Introduction:
Forget One-Size-Fits-All SCM
Design Scaleable Best Practices

Individual Practices:
Plan SCM ... 2
Reproducibility ... 2
Change ... 3
Workspaces ... 4
Baselines ... 5
Metrics ... 5
Reuse ... 6
Integrate ... 6
Distributed ... 7

Figures:
Practices Map ... 3
Branches ... 5

Conclusion:
Final Words ... 7
Bibliography ... 8
Author ... 8
Cover ... 8

Software Configuration Management Best Practices

By Uttam Narsu

Forget One-Size-Fits-All SCM

Two-thirds of all software projects fail, according to the 2003 Standish Group’s CHAOS study. Improper usage of software configuration management (SCM) is largely to blame. After project management, IT users cite configuration management as the process that most needs improvement, according to Ann Haas in Configuration Management Principles and Practice.

Process improvement eludes many organizations due to the increasing complexity of their development environment which includes: multiple software development methodologies; the adoption of advanced parallel development; requirements from audit and regulatory compliance regimes; the impact of distributed development and outsourcing; and the increased adoption of agile and iterative development processes.

Most shops try to simplify by pursuing a one-size-fits-all approach to SCM, an approach that usually fails. They’ll implement simple CVS version control for complex projects, or use complex NASA procedures for simple Web projects. The best alternative—right-sizing SCM to the project—requires both planning and a thorough understanding of which SCM disciplines to apply to the range of projects undertaken.

Design Scaleable Best Practices

Best practices must also be right-sized. For example, collecting thorough project metrics is overkill for one-off projects. Best practices must also be adaptable to different development methodologies. Most companies no longer use one methodology for all projects. Indeed, as Tiwana and Keil note, a “one-methodology-fits-all” approach poses a high risk of project failure, as it usually leads to poor methodology fit. Today, many companies approve a range of methodologies that best map to the types of projects they undertake.

One client, an ISV, chose an agile methodology for new projects that required close collaboration with beta customers while adopting a waterfall-based approach for older products that had entered an end-of-life cycle. The agile team rejected the rigorous SCM practices of the waterfall group and struggled with formal release management to their customers.

This problem intensifies at larger IT organizations or consulting service providers that approve a greater range of methodologies. They typically allow methodology to be chosen based on the type and scope of the project and its risk level.

Ideally, all of an organization’s methodologies should leverage a library of scaleable best practices for SCM, requirements management, project management, and quality assurance.

In the pages that follow, I’ve highlighted several best practices that are suitable for placement in such a library. See Figure 1 for the relationship of each best practice to the key SCM processes. Each is briefly described and its applicability is noted in the context of scope/risk.
Plan Your SCM Environment Carefully

Few companies plan each project’s SCM environment effectively. More typically, a project manager evaluates the project’s needs, modifies the global SCM project template, and then starts the project. Developer unhappiness or project delays may ensue as a result of this mismatch between SCM policy and a project’s scope and risk.

One financial services client suffered these pains when it ignored the likelihood of substantial new code and new developers being injected into a project as the result of an acquisition. Proper design of development branches, workflow, and integration points could have accommodated this change without great upheaval.

A better approach is to develop several SCM templates, depending on the type of project (Web, embedded, legacy, etc.), the size of the project (large, medium, small), and the risk of the project (high, medium, low). Larger, high-risk projects should have more numerous developer workspaces, a branch hierarchy with more integration areas, a more formalized release process, and a larger number of measurement points that allow a project’s state to be monitored.

Flexible SCM tools can simplify this process, as simpler hierarchies can be created initially and then scaled up as the need arises. For example, if the user interface team grows and needs to be partitioned, a new integration area between the resulting groups could be added without having been created earlier by the SCM plan.

For larger projects, use the workspace model (discussed below), which scales nicely to enable integration areas, multiple releases, and the greater team isolation required by component-based development. Since project scope may increase, formalize build and release management as a separate discipline, one that projects of all sizes can benefit from, with less rigor applied to small or low-risk projects. Small projects may allow relaxed guidelines for submissions to a particular build, while large projects may demand that unit and smoke tests be completed first.

Determine whether the project needs to trace release components, perform automated builds, support partial deployment (including patches), supply bills-of-materials to packagers, recreate previous releases, roll back a failed build or release candidate, or deploy to a network of worldwide servers. Then adjust your SCM processes and templates to fit. Too often I see companies assuming that a combination of development and operations personnel can control build and release management and the problems that ensue.

