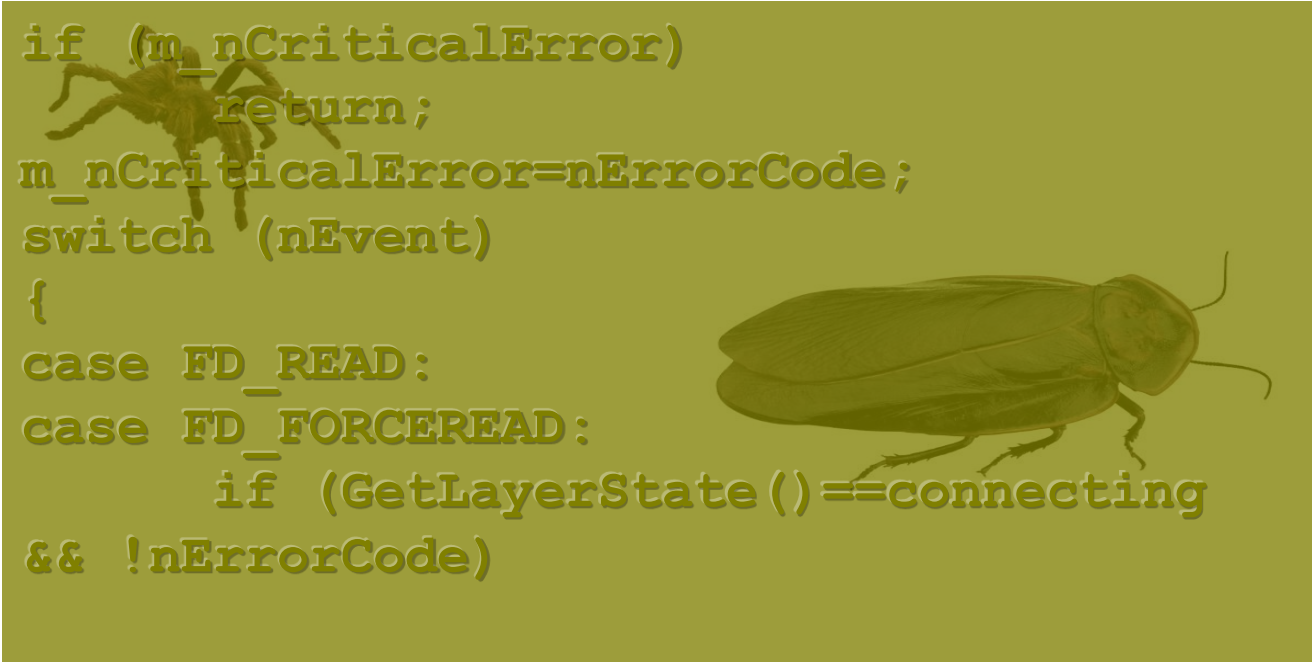
mfeathers@objectmentor.com



Background

- Want to do Extreme Programming
...but transitioning from a dirty code base

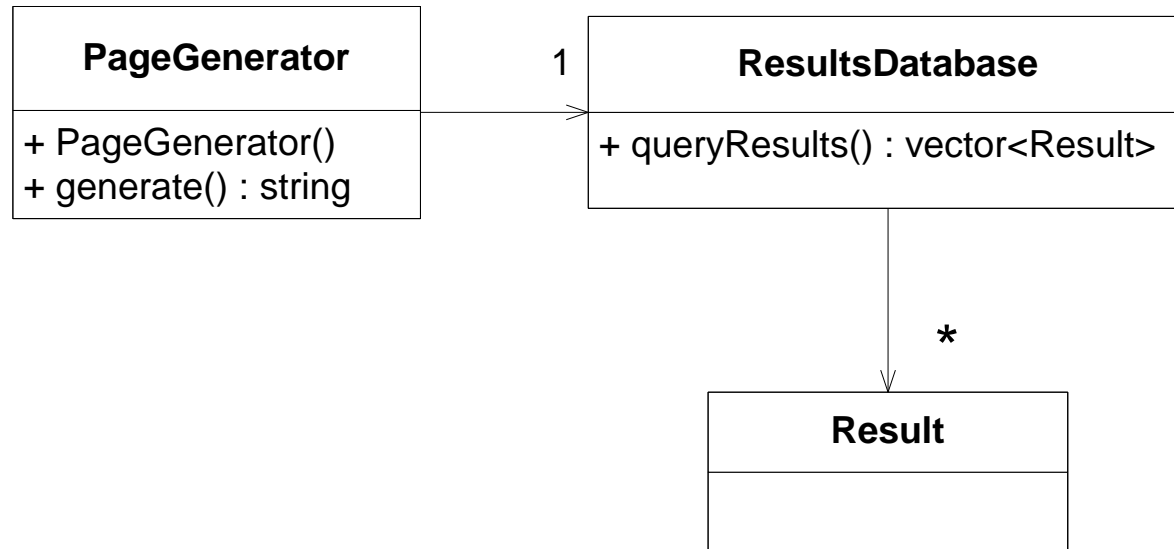
```
if (m_nCriticalError)
    return;
m_nCriticalError=nErrorCode;
switch (nEvent)
{
case FD_READ:
case FD_FORCEREAD:
    if (GetLayerState()==connecting
        && !nErrorCode)
```



First Reaction

- Pretend the code isn't there
- Use TDD to create new classes, make direct changes to the old ones, hope that they are correct
- We'd actually like to be even *more* conservative. We'd like to avoid having to go back to that code ever again.
- Is this viable?

Let's Try It



PageGenerator::generate()

```
std::string PageGenerator::generate()
{
    std::vector<Result> results
        = database.queryResults(beginDate, endDate);

    std::string pageText;

    pageText += "<html>";
    pageText += "<head>";
    pageText += "<title>";
    pageText += "Quarterly Report";
    pageText += "</title>";
    pageText += "</head>";
    pageText += "<body>";
    pageText += "<h1>";
    pageText += "Quarterly Report";
    pageText += "</h1><table>";
    ...
}
```

Continued..

```
...
if (results.size() != 0) {
    pageText += "<table border=\"1\">";
    for (std::vector<Result>::iterator it = results.begin();
