Abstract
As software development evolves toward more agile methodologies (SCRUM, XP, RUP), engineering best practices become key to ensuring the delivery of high quality software that best meets a customer’s needs. This paper will briefly introduce agile methodologies and the best practices that facilitate successful agile implementations. It will focus on software configuration management (SCM) practices especially that of continuous integration using a team at Bio-Rad Laboratories as example. It will focus on the details of a Perforce enabled system.

The history of software development is delineated by various process practices – waterfall, spiral, and now agile. What exactly is Agile? Why is it the way to go according to a growing number of practitioners? Agile isn’t actually new. The principles that underlie it have been practiced and developed over the last 100 years. They are driven by the desire to manufacture things more efficiently while meeting customers’ needs better. What makes an agile implementation a success? Why is software configuration management (SCM) so important to a successful agile implementation? These questions will be answered below as I examine in more detail an implementation of SCRUM – a current agile methodology – by Bio-Rad’s mass spectrometer data aggregation and analysis software development team. I will outline how we use Perforce to bring predictability to our product and how it helps us to work daily to eliminate chaos.

Agile as a software development methodology has a longer history than is commonly understood. The tenets of agile – the things that make agile, “agile” – have been a part of lean manufacturing practice for most of the last century. Beginning with Henry Ford and modern assembly line manufacturing, industrialists have been attempting to eliminate waste and make things more efficiently. This was first driven by profit motives but evolved into a desire to delight customers. The two are not at cross purposes and in fact build on each other to help to create a better overall product.

The broadest and most often cited application of agile principles in large scale manufacturing is that developed by Taiichi Ohno, Shigeo Shingo and Eiji Toyoda at Toyota. These principles and practices are collectively known as the Toyota Production System (TPS) and evolved into the movement of “Lean” manufacturing. The practices embodied by TPS include: continuous process flow, periodic reflection, building quality in, publication of control measurements, and empowering team members to control and improve work product, process and themselves.

TPS and Lean manufacturing began to be examined and adopted by American manufacturers during the 1980’s. Coincidentally, software was just beginning to be considered a formal discipline. During this pre-Internet enabled era, software was most often found as a component on larger projects. Projects that were often managed using what is called the “waterfall” or gated development process. Waterfall processes depend fundamentally on the idea that what is required at the end of a project can be determined at the beginning. Stages of development follow one after the other with little feedback between them. The development phases are commonly: specification, design, implementation, test and deployment. Since there is little feedback between the stages, each stage is driven to completion and then handed off to the next set of resources. Change becomes increasingly expensive the later it is introduced in the process. Any change must navigate the entire process again. Change is therefore resisted. Waterfall or gated processes work well in deterministic systems where important decisions can realistically be made up front. Software, as we have discovered, is often not deterministic and can, in fact, meet today’s rapidly changing business needs better if it is allowed to respond to change dynamically - if it is allowed to be more “agile”.

Using Perforce to Facilitate Agility
Victoria Hall
victoria_hall@bio-rad.com
Software Engineering Manager
Bio-Rad Laboratories
1000 Alfred Nobel Drive
Hercules, CA 94547
Software failed miserably to meet customer expectations during the 1980’s and into the 90’s with only 10-20% of the projects during the 1990’s considered succeeded. This means the vast majority of projects were either completely failed or “challenged”.\(^2\) Software was either late, over budget or both and it often didn’t implement the features that users wanted. Less than 20% of features implemented were used, “frequently” or “often” and fully 55% of the features were used, “rarely” or “never”.\(^3\) The need to change seemed obvious. Software development was expensive, time consuming and often didn’t meet the needs it had promised to address.

A number of thought leaders, Jacobsen, Rumbaugh, Booch, Fowler, Beck and others had been working largely as individuals to address the need to improve how we develop software. Ideas began to be published, practices began to gain traction and over a decade an agile movement was born. Jabosen, Rumbaugh and Booch codified their contributions by developing companies around them. Objectory, which merged with Rational in 1995, brought us the Rational Unified Process (RUP). Others published books and articles, met at conferences and developed ideas in the real world on an engagement by engagement basis. The book that, in my opinion, drew the most attention from the software process community to agile practice was, “Extreme Programming Explained”, by Beck published in 1999.

Extreme Programming, or XP as it came to be called, focused on a number of practices that are light weight, can be done by any team, and remarkably hearken back to the practices espoused by many lean manufacturing teams. XP talked about user stories, short iterations, unit testing, and peer review. It highlighted the importance of delivering working software that was fully tested all the time. It distilled what made software successful down to a small set of best practices that allowed software developers to be more responsive to the needs of their customers – to be more responsive to change.

