220

Working With Legacy Software

Dr Robert Chatley - rbc@imperial.ac.uk

@rchatley **#doc220**

In this section we look at how we can use some of the design principles and techniques that we have covered to help us to work effectively with existing (and possibly badly designed) code bases.

Legacy System

- *A legacy system is a piece of software that you have **inherited** and that is of **value** to you.*

Some other people have some other definitions for the term legacy. Some simply think of it as a system that is old. Others a system that is messy. Michael Feathers has quite a specific condition for a legacy system, he thinks of it as a system that has no automated tests.

Another term that is often associated with the term legacy, is the term maintenance. People often think of maintenance as being low value work, fixing up broken or ageing systems. They think of it as a lower class of work from the "value-adding" work of implementing new features.

Successful Systems Evolve

We think of the maintenance of a software system as being the process of changing it, guiding its growth and evolution throughout its lifetime. A valuable system will have different stakeholders constantly wanting to make changes to it to improve its scalability, or the range of features that it has. Managing this evolution and change, while also managing and controlling risk, is a key theme of this course.

Preserve Existing Behaviour

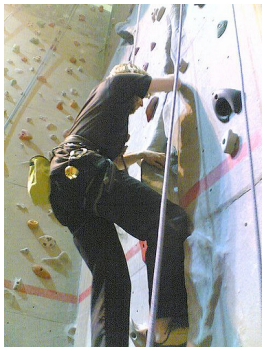
Keep the stuff that works

Don't change too much

So how do we approach making a change to a system? A guiding principle is to keep the stuff that works, and don't change too much. Try and keep any changes that you make minimal, just enough to effect that change that is required. Aimless refactoring is not generally of value. Changing things in areas of the code that we are not familiar with is risky and we can easily break something.

Confidence In Changes

Good tools and techniques give us courage to change things, otherwise it's too scary

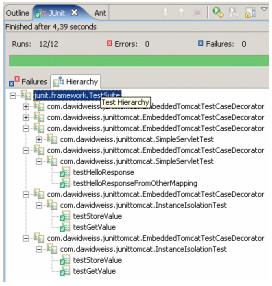


Good tools and techniques give us courage to change things, otherwise it is too scary to do so. How do we know if we broke an existing behaviour of the system? Did we introduce a bug? Even small bugs can be very expensive in high volume or high availability systems. We need to find a way that we can work where we are confident that our changes are correct, and that we have not broken the system.

Test Harness

Unit test at the micro level

System test at the macro level

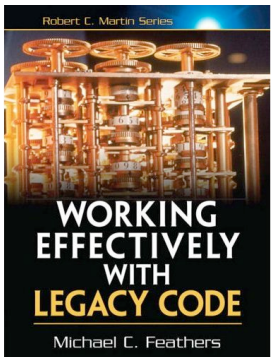


We utilise a lot of automated build and testing techniques to try and give us this confidence. Manual testing is too slow, expensive and error-prone, so we introduce and rely on automated tests around any changes that we make.

They way that we provide ourselves with this safety net is through a test harness. We test at different levels, on a micro-level for individual classes and methods using unit tests, and also at a system level, to test end-to-end behaviour, in terms of what the user requires from the system.

We will examine different types of testing technology that help us to design, work with customers, and add tests into systems that currently have little or no test coverage.

Testing The Untestable



In his book Working Effectively with Legacy Software, Michael Feathers defines a legacy system as any such system where the code is untested. He demonstrates techniques that can be used to introduce tests to untested code. Unfortunately, code without tests is normally not designed with testability in mind, and we can find it hard to access the parts of the code that we need in order to create an effective test. Conversely, when we design a system from the beginning using a test-driven approach, testability of each component is something that we think about all the time, and it effects the design of the system, but in a positive manner.

Unpicking A Mess

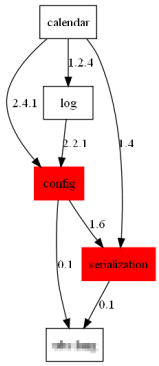
- *A legacy system may be messy*

More often than we would like, when we come on to new projects, we are faced with a mess. We must be careful not to judge a codebase as being a mess, or of low quality, just because the way that it has been written is different from the way that we would have done it. It may be consistent in its style and structure, and it may just take a little time to understand the design principles that underpin the system.

