Working Effectively with Legacy Code
By Michael C. Feathers

Foreword •

Ian Bruntlett, 8 June 2011

The Foreward, by Robert C. Martin, considers the route from a programmer's first experiences to being someone tasked to maintain legacy systems. He mentions that the book discusses removing the "legacy" aspects of the system – i.e. "reversing the rot".

Preface •

Ian Bruntlett, 8 June 2011

The Preface, by Michael C. Feathers, discusses "what is legacy code?" and decides that it is "code without tests".

Chapter 1: Changing Software •

Ian Bruntlett, 8 June 2011

Chapter 1, "Changing Software", lists four reasons to change software (Adding a feature, Fixing a bug, improving the design and Optimising resource usage). The final part of this chapter, "Putting it all together looks at these reasons and discusses factors that change – Structure, New Functionality, Functionality and Resource Usage. It concludes by discussing how programmers write "anti-refactoring" code (my neologism) – e.g. adding code to an existing method instead of a new method and how fear leads to big, unwieldy, classes developing over time so that programmers are too scared of breaking things to do something like breaking a big class into pieces.

Chapter 2: Working with Feedback •

Martin Moene, 20 June 2011

This chapter motivates us to say "yes!" to unit testing as one of the most important components of working with legacy code for the quick and localised feedback that it provides. The chapter also gives us a feel of how to go about to change such code.

I like the author's conversational writing style. He also takes what's needed to sketch a situation and to explain what's going on. A good example is the way the author builds the case for unit testing in the first section. It talks about ways to make changes in a system and ways of testing and it can be summarised as follows.

There are two primary ways to make changes in a system. The author calls them *Edit and Pray* and *Cover and Modify*.

Great care, both before and after making the changes, is what typifies *Edit and Pray*. However taking great care is not the same as being safe and effective. In *Cover and Modify* we work with a safety net, or better still a cloak over the code we're changing that protects us from bad changes to leak. Tests cover the code. Via the tests we quickly learn if our changes were good or bad. With this feedback we can make changes more carefully.

Funny, as I liked to describe my assembly programming for MC6805 micro-controllers without using an in-circuit emulator as just that: "programming very carefully". Even so, I felt the need for detailed feedback so strongly that I at least tested mathematical routines via an emulator. I think those must have been my first unit(-like) tests. This was sometime in the late eighties.

After the two ways of making changes, we look at two ways of testing: *testing to attempt to show correctness*[1] and *testing to detect change*.

In the first way of testing, tests are created after the software it tests and possibly by other programmers. The feedback loop is large both in the amount of work tested together for the first time and in its time scale. The second manner, also called regression testing, is used to check that what worked in the past still works now. Regression tests act as a software vise[2]: it lets us be in control of our work.

According to the author, regression tests are **often** done at the relatively high level of the application. I wonder ifn't the kind of project and the branch you're working in may make quite a difference in how far this statement is true.

A short story shows the ill effect of too high-level, too coarse-grained regression tests combined with time delay in obtaining feedback. Although the tests tell something is wrong, their coarse grain prevents us to see immediately what exactly is wrong. The long time that elapses between change and feedback from the tests make it difficult to act effectively.

In a contrasting story we make a change to unclear code that is covered with unit tests. First we refactor the code 1) to make it more clear, also for later readers that may be ourselves, 2) to better understand it, and 3) to have more confidence in the change we want to make. All these changes are supported by the existing unit tests and tests that we add to verify new behaviour.

The author concludes that unit testing is one of the most important components in legacy code work, because it gives feedback as you develop and it lets you refactor with much more safety.

What is Unit Testing?

I think the author quite clearly expresses his ideas on what tests to consider unit tests. At the same time he does not dumb down tests that do not fall in that category. As with the term 'legacy code', I'm happy to accept this as a working definition for the book.

In unit testing we strive to test 'components' (individual functions, classes) in isolation, independent from each other. If we test broader functional areas the tests are more like integration test. These larger tests are also important but they have problems with:

- Error localisation,
- Execution time and
- Coverage.

Thus good unit tests:

1. run fast, and
2. help us to localise problems.

For these reasons, a test is not a unit test if:

1. it talks to a database,
2. it communicates across a network,
3. it touches the filesystem,
4. you have special things to do to your environment to run it (e.g. edit a configuration file).

Such non-unit tests often are worth writing using the unit test harness. However make sure to be able to separate them from true unit tests so that you can run a set of tests fast whenever you make changes.

Higher Level Testing

Testing at a higher level than that of a unit is also important, as it can pin down behaviour for a set of classes at a time. Little more is said here, but in chapter 12 it reappears as testing *one level back*.[3]

Test Coverings

An example that gives an idea of bringing code under test leads us to *The Legacy Code Dilemma*:

> When we change code, we should have tests in place. To put tests in place, we have to change the code.

To test a component you must be able to instantiate it. As it is desirable to test components in isolation, much of the work to make that possible involves breaking dependencies on arguments, base classes, member objects. In some cases a class can be replaced by an interface that you then implement specifically for the test. Sometimes an involved argument can be replaced by a *primitive parameter* if that conveys all the information required.

As the code is not yet covered by tests, the advise is to begin with relatively safe refactorings and apply them very conservatively. Such changes can make code uglier at first, but they do make (the initial) testing possible. Lateron when this dependency-breaking code also has been covered by tests you have the opportunity to *heal that scar too*.

I wonder if *one class per file* (or closely-related set of -) is a presumed precondition to cover a class with a test.

The Legacy Code Change Algorithm

Changes we make to legacy code not only should bring functional changes, they also should bring more and more code under test, so that working with the code base becomes increasingly easy.

Here's a code-changing algorithm you can use:

1. Identify change points.
2. Find test points.
3. Break dependencies.
4. Write tests.
5. Make changes and refactor.

Short explanations of these items follow and point us to relevant chapters in the book. The next two chapters contain information on three critical concepts in legacy work: sensing, separation and seams.

Closing Remarks

A good part of this review actually is a summary, which together with the discussion forms a collective review. For me the summarising aspect is important to easily see the main points and to make it possible to come back later and dive in again (as with the PDF of [3]).

Notes

[1] Formulated as "testing to attempt to show correctness", because tests can only indicate the presence of errors, not their absence.

[2] I looked-up Vise at Dictionary.com: "any of various devices, usually having two jaws that may be brought together or separated by means of a screw, lever, or the like, used to hold an object firmly while work is being done on it." I like the explicit mentioning of "while work is being done on it" in this version.

[3] In "Growing Object-Oriented Software" (GOOS), Steve Freeman and Nat Pryce *expand the sphere of testing to include acceptance testing, integration testing and end-to-end testing as well as unit testing*. GOOS was the subject of the preceding ACCU mentored developers project. See http://www.eld.leidenuniv.nl/~moene/accu/mentored-goos/GOOS-ACCUMentoredDevelopersProject.pdf (gees/goos).

Chapter 3: Sensing and Separation •

Richard Barrett, 27 June 2011

When first starting to develop using unit tests, or writing them to test existing code, it is dependencies among classes that is the first cause of problems. An introduction to breaking dependencies is the subject of this chapter.

Why do we want to break dependencies when putting tests in place? Michael gives us two reasons: 'sensing' and 'separation'. When there is a dependency between two or more classes, discovering a change in a class-under-test often means having to use the other dependent classes - it's then not a 'unit' test. We break dependencies so that we can 'sense' the effects of our unit test method calls to our class-under-test, and so that we can run the unit test 'separately' from the rest of the application. Michael illustrates this nicely with an example.

Michael then asks which is the tougher: sensing or separation? Separation is the subject of a number of techniques that we will learn about later in our studies, but for sensing there is one primary tool, the subject of the remainder of the chapter...

Fake Collaborators

The gist of what we do here is this: when testing a piece of code, we fake the rest of the code with which our code-under-test collaborates. In object-oriented software, we call these fake objects. We fake the collaborators, so that we can sense the effects of our test's actions through the fakes.

Michael explains this via the classic example of an application that includes a GUI: say we have an input to the application, that input is processed, and some effect is displayed on the GUI. Without the separation of responsibilities into distinct ares of code, it is difficult to test, for example just the input processing. Identifying an interface for the GUI code, and then writing a 'fake' GUI that implements that interface, allows us to test just the input processing by 'sensing' the effects of the test in our fake. This is illustrated in the text with a point-of-sale class in Java; the example implementation shows the way that a fake collaborator is used when writing a unit test.

There's a discussion concerning the two 'sides' of a fake object: one side represents the functionality that only the code under test will see, and the other side is for the use of the code implementing the test only. This is a nice concept.

Having explained the benefits of fake objects, the author moves on to introduce 'mock' objects. Mock objects are defined as being like fake objects, but performing the test assertions internally. With plain fake objects, the test code uses the fake object's functionality to detect some effect and compare that against the expected effect. With mocks the comparison is done internally. This is a powerful technique, but you can still do a lot with just fake objects.

In summary, I thought this was a good introductory chapter to breaking dependencies by the use of, and rationale for, fake objects. I like the "sensing and

separation" and the "two-sides of a fake object" concepts; they'll be useful when explaining this to others.

Chapter 4: The Seam Model •

Andrew McDonnell, 4 July 2011

The chapter begins saying there seems to be really only one way to end up with a testable piece of software: write tests as you develop. So, I guess we can give up now?

Alright, we can't give up. So how do we make untestable software testable? Unit testing requires pulling individual classes out of the system to test in isolation. When you do this you become aware in excruciating detail of all the dependencies (I can vouch for this), which need to be broken. Now, it can be easy enough to break dependencies by going in and changing the code you want to test, but that violates our goal of introducing tests **before** changing the code.

This leads to the introduction of seams: "A seam is a place where you can alter behavior in your program without editing in that place". You can leverage seams to break dependencies on things you don't want to test with this class (separation) and to replace them with fakes or mocks (sensing). The chapter shows how to introduce seams to existing code and put them to use.

We also get the definition of enabling point, which is the code point where you make the decision of what happens at the seam.

As an aside, I have a minor quibble with the term seam, because to me a seam is something sewn and fairly permanent. Maybe "zipper" does a better job reflecting how easy it is to swap out behavior to me? Seam sounds better than zipper, though.

There are three main types of seam. Not all are available in every language, but they all have uses.

Preprocessor seam: Use the preprocessor to switch behavior.

Create a macro that replaces a function call with different behavior. Then when you #include that, you get the other behavior. Your code would conditionally #include the macro, for example by a preprocessor define called TESTING. The state of TESTING determines what code is compiled in (the enabling point). Obviously this seam is available in C and C++ but not Java or interpreted languages.

Link seam: Use the linker to switch behavior.

By changing what libraries you link to, you can change the behavior for everything in those libraries. In Java this would mean changing the classpath (enabling point) to point to testing classes with the same name. With C and C++ it's a little harder if you're linking to static libraries, because you have to create a whole testing library and make your build script choose where to link (so the enabling point would be a TESTING build script variable, for example).

The link seam is particularly powerful if you have a file full of calls to a third-party library. You can wholesale replace them by changing the link target, and you can even introduce sensing by adding some logging to the testing library. Of course this works best for a pure tell library - if you depend on getting things back from the library (feedback or object creation, .e.g.) there are more complications.

The final point is that the enabling point for link seams is difficult to notice, so you should make the difference between test and production environments obvious.

Object seam: Use function resolution to switch behavior.
As the name implies, this means switching out what object calls a method. Based on your choice of object, different behavior occurs. In this case you may have to make some changes to the test class (although not the line of the seam) to allow the seam to work.

In one example, the object is created in the function under test. This means there's no seam to swap it out. You have to pass the object as an argument somewhere to break that dependency and create a seam - which has the added bonus of making the dependency explicit. Furthermore, in order to be able to choose behavior at runtime, it of course needs to be overridable.

