

The Agile Difference for SCM

Brad Appleton, Robert Cowham, Steve Berczuk – October 2004

First, we want to bid fond farewell to our Agile SCM co-columnist Steve Konieczka, who is heading off to greener pastures to run a new business (completely unrelated to software/system development). *Best of luck to you Steve!*

Last week, the CM Specialist Group of British Computer Society (BCS CMSG) hosted a special event on Agile-versus-Traditional Configuration Management [1]. Presenters discussed the impact of agile methods to traditional SCM and posed the question: Is there a need for Agile CM - and what is it? This month, we describe what we believe are the root causes of some key differences between agile and traditional development, and how they change certain assumptions SCM has about software development.

Should I be worried about Agile Development?

Agile development practices can significantly impact software development practices. Of course, any significant process change is ultimately a change in the organization's culture. And underlying any cultural change is a set of values and beliefs that define and sustain the change in behaviors and mindsets. How will such changes affect us as SCM professionals?

- Should SCM professionals be worried about agile projects running out of control?
- Should agile teams be worried about SCM cramping their style and entangling them in red tape?

Our answer is hopefully reassuring to both camps! Agile software advocates push for lean, people-oriented, adaptable development processes. At an XP conference in London recently, one of us met both the wild enthusiasts fired up with XP fervor, and the more cautious people trying to find out what it was all about and take back some useful ideas.

This represents the real world: although there are increasing numbers of "pure" agile projects out there, there are a much larger number of projects and organizations which are taking agile practices and looking to introduce them to their current development processes. Some opt for a revolutionary approach to introduce agility into the workplace. However more and more are finding that an evolutionary approach to adopting methods from the agile camp is what is going to have the widest applicability and success.

Agile Software Development – What's the big deal?

Led by the popularity and success of [eXtreme Programming](#) (XP), and espoused by the same respected experts who introduced software designers to patterns, agile methods like [XP](#), [Scrum](#), and [FDD](#) have gained increasing popularity and notoriety! Agile methods emphasize people, and a "lean" approach to process and documentation. James Highsmith writes:

Agility is the ability to both create and respond to change in order to profit in a turbulent business environment. [2]

Responding quickly and effectively to change is easiest to do when we can minimize the following [3]:

- the cost of high fidelity knowledge-transfer between individuals
- the amount of knowledge that must be captured in intermediate artifacts
- the duration of time between making a project decision, and exploring its results to learn the consequences of implementing that decision (feedback & learning latency period)

So achieving agility hinges upon reducing the time and cost of effective communication and learning [4] (and, ultimately, the creation, transfer, and codification of executable knowledge) [5][6][7].

Agile Development Characteristics

On the surface, most agile development practices are not new (iterative development, pair-programming, refactoring, test-driven development, and others have all been around in some form or another for at least a couple decades). Referring again to Highsmith:

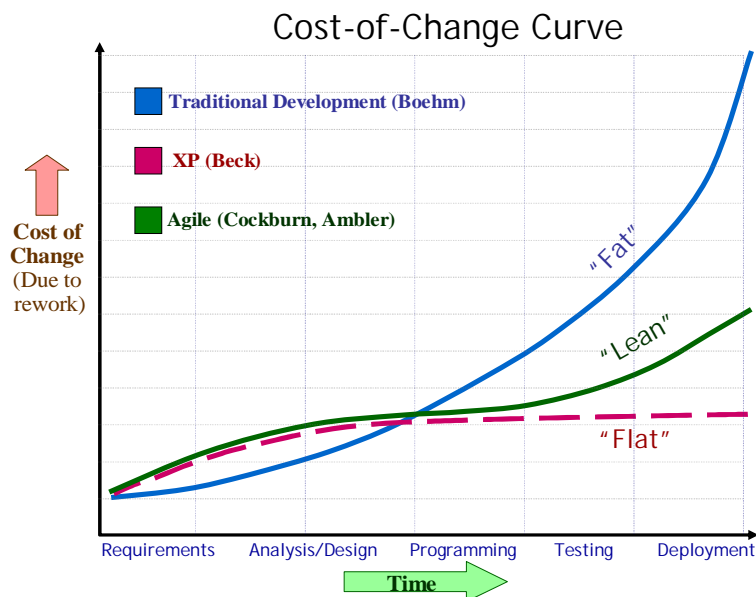
What is new about agile methods is not the practices they use, but their recognition of people as the primary drivers of project success, coupled with an intense focus on effectiveness and maneuverability. [2]

For this reason, agile methods focus upon the strengths of people and teams more than they do upon rigorous processes or detailed documentation. Projects employing the methods founded upon these “agile” principles and values [7] share the following key characteristics:

