CM Crossroads
the Configuration Management Community

Agile Build Promotion: Navigating the Ocean of Promotion Notions by Brad Appleton

In keeping with this month's theme of "Build and Deploy", it seems appropriate to revisit our earlier article on Agile Build Management [1] and reflect a bit further on the ramifications for software builds, packaging, release, and deployment. We will use this as springboard into the realm of build status accounting to discuss various build promotion models and how to choose an appropriate and effective implementation of a build promotion lifecycle.

Why are Builds so Important?
The Build is where "the rubber meets the road"; or, more accurately, where "the customer meets the code." The code has several customers:

- Other developers who must modify/debug, integrate, build and test the code
- SCM and QA/Test who must formally build and test the code
- Customers and end-users who must evaluate, operate, administer and upgrade the system

The build is also the first opportunity for holistic feedback of system: what was previously broken apart into individual changes and modules is now an integrated executable "whole". By the same token, the build is also the first post-development opportunity for systemic bottlenecks and "show-stoppers" to significantly slow down, or even halt the progress of the whole team.

We could go on, but we probably don't need to spend much time explaining to SCM-folk about how important the build is to the success of the project – most of us already know all too well (more times than we can count).

Agilists and the Three Builds (with apologies to Goldilocks)
In the story of "Goldilocks and the Three Bears", the character of Goldilocks "tests the functionality" of three different items, sampling each one three different ways until it is "just right" before going on to the next one (who would have guess that Goldilocks was a member of QA :-)). In our previous article on Agile Build Management [1], we talked about Agility and "The Three Builds", where each build represents a feedback-loop around an increasingly larger-grained audience:

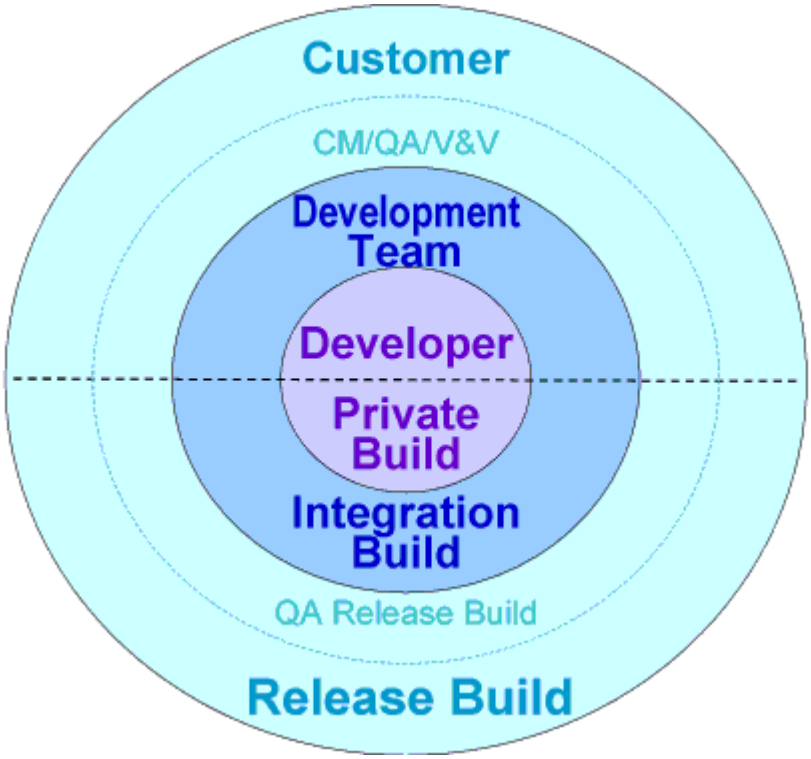
- **Private System Build**, for building a system to use for testing in your Private Workspace [2]. The consumer for this build is the developer doing ongoing work.
- **Integration Build**, to integrate everyone's changes in a central place [2]. The primary consumer for this build is the development team, both for the purpose of having additional assurance that the code integrates in a "clean" environment, as well as for integration level testing in advance of the release of the software.
- **Release Build**, which packages the software for release. The consumers of a release build are the testing team (for pre-release) testing, and the customers of the development team (for production use).

