Agile and Lean Service-Oriented Development:

Foundations, Theory, and Practice

Xiaofeng Wang
Free University of Bozen/Bolzano, Italy

Nour Ali
Lero- The Irish Software Engineering Research Centre, University of Limerick, Ireland

Isidro Ramos
Valencia University of Technology

Richard Vidgen
Hull University Business School, UK

A volume in the Advances in Computer and Electrical Engineering (ACEE) Book Series

Information Science REFERENCE

Managing Director:	Lindsay Johnston
Editorial Director:	Joel Gamon
Book Production Manager:	Jennifer Romanchak
Publishing Systems Analyst:	Adrienne Freeland
Development Editor:	Hannah Abelbeck
Assistant Acquisitions Editor:	Kayla Wolfe
Typesetter:	Nicole Sparano
Cover Design:	Nick Newcomer

Published in the United States of America by
Information Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

Library of Congress Cataloging-in-Publication Data

Agile and lean service-oriented development: foundations, theory, and practice / Xiaofeng Wang ... [et al.], editors.
p. cm.
Includes bibliographical references and index.
Summary: “This book explores the groundwork of service-oriented and agile and lean development and the conceptual basis and experimental evidences for the combination of the two approaches”-- Provided by publisher.
ISBN 978-1-4666-2503-7 (hardcover) -- ISBN 978-1-4666-2504-4 (ebook) -- ISBN 978-1-4666-2505-1 (print & perpetual access) 1. Computer software--Development. 2. Agile software development. 3. Lean manufacturing. I. Wang, Xiaofeng, 1971-
QA76.76.D47A379824 2013
005.1--dc23

2012023354

This book is published in the IGI Global book series Advances in Computer and Electrical Engineering (ACEE) Book Series (ISSN: 2327-039X; eISSN: 2327-0403)

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Chapter 14
Test Driven Decomposition of Legacy Systems into Services

David Parsons
Massey University, New Zealand

Manfred Lange
EFI, New Zealand

ABSTRACT

A number of questions have been raised by both practitioners and researchers regarding the compatibility of service oriented architectures and agile methods. These are compounded when both approaches are combined to maintain and migrate legacy systems. In particular, where test driven development is practiced as a core component of an agile development process, both legacy systems and service oriented architectures can present an incongruous set of development challenges. In this chapter, the authors provide experience reports on how legacy systems have been adapted to an agile, test driven development context by a process of decomposition into testable services. They describe two domains and technology contexts where automated agile testing at multiple interface layers has improved both quality of service and functionality.

INTRODUCTION

A service oriented architecture (SOA) is a collection of self-contained services that communicate with each other, passing data or coordinating some activity. Each service has a provider and one or more consumers. The primary aim of a service is to support business processes; implementing services is not an end in itself, but rather a means to deliver agile systems to support a business (Wilkes & Veryard, 2004). Nevertheless beyond this core requirement there are a number of general principles that can be applied to service oriented architectures. These principles include that a service should have explicit boundaries, be based on schemas and wire formats, not classes and APIs, and be policy-driven, autonomous, document-oriented, loosely coupled, standards-compliant,

DOI: 10.4018/978-1-4666-2503-7.ch014

vendor independent and metadata-driven (Tilkov, 2007). Unfortunately few of these criteria can easily be applied to the interface of a legacy system. In particular, the boundaries and coupling of a legacy system may be problematic. However the value of services in the context of legacy systems is that services do not have to be brand new. They can be fragments of old applications that were adapted and wrapped, or combinations of new and legacy code (Papazoglou & van den Heuvel, 2007).

