Spectrum Software, Inc.
11445 Johns Creek Pkwy.
Suite 300
Duluth, GA – 30097
www.spectrumscm.com

Subject: Advanced Branching Techniques for SpectrumSCM

Issue Date: **May 14th, 2002**

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

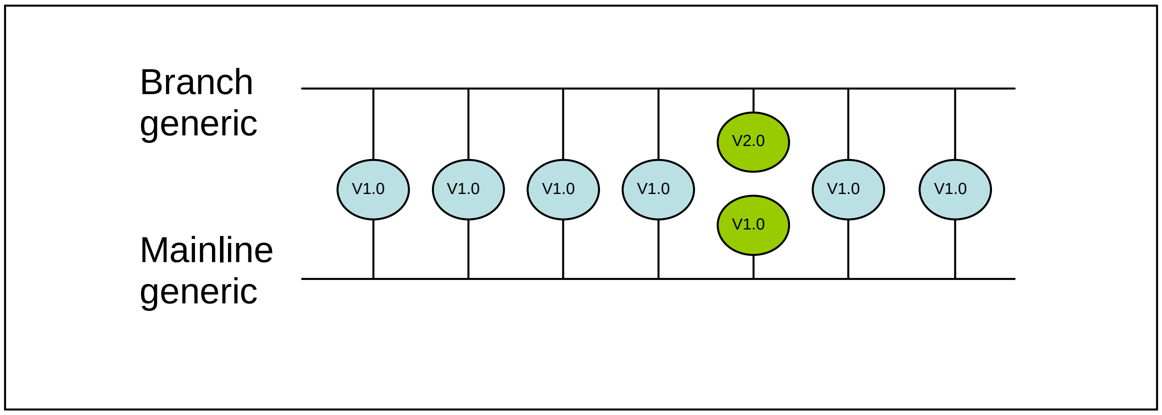
1.0 Introduction: Over the years, developers and system engineers have developed many unique branching techniques to solve difficult configuration management problems. The purpose of this paper is to describe several of the most common branching techniques and to illustrate how these techniques can be implemented using the SpectrumSCM system. SpectrumSCM approaches branching in a significantly different and powerful manner than most CM systems. The SpectrumSCM system introduces the concept of *Product Level Branching* that is unique to the CM industry. Product level branching ensures that branches are well known (documented), controllable and use repository space as efficiently as possible.

Branching techniques can be considered release management design patterns for use in CM (Configuration Management) systems. Like design patterns used in programming techniques, the application of proper design patterns to configuration management will result in the development and evolution of systems that are more maintainable, understandable, extensible and scalable. CM systems should not become a burden by adding to the work load of the development team. A properly used CM system should free the developer from the intricate details of branching and release management. The proper application of branching design patterns can result in systems that are as easy to use, after years of activity, as the day the first code snippets were versioned into the system.

2.0 SpectrumSCM: SpectrumSCM is a full featured CM system that emphasizes process management, issue tracking, release management and version control all in one tool. The system does not impose any one branching design pattern on the users of the system. Users of SpectrumSCM are free to use many different branching design patterns, including all of the patterns outlined in this paper. Most developers are familiar with the most common branching technique, which involves branching single files during code development. For a short period of time, the code is extended in a branch to resolve a particular problem or to introduce a new feature outside the mainline development effort. The branched code is eventually *merged* back into the mainline after the fix has been verified or the new feature set has been implemented. While this is a common technique, and one that is supported by

SpectrumSCM, it's not the best solution for every situation. The following design patterns are supported by SpectrumSCM and, in some instances, are unique to SpectrumSCM.

