Implementing Agile SCM in the Enterprise

Kevin A. Lee
kevin.lee@buildmeister.com

Synopsis

It does not seem long ago that agile development methods were introduced as new and somewhat controversial mechanisms for delivering software development projects. Today however, agile development practices such as incremental development, test driven development, refactoring and continuous integration are commonplace and have been accepted and absorbed as commonsense approaches to software development. Whatever your beliefs or experience, you cannot deny that agile development projects can and have proved successful in delivering functionality on time and to budget. This whitepaper discusses a specific aspect of agile development - the concept of ***Agile Software Configuration Management****, or Agile SCM - a well-designed, light form of SCM that can be used by software development projects practicing agile development methods. It particular it discusses how Agile SCM can be successfully implemented not just on individual projects but across projects in large enterprise organizations where specific operational and governance requirements need to be met.*

Agile Development

Agile development is an umbrella term that encompasses a number of distinct methods such as eXtreme Programming (XP), Dynamic Systems Development Method (DSDM) and the Scrum project management method. All of these methods are related in that they subscribe to the basic premises of the Agile Manifesto [AgileM01], co-written by several luminaries of the software development world. The fundamental principle of the Agile Manifesto is in recognizing the value of:

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

The approach that most agile methods share is the direct involvement and interaction with users or customers, and the development of functionality in frequent, short iterations (usually between two to twelve weeks). Typically, at the start of each iteration, agile teams negotiate with the customer to define new features or change requests. These are estimated by the developers and then subsequently prioritized by the customer for the next iteration as illustrated in Figure 1.

A **backlog** of any features or change requests that have not been implemented in an iteration are kept and, together with any new requests, are re-prioritized by the customer for the next iteration. Developers are permitted to work on requests for the current iteration or to carry out re-factoring and simplification of existing code as necessary. The intention behind this is to keep design simple and prevent gratuitous feature bloat. Code is also continuously integrated; which means it is implemented, tested, and committed frequently in very small units, with an automated build process being invoked at commit time to check for integration errors. Although agile methods generally proscribe continual testing at the development level, for internal, more system oriented testing **Feature Drops** are often made available. These are not intended to be end of iteration, customer consumable builds. Rather they are development builds with specific completed and testable features. Feature Drops allow “career testers” to exercise a product and customer representatives to visualize its content and steer it in the right direction.

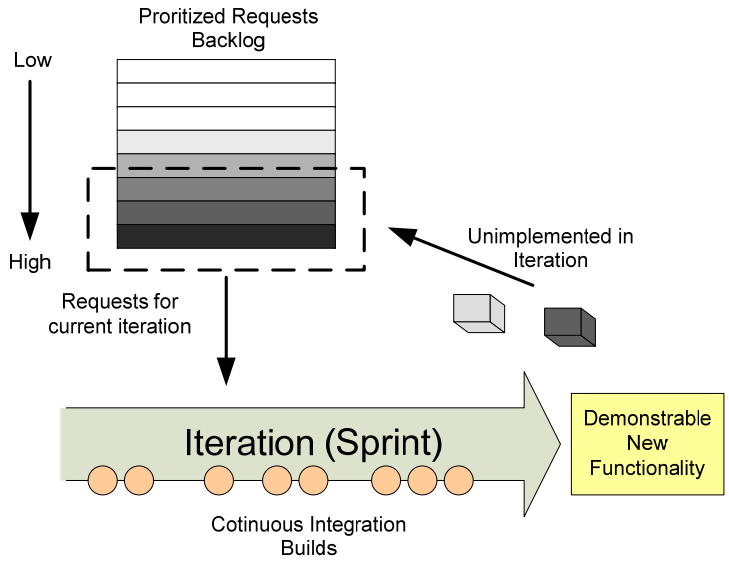

Figure 1 - Example agile method

In essence agile development practices are simply part of a common-sense **risk driven development** approach. Their implementation has consequently evolved beyond their initial agile development roots and many individual agile practices are now embedded in organizational software development processes – processes which their owners would probably not consider fully "agile".