We noted that each build takes place at different levels of scale. One of the varying levels of scale is that of visibility (or organizational scope/impact). Another varying level of scale is that of frequency and duration (or project planning scope). At each level of scale, a slightly different set of needs and concerns must be resolved using a slightly different set of trade-offs:

Build *(What)*	**Visibility** *(Who)*	**Frequency** *(When)*	**Purpose** *(Why)*
Private System Build	Individual Developer	One or more times per change task	Provide feedback to the developer of the changes just implemented in the private workspace.
Integration Build	Whole Development Team	After each change task and/or one or more times per day (e.g. Nightly)	Integrates the latest changes in the repository and runs automated tests against the release providing feedback to the development team.
Release Build	Independent Test (QA or V&V) Team and Customers	At the end of each iteration/release	Packages release for distribution to the development team's customer.

An Extra Level of Organizational Indirection

Discerning readers may have noticed that we have grouped together what some would consider two different kinds of Release Builds [3] under the same heading: QA Release Build and Customer Release Build. They were "lumped together" because they both represent a scope of stakeholder visibility outside that of the development team, and any build that will be visible outside of the development team needs to satisfy some more formal criteria and additional needs beyond that of development.



At the same time, the difference between QA Build and Release Build is an important one that

underscores some of the critical organizational issues that are not addressed by some of the more popular "Agile" methodologies like Scrum and Extreme Programming. The emphasis on smaller teams, shorter feedback cycles and close customer collaboration can make us forget that it is often not feasible to remove all the additional layers of stakeholders between the team and the end customer.

In an ideal world for example, the customer would receive every build that made it past QA, or at least the result of every iteration. In reality, the customer won't always agree to tolerate such frequent delivery and may agree only to take the end-release, or the result of every third or fourth iteration. So there is often an entire layer of organizational infrastructure and stakeholder communication sitting in between the team and the end-customer. Even if this extra layer is not ideal, it is often reality.

Build Status Accounting
With all these different kinds of builds, and knowing how important software builds are, it stands to reason that knowing the status of code that is ready-to-build or already-built is similarly significant. Accounting for the status of "built" or "to be built" code is a subset of Configuration Status Accounting (one of the four key disciplines of CM as defined by IEEE and Mil standards). Build Status Accounting is often accomplished using a technique called Promotion Lifecycle Modeling.

Promotion Lifecycle Modeling (or "Promotion Modeling") involves defining the "important" milestones when CM artifacts transition to a new level of control and/or ownership. A Promotion Lifecycle is a specific instance of a promotion model that identifies the lifecycle of Promotion Levels for a particular project and its development lifecycle. For some reason (perhaps historical), when talking about the status of CM activities like projects, requests, and change-tasks we tend to talk about "states"; but when talking about the status CM artifacts and work-products, the term "promotion level" is often used instead of the more familiar "state".

When defining and using promotion levels (or, for that matter, lifecycles and state transitions) it is always important to ask the questions "of what?" and "for what?" This forces us to clarify both the content and the context of a promotion lifecycle and select and appropriate implementation:

- **Content** -- What is being promoted? Is it a set of files or file-versions (e.g., a change-set)? Is it a build? Is it a branch (codeline)? Is it a label (or "tag")?

- **Context** -- For what is it being promoted? (What "thing" is it a lifecycle "of"?) Is it for a particular project or release? Is it for a particular market or platform variant within a release? Is it for development at a particular site for a given release and/or variant?

In an agile context, the important milestones in a build promotion lifecycle are likely to correspond very closely to each of the different kinds of builds and their audiences as described in the previous section.

Build Promotion "Patterns"
Once we have discovered and defined an appropriate promotion lifecycle for our project, the next question will be how to implement that using our SCM or version control tool. There are a multitude of different implementations that recur in practice, and different tools seem to support (or even prefer) some implementation approaches more strongly than others. The items undergoing the "promotion" range from versions to workspaces, branches/codelines, and labels/tags. The most commonly recurring "patterns" are as follows:

- **Version Promotion** (Promoted Versions)

- **Promotion Workspaces**
- **Branch Promotion** (Promoted Branch)
- **Promotion Branches** (Promotion Branching)
- **Label Promotion** (Promoted Label)
- **Promotion Labels** (Promotion Labeling)

Version Promotion (Promoted Versions)
This may be the best known of the promotion implementation approaches. It was popularized by some of the earlier version control tools like CCC and PVCS. Individual file versions are assigned attributes corresponding to their promotion level.