Returning to the first example in the chapter, it turns out you can use any of the seam types to replace the function call in question. The general recommendation, though, is to use object seams when possible and reserve the other types for more complicated cases. I'll note also that the preprocessor and link seams require your production code and/or build scripts to be aware of testing state, while the object seam does not.

I think it's also worth noting that the first example demonstrates the object seam, even though the call in question is a global method. To create the seam we're actually introducing an object method to intercept the call and choosing behavior there - so object seams aren't limited to things that are already objects!

The seam model is an interesting way to look at code, and it's kind of amazing how many seams you can find when you learn how to look. At my job we have a large C++ application I'd like to start getting under test. I've made some preliminary experimental efforts, which mostly impressed on me just how many dependencies there actually are (very close to the point of everything depending on everything else, actually). Now that I've read this chapter very thoroughly I'm looking forward to taking another whack at the code.

I can use object seams to break a number of those dependencies, but we also depend on CAD and graphics libraries, which make sense to work around with link seams, as the book suggests. I can't think of an application for preprocessor seams at the moment, but wouldn't be surprised to find some in the code. Does anyone have a good story about using a preprocessor seam?

Chapter 5: Tools •

Ken Duffill, 11 July 2011

This chapter, at ten pages, is the longest chapter we have seen so far; though I don't regard ten pages as a long chapter.

Its purpose is to introduce the reader to some of the 'currently available' tools and the roles that they can play in legacy code work.

My edition was published in 2005, and so is already six years out of date, I do not think that there is a later edition, but in our industry life moves on at a very rapid pace so I suspect that the list of currently available tools would by now be considerably longer.

The author starts with a discussion on, and brief history of, automated refactoring. Without repeating him, it is sufficient to say that it is important to select an automated refactoring tool that does not change the behaviour of the code during refactoring (and verify any such claims that the tool makes with your own tests). If you have an automated refactoring tool that works correctly a lot of time can be saved. However, a lot of time can be lost if you use a refactoring tool that subtly changes the behaviour of your code.

Next the author briefly mentions the use of mock objects as a dependency breaking tool, and points the reader to www.mockobjects.com as a good place to find references to most of the freely available mock object libraries. Maybe in 2005 this was a good place to go, but Steve Freeman's blog (for that is what it is), while containing some interesting thoughts on mocking is mostly overtaken by discussions on the Growing Object Oriented Software Guided by Tests book that some of us have recently been reviewing.

Next there is a discussion of unit testing harnesses with a reasonable coverage of JUnit and CppUnitLite, a brief mention of NUnit and a suggestion that for other ports of the xUnit framework the reader should visit www.xprogramming.com and go to the downloads section. Again maybe this was a good idea in 2005 but at present I could find no downloads section and no mention of xUnit on that site (admittedly after only a cursory look).

The final section is about General Test Harnesses and covers FIT and Fitnesse. There is very little real information here on FIT. I would have liked a more in depth view such as we got on JUnit and CppUnitLite (maybe with an example or two). As for the description of Fitness as being essentially FIT hosted in a Wiki, well that is interesting but doesn't tell my why this might be a good idea.

All in all I was disappointed with this chapter, but on reflection had it been any larger or more detailed then I expect that it would only be even more out of date. The rest of the book which deals more with principles and practices will probably not age so badly.

Chapter 6: I Don't Have Much Time and I Have to Change It •

John Penney, 17 July 2011

After Ken drew the short-straw with a rather out-dated Chapter 5, chapter six puts us firmly back on course with a great mix of theory and practice. It's quite a long chapter, so apologies if I've missed any important points in this review --- please post back!

Michael starts with a practical acknowledgement that we seldom have the time to achieve perfection. It can be very hard to recognise the tipping point at which non-functional coding such as refactoring or adding tests costs more than it saves. He makes the point, however, that changes cluster in systems, so "if you are changing it today, chances are, you'll have a change close by pretty soon". If you accept this and invest more time in refactoring and adding tests now, then you and your team will discover that you are "revisiting better code".

But what if you're under real pressure and you can't see a way to get that legacy code under test soon enough? Do you have to give up on unit testing entirely? Michael suggests not, and offers 4 techniques by which you can add your new code and - at least - put tests around your new code.

Sprout Method

If your change "can be made as a single sequence of statements in one place in a method", then rather than add it inline to your legacy code, we can use test-driven development to drive us towards the creation of a new method on the legacy class that can contain our new code. You can then add a call to your new method from the legacy method.

So why *wouldn't* you do this? Well, maybe the new method looks a little lost by itself -- perhaps its code really does belong in the legacy method you've left behind?

Sprout Class

So what if you tried Sprout Method and discovered that you can't even instantiate the class to which your new method belongs due to dependencies? The author suggests simply creating a new class to host your sprouted method.

Hmm, but might that lead to lots of little fragmentary classes? Michael acknowledges that "Well, at this point, that is true". But if we choose our class names and concepts carefully, perhaps this could be the start of something new and better? (I've recently had first-hand experience of this, and am much less inclined now to be worried about the existence of small classes with single methods... just think of them as opportunities :-)

Of course, maybe Sprout Class was a better choice than Sprout Method in the first place anyway -- your new code really does sit more naturally in a new class than in a method on a legacy class.

Wrap Method

Again you have a new method `DoTheFunkyChicken()` to add, and know *when* you want to call it, right after `DoTaxReturn()`. But calling `DoTheFunkyChicken()` just doesn't sit comfortably in `DoTaxReturn()` -- different responsibilities, right? So we use Wrap Method and - through a set of steps carefully outlined by Michael - we end up with `DoTaxReturn()` calling `FillInTaxReturn()` and then calling our new (tested) method `DoTheFunkyChicken()`.

This leaves us with future opportunities to `DoTheFunkyChicken()` in other, unrelated, contexts.

But the "real downside" is that awkward renaming of `DoTaxReturn()` as `FillInTaxReturn()`. Michael agrees that "Wrap Method can lead to poor names".

Wrap Class

Just as sometimes we find ourselves Sprouting a Class instead of a Method, we can also Wrap a Class instead of a Method, and for the same reasons: either it's hard to instantiate or it belongs more naturally in another class.

One way to implement Wrap Class is using the Decorator Pattern -- a pattern I rather like, but which I haven't actually found that many applications for. In the example we create a `LoggingEmployee` that subclasses Employee: it's `pay()` method does the new stuff, then calls down to the `Employee::pay()` method in the base class. The pure Decorator Pattern (IIRC) relies on `Employee` actually implementing an `IEmployee` interface, but the principle is the same. Decorator is most appropriate if you've got a number of unrelated behaviours that you want to *sometimes* add to `Employee`.

An alternative that is "not so decorator-ish" is simply to create a new, appropriately named class whose sole responsibility is to do the new stuff (`logPayment()` in this example) and then to delegate to the `pay()` method on `Employee`.

Michael identifies two cases where he would go to the extent of using Wrap Class: firstly when the new behaviour just doesn't belong in the legacy class (separate responsibility) and secondly if the legacy class "has grown so large that I really can't stand to make it worse". You might feel uncomfortable with the latter case, but if this process is repeated then over time, using all these techniques, then a new healthier, better tested structure will emerge.

Conclusion

So we have 4 techniques for testing our new code, supported by examples which I find very clear and easy to understand. None of this is rocket science, but that just makes it more likely we'll consider using them! Furthermore, each has a clear set of suitability criteria, a step-by-step implementation guide plus a list of pros and cons to help us decide whether the technique is appropriate. Excellent!

But we must be careful: as Michael says, "unless you [also] cover the code that calls it, you aren't testing its use. Use caution."

Chapter 7: It takes forever to make a change •

Ann Napier, 25 July 2011

This chapter feels more like a summary and linking chapter than a source of much new content, designed to guide the reader to different parts of the book depending why it is taking so long to make a change. Is the time taken up trying to understand the system in order to make the change, or checking that the change is correct and doesn't break anything else.

Understanding

For any large system, it will take some time to understand how to make a change. The difference that proper maintenance gives is how easy it is to make the change, and how much you understand that area of the system afterwards.

"Systems that are broken up into small, well-named, understandable pieces enable faster work."

If this doesn't describe your system, then Chapter 16, I Don't Understand the Code Well Enough to Change It, and Chapter 17, My Application Has No Structure, should provide more guidance.

Lag Time

From earlier chapters we already know the problems of waiting overnight to get test results. In this chapter a story about the Mars Rover examines the problem of waiting just 14 minutes to see the results of any command.

Seeing results infrequently leads to the temptation to test a bunch of changes at once. If the tests fail, you don't know exactly what caused the problem.

For software, Michael advocates being able to build and test the code you are working on in 5-10 seconds. This leads to short steps of code, compile, test, code, allowing much greater concentration than a system where you have 5-10s to code then a minute to wait for everything to compile and be tested.

Sidenote:

I got a bit lost at the end of this section – Michael seems to be advocating 5-10 seconds coding, then compile-test, then more coding and so on. I haven't done much Test Driven Development, and the only area where I've been able to make a 5-10s change in the code and still have it compile at the end is GUI modification, when just moving components around to try and get it to look right.

Breaking Dependencies

Breaking dependencies is the main way to reduce compile-test time.

First step – get the class into a test harness, and get the methods to run. See Chapter 9, I Can't Get This Class into a Test Harness and Chapter 10, I Can't Run This Method in a Test Harness. Hopefully the code can now be compiled and tested quickly. If the methods are still slow due to lots of calculations, rather than

external resource calls, look at Chapter 22, I Need to Change a Monster Method and I Can't Write a Test for It.

If getting the class into a test harness at all is proving too difficult, look at Chapter 12, I Need to Make Many Changes in One Area. Do I Have to Break Dependencies for All the Classes Involved?

Build Dependencies

Once you have the class running in a test harness, you can try to speed up compile times further by breaking dependencies. Using Extract Interface or Extract Implementer, you can unpick seams and replace them with velcro, poppers or a zip. This increases complexity and rebuild time for everything but reduces build time when only compiling classes that have changed. The aim is to get small groups of classes that are easy to work on.

The book goes through an example of this, and then gives the dependency inversion principle. A somewhat cut version of this is “It is better to depend on interfaces or abstract classes than it is to depend on concrete classes. When you depend on less volatile things, you minimize this change that particular changes will trigger massive recompilation.”

Conclusion

We now have a lot of resources to use when change is slow, and a guide through how to break down and organise classes and interfaces. There's also a goal – under 10 seconds to compile and run tests on any section of code.

Chapter 8: How do I add a feature? •

David Pol, 1 August 2011

Test-Driven Development (TDD)

Michael states here that he considers TDD to be the most powerful feature-addition technique that he knows and describes its well-known cycle:

1. Write a failing test case.
2. Get it to compile.
3. Make it pass.
4. Remove duplication.
5. Repeat.

He illustrates this process with an example taken from a financial application that needs a class to compute the first statistical moment about a point. First of all, he writes a failing test case that makes the related method to always return NaN. Then he makes it pass by writing the proper algorithm (noting that the steps taken to get a test compiling are usually much smaller). When the test passes, he checks if there is any duplication to remove. There isn't any, so he proceeds with extending the new method to take into account the possibility of it being called with no elements in the instrument calculator. A new test is added to trigger that condition, which fails initially because an `ArithmeticException` is thrown when dividing by zero in `firstMomentAbout()` instead of the desired `InvalidBasisException`. Once the method's declaration and implementation are both modified so that it throws an exception of that type, all is fine and he can go on (again, there is no duplication here).