Ensure Absolute Reproducibility for All Artifacts

Compare the cost of breaking a nightly build to the costs associated with failed patch deliveries to all customers. The first may result in a minor loss; the second, a much larger one. In the aggregate, these problems can greatly increase the cost of development. Absolute reproducibility resolves both of these types of scenarios.

Being able to accurately reproduce any build, any release, or any configuration is the hallmark of effective SCM. Yet many organizations struggle to achieve reproducibility. One former client, an ISV, had fifteen different releases in the hands of customers: four formal releases, with three internationalized versions of each, plus heavily customized releases for three very large customers. One of the customers, an aerospace firm, had a critical failure that required a high-priority patch to be constructed.

Unfortunately for the ISV, they couldn’t reproduce the exact configuration installed at that customer’s site. Creating the patch took three agonizing weeks, with multiple deliveries to an increasingly irate major client. The lesson was hard won, and now the client employs absolute reproducibility to a fault: they version control operating system include files, libraries, and even the platform’s compiler!

Three factors are critical to implementing this best practice: choosing the right branching/parallel development model; using an SCM system that supports change sets/packages; and ensuring that the SCM system can handle transient failures (power, network, and human error) gracefully. Absolute reproducibility should be used on all projects except for the lowest-risk, one-off projects, where the costs may outweigh the benefits.
Require Change Requests and Change Packages

One client at a large embedded software shop still relied on coding heroes, the modern-day equivalent of a Bronze Age Odysseus. One pony-tailed hero happened to be the release manager, who could handle with casual aplomb complex sets of changes submitted to the weekly builds.

Life was good until Jeff took a well-earned sabbatical and trekked off to Australia for 17 weeks. Development fell apart. Bugs mysteriously resurfaced after being fixed in previous builds. One release from a remote office needed to be backed out due to the absence of three crucial files, lost during a network failure during the submission. Lost work led to wasted rework hours. The basic problem was that file-based versioning was inadequate for identifying change. The company needed a way to group all the changes together as one related package.

Many high-end SCM tools provide the ability to group into change packages the changes produced during software development. The best of these tools also provide atomic transactions, so that the change package is applied with database-style all-or-nothing semantics. Atomic transactions would have prevented the remote office’s submission from being applied without all of the files present.

When used with change requests, change packages make it very easy to track which artifacts were changed to fix a particular bug; which code needs to be reviewed, tested, or regressed; and, most important, why the code changed. The most stringent form of this best practice requires that any change must be accompanied by an approved change request, and must be bundled into one or more change packages.

Stringency is best achieved by SCM tools that feature secure append-only databases, thus ensuring (continued)
Require Change Requests and Change Packages (cont’d)

that no user can tweak the system. Such guarantees are essential for the secure audit trails required by such compliance regimes as Sarbanes-Oxley and Basel II, as well as for ITIL certification.

Change packages ease release management, since a release is composed of well-defined change packages instead of hundreds of scattered files. Finally, proper use of change requests and change packages permits easy rollback of problematic updates and allows for almost transparent collection of project status metrics, ensuring that project managers can quickly spot when a project is heading off track.

While change packages may seem like unnecessary overhead to small teams using agile methodologies, continuous integration (see below) is difficult without them. Even small projects should strongly consider employing this practice, as the best SCM tools provide user interfaces that make employing change requests and change packages less intrusive than was the case with tools of years past.

Maintain Private Developer Workspaces

One client at an embedded systems company was ready to kill a colleague who always asked for help locating the edits to his files. They had been developing a product using their own brand of embedded eXtreme Programming (XP), and he was persuaded to use simple CVS version control because of a lead contractor’s prejudice that a commercial product’s workflow would slow them down. Without branches, the developers manually made backup copies of their source files as they worked on them, because they could only check the files in when they were done with all their changes. Sometimes a developer would forget to make a copy, and work was lost.

At other times, they had trouble locating the right backups. And once the correct backups were found, they still had a difficult time differencing them or merging them because the files didn’t have the proper ancestor relationships—they were just two text files to their merge tool. The mistake made here was choosing an inappropriate branch strategy for their development.

I recommended that they adopt the workspace (sandbox) model, which flexibly scales across projects and methodologies small and large. In the workspace model, each developer works with his or her own set of files, which are promoted to a more common area only when tested and approved by the developer. Workspaces allowed their developers to safely checkpoint any number of intermediate versions into the SCM repository. Since a file is shared with other team members only when it is ready, their developers checkpointed the files in their workspace as often as possible, and work wasn’t lost due to SCM issues again.

