JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2006**

Vol. 5, No. 7, September-October 2006

Agile Evolution
Towards The Continuous Improvement of Legacy Software

Dave Thomas, Bedarra Research Labs

1 AGILE EVOLUTION - A FRESH APPROACH TO SOFTWARE MAINTENANCE

The vast majority of improvements in software development tools and techniques focus on the development of new applications or components for greenfield projects. Unfortunately, this often means that organizations with substantial assets developed using older, once popular and accepted technologies and methods cannot easily migrate to new applications or components. Much has been written about using Agile development in the context of greenfield development; however, software experts increasingly see that Agile practices are well suited to software evolution. This isn't completely surprising since Agile methods stress the importance of people, incremental development, risk reduction, and continuous testing – factors which all contribute to effective software evolution.

2 SUCCESSFUL SOFTWARE ALWAYS LEAVES A LEGACY!

It is important to note that so-called Legacy software is at the heart of almost every major product and commercial enterprise. Each successful product produces a major legacy! Each successful technology produces a major legacy!

Even major software vendors have difficulty hiring developers to work on their massive C++ code bases which were state of the art only a decade before. Java, now past the age of 10, already has a legacy of applications and products. Furthermore, frequent changes in Java/C# frameworks and languages create legacy code in shorter periods of time as what was hot becomes orphaned legacy code.

Yet most of our industry is focused on the creation of new software, often using new technologies, with little regard for the critical need to improve and enhance existing software, which accounts for 70% of software development.

Cite this column as follows: Dave Thomas, "Agile Evolution -Towards The Continuous Improvement of Legacy Software", in *Journal of Object Technology*, vol. 5, no. 7, September-October 2006, pp. 19-26
http://www.jot.fm/issues/issue_2006_09/column2

3 WE CAN'T JUST REWRITE IT ALL AND THROW THE LEGACY CODE AWAY!

Every new generation argues that the only solution is to rewrite the legacy software using their new technology. However, there is simply too much software in use to rewrite it all, for both economic as well as technical reasons. Many older systems provide best-of-breed, strong, specific solutions with lower total cost of ownership, which as yet have not been demonstrated using newer technology. Unfortunately, we often don't understand the design choices for the original system and/or we underestimate the maturity of the new technology; hence the industry track record for major rewrites is dismal. Most such efforts have been way over budget and late, with many being cancelled outright.

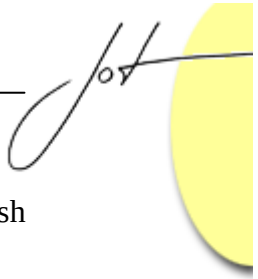
This is why experienced CIO/CTOs seek to apply a new technology where it brings business value and evolve the legacy code base only where appropriate. This is a difficult challenge when vendors, consultants and new hires promise the benefits of switching technology. Even where the new-technology-based applications do succeed, they too become a legacy. Have you tried hiring top developers to work on your legacy C++ code base lately?

4 SOFTWARE EVOLUTION IS CHALLENGING!

The entropy of a software asset increases substantially after its 3rd to 5th version. This is a function of both code bulk and interacting changes made by concurrent teams which often erodes the architecture. Many major applications and products were developed 10, 20 or even 30 years ago and contain hundreds of thousands to millions of lines of code. In all cases the platform, methodology, programming languages and tools will be different, often containing mixes from 2 or 3 generations of technology. The loss of key people over time also contributes greatly to the complexity of code maintenance. Despite the best documentation practices, typically only a few key people truly understand the inner workings of most systems. In extreme cases there may be few or no human experts who know the code base, and there may not even be complete binary or source code.

Unfortunately it seems that graduates only learn about software technology and practices that exist four years plus or minus their graduation. This leaves corporations with incredible problems spanning isolated techno cultures e.g. COBOL/PLI to C; 4GL/VB/Smalltalk/C++ to Java/C#; Java/C# to LAMP etc. Each techno culture uses different methods, languages and tools.