At this point, Michael proceeds to write a new method to compute the second statistical moment about a point. The process he follows is basically identical to that for `firstMomentAbout()`, but it turns out that the implementation for the second moment is actually only a slight variation of the implementation for the first moment. The idea of generalizing the method to accept an arbitrary value for the statistic moment is put on hold for the moment (because it is a burden for the caller and also something we don't want to allow). So, without further delay, he copies the code for `firstMomentAbout()`, renames it and changes the single line that needs to be different in order for the test to pass. Now that both tests pass, it's time to remove duplication. The process goes as follows: first, he extracts the body of `secondMomentAbout()` to a new method called `nthMomentAbout()` that takes a parameter `N` and, finally, he replaces the code for `firstMomentAbout()` and `secondMomentAbout()` with calls to `nthMomentAbout()` with appropriate arguments. I think this is general good advice, specially with the confidence provided by having tests in-place: when writing new code, I try not to tackle genericity first, but arrive to it naturally through refactoring (if it's needed at all).

Michael ends this section of the chapter noting how valuable the separation between writing code and refactoring that TDD provides is when working with legacy code. Based on this, he proposes a revised TDD cycle for legacy code:

0- Get the class you want to change under test.
1- Write a failing test case.
2- Get it to compile.
3- Make it pass (trying not to change code as you do this).
4- Remove duplication.
5- Repeat.

Which seems sensible enough to me.

Programming by Difference

In this section, Michael describes a technique known as programming by difference that makes it possible to introduce features to a class without modifying it directly by means of inheritance. He exemplifies it with a `MailForwarder` class from a program that manages mailing lists. The existing implementation has a method `getFromAddress()` that strips out the "from" address of a received mail message and returns it so that it can be used as the "from" address of the message that is forwarded to list recipients. He wants to support anonymous mailing lists, where the "from" addresses of posters are set to a specific e-mail address based upon the value of the domain. In order to add this new feature, he subclasses `MessageForwarder` and creates a new class `AnonymousMessageForwarder` that overrides the `getFromAddress()` method. This makes the test pass, but it isn't an ideal solution in the long term to subclass `MessageForwarder` to change just its "from" address.

Then, Michael comments on the big problem of making excessive use of inheritance: how to combine several features when they are distributed amongst many different subclasses. He describes a way of implementing the anonymous forwarding without inheritance: by using a configuration option passed to the constructor of the class. Which is cleaner, but may potentially break the Single Responsibility Principle [1]. He then explains an even better way to proceed when there are many configuration properties that consists on abstracting them into a `MailingConfiguration` class. The class starts as a simple wrapper over the properties collection we had before, but ends up supporting high-level functionality like `getFromAddress()` --to support the anonymous lists feature-- and `buildRecipientList()` --to support the off-list recipients feature--. The class name is also changed to `MailingList` in order to better reflect its purpose.

The important thing to note here is that programming by difference is a useful technique that allows adding features quickly and that having the code covered by tests enables rapid transition to better designs as the need for them arises. And that to do that well, we have to be careful with not breaking fundamental object-oriented principles like the Single Responsibility Principle (which we just covered) and the Liskov Substitution Principle [2].

The practical description of this second principle is the final part of the chapter. LSP says that objects of subclasses should be substitutable for objects of their superclasses throughout the code. The practical implication is that clients of a class should be able to use objects of a subclass without having to know that they are in

fact objects of a subclass (so violations of the principle in a class may happen or not depending on its users and their expectations). This was not taken care of before, when overriding the `getFromAddress()` method in `AnonymousMessageForwarder`. A better solution, Michael proposes, i to use normalized hierarchies, where no class has more than one implementation of a method.

[1] http://en.wikipedia.org/wiki/Single_responsibility_principle
[2] http://en.wikipedia.org/wiki/Liskov_substitution_principle

I liked this chapter. Although it covers well-known ground, I find it to be a great summary of why tests are useful when writing new code and how they facilitate the refactoring phase.

Chapter 9, I Can't Get This Class into a Test Harness •

Matthew Jones, 8 August 2011

We start off with an understatement: "this is the hard one". It certainly is. It is the hurdle at which most people immediately fall. This large chapter details seven common problems and analyses some of their solutions. I had been looking forwards to this one because I utter the title most days! It did not disappoint...

The Case of the Irritating Parameter

To test a class you have to create an instance in your test harness. The best way to go about this is simply to try it and let the compiler tell you what's missing. ("Lean on the compiler" is a common, powerful, and perfectly valid, tool.) What is often missing is a parameter that is itself an object that is hard or even impossible to create. The most common cause of irritation is that the parameter is a heavyweight, complex object, and/or one that has widespread dependencies. What we want to do is to fake the object and thereby avoid the effort of creating the real thing. To enable this we should consider applying "Extract Interface" to the object's class. If this technique can be applied, we can create a fake implementation of the interface and move on.

At this point we are asked to think about how strange the fake object concept is. But we are working in the test code domain, not the production code domain, so the rules are different. Writing a class that does nothing useful in and of itself might seem odd, but its utility is in enabling simple tests (or any tests at all) to be written.

We move on to a second, less irritating parameter: one that turns out not to be needed at all in the code being tested. (Maybe it is used in a different method, or a subclass, that we're not interested in right now). In this case why not pass null? Again, this might seem counterintuitive: a lot of production code is littered with defensive checks for exactly this case. But remember we are writing test code, not production code, and think more about how much easier the tests are to write if we use this technique.

At this point a sidebar goes into more detail about "Pass Null". In some languages the run time will catch dereferencing of a null pointer. This can be very useful if your tests do turn out to need the nulled parameter. However, in languages such as C and C++ it can be very risky, and even counterproductive, so proceed with caution.

I would add two points here

1. If things like null pointers are a problem it can be very useful to run your tests in a different environment from the production system: an environment that provides better run time support for your problem areas.
2. If you are passing null, use a self-documenting local variable:

```
CreditMaster dontNeedACreditMaster = null;
CreditValidator validator = new CreditValidator (connection, dontNeedACreditMaster, "a");
```

One last area of discussion about passing null is the Null Object Pattern. This replaces the null pointer with a valid instance of a Null Object. Refer to [Null] for more details.

If our irritating parameter is easy to construct, but irritating to use, we can use "Subclass and Override Method" to override the problematic methods and create a more benign subclass to use in test code only. But we must make sure we don't alter the fundamental behaviour that we want to test.

The Case of the Hidden Dependency.

Some classes are easy to instantiate but hide dependencies within. The example given is a constructor that new()s a heavyweight application object, then proceeds to manipulate it. We not only have a nasty dependency on some other part of the system, but we also have a problem sensing our effects on this object. Inability to sense implies inability to test effectively. So we need to introduce a fake somehow.

The first approach is to "Parameterize Constructor". If we pass the object in, rather than let the constructor create it, we can "Extract Interface", then provide a fake implementation. To hide the extra parameter change from the rest of the system, we can "Preserve Signatures": provide an overloaded constructor with the original signature that creates the extra parameter locally, then calls the new constructor. This works well if the object in question has no construction dependencies.

Other approaches deal more directly with the internals of the class: "Extract and Override Getter", "Extract and Override Factory Method" and "Supersede Instance Variable".

The Case of the Construction Blob

Some constructors create a lot of other objects, or access lots of globals. To attack such a class with "Parameterize Constructor" could lead to a large parameter list. An alternative is "Extract and Override Factory Method" but this only works in languages that allow calls to virtual functions from constructors. Even if the compiler allows it, this can be considered bad practice.

If the problem is simply getting to one of the internally created objects, to sense the effects of a test, then "Supersede Instance Variable" can be used to simply replace the original object with a test fake. However, be very careful about leaking the original object, or objects it refers to. In general we have to be very aware of object scope and lifetime. Even with these problems, "Supersede Instance Variable" can be useful; and sometimes in C++, the only option.

The Case of the Irritating Global Dependency

Global variables are one of the hardest dependency problems to deal with when testing. Chief among global variables is the singleton. We have quite a bit of discussion about the justification and pitfalls, but I shall not summarise that since I'm sure that being ACCU members, we already know this! Refer to [GoF] for more details, or Google "Singleton".

With respect to testing, it is the inability to control the lifetime of a singleton (or other form of global) that causes the most trouble. Tests can interfere with each other because state is preserved globally.

Two simple approaches are presented: the first is "Introduce Static Setter" whereby we can easily replace the singleton instance with a fresh one, and possibly even pre-set some state. The second is to add a static method "`resetForTesting()`" which is essentially a special case of the static setter that simply forgets the current instance, leading to a new one being created the next time the singleton is accessed. Relaxing access protection of the singleton instance, via public setters, might seem to be a backwards move (breaking encapsulation etc.). But remember that we are trying to add tests to improve other areas, and the price is almost always worth paying.

An additional technique is to "Subclass and Override Method" on the singleton class. Combined with a static setter this allows us to inject a fake instance of the singleton's base class.

We could choose to solve the source of the global dependency problem, but this requires far more effort. We are presented with a list of refactorings, and each has its downsides. If the effort is great, we must ask ourselves the bigger question: why? Why do we have hundreds of references to a global variable, scattered throughout the code? The answer will usually indicate some fundamental design problem such as insufficient layering or separation of responsibilities. These are dealt with in later chapters.

The Case of the Horrible Include Dependencies

In languages such as C++, where source files can include other source files, we can end up with "small files that end up transitively including tens of thousands of lines of code". I like that phrase. This leads to slow builds, and classes that are a nightmare to bring into a test harness.

If we take the class at face value, we can lean on the compiler (and linker) and incrementally include more and more of the system until it compiles cleanly. As we do this, we should examine each dependency a consider whether it is really necessary. (I have done this myself, and it is a very powerful technique. Sometimes just 10 minutes effort can radically simplify a module.)

If we can build our tests in a separate executable, we can stub dependencies by providing empty or minimal implementations that satisfy the compiler and linker, but do little else.

Although we are using what amounts to pre-processing and link seams, we are not changing the shape of the code, only making it easier to test. In fact we are more likely to make more problems in the future, because our duplicate definitions (the stubs) have to be maintained as the code they replace changes.

The Case of the Onion Parameter

This is the problem of needing to have many layers of objects created before we can instantiate our object under test. At some point in the hierarchy, we have to apply "Extract Interface" (or its opposite, "Extract Implementer") so that we can either pass a null object, a fake, or a very simplified implementation. We might even be able to simply pass null.

Although we might get caught up in defining convoluted class hierarchies, remember we are in the test code domain, not writing production code. In essence, "in any language where we can create interfaces [...] we can systematically use them to break dependencies".

The Case of the Aliased Parameter

I'm afraid I didn't understand what 'aliased' referred to here. The start of this section seemed to hark back to the others and offer nothing new. I have probably missed the point!

One nice point to take away is a refinement of "Subclass and Override Method": to create a fake class "on the fly", i.e. in the local scope of the test method. I first saw this demonstrated recently, since it is the style of testing heavily promoted in [GOOS]

We can run into trouble in poorly designed classes where it is not possible to isolate undesirable behaviour with "Subclass and Override Method". In this case we must apply other refactorings first. This subject is covered in a later chapter.

[Null] http://en.wikipedia.org/wiki/Null_Object_pattern
[GoF] http://en.wikipedia.org/wiki/Design_Patterns
[GOOS] http://www.growing-object-oriented-software.com/

Chapter 10: I can't get this method into a test harness •

Tim Barrass, 16 August 2011

So, after chapter 9 we can access our class in a test harness. The next chapter deals with getting the methods we're interested in under test. Here we might run into:

- complicated logic buried in private methods
- invocation awkwardness (like construction awkwardness in c9)
- side effects!
- unclear purpose requiring some sensing

The case of the hidden method

[Test through public methods; make method public; or make method protected, subclass and override.]

Sometimes you meet a public method that has hidden a significant chunk of logic in a private method. We might want to refactor that method to simplify it, or make it more generally useful. First, we need it under test. Here Michael suggests a multilayered approach.

The first thing to do is test through the public methods – we should do that when we can. Sometimes, however, that's awkward or not possible, and we want to test the currently-private method directly.

