# AGILE LEGACY RE-ENGINEERING

DRS. TOM LOVE AND JOHN WOOTEN

Shoulders Corporation

# AGILE LEGACY RE-ENGINEERING:

## A REPEATABLE TECHNIQUE FOR MANAGING MODERNIZATION RISKS

by Drs. Tom Love and John Wooten

Shoulders Corporation

## Abstract

This paper will describe four "modernization projects" that succeeded using agile development methods, open source development frameworks, real time project panels, and radically co-located development. The largest project involved replacing 11 million lines of legacy code (COBOL and C++) with only 500,000 lines of Java code. All projects were completed on time, on spec and on budget. The key was to re-build what already exists using contemporary technology and not try to build what everyone imagines, but no one has ever built before.

## Terms Defined

### Legacy System

Legacy systems are operational systems that an enterprise depends upon for its day to day work. Typically they were built with prior generation software technology (e.g., procedural languages such as COBOL, FORTRAN, C), not very well tested or documented, not well architected, and are considered fragile and difficult to extend or modify. Business rules and workflows are almost always deeply embedded in the software, and where there is documentation, it is rarely consistent with the current version of the software. The original source code is often, but not always, still available.

### Agile

Agile software development is a radically new approach to managing and planning software development projects which relies upon rapid application development in

very short (often two week) development cycles. The particular variant of agile development described here was developed and refined starting in 1997 in a series of 19 development cycles completed for commercial and government customers of Shoulders Corporation. This variant consists of the following:

100 day projects

- begin with a stakeholder signed, one page project charter
- daily stand-up meetings
- thin thread of functionality created ASAP
- start coding on day one
- weekly technical progress meeting

Empowered subject matter experts

- on site, all day, every day
- key technical and business decisions in 24 hours or less
- extreme co-location -- everybody works in the same room
- 2 week development sprints

Real time metrics of performance

- tracking development
- predicting completion based upon objective progress

Quality

- write tests before code
- remove worst errors first
- performance tests

- response time
- size
- number of users
- etc.

Of special importance is the fact that we have found that agile methods work well for substantial projects, not just bleeding edge, small research projects. One of these development and deployment teams grew to 175 people and is in production use by 10,000+ companies and all of their employees. That suggests that enterprise systems can be developed using agile methods.

## Re-engineering[1]

We make an important distinction between re-engineering and modernization. In a re-engineering project the goal is to replace all or a defined subset of existing functionality with a well architected, well designed new system that is more secure and also easier to maintain, extend, and use. In effect, the existing legacy system becomes a *functional specification*. The spec is valid because it is a proper subset of an "existing system" and it is continuously available to developers to determine exact functionality. No requirements document would ever contain all these details.

We might be lucky enough to use existing test cases to verify the correctness and completeness of the re-engineered system. We also aggressively harvest as many useful artifacts from the legacy system as we can including:

- test cases
- data elements (to determine completeness)
- screen by screen functionality

[1] Some people have trouble with the use of this particular term. It is the proper term in our opinion. Others question whether the word should be hyphenated or not. We prefer the hyphen.

- system or user level documentation
- performance metrics
- change request log

The key is "that which exists cannot be impossible". By contrast, you can't say that about a next generation system with functionality and underlying technology that may or may not work or perform as imagined. This is hugely important and probably accounts for our unbroken track record of success.

The completion criteria is being able to decommission the legacy system. Only then can you start to accurately measure the benefits of the re-engineered system.

## Why Re-Engineer?

Organizations need to replace existing systems for a variety of reasons including:

- the legacy system is too difficult to change or extend
- a phone app is needed to expose specific services to the world
- the business has changed and lots of changes are required to the legacy system
- the competition has better systems and is "gaining on us in some important way"
- we can no longer afford to maintain such a large system
- try as we might, we can't find a COTS solution that will solve all or even most of our problems
- the operational model of the business needs to change (e.g., cloud computing versus in house data center)
- baby boomers with their 1960's pony tails are beginning to retire and even die off

Well that's a compelling list of reasons to change. But is the risk of failure too great and the expenses too unpredictable for an organization to take on such a project? Especially

if the general manager is planning to retire in 3 years? Read on and make your own decision.

## A Case Study at Scale

Many examples of agile or rapid development are centered around very small and very lightweight solutions. Often they are showing how to replace a local SQL query based database system with a cute, often two tier, system to obtain and display data using Flash or some other quick building tool. These systems, however quickly they are built and however nice they look are not scalable, usually don't provide anything like the security needed, and don't include features such as adjudication, where it is possible to see what each actor in the system did, and when.