Though we can perhaps bridge legacy systems and service oriented architectures, there are other questions raised about how effectively an agile software development approach can be applied when working with these systems. Service oriented and agile approaches may conflict in areas such as architecture, team organisation and feedback (Elssamadisy, 2007). It has also been suggested that service orientation encourages upfront architecture while agile does not, though service oriented architectures can be incrementally introduced into an existing system. On the other hand it does pay to understand the strategic direction in advance, including layers and main system components such as infrastructure services. There are also potential conflicts in terms of organizing teams, where a service oriented approach encourages teams to split along functional lines while agile approaches encourage cross-functional teams. This can be a challenge in maintaining perspectives, for example ensuring that business logic is not added to the codified structure of a schema or document. In terms of feedback, a service oriented approach does not have the same focus on frequent feedback at both a technical and personal level. There may also be tensions in terms of the level of ceremony required for different types of software development. Chung et al. (2008) suggest that the mainstream agile methods focus on forward engineering new systems, and claim that a more formal approach related to the Unified Process is required to integrate legacy systems and services. Agile assumptions about delivering potentially shippable code at the end of every iteration are also challenged when dealing with live external services that may require extensive performance and stress testing (Puleio, 2006).

Despite these reservations, the experience reports in this chapter offer a different perspective to some degree. Perhaps the key aspect of the agile approach is flexibility or adaptability, so it is the flexibility aspect of services that assists agile transformation (Sucharov, 2007). Flexibility is required in order to address ever-changing market and customer expectations, and the move to an agile service oriented approach provides the capability to adapt to these ever-changing expectations.

In addressing both the mapping between legacy systems and service oriented architecture, and the relationship between both of these and agile methods, it is testing that can provide the overall glue to the process, providing that a test driven approach is taken. Test driven development drives design, as well as providing unit tests, and therefore enables us to define the interfaces between services and legacy systems. Sharing tests between client and service developers enables the two layers to be developed and maintained effectively. The internal design of services can also be performed in an agile way, once the outer interface to the service has been defined. In particular the interface concept already goes a long way towards making a legacy system more agile, for example by being able to componentize functionality into services, making testing easier or enabling it in the first place, or by plugging in different flavours of infrastructure services (security, transactions, logging, etc.) Tests provide a way of specifying the boundaries of vertical slices of functionality that might be decomposed into services. They also provide a way of defining the service's external contracts prior to those contracts being implemented. A unit testing philosophy also emphasizes testing services in isolation. The service under test can be provided with meaningful mocks for other services that it may depend on.

In the organisations used as the basis for this experience report, the generic approach taken was to gradually identify legacy features that could be specified by a set of interface tests, and then extract these into independent services. Though not initially test driven, since the module functionality was already in place, the extracted services could then be developed in a test driven manner as they were subsequently refactored. Feature identification was driven both by the requirements of other product components and services and customers who wanted to integrate with in-house or 3rd party systems. Therefore it was driven by external requirements, rather than trying to identify features in isolation. In addition, the development teams also used a service oriented approach for all new functionality, thus creating a hybrid architecture between a 2-tier legacy and a 3-tier service oriented architecture. From another perspective, testability was enhanced by an architecture that allowed interoperability with plugins for integrating external services. Again, plugins could be individually tested. This chapter provides a number of practical experiences and lessons learned in using tests to drive the wrapping of legacy systems into discrete, testable services.

WORKING WITH LEGACY CODE

Legacy systems are commonplace in software supported industries, and these systems can take many forms. They may be acceptable to their regular users even when they have significant bugs, as users become familiar with the required workarounds, and are aware of current 'characteristics' of the system. Sometimes customers do not characterise issues as bugs, but rather are aware of what they regard as usability problems. On the other hand the cost of customer-impacting errors on systems that are already deployed is high (Feathers, 2002). Given that legacy systems are typically the result of significant investment, and probably in live use, development teams considering tacking underlying issues such as large, complex and flawed legacy systems need to balance a number of trade-offs.

Thomas (2006) describes the problem of large systems with defective but poorly understood modules, making developers approach each fix or feature with great trepidation. As he indicates, the general solution to this kind of problem is to incrementally replace the faulty components, one component at a time. There are, however a number of problems with this approach. Legacy systems sometimes include legacy testing frameworks that may themselves need to be refactored or replaced before progress can be made (Puleio, 2006). It can also be difficult to find any inflection points where parts of the legacy system can be broken apart. Feathers (2002) refers to techniques such as dependency inversion to create such inflection points, but this is only possible if the legacy system is written in a language that makes such interventions feasible. Sometimes the only inflection point that you can find is the system boundary which can encompass the GUI, calls to other external libraries and the database (Feathers, 2002). However simply adding a large wrapper around monolithic legacy systems does not help address possible underlying issues of on-going maintenance costs and lack of flexibility (Sucharov, 2007). Furthermore having large monolithic blocks of code with many service interfaces interacting with them will not yield the full benefits of a service oriented architecture, where conceptually each single service can be replaced with a different implementation without the need to change any other part of the system.