Our next tactic is to make the method public. If that worries you, the class is doing too much, and you should think about extracting behaviour to a new class.

Before refactoring to a new class we need to characterize the method's behaviour. Michael suggests *subclass and override*, after first elevating the method to protected from private. Subsequently we can get the method under test and have a basis for a true refactoring to a new class.

Each tactic here results in a more-public method. Worried about breaking encapsulation? Michael asserts that this is a fair exchange on the way to better code. He also gives reflection short shrift, asserting that it just delays the cost of paying for a bad, hard to test code base.

The case of the 'helpful' language feature

[Extract interface and work to that, if the code is ours. Wrap in our own interface to heal round code that isn't ours.]

Ah, sealed classes. Once, we were told it was a good idea to seal by default; but no more. Unfortunately, that leaves us (from experience) with a lot of unnecessarily sealed classes that are hard to get under test. In this case, Michael threw me a loop: I expected the method under test to be part of a sealed class. Instead, his example is about testing a method that *uses* impossible-to-construct sealed classes.

His example shows a method that uses a list of `HttpPostedFile`, which is sealed, has a private constructor and is provided by a third party. How are we to pass in

instances of this class to the method in tests? We can't *subclass and override*; neither can we *extract interface* or *extract implementer*.

Instead, Michael indicates that we can use *adapt parameter* to pass in test instances to the existing method. We adapt by *skin and wrapping the api* – creating our own proxy wrapper structure (interface, fake class, proxy for real class) that we bury `HttpPostedFile` in.

We can then *lean on the compiler* we can intervene in our prod code, process the objects being passed in (`HttpPostedFile`), converting them into our new wrapping structure and hand them over to the method we're interested in testing.

The case of the undesirable side effect

[Decompose into distinct behaviours and extract method, then subclass and override.]

"Oh, while I'm iterating this list to do load market data, I may as well update the GUI with the count as well. It'll save time later!"

Where to start? Get a good handle on what the method is actually doing, decompose each independent item and *extract method* to simplify. Michael presents an example that draws out the *command/query separation* – a method should either be a command, and alter state without returning a result; or a query, returning a value without altering state. Methods with side effects – and methods that do too much generally – are likely to contain elements of each.

Once we've extracted the independent methods we can *sublcass and override* to test the code that's left behind. After that, we might look again at moving the extracted methods to new classes – but at least at this stage we should be able to start testing.

Chapter 10: Summary

Michael moves on from describing how to get a class under test, to dealing with a few of the issues we face in getting methods under test. Generally he seeks to isolate methods by using *subclass and override*, and most of the techniques here are ways of getting us to that point.

Chapter 11: I need to make a change •

James Byatt, 22 August 2011

So, we've got a good idea of what behaviour we need to change, and we know, roughly, the area of the code that needs to change to make that happen. Now that we're there though, we need to figure out exactly where (or, "around what") to put the tests.

Really, there are two things we need to test:
1. The existing behaviour (we need to 'characterise' it)
2. The new behaviour

Getting the first of these done is really what this chapter is interested in (I think). Can we perhaps rephrase the question to make this clearer? Are we actually asking:

"What should our characterisation tests look like?" (1)

Michael introduces the term "effect analysis" to start to deal with this process. This quote starts us off very nicely.

"for every functional change in software, there is an associated chain of effects"

Michael, inappropriately calmly, suggests we try to sketch out the effects of changing a given function (or class) first (without necessarily looking for call sites). This will give us some sense, at least, of the scale of the task we face. We can even use these effect sketches, later on, to help us see how to identify ways of simplifying the design.

For me, this is a little like doing some 'scratch refactoring'. We can quickly find flaws in the design of existing code that makes reasoning about effects difficult (mutable state, lack of command/query separation, poor encapsulation, etc). Unfortunately, at this point we are unable to fix these as we lack the necessary test coverage; we haven't really answered (1).

Michael moves on to an example involving a java class named InMemoryDirectory. The killer point, for me, hides within these lines:

"Fortunately, our application uses `InMemoryDirectory` in a very constrained way"

"The tests are just a description of how we use this [functionality]"

Just as importantly, we also have the following:

"Unfortunately, figuring out where to test isn't always that simple"

Defensibly though, the simple example gives us a reasonably general rule: our tests should describe the usage of the functionality (before we start changing it). I read

this as "We should use our knowledge of the call pattern for this code to drive the way we write our tests for it".

If we can cover the existing call patterns with tests, we'll have described the current behaviour of the code. One thing this chapter could perhaps emphasize more is that the existing behaviour may be *completely insane*. Unfortunately we still need to rigorously discover the full extent of the insanity before we can start making things better.

Let's take an extreme example (Perhaps others could share their stories of nightmarishly complicated effect propagation?). Imagine the code we want to change sits just underneath the interface of a public, shared, api (perhaps, worse, in a dll). We may not even know how many callers there are, or how far the value (and its effects) returned from our function propagates up the call stack.

Tracing the call pattern might be a lot harder (we may, *shudder*, have to communicate with our clients), but we can (and should) still do it. The data we get back from our investigations may make our lives considerably easier than attempting to cover the (possibly huge) theoretical set of interactions with the code. Indeed, with this knowledge, we may later on negotiate a different contract that more tightly expresses the desired usage pattern.

The example Michael gives doesn't deal with that level of complexity ("you would have gotten bored and closed the book" - I hope my example has not had this effect :-( ). I am immediately drawn to the neat, checklist like rules that can be applied in 'library-like' situations. Effect propagation is categorised as follows (parenthesised descriptions mine).

1. return values (usually OK)
2. modification of objects passed around as parameters (usually evil)
3. modification of global or static data that is used later (usually super evil, but you gotta do it somewhere)

We can probably forge our own list for where our function can grab input from, too. A brief think from me yields the following (certainly non-exhaustive) list:

1. function parameters
2. object state (includes parent's state, and all the 'reachable' state of members)
3. global state
4. out of process state (filesystem, sockets, etc). Could be combined with 3.

Theoretically, any function could access all of those things before returning a value, having tickled one or two parameters, some internal state, and some global state. ARGH! For sanity's sake, we should hope that our fellow programmers have not been so unkind as to leave us with systems that do such things. Michael mentions that more functional languages like Haskell and Scheme make it far harder to write code with hard to find effects. If only they were in more widespread use :-(. There is still hope though:

"...in OO languages, restricting effects can make testing much easier, and there aren't any hurdles to doing it"

I couldn't agree more! In my experience, even without language level support for it, writing in a functional style is a tremendous help, even if it is just by convention.

Summary:

The chapter ends with a discussion that concludes that

"Encapsulation isn't an end in itself"

It might have been more general to say: "our design should be driven by a desire to minimise the possible effect chain". Certainly, for me, that's the subtext of this whole chapter (which fits with the subtext of the whole book, i.e "please stop doing these horrible things in your code" - possibly this is me projecting my own feelings onto the author, however). The subsection on "simplifying effect sketches" really brings this through - simplicity is our goal (as soon as we're sure we haven't broken things).

And finally...

I want to discuss the heuristic given on places to look for effects. I'm really interested as to whether we can improve software based on checklists (in the same way that simple checklists can aid something as complicated as surgery. Type "checklists save lives" into your preferred search engine, and you'll hopefully see what I mean). Lint like tools take us a long way down this road, but can we go further, and talk about 'softer' practices in terms of checklists, too?

As a tiny exercise: can we (collectively) expand and annotate this checklist, to provide a reasonable set of places for a wary programmer to check, so that they may have a little more confidence they aren't about to kill the patient? Do we need different lists for different languages (or are there general principles that we can leave some language specific leeway in)?

(N.B I've taken the rather dangerous liberty of rewording these, and reordering them a little)

1. identify a method that will change
2. if the method has a return parameter, look at its callers
   - if the method modifies any values, trace the usage of the values
   - don't forget about super or sub classes here (or friends, in C++.
     Are there other, more evil ways that people could be getting at this state?)
   - you'll have to trace callers of other functions that use these values too
3. if the method has mutable parameters, trace their usage from the point that they are mutated
4. if global state is accessed or mutated during your function, find
   - all the places where that global state is written
   - all the places where that global state is read
   - the person who wrote it, to demand satisfaction
5. What about intra-process state?

Chapter 12: I Need to make many changes in one area. Do I have to break dependencies for all the classes involved? •

Chris O'Dell, 29 August 2011

Summary

This chapter builds upon the 'Effect Sketches' described in the previous chapter as a method of finding 'Pinch Points' to allow the developer to add higher level 'covering tests' in a scenario where it is impractical to break all dependencies necessary for Unit Tests to be added. These 'covering tests' work with the code directly affected by where you wish to make your change and effectively pin down the current behaviour allowing you to make changes knowing that these these will alert you of any unexpected changes in behaviour. These tests act as a support while you refactor the code they cover to allow for unit tests and in time should be killed off.

Review

In the scenario where you need to add a new feature, or make a change to a small number of closely related classes, it may be more practical to go "one level back" and find a place where with can write tests for several changes at once, thereby providing "cover" for more refactoring in that area as we will have pinned down the behaviour with tests.

In a side note, Michael reminds us that while higher level tests are an important tool, that they should not be a substitute for unit tests and instead should be a first step toward getting unit tests in place, i.e. we can refactor the code to break dependencies with the covering tests ensuring the behaviour is not modified.

Interception Points

An 'interception point' is a point in the program where you can detect the effects of a particular change. Michael then lists a simple example and uses his 'effect sketches' as described in chapter 11 to map out where a change to the 'getValue' method of the Invoice class affects other related classes. In another side note he states that it is generally a good idea to pick an interception point as close as possible to the required change - each step away from this is akin to a step in a logical argument and makes it harder to know that you've got it right. He suggests that once your tests written to temporarily alter the code under test to prove the tests capture the changed behaviour. This is a step I've done many times myself when adding tests to existing code.

In the cases when a change is on a public method on the class we're changing, using this as an interception point might not be the best choice. Michael suggests expanding the effect sketch and to look for a higher level interception point. Michael states that the benefits of this are twofold: we would have less dependency breaking to do and we'd have a bigger chunk in the vise of testing, meaning more cover for refactoring. In the example we discover that there is a single point through which all the changes would be detectable, Michael names

this a 'pinch point' - a narrowing in an effect sketch where tests against a couple of methods can detect changes in many methods.

Judging Design with Pinch Points

Micael states that pinch points are really encapsulation boundaries - where all of the effects of a large piece of code are funnelled. This knowledge can be used to carve out sets of classes within a program, write 'characterisation tests' (to be described in the next chapter) around them and start making refactorings. In a rather large side note, Michael explains this with an example to show how the effect sketches can highlight boundaries suggesting that new classes could be extracted out.

Pinch Point Traps

As stated previously in a side note - 'covering tests' such as pinch point tests are merely a stepping stone to get your code into finer grained unit tests. The pinch point test will be bulky, having to instantiate many classes and setup scenarios through many steps and this is cumbersome to run and difficult to maintain, so it is import that they are used for support but are killed over time.

Chapter 13: I need to make a change, but I don't know that tests to write •

Joes Staal, 5 September 2011

This chapter starts with some observations on how tests form a safety net for making changes and how tests specify the behaviour of our software. In a test driven design environment we know beforehand what the tests need to be, but how do we characterise the behaviour of legacy software? That is what this chapter is about.

