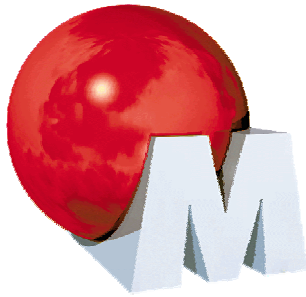


Test Driven Development in Legacy Code



Michael Feathers
mfeathers@objectmentor.com

Background

- Wanted to do XP
...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)
```

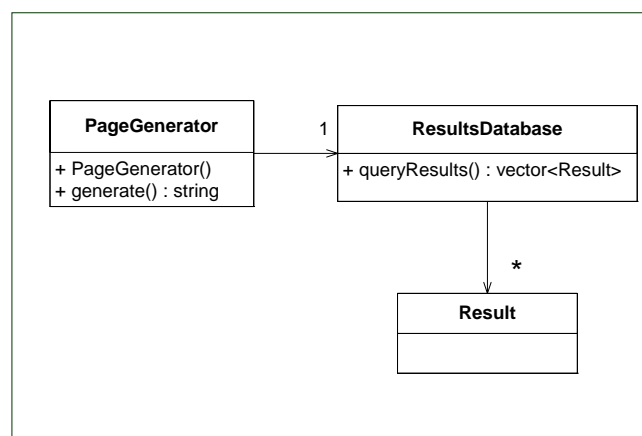
2

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?

3

Let's Try It



4

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>";
    ...
}
```

5

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 {
    ...
}
```

6

Continued..

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

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

    return pageText;
}

// Argh!!
```

7

Chia Pet Pattern

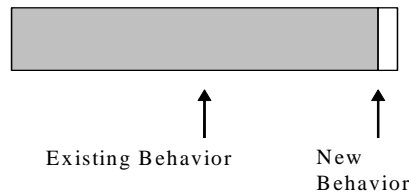
When you have legacy code, try to not to add inline code to it. Instead, write tested code in new classes and methods, and delegate to them. The techniques for doing this are called Sprout Method and Sprout Class



8

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



9

The Refactoring Dilemma

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

10

Most applications are glued together

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



11

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

12

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

13

'The Undetectable Side-Effect'

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

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

14

(told you)

```
public class AccountDetailFrame extends AppFrame
    implements ActionListener, WindowListener
{
    private TextField display = new TextField(10);
    ...
    private AccountDetailFrame(...) { ... }
    public void actionPerformed(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);
            ...
        }
    }
}
```

15

Separate a dependency

```
public class AccountDetailFrame extends AppFrame
    implements ActionListener, WindowListener
{
    private TextField display = new TextField(10);
    ...
    private AccountDetailFrame(...) { ... }
    public void actionPerformed(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);
            ...
        }
    }
}
```

16

After a method extraction..

```

public class AccountDetailFrame extends AppFrame
    implements ActionListener, WindowListener
{
    ...
    public void actionPerformed(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);
            ...
        }
    }
}

```

17

More offensive dependencies..

```

public class AccountDetailFrame extends AppFrame
    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);
            ...
        }
    }
}

```

18

More refactoring..

```
public class AccountDetailFrame extends AppFrame
    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);
            ...
        }
    }
    ...
}
```

19

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

20

A Test..

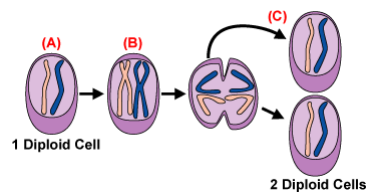
```
public void testPerformCommand() {
    TestingAccountDetailFrame frame =
        new TestingAccountDetailFrame();

    frame.accountSymbol = "SYM";
    frame.performCommand("project activity");
    assertEquals("SYM: actlevel=19/1/1/5", frame.displayText);
}
```

21

Classes are like cells..

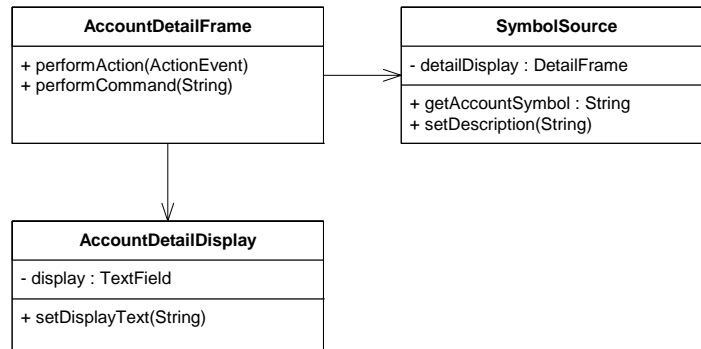
Most designs need some mitosis



22

Making it Better

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



23

Better vs. Best

“Best is the enemy of good” - Voltaire

24

The Legacy Code Change Algorithm

1. Identify Change Points
2. Identify Test Points
3. Break Dependencies
4. Write Tests
5. Refactor or Change

25

A Set of Unit Testing Rules

A Test is not a Unit Test if:

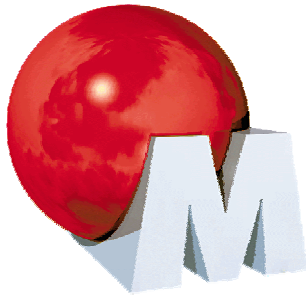
- It touches the file system
- Talks to a database
- It communicates across a network
- You can't run it at the same time as any of your other unit tests

These tests are important and you want to write them but they are not unit tests!

26

Breaking Dependencies

(Conservative Refactoring)



27

Getting Classes Under Test

- Problems
 - Unbuildable Class
 - Uninstantiable Object
 - Uncallable Method

28

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

29

Manual Moves

- Lean on the Compiler
- Preserve Signatures
- Use Sensing Variables

30

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

31

Leaning on the Compiler

```
double domestic_exchange_rate;  
double foreign_exchange_rate;  
  
// Becomes:  
  
class Exchange {  
public:  
    double domestic_exchange_rate;  
    double foreign_exchange_rate;  
};
```

32

Leaning on the Compiler

- Beware of bad cases: inheritance

```
public class CartesianPoint extends Point
{
    ...
    /*
    public int getX() { ... }
    */
}
```

33

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

34

Signature Preservation

```
public void process( List orders,  
                    int dailyTarget,  
                    double interestRate,  
                    int compensationRate  
  
{  
    ...  
}
```

35

Signature Preservation

```
public class Process {  
    List orders,  
    int dailyTarget,  
    double interestRate,  
    int compensationRate  
  
    public Process(List orders,  
                  int dailyTarget,  
                  double interestRate,  
                  int compensationRate)  
  
    {  
        // assign here  
    }  
  
    ...  
}
```

36

Breaking Dependencies

- Manifest Dependencies
 - Extract Interface
 - Adapt Parameter
- Hidden Dependencies (too many to list)
 - Introduce Static Setter
 - Parameterize Constructor/Method
 - Extract and Override Factory Method
 - Introduce Instance Delegator
 - ...

37

Seams

- Suppose we had to test *perimeter* and *distance* was a very expensive operation (not really, but we're using a simple example)

```
double perimeter(vector<point*> polygon)
{
    double result = 0;
    int n;
    for (n = 0; n < polygon.size(); n++) {
        point *next = polygon [(n + 1) %
                               polygon.size()];
        result += distance(polygon [n], *next);
    }
    return result;
}
```

38

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

39

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

40

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);
};
```

41

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

42

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

43

Extract Implementor

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

44

Adapt Parameter

- Sometimes you can't extract an interface or an implementor
 - All you can do is wrap the class
 - Often this is a chance for simplification

45

Tradeoffs – Adapting vs Interfaces

- Adapting allows simplification of interface but introduces another object
- In general interfaces are preferable
- Adapt only when the simplification is significant or use of an interface is impossible.

46

Encapsulate Global References

- Free functions and variables can be faked if they are references from a class
- The reference provides a seam

47

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);  
}
```

48

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

49

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.

50

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

51

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);
}
```

52

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.

53

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 InvalidFlowException;
        }
        ...
    }
}
```

54

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

55

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

56

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.

57

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();
    ...
};
```

58

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;
```

59

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.

60

Hidden Dependencies

- Some dependencies are not present at the interface to a class
 - Two types
 - Global References
 - Internal Object Instantiation

61

Global References

- Static methods and fields are global
- Problems
 - Static mutable data can cause test leakage
 - The state of one test can affect another
 - Statics are hard to mock out if you need to
 - They do not have a *seam* in most OO languages
 - To deal with them we have to introduce *seams*

62

Global References

- Singletons are a classic case where statics present testing issues
 - How do you supercede the singleton?

63

Introduce Static Setter

- Provide a setting method on the singleton that allows you to change its instance
 - Advantages
 - You can change the singleton value between tests (in your setup)
 - Disadvantages
 - Someone can call your singleton setter in production code and corrupt it.
 - How risky is this?