         it != results.end(); ++it) {
        pageText += "<tr border=\"1\">";
        pageText += "<td>" + it->department + "</td>";
        pageText += "<td>" + it->manager + "</td>";
        char buffer [128];
        sprintf(buffer, "<td>$$d</td>", it->netProfit / 100);
        pageText += std::string(buffer);
        sprintf(buffer, "<td>$$d</td>", it->operatingExpense / 100);
        pageText += std::string(buffer);

        pageText += "</tr>";
    }
    pageText += "</table>";
} else {
    ...
}
```

Continued..

```
        ...
        pageText += "<p>No results for this period</p>";
    }

    pageText += "</body>";
    pageText += "</html>";

    return pageText;
}

// Argh!!
```

Changes

- Making changes inline makes methods longer
- Changes are untested, the code never has a chance to get better
- We could add code to new methods, and delegate from here
 - Upside:
 - This method won't get longer
 - We can make progress
 - Downsides:
 - this method won't be tested
 - Sometimes, it doesn't work

Chia Pet Pattern

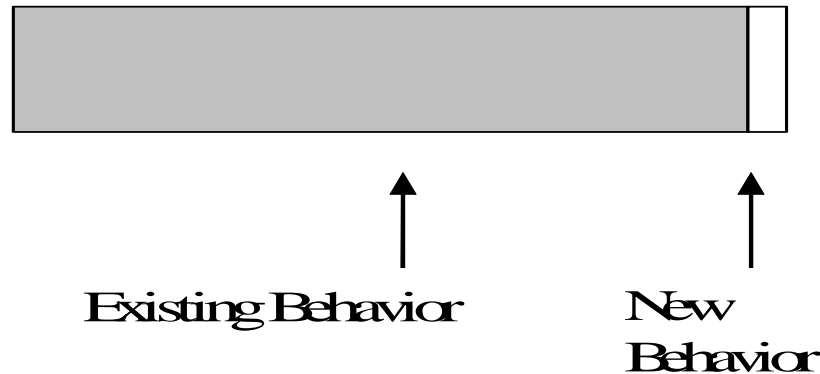
When you have legacy code, try to not to add inline code to it. Instead, write tested code in new classes and methods, and delegate to them. The techniques for doing this are called Sprout Method and Sprout Class



Why are we so Conservative?