The first part of the chapter explains how to write tests for finding out the behaviour of the software. Assuming you can instantiate the objects you want to investigate you call in your tests methods on the object and test against improbable outcomes. The result is that your test fails and shows what the expected value is meant to be. After replacing the improbable outcome with the expected value, the test will pass. Although this technique may feel strange (especially when used to test-first development) it does the job of building the aforementioned safety net (and I have used this approach myself with good results).
There are a few caveats: first of all you are quite likely to detect bugs, i.e. you'll find that the software doesn't behave the way you'd expect from either documentation or intent. The question is what to do: fix the bug or leave it as it is? My feeling is that Michael leans towards the former, while I in first instance believe one should leave it as it is for the following reason: we are trying to characterise the behaviour of the software as it is, not as it should be. There is of course a danger of forgetting about the bug if it is not fixed and we leave the test to pass with the wrong result. My solution to this is to write another test with the expected values as we think they should be. This test will fail until we come back to it and fix the bug (in which case the original test will fail which is intended because after fixing the bug we can delete that one).
The other caveat is that these characterisation tests normally are not unit tests but more often integration tests. We want to find out how the software works before we can start refactoring and breaking dependencies. I think it is good to come back to these characterisation tests after refactoring the software under test and then refactor the characterisation tests into proper integration tests (probably extending the tests as we have broken dependencies).

The next bit of the chapter gives some tips on how to find out what a piece of software is actually doing. Since we are working top-down the testing is often at a high level. If the software is complex we may find out how specific classes or methods are working by using sensing objects (as discussed in Chapter 3). Other things to take into consideration is to detect which assumptions can cause the software to misbehave and to find extreme values for the input parameters. Finally, try to find and test the invariants of a class. All these techniques will probably need some or extensive refactoring, for which tests are needed. I don't know how to break that vicious circle, although I think one can try to isolate specific dependencies and test those before moving on to larger refactorings and really break things down.

The last part of the chapter discusses how to test that the changes we want to make are actually correctly implemented. Make sure to write tests that follow the different paths through the code (the example given shows an if-else branch). And finally, make sure that each conversion along the path is exercised.

Again, a very good chapter.

Chapter 14: Dependencies on Libraries Are Killing Me •

David Sykes, 12 September 2011

This chapter starts out by noting that libraries can be an excellent opportunity for time saving code reuse, but with that opportunity comes some serious dangers. It is very easy to allow dependancy on the library to become embedded in the source code, making it vulnerable to changes to the library. An example is given of the license fees for a library increasing to make the product unviable. The chapter notes that every hard coded use of a library class is a lost opportunity for a seam, and that while many libraries are written in a way that facilitates tests, many are not. Libraries with classes that are final or closed, or with interfaces that are not virtual, can be hard to mock out, the suggested solution to this is a thin wrapper around the classes.

Finally the chapter suggests there is a tension between enforcing good design and testablility, a common example being the once dilemma; an assumption by a library that there is only one of something. This can make it difficult to use fake objects. If none of the dependancy breaking techniques prove appropriate then wrapping the singleton might be the only choice remaining.

The chapter finishes by mentioning the 'restricted override dilemma', where methods are made non virtual, which hinders sensing and separation. I am not able to distinguish this point from the earlier point about non virtual interfaces. The point is made, however, that restricting access to the library contents from code that uses it can be seen as good practice, without realising that it hampers the testability of the customer code.

The chapter concludes that a coding convention can be as good as a restrictive language feature.

Chapter 15: My application is all API calls •

Joes Staal, 20 September 2011

The chapter starts by explaining that it looks like that third-party libraries don't need to be tested, especially when starting a project in which the third-party library is not used heavily. However, over time it may get used more, become more hidden so that changes in the software may have side effects through the third-party library, etc. And then we are suddenly dealing with legacy code.

We are given an example application written in Java that acts as mailing list server and forwards incoming messages to a list of email addresses stored in a text file.

The application relies heavily on Java classes, some of which are sealed, so that testing is problematic (the sealed classes can't be overridden, so refactoring to interfaces is difficult). Still, by redesigning the mailing list server and high lighting the responsibilities we are shown that some testing without direct dependencies on the lower level API can be done. A referral to Chapter 20 (This class is too big and I don't want it to get any bigger) is made and a tip is given to consider the code as one big object we want to break up.

We are given to options to consider in general when we are confronted with direct exposure to an API:

1. Skin and wrap the API
2. Responsibility based extraction

Option 1 is a good option when the API is small and not too complex (and gives us the chance to build interfaces, so no sealed classes). The big advantage is that all dependencies on third-party libraries are broken. Option 2 is more appropriate when the API is large and/or complex. It helps if you can rely on a refactoring tool to do the extraction safely.

Note: In the final example, the function `sendMessage` calls a private function `getSMTPSession` to get a `Transport` object:

```
Transport transport = getSMPTPSession().getTransport("smtp");
```

Wouldn't it have been better to have a function `getTransport(String protocol)`?

Chapter 16: I Don't Understand the Code Well Enough to Change It

• Timothy Wright, 26 September 2011

This chapter introduces techniques to help understand a code base. Many of these are familiar, Michael gives them names and discusses their benefits.

Notes/Sketching

When reading through code, take notes and some simple sketches. These notes will help you understand the code and generate more questions. The sketches don't need to be complex or UML, they are to help you now. They are not a part of the formal documentation.

Michael explains that you don't need to require developers to use this technique. Just use it when working as a team and they will quickly see the benefit.

I also use the IDE search tools to help me jump around the code and sketch out what is happening. I find the need to re-experience the code for review and understanding can never be replaced by documentation.

Listing Markup

In this technique, the code is printed out and notes are written on the printout. The book lists different markup:

Separating Responsibilities

Use a marker to group similar ideas together so you can see where they are in the code.

Understanding Method Structure

Use a number or some other mark to line up the blocks in a large method. Often indentation can make the code hard to read.

Extract Methods

Circle the code segments you would like to extract from a large method. You can keep track of its coupling count.

Understanding the Effects of a Change

Mark the code you would like to change. Then find and mark all the variables and methods that could be affected by your change. This could help you understand how your change will propagate through the code.

I used to print out code all the time. It would clutter my desk forcing me to clean every week or so to find writing space. But now I find that there is too much code. It is much easier to read the code on the screen and jump around using the IDE tools. I keep notes and thoughts on a separate piece of paper

Scratch Refactoring

Scratch refactoring is when you check out the code and just start refactoring without tests. Just moving around code to see what happens. Michael believes this

is a great way to understand the basics of the code and how it works. However, there are a few risks. We could make a mistake which leads us down the wrong rabbit hole. We make a wrong assumption about the system. We could also get attached to our scratch refactoring, believing that it is the way to go. But we should let the real refactoring with real tests drive itself with this bias.

Delete Unused Code

If the code is not used or commented out then remove it. Why keep something that could be confusing? Keep the code simple. When developers first start working for me this is the biggest change for them. They have been conditioned to add comments and commenting out old code. (You never know when you might need it again.) But I think it just distracts from what the code does now and that is what is important. If there is deleted code you really need back then you can always get from the version-control system.

Chapter 17: My Application Has No Structure •

Tim Penhey, 5 October 2011

This chapter is very dear to my heart, having worked on a number of systems that had no apparent structure.

Almost all systems start with some overlying architecture, but over time, often with deadline pressure, this structure decays or is lost.

Michael gives three reasons behind the lack of awareness of the underlying structure:

- the systems is so complex it takes a long time to understand the big picture
- the system is so complex that there is no big picture
- the team is so reactive they lose sight of the big picture

Michael points to a common solution (or proposed solution) that is to have an architect whose job it is to maintain the big picture for the team. He also points out that this really only works when the architect is working with the team on a day to day basis. Otherwise it is often the case the big picture of the architect no longer matches the big picture viewed by the developers.

This chapter gives three ways to help describe the big picture and gain a better understanding of it.

Telling the story of the system

Start with very simple outline, as if explaining the system to someone who knows nothing about it. Iteratively go into more detail. When you have a change to make, go with one that better fits the simpler descriptions rather than adding more edge cases.

Naked CRC

This approach uses blank CRC cards as entities in the story of the system. Cards represent objects, composed objects or collections are overlapped cards. Lots of pointing and hand waving ensues

Conversation Scrutiny

Listen to the conversations you have about your system. Developers use a certain vocabulary when describing how the system works. Does the vocabulary used match entities and relationships in the system? If not, perhaps they should.

All in all a very short chapter with some good ideas. Personally I have used the "telling the story of the system" and "conversation scrutiny" before, often to a good end. The "naked crc" method to me seems a bit too hand wavey.

Chapter 18: My test code is in the way •

Ian Bruntlett, 10 October 2011

This is a brief chapter, consisting of 2 parts.

The first part, "Class Naming Conventions" gives a very good suggestion for naming test objects and fake objects.

The second part, "Test Location", discusses where to put production code and test code - in different directories? in the same file? And it warns that whatever you choose, if you force people to jump through hoops to write tests then they will just stop writing tests. Personally I have always had tests built into application source and used the preprocessor to strip out test code when building a release executable.

Chapter 19: My Project Is Not Object-Oriented. How Do I Make Safe Changes? • Martin Moene, 17 October 2011

We can make safe changes in any language, but some languages are, err, make change easier than others. In procedural code often the easiest thing is to do is think really hard, patch the system and hope that your changes were right. Provocative?

Given there are relatively few seams to break dependencies, it may help to create higher level 'covering tests' to get feedback while developing (See Chapter 12). Key here are pinch point, link seam and preprocessing seam.

An Easy Case

A test for a function that calls no other functions and that changes a variable passed to it that itself is easily created may be not difficult to write at all.

A Hard Case

In a less easy situation, a function to test does call another function that we'd rather not have called. One way to intercept the call is to substitute the function with one of our own that we supply via a library thus applying a link seam. However if we need several variations of the substituted function e.g. for sensing, this method quickly becomes tedious.

In C, the macro preprocessor enables a more flexible approach. In the file with the code of the function to test we use #define to substitute our own implementation of the function to fake and we terminate the file with main() that contains (or calls functions with) the test code. A preprocessing define controls the presence of these additions for testing.

To stay closer to the original code, we can move the above additions into two include files, with a further improvement to place all additional testing code in a single include file.

Adding New Behavior

In procedural legacy code prefer to add a new function over directly adding the code to an existing function. Doing so we can use Test-Driven Development that may help steer the design in a good direction.

One example shows how a function that performs some computational work with no external dependencies can be formulated so that it can be tested and integrated with the rest of the code. Key is to limit the behaviour of the function to what you can or want to test.

Another example shows how behaviour of a function that contains a sequence of many external calls can be made testable by applying object-oriented C. Here the function pointers provide the seam. (...Or write the function as best as you can if you're not using C ;)

Taking Advantage of Object Orientation

If you have a program written in The C Programming Language, you can try to compile it as a C++ program and further transform it to break dependencies. In the example given, the dependency on a function is replaced with an object of which we can control the behaviour by providing it via the object's constructor (Encapsulate Global Reference(339), Parameterize Constructor(379)): A wedge to break dependencies and move forward.

It's All Object Oriented

Now this section puts a smile on my face. Michael shows that procedural programs are in effect object-oriented, be it that many only contain one object. If you're working in a procedural language that has object-oriented extensions, it allows you to subdivide the system in ways that make it easier to work with. It also lets you move incrementally towards better object design.

This is an interesting chapter on non-object oriented code in a book about legacy code. If you like it the other way round, there's "Adding Tests to Legacy Code" in James W. Grenning's "Test-Driven Development for Embedded C", Addison–Wesley, 2011.

Chapter 20: The Class Is Too Big and I Don't Want It To Get Any Bigger • Richard Barrett, 24 October 2011

Classes get too big; it's a fact of life, or, at least in my experience, it takes a determined effort to keep them small. And its not just legacy stuff, new code bloats, too.

Why are large classes a problem?
- Understanding. Large classes can lead to confusion. What do you have to change in all this code?
- Side-effects. What other code will any changes to this method affect?
- Scheduling. Really large classes might have several engineers working on them with the consequent issues
- Testing. Large classes are a pain to test.

The introductory section sets the scene:-

Some of the techniques that we've met earlier can be used to reduce the size of over-large classes: Sprout Class and Sprout Method are our main tools. Sprout Class stops our class getting any bigger; Sprout Method aids us in understanding something else that the class does.