Workspaces are best supported by SCM tools that can understand the relationships between different “streams”—a first-class abstraction branches, and workspaces. Tools that understand streams allow the streams to be rearranged, reparented, or even to have a new integration area added between them. Workspaces are essential for large projects, but smaller projects may also benefit from them. If the project is likely to grow in size over time, forcing developers to adopt workspaces midstream may elicit resistance, resulting in retraining needs and project slowdown. In that case, adopting workspaces before being forced to do so may be prudent.

Even for Web development work, where parallel development is rare, workspaces can be helpful because of the sometimes-messy dependencies between templates, markup, page controls, and components found even in strict Model-View-Controller approaches. These dependencies force developers to serialize their changes to the Web project files.

“Where’s the copy of foo.c with my edits?”
Create and Work from Appropriate Baselines

When code is flying fast and furious, milestones can be missed as developers step all over each other. When the latest versions are always changing, it can be tough to even build the code in your workspace reliably. As you promote or merge the latest versions into your workspace, you might not be able to easily determine whether build or application failures are caused by your changes or changes from the latest version. The problem can be exacerbated if your team is using parallel iterations, which are sometimes used when distributed developers are working on the same code or when UI prototypes are being mocked up separately.

Agile methodologies use iterations to structure their milestones, but often they fail to use an SCM construct to serve as the starting point for the next phase of development. This construct, the baseline, consists of an identified set of versioned configuration items. See Figure 2 for the relationship between streams, branches, and workspaces.

I recommend creating baselines to mark significant milestones, or during each full iteration, or to structure parallel iterations during the development cycle. Doing so establishes a starting point for subsequent changes. These baselines must be fully tested and, if possible, the system under development should be in a usable state. Baselines are appropriate either when using a waterfall process or an iterative process. Baselines are less appropriate to small projects and many Web-based projects; however, the ability to roll back to a previous baseline can be a lifesaver on even the smallest project.

Leverage Metrics for Process Improvement

What project manager hasn’t lamented that the same mistakes are made time and again. The Capability Maturity Model hammered one lesson home to software developers: process improvement matters, if you want to stop making those same mistakes. Yet without a concrete measure of where you stand with your processes, it’s difficult to know where to focus your efforts. SCM systems can help provide metrics. One of the most potent metrics, “percentage of change requests delivered on time and within budget,” is a trivial calculation in a solid SCM implementation but costly and labor-intensive to produce without SCM.

It’s best to select a mix of technical, process, and business metrics, but be wary of how you introduce metrics. One company I worked for kept metrics on everything, including, rumor had it, keystroke frequency. Developers grew resentful, and lunchroom talk centered on creating hacks to “fool the keyboard logging metrics.” Needless to say, the metrics effort failed, and process improvement was slowed. A better approach may have been to support private, personal metrics, such as the kind advocated by Watts Humphrey with his personal and team software processes. Once developers are sure that metrics are not used for “evil” purposes but for individual, team, and company process improvement, they’re likely to be more supportive.

Additional key metrics to track are

- Defect density, cost, and origin
- Mean time to repair, mean time to failure
- Cost per unit of functionality
- Actual versus estimate
- Time to market
- Functional quality

As you track the metrics over time, focus less on blame games and more on diagnosing and determining how to prevent failures in the future. Metrics are beneficial for all project sizes, but pay off the best for larger projects.
Create Reusable Components

Reuse, or not reinventing the wheel, has long been the holy grail of the software industry. While it remains elusive compared to other industrial sectors such as electronics, some success in the form of software components has emerged. Component-based development pervades all domains of software these days, from embedded systems development to Web development, from gaming to IT. Moreover, components play a central part in implementing the services in a service-oriented architecture on platforms such as J2EE and Microsoft.NET.

Components and services must be versioned so that they can be replaced with newer versions as they are released or can be substituted with alternatives from different vendors. Requirements, issues, and defects must be traceable both to a component and within the component. It must be possible to create baselines and releases from specific versions of components, even when an application is partially deployed and updateable over a network.

Another challenge is how to handle larger-grained reuse in the context of open source software. One former client, an ISV, is developing an application based partly on proprietary code and partly on open source components from Hibernate, the Spring framework, and Tapestry.

The team members created their own baselines and placeholder components to keep themselves insulated from the rapidly changing open source code. They also annotated the metadata of the source code stored in their SCM system to point to their own Web site discussion threads, test plans, and requirements.