3.0 The Classic branching design pattern: The classic pattern is the basic branching pattern outlined above. This pattern is the most recognized and most often used pattern for branching code. In some cases it is the only pattern supported by many version control and high end CM systems. The classic pattern allows individual developers to create alternate branches of code extended from the mainline development stream. The existence of such a branch is not immediately obvious to the other users of the system; this can lead to confusion, especially if the owner of one of these *anonymous* branches leaves the company unexpectedly. This type of branching is done at the file level and the branched files are only conceptually linked to a specific branch by a branch *number*. This is readily evident in systems that are based on RCS (Revision Control System) where the third digit in the file version number is greater than "0", e.g 1.3.1.2 which usually means that a branch for this particular file was formed at version 1.3 and is now at version 1.2 of the new branch. Any branching relationship between this file and any other file in the system is known only by the developer who created the branch and is not managed by the CM system. In the SpectrumSCM system, branches are first class objects in the system and their existence is readily apparent to the users of the system. To perform ***classic*** branching in the SpectrumSCM system, a generic is created to ***contain*** the branched files. Notes and other artifacts can be associated with the branch to assure that the purpose of the branch is known and available to every user of the system. When a generic is created from another generic or release, all files shared between the original code stream and the new generic are ***common*** to the two streams. This means that only one first class object for each file physically exists in the system and both branches point to that object. Actual branching is accomplished by ***uncommoning*** a file from the mainline into the branch. When a file is ***uncommoned***, there are two first class system objects, one for each version of the file. The following diagram illustrates the point:



In this example, the green circles represent a single uncommoned file. Each branch contains a separate physical instance of this file and shared instances of all the other files. When the files are ***recommoned***, a single file instance will again be shared by both branches.

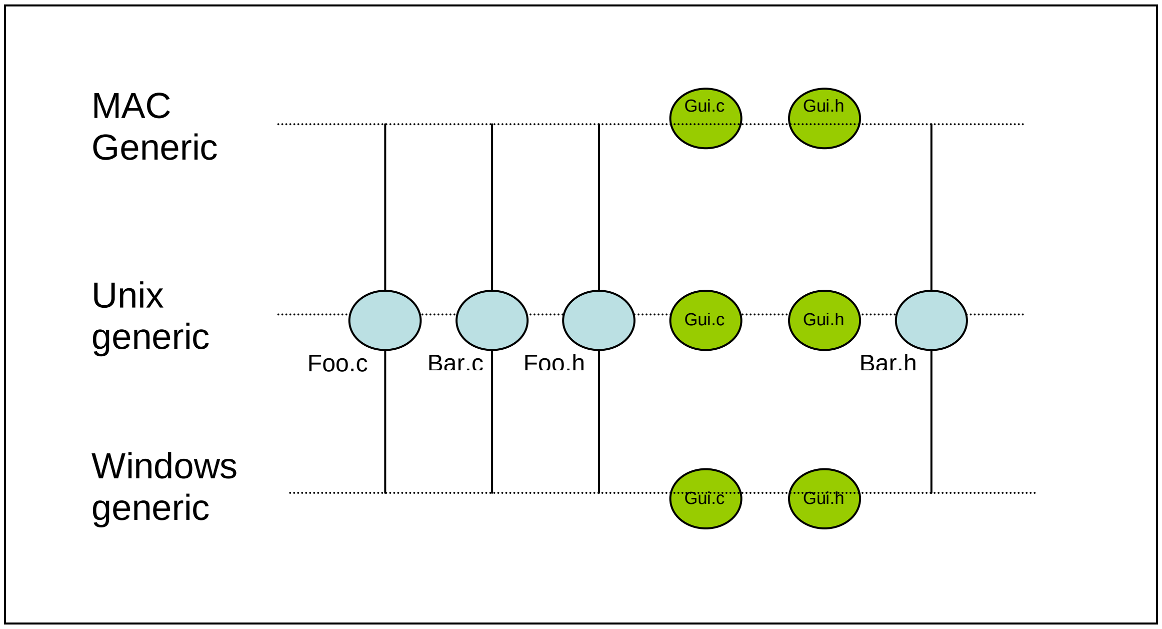
The objective of the classic branching pattern is to diverge one or more files from the mainline, usually for a short period of time, so that custom work or bug fixes can be applied outside the mainstream development effort. At a later date, the changes are merged or ***recommoned*** back into the mainline development stream. In the SpectrumSCM system there is a significant difference between merging and recommoning. **Recommoning** makes one single source instance out of two independent entities. **Merging** combines the contents of the two files, but the two files remain physically separate.