64

Introduce Static Setter

- Two variations:
 - Subclass Singleton
 - Interface Singleton

65

Singletons

- Sometimes singletons are easy under test:

```
public class SomeSingleton
{
    private static SomeSingleton instance;

    ...

    public void resetForTesting() {
        instance = null;
    }
}
```

66

Introduce Instance Delegator

- Static methods are hard to fake because you don't have an object seam
- You can make one

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

67

Introduce Instance Delegator

- After introduction..

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

68

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.

69

Development Speed

Mars Spirit Rover
“What can you do in 14
minutes?”



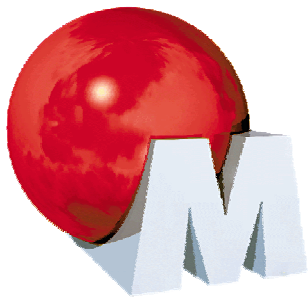
70

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?

71

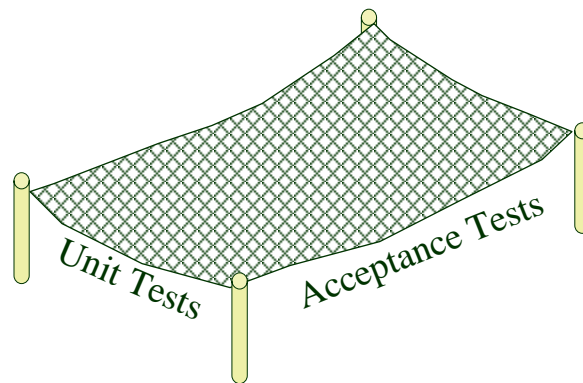
Answers



72

Test-Driven Development

- Tests are *far far* easier to add as you go
 - The ultimate safety net
 - An exoskeleton which allows *code motion*

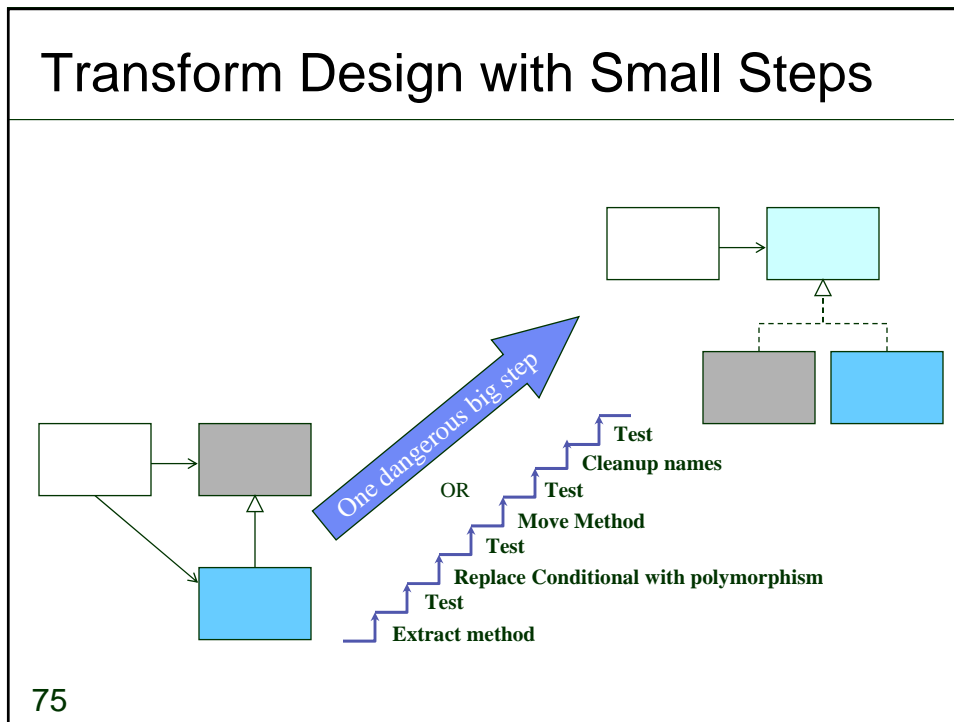


73

Forgiveness

- We have to develop languages and tools which make it easier to recover:
 - Refactoring tools which do deeper analysis
 - Environments which produce more feedback:
 - David Saff (Continuous Testing)
 - Jonathan Edwards (www.subtextual.org)
 - Design languages as if testing mattered

74



Memory

“When I was young, I could imagine a castle with twenty rooms with each room having ten different objects in it. I would have no problem. I can’t do that anymore. Now I think more in terms of earlier experiences. I see a network of inchoate clouds, instead of picture-postcard clearness. But I do write better programs.”

– *Charles Simonyi*

76