- **Adaptive** – plans, designs, and processes are regularly tuned and adjusted to adapt to changing needs and requirements (as opposed to being predictive) [2][8][9]
- **Goal-driven** – focus on producing end-results (tangible working functionality) in order of highest business value/priority [8]. “Agile approaches plan features, not tasks, as their first priority because features are what customers understand” [10][11].
- **Iterative** – short development cycles, frequent releases, regular feedback [2][9][10].
- **Lean** – simple design, streamlined processes, elimination of redundant information, and barely sufficient documentation and methodology [7][8][11]
- **Emergent behavior** – quality requirements/architecture/design emerges from highly collaborative, self-organizing teams in close interaction with stakeholders [7][8][11]

The Agile Premise: Streamlined Cost-of-Change Curve

Beneath all the agile hype lies an extraordinary challenge to previously prevailing notions of software development and economics: By relying upon intense collaboration, hyper-frequent feedback, and principles of lean production applied to software [12][13][14], the claim is that agile methods are the lever that allows us to “move the world” by streamlining the traditional “high cost of change” curve!



Around 1980, Barry Boehm published data showing how the cost of change and rework seems to increase approximately tenfold for each software development phase requiring rework activities as a result of the change [15]. Around 20 years later, while describing Extreme Programming, Kent Beck mused that the exponentially increasing cost of change curve could somehow be flattened [16], and speculated how different software development would be if changes could be made inexpensively without hindering quality.

If changes could be made quickly and inexpensively without degrading quality, then it would mean that changes in requirements (even late changes) need not strike fear into the hearts of project managers and software engineers. Instead, such changes could be accommodated quickly and easily. Furthermore, the ability to do so could become significant competitive advantage [14].

Alistair Cockburn [17] and Scott Ambler [18] subsequently dispelled the myth that agile methods truly “flatten” the cost-of-change curve. It is still exponential, but if development cycles and feedback-loops are kept sufficiently short and frequent, we can tolerate the early portion of the curve where growth is still almost linear. Even if we can’t truly make the curve go from “fat” to “flat”, we can instead make it go from “fat” to “lean” by striving for the aforementioned agile development characteristics.

Thinking “Lean” about Knowledge-Creation

Thus, much of eXtreme Programming and other agile methods are focused on practices to reduce the cost of making changes. They do this with a focus on executable end-results that strives to eliminate redundancy and waste while streamlining all other intermediate artifacts and activities.

Lean Software Development applies the principles and techniques of lean production to software development. The focus is on eliminating waste to maximize throughput of the value stream, and minimize latency-time for feedback and learning. Eliminating waste and optimizing the flow of value imply being able to see the value stream and identify where waste is present and the form that it takes. According to Mary and Tom Poppendieck [12], the seven forms of “*muda*”, or waste, in lean production map to the following **seven forms of software waste**:

1. Extra/Unused features (Overproduction)
2. Partially developed work not released to production (Inventory)
3. Intermediate/unused artifacts (Extra Processing)
4. Seeking Information (Motion)
5. Escaped defects not caught by tests/reviews (Defects)
6. Waiting (including Customer Waiting)
7. Handoffs (Transportation)

Again, these are predominantly minimized by relying upon frequent feedback and close collaboration to create closed-loop learning-cycles at all levels of scale throughout the development lifecycle. All activities are aggressively time-boxed and streamlined to minimize cycle-time, and to eliminate or minimize all forms of software waste. The result is the following:

- Minimal Deployable Releases
- Minimal Manageable Iterations
- Minimal Meaningful Models
- Minimal Marketable Features [19]
- Minimal Malleable Designs
- Minimal Mergeable Change-Tasks
- Minimal Testable Modifications
- Minimal Perpetual Peer-Review

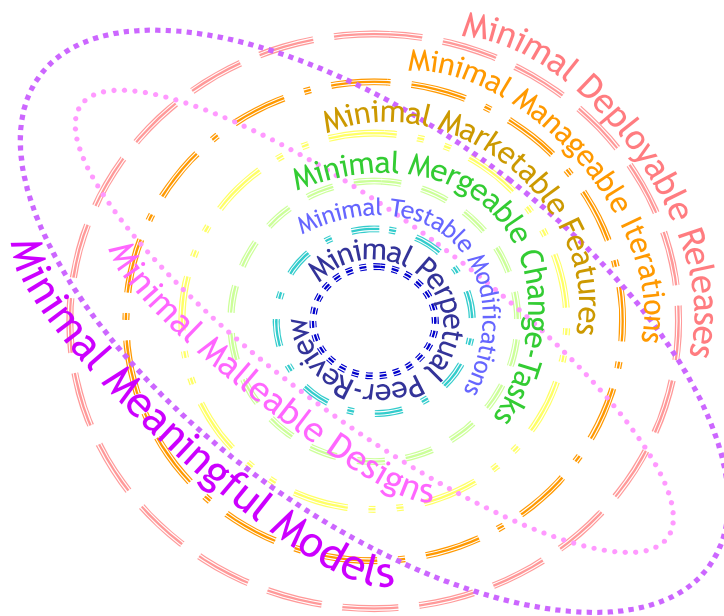
Crossroads Journal

A Monthly Publication for
Software and CM Professionals

These are briefly described below:

Minimal Deployable Releases	Release content is minimized to balance the impact of deployment and installation/upgrade against the amount of business value delivered.
Minimal Manageable Iterations	Iterations are kept short in order to minimize the opportunity for changes to their scope, and rapidly enable customer-feedback on tangible working results in the shortest time-span
Minimal Meaningful Models	Models are made as simple as possible to minimize intermediate artifacts while conveying the essential system metaphors, domain abstractions, and their important relationships.
Minimal Marketable Features [19]	Features are decomposed into the smallest marketable units of useful deliverable business value, in order to fit within short iterations and ensure critical functionality is implemented first.
Minimal Malleable Designs	Code is structured and refactored to be as simple as possible to minimize over-engineering, redundancy, and dependencies, while maximizing clarity, modularity, and encapsulation.
Minimal Mergeable Change-Tasks	Changes are checked-in and committed to the codeline as task-level transactions representing the smallest-feasible unit of working/tested functionality that won't break the build.
Minimal Testable Modifications	Classes/modules and their methods/functions are developed in the smallest feasible increments of testable functionality, and then tested prior to coding the next increment.
Minimal Perpetual Peer-Review	Code is reviewed in very-small increments as soon as feasible after it is written (possibly even as it is being written) to minimize error detection and rework, and enforce coding standards.

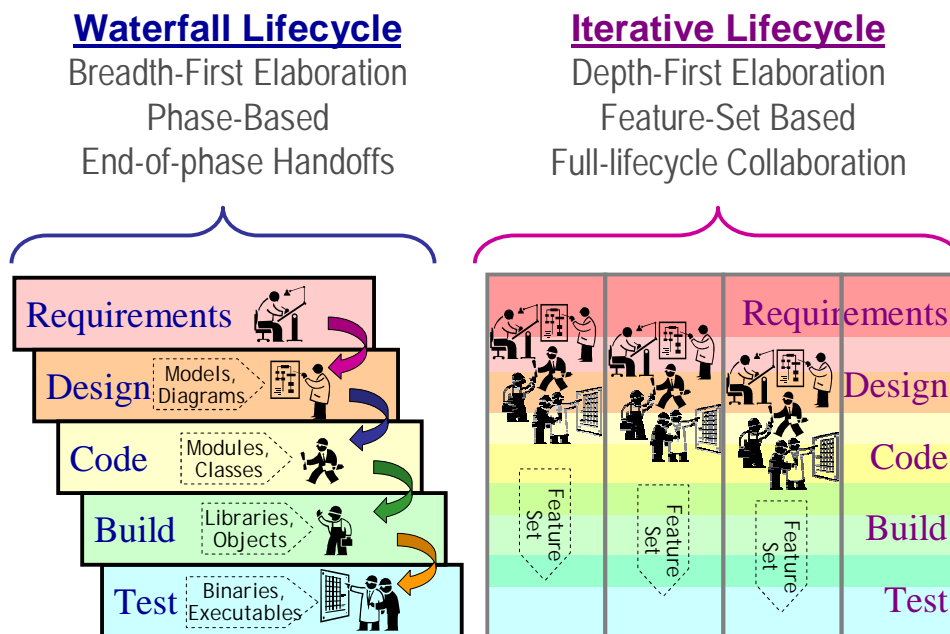
The following Saturn-like diagram depicts the various levels of feedback-loops in effect for the above:



The Agile Lifecycle, Feature Inventory and Set-Based Development

Equating partially developed/tested features with *inventory* (as Poppendieck does [12][14]) is a new way of thinking to many. If all untested code, and unimplemented designs, and partially elaborated use-cases are inventory, then they are depreciable assets whose value decreases over time the longer it takes them to reach the shelf. Hence, software development should ideally proceed in a depth-first fashion, small feature-set by small feature-set (rather than breadth-first like the canonical waterfall model).

Features may require paring-down so they may be completed within the time-frame for the iteration. As a result, large features get broken down into smaller ones, with the most critical functionality developed in earlier iterations and non-critical or non-essential features deferred to later iterations. Each “feature” in an iteration’s feature-set would correspond to the smallest-possible marketable unit of deliverable business value (a *minimal marketable feature* (MMF) [19]).



In Nonaka and Takeuchi’s **The Knowledge-Creating Company** [7] (the book that inspired the agile method known as “Scrum” [20]):

- The breadth-first waterfall somewhat resembles a relay race where each phase-specific role passes the baton (in the form of intermediate artifacts) to the next role in the assembly-lane (right after clearing the end-of-phase review “hurdle”).
- The depth-first iterations more closely resembles a rugby match where all roles collaborate closely throughout the event, and must dynamically adjust and regroup after each short-term win (or loss).
- They then propose that it is possible to combine the best-of-both approaches, and liken the result to “American football”.