Solution Summary: The Classic branching pattern is used to diverge small numbers of files from the mainline code stream, for a short period of time, in order to fix a known problem or to implement a new feature. The diverged files are merged back into the mainline code stream when the work has been completed.

Benefits: The classic branching pattern allows individual developers to separate files into a protected environment away from mainline development. When SpectrumSCM is used with the classis branching pattern the branches are clearly visible and well documented.

Consequences: In systems that do not treat branches as first class objects, the existence of a branch for a particular code stream may remain unknown to other developers.

4.0 Parallel Development pattern: The parallel development pattern is very similar to the classic pattern in that two or more branches are created, but in the parallel pattern, some files are never merged or recommoned back with the mainline. The parallel development pattern might be used during the development of a product for use on multiple operating systems. The vast majority of the functionality and source files are the same on all operating system, but some files must be unique to support the differences between the operating systems. For example, the direct video calls for any GUI components will most certainly be different and will thus require different code. The implementation of the second or third generic (branch) is exactly the same as in the classic pattern except that some files will never be recommoned. Each generic will become a platform-specific release of the product. The following diagram illustrates parallel development for an editor that will run on three different operating systems:



In this case, the files Gui.c and Gui.h are different for each operating system and must remain diverged. The MAC, Unix and Windows generics all share the vast majority of files and only the files necessarily different to implement the GUI on each OS are diverged.

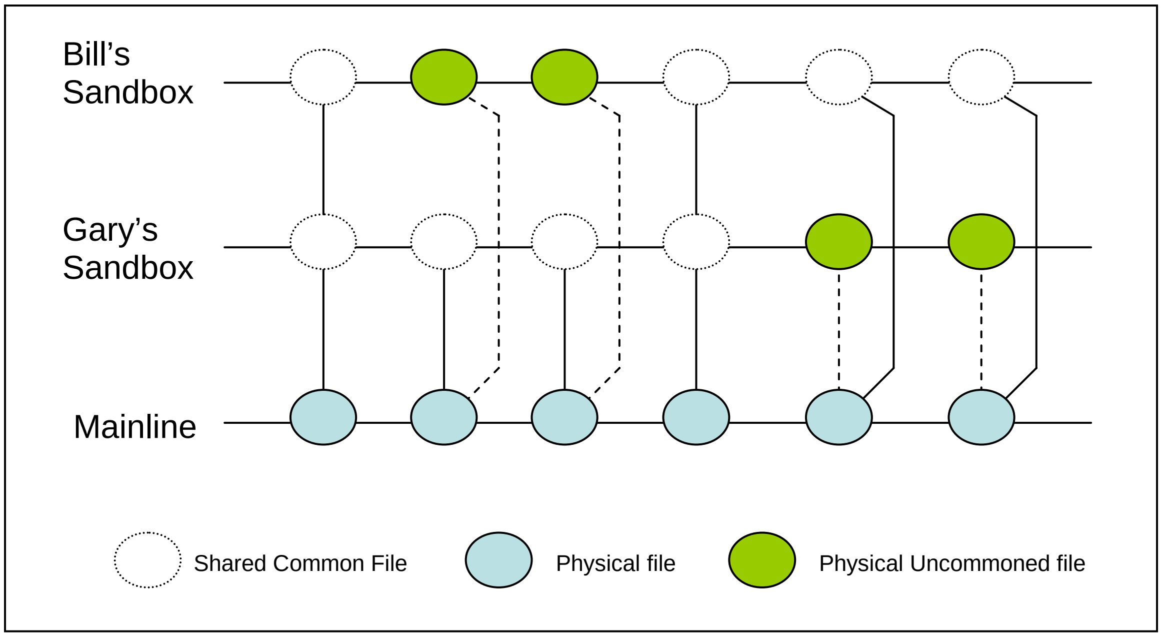
This is where one of the strengths of the SpectrumSCM system becomes very apparent. There are three separate streams of work, one for each of the three supported operating systems. But the vast majority of the files that make up the product are common. As a result, when problems are fixed in these common files, all three generics get the fix at the same time. The CR (Change Request) that is used to resolve the issue is available to be included in a release on all three generics. *This feature, which is unique to SpectrumSCM, relieves the developer from fixing the same bug three times in three separate branches of the code.*

Solution Summary: The parallel development pattern allows two or more separate code streams to be developed in parallel.

