Software Configuration Management in Small Organizations: An Overview

David M. Ihnat
dihnat@dminet.com

November, 2002

1. INTRODUCTION

Many small companies find themselves host to a rapidly evolving software environment on heterogeneous development and execution platforms. Current practices within a small development group often serve well enough to bring the software base to production in relatively short order. At some time, however, it is usually realized that a more rigorous Software Configuration Management (**SCM**) scheme is necessary.

This memorandum is intended to fill several roles. It will:

- Serve as an introductory discussion of SCM issues, particularly as they apply to small development organizations.
- Provide an overview of a desirable SCM environment, particularly with respect to the benefits and characteristics to be derived.
- Describe the steps by which the environment may be migrated from its current tool set, configuration, and conventions to the desired solution.

2. SCM SOURCES

Particularly as it's intended to be a fast, clean exposition, this is primarily an expression of the author's experience in implementing SCM solutions. However, it is grounded in and backed by numerous current writings and research in the overall area of SCM, Version Management, and software development methodology. Please see *Appendix I* for pointers to reference material for background and detail information on SCM.

3. OVERVIEW

A great deal has been written about SCM, including a wide range of definitions and "fundamental concepts". At the base of it all, however, is a simple concept: *No surprises*. When you actually deliver the product, it should have all the features intended, none you didn't intend—whether actual features or bugs—and you should know exactly what it *is*.

Always keeping that in mind, there *are* a number of features that are necessary to accomplish that main goal. In the following, when discussing a *target*, what is meant is any product of a software development cycle—a binary program executable, scripts, configuration and installation information and/or tools, design and end-user documents are all targets. That given, an SCM system should provide:

- *Specificity*. You must be able to specify exactly what comprises a target.

- *Repeatability*. You must be able to reliably and repeatably produce a target—be it an executable, documentation, or a full package—at any time after the original has been generated.
- *Verifiability*. You have to be able to verify exactly what comprises a target, without having to carry out a research project to collect the information.
- *Flexibility*. Any SCM model and the tools that are used to implement it must be capable of readily evolving to meet changing business and development requirements.
- *Scalability*. In terms of the size of projects that can be managed, or the number; or the number of people participating, an SCM scheme must allow for growth.
- *Simplicity*. Any scheme to implement an SCM system has to be simple enough to use in day-to-day development and testing. It isn't an end in and of itself, and shouldn't make the task of producing and maintaining software harder than it already is.

To accomplish these goals, there are three fundamental tasks that must be completed.

First, clearly defined roles must be established for the execution of SCM-related activities. Individual responsibility and oversight is an important element in creating a consistent and reliable SCM model.

Secondly, the SCM system must be created—those aspects, both physical (e.g., disk storage, directory structures, support tools) and process-based (processes, practices, and procedures and the documentation that define and explain them) that, taken *in toto*, provide the infrastructure on which the SCM scheme operates.

Finally, the SCM maintenance and execution procedures must be clearly defined, implemented, and assigned. These are the day-to-day operational tasks that make use of the SCM system to accomplish the task of developing and maintaining software systems.

A number of tools—commercial, free, or developed in-house—are used by different organizations to help carry out SCM tasks. But the most important thing to keep in mind when determining what to use is that a tool won't make up for a bad or missing underlying process. You can have a good, if tedious and slow, SCM scheme using paper forms and carefully followed manual procedures. You can, and often do, have powerful, complex and expensive SCM tools—Version Control, Tracking, Build, etc.—and still have a virtually worthless and confusing SCM environment.

4. ROLES

In a mature organization with all SCM elements in place, at least the following roles should be defined[1]. The specific names may change, but the general separation of responsibilities and tasks are fairly standard.

- § *Development Team Lead*
- § *Development Build Manager*
- § *System Test Build Manager*

[1] There may, of course, be even more roles in a very large organization—but that's a level of complication that need not be investigated in this context.

- *SQE Test Manager*

All of these report to some manager assigned oversight for the entire process, although such oversight may be only administrative—commonly, the CIO, VP or Director of Software Development, etc. Each role carries a separate set of responsibilities for the application of the SCM methodology in the development process.

The primary purpose for the roles defined is to introduce the SCM equivalent of firewalls to the development process. Each role allows the development organization to cleanly and verifiably guarantee the origin and reason for modifications; to assure that items can be reliably recreated under known conditions; and to assure that what was tested, and only what was tested, is approved and distributed for production.