In the agile methodology “space”, the methods of Feature-Driven-Development (FDD) [21][22] and Dynamic Systems Development Method (DSDM) [23] contain visible remnants of more traditional methods and could perhaps correspond to the “American football” approach to agile-development.

Hyper-Frequent Iterations and Special Relativity

So if in-progress software features are inventory that increase engineering lead-time for our time-to-market, then it naturally makes sense to employ evolutionary delivery with short iterations that realize discrete, tangible working subsets of the most highly-valued, yet independently useful and usable functionality.

Of course, iterative development is not new, and pre-dates agile methods by at least a decade. If we look at previously existing data and trends regarding traditional waterfall-based and iterative development [24][25], most projects still employ either the waterfall or “V” model as their predominant development lifecycle model, despite the increasing popularity of iterative/incremental, and spiral development models during the “hey day” of object-oriented programming and design in the late 1980s and early 1990s.

For those who were doing iterative development, many had their iterations last at least as long as what had previously been their waterfall-based development phases. A typical iteration was several months or several weeks long, and the basic waterfall or “V” model still applied to the phases within an iteration.

In agile development however, iterations are typically much shorter, usually between 1-4 weeks in length. When iterations last several days or a few weeks, the experience is drastically different from iterations that last several weeks or months. The time-box compression is 4X-5X shorter (e.g. from one month down to one week, or from 3-4 months down to 3-4 weeks). Phases that were previously at least a few weeks or a month are now only a few days or at most one week.

At this short time-span, the boundaries between phases become increasingly blurred as greater cross-phase collaboration is emphasized. It becomes less statistically significant to look at the individual phases within an iteration than it does to look at the iteration itself as a whole. Such phase compression and collaboration “dilation” is reminiscent of the time-dilation effect of special relativity in the sense that:

- The phase-based frame of reference for the waterfall view “breaks down” in the collaborative set-based frame of reference for the highly iterative view. Each perceives the passing of time (and cost-of-change due to rework) very differently from one another. Trying to compare them side-by-side is like trying to compare “apples and oranges.”
- This “changes the rules” for how to compute/measure and communicate/report our results (be they the rules of “Newtonian mechanics” for summing velocities, or the rules of “Phase-containment” and the cost-of-change curve for calculating rework costs).

Microscopic Incrementalism: Continuous Integration, TDD, TBD and TLC

The timebox-compression and collaboration-dilation effects of agile development don’t stop at iteration boundaries. The relative size and duration of development features and tasks becomes smaller as well. Particularly in an environment where *Continuous Integration* is practiced [16], the granule of change for a *Task-Level Commit* [26] may shrink from being weeks or days to instead being days or hours (or even shorter). This is the effect of test-driven development or TDD [27][28], upon task-based development [29]

- The test for a piece of code is written before the code itself is written.
- Developers write only “just enough” code to pass the next test, and then rebuild and retest their code before trying to make any further changes.
- To avoid becoming out-of-sync with the current configuration of the codeline, developers try to integrate their changes with the codeline as soon as each change is coded, correct, and consistent (using *Workspace Update* and/or *Task-Level Commit* [30]).

Coding is Designing (not Construction)

Another important way in which Agile methods differ from traditional waterfall-based methods is in how they regard the activity of writing source code using a programming language:

- Agile methods view programming source code as a *design* activity: it requires creativity and innovation, and is collaborative and unpredictable in scope/schedule.
- Waterfall-based methods view programming source code as a *construction* activity: it requires translation and elaboration of detailed designs, and should be straightforward and predictable in scope/schedule if the designs are specified in enough detail.

In most agile methods, the “construction” portion of the development cycle really begins with the build, where compilers and linkers and code-generators literally “construct” the executable. Programming languages have evolved enough in the past 2-3 decades that modern programming languages like C++, Java, C#, Ada, and even many popular scripting languages (to say nothing of their available libraries and frameworks) are a quantum leap more expressive than the likes of machine-language, assembler, FORTRAN, and C.

Even the pseudo-code that was considered the detailed design of yester-year isn’t as high-level as the more popular modern programming languages in use today. And the power of interactive development environments in the last couple decades makes coding even that much more like a creative activity that affords rapid-feedback.

Tracking vs. Control, and Current Configuration

So if coding is regarded as a creative design activity (rather than a predictable construction activity), the expectation is that such activities should be less rigorously controlled and constrained than construction-activities because they require greater freedom, collaboration and innovation. And anything that stifles or interrupts the productive flow of creative, collaborative work adds a great deal of “friction” to the development process.

Therefore, heavyweight configuration management environments that make developers wait non-negligible periods of time for things like checkout-authorization, presenting and approving all files to be modified prior to coding, protracted build-and-test cycles, waiting for a new baseline to be promoted for use, etc. are considered detrimental to the health of an agile development environment.