Benefits: In the SpectrumSCM system, code changes to common files are immediately visible across all parallel development efforts. Separate releases can be built to produce a product specific to a particular need (for example, different operating systems). The parallel pattern may also be used to develop multiple parallel releases of the same system, diverging only those files that will be impacted by each new feature.

Consequences: Parallel development requires the development lead or project manager to decide which files are to be branched. Failure to diverge the necessary files will result in case-specific code changes being visible (common) in all of the parallel branches.

5.0 The Sandbox pattern: In this pattern, all work is performed in separate generics before being integrated back into the mainline. The most attractive feature of this pattern is that the mainline branch and the development branch are always in a known good state. New features are first developed in separate sandboxes and then integrated into the mainline only after the new features have been thoroughly tested and approved by the testing organization. When the features are recommoned into the mainline, the developers can extract the code from the mainline and build a system with a known set of working and tested features. This pattern assures that the mainline is never in a quasi-buildable state, which happens quite often in traditional development. The development branch, because there is one branch per developer, always matches what the developer has in her private work area on her machine. The developer is free to work independently on a separate branch without impacting other developers. Consider the following diagram:



In this diagram the transparent circles are the shared common files which are common to the mainline. The blue circles are the actual physical instances of the files that the shared common files point to. The green circles are unshared physical files that have been uncommoned into Bill's and Gary's sandboxes. When these developers are finished with their parallel development work, the files will be recommoned with the mainline. After recommoning, the circles will become transparent like the others.

The sandbox pattern enables long-term parallel development. Most CM systems offer some form of parallel development in the form of concurrent editing. The problem with traditional concurrent editing is that it is file-based;

as soon as a programmer checks in a concurrently edited file, it must be merged back to the mainline code stream. Sandboxes allow long-term parallel development by allowing the programmer to freely check in and out any amount of code, for any amount of time, without disrupting work that may be in progress on the mainline. Only after the entire new feature has been tested and verified will the new code for the feature be merged back to the mainline branch.

The only caveat to this pattern is that it is an “all or nothing” pattern. All developers on the project team must use this pattern or they cannot use it at all. If files are uncommoned into the mainline and also into individual sandboxes, it becomes difficult to recommon the files back into all of the parallel generics. Consistent use of the sandbox pattern guarantees that the recommoning effort will be trivial, involving only a single merge of each file. The sandbox pattern closely resembles the parallel development pattern, except that all files will be recommoned into a single generic when development work is complete.

SpectrumSCM may be the only CM system that properly supports this pattern.

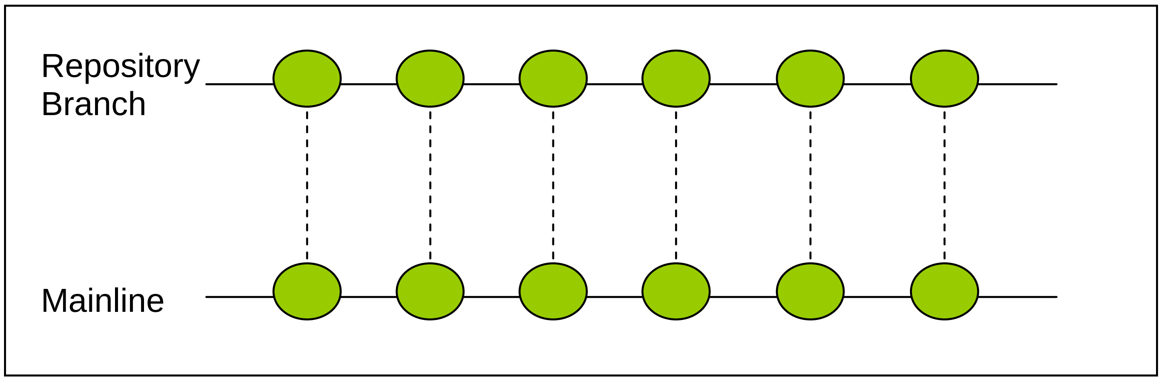
Solution Summary: The Sandbox pattern provides each developer with a separate environment in which to work on new features or bug fixes. This pattern provides for long term parallel development that is completely isolated from the mainline.