When we don't have tests it is easy to make unintentional errors when we change code

Most of development is about preserving behavior.
If it wasn't we would be able to go much faster



The Refactoring Dilemma

When we refactor, we should have tests. To put tests in place, we often have to refactor

Most applications are glued together

- We don't know this until we try to test pieces in isolation



Kinds of Glue

- Singletons (a.k.a. global variables)
 - If there can only be one of an object you'd better hope that it is good for your tests too
- Internal instantiation
 - When a class creates an instance of a hard coded class, you'd better hope that class runs well in tests
- Concrete Dependency
 - When a class uses a concrete class, you'd better hope that class lets you know what is happening to it

Breaking Dependencies

The way that we break dependencies depends upon the tools available to us. If we have safe *extract method* and safe *extract interface* in an IDE, We can do a lot

If we don't, we have to refactor manually, and very *conservatively*. We have to break dependencies as directly and simply as possible.

Sometimes breaking dependencies is ugly

'The Undetectable Side-Effect'

- An ugly Java class (it's uglier inside, trust me!)

AccountDetailFrame
- display : TextField ...
+ performAction(ActionEvent) ...

(told you)

```
public class AccountDetailFrame extends Frame
    implements ActionListener, WindowListener
{
private TextField display = new TextField(10);
...
private AccountDetailFrame(...) { ... }
public void performAction(ActionEvent event) {
    String source = (String)event.getActionCommand();
    if (source.equals("project activity")) {
        DetailFrame detailDisplay = new DetailFrame();
        detailDisplay.setDescription(
            getDetailText() + " " + getProjectText());
        detailDisplay.show();
        String accountDescription =
            detailDisplay.getAccountSymbol();
        accountDescription += ": ";
        ...
        display.setText(accountDescription);
        ...
    }
}
```


Separate a dependency

```
public class AccountDetailFrame extends Frame
    implements ActionListener, WindowListener
{
    private TextField display = new TextField(10);
    ...
    private AccountDetailFrame(...) { ... }
    public void performAction(ActionEvent event) {
        String source = (String)event.getActionCommand();
        if (source.equals("project activity")) {
            DetailFrame detailDisplay = new DetailFrame();
            detailDisplay.setDescription(
                getDetailText() + " " + getProjectText());
            detailDisplay.show();
            String accountDescription =
                detailDisplay.getAccountSymbol();
            accountDescription += ": ";
            ...
            display.setText(accountDescription);
            ...
        }
    }
}
```

After a method extraction..

```
public class AccountDetailFrame extends Frame
    implements ActionListener, WindowListener
{
    ...
    public void performAction(ActionEvent event) {
        performCommand((String)event.getActionCommand());
    }
    void performCommand(String source);
        if (source.equals("project activity")) {
            DetailFrame detailDisplay = new DetailFrame();
            detailDisplay.setDescription(
                getDetailText() + " " + getProjectText());
            detailDisplay.show();
            String accountDescription =
                detailDisplay.getAccountSymbol();
            accountDescription += ": ";
            ...
            display.setText(accountDescription);
            ...
        }
    }
}
```

More offensive dependencies..

```
public class AccountDetailFrame extends Frame
    implements ActionListener, WindowListener
{
    ...
    void performCommand(String source);
        if (source.equals("project activity")) {
            DetailFrame detailDisplay = new DetailFrame();
            detailDisplay.setDescription(
                getDetailText() + " " + getProjectText());
            detailDisplay.show();
            String accountDescription =
                detailDisplay.getAccountSymbol();
            accountDescription += ": ";
            ...
            display.setText(accountDescription);
            ...
        }
    }
```

More refactoring..

```
public class AccountDetailFrame extends Frame
    implements ActionListener, WindowListener
{
    private TextField display = new TextField(10);
    private DetailFrame detailDisplay;
    ...
    void performCommand(String source);
        if (source.equals("project activity")) {
            setDescription(getDetailText() + " " +
                           getProjectText());

            ...
            String accountDescripton = getAccountSymbol();
            accountDescription += ": ";

            ...
            setDisplayText(accountDescription);

            ...
        }
    }
    ...
}
```

The aftermath..