Given the problems of trying to tackle monolithic systems, and the likelihood that adding a service wrapper alone does not address underlying problems, there needs to be a systematic approach to converting legacy systems to robust, maintainable and modular service oriented systems. This requires techniques of discovery and transformation rather than classical design and development. The first step in the process is to bring test coverage and code modularity to the point where transformations can be applied frequently and with confidence (Thomas, 2006).

Visaggio (2001) asserts that when a legacy system has poor quality it is said to be *old* or *aged*. This perspective emphasises the quality issues in legacy systems. Feathers' (2002) definition of legacy code also relates to quality, but from a different perspective, that it is code without tests. The fundamental assumption, therefore, is that to bring legacy code into line with current development we must start with a complete test suite, which will both externalise the current quality of the system and provide a lever for improving it. Feathers also lays out a general strategy for the management of legacy code for which tests are essential, namely:

1. **Identify Change Points:** Decide where the changes should be made. Ideally the approach chosen should be the one that requires the fewest changes.
2. **Find an Inflection Point:** Identify a narrow interface to a set of classes where any changes to those classes can be detected; like a façade to part of the system.
3. **Add a Test Covering to the Inflection Point:** This means writing a set of tests for the interface identified above. Of course this cannot be test driven since the unit under test already exists. The tests for the inflection point should be unit tests, not integration tests, so both external and internal dependencies may have to be broken. This phase consists of the following steps:
   a. **Break External Dependencies:** It may be necessary to break dependencies on other components. Feathers suggests using dependency inversion, where the language of the legacy system allows this. Otherwise some kind of mocking layer would have to be used.
   b. **Break Internal Dependencies:** Some code under test may create concrete components that are outside the scope of the units under test. Feathers suggests a combination of subclasses and null classes to avoid these orthogonal components being created.
   c. **Write Tests:** The tests need to cover the inflection point as comprehensively as possible. Starting with boundary values is appropriate.
4. **Make Changes:** After an initial test covering is in place, further tests should become evident as changes are made to the legacy system. This takes the process into a more test driven mode as each change can be preceded by writing the test for the change.
5. **Refactor the Covered Code:** As is normal practice in test driven development, each change to the code must first pass the test, and then be refactored to improve its design, while still passing the test.

While this strategy provides a general guideline for the steps of a test driven process for working with legacy code, we need to also take into consideration a number of contextual issues such as commercial aspects. Perhaps the most significant of these is the extent to which this is seen as a process that aims to eventually replace the legacy codebase, or whether it becomes more of a mechanism to better control and leverage legacy systems. The extent to which the legacy system is changed and refactored depends on issues such as the current reliability and maintainability of the legacy system, the extent of new requirements, and the potential costs versus the payback of replacing legacy components.

EXPERIENCE REPORTS: FROM LEGACY SYSTEMS TO AGILE SERVICES

This article reflects on some real world experiences of a number of the issues introduced thus far, in the context of two different types of legacy systems, written in different languages in different domains. In both cases the lessons learned are similar. However experience has shown the limitations of over-simplistic models of the use of services as a half-way house to refactoring underlying systems. Such models advocate the use of a service wrapper to encapsulate a legacy system as the first step to re-engineering or replacing that system. In many cases, however, for practical, financial and pragmatic reasons, it is enough to build a testable service wrapper around a legacy system without extensive rewriting or refactoring. Thus we should not approach the wrapping of legacy systems with an assumption that the system itself will be extensively modified. Rather, the service wrapper enables the system to be safely refactored in cases where business requirements for such changes are identified.

