Testing Effectively With Legacy Code

By Michael Feathers

Date: Jan 21, 2005

Sample Chapter is provided courtesy of Prentice Hall Professional.

Return to the article

Testing code isn't the easiest thing in the world, and it gets even more complicated when working with legacy code. This chapter explains the theory behind modular coding and how testing can be difficult in this environment.

One of the things that nearly everyone notices when they try to write tests for existing code is just how poorly suited code is to testing. It isn't just particular programs or languages. In general, programming languages just don't seem to support testing very well. It seems that the only ways to end up with an easily testable program are to write tests as you develop it or spend a bit of time trying to "design for testability." There is a lot of hope for the former approach, but if much of the code in the field is evidence, the latter hasn't been very successful.

One thing that I've noticed is that, in trying to get code under test, I've started to think about code in a rather different way. I could just consider this some private quirk, but I've found that this different way of looking at code helps me when I work in new and unfamiliar programming languages. Because I won't be able to cover every programming language in this book, I've decided to outline this view here in the hope that it helps you as well as it helps me.

A Huge Sheet of Text

When I first started programming, I was lucky that I started late enough to have a machine of my own and a compiler to run on that machine; many of my friends starting programming in the punch-card days. When I decided to study programming in school, I started working on a terminal in a lab. We could compile our code remotely on a DEC VAX machine. There was a little accounting system in place. Each compile cost us money out of our account, and we had a fixed amount of machine time each term.

At that point in my life, a program was just a listing. Every couple of hours, I'd walk from the lab to the printer room, get a printout of my program and scrutinize it, trying to figure out what was right or wrong. I didn't know enough to care much about modularity. We had to write modular code to show that we could do it, but at that point I really cared more about whether the code was going to produce the right answers. When I got around to writing object-oriented code, the modularity was rather academic. I wasn't going to be swapping in one class for another in the course of a school assignment. When I got out in the industry, I started to care a lot about those things, but in school, a program was just a listing to me, a long set of functions that I had to write and understand one by one.

This view of a program as a listing seems accurate, at least if we look at how people behave in relation to programs that they write. If we knew nothing about what programming was and we saw a room full of programmers working, we might think that they were scholars inspecting and editing large important documents. A program can seem like a large sheet of text. Changing a little text can cause the meaning of the whole document to change, so people make those changes carefully to avoid mistakes.

Superficially, that is all true, but what about modularity? We are often told it is better to write programs that are made of small reusable pieces, but how often are small pieces reused independently? Not very often.

Reuse is tough. Even when pieces of software look independent, they often depend upon each other in subtle ways.

Seams

When you start to try to pull out individual classes for unit testing, often you have to break a lot of dependencies. Interestingly enough, you often have a lot of work to do, regardless of how "good" the design is. Pulling classes out of existing projects for testing really changes your idea of what "good" is with regard to design. It also leads you to think of software in a completely different way. The idea of a program as a sheet of text just doesn't cut it anymore. How should we look at it? Let's take a look at an example, a function in C++.

```
bool CAsyncSslRec::Init()
{
    if (m_bSslInitialized) {
        return true;
    }
    m_smutex.Unlock();
    m_nSslRefCount++;

    m_bSslInitialized = true;

    FreeLibrary(m_hSslDll1);
    m_hSslDll1=0;
    FreeLibrary(m_hSslDll2);
    m_hSslDll2=0;

    if (!m_bFailureSent) {
        m_bFailureSent=TRUE;
        PostReceiveError(SOCKETCALLBACK, SSL_FAILURE);
    }

    CreateLibrary(m_hSslDll1,"syncesel1.dll");
    CreateLibrary(m_hSslDll2,"syncesel2.dll");

    m_hSslDll1->Init();
    m_hSslDll2->Init();

    return true;
}
```

It sure looks like just a sheet of text, doesn't it? Suppose that we want to run all of that method except for this line:

```
PostReceiveError(SOCKETCALLBACK, SSL_FAILURE);
```

How would we do that?

It's easy, right? All we have to do is go into the code and delete that line.