The Agile Manifesto documented the principles that XP offered practices to implement. The Manifesto was written in 2001 and has four lines:

We value

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiations
- Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.\(^4\)

These are very simple statements but they were and, in some circles still are, radical in their implications for the changes they ask us all to undertake. The language of the manifesto is deliberately anti-waterfall in order to question the status quo and offer an alternative that has at this point been demonstrated to give superior results.\(^5\) Agile is not the one solution to all problems but it does provide a set of tools to give software practitioners a good chance of delivering what customers want, when they want it.

Today a number of processes exist that embody agile principles and seek to leverage software engineering best practices. These include: test driven development (TDD), feature driven development (FDD), Crystal, XP and SCRUM.

This paper will focus on an implementation of SCRUM. An article by Takeuchi and Nonaka called “The New New Product Development Game”\(^6\) which appeared in the Harvard Business Review in 1986 first referenced the development process which became SCRUM. The article was a survey of successful product development efforts at Fuji-Xerox, Honda and NEC. The commonalities it found are those that agile seeks to embody: involve the customer up front and engage them in the development process, work in short iterations completing all aspects of a product as soon as practical, periodically reflect on work products and processes and improve them, facilitate communication among team members of different disciplines, let the team self organize and build quality in at all stages of product development.
The first use of SCRUM was by Ken Schwaber and Jeff Sutherland at Easel Corp in 1993. Ken Schwaber and Mike Beedle’s book “Agile Development With SCRUM” codified the SCRUM methodology in 2001 and started a SCRUM movement that continues to grow through today.

SCRUM is most simply described as two documents, three roles and four ceremonies. The roles are: SCRUM Master, SCRUM Team Member and Product Owner. The SCRUM Master is the keeper of the SCRUM process and the SCRUM team coach. Their main job is to ensure that the SCRUM team has everything it needs to do a proper job and to remove impediments as they arise. The SCRUM Team is made up of anyone that contributes to the product being developed. Typically these are developers, testers, build managers, and technical writers. The SCRUM Team owns the implementation. They along with the SCRUM Master and Product Owner develop and commit to a plan for every implementation iteration. The Product Owner is the customer for the implementation. They develop user stories and prioritize the backlog. They negotiate with the SCRUM team to develop a plan that the team believes can be accomplished in the time allotted.

The four SCRUM ceremonies are: sprint planning, sprint review, daily standup and sprint retrospective. In Sprint Planning, the team takes the backlog (see below for a description of a backlog of features) for that iteration or sprint and breaks each user story down into tasks that can be accomplished during the iteration. The team develops finer grained estimates for each task. An owner claims each task and “commits” to getting that task done, to reporting on its progress and to publishing progress of time left on the task. The team reports in the daily “SCRUMS” or standup meetings. These meetings are brief, typically 15 minute or less. Only team members working on tasks can report and only three items can be reported on: what was accomplished yesterday, what is to be accomplished today and are there any impediments or road blocks encountered that need to be removed. The Sprint Review occurs at the end of each implementation iteration and includes a brief demonstration of what was accomplished with live running software. The Sprint Retrospective also occurs at the end of every iteration. In it, the team reflects on its performance the previous iteration, suggests ways to address issues and makes a plan for improvement during the next iteration.

The two documents important to the SCRUM process are the product backlog and the sprint backlog. The product backlog is a prioritized list of features, usually described in terms of user stories that could be implemented. It can also include technical or architectural needs of the product (e.g. refactor JDBC access layer). Each item is estimated. Estimates are only coarse grained. Many teams use something called T-Shirt sizing, Small – Medium-Large – Xtra Large. During Release Planning, the highest priority items from this backlog are assigned to sprints by the team. The Sprint Backlog is a list of tasks, again developed by the team, to meet the goal of the Sprint. The team self-organizes, estimates each task, and owners claim tasks for completion and reporting. The plan is developed so as not to exceed what the team knows it can accomplish in a given time boxed period.

Time boxing is common to many agile implementations. It ensures that the team doesn’t try to anticipate things they have no knowledge of and lets the customers know when they can expect the implementation of their next feature set. As a team begins to gel, it will develop a rhythm and get quite good at frequently releasing working software. This allows the team to build credibility with management and with customers. It also allows it to gain confidence in its ability to deliver. Other practices which have developed to facilitate agility include: unit testing, continuous integration, peer review, publicizing progress frequently and retrospectives. I will focus in the latter half of this paper on continuous integration.

