Spectrum Software, Inc.
6855 Jimmy Carter Blvd.
Suite 2150
Norcross, GA – 30071
www.spectrumscm.com

Subject: Version Control is Not Configuration Management

Issue Date: **February 11th, 2002**

From: **William C. Brown**
corey@spectrumsoftware.net
(770)448-8662

1.0 Introduction: Software configuration management is not defined by version control alone. There are many CM solutions available today that do little more than provide version control, but are promoted as complete *software configuration management* solutions. These limited use systems are generally built on top of existing platform dependent file versioning solutions and make use of a series of loosely connected management features. In contrast, complete CM solutions are generally built from the ground up with highly integrated management, labeling and versioning features integrated together. These systems utilize high performance versioning solutions and industrial strength database engines to guarantee the integrity of file versions and meta data. True CM systems are not only easy for developers to use, but also provide managers and configuration specialists (build engineers) with the tools they need to successfully build projects and manage the development team. The purpose of this paper is to describe and illustrate the major differences between simple version control systems and systems that provide complete configuration management capabilities.

2.0 Version Control Systems: Simple version control systems have existed in one form or another for many years. The primary job of any version control system is to track the change deltas that are created when source files are changed from one *version* to the next. Early version control systems, like SCCS[1] and RCS[2] rely on storing file delta information along with the actual sources, in one or more operating system supported files. SCCS in particular stores version deltas in several different files and rebuilds a complete version of a source file from the ground up each time the file is accessed**.** RCS on the other hand, stores the most recent version of a file as the head revision and then applies edits to retrieve earlier versions. Both systems rely on the native file system as their data repository**.** Using the native file system provides essentially endless scalability but also exposes potentially sensitive or proprietary information to anyone with file system access**.**

[1] SCCS stands for Source Code Control System
[2] RCS stands for Revision Control System

2.1 Branching: Branching in early versioning systems is done by adding additional version numbers to the existing release and level numbers. Here's an example of a simple progression of revisions:

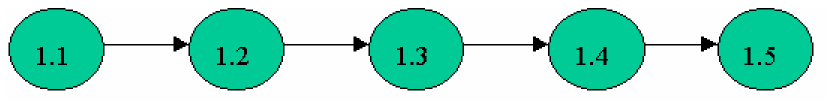

Diagram 1.0

In the progression above, the original file has progressed from version 1.1 up to version 1.5. Imagine for a moment that at version 1.3 of this progression a set of files were grouped together to form a particular release for a customer. Since creating this release, development work has continued and the file(s) are now at revision 1.5. Unfortunately, the customer has found a bug in the release and is expecting a patch. The customer is also not ready to accept the additional untested work that has gone into version 1.5, so a branch release is planned. The branch is performed at version 1.3 and the following progression shows the results:

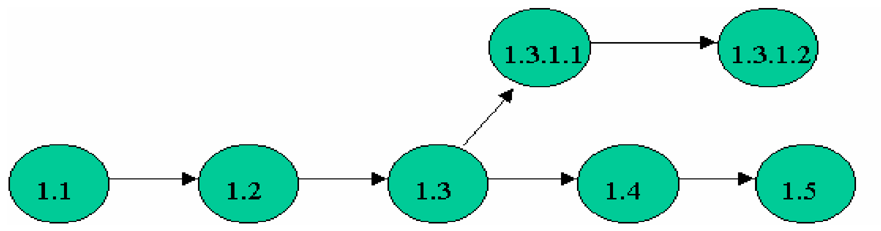

Diagram 2.0

In this progression, the branch has been formed at version 1.3 and the file has been progressed into versions 1.3.1.1 and 1.3.1.2. The branch was used to correct the bug found in the earlier release and the changes are then released back to the customer. Now the changes found in version 1.3.1.2 must be merged back into the mainline. If the merging process is not completed, the bug found in release 1.3 will be propagated into the next release of the main line. Here's what the code line will look like after the merging operation:

