eXtreme Programming In A Hostile Environment

Graham Wright
Workshare Technology
New Loom House
101 Back Church Lane
London, E1 1LU, UK
+44 20 7481 6100
graham.wright@workshare.com

Abstract

As the use of XP grows a debate is emerging about what type of software projects it can be successfully applied to.

This paper describes the successful adoption of XP in a project combining five of the features that are generally considered hostile to the implementation of XP; a large team, a legacy code base, C++, COM and a GUI intensive application. Such factors are generally presumed to make XP practices such as refactoring and test first design difficult.

Despite these problems the company described has successfully adopted XP, bringing products to market written in Visual C++ and with a heavy GUI and COM reliance. This was achieved with a programming team of twenty and with a large legacy code base.

This experience demonstrates that XP can be applied to projects not usually considered to be appropriate and scaled up beyond the generally accepted limit. This success is important to the XP community because these features are common to many Microsoft based projects, projects that form a significant proportion of today's software industry.

Keywords

C++, GUI, COM, legacy code, large teams.

THE TRANSITION TO XP

Workshare Technology [14] produces document change management software.

The code base consists of 2 major Windows applications sharing approximately 100 components. These contain 1200 classes and 185,000 lines of code. The software is written in Visual C++ using MFC (Microsoft Foundation Classes, the industry standard framework for developing C++ Windows applications) and ATL (Active Template Library, the corresponding framework for COM development). The applications are GUI and COM intensive.

The development team initially consisted of 15 programmers, 8 product managers (the XP customers) and 10 QA/testing staff.

XP was adopted in February 2000, by which time the majority of the final code base was already in place. The transition was kick started by sending a mixed team of programmers, customers and QA staff to a one-week XP course run by ObjectMentor [13]. This company then provided coaching over the following six months. Understanding that each reinforces the others and that the full value of XP does not come until all are in place [1] we adopted the XP practices wholesale and have tried not to deviate from any of them. Our experience also leads us to believe that such a transition requires external coaching.

The transition to XP has been an unqualified success and has resulted in several product releases within a short period of time, a doubling of productivity and a four-fold reduction in defect rate.

XP AND C++

XP grew out of the Smalltalk programming community and has initially been adopted mainly by those working with either Smalltalk or Java. It is generally supposed that these languages are more amenable to XP than C++.

Large C++ programs can be rigid and resistant to change. In these circumstances frequent refactoring is not possible as build times become a significant proportion of the development effort and changes in one module ripple throughout the code base resulting in program instability. Such code prevents the adoption of XP practices such as growing functionality using small incremental changes, rapid integration of those changes and refactoring of the code base whenever and wherever possible.

However XP can be combined with C++ as long as the code is structured to minimize dependencies and coupling between classes [12]. The techniques required to minimize dependencies are not particular to XP and represent the best practice that has accumulated over the last twenty years for reducing the coupling within large C++ programs (see for instance [10] and the references in [12]).

Specifically we restructured our code base to ensure that only one class was defined in each header file, header files did not include other header files and header files referenced client classes by forward declaration and pointers rather than by embedding that class. In addition we also ensured that code imported COM definitions from type libraries rather than from COM DLLs and further reduced COM specific dependencies by separat-

ing structure and enumeration definitions previously contained within IDL files into separate header files. Finally the creation of wrapper classes required to facilitate the testing of COM and GUI classes, as described in subsequent sections, further reduced dependencies and coupling within the system.

As a result of these changes the compile time resulting from an incremental change to the code base declined dramatically and refactoring became feasible.

Parallel to reducing compile times we introduced an automatic build script running on a scheduler twice a day and on demand. Although falling short of the "continuous integration" described by Fowler [7] this ensured that we always had a working build and that any change made by the team was within a build within half a day.

TESTING COM CODE

Unit testing was done using a derivative of CppUnit [5].

In addition to the complex dependencies discussed above, factors such as the lack of reflection are considered to make testing C++ code inherently more difficult than testing Smalltalk or Java code. This is made worse in many Windows applications as the functionality to be tested is often deep within layers of GUI or COM wrappers. However unless C++ code can be tested easily and rapidly, XP practices of test first design, enhancing functionality via small incremental changes and refactoring become difficult or impossible.

The requirement to isolate the functionality to be tested contradicts the code generated by frameworks such as MFC and ATL, which embeds functionality within GUI handlers and COM interface implementations. For instance ATL code cannot be tested without instantiating the COM coclass, making testing more complex than it need be and preventing the testing framework from accessing any methods or member data not exposed by the COM interface.

The first step in enabling testing is thus to refactor this code to separate the functionality from the GUI or COM framework. Essentially this involves implementing the envelope - letter idiom [4] or bridge pattern [8] creating an implementation class refactoring the code to move the functionality into that class.