There is room for flexibility, of course. Just because there are four separate roles defined, there may not be four separate individuals filling these roles. Most organizations, especially small ones, can't afford to dedicate a full-time staff position to carrying out such a role. Thus, it's often the case that one person "wears several hats". This is perfectly acceptable, as long as the roles don't interfere—for instance, the software manager responsible for meeting delivery deadlines really can't objectively fill the role of the SQE Test Manager, who may have to reject a release candidate.

Also, not all roles may be expressed in a small organization. It's not uncommon for the Development Team Lead and Development Build Manager to be one and the same. Similarly, the System Test Build Manager may also be responsible for carrying out integration/system tests. Again, as long as separation of function is honored—e.g., the output of a build in the development team environment isn't accepted for system test because the same individual fills Development Build and System Test Build manager roles—this isn't normally a problem.

The following explanations of the roles and responsibilities should be read while keeping in mind that each may be modified to meet the requirements of the SCM model in use by the organization.

4.1 DEVELOPMENT TEAM LEAD

This individual is assigned the task of coordinating day-to-day software development for his/her team. There may, depending on the complexity of the development effort or size of the organization, be multiple Team Leads, who will coordinate with each other. In a very small organization, this role may be subsumed by the *Development Build Manager*, resulting in nobody carrying the "official" title of Team Lead. This role almost certainly carries connotations of responsibility for software development decisions—perhaps architecture design, or coordination of feature design—but those are beyond the scope of this document. Only those tasks related to SCM functions are of concern.

The Development Team Lead is responsible for auditing change management issues assigned to his/her team members, tracking the progress of those assignments, and verifying that software is checked into the version control database by developers with the proper attributes and annotations when development is done. He/She will verify that actions are taken to guarantee timely and proper promotion of modified software and documentation to the highest level under software developer control, such that it can be handed off to the Development Build Manager.

4.2 DEVELOPMENT BUILD MANAGER

This individual is assigned the task of carrying out unit, integration, and system builds for all development groups at the times, and in the order, specified by the development methodology. He/She is responsible for determining when source will be labeled for a Development Build, and

notifying the development team(s) of submission deadlines, and also when the source database is available for further modifications.

He/She verifies that all objects under version control that should make the level of Development-approved build as a candidate for test release actually are properly included and successfully build; and if they don't build, for demoting them and reassigning the work to the responsible developer.

Finally, when a candidate build is ready for test release, it's the Development Build Manager's responsibility to release it to the System Test Build Manager.

4.3 SYSTEM TEST BUILD MANAGER

Once notified by the Development Build Manager of a candidate build, the System Test Build Manager will create a labeled version of the build which may be submitted for a build in the appropriate controlled build environment. Once the integrity of the candidate has been determined per applicable organizational software development requirements, the System Test Build Manager will label the candidate—including derived objects (e.g., libraries and executables) resulting from the build—and pass it on to the SQE Test Manager.

4.4 SQE TEST MANAGER

Software Quality Engineering (SQE) is a title commonly applied in organizations to denote the group or individual(s) tasked with verification and testing of products before release to production.[1] This process is often called System Test, and that term will be used from this point on in this document.

Once a release candidate has been released by the System Test Build Manager, the SQE Test Manager[2] carries out whatever test cycle has been designated by the organizational development methodology. Detected flaws are passed back to the development team through the Change Management procedures for correction; depending on the criteria in effect, detected errors may be acceptable for release or may result in rejection of the release candidate.

5. COMMON CORE SCM ELEMENTS

No matter the approach or tool set used, there are common elements that are required to make the SCM scheme feasible:

- *Bill of Materials*. A means of keeping track of everything that is necessary to create a target—source, environment, tools, specifications, etc—is absolutely necessary to provide repeatability and verifiability of target creation. This is, at this point in the maturity of SCM tools, one of the most nebulous features. PVCS Build Manager, Makefiles, and ClearCase "configuration record" are examples of attempts to resolve this issue, but most often it comes down to actually maintaining a file that documents all aspects of the software creation/maintenance system, which is itself versioned.
- *Self-Identifying Configuration*. A means of identifying a set of software, documentation, etc. in such a way that the identification is an inherent part of the

[1] Of course, depending on the organization, this may only be one aspect of the job.

