JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2008**

Vol. 7, No. 6, July-August 2008

The Legacy and Liability of Object Technology
The Dark Side of OO

Dave Thomas

LARGE OO LEGACIES REVEAL THE DARK SIDE

It took until the mid 80s for the pioneering ideas of object-orientation from Simula 67 to appear in industrial languages such as Objective-C, C++, Smalltalk, Eiffel and until the mid 90s for OO to become mainstream with Java and C#, UML etc. Now, even Cobol and Fortran have modern dialects that are OO. Java has reportedly passed COBOL in terms of usage. Many companies have the majority of their legacy code in C++ or Java. Since COBOL has for many years been the dominant legacy language these companies are now looking at their OO legacy with increasing concerns about their ability to cope with it.

Many of the most successful products currently in use were developed using C and then C++, while others are written all or substantially in Java.

OO provides great potential benefits of increased modularity through components and reduced code bulk through reuse, however, little is said about the downsides of object technology used inappropriately. Unfortunately, a disproportionate number of beautiful 80s and 90s products and applications have morphed into complex, scary legacies that hold their owners hostage, limiting their ability to deliver timely new value to their customers despite significantly increased investment in people and tools. More and more time is spent delivering less and less.

The OO generation is learning that despite having a great technology, their once wonderful code is turning into a complex legacy. Almost every organization seeking to adopt new practices such as TDD faces their own code mountain with numerous dangerous caves where both new and old developers fear to enter.

OO Legacy code is particularly difficult to deal with due to the acute lack of modularity and the additional dependencies introduced by open frameworks. These dependencies make Builds slow and error prone and they make refactoring code difficult and risky.

Beyond that, much OO code has been written by developers with little appreciation of OO design and development disciplines leading to cut and paste reuse, and code bulk due to casual even unintended inclusion of frameworks or libraries.

Clever programs leverage dynamic loading and reflective features, which unfortunately often increases the difficulty of understanding legacy code.

FRAMEWORK HELL

Many frameworks force developers to work inside the framework rather than just through the framework APIs. Often they require the developer to develop an intimate knowledge of framework internals to instantiate and extend the framework. Indeed, AOP has found traction in part because it allows one to modularize such framework modifications. Many framework APIs are rushed through too quickly to establish a control position in the market or to meet the need for architecture to get development going before things have been properly thought through.

Finally, many frameworks are enabled by other frameworks but often the ease of using another framework forces a huge amount of additional code to be unnecessarily included. Eclipse has a very nice plugin mechanism, however, many lazy developers, seeing something useful in other plugins, just include them all, rather than selecting the appropriate classes and methods that deliver the small amount of functionality needed. Framework dependencies have become even more problematic with the large number of open source frameworks and Java and MS.Net framework upgrades. Each year developers must decide if they should maintain the their current code, which deviates from this years and next years frameworks, or if they should take the business and technical risk of migrating to newer versions of their current frameworks or their even newer alternatives. E.g. Swing, AWT, SWT, Jaces, Ajax or EJB1,2,3 or Spring, ORM mapping etc.

LACK OF MODULARITY - HEY! WHERE ARE THE INTERFACES?

Modularity relies on well defined and explicit interfaces. It is shocking, given encapsulation is the hallmark of OO, that most legacy application are so lacking when it comes to well defined interfaces. Even though Java and C# have explicit support for interfaces they are seldom used and when they are often not used properly. It is telling that only recently have speakers and authors started talking about APIs.

An API is a stable well defined interface that a client can count on without needing to know the intimacies of the provider code it is using. It is a contract between the interface provider and their clients.

Unfortunately far too much legacy code lacks well managed APIs, and even more so how little API testing is done. This lack of clean, well defined interfaces results in an ugly tangle of legacy code appropriately called a tar ball. It is really a zipped tar baby, touch it and it will stick to everything you touch it with.

BUILD HELL

Many large OO systems have such complex dependencies, especially in C++, that build scripts are like mythical writings on cave walls, which none dare change lest the build be broken forever. The Java Build story has better language and tool support, however, the massive dependencies induced by Java frameworks and configuration scripts are quickly challenging the C++ world. The lack of modularity limits the ability to do frequent builds, greatly increasing the risks of making changes of legacy code makes "the build" the first monster of OO legacy development. Fortunately, in most cases, a small 3 – 5 person team can substantially improve build modularity and performance in 1 – 2 months but most organizations fail to see the huge benefit until it is way too late.

WORKING WITH LEGACY CODE

We are beginning to understand how to work with Legacy OO code, but only just so. Mike Feathers' book outlines how to tease apart classes and use TDD to pick apart an existing program but Mike himself readily admits that this is very hard for a large legacy code base. Unlike many Agilists, Mike points out the critical importance of carefully defining APIs and the naivety of thinking that one can just refactor the code mountain using your favorite IDE. Working with legacy code currently takes brave experts who are not afraid to jump into the code mountain knowing they may die in a cave before succeeding. Legacy refactoring is an important constructive intervention but it can often take 3 – 4 months with a top notch team to make a significant progress on a code mountain. This important entropy reduction activity seldom gets the investment until it is too late.

Recently there have been numerous tools to help understanding OO legacy code. Klocworks and Software Tomograph provide graphical visualizations of the architecture and code. Structure 101 and Lattix both produce dependency structure matrices (DSM) to illustrate and manage complex dependencies. Both the visualizations and the matrices will very quickly scare anyone when you point them at your unsuspecting legacy code base. Unfortunately, none of these tools is yet up to dealing with the really large legacies of mixed C, C++, Java, QL etc that are typically found in large legacy systems, but at least there is some light to see into the legacy tar ball.

However, even with these tools restructuring the legacy code base remains wizard's work. This is even more challenging if the code base is moving underneath you.

About the author

Dave Thomas is cofounder/chairman of Bedarra Research Labs (www.bedarra.com), www.Online-Learning.com and the Open Augment Consortium (www.openaugment.org) and a founding director of the Agile Alliance (www.agilealliance.com). He is an adjunct research professor at Carleton University, Canada and the University of Queensland, Australia. Dave is the founder and past CEO of Object Technology International (www.oti.com) creator of the Eclipse IDE

Platform, IBM VisualAge for Smalltalk, for Java, and MicroEdition for embedded systems. Contact him at dave@bedarra.com or www.davethomas.net.