Refactoring, by breaking down the over-large class into smaller ones is the key remedy. We use the SRP to identify what are the responsibilities of the smaller classes.

We can look at the following areas to try and identify those responsibilities:
- the name of the class (hopefully, it's not a ...Manager!)
- the class's fields and how the class's methods use them
- method grouping. This is a good idea: try to group the methods into their individual responsibilities and create new classes from those responsibilities.

But, there's always the danger of over-engineering the solution: the bulk of this chapter looks at how we can identify responsibilities and then move forward to create what Michael calls “focussed responsibilities”.

Seeing Responsibilities

The author points out that the ability to see responsibilities isn't restricted to working with legacy code, it is also a key design skill and takes practise. He presents seven of heuristics that can be used to uncover responsibilities (therefore, these will be useful in the design sphere as well):-

Group Methods.

Here, you're looking for commonality of responsibilities - especially those that appear to be outside of the main class responsibility. I've never done exactly this as a group activity, but it sounds like it would work as such

Look at Hidden Methods.

This is an interesting thought: if you see a private method that looks like it should be tested, then it probably shouldn't be private and also makes a good candidate for splitting out into another class. This is well illustrated with an example.

Look for Decisions Than Can Change.

I like this one. Do some method refactoring - whilst looking at the assumptions that have been made when implementing the method. Can methods be extracted that present a higher-level functionality? Method grouping may lead to a better class extraction.

Look for Internal Relationships.

Look for relationships between methods and variables. If a subset of the methods are the only ones that uses certain variables, then this 'lumping' can imply that these methods and variables are candidates for the extraction of a class.

Michael shows us how to use feature sketches to illustrate a way to discover the associations between variables and methods. He runs through this technique with an example, and shows how clustering is discovered. Then what would be the effect of extracting a cluster of methods and variables into a separate class? Feature sketches are a great idea; I've done something similar myself but reading this section makes me think that I should use this technique more often.

Look for the Primary Responsibility.

This reminded me of the “what is our mission?” question beloved of management. Can we identify what a class does in a single sentence? We're looking for a statement that supports the Single Responsibility Principle. If a class has features that aren't really part of its main responsibility, then could they be hived-off to new classes? Single responsibility violations can occur at the interface and at the implementation level. At the implementation level, we're looking for extraction of classes that themselves have a single responsibility. At the interface level, we looking for segregation of interfaces so that clients are only dependent on the interfaces that they need. (As Michael earlier pointed out, this is also a key design skill.)

When All Else Fails, Do Some Scratch Refactoring.

We've met scratch refactoring before in Chapter 16. It's another technique that we can use to discover responsibilities in a class.

Focus on the Current Work.

I think that this is another way of saying: do what you need to do when making changes. If you've learnt that there are many more distinct responsibilities that could be changed, don't necessarily do all of them but try to remember what you've learnt. What do others get from this paragraph?

Other Techniques

Apart from the seven heuristics listed, what else can we do? Learn from others to improve our skills. Read code: look how others name classes and methods. Read books about design patterns. (We just need to find/prioritise the time....)

Moving Forward

OK. So we've used the heuristics and have identified the separate responsibilities of your bloated class. What do we do next?

Strategy

Rather than taking time-out to split our big class (or classes) down into new single-responsibility a complete set of single-responsibility classes, Michael suggests the better approach is to ensure that the team understands the responsibilities of the class and split out the other classes on an as-need basis.

Tactics

Realistically, we'll start extracting at the implementation level. That way the clients of our big class won't need to change. So, we should identify all of the methods and instance variables that we'll have to move - giving us a good idea of the methods that require tests. Implementing these tests on big classes in a test harness can be tricky. We can use the techniques previously identified in Chapters 9 and 10 to help here.

If we can get the tests in place, then Michael refers us to Martin Fowler's "Refactoring" book to perform the class extraction.

If we can't get the tests in place, then Michael presents a procedure for extracting the responsibility into a separate class. This involves extracting method bodies to renamed methods in the existing class and moving them along with instance variables to a separate area in the class declaration. Then moving the renamed methods and instance variables to the new 'responsibility' class and wiring it all up. We get advice on avoiding some OO gotchas.

After Extract Class

Michael warns us, rightly, against being overambitious with extracting classes. Again, he suggests moving towards a better design rather than plunging pell-mell into trying to do it all at one.

This is another great chapter. We've probably used some or all of the techniques that Michael provides at some point. It's good to read about the rationale for their use and to be warned about their pitfalls.

Yet more reason to keep "Working Effectively with Legacy Code" close to hand.

Chapter 21: I'm Changing the Same Code All Over the Place •

Nigel Evans, 31 October 2011

Michael opens this chapter by saying *This can be one of the most frustrating things in legacy systems…*. This sentiment is widely echoed in the literature: Hunt & Thomas [2000] raise *The Evils of (code) Duplication* as the primary principle in their book *The Pragmatic Programmer*; Fowler & Beck [1999] are equally forthright in the seminal book *Refactoring - Number one in the (bad code smells) stink parade is duplicated code*.

Unsurprisingly, Michael proposes refactoring as the solution to this problem. But he stresses the importance of not *over-thinking* the process. Removing duplication doesn't have to be as difficult as reengineering or *re-architecting*; it can be done in small incremental chunks, using a fairly mechanical process. But is the effort worth it? Michael promises us a *surprising* result. He then takes us on a journey through a worked example to illustrate the process.

Step Zero

The first thing we need is a set of tests that we'll run after each refactoring. We're sufficiently far through the book now that Michael assumes we know enough techniques to get these tests in place, and omits their description for brevity.

First Steps

This is where *over-thinking* needs to be avoided. Start by removing **small** pieces of duplication. The more small pieces you remove, the easier it becomes to see large areas of duplication.

Deciding Where to Start

The truth is, it doesn't make much difference…. But thinking about how you'd name new methods that result from your proposed refactoring can give you clues as to whether your refactoring will help to make more sense of the code, or not. Also, follow the *start small* heuristic.

Superclass

If you have two classes doing similar things at the same structural level, then you can often create a single superclass to replace duplicated code.

Methods

When two methods look roughly the same, try to isolate their differences and extract them to new methods. If you can make the remaining original methods exactly the same, then you can get rid of one of them.

Abstract Methods

If you've used a Superclass to reduce duplication, it can help to introduce abstract methods into the Superclass, with implementations in the subclasses that incorporate only the differences between the subclasses.

Generalisation

Where two methods in different classes perform the same operation, but on different sets of data, it may be possible to make the method more general and move it into a single superclass. The subclasses then just need to pass their own data sets to the generalised method to achieve the same functionality.

Renaming Classes

The result of pushing duplication upwards into Superclasses is likely to change the role or focus of the original subclasses. It's a good idea to review the class names and, if necessary, change them to ensure they are self consistent. Michael recommends avoiding abbreviations in names. If they are used consistently they may do no harm, but it's simpler not to use them.

Surprise!

Remember how we were promised a surprise? No, I didn't either until I reread this chapter to write this review. *The startling thing that you discover when you start removing duplication zealously is that (good) designs emerge*. The process of removing duplication across classes produces small, well focused methods. Each method does something unique. Michael refers to this property as *orthogonality*. I recognise it as *low coupling* - something we would hope to design into our code, but which here has emerged through the application of a fairly low-level, mechanical process. In other words, we get the benefits of improved design without having to think through how to achieve it!

Open/Closed Principle

Bertrand Meyer [Meyer 1988] articulated this principle that well implemented classes should be open to extension (through inheritance) but closed to modification. To change or add functionality we shouldn't need to modify existing class implementations, but we should be able to reuse them through inheritance. *When we remove duplication, our code... starts to fall in line with the Open/Closed Principle*.

References

[Fowler 1999] Fowler, M. *Refactoring*, Addison Wesley Longman Inc, 1999.
[Hunt 2000] Hunt, A & Thomas, D. *The Pragmatic Programmer*, Addison-Wesley, 2000.
[Meyer 1988] Meyer, B. *Object-Oriented Software Construction*, Prentice Hall, 1988.

Chapter 22: I Need to Change a Monster Method and I Can't Write Tests for It • Andrew McDonnell, 7 November 2011

Erratum: the first sentence on page 306 should be:

> "Chances are, you will, too. I skeletonize when I feel that the control will need to be refactored after it is clarified."

First off, you've got large methods (oh yes I do). Sometimes you can avoid having to refactor those methods for a specific change by using Sprout Method and Sprout Class, way back from chapter 6. But even if you can it's a bummer, because the large method is still there, ready to suck up time understanding things all over again next time you have to go do some work there.

At some point a method becomes so large and complex and unwieldy it's not even just large anymore. This chapter is about techniques for dealing with monsters, which can also be applied to your simple everyday large methods.

Varieties of monsters

Bulleted Methods:

There won't be a lot of deep indentation, just a sequence of code chunks. "Do A, now do B, now C, ...". Of course it's not quite so clean because there are probably temporary variables used throughout the sections (and they might be less temporary than they look).

Snarled Methods:

In the simple case, this is something dominated by a single large indented section. But more likely that section will in turn have different indented sections at different levels of indentation, which makes it that much harder to reason about.

Of course nothing prevents a method from being both. Regardless, the snarl makes it difficult to figure out how to write tests, since it's hard to tell what the behavior is even supposed to be. So how can we break behavior into chunks small enough to understand?

Tackling monsters with automated refactoring support

Specifically, a tool that extracts methods for you without introducing errors. I use Visual Assist X to do this in C++ and it definitely makes life better.

Michael notes that when you start working with this kind of tool, you should resist the temptation to make little manual embellishments along the way - reordering statements to make them more extractable, breaking up expressions - because these can't be checked for safety like the automated extraction. If your tool supports variable renaming you can use that (VAX does, for example), but if not stick to the automated refactorings for now. The idea is to perform only safe refactorings until you can get tests in place. I can attest that this temptation is really hard to resist.

The goals of these extractions should be:

1. Separate logic from awkward dependencies
2. Introduce seams to make it easier to get tests in place for more refactoring

Conspicuously absent in the list of goals is the first one I think of when I think refactoring: break things up into logical units. That's getting ahead of ourselves: first we need to be able to test so we can preform those more complicated refactorings safely.

Sometimes this work can seem like it's not getting you much, but as you extract logic into methods and give them names you learn more about the structure, how you can leverage it for testing, and where you want to go next (for example, back to another chapter's technique for the next step).

The manual refactoring challenge

There's a lot to go wrong when extracting a method by hand. However, once you can build that code in a test harness there are ways to move forward with more confidence.

Introduce sensing variable:

If the behavior you're looking at is hard to sense directly, you can add a sensing variable to the class, which is just a variable you'll set when you perform the action you want to sense. Your test then only has to look at the sensing variable instead of some downstream result of the behavior. After you've done your refactoring, you remove the sensing variable and the tests that work with it.

It's good to keep your sensing variables and tests around for a full refactoring session, not just a single refactoring, so you can easily back up and do earlier extractions differently if that's where the refactoring leads you.

Extract what you know:

Start small, with things you know you can extract safely. Small here means two or three (ok, no more than five) lines of code: a little chunk that's easy to name. In addition to line count you want to keep the coupling count (how many values go into and out of the method you extract, through its interface) low. The best coupling count, of course, is 0, which basically represents a command to the object to do something. As you name these things you can gain insight into what they're doing.

As you extract, if the coupling count is more than 0 consider using a sensing variable if it helps, and write a few tests for the extracted method. Watch out for parameter/return type errors in particular.

Michael suggests starting with the 0-count methods to get a better sense of the general structure and seeing where things stand then, rather than moving right in to more coupled sections. It may give you enough insight to start another refactoring approach. He also notes that even though a bulleted method can seem like an obvious candidate for finding 0-count methods in the chunks, you may be

thwarted by temporary variables. It pays to look for low-count methods within or across chunks too.