[2] And his/her team, if so lucky as to have one.

versioning scheme is critical. Version control (**VC**) systems[1], such as SCCS, PVCS Version Manager, or ClearCase provide individual and set management of object versions (e.g., files, documents). Promotion Groups or Levels, for instance, are a common means of extending the object versioning to sets of objects on the build side. Embedded version information—SCCS What strings, version blocks, etc.—in the targets are examples of methods used to carry this information forward into the end products.

- *Modification Request Tracking*. Unique identification of the reason for creation or modification of software or documentation allows planned determination of priority, scheduling, and assignment of tasks. Associating every individual change with such a request allows determination of both when the work has been completed, and that only that work that has been planned and agreed upon has been carried out. Finally, such feature-based tracking allows coherent, planned testing to verify the presence and behavior of new and/or changed capabilities. Such tools as PVCS Tracker or DevTrack are examples of systems that support greater or lesser integration with VC systems. This SCM element is often overlooked in ad-hoc or simple schemes, sometimes due to lack of integration with selected VC and/or build tools, other times due to a mistaken belief that such tracking will make development more complex. Neither need be true.

6. COMMON SUPPORT ELEMENTS

In its simplest form, an SCM scheme providing version management to coordinate changes and a self-identifying configuration capability to permit regular developer builds can provide most, but not all, of the features of a full SCM model. However, this can result in a number of undesirable effects; for example:

- *Wasted machine resources*. Developers may have to rebuild the entire project just to carry out unit integration on changes or additions they're working on. This can waste disk space—no matter how cheap today, there are limits to available storage, and this also stresses backup schemes—and CPU cycles.
- *Increased developer environment complexity and maintenance overhead*. In such an environment, each developer is responsible for building and maintaining a private copy of the system to the scope necessary to work on his/her individual assignments. This may be relatively simple, if the scope is constrained, or complex as development moves to system integration; but it's work, nevertheless, that isn't directly related to development of software features or fixes. Multiply that effort by the number of developers, and it results in a measurable productivity impact.
- *Integration effort increases*. With each developer working in a private workspace, system integration is delayed until such time as changes are submitted to the common pool.

A number of concepts have been used, either individually or in conjunction, to help reduce or eliminate these and similar issues. In some cases selected tools may not natively support the

[1] The term *version control* is usually used specifically to refer to the software that provides the version control functions and database management for archives. The term *version management* will often be used; it commonly refers to the set of processes and procedures that guide the use of a version control system, as well as the version control system itself.

concept, in which case the choices are to abandon implementation of that feature; change tools; or determine a workaround with the existing toolset. There are real benefits to each of these, however, that argue for inclusion in the SCM model if at all possible.

6.1 PROMOTION GROUPS

Version control fulfills two essential needs:

- Prevent corruption and/or loss of software and documentation.
- Permit selective re-creation of selected collections of software and documentation.

Note that this doesn't require a description of just how this is accomplished. That is fundamentally a function of the underlying system that implements the version control. There are two common models that are used to accomplish version control in current systems. (In the following discussion, please remember that an object is, usually, a file, although some systems support non-file version-controlled object, e.g., a virtual bill of material.)

6.1.1 Version Control Common Models

6.1.1.1 Virtual

There are two common methods of selecting and providing access to a collection or version of a multi-object project. More capable (also read expensive and complex) packages, such as Rational ClearCase, provide a virtual view into the version control archive database. Essentially, explain to the package that you want to work with "Release 1.2.3" of the system, and that's what it will tailor the environment to display.

There are a number of attractive features to this model. Probably the most attractive feature is that there is strong coupling between the data under development and that in the version control database, because it is one and the same. The only way to really work with the wrong version is either for the project version description to be wrong, or to have actually asked for the wrong version.

However, this is, as currently implemented, an intrusive and infrastructure-heavy model. ClearCase, for instance, requires a custom virtual filesystem on each supported platform. And all mature versions of such systems today are at the high end—usually the very high end—of the cost curve.

6.1.1.2 Reference Copy

Another model requires reference copy extraction. That is, whenever someone wants to work on "Release 1.2.3" of the system, the version control package uses that information to extract all the objects associated with that release, making sure they're at the proper level on a per-object basis. This has the advantage of a greater general degree of independence from the underlying operating platforms, at the expense of a much weaker coupling between the extracted reference copy and the archive database. With proper support from the version control system, conventional usage, and administrative and user tools, however, this model has proven adequate for even very large projects. Common examples of this model are Microsoft's Visual SourceSafe and Merant's PVCS Version Manager.