Feature Driven Development

Feature Driven Development (FDD) is generally accepted as an agile method in that it promotes incremental change by small teams, however it also requires more traditional upfront design and class (or component) ownership rather than collective code ownership. In FDD, a Feature Team is constructed that consists of a small team of developers (say 4 or 5) who are experts on the particular classes that will be required to be changed in order to implement a specific new feature. Developers might potentially be part of more than one Feature Team, but for each feature a chief programmer (or Feature Lead) is assigned. When the feature has been completed it is typically up to the chief programmer to integrate it into the product's overall integration build. FDD is a good compromise for those seeking "agility" but in the form of a more structured implementation.

Agile SCM

Agile SCM as a concept in its own right was probably first discussed in detail by Brad Appleton and Stephen Berczuk in their book *Software Configuration Management Patterns* [Berczuk03] and on the SCM portal CM Crossroads.[1] One of their observations was that:

> ...Configuration Management is going to be critical "fulcrum" in leveraging a balanced and effective set of SCM processes and criteria for agile development methods. With such a large emphasis on lean and lightweight from the "agilists," CM on agile projects will need to be less

[1] www.cmcrossroads.com

> intrusive/invasive (what Grady Booch would call low-friction) to allow agile projects to succeed while at the same time not being so minimal (due to overreaction) as to contribute to their failure.

The main message in this observation is that although SCM on agile projects tends to be more lightweight and less visible, SCM itself is a critical requirement for such projects. The majority of agile projects currently adopt entry-level or lightweight version control tools such as CVS, Subversion, or BitKeeper - tools that have limited but sometimes sufficient functionality for a large number of projects. They are also tools that tend to be less intrusive and value the development experience. Although such tools may be sufficient for individual projects, in most cases they do not meet the agile AND enterprise SCM requirements of large organizations.

In general, SCM is about "**governance**" of the development process; that is, SCM allows projects to retain a measure of control, but at the same time should also allow developers the freedom to create within the controlled process. One of the paradoxes of agile development practices such as continuous integration and test-driven development is that if implemented correctly, they can actually enable a well-disciplined and almost self-governing approach for SCM. For example, **test driven development** requires that on every code change, agile developers must first write a unit test, then write sufficient code just to make the test work, and then subsequently refactor as necessary to complete the change. The code change is committed (or checked in), and its unit tests become part of the integration suite. Any side-effects of the change are made immediately visible through the integration build mechanism compiling and executing the unit test suite -- any problems that are found can then be fixed immediately. Such an approach will generally lead to better quality code being committed to the SCM repository.

In Agile SCM, the governance model should be a natural part of day-to-day development activities and in all consequence transparent to the developers. To understand this model in more detail, let us look at some different characteristics of SCM and how they would typically be implemented to support agile development methods:

- **Versions**. Agile projects implement refactoring, that is they implement the simplest solution possible first and refactor code to redesign the solution as and when necessary. This means that code versions are continually updated in the SCM repository and refactoring operations such as add, delete, move and rename need to be well supported by the SCM tool and the tools into which SCM integrates.
- **Branches**. Agile projects implement simple branching strategies, typically an Active Development Line[2] and maybe a Release Prep Line. The Active Development Line is used by developers to commit their changes and is the means by which continuous integration builds are carried out. The Release Prep Line is used to stabilize or engineer a release before making it available to customers; developers are typically not allowed to commit changes to it.
- **Workspaces**. Developers typically have a single private workspace initially pointing at the collective set of latest versions of the Active Development Line. Their workspaces are updated at a minimum when they start work on a new feature or change request and just before they commit their changes to the Active Development Line to check for integration issues.