For example, if our promotion levels were named {Private, Team, QA, Customer, and Failed} then each file version would be associated with at most one of these promotion levels. When a set of versions progressed thru a new promotion milestone, those versions would be "promoted" to the next level by changing their promotion-level attribute to the next value in the progression (e.g., from "Private" to "Team", or from "Team to "QA").

In theory, every time a file is checked-out and then back in, it starts over again at the initial promotion level. So you can always tell which files on the "tip" of the codeline are "fresh" and which ones are more "mature." You can also easily discern if all the files on the tip of the codeline are all at the correct promotion level prior to releasing.

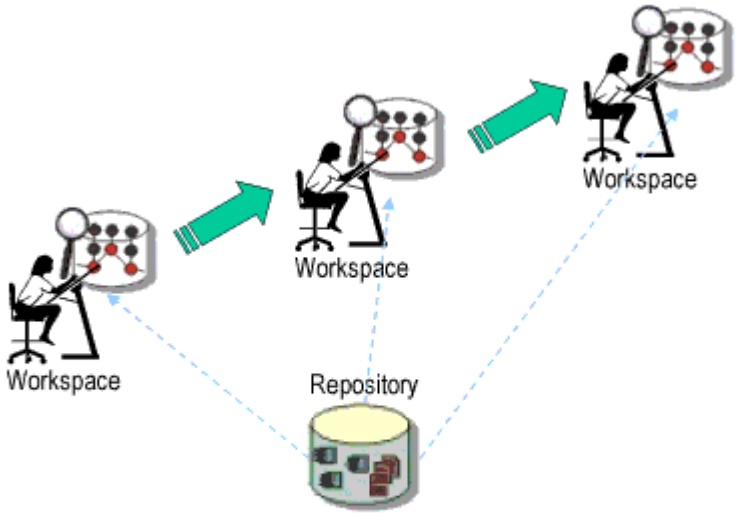
This seems fairly straightforward, and some version control tools assume this particular approach and make it available "out of the box." If you have to implement it yourself though, it can be cumbersome to apply/update the attribute for all affected versions, or to even identify all the affected versions. Some would also argue it can be confusing to keep track of which versions are the "current" ones to use.

Promotion Workspaces
This is quite possibly the oldest of the promotion implementation approaches, dating back to when source-code version control and related tools were in the "stone age" and a "version control tool" was only slightly more than a backup/restore mechanism. One of the earliest published examples is from the IBM OS 360 project [4] where "source-code libraries" were stored in so-called "vaults": one for development, another for integration/test, and a third for release/distribution.

In more modern times, a workspace (or sandbox) would be used to represent several of the promotion levels, especially those that corresponded to a distinct level of integration and test. Code would be "promoted" by merging it and checking it in to the next "promotion workspace" in the progression.

This approach is also straightforward, and doesn't have the drawback of having to apply a named "promotion level attribute" to every version. It does suffer the overhead of having to copy/merge versions from workspace to workspace, often even if not all the versions changed. Furthermore, workspaces aren't necessarily "first class" entities in a version-control system, so it may be difficult to know what promotion-level a version corresponds to without knowing which promotion workspace it

came from, and whether or not the other promotion workspaces have the same or different version of that file.