An example of such a change is shown below. In this sample CFileVersion is a thin wrapper implementing the COM interface and CFileVersionImp is the worker class implementing the functionality that was originally in the COM class before refactoring and to which the COM wrapper class now delegates all its calls;

```
CFileVersion::FinalConstruct()
{
        m_imp = new CFileVersionImp;
        …

CFileVersion::get_BuildVersionNumber(long *pVal)
{
        hr = m_imp->get_BuildVersionNumber(pVal);
        …

 CFileVersionImp::get_BuildVersionNumber(long
*pVal)
{
        …
```

The benefit of this is that the implementation class may be shared by both the original COM component and the testing framework, enabling the tests full access to the class without the overhead of COM. If necessary the testing class may be made a friend of the implementation class.

This design also has benefits in production code as it encourages the storing of member data in the form of standard data types rather than COM types such as BSTRs and VARIANTs. This avoids a series of bugs associated with storing member data in these COM formats.

Essentially the outer COM class should contain no member data other than a pointer to the implementation class and merely exposes the COM interface to the outside world.

Testing the COM layer

This COM wrapper still requires its own testing to verify the handling of COM specific issues and the translation of COM data types to and from the C++ types used by the worker object. Such testing is achieved using mock objects [11]. When tested in this way the COM wrapper does not instantiate the normal worker object but a mock object or stub.

In this configuration three issues can be tested. Firstly the handling of invalid or missing parameters in the COM calls, for instance empty or corrupt BSTRs. Secondly the translation of COM data to and from the worker object, for instance by having the mock object expect or return a known data value. Thirdly the robustness of the COM wrapper to errors in the worker class, for instance by having the mock object throw an exception or generate an access violation.

TESTING GUI CODE

Despite being nominally object oriented, frameworks such as MFC do not completely separate data/functionality from view/GUI. To some extent this reflects the GUI intensive nature of most Window's programs but the side effect is to make testing more difficult. This coupling is integral to MFC, which maintains a map between member data and the control displaying that data through functions such as DoDataExchange. In refactoring such code the aim is to make core functionality testable outside the GUI or, at a minimum, testable is such a way that no human interaction with the GUI is required.

This refactoring is similar to the COM code described above with the additional requirement of redirecting the

MFC generated data mappings. This results in code such as;

```
CGuiTestDlg::CGuiTestDlg(CWnd* pParent
{
        m_pWorker = new CGuiTestWorker;
        …

void CGuiTestDlg::DoDataExchange(CDataExchange*
pDX)
{
 //{{AFX_DATA_MAP(CGuiTestDlg)
DDX_Text(pDX, ID1, m_pWorker->m_SomeData);
…

void CGuiTestDlg::OnOK()
{
        UpdateData(TRUE);
        m_pWorker->DoSomeWork();
        UpdateData(FALSE);
```

As with the refactoring of ATL generated code, the outer GUI class contains no member data other than a pointer to the worker class. The only unsatisfactory feature of this solution is that the MFC data mapping still requires direct access to the worker class's member data, either by making this data public or making the GUI class a friend of the worker class.

Testing GUI dependent code

Some functionality is genuinely coupled to the GUI and so cannot be tested in isolation from that GUI. In these cases testing may only be possible using screen scraping tools.

However before adopting screen scraping, which is notoriously fragile to even minor layout changes, other methods should be investigated. Driving the GUI using SDK calls is often possible. Functions such as EnumWindow and GetWindow can locate the target GUI element, PostMessage can control that element and GetWindowText can query the data displayed by that element. In many cases the combination of these functions provides the equivalent functionality as screen scraping without the associated layout sensitivity.

Alternatively the GUI output may be redirected during testing to an isolated Windows control. This scenario is similar to using a mock object but differs in that a real Windows control is used. Rather than substituting the control with a mock object, the GUI output is captured in a control within the test framework by changing the method of instantiating the object under test.

Test first GUI design

Although in general unit tests should run automatically without human interaction we found it useful to occasionally switch on the testing of GUI components such as dialogs. This enabled "test first GUI design" in which the test framework allows dialog layout and functionality to be modified in isolation from the application that the dialog would normally be embedded in.

For instance a file save dialog may only appear in an application after the user has completed a number of complex steps. Using the test framework to host the dialog independently of the application reduces the duration of this design test cycle dramatically.

The .Net framework

Over the coming years it is probable that Windows applications that would previously been written using either MFC or ATL will be written using the .Net framework. The code generated by this framework also combines GUI and data and will require similar refactoring to enable testing.

WRAPPING LEGACY CODE

When we adopted XP we were faced with a paradox. We wanted to refactor the legacy code but we had no unit tests to verify any code changes we made. However we could not easily write unit tests because of the structure of that code.

