Retrofitting unit tests

Steve Freeman

M3P,
12 Montagu Square
London
W1H 2LD, UK
+44 (0) 797 179 4105
steve@m3p.co.uk

Paul Simmons

Independent
6 Copse Close, Pattens Lane
Rochester, Kent
ME1 2RS, U.K
+44 (0) 7967 966203
pas@pobox.com

Stroking his chin sagely the old man replied, "Well now, if I was going there I wouldn't be starting from here." Ancient joke.

"If you do not start adding unit tests today then one year from now you will still not have a good unit test suite." Don Wells[1]

ABSTRACT
In this paper we describe techniques that we have found helpful for adding unit tests to existing code that has been written without tests. The paper presents some common coding practices that make unit tests hard to retrofit, and why. For each practice we suggest minimal refactorings to open up the code for testing.

Keywords
Refactoring, Unit Testing, Legacy Code, Retrofitting

1 INTRODUCTION

Unit tests can be hard to retrofit to legacy code, but not as hard as many developers believe; for our purposes, "legacy" is working code that must be maintained but that has been written without unit tests. We believe that it is worth attempting to improve the internal quality of any system that matters and that unit testing is a key technique for doing so.

Relentless unit testing is a core practice in Extreme Programming (XP) [1]. It gives the developers the confidence to make changes as new requirements arise or new refactorings are discovered. Furthermore, when written before the code, unit tests are a powerful design tool that act as executable specifications; they concentrate the programmer's mind on what is really needed and help to drive the code towards good coding practice [2].

Many projects, however, convert to XP after starting with another methodology, which usually means that there is an existing code base that does not have a thorough unit test suite. The dilemma for the team is that they need a testing safety net to support the agile development practices they want to adopt but cannot write unit tests for the entire code base for two reasons. First, retrofitting unit tests is expensive, full coverage can easily take as much effort to write as did the original system without adding any visible functionality. Second, there is an obvious deadlock in that legacy code often needs some refactoring to make it testable, but refactoring should not be undertaken without tests in place to prove that it's safe.

Both problems must be addressed by a combination of skill and compromise. First, unit tests can be added incrementally, perhaps before changing a component for the first time during subsequent development. Combined with some judicious functional testing, the team can give themselves enough confidence to make progress, although at less than full speed, whilst improving the quality of the code. Second, our experience is that carefully fixing a few "code smells" without unit tests can give the developer enough leverage to bootstrap the writing of a full test suite. As the test suite builds up, the developers should look for opportunities to improve it as suggested by [3].

In this paper we concentrate on those careful fixes. We describe some common code smells that we have found inhibit the retrofitting of unit tests, and suggest tactical refactorings to make such code more accessible. Most of the smells we have identified are concerned with the difficulty of isolating the code we wish to test from the rest of the system, a key requirement for effective unit testing. Our experience is that changing code to make it testable usually improves its quality, with a clearer and more flexible structure. When we retrofit unit tests, we can also try to retrofit the design benefits that come with test-first programming.

Our experience is mainly based on Java, but we believe that most of these patterns apply to other object-oriented languages. We assume that the reader is familiar with test-first development, the JUnit framework [4], and refactoring as described by Fowler [5]; we annotate patterns and refactorings from Fowler using *[F]*.

2 CODE SMELLS

This section describes some common code smells that make unit tests difficult to add to legacy software. For each smell, we offer an appropriate refactoring in the next section.

Singleton

The Singleton is perhaps the most widely used and misunderstood pattern in Gamma *et al* [6], and is often found in legacy code. A common use of Singleton is to encapsulate external resources such as databases or files. Since it provides a single access point, calls to a

[1] http://c2.com/cgi/wiki?UnitTestingLegacyCode

singleton are often scattered throughout the code.

The issues for unit testing are: first, sometimes the singleton object cannot be changed because, for example, it is set up in a static initializer (see below). This makes it impossible to isolate the tested code from its environment by substituting a mock implementation [7] of the singleton. Second, even where the singletons can be replaced, the tests for objects that refer to many singletons will be tedious and error-prone to set up. Finally, many uses of a singleton will repeat behaviour that must be tested separately for each case, increasing the testing effort.