What do you do however, when you need an application that will provide secure payroll and personal information for over 10,000 companies, each with more that 500 employees each, and which must produce payrolls on a weekly, bi-weekly, and monthly basis? How do you handle the need for these employees to do their own employee self-service? Now, do all of this with a thin-client, i.e. in a web-browser. Can this be done as an agile project and still meet these business requirements?

As an example, we will describe a project that had to account for all the taxing rules for every federal, state and local taxing jurisdiction in the United States, and had to do it for 10's of thousands of companies with millions of employees. The legacy system consisted of a client-server implementation, with C++ code sending information to and from a COBOL based mainframe, where essentially the "update" tape was run in a batch mode against the proper companies "master" tape, resulting in the creation of the new master and the output of the check payment data. The system performed only payroll updates and changes, but each software change resulted in a massive distribution of

new version CD's. Some customers applied the changes, some didn't, and a hard training, support, and problem solving issue arose.

The customer then wanted to move to a thin-client, i.e. web-browser based system, that would provide the same functionality, but in addition, also include human resources capabilities and allow for self-service. This was captured as a "one pager" expressing the desires, not as 10's of thousands of pages of requirements. The customers culture showed that the underlying infrastructure could be changing at any time and that developers worked alone on specific tasks usually by assignment from a priortized list of problems or features that needed fixing.

Two previous attempts had been made to re-engineer the system, both of which had failed. After a two year effort, one attempt had resulted on 10's of thousands of pages of requirements, and one login screen. Many powerful tools such as Rational and others had been purchased and used -- to no effect.

Starting with the one page description of desire, a rapid review was made and a small team of very good developers was asked what could be done in 6 weeks. It was determined that a "thin-thread" could be done which would go from login through collection of payroll data, and exchange with the main-frame to quickly go through the cycle using the current tax calculation back-end. Ultimately, that back-end would be replaced also, but the focus of the intial step was to show existing functionality with a new n-tier web-based application.

A loosely coupled framework consisting of a work-flow engine driven by XML files written in Business Process Execution Language (BPEL), a rules engine driven by XML files using Business Rule Markup Language (BRML), a messaging layer, and adapter layer, and an "archiving" layer for managing data access and persistence was quickly created and tested. Instead of custom coding behavior, we focused upon the "core" of the system, the ability to start processes, assign them to roles, service those process activities, apply rules to them, obtain information in an abstract manner, then through adapters provide those services from whatever database, presentation, web server, remote procedure call, etc. was required. Since it was known that human resources and

self service would ultimately be needed, all accesses, rules, services, etc. were designed so that they were not restricted to simply payroll information.

This process of "abstraction" and the use of loose coupling between components means that the system is less fragile, i.e. changes in one area don't produce problems in another as the two areas are unaware of each others existence. Data doesn't become PayrollData, but rather it is Archivable, i.e. it obeys a simple interface that anything that can be retrieved and saved must implement. This results in much less code being written, much better code being written, many tests applied to that foundation code, and the ability to re-use that code for many purposes.

The first phase was finished on-time and produced the "thin-thread" showing role-based activities, from start to finish with the added bonus that it was multi-lingual. This had been added as it was simple to do in modern languages and much easier to do as a design feature than as an add-on later.

The project proceeded through 5 more 100 business day cycles. Each had a particular focus of new functionality that would be available for release at the end of that cycle.

This resulted in the complete replacement of 11 million lines of COBOL and C++ code by 500,000 lines of Java code. While the number of different looking screens that the users saw was above 240, most were based upon patterns with run-time substitutions, meaning that only about 15 different screen types were maintained. The final project ended with 1856 commercial grade Java classes with hundreds of tests in the regression suite. It survived 3 client driven changes in database technology, 2 changes in source code repositories, and the hardest thing of all, continued addition of new developers to the team by the customer as they wanted their developers to understand and be able to use this new development methodology.

From the beginning, there were some hard items to deal with. The customer, after agreeing to the expectations for the next 100 days, would often half-way through want to add some additional functionality or change some features. In the past, the development cycles were so long that this had been forced upon them. The use of 100 day

cycles meant that it was possible to accumulate those requests and be able to promise them for the next 100 day cycle. Ultimately, trust in this process, and the fact that the horizon was not too far away, allowed us to manage the scope of a particular 100 day project much better. During those 100 day projects, there were releases of new and hardened functionality every two weeks, with demos always on-going showing the current state.

One surprise to the customer was the extreme flexibility of the new design. When marketing suddenly appeared a few weeks before the end of a major 100 day release with the request that we stop working and give them an estimate of what it would take to implement 5 new human resource screens because it just had to be in the next release, even if that meant a delay. At the end of the day, the team was able to demonstrate the 5 new screens, integrated into the application. Instead of figuring out how to add new custom software, the developers were able to configure the workflow for those screens in XML, add the rules to the rule XML, layout the desired fields upon the screens using our "standardized layouts" and present the working solution.