Tight controls would be relaxed considerably so that sufficient tracking and auditing could still take place while still allowing the rapid, full-cycle closed-loop feedback that agile development mandates. And development will typically want to use the most recent working and tested state of the codeline rather than wait for a formal build and baselining activity that takes as long to baseline as it takes to complete a multiple development tasks.

Agile development will require what it deems the true “construction” portion of software development (namely build and test) to be as automated and integrated as possible so it can be executed as frequently and as quickly as possible. Development would be able to use the *current configuration* (latest baseline plus approved changes) as the basis for each development task, and rigorous build/test criteria prior to each *Task-Level Commit* in order to preserve codeline integrity and stability while still enabling rapid integration and test feedback.

Key Agile Practices and Their SCM Implications

So what are some of the key agile practices that organizations look to introduce? In a recent article, Peter Schuh [31] identified a number of ways to “add some agile without going to extremes” (note Peter recently presented a CM Crossroads Webcast):

- Automate and share the build process
- Implement a test framework and start writing unit tests
- Adopt a continuous integration process
- Plan and deliver in short iterations and small releases
- Identify and collaborate with your customer
- Manage your test data; don't let it manage you
- Embrace collective ownership and share code

Let's examine the impact that each of these has on SCM.

Automate and Share the Build Process

The build process is a core part of SCM. The fundamental SCM principles that apply are to always deliver a known reproducible build from a clean environment. This has a tendency to lead to a "build master" or build team producing the "golden build", and thus becoming a major bottleneck.

The agile recommendation is to make it easy for everyone to build the system at the push of a button. This saves time:

- It frees programmers from repetitive and mundane tasks
- Reduces time spent chasing down compilation and convergence issues
- Reduces dependencies and bottlenecks – people won't get called back to fix the build.

Continuous Integration

A presentation and demonstration of a continuous integration framework at the recent BCS CMSG event demonstrated the advantages:

- Improving quality due to much more immediate feedback of integration problems
- Improved speed as it is done so often, problems don't lurk unnoticed

Manage Your Test Data

There are various schemes to do this to this including one called *ObjectMother* [32]. If you have any form of database then you will need to address this issue. The pain of updating a test database to reflect the production database tends increase the likelihood that they drift "out of sync" leading to a variety of errors which really should be caught much earlier. The SCM implication is simply that test data and resulting frameworks need to be version-controlled along with the code.

Embrace Collective Ownership

Collective ownership both reduces the risk of the "key team member falling under the bus," and improves quality through practices such as paired programming. It diffuses knowledge throughout the team and avoids isolated silos of knowledge.

This may also encourage a matching SCM attitude that everybody does SCM on a project. Thus the SCM person needs to become a focused contributor to the team rather than an enforcer (and not just "do" SCM but also code etc).

In our experience, this makes for a more enjoyable job, but we've come across quite a few SCM people who like to sit in their little dungeons making occasional forays out to frighten the natives a bit. Being "in the trenches" will also highlight the appropriate automation necessary to make things faster and easier.

Another related aspect of this is the need for the team to become fully conversant with SCM and comfortable with their tool and how to do things like branching and merging - which many are somewhat

reluctant to do. Too many bad experiences in the past perhaps? This is where we can break down some of the barriers between SCM and development.

SCM Concerns for Agile Development

There are some areas where agile proponents view SCM with suspicion, and consider the activities to be “high ceremony and low value”. Let us consider several of these of these.

What about the Cost of Deployment/Upgrade?

The flattened cost-of-change curve is all well and good when it is dominated by the cost of making development changes. But what if it is dominated by the cost of deploying and upgrading a change? Suppose it's easy to make the change, but expensive to deploy it to every applicable instance of the software stored on an end-user's device? This might be the case if:

- There are corresponding hardware changes required with the software change
- The system is a highly-available one and there is an associated system down-time where the system cannot be available for all or part of the upgrade.
- The system is deeply tied to and/or used for significant business process workflow or decision-making-support, and an upgrade/downtime requires new training to be developed and users to be re-trained.

In each of these cases, there are no “pat” easy answers. We can try and apply lean thinking and agile principles to mitigate the risk and rework for such events, but that may be the best we can do.

What about the Cost of Unforeseen Feature/Requirements Interactions?

Working in microscopic increments may be swell! In theory, I can see that the smaller the change/feature/iteration and the tighter the feedback loop, the smaller the rework associated with any such task/feature/iteration: there is no code that is more adaptable than no code (*code that doesn't exist yet*).

But isn't one of the main purposes of up-front requirements, analysis, and design to be able to anticipate what would otherwise have been unanticipated impacts of certain requirements upon other requirements, and avoid having to rework the architecture because of such last-minute discoveries? By developing features in business-value order rather than impact/risk-driven order, aren't you basically rolling the dice that you won't have to scrap something major and rework it all over from scratch?

