Chapter

11

11 Branching, Merging and Re-common

SpectrumSCM provides a very simple but extremely powerful way to make and manage branching decisions. Branching in the SpectrumSCM system is handled very differently from most other CM systems, providing many more options in how branching is handled. SpectrumSCM supports numerous branching patterns, allowing simple and effective management of activities, source or project artifacts during the various scenarios that could occur in any project's life cycle.

In this chapter you will learn how to manage branching in SpectrumSCM, how to merge two different versions of files, how to recommon two different versions of files, and how to use the merge editor to merge and recommon files across branches.

11.1 Branching

The concept of branching in an SCM system can be one of the hardest concepts for users of the system to understand. Simply put, branching is a deviation from a main line of file changes. Branching is often used to aid in parallel development and for creating new project feature sets or file content in support of some special needs.

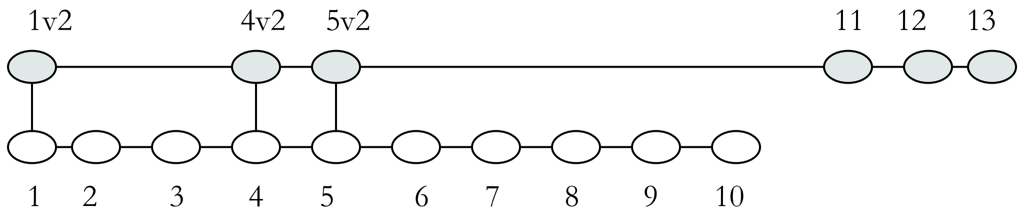
In the SpectrumSCM system, a branch is known as a ***generic***. A generic is a special form of branching that is highly visible to the user and is not limited to single file branching. That is, multiple files may join the same generic "branch" in order to form a specialization of a product or project.

A generic then, is a single branch structure that can accept one or more files off of the main line development stream or another generic. Each generic contains the file changes necessary to customize the product for different platforms. Creating an editor for each OS platform then is as simple as creating a release, or multiple releases, on each generic.

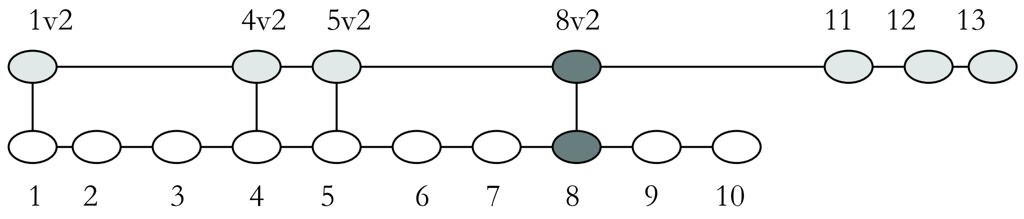
Generics can be thought of as containers for custom work and pointers back to common components. Except for the customizations needed in particular files, all files in a generic share the same physical disk space and physical instances of files with the main line or previous generic upon which it is based. Files that need to be different for the generic are uncommoned. If a file has been uncommoned among generics, it can be recommoned (made the same).

Uncommoning a file is creating two physical disk images of a single file thus creating divergent images of the same file. **Recommoning** is the act of bringing the divergent files back together into the same physical disk image.

As an example, consider a part of the development team being charged with developing advanced features for the editor, in parallel with the current work. The new work could be done in a separate generic on uncommon files. Once the work has been completed and tested, the new features for the editor can be rolled up into (recommoned back) with the main development code stream.
As an example, if a project team has developed and released version 1.0 of a system and they are currently developing generic 2.0, all modules that are changed (modules 1, 4 and 5) or added (modules 11, 12 and 13) during the 2.0 development effort will be "uncommon" - changed only for generic 2.0.