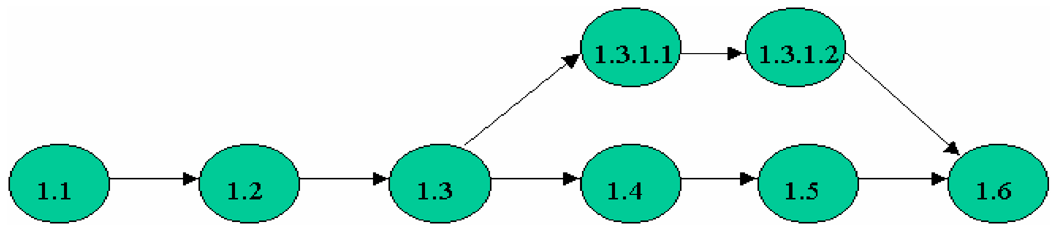

Diagram 3.0

Now version 1.6 of the file includes all the changes up to 1.5 and also includes the patch done in version 1.3.1.2, which was used to correct the bug found by the customer.

This is a trivial branching example involving one file. Imagine the complexities involved with manually managing all the individual branches necessary to merge a large number of changes, involving many different files, back into the main line. This is the first area where true CM systems begin to out perform simple versioning systems. True CM systems make the branching and merging process much easier for the user. While the underlying mechanism may still work with single file versioning, the solution that is presented to the user usually involves a much larger container. Systems like SpectrumSCM, Perforce, AccuRev and ClearCase all use similar concepts to handle branching operations. In these systems, files are not branched individually. Branching is accomplished by creating a container that acts as a branch repository. Files that are normally visible in the main line are also visible in the branch. The files also share the same version numbers until they are checked out or *uncommoned* into the branch. This mirroring effect allows individual developers or groups of developers to work together on isolated versions of individual files, while still sharing a view of the work that is proceeding along the main line. The technology is called different things in different tools, but it essentially accomplishes the same thing. Most of the better tools provide a mechanism for combining groups of branched changes back to the main line through a promotion or integration process. The process is usually fairly automated and can easily move new files that only exist in the branch into the main line, while also assisting in the integration of files that require a merge operation.

2.2 Labels: In simple CM systems, labels are used as a collection mechanism for storing file versions. Labeled collections can be used as repositories for software releases or for other activities. The primary problem with simple labeling systems is that adding file versions to different labels is a very manual process and leaves ample opportunities for error. Labels also have a tendency to change over time, which means that unless locked, additional file versions can be added to or removed from a label even after the label is associated with a particular software release. Locking can solve the problem with labels changing over time but not all simple CM systems support label locking.

Labels can also be used to define branches or user work areas. In the **CVS**[3] system, which is built on top of the RCS system mentioned above, these labels are known as tags. Users must manually associate file versions with particular tags and they must be prepared to manually merge their tags back into the main line at a later date. The task of merging tags can be a frustrating process and is usually left up to a person designated as the *merge manager*. The number of outstanding tags or labels that must be merged back into a main line or other branches can lead to a nightmare of

[3] CVS stand for Concurrent Versions System.

merging activity and often leads to a system that will no longer compile or operate properly.

Labels and tags are not a manageable solution. From the standpoint of identifying a particular feature or bug and following that feature throughout the development lifecycle, labels and tags fail miserably. Better CM solutions combine the benefits of labels (collections) with a tracking system that allows the feature set or problem (bug) to be tracked throughout the software development lifecycle. These tracking mechanisms are usually known as Change Requests or Change Lists and are created as part of a need to add additional features to a system, or to track the resolution of a particular problem.

2.3 Tool Layering: Tool layering is a method that expands upon the concepts of labels and provides additional manageability for system changes. Tool layering can usually be found in lower end CM systems that have grown out of simple versioning systems into more complex systems. The individual pieces of a layered system can usually be purchased separately, as the version tracking engine and the management tools are usually not tightly coupled together.