Okay, let's constrain the problem a little more. We want to avoid executing that line of code because `PostReceiveError` is a global function that communicates with another subsystem, and that subsystem is a pain to work with under test. So the problem becomes, how do we execute the method without calling `PostReceiveError` under test? How do we do that and still allow the call to `PostReceiveError` in production?

To me, that is a question with many possible answers, and it leads to the idea of a seam.

Here's the definition of a seam. Let's take a look at it and then some examples.

Seam

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

Is there a seam at the call to `PostReceiveError`? Yes. We can get rid of the behavior there in a couple of ways. Here is one of the most straightforward ones. `PostReceiveError` is a global function, it isn't part of the

`CAsynchSslRec` class. What happens if we add a method with the exact same signature to the `CAsynchSslRec` class?

```
class CAsyncSslRec
{
    ...
    virtual void PostReceiveError(UINT type, UINT errorcode);
    ...
};
```

In the implementation file, we can add a body for it like this:

```
void CAsyncSslRec::PostReceiveError(UINT type, UINT errorcode)
{
    ::PostReceiveError(type, errorcode);
}
```

That change should preserve behavior. We are using this new method to delegate to the global `PostReceiveError` function using C++'s scoping operator (::). We have a little indirection there, but we end up calling the same global function.

Okay, now what if we subclass the `CAsyncSslRec` class and override the `PostReceiveError` method?

```
class TestingAsyncSslRec : public CAsyncSslRec
{
    virtual void PostReceiveError(UINT type, UINT errorcode)
    {
    }
};
```

If we do that and go back to where we are creating our `CAsyncSslRec` and create a `TestingAsyncSslRec` instead, we've effectively nulled out the behavior of the call to `PostReceiveError` in this code:

```
bool CAsyncSslRec::Init()
{
    if (m_bSslInitialized) {
        return true;
    }
    m_smutex.Unlock();
    m_nSslRefCount++;

    m_bSslInitialized = true;

    FreeLibrary(m_hSslDll1);
    m_hSslDll1=0;
    FreeLibrary(m_hSslDll2);
    m_hSslDll2=0;

    if (!m_bFailureSent) {
        m_bFailureSent=TRUE;
        PostReceiveError(SOCKETCALLBACK, SSL_FAILURE);
   }

   CreateLibrary(m_hSslDll1,"syncesel1.dll");
   CreateLibrary(m_hSslDll2,"syncesel2.dll");

   m_hSslDll1->Init();
   m_hSslDll2->Init();

   return true;
   }
```

Now we can write tests for that code without the nasty side effect.

This seam is what I call an *object seam*. We were able to change the method that is called without changing the method that calls it. *Object seams* are available in object-oriented languages, and they are only one of many different kinds of seams.

Why seams? What is this concept good for?

One of the biggest challenges in getting legacy code under test is breaking dependencies. When we are lucky, the dependencies that we have are small and localized; but in pathological cases, they are numerous and spread out throughout a code base. The seam view of software helps us see the opportunities that are already in the code base. If we can replace behavior at seams, we can selectively exclude dependencies in our tests. We can also run other code where those dependencies were if we want to sense conditions in the code and write tests against those conditions. Often this work can help us get just enough tests in place to support more aggressive work.

Seam Types

The types of seams available to us vary among programming languages. The best way to explore them is to look at all of the steps involved in turning the text of a program into running code on a machine. Each identifiable step exposes different kinds of seams.

Preprocessing Seams

In most programming environments, program text is read by a compiler. The compiler then emits object code or bytecode instructions. Depending on the language, there can be later processing steps, but what about earlier steps?

Only a couple of languages have a build stage before compilation. C and C++ are the most common of them.