Dependency

One of the first things that we would like to be able to do is to discover the structure of the system. This may be in terms of classes and objects, modules and components, packages, layers, processes and/or the interfaces and interoperation with other systems. In some sense we may think of this as being the architecture of the system - although the term software architecture also covers a range of other concerns. The links between these components form dependencies.

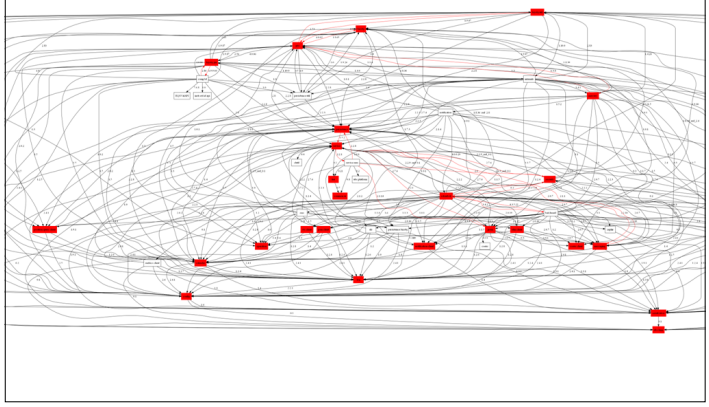
Visualising



These may be a linking dependency, say of one component on another, for example one class referencing another, or perhaps a third party library. They may also be dependencies on external services such as databases, or other services that our system calls, or receives messages from.

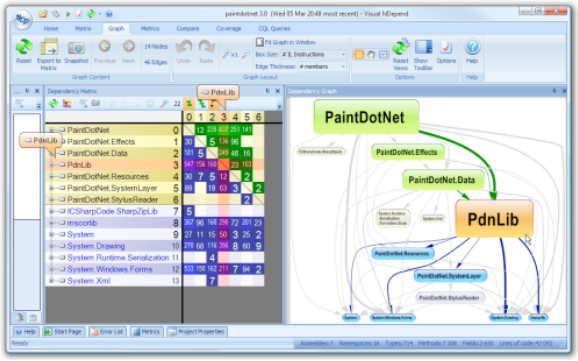
We can try using some tools to create some visualisations of the dependencies to try and help us out. Some tools will analyse dependencies and show them in a graph - typically boxes with arrows between them.

The Bigger Picture



This is typically the sort of diagram you might draw by hand to indicate dependencies between system components. In a large system, and/or one that it poorly modularised, such a graph can become cluttered and difficult to read.

Dependency Structure Matrix



An alternative representation, which may be easier to find patterns in for large systems is the Dependency Structure Matrix, produced by tools such as NDepend.

http://en.wikipedia.org/wiki/Design_structure_matrix
http://www.ndepend.com/Doc_Matrix.aspx

Seams

- "A ***seam*** is a place where you can alter behavior in your program without editing in that place" - M. Feathers

Photo by Fraser Speirs

In order to be able to effectively test particular units of a system, we need to be able to break dependencies so that during the test phase we can test an isolated unit. For example, during our unit test, we do not want the code to be writing data into a database, sending emails, or booking million dollar financial trades. We can break these dependencies by making use of what Feathers calls a *seam*. A seam is a place where you can alter the behaviour of your program without editing it in that place.

Seams

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

Every seam must have an *enabling point*. This is the point where you decide to use one behaviour or the other. To make use of an object seam, we must be able to pass in our test implementation, instead of the real implementation of the dependency. This may not be possible if the code under test simply uses the new operation to create its dependency, or refers to a singleton instance. We may need to perform a refactoring, for example introducing a method parameter, or a constructor parameter, that allows us to pass in a reference to the dependency from outside. This provides the enabling point for the seam, allowing us to switch in our test object instead.

Sensing

- "We break dependencies to ***sense*** when we can't access values our code computes"

In addition to breaking dependencies using seams, we may also want to use our test implementations to help us with *sensing* effects of running our code during the test. Say we have some code that writes its results into a database, within a unit test it will be hard for us to verify the results of our test. We would have to write code within the test to query the database after we had run our test, to check what values were present. This would be slow to execute, and would also be dependent on the state of the database, which we might not be sure of if the test runs repeatedly, or if the database is shared with other users. If we replace the dependency on the database with a test implementation, using an object seam, then we can add code to our test implementation that allows us to verify what was called on it.