[2] www.cmcrossroads.com/bradapp/acme/branching/

- **Labels**. As with traditional SCM, labels (or baselines) are placed at significant milestones on a collective set of code versions, at a minimum on every release build, so that it is possible to reproduce a build environment if necessary.
- **Builds and integration**. An automated build process is a key factor of successful agile development. The build process typically monitors the Active Development Line for commits and, if found, automatically executes (after a grace period) an integration build and unit test. Notification of the success or failure of this build is a key communication factor in agile teams.
- **Change control**. There is an implicit authorization process with agile development teams – at least at the implementation level. Developers are authorized to make changes based on customer priority or refactoring as necessary. Requests are recorded either in change control systems, or even for more informal projects (particularly within eXtreme Programming projects) on cards or flip charts.
- **Deployment.** Agile projects deploy testable applications frequently, particularly where application server environments are in use (e.g. J2EE/Microsoft.NET application servers). Therefore the deployment of frequent incremental baselines or change requests needs to be supported by the SCM tool, as well as the ability to report and query on the versions of the application in each of the environments.

Although SCM characteristics such as these are typical in agile processes, they are just as likely to be "tuned" depending on the amount of agility that a particular project requires. For example, some projects might not be able to build on each commit but instead initiate a single daily or nightly integration build.

Agile SCM and the Enterprise

One of the first observations to be made clear is that SCM is much more than just version control. It has been defined, refined and implemented to help large enterprise organizations and projects control and manage their entire software development lifecycle. It is generally agreed that any SCM solution would supports the following four functional areas:

- **Version Management** - Enables the secure and controlled storage, retrieval and versioning of all project assets (source code, documentation, test results and so on).
- **Change Management** - Enables the capture, tracking and management of all types of project changes (requests for change, enhancements, defects) and direct comparison of these changes to the versions of the project's assets used to implement them.
- **Build Management** - Enables the consistent and reliable construction (build) of software applications and reports on their contents and results.
- **Process Management** - Enables the reporting of actions, history and milestones and provides the capability to customize or enforce development workflow and procedures.

Taken in isolation, individual agile projects will typical concentrate on the "**bottom-up**" view of SCM – implementing basic Version Management and a lightweight form of Build Management. Enterprise organizations as a whole however are more concerned with the "**top down**" view of SCM –

implementing Change and Process Management to control and secure development efforts across their organization. Reaching a compromise between these two implementation views is and will be the main challenge for enterprise organizations implementing Agile SCM.

In theory, there is no reason why you can't use any SCM tool to support agile development practices. You certainly don't have to use all the features of a tool, and most tools allow some degree of process customization, from heavyweight to lightweight and all places in between. The danger is that with enterprise tools, some organizations are tempted to use all the "bells and whistles" to define a heavyweight SCM process - simply because the tool can support it. However, such a process won't necessarily meet the requirements of your project. To find the right process and level of customization, you should first identify and define your requirements, which means understanding exactly what your development process is, or should be, and then determining how SCM can support it. Also, projects don't exist in isolation; they are normally one of many projects in an organization. Often, an enterprise organization contains a mix of both agile and traditional plan-driven projects. At the end of the day however, the enterprise context is often the strongest factor in deciding which SCM toolset is chosen. There are typically two main reasons for doing this:

- To reduce the total cost of ownership of the toolset and its processes
- To be able to conform to a desired (or mandatory) compliance or regulatory framework

Total cost of ownership is often a subjective issue, since it includes quantifiable aspects such as license, administration, and support costs, but also other subjective aspects such as capability or scalability. Enterprise toolsets often have higher licensing, administration, and support costs (certainly initially), but if implemented correctly, these toolsets can increase organizational capability as a whole. They also have proven scalability, with a single, consolidated infrastructure able to support large, geographically dispersed development organizations. As noted above, the main danger of such a toolset is the temptation to use more functionality than is necessary, which can strangle agile projects. An organizational SCM framework will need to be established, and its implementation should be in some way configurable so as to meet the needs of different projects.

Recent industry regulations such as Sarbanes Oxley, Basel II, and CFR 21 Part 11 can place onerous overheads on SCM processes, particularly with respect to change control. Although practices such as "segregation of duties" - not allowing developers to deploy to live environments - should be implemented on all projects, a complex and rigorous enforcement of change control on agile projects can strangle them. However, the business cost of not meeting these regulations is massive, so while agile projects might want to avoid non-essential governance practices, they have to accept some additional overhead in most cases. The good news is that if implemented correctly, an automated SCM toolset can take on most of this governance aspect, allowing businesses to maintain organizational control, while allowing individual projects and their developers to work on the creative aspect of developing new software functionality to solve business problems.