Initially we wrote tests, often using scripting, to test broad areas of application functionality. These large-scale tests were not unit tests as generally understood in XP and were closer to acceptance tests. However they gave an immediate indication that code changes had not compromised the application's core functionality and provided the team with the confidence to begin the initial refactoring necessary to enable true unit testing. These tests were retired as the proportion of code covered by genuine unit tests increased.

Some legacy code was resistant to refactoring without significant rewriting due to the complexity and instability of the original code. Rather than rewrite such code its functionality was treated as a black box and isolated from the rest of the application by redirecting all calls to it via wrapper classes. These wrapper classes were developed using standard XP test first design and provide indirect testing of that legacy code.

As well as isolating legacy code these wrappers have significant advantages in hiding the complexity of the original code, enabling the renaming of functions to more closely describe their intention, providing a robust exception handling mechanism and the incorporation a standard error reporting module.

LARGE TEAMS

XP originated in small development teams and a team size of between 2 and 10 is still considered to be ideal [9]. Many development teams are bigger and the team size limit beyond which XP will not scale is still to be determined. Our development team contained 15 programmers when we adopted XP, has expanded to 20 and is scheduled to grow to 28.

Much of the concern about scaling XP to larger teams derives from the assumption that the communication and

management overhead always increases exponentially with team size. The evidence for this assumption [2] derives from projects with a heavier, more paper orientated development process than XP. Its not yet clear if this assumption is still correct when applied to the less formal, mainly verbal communication that characterizes XP. So far our own experience contradicts this assumption. A key to this has been designating an individual member of our 8 strong product management team as the sole customer for each story within an iteration.

More problematic has been the psychological factors in maintaining team cohesion. It is easier for people to identify with a programming team of 8 than one of 20. Maintaining the feeling that all the programmers, customers and QA are one 40 strong team is even harder. Essentially our experience of scaling up XP has been the centrality of maintaining morale and motivation within the entire development team. Such concerns are not specific to XP and are shared by any development methodology dependent upon highly productive and coherent teams [6, 3].

At one point during the project we effectively split the programming team into three teams, each team concentrating on a functional area within the product. Whilst this had the positive effect of increasing the unity and communication within each team it had the negative effect of disrupting the cohesion of the entire development group. It also became clear that the longer these separate teams existed other negative effects would become paramount such as a decrease in the understanding of the other teams code and a decline in the morale of those assigned to teams considered to have less desirable stories. We reverted to a single programming team within two months of this experiment and have retained a single team since. An exception to this is the occasional formation of "swat teams" dealing with urgent problems. Such teams exist only for the duration of a single iteration.

The only deviation from maintaining a single programming team we have found useful is during iteration planning when breaking the team into smaller groups to discuss the detailed tasks of a story is more effective than having the entire team involved. However the initial presentations by customers of the iteration's stories are still made to the programming team as a whole.

INFORMATION AND QUESTIONS

For more information, contact: graham.wright@workshare.com

ACKNOWLEDGEMENTS

Visual C++, Windows, MFC, ATL and .Net are registered trademarks of Microsoft Corporation.

REFERENCES

1. Beck, Kent, *Extreme Programming Explained*, 149-150. (Addison-Wesley 2000).

2. Brooks, F. *The Mythical Man-Month,* 14-26 (Addison-Wesley 1975, 1995)

3. Cockburn, Alistair. *Agile Software Development*, 75-111. (Addison-Wesley 2001)

4. Coplien, James O. *Advanced C++ Programming Styles and Idioms*, 133-134. (Addison-Wesley 1991)

5. CppUnit. C++ Unit testing framework. http://cppunit.sourceforge.net/

6. Demarco, T and Lister, T. *Peopleware: Productive Projects and Teams* (Dorset House 1987, 1999)

7. Fowler, Martin. *Continuous Integration*. (ThoughtWorks) http://www.thoughtworks.com/Continuous Integration.pdf

8. Gamma, E, Helm, R, Johnson, J and Vlissdes, J. *Design Patterns*, 151-161. (Addison-Wesley 1995).

9. Jeffries, R. When should Extreme Programming be Used? http://www.extremeprogramming.org/when.html

10.Lakos, John S. *Large Scale C++ Software Design*, 1-95. (Addison-Wesley 1995)

11. Mackinnon, T, Freeman, S and Craig, P. *Endo-Testing: Unit Testing with Mock Objects* in *Extreme Programming Examined* (Addison-Wesley 2001).

12. Martin, Robert C. *Can XP be used with C++*. (Object Mentor, 2000) http://www.objectmentor.com/xp/xpwithc.html

13. ObjectMentor. http://www.objectmentor.com/

14. Workshare Technology. http://www.workshare.com/