Benefits: The ability to manage long-term parallel development activities and isolate them from other development efforts and fixes.

Consequences: All developers must use this pattern for the application of the pattern to be successful. *The SpectrumSCM system may be the only system that properly supports this pattern.*

6.0 The Promotion (Repository) Pattern: This pattern is similar to the sandbox pattern, but it operates in reverse. Using the promotion pattern, all work is done in the mainline and only after a feature has been thoroughly tested and approved is the feature promoted to another branch. The promotion branch or **repository** branch is where all feature sets are included to create system releases. The mainline becomes the development sandbox for all developers on the team. The objective of the promotion pattern is to produce a code repository for all known good work. System releases are generated only from the repository branch. At any time a good system can be extracted and built from the contents of the repository branch.

This pattern depends on classic concurrent editing and at any time the mainline may be in a severe state of flux. Fortunately, developers don't often check unfinished code back into the mainline until it is done; thus the mainline should remain relatively clean, but that is a process issue. The following diagram illustrates the point:



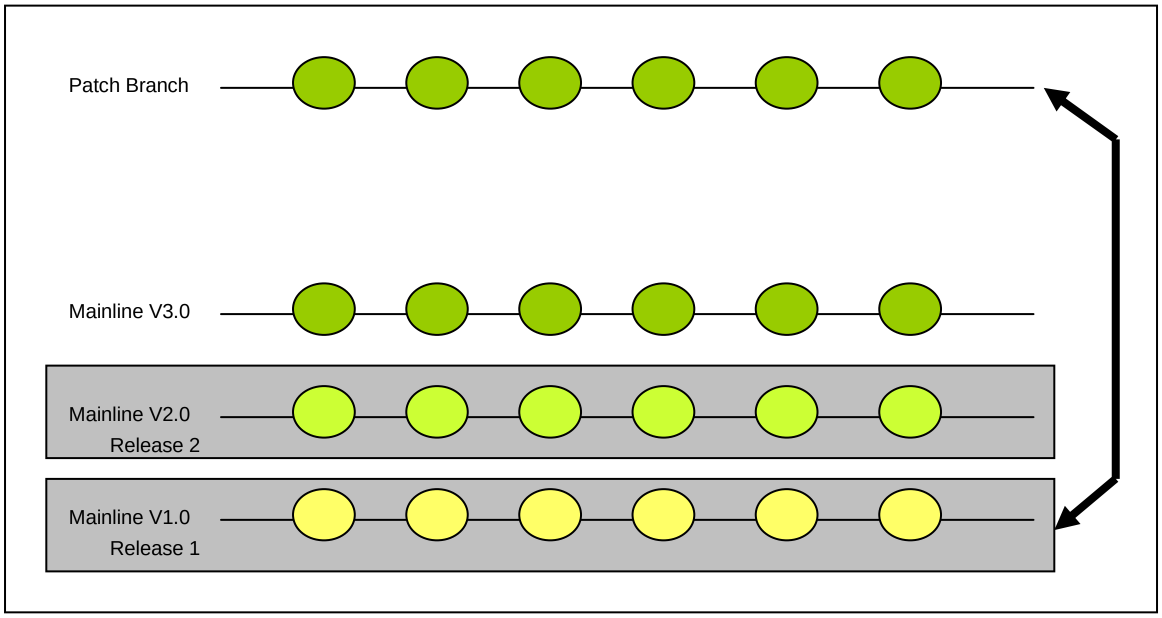
SpectrumSCM easily supports this pattern. After the repository branch is created, all files are checked out uncommon from the mainline. SpectrumSCM allows the project leaders to enforce this behavior by locking the repository generic. Locking guarantees that all files checked out or into the mainline will be uncommoned from the repository branch. Later, when the new features have been developed and tested, the new code is **merged** into the repository branch using the SpectrumSCM Merge Editor or by simply adding the new files to the repository generic. When features are rolled into the repository generic, the work is done by creating a new CR (Change Request) and that CR is used for adding or merging the files into the repository generic. Each CR in the repository generic represents a complete system feature or problem resolution. These CRs are easily included in new releases created from the repository branch. It is extremely easy to determine which feature sets and which bug fixes have been included in a release from the list of CRs associated with that release.