Recommendations

Given all that we have discussed so far, let us now consider how a balance between governance rigor and agility can be achieved for agile projects (using an enterprise SCM toolset) in an enterprise organization. In this section I list a number of recommendations. Each recommendation could be

implemented in its own right; however I believe that the majority would typically need to implemented to successfully implement Agile SCM.

1. Implement a simplified branching strategy.

Agile projects implement simple branching strategies. In most SCM tools branches can be created easily and cheaply. There is therefore often a temptation to define a more complex branching strategy than is strictly necessary. agile projects actively encourage continuous integration and refactoring, but if developers are isolated on a branch, problematic or complex merges can occur. This is particularly true in the case of namespace refactoring (renaming, adding, or deleting files or directories). Consequently, most agile projects implement a specific branch to act as the **Active Development Lin**e and which the developers work on directly.

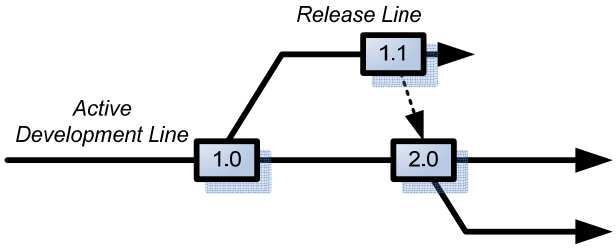

Figure 2 - Agile branching

One of the main reasons for creating branches is for support after the release of the application has been made (either internally or externally). If issues are found that need to be resolved a decision needs to be made on whether to fix the issue on the Active Development Line or create a **Release Line** to collect the fixes. A diagram illustrating this structure is provided in Figure 2. Although it is possible to fix on the Active Development Line, the customer would receive new features as well as the fix. This is not necessarily an issue if the features have been continually tested and can be "configured" to be enabled. However, if this is not possible then use the Release Line approach. You should still try and make the Release Line short lived though, use it infrequently and integrate any changes back in as early as possible.

2. Use optimistic file locking and checkout.

In combination with a simplified branching strategy, agile projects also tend to configure optimistic locking in their SCM tool. Optimistic locking allows multiple developers to perform concurrent changes to a single file (i.e., checkout the same version on the same branch of a single file). Most SCM tools have a process around this capability that allows the first developer to check-in but force subsequent developers to progress through a guided merge process as illustrated in Figure 3. Since agile projects aggressively write unit tests, any merging can usually be successfully validated through private developer build and/or automated integration builds.

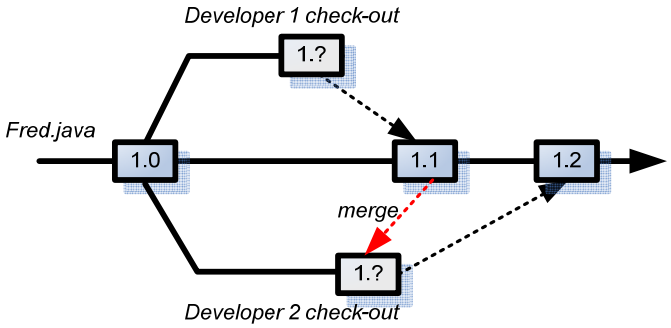

Figure 3 - Optimistic locking

As well as optimistic locking, a large number of Agile projects use optimistic checkout. In such a model there is no specific check-out action. Instead developers make the changes they need directly on the files in their workspace and then subsequently upload their changes to the repository. The repository is then updated with new versions for the changed files. In the optimistic checkout model it is good practice for developers to synchronize their workspace before commit and resolve any merges in place before committing. Optimistic checkout works well where developers are remote and not connected to a high bandwidth network.