In C and C++, a macro preprocessor runs before the compiler. Over the years, the macro preprocessor has been cursed and derided incessantly. With it, we can take lines of text as innocuous looking as this:

```
TEST(getBalance,Account)
{
    Account account;
    LONGS_EQUAL(0, account.getBalance());
}
```

and have them appear like this to the compiler.

```
  class AccountgetBalanceTest : public Test
      { public: AccountgetBalanceTest () : Test ("getBalance" "Test") {}
            void run (TestResult& result_); }
    AccountgetBalanceInstance;
      void AccountgetBalanceTest::run (TestResult& result_)
{
      Account account;
{ result_.countCheck();
  long actualTemp = (account.getBalance());
  long expectedTemp = (0);
  if ((expectedTemp) != (actualTemp))
{ result_.addFailure (Failure (name_, "c:\\seamexample.cpp", 24,
StringFrom(expectedTemp),
StringFrom(actualTemp))); return; } }

}
```

We can also nest code in conditional compilation statements like this to support debugging and different platforms (aarrrgh!):

```
...
m_pRtg->Adj(2.0);

#ifdef DEBUG
#ifndef WINDOWS
    { FILE *fp = fopen(TGLOGNAME,"w");
    if (fp) { fprintf(fp,"%s", m_pRtg->pszState); fclose(fp); }}
#endif

m_pTSRTable->p_nFlush |= GF_FLOT;
#endif
```

...

It's not a good idea to use excessive preprocessing in production code because it tends to decrease code clarity. The conditional compilation directives (`#ifdef`, `#ifndef`, `#if`, and so on) pretty much force you to maintain several different programs in the same source code. Macros (defined with `#define`) can be used to do some very good things, but they just do simple text replacement. It is easy to create macros that hide terribly obscure bugs.

These considerations aside, I'm actually glad that C and C++ have a preprocessor because the preprocessor gives us more seams. Here is an example. In a C program, we have dependencies on a library routine named `db_update`. The `db_update` function talks directly to a database. Unless we can substitute in another implementation of the routine, we can't sense the behavior of the function.

```
#include <DFHLItem.h>
#include <DHLSRecord.h>
extern int db_update(int, struct DFHLItem *);

void account_update(
    int account_no, struct DHLSRecord *record, int activated)
{
    if (activated) {
        if (record->dateStamped && record->quantity > MAX_ITEMS) {
            db_update(account_no, record->item);
        } else {
            db_update(account_no, record->backup_item);
        }
    }
    db_update(MASTER_ACCOUNT, record->item);
}
```

We can use preprocessing seams to replace the calls to `db_update`. To do this, we can introduce a header file called `localdefs.h`.

```
#include <DFHLItem.h>
#include <DHLSRecord.h>

extern int db_update(int, struct DFHLItem *);

#include "localdefs.h"

   void account_update(
   int account_no, struct DHLSRecord *record, int activated)
   {
   if (activated) {
   if (record->dateStamped && record->quantity > MAX_ITEMS) {
   db_update(account_no, record->item);
   } else {
   db_update(account_no, record->backup_item);
   }
   }
   db_update(MASTER_ACCOUNT, record->item);
   }
```

Within it, we can provide a definition for `db_update` and some variables that will be helpful for us:

```
#ifdef TESTING
...
struct DFHLItem *last_item = NULL;
int last_account_no = -1;

#define db_update(account_no,item)    {last_item = (item); last_account_no = (account_no);}
...
#endif
```

With this replacement of `db_update` in place, we can write tests to verify that `db_update` was called with the right parameters. We can do it because the `#include` directive of the C preprocessor gives us a seam that we can use to replace text before it is compiled.

Preprocessing seams are pretty powerful. I don't think I'd really want a preprocessor for Java and other more modern languages, but it is nice to have this tool in C and C++ as compensation for some of the other testing obstacles they present.

I didn't mention it earlier, but there is something else that is important to understand about seams: Every seam has an *enabling point*. Let's look at the definition of a seam again:

Seam

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

When you have a seam, you have a place where behavior can change. We can't really go to that place and change the code just to test it. The source code should be the same in both production and test. In the previous example, we wanted to change the behavior at the text of the `db_update` call. To exploit that seam, you have to make a change someplace else. In this case, the enabling point is a preprocessor define named `TESTING`. When `TESTING` is defined, the `localdefs.h` file defines macros that replace calls to `db_update` in the source file.