AccountDetailFrame
- display : TextField - detailDisplay : DetailFrame
+ performAction(ActionEvent) + performCommand(String) + getAccountSymbol : String + setDisplayText(String) + setDescription(String)

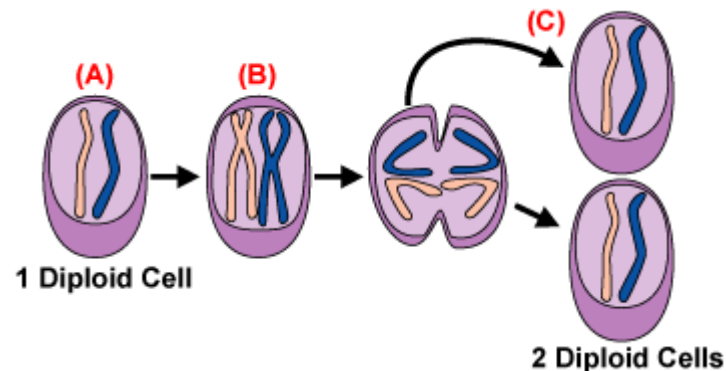
We can subclass and override *getAccountSymbol*, *setDisplayText*, and *setDescription* to sense our work

A Test..

```
public void testPerformCommand() {  
    TestingAccountDetailFrame frame =  
        new TestingAccountDetailFrame();  
  
    frame.accountSymbol = "SYM";  
    frame.performCommand("project activity");  
    assertEquals("06 080 012", frame.getDisplayText());  
}
```

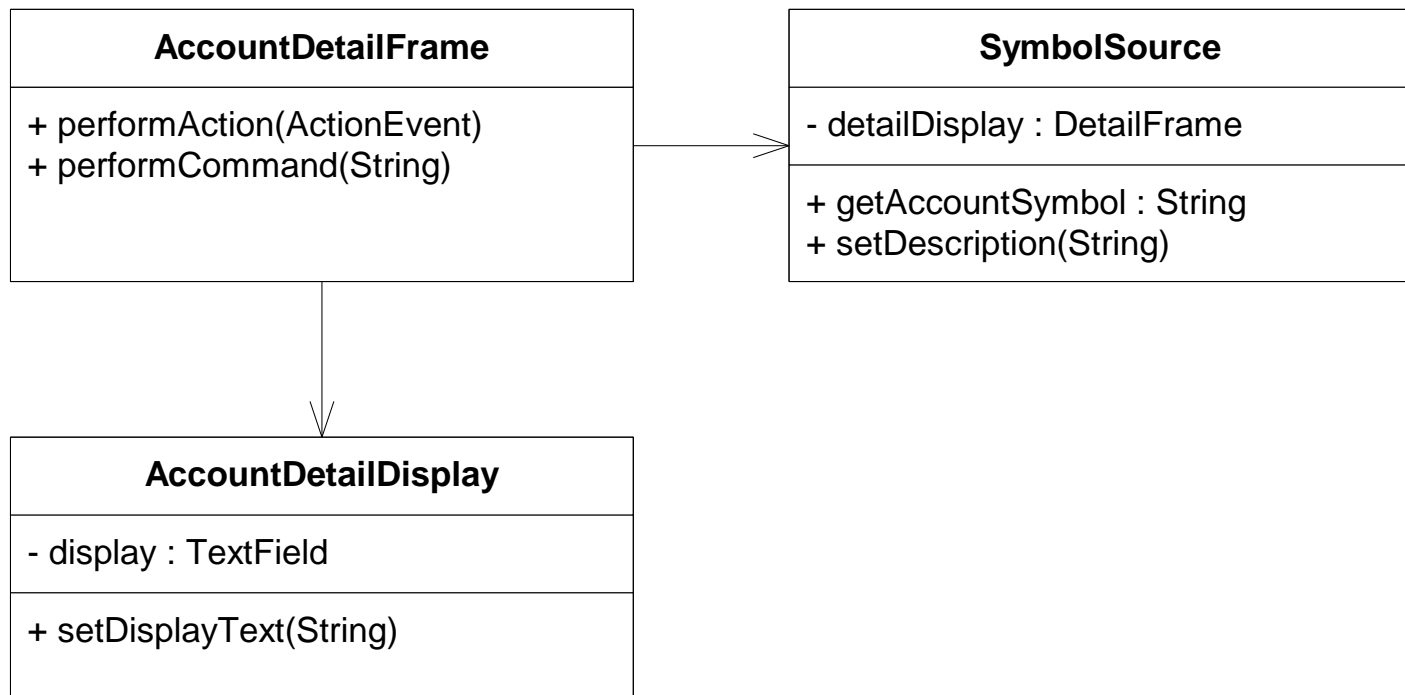
Classes are like cells..