The first experience report, which is relatively brief, may be regarded as setting the scene for the second, in that it provided the context for a service oriented approach that began by making conventional assumptions about the gradual replacement of a legacy system by means of inflection points and testable wrappers, as described by Feathers (2002). The second report, which is somewhat more detailed and elaborated, may be regarded as putting into practice some of the learning that took place in the context of the first report. It certainly takes a more circumspect view of the process within which legacy code may be changed and refactored, driven by pragmatic requirements and considerations.

Experience Report 1: The Green Pen and the Blue Code

The first system described here was in the context of a customer relationship management system for the utility industry. In this case the legacy system had a billing engine written in a 4GL as its core component, with some Java connectors for integration with other functionality implemented in Java. In order to render this system more easily interoperable with client facing systems, it was originally intended to replace this system with a full Java implementation. However this task was made more complicated because the legacy system was still under maintenance, so was a moving target in terms of attempting to replace it with a new system. Therefore, despite an original intent to replace the old system, this was adapted over time into a more pragmatic approach. To provide the development team with a strategic direction, the vision of a target system design was created. This was referred to as the 'green' system design, while the legacy system was coded 'blue'. The team adopted the metaphor of the 'green pen' based on this; Everyone gets a green pen. Colour in as much as you can, but no more than makes business sense. Practically, it meant that when the 'green' pen was used the resulting design and implementation was expected to be in line with the target design. The colour coded approach also recognised some of the more valued features of the 'blue' legacy system, including its transactional support, which was exposed via Java connectors to enable the integration of transactions across the blue and green components. This aspect of the system enabled a transaction started in the client facing layer to propagate into the billing engine. This represented a business critical feature of the legacy system that also assisted the service orientation of that feature via its external connectors. In terms of the metaphor the introduction of this cross-cutting transaction mechanism itself was an example of using the 'green' pen as it was designed and implemented in line with the target system design, but provided a service view of the 'blue' functionality in the legacy system.

Despite the limitations of working with a live legacy system under maintenance, and the impracticality of replacing this system, the overall approach of developing a service oriented wrapper enabled the green and the blue code to work effectively together within an agile development approach. In time, with the green pen colouring in all the external interfaces, the whole system looked service oriented, regardless of the legacy components behind the service layer. All the blue code eventually had a green interface. In addition, new features were developed completely using the green pen. The result was a hybrid system design proving that a legacy system (the 'blue' parts) and a service oriented architecture (as in the 'green' design) can co-exist.

The main driver for compromise in the extent of replacing legacy systems turns out to be a combination of marketing and risk, so in the end it comes down to commercial considerations. What sells is what can be well presented to decision makers, which is not necessarily the same as daily functionality for users. Sometimes a new wrapping (e.g. updated user interface) is all that is required to present a product or a feature as 'new' to the market. Quality of itself is not the key selling point. Given that quality is not an absolute, only when software failures affect a customer's bottom line does quality become the most important issue. However the risk to quality of rewriting the entire billing engine was also a significant consideration.

Experience in the first context revealed that the most important factor in any approach to moving a project forward from a legacy past is to choose the overall direction. The original intent to replace the legacy system turned out, for various reasons, not to be economically practical, nor indeed particularly valuable. An embodiment, perhaps, of the agile phrase 'YAGNI' ('you ain't gonna need it'). This experience underlined the concept that a simple replacement of a legacy system with new code is often not the most realistic or economical approach. Therefore a disciplined way of bringing legacy systems into more managed, service oriented architectures is important, along with an agile mindset that promotes 'the simplest thing that could possibly work' as the default technical architecture, maintained and refactored as needed. This was the approach that was brought forward into the second experience report.

Experience Report 2: The Big Picture – Interfaces over Business Logic

The second domain described in this article is production management for the printing industry. In this case, the legacy system was written in C++ with some portions written in C#. It lacked a sufficiently clear strategic design, with code written from disparate perspectives. There were also known problems with the existing codebase, with customers experiencing problems that could be serious enough to affect their productivity. However, despite these issues, given the experience at the previous organisation, replacing the entire existing legacy system was not considered an option. Instead, the first priority was to regain control over the system by introducing adequate test coverage so that a solid foundation could be created for future development.