In theory, the answer to the above question is “Yes!” In practice, it is “Hardly ever!” provided that your team is very disciplined about doing aggressive refactoring and dependency management in even the smallest possible increments of change:

- If refactoring and integration and testing are rigorously and ruthlessly performed at the fine-grained levels of minimal mergeable tasks, and minimal testable modifications, then you should be able to realize the benefits of *emergent design* [9].
- If you slack off and/or don't have enough discipline to do it so thoroughly and frequently, you may need to make a trade-off by doing a little bit more up-front requirements and design (which FDD does, for example [21]).

Merging and Building: Continuous “Push” Integration vs. Discrete “Pull” Assembly

In the agile world, developing a fully executing system is a *holistic gestalt* approach in which *the whole is greater than the sum of its parts*. In the traditional SCM world, many of us have grown accustomed to a more separationist assembly view in which we believe that the whole is *exactly* the sum of its parts!

Crossroads Journal

A Monthly Publication for
Software and CM Professionals

- We like being able to separately isolate the changes for each feature, fix, and enhancement so that we can “back it out” if it breaks, or propagate (merge or reintroduce) it to one or more other projects or project variants, or to a remote geographical development site.
- We may prefer to be the ones doing the building/merging rather than having developers attempt it themselves (particularly for a remote site). This is sometimes called an *Integrator-Pull* model of integration (as opposed to a *Developer-Push* model of integration) [34].
- We may prefer that each development task be based off the latest stable (and “blessed”) development baseline instead of the off the “latest and greatest” state of the codeline (which might not be baselined).

Fortunately there are ways to accomplish the same objectives without requiring the same old methods – but some compromising and trade-offs will be necessary. Few people like having to do multi-project, multi-variant, or multi-sited development and to have to support them concurrently. If you really and truly can get away with having only 1-3 codelines, then challenge yourself to do so, and then do whatever it takes to stay that way.

If you cannot avoid multi-project, multi-variant, or multi-site development, then:

- Use the *Mainline* pattern [26] and maintain a codeline for each concurrent release that must be actively supported.
- When possible, prefer an architectural solution to provide variant functionality (e.g., conditional compilation, run-time configuration, and principles and patterns of component-based, service-oriented, product-line architecture) instead of a branching solution.
- If time-zones aren't too far apart and WAN/LAN performance isn't an issue, have remote sites use the same central repository rather than a replicated/mirrored repository. If they are too far apart or if network performance is a problem, then use a site-specific integration branch and have it merged to (and updated from) the “master” integration branch as frequently as feasible.
- You can still use daily or nightly (or more frequently) automated builds to verify the stability, consistency, and reproducibility of the development codeline. (This also works for site-specific codelines.)
- Use *Continuous Update* with *Private-Branched* or *Task-Branched* [30] to group together multiple tasks for the same overall feature/fix so that they can still be easily propagated to other codelines while at the same time staying reasonably up-to-date. Don't forget to propagate fixes from the older (least evolved) codelines that need them to the more recently (and more evolved) codelines.
- Use *Continuous Staging* [33] to sync-up from multiple sites and/or multiple component teams.
- Use *Task-level Commit* and *Task-based Development* (TLC and TBD [29]) and appropriate pre-commit entry criteria to ensure that the tip of the codeline always corresponds to the “current configuration” (the last official baseline plus latest approved changes). Even if it's not a baselined development configuration, if it is a “true” current configuration, then it's good enough for development as long as it is still identifiable, correct, consistent, reproducible, and auditable.
- Collaborate with the development team from the very beginning to ensure their builds are as clean, automated, and reproducible as possible, and that the codeline policy is enforced (preferably self-enforced).

Change Tracking and Story Cards

Tracking and reporting the status and content of requests and changes across many artifacts and activities can be extremely unwieldy. We often require sophisticated tools to assist us in this complex and tedious endeavor. Agile approaches strive to minimize the number and size of non-code artifacts that are created. Fewer artifacts and documents mean fewer items to tracked, and less effort and complexity to track them.

XP teams use physical story cards for project planning and status reporting, whereas in traditional software development, tools such as defect trackers are used to track defects electronically and provide workflow etc. Tools bring the advantage that requirements/changes/defects can be tracked, reported and versioned rather easily to those not in the immediate physical proximity the “information radiator” (the wall or whiteboard showing the story-cards, and progress charts).

In the case of XP, its proponents say index cards are preferred in favor of tools. Their physical limitations force developers and customers to keep things short and simple, and physically associate a request, its corresponding requirements, and its tests with the same physical artifact (in XP, the customer acceptance tests are the detailed requirements). Scrum and other agile methods appear to be less insistent upon the use of index cards, and less resistant to the use of tools (such as spreadsheets or change/request tracking systems).

Most experienced change and configuration managers would gasp at the seeming low-tech inefficiency of index cards as means to track and report conformance and status. At the same time, it is hard to combat the efficacy of index cards to engage customers in a more participative and collaborative dialogue for eliciting requirements, and drawing simple diagrams.

