Crossroads News
A Monthly Publication for Software and CM Professionals

Lessons Learned from Agile SCM

Steve Konieczka, Brad Appleton, Steve Berczuk – December 2003

Over the past 8 months, we learned some important lessons about a couple of topic areas during our discussions of Agile SCM. This month we will cover these topics with the hope of clarifying the lessons and furthering discussion within our CM community. The topics are *Communicating Agile SCM Concepts* and *Right-sizing documentation on your projects*.

Communicating Agile SCM Concepts

When discussing any new concept, one of the biggest challenges is to establish a common terminology. Agile SCM is no different. Because it's a variation on an already existing discipline, terminology can become a barrier to discussion. There are so many pre-conceived notions of terms such as "light-weight", "agile", "extreme", "process" and "structure" that using these terms without clarification or further discussion can occasionally lead to gross misunderstanding.

The term "Agile SCM" can (and has) elicited very different reactions based upon differing perspectives and experiences. There are those who believe that Agile SCM is negating many of the process benefits we've made over the past 10 to 15 years, resulting in lower quality software.

Have you had thoughts that Agile SCM has declared war on discipline and control? You're not alone – many in the CM community have reacted with these same feelings. Do discipline and control lead to quality software? Certainly agile methodologies are ultra-focused on quality through their use of concepts like test-driven design (TDD), refactoring, close customer collaboration, pair programming, etc. They are also ultra-focused on being simple, and "lean" - doing only that which "adds value" and removing that which does not. They are also about effective communication and knowledge sharing in a way that relies more on rapid feedback-cycles of face-to-face dialogue than on static knowledge captured in large formal documents that attempt to complete all the requirements before coding begins.

The Discipline of KISS (Keeping It Simple and Short)

Agility is partly about removing redundancy and complexity from all aspects of development: from designs, from requests and requirements, and even from and processes and tools. It is partly about separating niceties from necessities, forcing us to question what really is needed, and removing the things we don't really need. Many times, there are gold-plated requirements, over-engineered systems, and overly complex or laborious tools and processes employed in many software development shops. These extraneous elements add complexity and/or redundancy that can create more friction than forward motion on development projects. The result is that they add little if any value to the bottom line – quality software.

The "Grand Illusion" of Control

"Control" is a term that gets thrown around a lot in the CM community. In many ways, Agile SCM is more about accepting and embracing change than preventing or controlling it. Agile methods acknowledge that "control" is merely an illusion, and a damaging one at that *[1]*. Discipline and predictability are quite different from control *[2]*, and Agile SCM requires high discipline with short-term effective feedback. An Agile SCM solution is one aimed at helping developers make accurate code changes to the appropriate set of code while collecting data on changes as they happen. Effective Agile SCM solutions then provide information to help users access that change data as well as enable consistent and predictable releasing of that code.

The Power of Tight-Looped Feedback

Many opponents of agility don't have an accurate understanding of its highly iterative nature and close-knit feedback loops at multiple levels of scale. Even those familiar with iterative development may not be familiar with iterations that are only a few weeks in length. The difference between 2-3 week iterations and 2-3 month iterations is tremendous.

The longer the period of time in between making a decision and being able to execute and validate/evaluate the results of that decision, the more we need to be sure of ourselves because rework is more costly. So more formality and more formal knowledge capture and signoff is necessary. When the period of time is contracted down to a couple of weeks, days or hours, rework is an order or magnitude less costly and decisions are more malleable and less needs to be "written in stone" because changing it isn't nearly so time and labor intensive.

The closer the feedback loops, the more accountability becomes shared between customers and development. And the more frequently it is shared, the more decisions become a matter of collaboration rather than negotiation: We can quickly try something, see the result, and then adjust/correct our speed and direction by working together on small micro-sized increments of functionality at a time.

While many of the terms we use to discuss Agile SCM come with "conversational" meanings, we need to understand the limits of the language and understand the best that we can do is to find a metaphor that starts us on a path to understanding. In as much as Agile SCM is based on acknowledging the limits of our abilities to control every aspect of development, learning about Agile SCM needs to be grounded in the understanding that words have limits.

Documentation – How much is too much?

One of the biggest lessons learned is with deciding how much is the appropriate amount of "textual" documentation required to effectively support an Agile SCM solution. It's safe to say that every project differs in the amount of documentation it requires. So what questions should we ask to help decide how much is the right amount? Maybe the appropriate place to start is to define the basic reasons why we create "textual" documentation in the first place (from a software perspective).

- To consistently share knowledge across many people in a scaleable way: Although writing is certainly not the most effective form of communication, it does scale in that you can share knowledge across many people with relatively less effort than talking to those same people individually.
- Provide a persistent media from which to collaboratively build knowledge: As we work with others to build knowledge, it often helps to have the knowledge-to-date persist so that we can build on it.
- Make knowledge persist over time: There exists knowledge that is somewhat stable and offers value in a persistent form.