The system had been developed over a long period of time by many people, so although there were islands of consistency in the way the code was written, there were also many inconsistencies, including different approaches to C++ metadata, and seven different ways of any two product components talking to each other (database, files, COM etc.) The most challenging feature was the management of database schema upgrades which were part of the native client, thus occasionally causing avoidable issues in live deployments. As a first step to taking the system into a service oriented architecture, the Web UI was moved from a 2-tier to a 3-tier architecture, at the same time introducing an application server as a new product component. Upgrade code for the database was moved to the application server installer to solve the brittle update problem, replacing it with a solid and reliable mechanism.

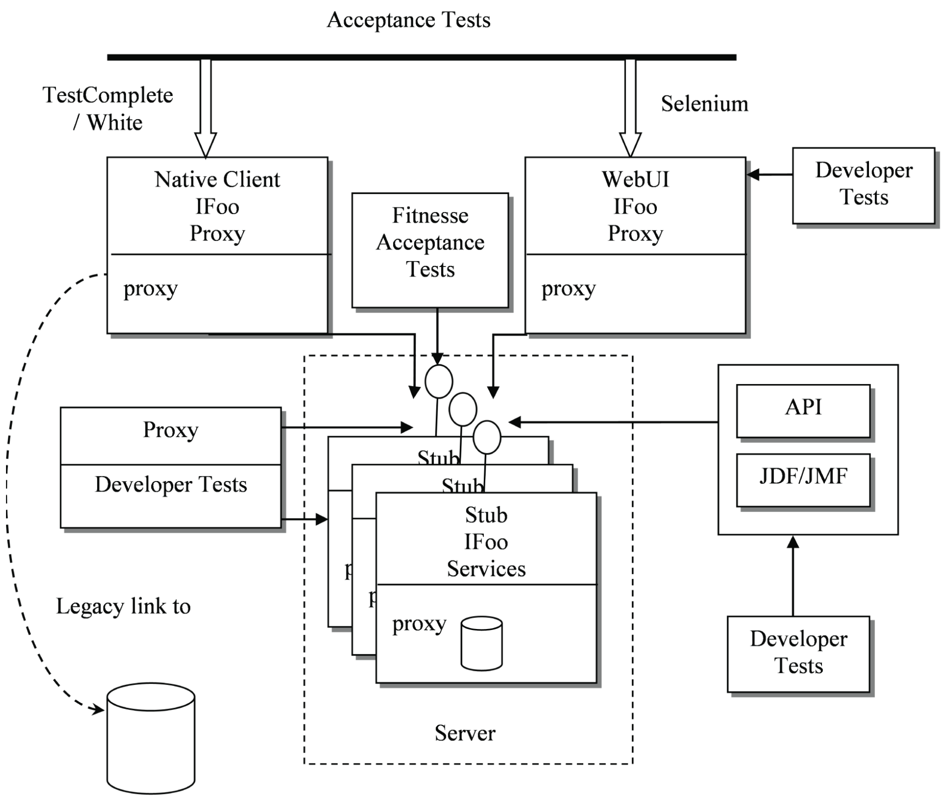
Over time, the legacy system has been encapsulated behind a series of service wrappers, each one with comprehensive test suites. Legacy unmanaged C++ has become managed, with new code in C#, but all residual unmanaged code behind managed wrappers. For the native client, a hybrid approach was established to allow features using services to co-exist with features using a 2-tier approach, i.e. talking directly to the database.