Enabling Point

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

Link Seams

In many language systems, compilation isn't the last step of the build process. The compiler produces an intermediate representation of the code, and that representation contains calls to code in other files. Linkers combine these representations. They resolve each of the calls so that you can have a complete program at runtime.

In languages such as C and C++, there really is a separate linker that does the operation I just described. In Java and similar languages, the compiler does the linking process behind the scenes. When a source file contains an `import` statement, the compiler checks to see if the imported class really has been compiled. If the class hasn't been compiled, it compiles it, if necessary, and then checks to see if all of its calls will really resolve correctly at runtime.

Regardless of which scheme your language uses to resolve references, you can usually exploit it to substitute pieces of a program. Let's look at the Java case. Here is a little class called `FitFilter`:

```
package fitnesse;

import fit.Parse;
import fit.Fixture;

import java.io.*;
import java.util.Date;


import java.io.*;
import java.util.*;

public class FitFilter {

    public String input;
    public Parse tables;
    public Fixture fixture = new Fixture();
    public PrintWriter output;

    public static void main (String argv[]) {
        new FitFilter().run(argv);
    }

    public void run (String argv[]) {
        args(argv);
        process();
        exit();
    }

    public void process() {
        try {
            tables = new Parse(input);
            fixture.doTables(tables);
        } catch (Exception e) {
            exception(e);
        }
        tables.print(output);
    }
    ...
}
```

In this file, we import `fit.Parse` and `fit.Fixture`. How do the compiler and the JVM find those classes? In Java, you can use a classpath environment variable to determine where the Java system looks to find those classes. You can actually create classes with the same names, put them into a different directory, and alter the classpath to link to a different `fit.Parse` and `fit.Fixture`. Although it would be confusing to use this trick in production code, when you are testing, it can be a pretty handy way of breaking dependencies.

Suppose we wanted to supply a different version of the Parse class for testing. Where would the seam be?

The seam is the `new Parse` *call in the* `process` *method*.

Where is the enabling point?

The enabling point is the `classpath`.

This sort of dynamic linking can be done in many languages. In most, there is some way to exploit link seams. But not all linking is dynamic. In many older languages, nearly all linking is static; it happens once after compilation.

Many C and C++ build systems perform static linking to create executables. Often the easiest way to use the link seam is to create a separate library for any classes or functions you want to replace. When you do that, you can alter your build scripts to link to those rather than the production ones when you are testing. This can be a bit of work, but it can pay off if you have a code base that is littered with calls to a third-party library. For instance, imagine a CAD application that contains a lot of embedded calls to a graphics library. Here is an example of some typical code:

```
void CrossPlaneFigure::rerender()
{
    // draw the label
    drawText(m_nX, m_nY, m_pchLabel, getClipLen());
    drawLine(m_nX, m_nY, m_nX + getClipLen(), m_nY);
    drawLine(m_nX, m_nY, m_nX, m_nY + getDropLen());
    if (!m_bShadowBox) {
        drawLine(m_nX + getClipLen(), m_nY,
                    m_nX + getClipLen(), m_nY + getDropLen());
        drawLine(m_nX, m_nY + getDropLen(),
                    m_nX + getClipLen(), m_nY + getDropLen());
    }

    // draw the figure
    for (int n = 0; n < edges.size(); n++) {
        ...
    }

    ...
}
```

This code makes many direct calls to a graphics library. Unfortunately, the only way to really verify that this code is doing what you want it to do is to look at the computer screen when figures are redrawn. In complicated code, that is pretty error prone, not to mention tedious. An alternative is to use link seams. If all of the drawing functions are part of a particular library, you can create stub versions that link to the rest of the application. If you are interested in only separating out the dependency, they can be just empty functions:

```
void drawText(int x, int y, char *text, int textLength)
{
}

void drawLine(int firstX, int firstY, int secondX, int secondY)
{
}
```

If the functions return values, you have to return something. Often a code that indicates success or the default value of a type is a good choice:

```
int getStatus()
{
    return FLAG_OKAY;
}
```

The case of a graphics library is a little atypical. One reason that it is a good candidate for this technique is that it is almost a pure "tell" interface. You issue calls to functions to tell them to do something, and you aren't asking for much information back. Asking for information is difficult because the defaults often aren't the right thing to return when you are trying to exercise your code.