One solution is to add a setter method to the singleton class to overwrite its static instance. This weakens the encapsulation of the singleton itself but may be suitable for cross-application features such as a logging interface. The test suite can use the setter to assign a mock implementation and the application can continue to use the singleton as before. Rainsberger [8] suggests aggregating singletons in a *Toolbox* so that their lifetimes can be managed by the application. An alternative approach that does not alter the singleton class is to *Pass singletons through*.

Complex construction
Sometimes most of the implementation of a class is concerned with setting up its initial state and is not used again after instantiation. For example, a class to represent a financial yield curve requires complex calculations to work out its initial values, but only simple lookups when in use. Similarly, a class that represents a user may refer to an external directory service only during initialization.

The issues for unit testing are: first, it is cumbersome to create instances of the class when testing both the class itself and classes that interact with it; for example, it may be too hard to create every state that needs testing via the public constructors. Second, construction that relies on external resources is an unnecessary dependency when managing unit test suites. Third, the test suite for class instances will be less readable because it will be swamped with tests for construction rather than tests for use. These are all symptoms of a poor separation of concerns.

A first step would be to add a simple constructor to the class and to write separate test suites for construction and use. A better approach is to refactor using *Separate construction from use*.

Data class
Data class, which consists mainly of fields and their getters and setters, is described in Fowler. Data classes are often found with utility classes to support common operations on them.

The issue for unit testing is that data classes often imply that some related behaviour has been scattered around the clients of the class, so related test code has to be repeated or gathered into helper code. Furthermore, code that passes data objects around tends to have *Long methods* (see below) that are hard to test.

Even where data classes are required, perhaps for use in a reflective framework, it is often possible to move responsibility to the data class by combination of extraction, encapsulation and moving, as described in Fowler.

Static initialization
Many developers use static initialization, code that is run when a class is loaded, to set its initial state; common examples are initializing singletons, starting loggers, and loading property values from files. Whilst this technique is useful for reducing the intellectual load on the programmer and for ensuring the internal consistency of a component before it is used, there are maintenance costs if the static code is complex or refers to external resources.

The issues for unit testing are: first, it can be difficult to run repeated tests over such code. To do so requires repeated reloading of the class, it may be hard to set up conditions to test failures, and errors may be hard to trap for test results. Second, such classes are hard to instantiate outside their framework when they are required for testing other classes, especially when the source code is not available. For example, one of the authors got stuck trying to create a parameter object from an application server because a static initialization in a super type was failing silently.

The solution is to *Remove complex static initializers*.

Bleeding across layers
It is quite common to see business domain code use framework classes, such as Servlets, so that package dependencies "bleed" across the layers of an application. Examples include passing a Servlet request as a parameter to a domain class, or throwing a Servlet exception from within a domain class. This risk may be higher on Extreme Programming projects, where the programmers aspire to "Do The Simplest Thing That Could Possibly Work."

The issue for unit testing is that bleeding across layers introduces unnecessary dependencies between components and, hence, between tests. First, anyone reading or writing a test for the business class must understand both layers and the tests are less likely to read well. Second, if classes from the framework layer change, this may require business layer tests to be changed. Finally, test setup may be difficult if, for example, some framework classes do not have constructors that are accessible outside the framework.

The solution is to refactor at the places where the layers touch and *Weaken dependencies between layers*.

Classes as parameters
In Java, it is worth specifying the parameter and return types of a method (its signature) in terms of interfaces rather than classes, if those classes are at all complex.

The issue for unit testing is that, for parameters that are defined as classes, a mock or stub implementation can only be substituted by subclassing, which has two limitations. First, it cannot also inherit from common mock or stub implementation classes, nor can it take advan-

tage of Java proxies, as with EasyMock [9]. Second, if the parameter class, or one of its ancestors, changes or adds a method, the stub class will no longer override all the real methods and the test case might pick up the wrong implementation. Such bugs in the test environment can be difficult to find when the test unexpectedly fails (or, worse, passes). Similar issues arise with return types; when the class itself is stubbed out for testing other classes in the code base, it may be easier to return a simple stub than an instance of the real type.

The solution is to *Replace class with interface* in the signature. If this solution is too difficult to apply at first, perhaps because the parameter class is used in many places, then first create the stub implementation as a subclass of the parameter class and later refactor both the stub and original classes with *Extract Interface [F]*.