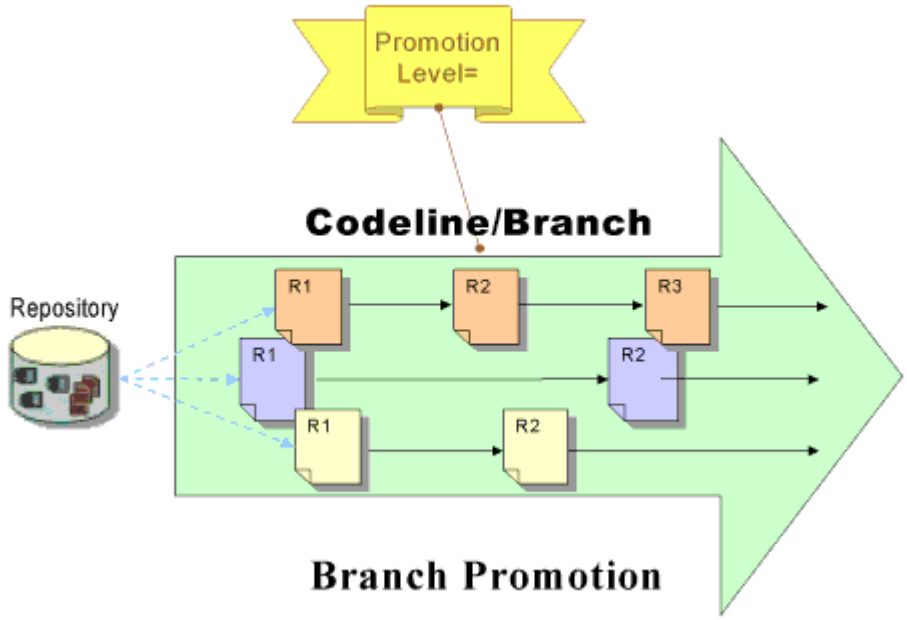


Promotion Workspaces

Promotion Workspaces are often well suited for promotion levels that correspond to distinct levels of integration and/or ownership, particularly when combined together with some of the other approaches. (For example, it can sometimes make sense to have a separate branch and workspace for a particular promotion level.)

Branch Promotion (Promoted Branch)

Branch Promotion is when a set of promotion levels applies to a particular branch or codeline. When the branch/codeline is "promoted", it means all versions of all the files on the tip of the codeline have progressed to the next integration level.

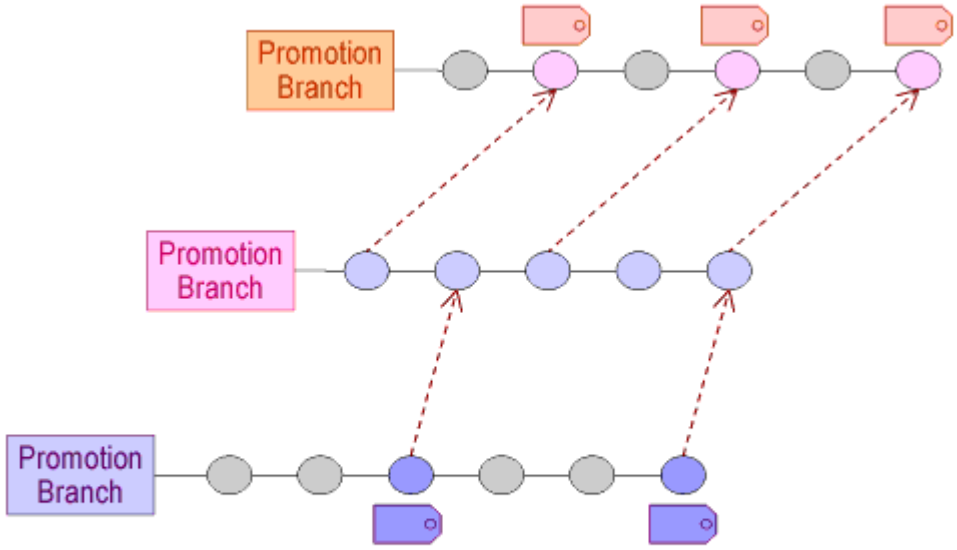


Branch Promotion

This can be implemented very efficiently, often by associating an attribute or meta-data with the codeline name. It has somewhat limited use however: it cannot be used if only a subset of files on the codeline need to be promoted. The branch is treated as the unit of encapsulation for the set of files and versions that are promoted together (the same behavior that makes it efficient also limits its applicability).

Branch Promotion is appropriate to use for promotion-transitions that are “all or nothing” for the entire codeline, and when the codeline is primarily referenced by its “tip” (and not some previous state of the codeline). It can be weak at providing traceability to (and reproducibility of) earlier states of the codeline corresponding to previous promotion levels.