Gleaning dependencies:

This can be used when your method has some code that's secondary to the main purpose. In addition it's not too complicated, and it will be obvious if you break it. Presumably the primary behavior is more complicated and it's harder to detect if your refactorings break it. In this case you can write tests to cover the primary behavior, perhaps using a sensing variable. Then you can extract away the secondary behavior (only primary and secondary in that method? what luxury!) with confidence that you're not affecting the main part. This is especially useful when critical behavior is tangled up with other things you'd like to extract.

There are several definitions of glean:

1. Extract (information) from various sources:
   the information is gleaned from press clippings.
2. Collect gradually and bit by bit:
   objects gleaned from local markets.
3. Gather (leftover grain or other produce) after a harvest:
   the conditions of farm workers in the 1890s made gleaning essential.

I like to think Gleaning Dependencies takes it's name from the third definition: the stuff left over after the main body is under test is the gleaning to be gathered up and removed. Not to mention working with a monster method can really feel like this: http://en.wikipedia.org/wiki/The_Gleaners

Break out method object:

You may notice that you already have variables that would be perfect for sensing purposes, but they're local to the method you're working on. You wouldn't want to promote them to instance variables just for testing, of course. But by creating an object whose sole purpose is to do the work the method does, you can promote local variables at will without muddying the waters in the initial class. You create a class with a constructor that takes the parameters of the method and has a single method containing the logic from the monster method.

An advantage of this over a simple sensing variable is that you get to keep the tests your write for this new object: the sensing variable is also a part of the regular code.

Strategy

These are some directions to take when working on monster methods.

Skeletonize methods:

Extract each bit of logic (say, a condition or the body of the conditional) into methods. You're left with a skeleton, that is the control structure and delegations to other methods, but not the meat that does the work. This may help to reorganize the control logic.

Find sequences:

On the other hand, it may work better to extract a condition and its body together into a method. This can help notice common sequences of operations.

Skeletonize and Find Sequences can be complementary, even though they're at odds with one another at first glance, and both are applicable to any kind of monster. Although Michael tends towards Find Sequences in bulleted methods and Skeletonize for snarled ones.

Extract to current class first:

As you start extracting and naming, you may notice that the extracted methods don't really belong in the current class. For example, if the new method name includes the name of one of the variables it uses, that method may really belong in the class of that variable. Resist the temptation to move it immediately - stick to small changes that are easy to undo if you decide to choose a different refactoring direction. You can always move that method to a new class later.

Extract small pieces:

This is a general reminder of a recurring theme: start small. Even though it seems like 3 lines here or there won't make any difference in a 3000-line monster, eventually that helps you see the function differently and can offer ideas on how to move forward. This is easier and safer than trying to break something into big chunks right from the start.

This is interesting: my initial thought would be that a first-pass refactoring would involve, say, breaking down a 3000-line monster into a few 1000-line monsters that have a slightly more limited scope. But I can see now that first off, I'd almost certainly break something, and there's no way I could test the mini-monsters that come out of it.

Be prepared to redo extractions:

Since there are so many ways to break down a huge function, don't worry if you realize the best way to move forward is to undo some extractions and approach them differently. It's not wasted effort; the first extractions helped you see a better way.

I like how this chapter provides a number of simple rules and techniques for dealing with overwhelming methods. It makes getting them under test seem downright possible! The timing is perfect, because this week I plan to spend some time on a monster function in the code base I work on (both snarled and bulleted, by the way).

Chapter 23: How do I know that I'm not breaking anything •

Joes Staal, 17 November 2011

For me, the main message to take home from this chapter is: do one thing at a time.

Michael starts by drawing parallels between a mechanical machine and a computer program. The machine will break down through wear and tear, but a computer program will break down when we change it.

Hyperaware Editing

When editing code you can be reformatting, adding comments or make functional changes. Michael stresses again about tests and feedback. He presents an interesting idea on IDE's that run tests every time you press a key and calls it edit-triggering testing. Getting feedback increases your confidence (whether it is through tests or by pair programming (which shouldn't exclude writing and running tests)).

Singe-Goal Editing

It may look smart to be able to store the architecture of a system in your mind and knowing all the side effects if you change the system in one place. But it also looks highly improbable. It is better to focus on one task at the time and not to be seduced to 'fix' other problems on the way. Michael gives some ideas on how to avoid such behaviour e.g. by keeping a list of functions that need attention after the original task has been completed.

Preserve Signatures

Refactoring is in general a rather invasive way of editing and especially when dependencies need to be broken (and we have no tests available yet) we must be very conservative on making changes. Again, the focus is on doing one thing at the time and not doing too much. Break your dependency, but leave all the other refactoring until you have tests in place. Michael shows how he uses 'Preserve Signatures' to minimise the risk on getting parameters in the wrong order or with wrong types.

Lean on the Compiler

Changing types of variables or commenting out functions will give compilation errors so that the locations where changes need to be made become visible. One must be careful though. E.g., when commenting out a member function that overrides a concrete implementation in a base class, the code will cleanly compile and probably do the wrong thing.

Pair Programming

One of the few places where the book shows its age. Still, it is good observation that dependency breaking is like surgery and that doctors don't work alone in complex procedures. So, get that second pair of eyes to help you.

Chapter 24: We Feel Overwhelmed, It Isn't Going to Get Any Better (21 November 2011, John Penney)

Chapter 24: We Feel Overwhelmed, It Isn't Going to Get Any Better • John Penney, 21 November 2011

Part II of WEWLC concludes with a short essay that reflects upon how easy it is to become dispirited as a programmer, but also how it is possible to enjoy and relish working with even the most deep and dark legacy code.

Michael talks about how most of us enter the world of programming in our youth full of enthusiasm and energy. I think that's true of most of the programmers I've worked with: it certainly includes me. As we get older, however, other distractions in life take precedence (family, mortgage, you know the stuff!) and it's tempting to view it as Just a Job and especially to view the mountain of legacy code you face on a daily basis with some despair.

But is the grass really any greener on a greenfield project? Michael graphically describes how a shiny new system can degrade into the same sad state as it's predecessor, or how a hopeful replacement system can falter as the changing state of the original system mean the greenfield team are chasing an ever-receding goal. Greenfield projects need TDD, continuous delivery and the like just as much as legacy projects. And - depending on the politics of your workplace - there can be a greater pressure on a greenfield team to achieve, which can make the work stressful and unpleasant.

I can agree with Michael's suggestion too that TDD'ing some code outside of work can be invigorating. A couple of years ago I learned Python and improved my TDD skills to develop a program outside work: this little project really raised my spirits and I started to enjoy the day job more as a result, seeing opportunities to use little snippets of the stuff I'd done at home.

Michael also recommends connecting with the larger community. Of course, that's what we're all doing now, on this group! Personally, I'm fortunate enough to live in a major city and can attend tech-groups where like-minded folk "play" with code or ideas for an evening: this again gives me little insights into how I can do my day job a little better. And it goes without saying that the ACCU conference is another excellent opportunity to stoke up the fires of your enthusiasm!

This chapter also touches on how the team's attitude can make even a massive, daunting legacy system an enjoyable experience. Michael suggests collectively "swarming" on a nastiest bits of code and getting them under control. That would give the team a collective sense of achievement, and also make them appreciate what's possible. I think positive metrics might help here too: a canny team leader might point out that the number of unit tests is mushrooming, or that code coverage is steadily improving.

My conclusion: what's important in any job is to feel you're making a difference, and legacy code offers unrivaled opportunities to do just that.

Chapter 25: Dependency-Breaking Techniques, part 1 •

Matthew Jones, 28 November 2011

This chapter pretty much *is* section III of the book, and is a directory of dependency breaking techniques that are applicable to code that is not yet under test. Each item finishes with a detailed list of steps, which if followed carefully, should prevent errors caused by the very act of editing. Even so, whilst editing, remember to bear in mind Chapter 23, "How do I know That I'm Not Breaking Anything?".

The book points out that some of the techniques might "make you flinch" and some certainly go against good practice. But remember we are using these techniques as stepping stones to getting the code under *some* form of test. Once under test we can increase the pace of refactoring, ending up at a good place. At that point, traces of the flinch-inducing stages should have vanished.

Adapt Parameter

As the name suggests, this is the application of the GoF Adapter Pattern to a troublesome parameter. The trouble is usually that the parameter is hard to create during test. We should use this approach when Extract Interface won't work.

We change the code to use an adapter interface in place of a specific instance of a parameter type. The production implementation uses the old parameter type, but the test implementation can do what we like.

A nice side effect is emphasised: moving towards interfaces makes the code easier to read as we communicate responsibilities rather than implementation detail.

A common problem in legacy code is not enough abstraction layers, so adding one will usually help in the grand scheme of things too.

Break Out Method Object

If you have a large method that can't be made static (Expose Static Method), consider breaking the method out into a new class. In the original code the method is accessed via an instance of the new class. [I think this is a form of functor ?]. The new class becomes a sort of life support system for the unwieldy method, and over time can grow into a more appropriate class as the method is broken down, and other code joins it.

The constructor for the new class takes a reference back to the original object, and all the parameters for the original method. The parameters are saved as instance variables, and the method then takes (void). Any interaction (methods or variables) between the method and the original class now go via the calling-object reference. We lean on the compiler to find these.

We now have a sort of inside out class hanging off the side of the original one: the old code with its guts hanging out. There is one further step that starts to make sense of the mess, and that is to Extract Interface on the original calling class so

that the new class deals with the interface, not the (old) concrete type. We lean on the compiler again, and it tells us that the methods on the interface are those of the original class that the new class calls, plus any getters for variables. Its a bit circular, but the diagrams and code in the book make it clear.

Once the dust has settled, we can make use of a GOOS idea: let the code talk to us. We will probably find other methods that belong in this new class, and it might even end up being the major player with the original, troublesome, class shrinking into the shadows. The large method should also be subdivided, but all in a new cleanly interfaced, testable, home. What ought to happen is that some kind of design will start to appear and we can continue to refactor the code towards this now that it can be tested properly.

I'd never read about, considered, or even discovered this one for myself, and once I'd got my head round it, it is very elegant. Its a nice application of OO principles, and I'll bet its very satisfying to use.

Definition Completion

This is basically the common or garden C/C++ stub (using the linker seam). Provide a test implementation of a function or class in the .c/.cpp file, given a production .h file.

It is only possible in some languages, and is a maintenance burden if the stub is non-trivial. It should be avoided unless there are no better seams to attack, and even then you should plan to get rid of the duplication once the code is under test.

Encapsulate Global References

If you have problems with globals during test you can try to make them act differently, use the linker seam to link to different globals in the test build, or decouple by encapsulating them in a class.

A very common code smell could appear here: if a number of globals are always used together, they all belong in the same class.

Think about what you name this class, but don't worry too much since it can always be re-named. The larger the scope, the harder it is to choose the right name, and global scope is the worst of all. If the right name is already taken, consider renaming the existing class.

[Why do they have the same name? Are they trying to be the same thing? Maybe the globals should be part of the existing class? Think before you act.]

Once the globals are in a class we can remove the originals and use the compiler to tell us where the new class is required. At this point we haven't changed the behaviour or signature of any code, so it all should have gone smoothly. Now we can attempt to inject the dependency (Parameterise from Above!) into the code we are testing: then we can fake it or use other test techniques.

We are encouraged to make small steps at first, while we get sufficient global-using code under test. 'Sufficient' is context specific, but once we are protected by tests, we can start refactoring the code, and the class holding the globals.

When dealing with global (i.e. free) functions, rather than variables, we can create an interface under which production and test versions can exist. Initially the production version will simply delegate to the old global function.