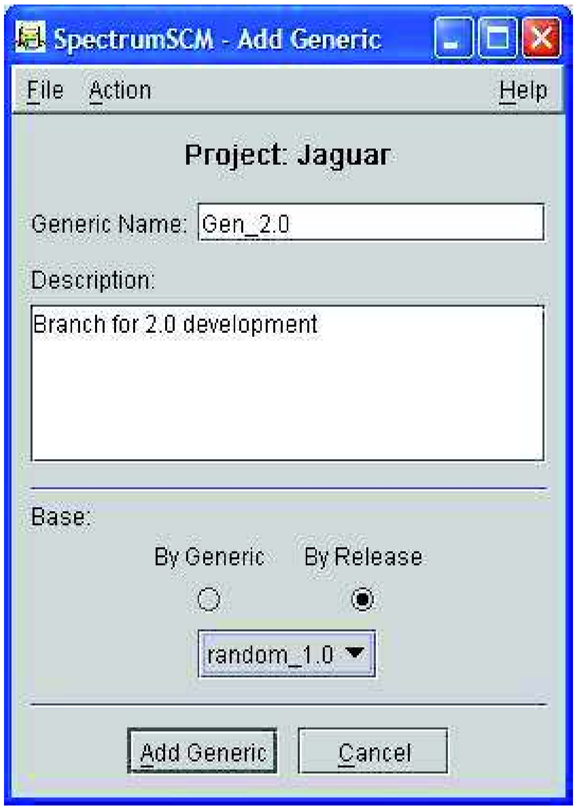


However, if during the development of generic 2.0, a problem is discovered in Release 1.0, the fix for the problem might be made common to both generics. In this example, the problem is in file 8. A CR is created, the code is edited, the problem fixed and the fix is made common to both generic 1.0 and generic 2.0. If the fix needs to be distributed, Release 1.1 can be created, containing the contents of release 1.0 and the fixed file 8. Development of Generic 2.0 can proceed, knowing that the fix will be carried forward.



As another example, there are times when multiple generics are developed in parallel (for example, to maintain 2 similar source bases for 2 different customers). The Generic Engineer must decide who is going to make the branching decisions – who will determine which of the modules will be common or uncommon with other generics.

When a new release of the system is being developed, the changes and additions are based on the previous generic or release as defined on the Add Generic Screen.



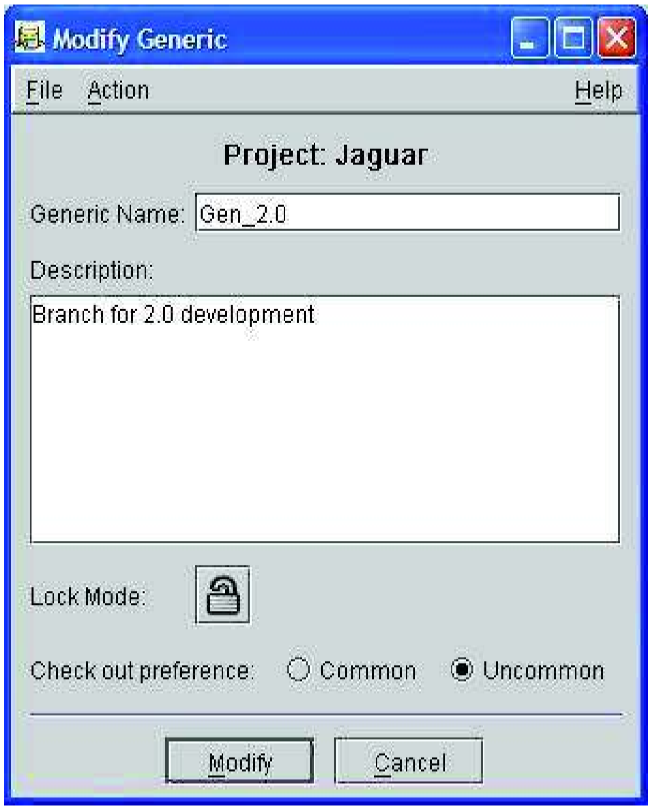
Project: Jaguar

Generic Name: Gen_2.0

Description: Branch for 2.0 development

Base: By Generic ○ By Release ●

random_1.0



Project: Jaguar

Generic Name: Gen_2.0

Description: Branch for 2.0 development

Lock Mode:

Check out preference: ○ Common ● Uncommon

(See chapter 6 for details on creating a new generic)

Changes and additions are typically made uncommon to the previous release. Notice that the default checkout mode (checkout preference) is set on the Modify Generic Screen. If the generic is locked, all changes made on related generics will uncommon the associated files on this generic. If the generic is unlocked, the developer can choose to make a specific change uncommon or common. The Generic Engineer can lock a particular generic to prevent inadvertent changes to other generics or to enforce process control issues. The generic can be unlocked at a later stage, if a need arises to make a common change.

In overview, checking out "uncommon" will mean that any file changes will be isolated to that specific generic. If a checkout is performed "common" then the file changes will be made against ALL the generics that that file is currently in common with.