Promotion Branches (Promotion Branching)
This is becoming an increasingly more popular approach as version control tools provide better support for quick and easy branching in a project-oriented fashion (rather than per-file). Each promotion branch corresponds to a promotion level in the promotion lifecycle. When a set of versions is to be promoted from one level to the next, they are “merged” (often “copy-merged”) from the current promotion branch to the next promotion branch in the promotion lifecycle.



Promotion Branching

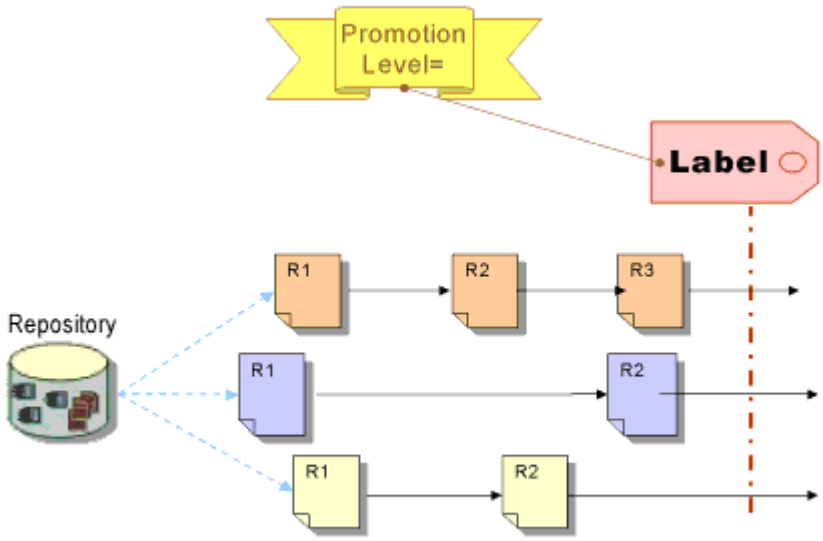
As with Branch Promotion, Promotion Branching uses each branch as a unit of encapsulation. This time however the promotion branch doesn’t represent a set that is promoted together, it instead represents a consistent set of versions all at the same promotion level.

The major drawback of this approach should be (hopefully) obvious. It can require (and encourage) additional branching and merging for work that doesn’t represent “true” parallel activity, and require additional time for copy-merging files whose contents didn’t change between promotion levels. Despite this drawback, a promotion branch can still be extremely appropriate when it corresponds to a transition that is another level of integration (and hence merging would be performed anyway).

Promotion Branching can also be appropriate for a promotion level that corresponds to a change in ownership between two groups that would otherwise require incompatibly different Codeline Policies

(as recommended in the best-practice "Branch on Incompatible Policy" [5]). In some circumstances, even when parallel development isn't required, if two different groups of people want to use the same codeline but have very different needs regarding codeline volatility/stability, integration frequency, and access control, it can be more productive to give them each their own codeline.

Label Promotion (Promoted Label)
Label Promotion associates a promotion level with a label or tag (and thus, with all the file versions referenced by the labeled configuration). Since a label can span versions of different files across multiple branches, so can label promotion levels.



Label Promotion

As with branch promotion, label promotion transitions an entire configuration of files to a new promotion level in one fell swoop. Label promotion is appropriate when an entire build passes from one promotion level to the next without modification (hence, no modifications are necessary and no new label is needed). As such, label promotion is particularly well suited for a promotion level that progresses from a built/integrated or test/QA promotion level to another test/QA promotion level. If a label would have to be created just for the sake of having a promotion-level (and otherwise wouldn't be needed), then label promotion can be very inefficient in such cases.

Label promotion can also work well together in combination with promotion branches: Promotion branches can be used for promotion transitions that correspond to a new level of integration, and label promotion can be used to avoid what would otherwise be unnecessary copy-merging from branch to branch. Thus, for a given labeled configuration that spans multiple branches – the version of a file in the configuration indicates the corresponding level of integration for that version and the promotion level of the label indicates the corresponding level of testing. (Of course combining promotion strategies together adds to the complexity of the overall promotion model implementation.)

Promotion Labels (Promotion Labeling)
Promotion Labels are almost identical to Version Promotion. In fact Promotion Labels can be regarded as a form of version promotion where a label is used to make the association between a version and its corresponding promotion level.