Solution Summary: All work done by the development team occurs in the mainline branch. Only after features and bug fixes have been thoroughly tested and approved are they merged into the repository branch for creation of releases.

Benefits: The repository pattern guarantees that all code on the repository branch is in a known good state. The features and bug fixes included in the repository branch are well documented by the CRs used to create the releases. A release can be created at any point with the features that have been completed.

Consequences: Like all forms of branching, the repository pattern requires a fair amount of file merging to be performed as the new features or bug fixes are promoted to the repository. *SpectrumSCM includes a very powerful merge editor that makes the merging operation fairly painless.*

7.0 The Patch Pattern: The patch pattern is used to repair and re-release previously shipped releases. Typically this pattern is exercised when a customer calls to report a bug, in a particular release of the system. By using the patch pattern, the particular release that the customer is having problems with can be extracted and placed into a new working branch. The branch must be immediately locked so that common files are uncommoned during edit operations. The new branch allows developers to work on the exact file versions that were used to initially create the release. This allows the developers to faithfully reproduce the system as it was released to the customer and to debug and fix the problem. The problem files are extended directly from the released version numbers to add the fix. Once the fix(es) have been added to the patch branch, a new release can be generated, tested and released back to the customer and made available to other customers using that release. Consider the following diagram:



In this example, release 1.0 has been extracted into a new generic called the patch branch. The generic is locked and all of the files that are edited to resolve the problem are either already uncommoned or will become uncommoned as part of the edit operation.

The creation of the patch branch pattern allows developers to use the merge and recommon editors to apply bug fixes from subsequent releases into the newly created patch release. *SpectrumSCM easily supports this pattern and, again, may be one a few CM systems that supports this pattern correctly*. Any system can be used to dump out a previous release (one hopes) but very few systems actually allow the creation of a branch from a previously released system.

Solution Summary: Sometimes previously released systems must be patched due to unexpected problems. End users of the previously released system may be reluctant to upgrade to the latest release due to testing issues and possible downtime. SpectrumSCM allows for any release to be recreated, and patch branches off that release to be easily created. This is done by creating a new generic that is rooted in a previous release of the system. Files in the new branch are visible exactly as they were when the release was created. The calendar is essentially turned back to that time frame and the revision numbers for files in the patch generic are based on the revision numbers of the released product.

Benefits: The patch pattern allows for a previously released version of a system to be easily extended and re-released without impact on other releases or current work. This pattern is not often needed, but when it is, the ability to actually create a branch from a previous release results in a collective sigh of relief from the product developers and development managers.

Consequences: The only consequence is that the branch will be extended and is only needed for a very short period of time. SpectrumSCM allows these short-term generics to be *soft deleted*, which removes the generic from the list of active generics, but they can be restored with a few mouse clicks.

Conclusion: Several different branching techniques have been outlined above. The application of these patterns can result in systems that are easy to maintain, manage and extend. The application of the wrong pattern at the wrong time, or simply not applying any patterns to everyday development work, can lead to the development and evolution of systems that are confusing at best and extremely hard to manage at worst. Some shops avoid concurrent editing or any form of branching simply because their CM tools do not make branching and merging an easy process. Branching, merging and concurrent editing should not be difficult subjects that are only spoken about in soft whispers around the water cooler. If an organization's CM tools do not easily support these features, then it's probably time to think about some new tools. SpectrumSCM supports all of these patterns easily. Branching, merging, concurrent editing, and recommoning are all features of the Spectrum tool that are very easy to use and make difficult CM situations easier to manage.

For additional information on SpectrumSCM please visit our website at www.spectrumscm.com.
Or contact Spectrum Software at 770.448.8662