Test coverage will also vary considerably. Most client server applications will require testing through the GUI with fragile platform UI dependent scripts. Many products will have multiple code bases to support mainframe, client server and web technology, often supported by geographically distributed development teams. Finally the applications and products often contain unique creative solutions to domain-specific problems such as rule engines and application specific code generators supported by special purpose tools and runtimes.



Given these substantial challenges, it is natural for businesses to consider a fresh approach.

5 RETHINKING PROFESSIONAL SKILLS FOR SOFTWARE EVOLUTION

In many companies, there is still an attitude that new development is difficult and needs the top developers, perhaps even outside developers. The less experienced or less skillful developers are then assigned to perform the system evolution. Oftentimes, there is also little training or mentoring allocated to the evolution effort. Most companies assume that on-the-job training will suffice. The standard practice is to promise that a developer assigned to an evolution project can only move to a new project once they have trained a usually less experienced replacement.

In his Dahl-Nygaard Prize ECOOP 2006 keynote (http://www.emn.fr/x-info/ecoop2006/keynote3.html), Ralph Johnson clearly and effectively articulated the need to change the emphasis in education, research and professional practice to focus on working with existing software rather than working on creating new software from scratch. The skills and practices required to successfully enhance such legacy software differ significantly from those used to design and develop new applications. Ralph argues that the emphasis should be on *Discovery* and *Transformation* [1] rather than classical *Design* and *Development*. Further, software needs to be written in a literate style that exposes the key artifacts, including requirements and associated acceptance tests, as well as architecture, design and implementation. Literate programming emphasizes code reading over code writing, a practice which has long been argued as best practice for educating software developers.

Discovery

The older the code base and the more mission critical the application, the more difficult it is to maintain or enhance it. The software takes on the qualities of a dangerous, dark, unexplored cave, complete with cryptic symbols in design notation and scrolls in older dialects of programming languages. The rumor mill warns of dangerous caverns in the software where few developers have succeeded in making changes that work. In order to reduce the risk of failure, the bug fixes and change requests for these modules are slowed to a crawl and only the bravest developers dare to make major changes. While some of the original development team and their expertise may remain, their own knowledge will be incomplete. Unfortunately, sometimes the resident expert, perhaps fearing a loss of employment or stature, also chooses to become a less than cooperative guide.

Working with large legacy software always involves risks of the unknown for new developers. This makes it essential that development begin with an in-depth discovery activity to increase their understanding of the code base. Discovery combines the stories obtained from experienced developers and customers with knowledge gained by analyzing the code and associated documentation and test cases. Efforts in reverse

engineering (http://www.iam.unibe.ch/~scg/cgi-bin/scgbib.cgi?query=famoos) have adapted language technologies such as static (http://www.klocwork.com/products/klocworkk7.asp) and dynamic control and data flow analysis to mine code for information seeking to identify the hidden dependencies which impact the ability improve code.

Transformation

Evolving a large code base is best viewed as a careful and systematic transformation in which each change is carefully tested before moving to the next. Such transformations go well beyond refactoring, which technically is defined as equivalence preserving, to major program restructuring. Further successful restructuring, like successful refactoring, requires the support of comprehensive unit [3] and acceptance tests. A critical activity in working with legacy code is bringing the test coverage and code modularity to the point where transformations can be replied frequently with confidence.

To date we have good tools and techniques for analysis and for refactoring (http://www.refactory.com/tools.html) but we are only beginning to understand how to programmatically query programs and transform them. Unfortunately most current tools only operate in modern single language IDEs, leaving those with legacy languages without tool support. There are numerous reverse engineering tools but most of these have been developed for quick and dirty language migrations as part of a services engagement rather than robust life cycle evolution.