The addition of a tool layer is an attempt to provide some form of manageability to the process of identifying and tracking new feature sets or bugs. Where layered tools fail is in their inability to identify individual code changes with a particular feature set or bug. The process of actually adding a new feature or fix for a known bug is handled in the usual fashion. It is left up to the developer to identify what changes were made to the system and to faithfully enter a summary of those changes into the layered tracking system. The contents of the tracking system cannot actually be used to identify a particular release or to properly identify what features or changes actually exist in a particular release. As usual, unless the management tools are properly integrated into the versioning system, the contents of the two systems will diverge and the management tools will simply become a bulletin board for tracking requirements or testing issues.

Full featured CM systems require that all changes to managed sources are done through a change request or change list. This process guarantees that all changes made to the system are properly tracked. The end result is a system that is defined as a set of change requests, which are then used to actually construct a releasable system. Without this type of manageability, it is impossible to track the changes to the system made by the development organization. What usually results from not using integrated management tools, is that features slated for a future release will creep into an early release before they have been properly tested. A prime example of this can be found in the Telecommunications industry. Telecom equipment manufacturers must release features in their communications equipment at the same time that the corresponding management features are released in their EMS (Element Management System). By using properly integrated tracking tools and change requests, both features could be managed and released at exactly the same time. Simple versioning systems don't provide the management features required to handle this type of phased delivery.

2.4 Release Management: Release management is the ultimate end game in configuration management. The ability to properly reproduce any previous release of a software system and to be able to identify the feature sets and bug fixes in a release is very important. CM systems that allow branches to be removed and re-added at different locations, or allow labels to be changed after a release has been generated, fail to meet these requirements. Most simple version control systems fail to provide proper release management capabilities. Label based systems only provide the ability to identify a collection of file versions as a particular release. They don't provide the ability to identify which feature sets or bug fixes are included in the release. Tags and labels used together are only marginally better as the creation of the tags is still completely under developer control.

2.4.1 Release Management and Change Requests: Change requests and issue tracking provide the only rock solid way to satisfy the requirements of release management. Only change request based systems allow the system engineer to associate a known set of features and bug fixes to a particular release. Consider the following diagram:

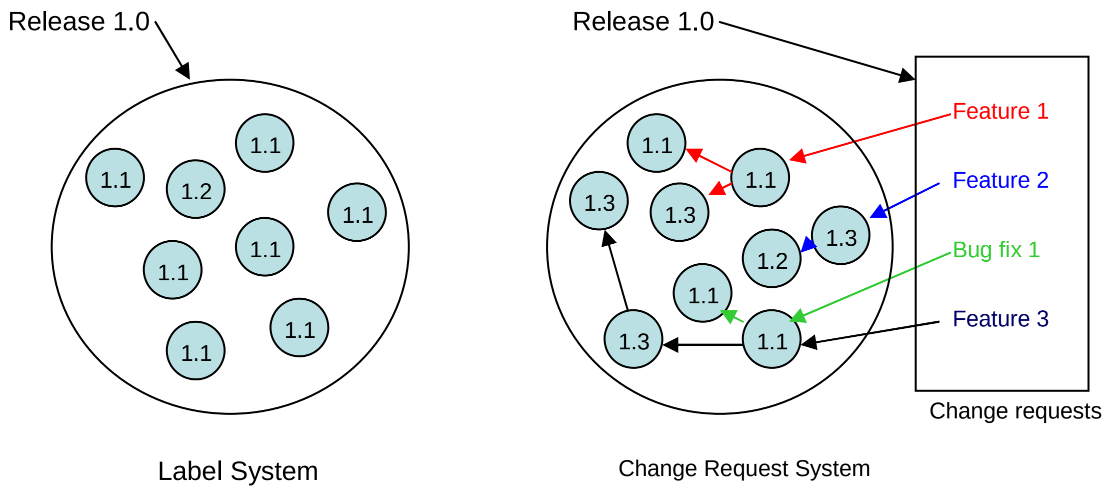

Diagram 4.0

The label based system on the left only provides the identification of a set of versions that make up a complete release. The change request system on the right actually creates the release by associating a set of change requests to the release name. The collection of change requests properly identifies the features and bug fixes that are included in the release.