Imprecise exceptions
Java supports *checked exceptions*, where the compiler will validate that all the exceptions that might be thrown from within a method are either handled or declared as part of the signature. Some developers avoid checked exceptions by catching and dropping exceptions they don't know how to handle (that is, by ignoring the signal), or by declaring the method to throw the generic type Exception. An equivalent to the latter is to always throw unchecked exceptions.

The issue for unit testing is that exception handling must also be tested. First, it may be hard to detect a result that will confirm that an exception has been thrown if the target code drops it. For example, if the beginning of a method drops an exception, its unit tests ought to be run twice, once with the exception thrown and once without. Second, where exception checking is ignored, it can take some time to work out and unit test all the possible exception paths through the code.

The solution is to be precise when managing checked exceptions. Dropped exceptions should be encapsulated by *Extract Method [F]*, which will often suggest a further *Extract Class [F]* to reify the interaction with the component that throws the exception. Checked exception lists should be narrowed to just those exceptions that a method can throw, this can be propagated incrementally from where the code touches external libraries. Our experience is that a little rigour applied to indistinct Java exception management can greatly simplify the code and, hence, the unit tests to drive it.

Long method
Long method is described in Fowler. The additional issue for retrofitting unit tests is that such methods are also painful to test. Typically this involves writing a long series of tests, each of which progresses a little further through the method before forcing the next exit condition. Setting up enough state in a test to get through the entire method is, at best, complicated.

If the method is too long to test as it stands, one solution is to test and refactor incrementally. Long methods often contain several logical sections, for example: check the inputs, perform operations, and assemble the result. Test a section at a time and extract helper methods to isolate it. If possible, extract a section and its tests as a class, perhaps as a policy object. Subsequently, this may be replaced with a Mock Object and the tests for the method simplified.

In the best case, a long method collapses either to a class in its own right, or to a collaboration between a set of smaller objects, that can be tested separately. The tests for the refactored method need only exercise the routing between those objects.

3 REFACTORINGS

Pass singletons through
Objects that are neither ubiquitous, such as loggers, nor constant values should be passed through as method parameters, rather than retrieved as singletons; a common example is a database connection. This can be done incrementally by first creating the singleton DBConnection instance, and passing it as a parameter to low-level methods. Then later propagating the new parameter up the call stack, until it can be set up from some suitable high-level routine. There is a risk that parameter lists will become too long as more singletons are removed, but in practice we have found that ex-singletons, such as external connections, are usually local to a sub-system or package. Furthermore, passing singletons through as parameters often leads to *Introducing Parameter Objects [F]* which, in turn, suggest useful refactorings.

The advantage for unit testing is that a parameter, particularly if it is an interface, is easier than a singleton to replace with a mock implementation, thus isolating the test from the rest of the application.

Separate construction from use
Where most of the implementation of a class is taken up with constructing an instance, such as calculating the yield curve on a financial instrument, consider separating the construction aspects into a factory object—our mental image for this is the way that booster sections are jettisoned during the launch of a space rocket.

This technique is most likely to apply when the construction phase uses different resources or libraries from the use of the object. The benefit for unit testing is that the two classes should have more focused responsibilities and so be easier both to test and to stub out.

Remove complex static initializers
A first step is to move static initialization code into static methods so it can be referred to by name and parameters and results passed through. Techniques such as lazy initialization allow such methods to be called explicitly, for testing, or automatically when in production.

It may be, however, that code of any complexity should not be run implicitly, but should be made visible and called directly from the application startup sequence. This makes error handling easier to manage and ensures that failures occur at the right time. One of the authors used this technique when porting a component between two frameworks that used different error reporting. The move revealed a failure in initializing the logging li-

brary that had previously been hidden by an incorrect startup sequence.

Weaken dependencies between layers
To reduce class dependencies between layers of an application, there are three cases to consider: First, where explicit creation occurs across the boundary, such as creating a new Customer object from a servlet, consider *Replace Constructor with Factory Method [F]*. Thus the servlet might now use a CustomerFactory to create a Customer, rather than instantiating one directly. When unit testing we can substitute a mock CustomerFactory that instantiates a mock Customer.

Second, where several values are passed across a boundary, consider *Introduce Parameter Object [F]*. For example, when passing start and end dates from a user's http request to an Account object, we might bundle these into a DateRange type. This clarifies the relationship between the layers and we are likely to be able to move behaviour to the new parameter object, which can then be tested in isolation.