Most designs need some mitosis



Making it Better

- We can make the design better by extracting classes for those separate responsibilities



Better vs. Best

“Best is the enemy of good” – Voltaire (paraphrased)

Automated Tool Moves

- If you have a safe refactoring tool and you are going to use it to break dependencies without tests, do a *refactoring series*: a series of automated refactorings with *no other edits*.
- When tool operations are safe, all that does is move the risk to your edits. Minimize them by leaning on the tool

Manual Moves

- Lean on the Compiler
- Preserve Signatures
- Single Goal Editing

Leaning on the Compiler

- You can use the compiler to navigate to needed points of change by deliberately producing errors
- Most common case:
 - Change a declaration
 - Compile to find the references that need changes

Signature Preservation

- When you are trying to get tests in place favor refactorings which allow you to cut/copy and paste signatures without modification
- Less error prone

Single Goal Editing

- Do one thing at a time

The Legacy Code Change Algorithm

- Identify Change Points
- Identify Test Points
- Break Dependencies
- Write Tests
- Refactor or Change

Development Speed

Mars Spirit Rover

“What can you do in 14 minutes?”



Breaking Dependencies 0

Extract Interface

Extract Implementor



Extract Interface

- A key refactoring for legacy code
- Safe
 - With a few rules in mind you can do this without a chance of breaking your software
- Can take a little time

Extract Interface

- Interface
 - A class which contains *only* pure virtual member-functions
 - Can safely inherit more than one them to present different *faces* to users of your class

Extract Interface Steps

- Find the member-functions that your class uses from the target class
- Think of an interface name for the responsibility of those methods
- Verify that no subclass of target class defines those functions non-virtually
- Create an empty interface class with that name
- Make the target class inherit from the interface class
- Replace all target class references in the client class with the name of the interface
- Lean on the Compiler to find the methods the interface needs
- Copy function signatures for all unfound functions to the new interface. Make them pure virtual

The Non-Virtual Override Problem

- In C++, there is a subtle bug that can hit you when you do an *extract interface*

```
class EventProcessor
{
public:
void handle(Event *event) ;
};

class MultiplexProcessor : public EventProcessor
{
public:
void handle(Event *event) ;
};
```

The Non-Virtual Override Problem

- When you make a function virtual, every function in a subclass with the same signature becomes virtual too
 - In the code on the previous slide, when someone sends the *handle* message through an EventProcessor reference to a MultiplexProcessor, EventProcessor's *handle* method will be executed
 - This can happen when people assign derived objects to base objects. Generally this is *bad*.
- Something to watch out for

The Non-Virtual Override Problem

- So...
 - Before you extract an interface, check to see if the subject class has derived classes
 - Make sure the functions you want to make virtual are not non-virtual in the derived class
 - If they are introduce a new virtual method and replace calls to use it

Extract Implementor

- Same as Extract Interface only we push down instead of pull up.