6.1.2 Promotion Groups

No matter the model, however, there is the common theme that a *project version* needs to be identifiable as a unique collection of individual objects, each of which may be at a different

internal revision level. Thus, Release 1.2.3 of a project may consist of source, object, library, or executable files, for instance, as well as copies of text or word processor documents describing both internal design and end-user manuals.

An approach that has proven portable across version control architectures is that of *Promotion Groups*. Usually this makes use of the capability of most version control software to assign a *label* to one or more objects at a given state or version.[1] It's a simple concept:

- Define levels of code reliability and maturity that make sense within the organization, identified by an unversioned label.
- Developers, testers, etc. who own responsibility for each level are authorized to move individual objects to that level when they've passed testing and/or QA criteria.
- Periodically, a unique, versioned label is drawn for all the objects named by the unversioned label. This becomes a unique release at that promotion group level.

A particularly attractive element of this model is that the amount of time the entire version control database is unavailable to developers or testers is limited to however long it takes to create a new label—usually on the order of a few minutes in the worst case. As soon as a label is created, development can continue.

To be successfully implemented, it requires four supported capabilities of the underlying version control software:

- It must be possible to assign an alias to a specific version of an object under version control. This is alias usually called a *label*.
- It must be possible to assign a label from an existing label.
- It must be possible to assign multiple different labels to the same version of an object.
- It must be possible to carry out label operations on selections of objects selected by a common label.

In most cases, if one or more of these features are missing, it's possible to construct a workaround, especially if there is a programmatic API permitting scripted or program-driven version control operations in the VC database. This is, of course, less desirable than if the feature is directly supported by the VC software, for a number of reasons. Just a few examples of the problems inherent in implementing this on systems that don't support it natively:

- It requires custom coding that must be developed or bought, then maintained.
- Since the capability isn't an integral component of the underlying VC database, verification and validation is also an external task, requiring software and additional steps in daily use.

[1] At least one version control system—SCCS—only had the concept of individual object versions. Eventually, layered on top of this was the concept of a project label set—which was totally unsupported by the base version control software. External tools—in most cases, shellscripts—extracted, built, and maintained individual file version numbers in a list that became the project version. The end result, with many evolutionary changes and grown features, is still available today as a product known as SABLIME. It is, however, mainly a legacy upgrade path. This example does show that the concept of a promotion group can be implemented even in systems that don't have native support—albeit with much work.

- There's no inherent connection between the promotion groups and the VC database—it's managed by external elements—"loose data coupling", again.

Thus, when selecting a VC package, this is one of the desirable features to look for.

An important concept in this model is that of *unversioned* versus *versioned* Promotion Groups. This allows "leapfrogging" to pass labels on to higher promotion group levels without locking the ongoing development and promotion process. This will be demonstrated in the following section.

Another advantage is the fact that there can be virtually any number of Promotion Group levels, depending on how simple or complex the development methodology may be in terms of defining the development/test/release cycle. In fact, a simple model can be used early on, and additional levels can be added when needed.

6.1.2.1 Example Promotion Group Model

As an example, let's assume that a development organization has a rather simple software development model: The only SCM roles defined are the Development Build Manager, and the System Test Build Manager. The developers work on code until they say it's ready for testing, whereupon the Development Build Manager promotes it. It's then built by the System Test Build Manager and tested for production release. When it passes that test it's considered production code and is labeled as such.

Given the development organization has decided on the following promotion groups:

- *Frontier*. This belongs solely to the development group. ALL that's promised at this point is that—well, it's in the database. Actually, this label is rarely created for the project as a whole; it's synonymous with the "tip" revision in most VC software packages.
- *DevBuild*. Standing for *Developer Build*, it again belongs solely to the development group. All that's promised is that everything will compile and, as appropriate, link-edit. If developers want to share code with other developers in the course of a development cycle, they promote files or documents to *DevBuild*. (This is also the source for developer-level *daily build*s carried out by the Development Build Manager.) Again, this isn't usually versioned; the developers consider it a rolling target of which a "snapshot" is taken for integration testing when they say it's stable. The Developer Build Manager carries out periodic builds, in addition to or instead of daily builds, to guarantee consistency.
- *SystemTest*. When the software at *DevBuild* is considered ready for testing by the developers, they (or more appropriately, the Developer Build Manager) notify the System Test Build Manager, who then carries out the promotion. All objects currently labeled *DevBuild* are also given the *SystemTest* label. This is the first level at which versioned labels are created. At this level, also, production targets—libraries, executables, etc.—are actually placed under version control, so that whatever has undergone system test is what actually gets released. When everything is in place, the testers create a versioned System Test label, freeing the unversioned label to start accepting new promotions.
- *Production*. After the software has passed all testing, a versioned *Production* label is created from the versioned *SystemTest* label. (This group has decided they don't need unversioned *Production* labels, since there isn't a separate group doing system test and production release.) The software development cycle is done for this release of the system. Note that there may be several versioned *SystemTest* labels before there is a single versioned *Production* label, if candidates fail testing.