6 MAINTENANCE AS AGILE EVOLUTION – WORKING WITH LEGACY CODE

Recently several organizations have been considering agile techniques for their maintenance process. They quickly identified many similarities between agile concepts and their own software maintenance experiences as shown in the table below. The Agile development process includes the tools and techniques required to effectively deal with the common issues organizations face when maintaining software.

Traditional Software Maintenance	Agile Development
Understanding the essence of the system	Metaphor and Stories
Customer defect and feature requests	Customer and Stories
Test suites	Test first, Unit test, Acceptance test
Regression testing	Continuous integration and test
Fixes and “Dot” Release	Small Increments
Change Management	Scrum, Planning Game, Stand Up

	Meetings
Reverse Engineering, Debugging, Critical Patches	Pair Programming
Code Reorganization, Code Simplification	Refactoring

7 LEGACY TALES

How can Agile development address the challenge of passing on an understanding of complex software? Many systems have lots and lots of use cases, massive requirements and design documentation. Unfortunately, these documents often fail to communicate the essence of the system to developers responsible for system evolution. A second problem is that existing documentation is not up-to-date and often has gaps in important areas. One of the best ways to develop a shared essence is through real storytellers who have lived part of their lives "in" the system. Recently there has been a great resurgence in the use of story telling as means for preserving corporate memory [3].

For many years, young recruits who joined companies like Nortel and IBM were taken through intensive boot camps during which they were often exposed to massive amounts of source code along with the wisdom of one or more of the key architects or developers. These developers communicated the essence of the systems to new developers. While the stories they imparted were imprecise and sometimes inaccurate, they frequently provided sufficient context to enable new developers to work on many parts of the company's products. They learned quickly where the minefields lay, why the system didn't work they way they thought it should, or even the way it was documented, etc.

This essential understanding of how the system works is still best communicated by knowledgeable storytellers to inquisitive learners who are climbing difficult code mountains. The "big story" is what XP calls Metaphor and the "little stories" are story cards or use cases. These practical descriptions slice through the system exposing its essence. Passing on an understanding of the essence of the system is an important part of software maintenance that can be facilitated through the use of Agile development. Dialog beats documentation every time.

8 CUSTOMERS' CHANGE REQUESTS

One of the challenges in new development is finding the right customer and building the right thing. This is a critical success factor for the organization and key to Agile development. It can be a particular challenge when building a shrink-wrapped product for a general external market where customer requirements can often only be derived from surrogate customers, focus groups and beta customers.

In addition to defining the requirements for new development, Agile development can address the challenge of requirements analysis for established products. An established product has lots of customers who have identified defects as well as features which they feel are missing from the product. The customer feedback is systematically collected through problem tracking systems in which customers prioritize the important problems and features. These change requests are used to drive the evolution of the system. Unlike initial requirements, which are often only very high-level, problem reports and feature enhancement requests are usually very specific. In many organizations, customer focus groups are used to represent the customer by organizing and prioritizing development activities in collaboration with developers. Agile evolution therefore has the key ingredients for development stories in the problem reporting system and can engage the team in scrums, the planning game associated with standard agile development.

9 PAIR DEVELOPMENT: SHARE THE RISK AND THE RETURN

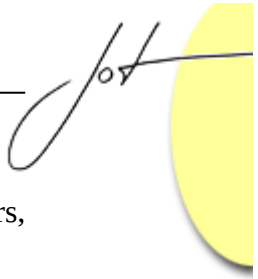
Effective software maintenance requires a way for developers to reverse engineer, debug and apply fixes in order to make critical software updates. Agile developers can effectively use Pair Programming in these situations. Note that Pair Programming in general refers to more than just coding, and in fact also provides the well known benefits of code and design inspections/reviews, test case development, code reorganization and refactoring applied from the point of development to deployment.

Software developers working on large existing systems often are required to make changes to software that they don't understand. They need to reverse engineer legacy code to identify the defective modules, determine which modules need to be changed and determine the order in which to make the changes. Given the uncertainty, these reverse engineering and debugging activities are often done by pairs of developers.