Breaking Dependencies 1

Parameterize Method

Extract and Override Call



Parameterize Method

- If a method has a hidden dependency on a class because it instantiates it, make a new method that accepts an object of that class as an argument
- Call it from the other method

```
void TestCase::run() {  
    m_result =  
    new TestResult;  
    runTest(m_result);  
}
```

```
void TestCase::run() {  
    run(new TestResult);  
}
```

```
void TestCase::run(  
    TestResult *result) {  
    m_result = result;  
    runTest(m_result);  
}
```

Steps – Parameterize Method

- Create a new method with the internally created object as an argument
- Copy the code from the original method into the old method, deleting the creation code
- Cut the code from the original method and replace it with a call to the new method, using a “new expression” in the argument list

Extract and Override Call

- When we have a bad dependency in a method and it is represented as a call. We can extract the call to a method and then override it in a testing subclass.

Steps – Extract and Override Call

- Extract a method for the call (remember to preserve signatures)
- Make the method virtual
- Create a testing subclass that overrides that method

Breaking Dependencies 2

Link-Time Polymorphism



Link-Time Polymorphism

```
void account_deposit(int amount)
{
    struct Call *call = (struct Call *)
        calloc(1, sizeof (struct Call));

    call->type = ACC_DEPOSIT;
    call->arg0 = amount;
    append(g_calls, call);
}
```

Steps – Link-Time Polymorphism

- Identify the functions or classes you want to fake
- Produce alternative definitions for them
- Adjust your build so that the alternative definitions are included rather than the production versions.

Breaking Dependencies 3

Expose Static Method



Expose Static Method

- Here is a method we have to modify
- How do we get it under test if we can't instantiate the class?

```
class RSCWorkflow {  
    public void validate(Packet packet) {  
        if (packet.getOriginator() == "MIA"  
            || !packet.isValidChecksum()) {  
            throw new InvalidFlow();  
        }  
        ...  
    }  
}
```

Expose Static Method

- Interestingly, the method doesn't use instance data or methods
- We can make it a static method
- If we do we don't have to instantiate the class to get it under test

Expose Static Method

- Is making this method static bad?
- No. The method will still be accessible on instances. Clients of the class won't know the difference.
- Is there more refactoring we can do?
 - Yes, it looks like `validate()` belongs on the packet class, but our current goal is to get the method under test. Expose Static Method lets us do that safely
 - We can move `validate()` to `Packet` afterward

Steps – Expose Static Method

- Write a test which accesses the method you want to expose as a public static method of the class.
- Extract the body of the method to a static method. Often you can use the names of arguments to make the names different as we have in this example: *validate* -> *validatePacket*
- Increase the method's visibility to make it accessible to your test harness.
- Compile.
- If there are errors related to accessing instance data or methods, take a look at those features and see if they can be made static also. If they can, then make them static so system will compile.

Breaking Dependencies 4

Introduce Instance Delegator



Introduce Instance Delegator

- Static methods are hard to fake because you don't have a polymorphic call seam
- You can make one

```
static void BankingServices::updateAccountBalance(  
    int userID,  
    Money amount) {  
    ...  
}
```

Introduce Instance Delegator

- After introduction..

```
public class BankingServices {  
    public static void updateAccountBalance(  
        int userID,  
        Money amount) {  
        ...  
    }  
    public void updateBalance(  
        int userID,  
        Money amount) {  
        updateAccountBalance(userID, amount);  
    }  
}
```


Steps –Introduce Instance Delegator

- Identify a static method that is problematic to use in a test.
- Create an instance method for the method on the class. Remember to preserve signatures.
- Make the instance method delegate to the static method.
- Find places where the static methods are used in the class you have under test and replace them to use the non-static call.
- Use Parameterize Method or another dependency breaking technique to supply an instance to the location where the static method call was made.

Dependency Breaking 5

Template Redefinition



Template Redefinition

- C++ gives you the ability to break dependencies using templates

```
class AsyncReceptionPort
{
private:
    CSocket m_socket;    // bad dependency
    Packet m_packet;
    int m_segmentSize;
    ...
public:
    AsyncReceptionPort();
    void Run();
    ...
};
```

Template Redefinition

```
template<typename SOCKET> class AsyncReceptionPortImpl
{
private:
    SOCKET m_socket;
    Packet m_packet;
    int m_segmentSize;
    ...
public:
    AsyncReceptionPortImpl();
    void Run();
};
typedef AsyncReceptionPortImpl<CSocket>
AsyncReceptionPort;
```

Steps – Template Redefinition

- Identify the features you want to replace in the class you need to test.
- Turn the class into a template, parameterizing it by the variables you need to replace and copying the method bodies up into the header.
- Give the template another name. One common practice is to use the word “Impl” as a suffix for the new template
- Add a typedef statement after the template definition, defining the template with its original arguments using the original class name
- In the test file include the template definition and instantiate the template on new types which will replace the ones you need to replace for test.