Continuous integration embodies a set of engineering best practices to ensure that a code line is deliverable (including validation and documentation) as a product at any time. It is one of the most important practices to enable agile principles especially the agile principle of “embracing change.” In order to give the customer the most flexibility in making requests of a development team, the team needs to know where it stands at any time. Their estimates are better and they discover what they have broken by any one change quickly if they have a disciplined continuous integration infrastructure. They are willing to embrace change because they understand that they can mange the change. It doesn’t disturb the system and mean a lot of extra work. It is what the system is designed to accommodate.

At Bio-Rad Laboratories, we have implemented SCRUM for the development of software in support of proteomic biomarker discovery. The software controls a mass spectrometry instrument, aggregates data and allows biologists
to analyze that data. The goals are primarily to understand disease progression and diagnosis, but can be applied to areas as broad as drug development and food safety.

**Bio-Rad Proteomics SELDI Instrument**

**Proteomics analysis software**

Our product is JAVA based and runs on the Windows and Linux platforms. It is a three tier product that uses either Oracle or mySQL as the database and Tomcat or Jetty as the application server. We build with ANT, unit test with JUNIT, JFCUNIT, automate regression tests with Borland SilkTest and control source, documents, and tests with Perforce. We manage our SCRUM implementation using RallyDev, an agile project management tool.

We develop using four week iterations. We’ve found this works best given our team and the infrastructure within Bio-Rad that we must interact with. Our developers are required to keep the code line buildable and tested at all times. They are also required to use and maintain the same scripts that the build and installation engineers use. We resist the notion that separate developer script environments saves time recognizing that if we keep everything in sync we can release better product faster. Errors are discovered and corrected quickly. Engineers from different disciplines must discuss status and negotiate changes daily; unintended consequences are discovered before they can do much harm and are not perpetuated throughout the system.

We use a suite of unit tests written in JUnit and JFCUnit to ensure that implemented features work as intended and that their implementation does not break anything else. New feature implemented must have a test as do defects fixed. Developers must retrieve others’ changes from Perforce into their client workspace, do a local build, integrate and test, fix anything found and then, finally, check in. We extend this philosophy even to installation code and have instituted a manual unit test for even the smallest installer changes.

Our continuous integration system is a series of shell scripts that our configuration manager has knitted together over the last year. The system interacts with Ant and with Perforce directly. The basic process flow is that once a change is checked into Perforce, the build number is determined via `p4changes` and applied to the build, a build is then kicked off and completes, once complete a JFCUnit smoke test runs and sends email depending on its result to affected parties. Depending on the rate of development, many builds can be completed and tested a day. We run a full regression test suite using SilkTest nightly on the last build of the day that passes the smoke test. The results are published to a web page available to all.

For the detailed description: a cron job requests of Perforce the latest change list number and if that number has been incremented since the last enquiry, a build kicks off. The build script is run as its own utility user with its own Perforce client on a build machine. The script gets the latest change list number using `p4 changes` This is the “build number” – we avoid labeling source this way and take advantage of built in Perforce functionality. A product build is completed. The about box templates, README’s etc. are updated with the build number for this build. The
installer build completes and a CD image is generated. The CD image is auto-deployed to a neutral area as a live
build. The script installs the build from this CD image and smoke tests the build. The results are sent to the team
via email. For the last good build, once a day in the evening, the SilkTest scripts detect a new build is available and
kick themselves off for a full regression test. The SilkTest scripts are self documenting and a pass/ fail web page is
generated. The whole process takes several hours including full regression testing. The initial smoke tested build
takes approximately 10-15 minutes. The final regression testing can take approximately two hours to run through
everything.

It is worth noting that final documentation is included in each and every build as well as release notes with changes
checked in up to that point in development. We take seriously the idea that our Product Owner could decide that
this release is “good enough” and could be released to the real world with only minimal hardening and wrapping.

To ensure we have a way of tracking exactly where we are at any time, we leverage Perforce to control change on
not only product code source but on documentation, help source, test source and results, installer source and build
source and results. We have a complete history of what was done by whom and when. We have found this tool to
be robust through the four complete releases we’ve done this year as well as the introduction of a new product.

For all of this, at the end we are reasonably confident that we did not break anything that already existed in the
product and that new features introduced meet their specifications. We are free to change things and to know what
the implications of those changes are to the real deployed product. This does not free us from doing careful and
deliberative design and implementation but is does allow us to catch things that we overlooked during that process
or that we have no way of anticipating due to a confounding of information across disciplines.