For example, mission critical applications changes often need to be made to a system running live in a customer's location. A common practice among seasoned maintenance developers is to approach such critical changes in pairs, relying on the human redundancy to reduce error. Similarly, design and code reviews are well known ways to eliminate bugs by "staring them out". The benefits of an additional pair of hands and eyes include the ability to provide feedback, increased confidence with estimates, as well as reduced risk and improved quality.

10 REGRESSION TESTING AND CONTINUOUS INTEGRATION

Regression testing is a key factor in effective software maintenance. The use of Agile development allows for more frequent regression testing through continuous integration. For many years, strict regression testing has been the mantra of a successful maintenance organization. Recent efforts in the Agile community to develop efficient continuous integration approaches can easily be embraced by regression testers to allow them to



increase the frequency of regression testing through the use of dedicated test servers, mock data bases, etc.

11 MAINTENANCE AS A DEVELOPMENT ACTIVITY – INCREMENTAL REENGINEERING AND DEVELOPMENT

> *"The most amazing thing was that I learned as much in a maintenance month three years into the project as I did in a programming month before release. In fact, I came to think of the practices we'd put in place as a safety net that made maintenance a safe place to learn. The very best objects in the product weren't discovered, couldn't even be discovered, until we had the full richness of the (agile) maintenance environment in place."*
>
> *-Ward Cunningham [4].*

How can Agile development be used to deal with large systems that have well-known modules, which for one reason or another are known to contain a substantial number of defects? Historically, these modules are often the least understood by developers, hence each new fix or feature is often approached with great trepidation. In some cases it may be very difficult to make timely releases due to concerns about touching the core components of the system. The time-honored solution to these problems is to incrementally replace the faulty components – one component at a time. This approach is often called developing your way out of maintenance.

The Agile/XP approach to this problem is to develop stories and then test cases to try to ascertain the correct operation of this component. Theses test cases will include existing regression test cases as well as new test cases identified by developers or customers. Once one has sufficient test cases, changes can be safely made to the defective module or it can be replaced, both without fear of unknown side effects. It is important that both management and developers gain confidence with this approach and move slowly, first making changes in single functions or data structures rather than making wholesale change to whole classes or hierarchies.

12 REFACTORING TO REDUCE DEFECTS AND EASE CHANGES

Code reorganization and simplification is an important software maintenance activity that can be addressed through refactoring in Agile development. In general, any activity that substantially reduces the number of lines of code and/or improves the readability will reduce defects and facilitate future changes. This is one of the primary goals of refactoring, which seeks to reduce duplication of code, simplify overly complex code and introduce improved names as well as class and method organization. While these changes can be implemented with an editor, we strongly recommend industrial strength refactoring tools if they can be obtained. These tools support the developers in making and unmaking small changes and reduce the risk associated with a refactoring effort. One

of course should not attempt major refactorings unless the test cases provide full coverage of the component to be refactored.

13 AGILE EVOLUTION - A POSITIVE ALTERNATIVE APPROACH TO SOFTWARE MAINTENANCE

The essence of Agile evolution is to gradually transform a typically conservative, risky and unattractive activity into a positive and proactive development activity. We argue that the impact of Agile practices on the evolution of large software systems will be even more important than its impact on new application development. Further, by adopting similar practices throughout the life cycle, the schism between new development and maintenance can be reduced.

REFERENCES

[1] Ralph Johnson, Software Development Is Program Transformation, http://www.cincomsmalltalk.com/userblogs/ralph/blogView?showComments=true&entry=3319915651

[2] Adele Goldberg , Story Telling: Collaborative Software Engineering, Journal of Object Technology, http://www.jot.fm/issues/issue_2002_05/column1

[3] Michael Feathers, Working Effectively with Legacy Code, Prentice Hall, 2004

[4] Ward Cunningham Personal Communication, http://fit.c2.com/wiki.cgi?WhatsWhat

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.