2.4.2 Patching a Released System: Customers cannot always upgrade to the latest release of their software systems when they find a bug. The ability to reproduce a particular version of a software system and then to be able to patch that system without accidentally including new features, is very important. Change request based systems, in particular the SpectrumSCM system, allow for the creation of a branch based on a previous release. The branched system contains only those file versions that were used to create the particular system. Once the branch is created, developers are free to extend the system (using a change request) to fix the problems. When the bugs have been addressed, a new release can be constructed from the old release and the newly added change requests.

3.0 Complete CM Systems: Complete CM systems combine all of the features outlined above, into a system that works throughout the entire development life cycle. Unfortunately, there is no one single definition of the development life cycle. Depending on the organization, the development life cycle may be very minimal or extremely complex. Full featured CM systems must be able to work within the context of the life cycle as it is defined by each individual organization. Several full featured CM systems, including SpectrumSCM, allow for custom tailoring of the change request life cycle to fit the needs of the organization.

3.1 Change Requests and the Development Life Cycle: Change requests are an integral part of the development lifecycle. As work progresses through each development phase, change requests are used to initiate, track and eventually complete work. Consider the following development lifecycle:

3.1.1 Feasibility and Requirements: Requirements gathering and feasibility studies define some of the earliest stages of software development. It is at this stage that hard requirements usually start to take the shape of a collection of UML based Use Cases or some other form of requirements gathering methodology. It is at this point that a true CM system becomes useful. Not only can the actual requirements documents be versioned in the system, but the use cases can be expressed as initial change requests for particular features. As the requirements change, the documents under version control can be changed and the feature change requests can be updated to include insight into the decision making processes that were used to change any initial designs.

3.1.2 Object modeling and Architecture Discovery: At this stage in the development lifecycle the system designers have started to create analysis models and system developers have started to create design models for framework prototypes. Initial sources for the development frameworks are loaded into the CM system using a single change request. This change request will be used as the foundation for the rest of the system. As soon as possible, an initial release is constructed using the initial source change requests. From that point on, all of the individual feature sets will be developed with independent change requests. At this stage, large amounts of work are being done under a handful of open change requests.

3.1.3 System Development: Once the base frameworks have been added to the CM system, open system development can begin. All changes to the system are done through individual change requests. This will allow the build engineer to selectively exclude some features from a nightly build operation while allowing others to be included.

File level dependencies play a major role in determining which features need to be included in nightly builds. In the beginning, the ratio of sources to change requests is large and more file level dependencies are created. In later stages of development, when less work is done against each change request, the amount of file level dependencies drops and individual change requests can be excluded from a build more easily.

3.1.4 Unit and Integration Testing: In the unit and integration test phases, new change requests are created to address bugs found in the system and old change requests are either assigned back to the developers for additional work or promoted into a ready for system test state. Now the ratio of source changes to change requests begins to drop and individual change requests can be included or excluded more easily.

3.1.5 System Testing and Final Release: In the final testing phases, individual change requests are tested against the actual requirements. Problems found during this phase are noted in the feature change requests and they are reassigned back to the developer for additional work. The number of times that a change request is promoted and then reassigned is recorded within the change request itself. At this point, examining troublesome change requests can reveal weak areas in the system that may need additional attention.

The first official release of the system is created by the build engineer by including completed, tested and dependency free change requests into a new release. The build process is initiated by extracting the release from the system and then invoking the actual build process.

3.1.6 Development Life Cycle Summary: The integration of testing phases and the close interaction between system engineers, developers and system testers is enabled by the power of a complete CM system. Without the ability to track and manage changes throughout the development life cycle, the development process can degrade into a chaotic mess. Losing track of the features and bug fixes included in the release will lead to trouble later on when an earlier release of the system must be created and patched.