To set up a continuous integration system, you and your team need to commit to a few basic principles; it helps to
then invest in some personnel with configuration management skills and some hardware, but that is not strictly
necessary. A single developer working on a single box can implement continuous integration. I’ve discussed
above the need to commit to the idea of “DONE”. This is one of the basic principles of SCRUM and indeed of most
agile practices. The product should be developed incrementally – moving along all aspects of the piece in parallel as
much as possible. This encourages teams not to commit to more than they can realistically get done in a short
period of time and gives outsiders a way to hold the team accountable. Does the software perform the features as it
should? Is the documentation present and accurate? Can a customer install and use this piece? No interpretation is
required on the part of the customer. They can actually take the software, install it and use it to see if it meets their
needs.

On the infrastructure side, the team needs to acquire a source control tool such as Perforce and commit to using it. It
is best if the tool has a command line option and integrates with your build environment, Ant in our case, as Perforce
does. It is easiest if the team has access to machines that will allow separation of roles between the disciplines -
development, build, test, integration and deployment validation. We have found it easiest to keep these things
separate to allow us to move quickly and to debug easily. We do not pollute environments accidentally and
developers can try new things without disturbing environments that should not be changed - in particular those that
mimic the deployment environments of customers.

As final release approaches, we implement code freeze by changing the team’s access in Perforce to read only. The
release candidate is run through an additional suite of manual tests emphasizing actual (not simulated) instrument
data acquisition and processing. As defects are discovered and prioritized for fixing, we unlock the Perforce
repository for the developer designated to fix the defect. Once we’ve gone through acceptance tests, we go back and
label the code using the change list build number as a link as our release candidate. Once the release candidate
passes alpha and beta testing we change the label to “released”. It is then launched in our internal manufacturing
processes for CD production and inclusion in an instrument bill of materials.

A final note on continuous integration and scripting; we use a rather extensive library of custom developed scripts.
These can be in any language that allows for command line interaction between source control, build, deploy and
test. We obviously make heavy use of the p4 command line options: to check if changes have occurred, to track
client workspaces, to retrieve the proper source, to label, to freeze development and to track changes for release
notes. You can also achieve many of the same results with open source tools such as Anthill13, Hudson14, and
CruiseControl. We are looking at implementing with Hudson later this year. To facilitate an agile adoption, it is not that important how continuous integration is done. The important thing is to do it.

Summary

The most important step in facilitating agile in daily practice is continuous integration. Change control is mandatory for rapid specification, development, release and incorporation of customer feedback. A team needs to know that what they are building is complete and of a high quality. Only then can an organization move forward with speed and impunity being open to change, responsive to business needs and able to take advantage of improvement opportunities.

Perforce is very easy to build a continuous integration system around. It is robust, fast and behaves as promised without corrupting data. It also integrates nicely with ANT, Junit, and Silk for Java but with the command line tool you can integrate with any build/test/defect system using minimal scripting.

Even though the team described in this paper is small, this process and the lessons scale easily and have been used successfully at a number of larger organizations including Google and Yahoo!. There are a number of agile methodologies out there. If you’re interested in implementing agile in your organization, the most important step is committing to the process and giving it a chance to work.

Acknowledgements

I’d like to thank my colleagues at Bio-Rad and at Santa Cruz Software Release Specialists for their contributions and assistance in developing this article. Iztok Marjanovic, John Dunaway, Niels Thomsen, Sangeeta Ahuja, Curt Patrick, Tom Greer, Robin Patrick, Conan Nado, Scott Conradson and Brian Robertson.

Notes:

1 Ohno, Taiichi (1988), Toyota Production System: Beyond Large-Scale Production, Productivity Press
4 www.agilemanifesto.org
8 www.mysql.com
9 www.junit.org
10 jfcunit.sourceforge.net
11 www.borland.com
12 www.rallydev.com
13 www.anthillpro.com
14 hudson.dev.java.net
15 cruisecontrol.sourceforge.net
Resources

- Home of Scrum [www.controlchaos.com](http://www.controlchaos.com)
- The SCRUM Alliance [www.scrumalliance.org](http://www.scrumalliance.org)
- Agile University [www.agileuniversity.org](http://www.agileuniversity.org)
- Agile Project Leadership Network [www.apln.org](http://www.apln.org)
- Agile Alliance [www.agilealliance.org](http://www.agilealliance.org)
- Yahoo Groups: scrumdevelopment, XP, XPUK, agiletesting

Suggested Reading

- Agile Software Development with Scrum, Ken Schwaber and Mike Beedle, Prentice Hall 2002
- Agile Project Management with Scrum, Ken Schwaber, Microsoft Press, 2004