Their SCM system thus enabled them to practice reuse more effectively. They used impact analysis and dedicated refactoring tools, which are essential add-ons to any SCM environment. They could then construct dependency trees for all components and show the effect of any change made to the interface of a component. They also ensured that they could track every artifact, including build flags, deployment descriptors, published SOAP interfaces, and so on.

Best practice requires that you test components early and often. Make sure your developers understand that integration testing is the last defense for a component-based system. Develop a process for upgrading components and services on your timetable, not the suppliers'.

Structuring assets into reusable components is more expensive to do for smaller projects; however, with the increased use of open source components, it’s really a discipline that benefits all projects.

Merge and Integrate as Often as Possible

Taking four days to merge changes from workspace code isn’t fun. Kent Beck, the father of eXtreme Programming, developed the concept of continuous integration to address this very issue. With continuous integration, code is integrated several times a day rather than just every night or once a week. Code doesn’t have a chance to get stale, and progress is more rapid without big, stop-the-world merge-and-integrate fests.

While few projects in the IT world can achieve the level of continuous integration that eXtreme Programming demands, the trend is towards shorter and shorter integration cycles. Guidelines are followed so that developers don’t integrate whenever they feel like it; the code after integration must be in a usable state. Most shops pursuing iterative development are now doing integration builds several times a week, with many pursuing and achieving the goal of nightly builds.

Choosing the right SCM approach to foster merging and integration can be challenging. I’ve seen some teams giving in to merge-fear and forgoing parallel development. They adopt a shared mainline approach, with pessimistic locking (in other words, whoever grabs the file first locks it), and team members must communicate outside the SCM system to resolve who gets access to a file. As discussed earlier, the workspace model is a big improvement, especially since it scales quite easily to handle geographically distributed teams. Continuous merging and integrating is appropriate to projects of all sizes.
Structure for Distributed Development

With onsite developers, telecommuters, consultants, and outsourced team members all part of today’s development team, it seems like development gets more geographically distributed every day. SCM systems generally support only a subset of the range of development models, from single repository to multiple repositories, with various forms of synchronization or federation possible.

Management of distributed SCM is challenging, especially over WAN/LAN boundaries, so often a non-local team will ask for their own local repository or, worse, a different SCM system that they are more familiar with.

I’ve observed this problem firsthand with one project that used very poor partitioning. Developers found it difficult to identify conflicts in code that had been changed at both locations. The SCM tool that they had chosen forced them to utilize multiple repositories because it was fairly unusable from the slow uplink at the remote site.

Rather than reinvest in a new product, they the company first tried to limit their parallel development, which made it difficult for the remote developers to fix bugs discovered during unit testing—they had to serialize the changes through the North American development center, which meant costly delay.

For reliability and manageability, it’s often best to go with a single server designed for TCP/IP networks, including caching support, batch updates, and atomic transactions. Tests should be run with large updates over the longest latency subnet, and with transient failures simulated. Simulate multiple streams, multiple workspaces, and several integration areas to determine whether any rules of thumb (for example, checkpoint often) need to be modified.

If the SCM system does not scale effectively, or if network failures are frequent, it may be best to use a federated or multisite solution. Be careful: project partitioning without the aid of impact analysis tools can be difficult and costly.

Rather than having to change tools midstream, often it’s wise to invest in an SCM system that is ready for distributed development.
Bibliography


About the Author

**Uttam Narsu** has more than 20 years of experience as an analyst, architect, software developer, trainer, and consultant. Uttam is quoted frequently in industry publications such as *Application Development Trends*, CNET (News.com), *Computerworld*, *eWEEK*, *InformationWeek*, and *Internet Week*, as well as in *Business Week* and the *Financial Times*. Uttam has also spoken at numerous industry conferences.

Formerly a well-known SCM analyst with Forrester Research and Giga Information Group, Uttam now freelances, consulting on a broad range of enterprise application development areas including SCM, agile development methodologies, testing, XML, and Web Services.

Uttam has worked at various companies, including several startups, research firms, and a stint in IT. Uttam attended the University of Chicago and earned a B.S. in computer science from the State University of New York, Albany.

About the Cover

The cover image features a boat wake at Providence’s Waterfire, an art installation and almost weekly event. See [http://www.waterfire.org/about/index.html](http://www.waterfire.org/about/index.html) for details on Waterfire.

Image copyright © 2003 Andrew Shearer. Andy graciously provided permission to reproduce the image; see Andy’s Web site at [http://www.shearersoftware.com](http://www.shearersoftware.com) for more information.