3. Perform incremental check-in.

When developing a new feature or change request, there is always a temptation to delay check-in (or commit) until the work has been fully completed and unit tested. Although this approach sounds fine in principal it actively delays integration potentially causing unexpected rework. In agile projects, if test driven development is implement correctly, then the skeleton unit tests for a feature or change request are developed first; then code is implemented to make the unit-tests pass. This is a process that is repeated over time, with incremental functionality added to the unit-tests and implementing code. Since this is a continual process, with buildable and testable code throughout, the feature or change request can actually be checked in to the repository before it is complete! Although it might not be fully functional (and worth acceptance testing), continually integrating the code will highlight any issues at an early stage and can save significant effort and rework in the long term.

4. Automate the build process.

Single branch development and incremental check-in can only work when automated builds and tests are executed frequently. Most agile projects configure their build process to execute on every developer check-in (or commit). As well as compiling code, such a build process also validates new code to see if it conforms to pre-defined coding conventions and executes unit tests where necessary. In agile development methods, this practice is often called **continuous integration**. Its desired result is to expose integration problems as early as possible so that they are easy to address, and also to have a built, tested, and potentially releasable build at every stage of the project. Continuous integration is often intricately linked with the practice of test-driven development. This is because developers need to implement unit tests for all aspects of their code to validate not only that the build has been compiled, but also that it conforms to some minimum level of functional quality. In SCM terms, the agile project's build process is set to monitor the project's integration branch and then execute a build

script on check-in. This is typically carried out by build execution tools such as CruiseControl[3] or IBM Rational BuildForge[4] - there are also a large number of similar tools.

5. Stage and re-use pre-built binaries.

Building any reasonably complex software application can take time, from several minutes to several hours. In agile projects, developers will be delivering and integrating small changes frequently; therefore they obviously cannot wait for a two-hour build to complete before getting any feedback. To avoid this situation, agile projects typically "**stage**" and re-use pre-built binaries, and only rebuild the whole system when necessary (for example, nightly or weekly). This can effectively create a **build pipeline** where the output of one build process can trigger the start of another – a process which is illustrated in Figure 4.

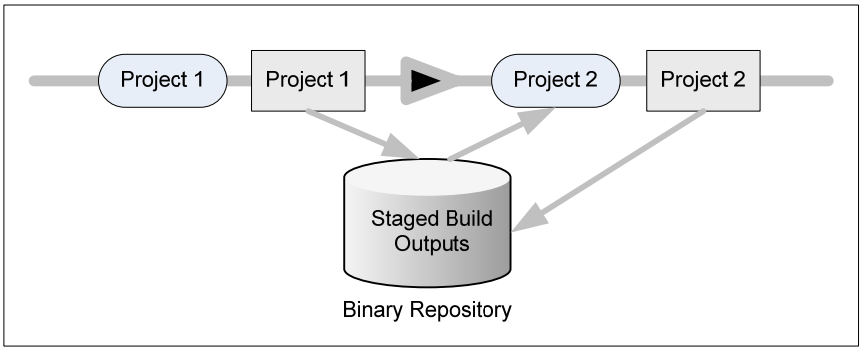

Figure 4 - Build pipeline

There are a number of ways of staging binaries. The SCM tool itself is often used to stage binaries, with labels or baselines placed down on related versions to indicate a composite set of re-usable binaries. For C/C++ projects, the ClearCase clearmake or ElectricCloud ElectricAccelerator[5] utility have build-avoidance mechanism that can be used to automate much of this staging and re-use process (as well as reducing the overall build time). For Java projects, an alternate approach is to stage pre-built libraries in a dependency management tool such as Apache Ivy[6] or Apache Maven[7]. These tools are particularly appropriate where a project re-uses a significant amount of open-source components or where there are issues with transitive dependencies (i.e., where libraries are dependent on other libraries).