Separation is often a reason to use a link seam. You can do sensing also; it just requires a little more work. In the case of the graphics library we just faked, we could introduce some additional data structures to record calls:

```
std::queue<GraphicsAction>  actions;

void drawLine(int firstX, int firstY, int secondX, int secondY)
{
    actions.push_back(GraphicsAction(LINE_DRAW,
            firstX, firstY, secondX, secondY);
}
```

With these data structures, we can sense the effects of a function in a test:

```
TEST(simpleRender,Figure)
{
    std::string text = "simple";
    Figure figure(text, 0, 0);

    figure.rerender();
    LONGS_EQUAL(5, actions.size());
    GraphicsAction action;
    action  = actions.pop_front();
    LONGS_EQUAL(LABEL_DRAW, action.type);

    action  = actions.pop_front();
    LONGS_EQUAL(0, action.firstX);
    LONGS_EQUAL(0, action.firstY);
    LONGS_EQUAL(text.size(), action.secondX);
}
```

The schemes that we can use to sense effects can grow rather complicated, but it is best to start with a very simple scheme and allow it to get only as complicated as it needs to be to solve the current sensing needs.

The enabling point for a link seam is always outside the program text. Sometimes it is in a build or a deployment script. This makes the use of link seams somewhat hard to notice.

Usage Tip

If you use link seams, make sure that the difference between test and production environments is obvious.

Object Seams

Object seams are pretty much the most useful seams available in object-oriented programming languages. The fundamental thing to recognize is that when we look at a call in an object-oriented program, it does not define which method will actually be executed. Let's look at a Java example:

```
cell.Recalculate();
```

When we look at this code, it seems that there has to be a method named `Recalculate` that will execute when we make that call. If the program is going to run, there has to be a method with that name; but the fact is, there can be more than one:

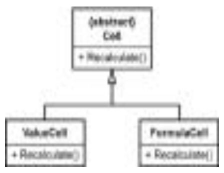

Figure 4.1 Cell hierarchy.

Which method will be called in this line of code?

```
cell.Recalculate();
```

Without knowing what object `cell` points to, we just don't know. It could be the `Recalculate` method of `ValueCell` or the `Recalculate` method of `FormulaCell`. It could even be the `Recalculate` method of some other class that doesn't inherit from `Cell` (if that's the case, `cell` was a particularly cruel name to use for that variable!). If we can change which `Recalculate` is called in that line of code without changing the code around it, that call is a seam.

In object-oriented languages, not all method calls are seams. Here is an example of a call that isn't a seam:

```
public class CustomSpreadsheet extends Spreadsheet
{
    public Spreadsheet buildMartSheet() {
        ...
        Cell cell = new FormulaCell(this, "A1", "=A2+A3");
        ...
        cell.Recalculate();
   ...
   }
   ...
   }
```

In this code, we're creating a cell and then using it in the same method. Is the call to `Recalculate` an object seam? No. There is no enabling point. We can't change which `Recalculate` method is called because the choice depends on the class of the cell. The class of the cell is decided when the object is created, and we can't change it without modifying the method.

What if the code looked like this?

```
public class CustomSpreadsheet extends Spreadsheet
{
    public Spreadsheet buildMartSheet(Cell cell) {
   ...
   cell.Recalculate();
   ...
   }
   ...
   }
```

Is the call to `cell.Recalculate` in `buildMartSheet` a seam now? Yes. We can create a `CustomSpreadsheet` in a test and call `buildMartSheet` with whatever kind of `Cell` we want to use. We'll have ended up varying what the call to `cell.Recalculate` does without changing the method that calls it.

Where is the enabling point?

In this example, the enabling point is the argument list of `buildMartSheet`. We can decide what kind of an object to pass and change the behavior of `Recalculate` any way that we want to for testing.