Although there was no overall requirement to re-engineer the entire legacy system, there were business drivers that triggered steps 6 and 7 in Feather's strategy for legacy code; to fix and refactor. The time when this most urgently had to be done was when the code had failed, particularly if a customer suffered a major software issue. When this happened with unmanaged legacy code, a testable wrapper was added to enable the code to be fixed and refactored. As a result of having these testable wrappers around the legacy code, the number of these reactive emergency fixes having to be done inside legacy code dropped to practically zero. In the longer term process of tackling less critical legacy issues, there are on-going quality improvements of 10%-20% a year in reported errors. In that less urgent context, tests are added to the test suite as needed. Whilst new code is continuously refactored, legacy code is not refactored as a matter of course, but only for major issues or where it is 'obvious'. Some of the more obvious refactorings that have proved worthwhile have been related to the inconsistencies of past coding practices. The large product codebase has 70 projects, and 5,000 files, with a build that used to take an hour. There were many reasons for this, including empty files left in the system and the same code appearing in binaries more than once. Visaggio (2001) defines 'useless components' as those that provide worthless output, but it is also the case that a legacy system may contain useless components that do nothing at all, and these should certainly be refactored out of a system. A related feature of a legacy system may be useless duplication. One of the important pieces of test code added to the system does not test functionality but consistency. It tests all the source header files to check that the first line is the '#pragma once' pre-processor directive that causes the current source file to be included only once in a single compilation. Other tests were added that eliminate risks from non-code related items such as preventing the introduction of business logic to the service interfaces (service contracts, data contracts, etc.).

Other important refactorings relating to header files included using the correct syntax for including library headers versus programmer defined headers to optimise the search path and removing multiple includes. As a result of refactoring the system's naming and header conventions, the build time has now been reduced from over 1.5 hours to 25 minutes (of which about 5 minutes are running the suite of developer tests), with further improvements restricted by the linker being single threaded rather than by the compilation stage. This, however, is still not seen as a final destination. Further improvements will be applied in the future.

Figure 1 provides an architectural view of the use of testable service wrappers around legacy code, and the various communication and testing inflection points in this particular experience context. Windows Communication Foundation (WCF) is used to expose services behind URLs, giving a range of communications options to services. The communications protocol can be configured without changing the server or the client. The most efficient protocol can therefore be chosen both for testing and deployment. Tests can even bypass the WCF proxy/stub code where appropriate to speed up testing. Each service can be tested in isolation. The legacy system features are encapsulated in services that expose interfaces, give additional testability and overall system consistency. From the service consumer perspective, it makes no difference if the underlying code is new or legacy. There is the capacity to replace legacy code with new code if required without affecting the client interface. Figure 1 shows that there are a series of test inflection points; the native client (which still has some legacy features to be refactored out, such as direct database connections), the Web UI, the service interface and the services themselves. This enables tests to be

Figure 1. Service oriented architecture, inflection points, and tests



written in TestComplete or White for the native UI, Selenium for the Web UI, direct developer tests on the various APIs and service endpoints, and Fitnesse acceptance tests for business processes using the service interfaces. The loosely coupled service oriented architecture also enables integration with non-Microsoft systems, and with the standard Job Definition Format (JDF) and Job Messaging Format (JMF) used within the printing industry.

COMMERCIAL AND OTHER BENEFITS

While it is important to give sufficient weight to the technical perspective on transforming legacy systems into services through test driven development, the key drivers behind the approaches described in this chapter are commercial aspects. With the approach now taken it is possible to drive the system in any direction that may be required by the market and/or customers. Integration with partner products, support of mobile clients, plugging-in customer-specific functionality, and other outcomes of this architecture are the means to help the company and the product to address new markets.

The benefits of service oriented architectures also include scalability. The product in the second experience report started with a target of 5 to 25 users many years ago. With the new approach it scales to hundreds or thousands of users. Again, this creates additional commercial options.

The 2-tier approach also had a security challenge as a web application, which is typically deployed in a demilitarized zone (DMZ), needs direct visibility to the database server. With the new design in place for the web user interface, the product has eliminated this requirement thus resulting in strengthened security in live deployments.

By standardizing on service oriented integration the system becomes more consistent and easier to maintain. Code and design that does not conform to the coding and design conventions is easier to spot and eliminate, further enhancing quality.

Finally by using an agile approach, in particular through the support of automated tests and builds, the service oriented architecture can be rolled out incrementally, focusing first on the most important areas of the product. This maximizes customer satisfaction and has a positive impact on the bottom line of the company.