6. Release and deploy continually.

When it comes to releasing your application, try to release all your files rather than releasing individual sets. Most agile projects prefer to deploy the whole or **composite application** each time rather than just the individual files that have changed. Although this is not a hard and fast rule, releasing the whole application in the same way each time tends to make the process more repeatable and prevents problems with missing files. This practice is more important for agile projects than

[3] http://cruisecontrol.sourceforge.net

[4] www-306.ibm.com/software/rational/buildmanagement

[5] www.electric-cloud.com/products/electricaccelerator.php

[6] http://ant.apache.org/ivy/

[7] http://maven.apache.org

traditional ones, because they produce executable, testable, and releasable applications to be deployed at the end of every iteration. Note that this also depends on the deployment type of the application. If it is Web portal, then this approach is recommended, however if it is a consumer application (running on, say, Microsoft Windows), then obviously a full release cannot be made to the customer each time and therefore patches (containing individual files) will be required. It is good practice to define a **Deployment Unit** (a consolidated set of deployable files and installation scripts) that is to be deployed rather than deploy the individual files themselves. This approach typically simplifies the deployment process and makes deployment roll-back (on failure) much simpler.

If your application needs to be deployed to multiple environments, then you should secure and lock down environments to only those users that are directly permitted access to them. As an example developers might have permission to deploy (or invoke deployment to) Development and Integration Test Environments, build engineers would have permission to deploy to System or Acceptance Test Environments and deployment engineers (who are usually in the operations department) would have permission to deploy to the Production Environment. It is good practice to define your deployment lifecycle early on and automate it as much as possible. An example of a typical deployment lifecycle is illustrated in Figure 5.

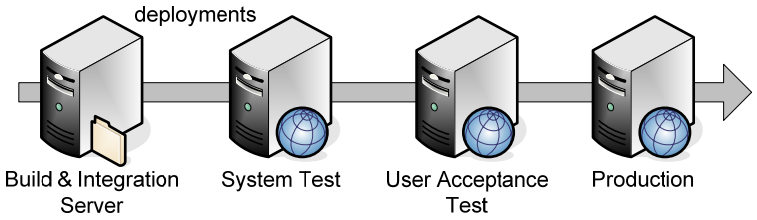

Figure 5 - Typical Deployment Lifecycle

Most agile projects tend to delay the realities and consequences of continuous deployment until a later iteration. However doing so can cause rework since the application might need to be implemented to allow configuration for more complex environments (particularly production). It is therefore good practice to implement deployment strategies at an early stage. For more information on deployment strategies see *The Buildmeister's Guide – Achieving Agile Software Delivery* [Lee07].

7. Limit the use of replicated repositories.

Distributed agile projects require unrestricted access to a single codeline. SCM distribution technologies typically replicate databases from site to site and lock files or codelines from being changed at multiple sites. In essence this violates the practice of optimistic file locking from being implemented. For this reason, agile projects tend to avoid replication technology. They either implement a centralized repository (usually accessed via HTTP) or use caching technologies to cache frequently changed files at local sites.

8. Avoid over-implementation of change control.

Enterprise SCM tools can provide very powerful and sophisticated change- and defect-tracking capabilities. Although a number of agile projects have avoided using change control entirely, more and more projects are looking at how best to implement lightweight change control to help them meet their organization's compliance and regulatory requirements. Enterprise SCM tools can certainly be used for automating the capture, prioritization, and allocation of a typical change request and feature backlog - as mentioned at the beginning of this whitepaper. However, too much process enforcement

or micro-management can make it time-intensive for developers to start work on new change tasks. If not done with care, this can significantly slow down the overall development process and reduce the productivity of the agile project.

To avoid this, agile projects typically implement a lightweight or customizable change request workflow. Such a workflow should allow traceability from business requests to technical implementation tasks as illustrated in Figure 6 – with minimal effort and management.