There are other techniques and seams that can be used to deal with globals, but encapsulation is the easiest to manage, and yields the most explicit seams.

Expose Static Method

Some classes are impossible to instantiate in a test harness. If we are lucky, there might be a method that uses no attributes or methods of its class. These can be made static and called without instantiating the class. [If you are lucky your IDE or lint-like tool will point this out to you for free.]

There is an interesting point made here: to be testable the method must be public. But why are we exposing some internal part of the class that the original design didn't expose? Static methods are arguably (and in some cases, actually) not part of the class. [They are dressed up global methods, IMHO!] It is better to make the method testable that to maintain some (quite possibly inappropriate) design standpoint. Later, protected by tests, the static code might find its way into a new home. There is always the option of making the static method protected, then access it through a testing subclass: Subclass and Override Method (later in the chapter).

Extract and Override Call

If the code we wish to test makes calls to a dependency that is hard to create in a test harness, or we want to sense through a class that is closed to us, we can use this technique.

The troublesome call is replaced by a call to a new, virtual, method on the same class. This new method makes the same call as before. In the test environment we create a subclass with an override for the new method. Here we can do what we like.

It is pretty simple, and straightforward to implement. In some cases Replace Global with Getter, or Parameterize Constructor might be more appropriate: they are presented later in the chapter.

Extract and Override Factory Method

This technique circumvents object creation in the constructor, but requires the language to allow calls to more-derived virtual functions from the constructor. C++ does not allow this. All is not lost because there are alternatives for C++, one being the next item.

The '"vexing" object creation code is placed in a virtual method which is called in the constructor. Extract and Override Call can then be applied.

Extract and Override Getter

If we want to manipulate access to a single instance variable during test, we can convert the class to always access it through a virtual getter rather than directly. [Note that later we are told that a downside is possible use before initialisation. This can be prevented by leaning on the compiler: simply rename the variable and resolve all resulting errors with calls to the getter. This trick also happens to make the technique very easy to implement.] In the test harness, we apply Extract and Override call to this getter, then manipulate or replace the instance variable however we wish.

[The book presents this technique as a solution for a class which creates an object at construction but does nothing else to it until later. I think it is more generally applicable, but there are details specific to the object creation case ...]

We can postpone the actual creation of the object with a lazy getter. In the test version the overridden getter can then eliminate creation of the original object entirely, replacing it with a test instance. [For this to work polymorphically I presume the return type of the getter would have to be an interface, or at the very least the type of the original object being created should be open to derivation.]

This technique wins over Extract and Override Call if there are multiple problematic calls to override on the same object.

Chapter 25, Dependency-Breaking Techniques, part 2 •

David Pol, 5 December 2011

Continuing with this long (but nonetheless very interesting) chapter...

Extract Implementer

This technique is used when we are extracting an interface but the name we want to use for it is already the name of the class and the IDE does not provide refactoring tools that may be helpful. To Extract Implementer, first we make a copy of the declaration of the original class and give it a different name[1]. Then, we delete all non-public methods and all variables from the original class, effectively turning it into an interface, and make all of the remaining public methods abstract. At this point, we can revise if all the imports in the interface file are really necessary (see Lean on the Compiler). Then, we make our production class implement the new interface and compile it to make sure it properly implements every method it should. After that, we compile the rest of the system to find and replace all the occurrences of the original class. Finally, we recompile everything and test.

This technique, Michael notes, is simple to apply when the original class doesn't have any parent or child classes in its inheritance hierarchy. When it does, we need to get more sophisticated or just go with the more direct Extract Interface.

[1] Michael makes an important point about naming being a key part of design. I agree completely: good design starts with good names. I also normally dislike having prefixes in type names (like an I for interfaces), but at the same time understand it may be useful to have a naming convention for the classes we've extracted an interface from.

Extract Interface

This technique, not surprisingly, is used when we want to extract an interface from a class. To Extract Interface, first we create a new, empty interface with the name we want to use. Then, we make the class we are extracting from implement the interface. After that, we change the place where we use the object so that it uses the interface rather than the original class. Then, we compile the system and introduce a new method declaration on the interface for each method use that the compiler reports as an error (again, Lean on the Compiler).

Introduce Instance Delegator

Michael introduces this section of the chapter by noting that people use static class methods for a variety of reasons. Two of the most common ones are writing a Singleton and creating a utility class (i.e., a class that doesn't have any instance variables or instance methods, like the ubiquitous Math class in languages that do not support free functions).

Sometimes, we have static methods that are problematic to use in a test. Introduce Instance Delegator is a technique that helps with that and goes like this: first, we identify the problematic static method and create an instance method that

delegates to the static method on the class (remembering to Preserve Signatures). Finally, we find the places where the static methods are used in the class under test and use a dependency-breaking technique like Parameterize Method to supply an instance at the location of the static method call.

Introduce Static Setter

This technique helps testing code with singletons[2]. In order to apply it, first we decrease the protection of the constructor so that the singleton class can be subclassed from. Then, we add a static setter to the singleton class that takes a reference to the singleton class (of course, it should properly destroy the previous singleton instance). If access to private or protected methods in the singleton is needed for testing, we need to consider creating a subclass or extracting an interface on the singleton class and supplying a setter that accepts an object with that interface.

[2] Personally, I think the best way to avoid problems when testing code with singletons is to not write them in the first place unless _really_ needed. In my experience (game code, where you see Renderer singleton classes and hundreds of other FooManager singleton classes everywhere), singletons are heavily misused as an unneeded abstraction over plain globals in a lot of situations where restricting the code to a single instance of the class makes it impossible for certain future requirements to be met in a sane way and can be much more easily enforced by convention (if you need a single instance, just write one! And if tomorrow we actually need _two_ Renderers, we can have that because we decided not to single-instance-lock the design for some of the more fundamental concepts in our codebase). Also, in a lot of these cases, not even the global access is justified.

Link Substitution

This one is a simple technique useful to fake a set of functions or classes. The process goes as this: first of all, we identify the functions or classes that we want to fake. Then, we write alternative definitions for them. Finally, we change our build so that the new definitions are used rather than the original, production versions. Michael notes that the best libraries to fake are those containing functions whose return values you don't often care about.

Parameterize Constructor

This technique is used when we want to replace an object that is currently being created inside the constructor of a class. The easiest way to do this is to pass the object from the outside. We need to follow these steps: first of all, we identify the constructor we want to parameterize and write a copy of it. Then, we add the desired parameter to the constructor for the object whose creation we are replacing. Finally, we remove the body of the old constructor and replace it with a call to the new constructor (in languages supporting delegation of constructors, C++ not being one of them up until recently with the new standard).

Parameterize Method

This technique is used when we want to replace an object that is currently being created inside a method. Again, as was the case with Parameterize Constructor (which could be considered a particular scenario for this technique), the easiest way to do this is to create the object externally and pass it to the method. We need

to follow these steps: first of all, we identify the method that we want to replace and write a copy of it. Then, we add the desired parameter to the method for the object whose creation we are replacing. Finally, we remove the body of the old method and replace it with a call to the new, parameterized method.

Chapter 25, Dependency-Breaking Techniques, part 3 •

Ann Napier, 12 December 2011

Continuing Chapter 25, there are yet more ways you can refactor your codebase to get it under test, with minimal chance of breaking something in the process.

I'd really like to see summaries of when to use the refactorings here – after the rest of the book was organised as a series of problems with solutions, a series of solutions feels slightly weird. That, and when I'm working in a Java codebase, some of the refactorings are for different languages entirely, so not immediately helpful.

Primitivize Parameter

This is for when you are trying to add new functionality to a hard to create object, which would use arguments that are also hard to create. However, only certain properties of the arguments are needed, so it's possible to create a simpler representation of the information you need and pass that to the new function. Another method can convert between the two representations.

It's a horribly hacky way of doing it, but at least allows you to get most of the new code under test.

Pull Up Feature

If there are some methods you want to test on a class which can be separated from the problematic dependencies, then try creating an abstract class, making the current class subclass it, and then pull up the 'nice' methods into the new abstract class. The abstract class can then be subclassed during testing.

Push Down Dependency

For classes with problematic dependencies in a lot of methods, instead of trying to subclass and override it might be necessary to make the current class abstract, and then push all the problematic dependencies down into a concrete class, keeping the testable parts in the abstract class. This way the abstract class can be subclassed during testing to avoid the dependencies.

It might make sense to move the methods in the subclass out to a different class later, if they handle a specific set of behaviour.

Replace Function with Function Pointer

This is for breaking dependencies in procedural languages with function pointers, like C. I haven't used function pointers, so hopefully this makes sense. This refactoring is also flagged as dangerous – many teams don't like to use function pointers because of possible corruption.

Here you replace functions with pointers to functions, which means that during testing it's possible to change the functions pointed to.

Replace Global Reference with Getter

Instead of using a global reference directly (like a static method in Java) access it via a getter, which can be overriden. Then you have a seam you can use when testing, subclassing and overriding the getter.

Subclass and Override Method

Probably the core refactoring to get code under test. Find the methods which are causing problems in testing then modify them so that you can override them. Create a subclass of the class you want to test and override the problem methods so that they can be used in testing – either making them do nothing or record the calls for testing.

Supersede Instance Variable

This is given as a way to get around the fact that you can't use Extract and Override Factory Method in C++. This is for removing a problematic object created in a constructor and replacing it with a fake object for testing.

In Extract and Override Factory Method instead of creating the object in the constructor, create it in a factory method called by the constructor. To test you can use a subclass which has the factory method overriden to provide a fake object.

However C++ won't let you call virtual functions from the constructor. So create the problematic object as normal. Add a new method which lets you set the object to one you supply, and use this in your test code to remove the problem.

Name the method in the pattern supersedeProblemObject() because that way it's easy to spot if anyone has been using a method from this refactoring in production code.

Template Redefinition

A slight problem for summarising this – I've never used templates in C++, and haven't done that much work with the language.

Template redefinition is another way to replace a problematic object with one that is more suited to testing. Instead of using inheritance, turn the class into a template, taking a generic version of the problematic object as a parameter. Then for normal code supply the actual object, and for test code put in a fake or sensing object.

To avoid having to use the template-style class name everywhere, a typedef is used to map it to the non-generic class name.

Text Redefinition

In Ruby you there's a way to replace a method without using subclass and override – you can just redefine it on the fly, before running tests using it. Import the class you want to modify into the test file, then provide an alternate definition for the method.

One caveat: this method stays as your test version for the rest of the tests. This sounds like a good way to introduce a confusing test issue.

Appendix •

Tim Barrass, 19 January 2011

In the Appendix Michael rounds off with an example of a specific refactoring ("Extract Method"), as described in Fowler's book.

It struck me as a little odd that he included this refactoring. It might make sense in context, if refactoring was still little appreciated at the time the book was written. I suspect that if the book were released today less attention would need to be drawn to the mechanics of it -- unless to highlight that people don't necessarily think about the mechanics of their refactoring tool of choice does for them. Then -- I'm a C# developer.

Most of Michael's approaches however -- especially the seam metaphor, which I use more and more every day -- aren't trivially bundled with a refactoring tool.

"Extract Method" is described in clear and useful detail, but I think perhaps the most useful thing the Appendix does is bring to mind the difference between relatively safe and relatively unsafe refactorings. When I think back over the book, I've gained a great deal from Michael's description of processes that help bring untested code under test -- refactorings in their own right that can be applied with some confidence without a safety net. From chapter 1:

"Changing code is great. It's what we do for a living. But there are ways of changing code that make life difficult, and there are ways that make it much easier. In the industry, we haven't spoken about that much. The closest we've gotten is the literature on refactoring. I think we can broaden the discussion a bit and talk about how to deal with code in the thorniest of situations. To do that, we have to dig deeper into the mechanics of change."

Digging deeper into the mechanics of change can only make us more self-aware as developers, and that's a very good thing.