And yet even the simplest of spreadsheets and tracking tools provides a mechanism for fast and easy reporting, querying, searching, sorting, as well as for real-time dissemination across the project's organization and stakeholder-sites (not to mention affording more efficient and reliable storage, record retention, archival, and retrieval/recovery). Keeping “stories” in a tool also lets you apply a simple workflow see how the work is progressing.

The reality is that it doesn't have to be an either/or choice. By keeping things as simple as possible, and remembering to prefer face-to-face interaction over face-to-machine interaction, the “right” amount of process using a simple and “smart” tool can yield the best combination of increased productivity and better coordination. The bottom line is really to do what you know works for you, and keep it as simple as possible, applying the principles of lean development [12][13][14] every step of the way.

What Happened to Traceability?

Don't you still need traceability? Even if you don't use it to assist with impact analysis or estimation, don't you still need traceability to ensure your software conforms to its promised functionality, or to ensure compliance with contractual or organizational standards that require traceability?

Right you are! Even in a low-formality, high-discipline method like XP, traceability for specification conformance is ensured by having a mapping between story-cards and their corresponding customer acceptance tests. In the case of XP, the written customer acceptance test is the formal requirements specification, and is written on the back of the same index card as the user-story (high-level use case description) was. Plus the story-card eventually indicates the estimate, the release and iteration, and completion date (so it is also used for status accounting).

For other agile-methods besides XP, it is more common to see the use of simple tracking tools. Here too it is very easy (and exceedingly common) to capture planning, estimation, and completion status. Plus change-requests typically have a unique name/identifier associated with them that is easy to reference from

a use-case and/or a test-case (and vice-versa) as well as with a checkout/checkin in the version control repository.

So those are some basic ways of capturing request-level and task-level traceability that are minimally intrusive/invasive and impose low-friction on the development team. If more formal traceability is mandated by contractual, organizational, or industry-imposed standards, then keep in mind the following tips for making traceability and documentation as lean as possible:

- **Encapsulation and Modularity** – if we do a good job of applying encapsulation and modularity to achieve separation of concerns and minimize dependencies, then it becomes less necessary to trace to fine-grained levels of details because odds are that impacts will be localized at the encapsulation boundary of the use-case, package, or class: anything that impacts one element of a well-encapsulated entity will likely impact many elements internal that entity and little of no elements external to that entity. This means it can be sufficient to trace only to the coarse-grained level of features/use-cases and classes/modules instead of their individual requirements/routines.
- **Documentation and Locality** – the *Principle of Locality of Reference Documentation (LoRD)* [35][36] 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.” In other words: “out of sight, out of mind!” We can apply the LoRD principle to help minimize traceability and eliminate redundancy by using fewer artifacts and decrease the “distance” between related pieces of information in several commonly practiced ways:
 - Documentation in the Code (e.g. JavaDoc, Perl5 PODs)
 - Documentation in the Model (e.g., UML with formal/informal text annotations)
 - Documentation in a Database with links/metadata (e.g. Telelogic DOORS, IBM/Rational RequisitePro)
 - User Reference Manual as Requirements Spec
 - Acceptance Tests as Requirements Spec
 - Interface/Implementation Co-location

Conclusion

We think that the traditional and the agile camps need not fear each other. SCM can be relatively easily implemented in agile processes without removing the agility, and yet providing that bedrock of security which more traditional methods like. It requires adapting of the methods implemented, while remaining true to the basic principles of SCM.

Thus we would define **Agile SCM** as the pragmatic application of sound CM principles & practices, in accordance with Agile Values, and using Lean Thinking, to serve the needs of the Business!

References

- [1] *Configuration Management - Agile versus Traditional*; British Computer Society Configuration Management Specialist Group special event; London, October 2004. (see <http://www.bcs-msg.org.uk/events/e20041012.shtml>)
- [2] *Agility* from **Agile Software Development Ecosystems**, by James Highsmith; Addison-Wesley, March 2002
- [3] *Agile Software Development: The People Factor*, by Jim Highsmith and Alistair Cockburn; IEEE Computer: November 2001 (Vol. 31, No. 11), pp. 131-133