Agile and Lean methods suggest that since knowledge changes over time, we should delay decisions as late as possible and decide as "low" as possible. This translates into writing as few documents as possible as late as possible, and when required, use documents that are truly central to the actual point of the project, for example generating software. Don't spend valuable resources writing documents that will never be read again, or become obsolete in a short period of time.

This seems to make sense, yet something still may not feel right with taking such a strong stance on documentation. Maybe it's because of the "It's the way we've always done it" syndrome, or maybe it's because we're thinking of trying to apply a single principle to all projects. This agile approach to documentation flies in the face of traditional formal traceability, something that's necessary to support large and complex software development efforts. It's all about choosing the appropriate amount of traceability to satisfy the needs of your project. Document when you have a REASON to document and understand the reason.

Whence Formal Traceability?

The mandate for formal traceability originated from the days of Department of Defense (DoD) development with very large systems that included both hardware and software, and encompassed many geographically dispersed teams collaborating together on different pieces of the whole system. The systems were often mission critical in that a typical "bug" might very likely result in catastrophic loss of some kind (loss of life, limb, livelihood, national security, or obscenely large sums of money/funding).

At a high level, the purpose of formal traceability was three-fold:

1. Aid project management by improving change **Impact Analysis** (to help estimate effort/cost, and assess risk)
2. Help ensure **Product Conformance** to requirements specs (i.e. ensure the design covers every requirement, the implementation realizes every design element and every requirement)
3. Help ensure **Process Compliance** (only the authorized individuals worked on the things [requirements, tasks, etc.] they were supposed to do)

On a typical agile project, there is a single team of typically less than two-dozen. And that team is likely to be working with less than 10 million lines of code (probably less than 1 million). In such situations, many of the aforementioned needs for formal traceability can be satisfactorily ensured without the additional rigor and overhead of full-fledged formal requirements tracing.

Rigorous traceability isn't always necessary for the typical agile project, except for the conformance auditing, which many agile methodologies accomplish via TDD. A "coach" would be responsible for process conformance via good practices and good "teaming", but likely would not need to have any kind of formal audit (unless obligated to do so by contract or by market demand or industry standards).

Agile methodologies turn the knob up to 10 on product conformance by being close to the customer, by working on micro-sized changes/increments to ensure that minimal artifacts are produced (and hence with minimal reconciliation) and that communication feedback loops are small and tight. Fewer artifacts, better communication, pebble-sized change-tasks with frequent iterations tame the tiger of complexity.

The Principle of Locality of Reference Documentation (LoRD)

Not all software projects fit into the ideal smaller-scale environment with closely collaborative project communities. These larger projects require more artifacts. More artifacts, means more things to trace, and more differences to reconcile, and more effort to track and maintain them. Here, the principle of locality of reference can be applied to documentation (as well as to a configuration item and the configuration identification that describes it). The *Principle of Locality of Reference Documentation* (LoRD) *[3]* states that:

> The likelihood of keeping all or part of a software artifact consistent with any corresponding text that describes it is inversely proportional to the square of the *cognitive distance* between them.
>
> A less verbose, less pompous description would be simply: *Out of sight, out of mind!*

Agile methodologies address artifact traceability by minimizing the number of different artifacts produced (especially non-code artifacts). In the most extreme case, LoRD says that when the distance between two things is effectively zero, then there is nothing to trace. For example, in an extreme programming (XP) project, how do I trace a story to its tests and vice-versa? An "*extremist*" might say:

> "Simple! Just flip over the index card that contains the user story. They're on the same physical artifact - problem solved because the pieces of information to trace were never split up into separate physical elements in the first place."

> Regarding tracing requirements (stories) to code changes ... if it's required (for whatever reason), might not a checkin or checkout comment simply identify the corresponding story? Seems to me that would do it for tracing to unit-tests too. So that takes care of the cards and code.

Now maybe not all agile projects use index cards as the sole means of requirements capture (cards are often just an initial capture mechanism, with a tool being used to store and track/sort/report the requests for features and fixes.), but the basic idea is the same: *Placing the related (traceable) information in the same "storage" container minimizes the burden of maintaining linkages*.

This is the essence of applying the LoRD principle! Ideas such as "Literate Programming" and the ability to declare variables in C++ and Java just before their use (instead of "up front" at the very beginning) are all based on this same principle of locality of reference!

Configuration Item Identification/Reports

Regarding creating CI reports, managing dependencies between them and keeping them up to date with the actual code … from an agile perspective, today that happens during the build process. Build time is when we can capture the actual CI's for that release, know the dependencies between them and can relate them to the actual code with confidence that they won't change for that release.