Writing Tests



Characterization Testing

- The first tests that you write for a class you are changing.
- These tests characterize existing behavior.
- They hold it down like a vise when you are working

Characterization Testing

- What kind of tests do you need?
- It's great to write as many as you need to
 - Feel confident in the class
 - Understand what it does
 - Know that you can easily add more tests later
- But,
 - The key thing is to detect the existence of behavior that could become broken

Characterization Testing Example

- What kind of tests do we need if we are going to move part of this method to BillingPlan?

```
class ResidentialAccount
void charge(int gallons, date readingDate){
    if (billingPlan.isMonthly()) {
        if (gallons < RESIDENTIAL_MIN)
            balance += RESIDENTIAL_BASE;
        else
            balance += 1.2 *

priceForGallons(gallons);
        billingPlan.postReading(readingDate,
                                gallons);
    }
}
```

Characterization Testing Example

- If we are going to move method to the BillingPlan class is there any way this code will change?
- Do we have to test all of its boundary conditions?

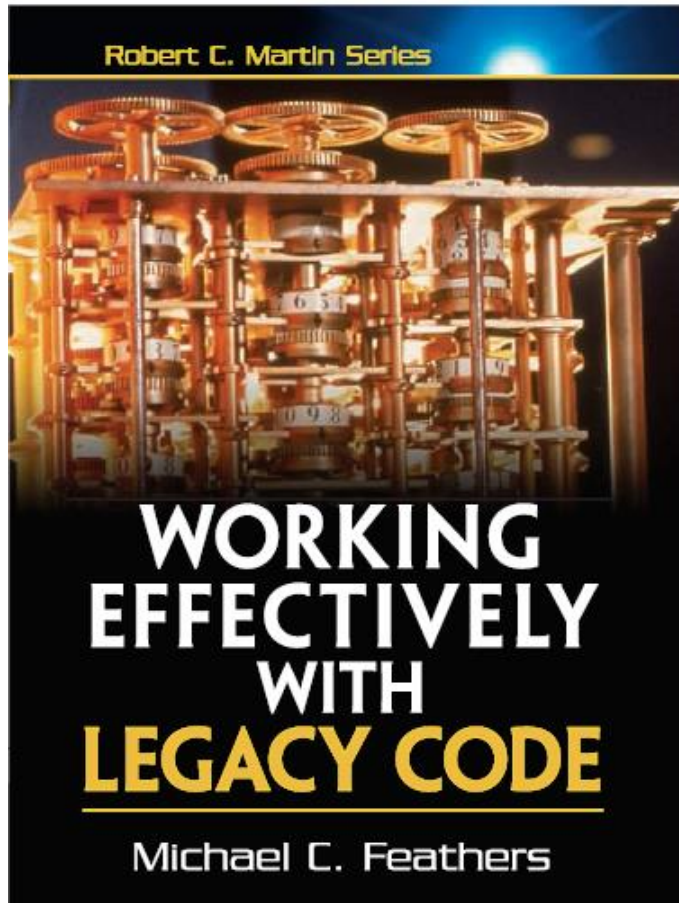
```
if (gallons < RESIDENTIAL_MIN)
    balance += RESIDENTIAL_BASE;
else
    balance += 1.2 *

priceForGallons(gallons);
billingPlan.postReading(readingDate,
                        gallons);
```

Questions & Concerns

- Why is my code uglier? What can I do to alleviate this.
- Does this testing help?
- What about access protection? Can I use reflection for these things?

WELC



**Understanding, Isolating,
Testing, and Changing**

..Untested Code

www.objectmentor.com

www.michaelfeathers.com