FUTURE WORK

Going forward, areas for further exploration include further decoupling using services. For example certain infrastructure elements such as security, transactions, logging, monitoring, etc. could be provided using a combination of services, aspect oriented programming (AOP) and a dependency injection framework. With this in place a test-driven approach towards implementing these cross cutting concerns is expected to become even easier. Applied to legacy systems it could mean that code that is related to these infrastructure elements can be removed to separate these concerns from the business functionality.

RELATED WORK

Legacy systems provide a number of challenges to enterprises. They may be poor quality, monolithic and difficult to update and reuse. Approaches to the refactoring of legacy systems stress the importance of testing, though the types of testing recommended may vary. Feathers (2002) addresses common issues of working with legacy code, including the agile approach of test driven development, but a number of the specific strategies he suggests are only appropriate to legacy systems coded in object oriented languages. In contrast, renewing more traditionally coded systems may need to rely more on acceptance testing. Visaggio (2001) outlines three tasks in the process of legacy systems renewal, covering both reverse engineering and restoration; automatic processes performed by tools, reading code and documentation, and interviewing the users, maintainers or managers of the system. The key goal is to isolate stable information (embodied in some consistent entity) and unstable information (implicitly held by people), and then to transform the unstable information into stable information. One important feature of this is the role of acceptance testing in rendering the unstable, stable. Service oriented architectures have also been analysed in the context of legacy systems by a number of authors, including Heckel et al (2008) who also stress the importance of automation on some reengineering processes, extracting layered service oriented architectures from monolithic legacy systems. The theme of reengineering a legacy system into a 3 layer architecture is also highlighted by the SOSR approach (Chung et al., 2008)

Table 1. The relationship between traditional software maintenance and agile development (from Thomas, 2006)

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

In terms of agile, we see two areas where agile methods relate to the various aspects of legacy systems of service oriented architecture. First, we see agile techniques such as test driven development and refactoring being applied to legacy systems renewal. Second, we see the role of services in supporting more organisational agility. As Papazoglou & van den Heuvel (2007) indicate, an agile approach is needed to support the rapid construction and assembly of business services into larger architectures. Thomas (2006) also notes that many agile practices actually map well to more traditional aspects of the maintenance of legacy systems, such as regression testing and change management (Table 1).

SUMMARY AND CONCLUSION

In this experience report we have reported on two different contexts in which legacy systems have been integrated into service oriented architectures using an agile test driven approach. Despite the two contexts being in very different domains using very different technologies, in both cases the wrapping of legacy systems with test driven wrappers not only exposed the legacy systems as more flexible and interoperable services, but also enabled them to be substantially improved and refactored.

Being an experience report this chapter is not intended to present new approaches to test driven decomposition of legacy systems into services. It does, however, provide an opportunity to reflect on how the work of others has been applied to this particular context. It also gives an opportunity to consider how an individual enterprise might adopt and adapt a particular approach to the renewal of legacy systems via an agile service oriented architecture taking into account factors such as the nature of the original legacy system, the quality of that system and the market forces that may or may not require complete refactoring of such systems once they have been encapsulated as services. The concept of automation, as outlined by Visaggio (2001) and Heckel et al (2008), and going beyond simply wrapping systems to increase their quality, as indicated by Sucharov (2007), have been shown to be important, while the general approach of Feathers (2002) has been shown to be a useful framework but one that has to be adapted to the nature of the legacy system.

In the introduction to this chapter we reported on a number of issues that some authors claim make the integration of legacy systems, service oriented architectures and agile software development problematical. In this experience report we have endeavoured to suggest that in fact these three very different axes of software can be effectively integrated, with the primary means to do so being a comprehensive approach to testing, applied at every level of the system from the UI, to acceptance tests, to service interface tests to developer (unit) tests. With these tests in place the agile ability to dynamically refactor live services provides the kind of responsiveness that more traditional methods could not deliver.