In the future when "Model-Driven Development" and MDA tools are more mature, we predict they will become the source of configuration and build information (in keeping with the ideal of round-trip engineering and minimizing the "cognitive distance" between the description of the system and the working system itself):

- The Implementation View and Deployment Views of the UML-based 4+1 views of architecture *[4]* already have the elements necessary to model the configuration items (CIs) and their dependencies, and to capture their "attributes" and associations with other entities in the other views (such as classes/objects).
- What they don't currently have is the ability to automatically create and update/maintain the dependency information or a way to associate ANT/Make recipes and actions so that the build script itself (or at least its dependencies) can be auto-generated.

The Litmus Test

Should we formally document or not? Here are some questions that can help us decide for our project:

Is the document central to the overall objective/goal of your project? If the document discusses some part of your project that can be better served by discussing it in person, question the expenditure of resources to put it to paper.

Does the document really require persistence? It's not uncommon to have project documents that are created without considering this question. Is it really necessary to have a document that describes the thought process behind the development methodology in use? If so, make sure that the information is somewhat stable and closely managed to the CI that it is written for – and furthers the team to the project goal. If not, the expense in creating and maintaining the information in manual, textual means will outweigh the benefit it provides and result in a waste of resources.

Are you trying to communicate to many people? Documents can be efficient in the sense that we can write them once and speak to many listeners. They are inefficient in the sense that they are substantially less effective in communicating than conversation.

Manual processes versus automated processes: Is there a way to automate a procedure instead of documenting it? Well documented manual builds are a classic example of a situation where the project would typically be far better off by automating the build and letting the automated build script become the document – effectively making the separation between the document and source zero.

How close is the document to the actual CI that it represents? When creating the document, look for ways to close the gap between the document and the source it refers to. This will minimize the expense in maintaining the document over time.

Summary

Agile SCM is an approach requiring significant discipline from participants. It's very focused on building quality into the software and in releasing more software in a shorter period of time. A more effective way of communicating agile concepts is through the use of metaphors.

Different projects have different tolerances for risk, which will dictate the amount of knowledge traceability required. The *LoRD* principle tells us that the cost of knowledge traceability approaches zero as the distance between the two knowledge references becomes closer. Therefore, when we do require documentation for a project, for whatever reason, work to bring the documentation as close as possible to the referenced item it discusses. This will significantly reduce the opportunity for the document to become out of date as well as reduce the cost of maintaining that document.

References

[1] ***Managing Software for Growth: Without Fear, Control, and the Manufacturing Mindset***, by Roy Miller; Addison-Wesley, July 2003

[2] ***Balancing Agility and Discipline: A Guide for the Perplexed***, by Barry Boehm and Richard Turner; Addison-Wesley, August 2003

[3] *Locality of Reference Documentation,* by Brad Appleton; drafted January 1997. http://c2.com/cgi/wiki?LocalityOfReferenceDocumentation

[4] *The "4+1" View Model of Software Architecture*, by Phillipe Kruchten; IEEE Software Vol. 12 No. 6 (November 1995). http://www.rational.com/media/whitepapers/Pbk4p1.pdf

[5] *Agile Documents Discussion*, from agile-testing discussion group on Yahoo.com; December, 2003 http://groups.yahoo.com/group/agile-testing/message/2814

[6] *People Factors in Software Management: Lessons From Comparing Agile and Plan-Driven Methods*, by Richard Turner and Barry Boehm; in CrossTalk: December, 2003 http://www.stsc.hill.af.mil/crosstalk/2003/12/

[7] *Agile Software Development: The Business of Innovation*, by Jim Highsmith and Alistair Cockburn; IEEE Computer: September 2001 (Vol. 31, No. 9), pp. 120-122 http://www.jimhighsmith.com/articles/IEEEArticle1Final.pdf

Brad Appleton is co-author of **Software Configuration Management Patterns: Effective Teamwork, Practical Integration**. He has been a software developer since 1987 and has extensive experience using, developing, and supporting SCM environments for teams of all shapes and sizes. In addition to SCM, Brad is well versed in agile development, and cofounded the Chicago Agile Development and Chicago Patterns Groups. He holds an M.S. in Software Engineering and a B.S. in Computer Science and Mathematics. You can reach Brad by email at *brad@bradapp.net*

Steve Berczuk has been developing object-oriented software applications since 1989, often as part of geographically distributed teams. In addition to developing software he helps teams use Software Configuration Management effectively in their development process. Steve is co-author of the book **Software Configuration Management Patterns: Effective Teamwork, Practical Integration**. He has an M.S. in Operations Research from Stanford University and an S.B. in Electrical Engineering from MIT. You can contact him at *steve@berczuk.com*. His web site is www.berczuk.com

Steve Konieczka is President and Chief Operating Officer of SCM Labs, a leading Software Configuration Management solutions provider. An IT consultant for 14 years, Steve understands the challenges IT organizations face in change management. He has helped shape companies' methodologies for creating and implementing effective SCM solutions for local and national clients. Steve is a member of Young Entrepreneurs Organization and serves on the board of the Association for Configuration and Data Management (ACDM). He holds a Bachelor of Science in Computer Information Systems from Colorado State University. You can reach Steve at *steve@scmlabs.com*