In another case, after getting the web front end working, they asked how much effort would it take to add the ability to capture payroll data from telephone input. We explained that if they would wrap the numbers and commands into a standard form of XML that was used for all data moving between components, then the integration would be done. To their amazement, it was completed in two days, using all the existing payroll rules, all the existing workflow, by only providing the data in the standardized for we had created.

A big change to the customers development organization was our request to tear down the walls in the development area and provide waist-high working areas. Each area had four developers in the corners, with a small table in the middle. Each developer had at least two large screens. All chairs could roll around. The teams often consisted of a lead developer, a junior developer, a graphics designer, and a rule/workflow expert or subject matter expert. While many thought this would increase the noise level, it actually reduced it. Workers learned that they could rapidly move around, share work,

pass it to and fro in order to get a new view of their problem. In addition, instead of relying so heavily on a testing group at the end to test from a user perspective, there was heavy reliance on extensive component level testing, with significant regression suites. Since the application was designed around a set of very general and abstract components like the workflow engine, the business rule engine, and the data adapters, the extensive testing of these components lead to a very stable platform with the development effort going towards refining the process XML, the rules XML, and the improvement of a set of JSP tags that allow for much simpler creation of web pages.

All project workers including documentation and management were taught how to use the selected source code repository. All builds were made from the checked in code. Code analysis tools like JIRA and other dependency analyzers were applied to the source code repository to determine the frequencies of check-ins, numbers of problem reports of various levels, frequency of re-fixes, places where parents know about their sub-classes, or lower level classes know directly about classes in other packages. All of these items raise flags and require code reviews, refactoring, and retesting.

The total cost (contractors, in-house developers, trainers, QA, operations, management, product marketing, etc.) was a little more than $100 per line of newly developed/tested/documented code. The development alone costs less than $20 per line of new code.

## Other Projects

Prior to this large project the same project manager and chief architect/developer completed three other projects using a variation of the approach described above. These projects are described briefly below.

### Order Management Application

The project was to re-engineer an existing COBOL based order management application for process industries in general and the steel industry in particular. This was the first project that we developed and used a workflow management framework.

### Equity Risk Management Application

Back in 1997, an innovative startup company in Silicon Valley wanted to showcase their risk management tools by developing a web-based application to do international equity risk management using daily data from 26 countries and make this information available freely on the Internet. The project was completed in 100 calendar days and had 100,000 users the first week.

### Spare Parts Catalog

The goal was to redesign, rebuild and redeploy a very large, customizable on-line catalog of spare parts. The existing software worked well for small catalogs, but had totally unacceptable performance when dealing with 386,000 spare parts for a Fortune 10 company. Of course, each catalog customer wanted to customize their catalogs. In three 100 day development cycles all agreed upon functionality was implemented and fully tested.

## Surprises

One of the biggest surprises as we look back at over 19 successful 100 day agile projects has been the extent to which reuse was obtained. By developing the proper abstractions to common problems and then building a set of abstract business objects along with loosely coupled mechanisms for managing the life cycles of those objects, using role-based workflows and rule engines and expressing almost all business logic for applications in XML ( either BPEL or BRML ), the core for all of our agile legacy re-engineering projects remained the same. Each new project required understanding and creating the proper roles, workflows, rules, and persistence actions and using these with the core framework that was continuously being refactored and improved through each iteration.

When it became time to do a design for a new action or a workflow, it became obvious that the best design language was Java itself. By creating tests before code, and then stubbing in and "mocking" the behaviour of complex objects, the code itself became the design language. By providing comments, adhering to the principles of good object-oriented design and using continuous refactoring with a growing regression suite, the actions themselves are self documenting, but the most important "discovery" was that

the workflows and rules themselves could be automatically transformed using xslt transforms into the documentation for the application. Since customers can and will change their minds, it usually requires only a change in the XML for the business process or rule, then as a part of the build and test process, the documentation is automatically updated and available on-line by application name, process name, and cross-referenced by role.

## Summary

We can't warrant that if your organization re-engineered an important legacy system that you would incur a 20:1 savings in maintenance costs. But one customer has! Others report 5:1 savings. Usually numbers anything like this large make ROI calculations quite acceptable.

A second concern is simply the risk of failure. Modernization projects as usually organized and managed fail often and dramatically. Re-Engineering projects seem to succeed most of the time, by stark contrast.

We don't yet know whether other teams can follow the lessons learned from this project and duplicate the pattern of success. We do know that one small compnay managed to do so without a single failure over 19 development cycles.

Please share with us and the community your team's experiences after reading this short article. Were you successful? What did you do differently that should be shared with the community? What did not work?