3.2 Branching and Integration: As outlined above, branching and merging in high end CM systems is handled differently than the way these activities are handled on simple versioning systems. The SpectrumSCM system in particular, uses a generic branching concept that essentially mirrors an entire system. Other systems that operate in this fashion refer to the main line or previous branch as a code stream. Additional streams (branches) are *backed* by another code stream. Code streams can be previous generics, including the main line trunk, or a previous code release. Creating a branch from a code release is an extremely powerful concept, which allows the development team to patch an existing code release, without disturbing other code streams. Consider the following diagram:

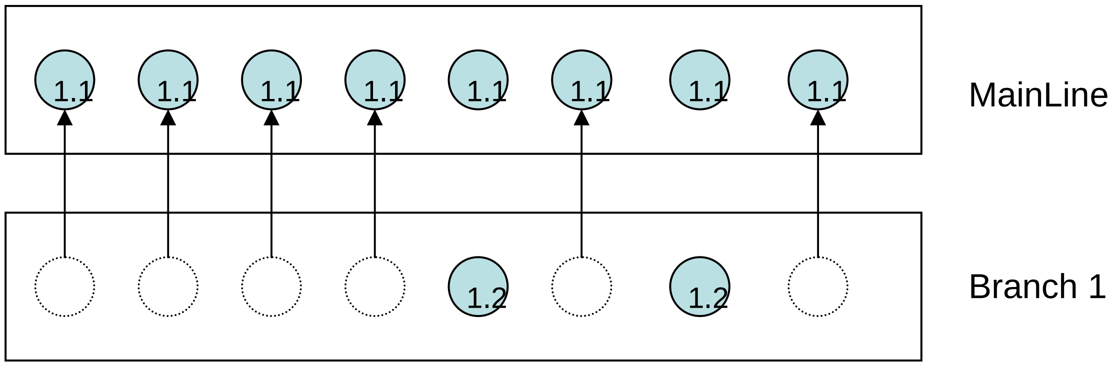

Diagram 5.0

The actual code branching is done at the file level, but unlike RCS based systems, the branched (*uncommoned*) files are duplicated within the new branch. Only the files that actually change and are marked *uncommon* are physically duplicated. Otherwise the file pointers for each branch remain the same. The generic branching solution hides the difficulties of individual file branching from the user. The user is no longer burdened with having to remember each individual branched file and to later add the revision numbers to some form of tag or label. This is all handled by the CM system through the generic process in conjunction with change requests.

Edits performed on files in a branch can either be common with the original trunk line or can be uncommoned with the trunk. Performing an edit common to the trunk allows for problem resolution in two or more branches of the code at the same time. Uncommon edits allow for parallel development of new feature sets that will not impact work proceeding on the main trunk line. All editing activities are done relative to an active change request, to ensure that the work can be properly tracked and built.

Integration of completed work is done at the change request level. Each file uncommoned by a particular change request is recommoned back with the original trunk line through a merging/recommoning process. This process involves using a merging editor to reconcile the new code changes with the original branched files and the addition of new sources common with both branches.

3.3 Release Management: As outlined above, release management in a high end CM system is handled differently than in simple versioning systems. In the SpectumSCM system, releases are created by including change requests into a physical release. The release itself is a first class object in the system and not simply a label associated with a series of file versions.

Extracting a release for the build process is as simple as selecting the release in the GUI and selecting a destination for the extracted files.

Once created, a release can remain open (extensible) for an indefinite period of time. The release engineer has the ability to physically lock a release, which prevents the addition of new change requests to the release or the deletion of existing change requests from the release.

4.0 Conclusion: In the end, a choice has to be made whether to standardize on a simple version control system or to go with a more comprehensive CM system. Simplicity may be a deciding factor in making a final choice but simplicity via simple file versioning may lead to an overwhelming amount of complexity in the later stages of software development. Extensive branching and merging in simple version control systems can often lead to highly unmanageable situations, especially in shops that do not have extensive configuration management skills. Complete CM systems introduce the concept of change requests, which tend to impart a process on software development. It is the definition of a software development process that improves the manageability of the entire software development life cycle. Day to day development activities, branching/merging, requirement tracking, feature tracking, bug tracking and finally release management are all areas that are improved upon by a complete CM solution. Version control systems are not configuration management solutions. Hopefully this document has shed enough light on the differences between version control and configuration management, to enable the CM consumer to make an informed choice when selecting a complete CM solution.