Promotion labels share the same drawbacks as Version Promotion. Labeling the configuration of files

and versions to be promoted can be very time-consuming for a large project (even when fully automated). And because it is common to name the promotion label only after the promotion level (and not include context information in the promotion label name), it can't be easily used when multiple projects/codelines are in fact which may each have different versions of the same file at the same promotion level at the same time for different projects. However, some like the fact that it lets them see all the labels for each promotion level on each version so they can easily audit to see if a level was "skipped".

Make it Lean and Agile

So how do we take a "big picture" approach regarding how to keep our promotion schema "agile" and in accordance with the principles of lean thinking and lean programming/development [6][7][8]? We start by keeping it simple (KISS) and eliminating any unnecessary and redundant promotion levels and promotion mechanisms:

- Don't use promotion branches where it doesn't already make sense to branch
- Don't use label promotion where it doesn't already make sense to create a label
- Try to avoid unnecessary copy-merging
- Let the promotion structure and mechanism "emerge" from necessary usage; don't try to force-fit a particular model or mechanism

Those guidelines may seem obvious, but can often be easy to forget when concentrating on a particular promotion lifecycle and/or its implementation. If we look at some of the patterns from the SCM patterns book [2], we can actually see that the skeleton of a basic promotion model is present, without having attempted to explicitly define or impose a particular promotion lifecycle or implementation:

- Task Branch, Active Development Line, and Release Line can each correspond to a kind of promotion branch (as can Release-Prep Codeline, which might correspond to a QA Release build while the Release Line corresponds to a more formal Customer Release Build). And each such codeline may warrant it's own integration workspace.
- An Active Development Line may be distinctly separate from a Release Line when the two codelines have different sets of users with incompatible policies
- Integration Test and Integration Build can (when necessary) correspond to two different promotion levels for the same label and/or branch

In general, applicability of the various promotion mechanisms is as follows:

- Either a task-branch or a tool's built-in "task-based development" mechanism is suited for promotion of a change-set.
- Branch Promotion is appropriate to use for promotion-transitions that are "all or nothing" for the entire codeline, and when the codeline is primarily referenced by its "tip" (and not some previous state of the codeline)

- Promotion Branching is appropriate for a transition that is another level of integration (and hence merging would be performed anyway).
- Promotion Branching can also be appropriate for a promotion level that corresponds to a change in ownership between two groups that would otherwise require incompatibly different Codeline Policies.
- Label Promotion is particularly well suited for a promotion level that progresses from a built/integrated or test/QA promotion level to another test/QA promotion level.

References

[1] *Build Management for an Agile Team*; by Steve Konieczka, et.al.; CM Crossroads Journal, October 2003 (Vol. 2, No. 10)

[2] **Software Configuration Management Patterns: Effective Teamwork, Practical Integration**; by Stephen P. Berczuk and Brad Appleton; Addison-Wesley, November 2002

[3] *Beyond Continuous Integration. A Holistic Approach to Build Management (Part 1 of 2)*; by Maciej Zawadski; CM Crossroads Journal, October 2003 (Vol. 2, No. 10)

[4] **The Mythical Man Month: Essays on Software Engineering** (20th Anniversary Edition); by Frederick P. Brooks; Addison-Wesley, 1995.

[5] *High-level Best Practices in Software Configuration Management*; by Laura Wingerd and Christopher Seiwald; presented at the **Eighth International Workshop on Software Configuration Management**, Brussels, July 1998; and at the **1998 Perforce Annual User's Conference** (P4UC'98 – see http://www.perforce.com/perforce/bestpractices.html)

[6] *Principles of Lean Thinking*; by Mary Poppendieck; **2002 Conference on Object-Oriented Programming Systems, Languages, and Appplications** (OOPSLA 2002); Seattle, WA, November 2002; (see http://www.poppendieck.com/papers/LeanThinking.pdf)

[7] *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)

[8] **Lean Software Development: An Agile Toolkit**; by Mary and Tom Poppendieck; Addison-Wesley 2003 (see http://www.poppendieck.com)

[9] *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)

[10] *SCM Patterns: Building on 'Task-Level Commit'*; by Austin Hastings; CM Crossroads Journal,

June 2004 (Vol 3. No. 6)

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