Third, where a framework layer needs to interrogate its client layer, it should define a callback interface that the client layer can implement. For example, where an Account object needs to extract session values from an http request, define an AccountSession interface that makes explicit what an Account needs to know about its context, then implement an HttpAccountSession class for use with servlets. We can now unit test separately the extraction of the values from the http session and the use of those values in the Account. For the Account class, we can create a MockAccountSession to isolate its tests from the servlet framework.

Replace class with interface
In Java, where the input parameters or return value of a method are typed as classes that are at all complex, consider changing those types to interfaces and renaming the classes. Types based on interfaces are easier to substitute with stub or mock implementations, so it becomes easier to test a class in isolation from the rest of the system. The overhead of maintaining the extra type is mitigated by modern development environments and by the flexibility it adds to the code. One implication of this technique is that the coding standard should not use type names to distinguish interfaces or classes, such as with a leading or trailing 'I', as this hinders refactoring between the two.

With some care, the same technique can be applied in C++ by using abstract classes as interfaces and multiple inheritance to bind them to implementation classes.

4 RELATED WORK AND OTHER TECHNIQUES

There is a growing body of experience with test-first development: Fowler [5] catalogues the core code smells and refactorings, and there are links to papers and discussions from the JUnit site [4] and on the C2 wiki [10]. This paper focuses on code smells and refactorings related to retrofitting unit tests.

There have been some interesting discussions about the use of Aspect Oriented Programming [11] for unit testing. The idea is to intercept the calls the target code makes to other objects in the application. One idea is to use this technique to implement Mock Objects, tracking calls and returning preloaded results [12]. An alternative is to log important values when running functional-level tests and check that these don't change during refactoring. In our view, these are valuable intermediate techniques to help with opening up opaque code, but we are wary that they change the actual code under test.

5 CONCLUSIONS

In this paper, we have identified some coding practices that make the retrofitting of unit tests difficult. We have identified some related refactorings that we have found allow us to "chip away" at the code enough to start adding unit tests. These tests then give us the confidence to refactor, add new functionality, or fix bugs using test-first programming.

Those of us who practice test-first programming do so because we believe that it is more effective and drives us to writing better code. Many of us, however, also have to work with existing code that we cannot break, but need to change. The authors have found that retrofitting unit tests helps to support programmers when making changes and to guide the code to a better design through refactoring.

How much time to spend on retrofitting unit tests, or whether to do so at all, is outside the scope of this paper; it can be an expensive exercise. For those who chose to do so, we hope that this paper embodies some useful experience. Before starting to refactor for testing, we also recommend that the developers write some functional tests that touch the components concerned to catch any gross errors that they might introduce.

Finally, the real point of this paper is that, given the will and enough slack in the immediate schedule, it *is* possible to add unit tests to almost any existing code base—and for a team that wants to be agile, it is essential.

ACKNOWLEDGEMENTS
Thanks to Michael Feathers, Tim Mackinnon, Duncan McGregor, and Rachel Davies for their comments on early versions, and to the members of the Extreme Tuesday Club for being part of the community.

REFERENCES

1. Beck, K, *Extreme programming explained: embrace change*. Addison-Wesley, 1999.
2. http://c2.com/cgi/wiki?UnitInUnitTestIsntTheUnitYouAreThinkingOf
3. van Deursen, A., Moonen L, can den Bergh, A, Kok G, *Refactoring Test Code*, XP2001, Sardinia, 2001.
4. The JUnit web site. http://www.junit.org
5. Fowler M., *Refactoring: improving the design of existing code*, Addison-Wesley, 1999.

6. Gamma E, Helm, R, Johnson, R, Vlissides, J. *Design Patterns*, Addison-Wesley, 1995.

7. Mackinnon T., Freeman S., Craig P., *Endotesting: unit testing with Mock Objects*, in *Extreme Programming Examined*, Addison-Wesley, 2000.

8. Rainsberger, J. *Use your singletons wisely*, http://www-106.ibm.com/developerworks/components/library/co-single.html

9. EasyMock http://www.easymock.org/

10. http://c2.com/cgi/wiki?UnitTestingLegacyCode

11. http://www.aspectj.org

12. http://groups.yahoo.com/group/extremeprogramming/message/37004