For this model, the unversioned labels are actually as shown—*Frontier* (if used)*, DevBuild*, *SystemTest*, and *Production*. The versioned labels have been encoded to indicate the marketing version number of the product (indicated as **N.M**, where **N** is the major release number and **N** the point release, e.g., **4.2**), promotion group, date/time, and daily build number. An example of the second candidate of the day for a 4.2 developer build for System Test, then, would be:

4.2 ST 20010817-02

Let's assume this particular build candidate makes it through the entire development/test/release cycle—it's the last candidate and has fully passed testing. Its label migration would look like this:

Frontier	à	**DevBuild**	Remember—everything at *DevBuild* can be used by everyone in the development group.
DevBuild	à	**SystemTest**	This is the "snapshot" for testing. After creating this label, **DevBuild** can begin to change again.
SystemTest	à	**4.2 ST 20010817-02**	This is the first versioned label. **SystemTest** can begin to change again.
4.2 ST 20010817-02	à	**4.2 PR 20010817-02**	Release the products of this tested build.

6.2 DAILY BUILD

Simply put, the idea is to build the world daily and, if possible, carry out a minimum regression test—even if it's nothing more than making sure everything can execute (commonly called a "smoke test").

At the very least, this guarantees that the common development frontier is reasonably sane and prevents integration deviations from becoming large and encompassing a large amount of source. It also can foster a sense of progress for the development group, as new features show up in the reference build on a daily basis.

If a project gets too large for a full daily build, it's usually broken into a rolling set of subsystem builds, with the target of "rebuild the world" in as short a cycle as possible.

The Development Build Manager often incorporates a DevBuild-level daily build in the team's process to guarantee that the software remains sane even at that early level. This may be replaced by unscheduled manual builds if a daily build isn't deemed necessary.

This process, incidentally, was incorporated in the SCM model used to develop Microsoft Windows NT.

6.3 SHARED SOURCE CACHE

The shared source cache only applies to version control systems that use the *Reference Copy* version control model as described in section *6.1.1.2*.

When every developer has their own workspace, it's necessary to extract as many of the source files, headers, resource files, etc. as necessary from the version control database to be able to compile and/or link the software for developer unit or integration testing. This can eventually require extracting the entire project to be able to make sure software elements can be incorporated in the full system. This can cost not only a lot of disk space, but also time and effort on part of the developers; and the process itself introduces the possibility of errors (selecting the

wrong version, for instance, or failing to get fresh copies over a period of time, resulting in development against old source.)

The essential goal of a shared source cache is to have the system extract a single reference copy on a regular basis—daily, in most cases—for all the promotion levels that developers would want to use in a common location per level. In the example given, it would be expected that the *DevBuild* level would be extracted nightly, while *Production* would only be extracted when a new production candidate passes testing and is released. Only in the case that you had to have the absolute latest, bleeding-edge software would you have to go back to the version control database to extract source code for compilation and/or linking when you're testing your changes.

This requires some way to actually *use* this source. In the crudest model—where there's no support for shared source caches (commonly known as *viewpathing*), scripts or programs can actually copy the source trees to the developers' work area. More commonly, though, build tools can be told there are alternative locations to look for sources to satisfy dependencies.

6.4 SHARED OBJECT CACHE

Most developers are only changing a small subset of the entire software application. When they want to carry out unit or integration testing, even with a Shared Source Cache they'd have to recompile and/or link objects, libraries, or executables. Given the unchanging nature of most of these objects in the course of development, this is wasteful and slow.

Thus, along the same line of reasoning as given for the Shared Source Cache, the concept of a Shared Object Cache is to provide a single copy of all the objects, libraries, executables—essentially all the *derived object* that result from a build cycle—in a common location. This usually is the result of the Daily Build.

The same issue exists for the objects as for the Source Cache in terms of the build tools—they should know how to search alternative locations to satisfy dependency determination for builds.