- [4] [Retiring Lifecycle Dinosaurs](#), by Jim Highsmith; [Software Test and Quality Engineering Magazine](#) (STQE), July/August 2000 (Vol. 2, No. 4)
- [5] [Software is not a Product](#), by Phillip Armour; Column "The Business of Software" Communications of the ACM August 2000 (Vol. 43, No. 8)
- [6] **The Laws of Software Process**, by Phillip G. Armour; CRC Press / Auerbach Publications, September 2003.
- [7] **The Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation**; by Ikujiro Nonaka and Hirotaka Takeuchi. Oxford University Press, 1995 (see summary and review at <<http://www.xp123.com/xplor/xp0402/index.shtml>> and <<http://www.meansbusiness.com/Knowledge-and-Learning-Books/The-Knowledge-Creating-Company.htm>>
- [8] [The Agile Manifesto](#), also appearing in [Chapter 1: Agile Practices](#), pp. 3-11 of [Agile Software Development: Principles, Patterns, and Practices](#), by Robert C. Martin; Addison-Wesley, November 2002
- [9] [The New Methodology](#), by [Martin Fowler](#); created July 2000, revised April 2003; also published in abridged form as "[Put your Process on a Diet](#)", [Software Development Magazine](#); December 2000 (Vol. 8, No. 12)
- [10] [What Is Agile Software Development?](#), by Jim Highsmith; [Crosstalk – the Journal of Software Defense Engineering](#) (special issue on Agile Software Development), Vol. 15, No. 10, pp. 4-9
- [11] [Agile Software Development: The Business of Innovation](#), by Jim Highsmith and Alistair Cockburn; IEEE Computer: September 2001 (Vol. 31, No. 9), pp. 120-122
- [12] [Principles of Lean Thinking](#); by Mary Poppendieck; **2002 Conference on Object-Oriented Programming Systems, Languages, and Applications** (OOPSLA 2002); Seattle, WA, November 2002; (see <http://www.poppendieck.com/papers/LeanThinking.pdf>)
- [13] [Lean Programming](#); by Mary Poppendieck; [Software Development Magazine](#), May-June 2001 (Vol. 9, No. 5 and 6 -- see full article at <http://www.poppendieck.com/lean.htm>)
- [14] **Lean Software Development: An Agile Toolkit**; by Mary and Tom Poppendieck; Addison-Wesley 2003 (see <http://www.poppendieck.com>)
- [15] **Software Engineering Economics**; by Barry Boehm; Prentice Hall PTR, October 1981
- [16] **Extreme Programming Explained: Embrace Change**; by Kent Beck; Addison-Wesley, 2000
- [17] [Reexamining the Cost of Change Curve, year 2000](#); by Alistair Cockburn; [XP Magazine](#), September 2000
- [18] [Examining the Cost of Change](#); by Scott Ambler, [Agile Modeling Essays](#) excerpted from the book **The Object Primer, 3rd ed.: Agile Model-Driven Development with UML2**; by Scott Ambler, Cambridge University Press, 2004
- [19] **Software by Numbers: Low-Risk, High-Return Development**; by Mark Denne and Jane Cleland-Huang; Addison-Wesley, 2004
- [20] **Agile Software Development with Scrum**; by Ken Schwaber and Mike Beedle; Prentice-Hall 2001
- [21] **A Practical Guide to Feature-Driven Development**; by Stephen Palmer and John Felsing; Prentice-Hall PTR, February 2002
- [22] **Agile Management for Software Engineering**; by David J. Anderson; Prentice-Hall, 2003
- [23] **DSDM: Business Focused Development** (2nd ed.); by DSDM Consortium, Jennifer Stapleton; Addison-Wesley, January 2003

- [24] **Agile and Iterative Development: A Manager's Guide**; by Craig Larman; Addison-Wesley, 2003
- [25] **Managing Software for Growth: Without fear, control, and the manufacturing mindset**; Addison-Wesley, 2003
- [26] **Software Configuration Management Patterns: Effective Teamwork, Practical Integration**; by Stephen P. Berczuk and Brad Appleton; Addison-Wesley, November 2002
- [27] **Test-Driven Development: By Example**; by Kent Beck; Addison-Wesley, 2003
- [28] **Test-Driven Development: A Practical Guide**; by Dave Astels; Prentice-Hall PTR 2003
- [29] *SCM Patterns: Building on 'Task-Level Commit'*; by Austin Hastings; CM Crossroads Journal, June 2004 (Vol 3. No. 6)
- [30] *Codeline Merging and Locking: Continuous Updates and Two-Phased Commits*, by Brad Appleton, Steve Konieczka and Steve Berczuk; CM Crossroads Journal, November 2003 (Vol. 2, No. 11)
- [31] *Traditional with a Twist*, by Peter Shuh in **Better Software Magazine**, July/August 2004.
- [32] *ObjectMother: Easing Test Object Creation in XP* (XPUniverse, 2001)
<<http://www.peterschuh.com/it/index.html>>
- [33] *Continuous Staging: Scaling Continuous Integration to Multiple Component Teams*; by Brad Appleton, Steve Konieczka and Steve Berczuk; CM Crossroads Journal, March 2004 (Vol. 3, No. 3)
- [34] **Software Configuration Management Strategies and Rational ClearCase**; by Brian White; Addison-Wesley, August 2000.
- [35] *Locality of Reference Documentation*, by Brad Appleton; drafted January 1997.
<<http://c2.com/cgi/wiki?LocalityOfReferenceDocumentation>>
- [36] *Extreme Locality*, by Brad Appleton; **Agile Times**, February 2001 (Vol. IV, No. 1)