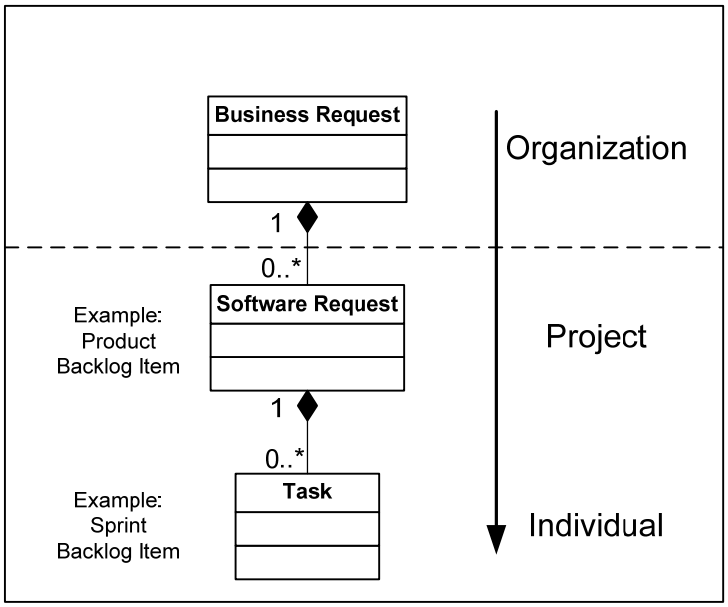

Figure 6 - Agile Change Management

This lightweight approach is particularly difficult to implement where the SCM tool is being used across an entire organization. Some projects in that organization might require more process control and management, a lot of which can be embedded in the tool. However, agile projects might not need or desire the same amount of process control and management. Organizations working with such environments have been successful by allowing the change request workflow to be configurable by each project. However, such an implementation requires detailed knowledge of both the tool and the requirements of different project processes and should not be undertaken lightly.

Summary

In this whitepaper, I have discussed the concept of Agile SCM and described a number of recommendations regarding how it can be implemented within the context of an enterprise organization. Hopefully, this discussion has convinced you that Agile SCM can be achieved in your own environment. It is worth noting that the implementation of an Agile SCM environment is not something that should just be restricted to agile projects. There are many projects that do not consider themselves as agile, but yet have similar SCM requirements. These tend to be smaller IS/IT projects where the configuration of a similar environment would be very practical. It is probable that larger projects could learn something by implementing more lightweight and well-designed SCM process based on some of the practices described here.

In summary, there is no reason why an enterprise SCM toolset, cannot be used to support the implementation of agile development methods. The key is to define and implement an Agile SCM

process that focuses on supporting, rather than restricting, the agile team. It can be a tricky exercise to find the right governance model for such a team, but a general recommendation would be to start with a more open process, only implementing restriction where necessary. Feedback is also an essential part of agile development methods. One of the ways that SCM can help with this feedback mechanism is via an automated build and testing (and deployment) process. It is therefore recommended that you invest a significant amount of time looking at this aspect of your overall process.

References

- [AgileM01] Kent Beck et al. *Manifesto for Agile Software Development*. http://agilemanifesto.org/
- [Berczuk03] Stephen P. Berczuk and Brad Appleton, *Software Configuration Management Patterns. Effective Teamwork, Practical Integration*. Addison Wesley 2003. See also Brad Appleton et al., *Streamed Lines: Branching Patterns for Parallel Software Development*. PLoP Conference: 1998. www.cmcrossroads.com/bradapp/acme/branching/
- [Lee07] Kevin A. Lee. *The Buildmeister's Guide – Achieving Agile Software Delivery*. Lulu Press 2007.

About the author

Kevin A. Lee has over 15 years of experience in defining, architecting and implementing software technology solutions. Although a Technologist at heart his real passion is in applying technology to help solve real world business problems. Kevin's expertise and interests include Technical Sales, Solution Architecture, Java Development, Application Lifecycle Management and the application of Agile Development practices.

Kevin is the author of a number of publications including *The Buildmeister's Guide - Achieving Agile Software Delivery* and *The Java Developer's Guide to Accelerating and Automating the Build Process*. He has also published a number of articles and tutorials on the Internet and the creator of the build portal buildmeister.com. Kevin holds a B.S. in computer science from the University of East Anglia and can be contacted at kevin.lee@buildmeister.com.