6.5 SHARED SOURCE ESCALATION

This is a concept that has gained increasing approval in recent years. The problem is that there is often code that is shared between multiple projects or products in a development organization—utility libraries, for instance. Software version control system developers have expended a fair amount of effort on trying to share this source between different projects—and in most cases, it simply doesn't work well or reliably. Complex issues exist, such as determining order and interweaving of change requests from different projects; synchronizing code changes; identifying release levels; and "network lockouts" of common code stalling development in multiple projects.

It's turned out to be more reliable and manageable to take a different approach, and consider any software that is used in more than one product or project a versioned software project in its own right. A simple concept—if you want a different release of a common software component, run it through its own development cycle. It then has a version, change description, etc.

The usual objection is that this "takes too long", or is "too much work". In practice, however, this need not be the case. Interim or development releases can be used for ongoing development. Moreover, this makes very visible changes that may affect multiple development efforts, and guarantees that changes are reviewed and discussed before simply "appearing" in all users' code base. It also encourages data encapsulation and reduces side-effect coding.

7. CURRENT ENVIRONMENT

At this point, the document ceases being a presentation, and becomes a working document for planning a new SCM environment. In following sections, carry out any necessary research and either fill or follow the directions in the <<*bracketed*>> sections as appropriate. Delete the bracketed sections afterward.

7.1 CURRENT SCM MODEL

The current SCM model evinces the following characteristics:

- *<<What are characteristics, advantages and disadvantages of the current software management scheme, e.g., what provides Source Control?>>*.

<<Provide more detailed explanations of bullet items above if necessary.>>

7.2 PROPOSED SCM MODEL

The following roles should be implemented in the SCM environment:

<<In the following, ***don't*** *just delete a role—but if one person will fill two or more roles, explain that and* ***then*** *delete the role.>>*

- *Development Team Lead*
- *Development Build Manager*.
- *System Test Build Manager*.
- *SQE Test Manager*

The proposed SCM model will incorporate the following features and SCM concepts:

<<Include whatever features you've decided you need, after reading the introduction sections.

- *Bill of Materials*.
- *Hierarchical Promotion Group Model*. The model will utilize four promotion group levels: *Frontier*, *DevBuild*, *SystemTest*, and *Production*.
- *Change Management and Modification Request Tracking*.
- *Daily Build*.
- *Shared Source Cache*.
- *Shared Source Escalation*.

Note that in the Matlock development environment, the advantages to be gained by implementation of a Shared Object Cache are outweighed by the inability of the base Microsoft DevStudio build environment to support viewpathing. Furthermore, the entire code base can be rebuilt in significantly less than an hour, which is more than tolerable.

To provide these features, the following activities are required:

- *Replace Visual SourceSafe with a version control tool that fully supports labeling.*
- *Define and document the development cycle with respect to SCM tracking and controls.*

- *Acquire a change management tracking tool, particularly one that can be integrated with the software development and version control cycle and tool(s).*
- *Restructure the current development archive and import into the new VC tool archive.*
- *Assure all objects are collected under version control.*
- *Provide designed and controlled build environments and procedures.*
- *Provide for planned and reversible software upgrade processes.*

7.3 MIGRATION SCHEME

Note that this section has been stripped due to the planning changes discussed on 08/22/01. Its organization and content needs to be discussed.

7.3.1 Short-Term Relief

7.3.2 Design New Archive Structure

7.3.3 Document New SCM Environment Usage and Design Model

7.3.4 Carry out Full Testing on a Copy of the New Archives

7.3.5 Freeze Current Archives and Migrate with Backout Option

Appendix I

SCM References

For more detailed treatments from a number of sources, I'd suggest starting with papers and locations such as:

- *Software Reconstruction: Patterns for Reproducing Software Builds*, available as part of the Proceedings of the Pattern Languages of Programs '99 (http://jerry.cs.uiuc.edu/~plop/plop99/proceedings/)
- The ACME project, www.enteract.com/~bradapp/acme, itself a pointer to numerous SCM references (but check references--a number are stale today. The Web evolves rapidly.)
- *The CM Yellow Pages*, http://www.cmtoday.com/yp/configuration_management.html http://www.cmtoday.com/yp/configuration_management.html);
- Watts S. Humphrey's *Managing the Software Process* (still a seminal reference, despite its publication date.)
- The Carnegie-Mellon CM archive http://www.sei.cmu.edu/legacy/scm/scmDocSummary.html

There are a huge number of other sources, but these alone are "nested" enough to provide a broad base with which to start.