REFERENCES

Chung, S., Davalos, S., An, J., & Iwahara, K. (2008). Legacy to web migration: Service-oriented software reengineering methodology. *International Journal of Services Sciences*, *1*(3/4), 333–365. doi:10.1504/IJSSCI.2008.021769

Elssamadisy, A. (2007). SOA and agile: Friends or foes? *InfoQ*. Retrieved March 29th, 2011, from http://www.infoq.com/articles/SOA-Agile-Friends-Or-Foes

Feathers, M. (2002). *Working effectively with legacy code*. Retrieved March 29, 2011, from http://www.objectmentor.com/resources/articles/WorkingEffectivelyWithLegacyCode.pdf

Heckel, R., Correia, R., Matos, C., El-Ramly, M., Koutsoukos, G., & Andrade, L. (2008). Architectural transformations: From legacy to three-tier and services. In Mens, T., & Demeyer, S. (Eds.), *Software evolution* (pp. 139–170). Springer. doi:10.1007/978-3-540-76440-3_7

Papazoglou, M., & van den Heuvel, W. (2007). Service oriented architectures: Approaches, technologies and research issues. *Journal on Very Large Data Bases*, *16*, 389–415. doi:10.1007/s00778-007-0044-3

Puleio, M. (2006). How not to do agile testing. *Proceedings of the Conference on Agile,* 2006 (pp. 305-314).

Sucharov, T. (2007). Mainframe makeovers. *Information Professional*, *4*(6), 36–38. doi:10.1049/inp:20070606

Thomas, D. (2006). Agile evolution: Towards the continuous improvement of legacy software. *Journal of Object Technology*, *5*(7), 19–26. doi:10.5381/jot.2006.5.7.c2

Tilkov, S. (2007). 10 principles of SOA. *InfoQ*. Retrieved March 29, 2011, from http://www.infoq.com/articles/tilkov-10-soa-principles

Visaggio, G. (2001). Ageing of a data-intensive legacy system: symptoms and remedies. *Journal of Software Maintenance and Evolution*, *13*, 281–308. doi:10.1002/smr.234

Wilkes, L., & Veryard, R. (2004). *Service-oriented architecture: Considerations for agile systems*. CBDI Forum, April 2004.

ADDITIONAL READING

Astels, D. (2003). *Test-driven development: A practical guide*. Upper Saddle River, NJ: Prentice Hall.

Beck, K. (2003). *Test driven development by example*. Boston, MA: Pearson.

Beck, K., & Andres, C. (2004). *Extreme programming explained: Embrace change* (2nd ed.). Addison Wesley.

Channabasavaiah, K., Tuggle, E., & Holley, K. (2003). *Migrating to a service-oriented architecture.* IBM Developer Works. Retrieved March 28th, 2011, from http://www.ibm.com/developerworks/library/ws-migratesoa/

Feathers, M. (2004). *Working effectively with legacy code*. Upper Saddle River, NJ: Prentice Hall.

KEY TERMS AND DEFINITIONS

Agile Software Development: Building software using the methods and techniques outlined by the agile manifesto, which values individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation and responding to change over following a plan.

Inflection Point: In the definition used by Feathers (2002), an inflection point is a narrow interface to a set of classes behind which meaningful changes to the code base can be detected.

Job Definition Format (JDF): A standard domain specific XML messaging format developed by the graphic arts industry to assist the development of workflow systems including multiple vendors.

Job Messaging Format (JMF): An XML messaging format that is part of the Job Definition Format specification, used to communicate events, status information and results between JDF agents and controllers.

Legacy System: Systems that may be technically obsolete but are still mission critical. Often too frail to modify and too important to discard, they must be reused.

Refactoring: Improving the design of existing code without changing its behaviour.

Service: A software component that consists of business logic, the data it operates on and an interface to access both. A service also has meta-information, e.g. its interface.

Service Oriented Architecture: A loosely coupled architecture of interoperable services that may be implemented in different languages on different platforms, but communicate using common messaging formats over standard protocols.

Test Covering: A set of tests that covers the behaviour of a small area of a system just well enough to provide some 'invariant' that can indicate if the behaviour of the system has changed.

Test Driven Development: Designing units of code by starting with unit tests and then writing the unit under test, incrementally building units of code using both black box and white box testing techniques. The key is to drive the design by carefully selecting tests and mercilessly refactoring code.