Okay, most object seams are pretty straightforward. Here is a tricky one. Is there an object seam at the call to `Recalculate` in this version of `buildMartSheet`?

```
public class CustomSpreadsheet extends Spreadsheet
{
    public Spreadsheet buildMartSheet(Cell cell) {
        ...
        Recalculate(cell);
   ...
   }

   private static void Recalculate(Cell cell) {
   ...
   }

   ...
   }
```

The `Recalculate` method is a static method. Is the call to `Recalculate` in `buildMartSheet` a seam? Yes. We don't have to edit `buildMartSheet` to change behavior at that call. If we delete the keyword `static` on `Recalculate` and make it a protected method instead of a private method, we can subclass and override it during test:

```
public class CustomSpreadsheet extends Spreadsheet
{
    public Spreadsheet buildMartSheet(Cell cell) {
        ...
        Recalculate(cell);
        ...
    }

    protected void Recalculate(Cell cell) {
   ...
   }

   ...
   }

   public class TestingCustomSpreadsheet extends CustomSpreadsheet {
   protected void Recalculate(Cell cell) {
   ...
   }
   }
```

Isn't this all rather indirect? If we don't like a dependency, why don't we just go into the code and change it? Sometimes that works, but in particularly nasty legacy code, often the best approach is to do what you can to modify the code as little as possible when you are getting tests in place. If you know the seams that your language offers and how to use them, you can often get tests in place more safely than you could otherwise.

The seams types I've shown are the major ones. You can find them in many programming languages. Let's take a look at the example that led off this chapter again and see what seams we can see:

```
bool CAsyncSslRec::Init()
{
    if (m_bSslInitialized) {
        return true;
    }
    m_smutex.Unlock();
    m_nSslRefCount++;

    m_bSslInitialized = true;

    FreeLibrary(m_hSslDll1);
    m_hSslDll1=0;
    FreeLibrary(m_hSslDll2);
    m_hSslDll2=0;
    if (!m_bFailureSent) {
        m_bFailureSent=TRUE;
        PostReceiveError(SOCKETCALLBACK, SSL_FAILURE);
   }

   CreateLibrary(m_hSslDll1,"syncesel1.dll");
   CreateLibrary(m_hSslDll2,"syncesel2.dll");

   m_hSslDll1->Init();
   m_hSslDll2->Init();
   return true;
   }
```

What seams are available at the `PostReceiveError` call? Let's list them.

1. `PostReceiveError` is a global function, so we can easily use the *link seam* there. We can create a library with a stub function and link to it to get rid of the behavior. The enabling point would be our makefile or some setting in our IDE. We'd have to alter our build so that we would link to a testing library when we are testing and a production library when we want to build the real system.
2. We could add a `#include` statement to the code and use the preprocessor to define a macro named `PostReceiveError` when we are testing. So, we have a *preprocessing seam* there. Where is the *enabling point*? We can use a preprocessor define to turn the macro definition on or off.
3. We could also declare a virtual function for `PostRecieveError` like we did at the beginning of this chapter, so we have an *object seam* there also. Where is the enabling point? In this case, the enabling point is the place where we decide to create an object. We can create either an `CAsyncSslRec` object or an object of some testing subclass that overrides `PostRecieveError`.

It is actually kind of amazing that there are so many ways to replace the behavior at this call without editing the method:

```
bool CAsyncSslRec::Init()
{
   ...
   if (!m_bFailureSent) {
       m_bFailureSent=TRUE;
       PostReceiveError(SOCKETCALLBACK, SSL_FAILURE);
   }
   ...

   return true;
   }
```

It is important to choose the right type of seam when you want to get pieces of code under test. In general, *object seams* are the best choice in object-oriented languages. *Preprocessing seams* and *link seams* can be useful at times but they are not as explicit as *object seams*. In addition, tests that depend upon them can be hard to maintain. I like to reserve *preprocessing seams* and *link seams* for cases where dependencies are pervasive and there are no better alternatives.

When you get used to seeing code in terms of seams, it is easier to see how to test things and to see how to structure new code to make testing easier.