- **Common:** Versioned files that are physically the same across generics
- **Uncommon:** The act of physically separating versioned files from multiple generics

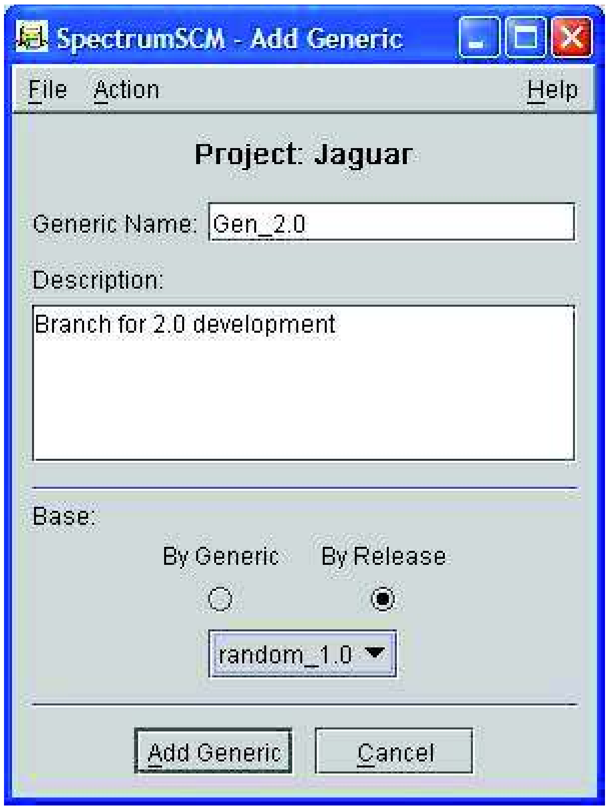
Checking out "common" is a powerful feature since it can be used to apply a single "fix" to multiple generics in one edit, however the developer would have to be careful of side-effects.

The ability to create generics solves the basic file branching problems. A generic is, in essence, one large branch. Files may be added to the branch or remain common with the other files in a previous or concurrent generic. By creating a separate generic to represent a unit of work it becomes much easier to keep track of all the changes necessary to create a new product release. Consider the example of developing an operating system. The initial generic might be developed to create an OS for the Intel platform. By creating a new generic where all the files are initially common between the original and new generic, we can easily break the commonality between generics for specific files in order to create an OS for the Sparc platform. Generics allow us to more easily organize, manage and track our source files for the purpose of creating releases and managing current and future work.

11.2 Creating a Branch (Generic)

(See chapter 6 for details on creating a new generic)

1. Determine the generic or the release upon which the new generic will be based. When a new release of the system is being developed, the changes and additions are based on the generic or release selected on the Add Generic Screen.



Project: Jaguar

Generic Name: Gen_2.0

Description:

Branch for 2.0 development

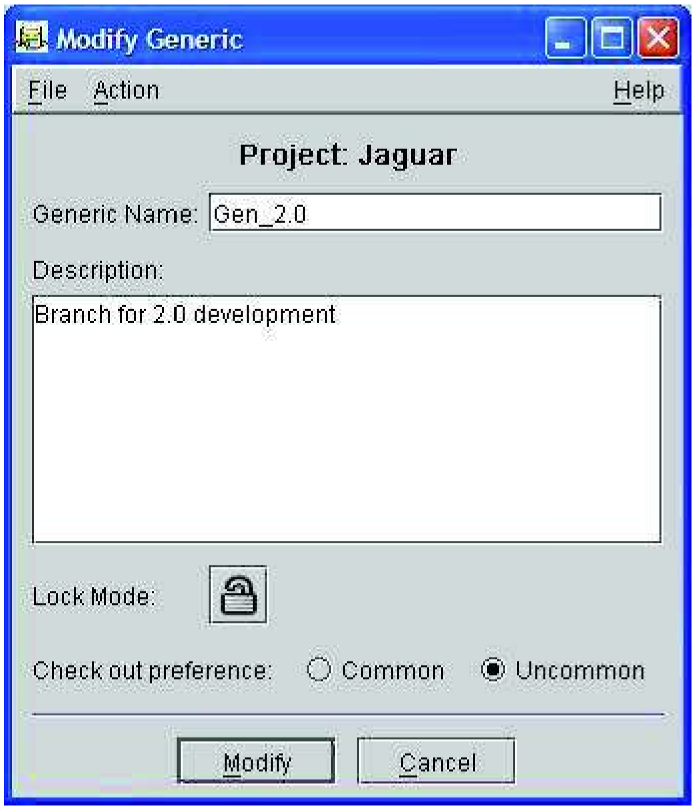
Base:

By Generic [✓ By Release]

random_1.0

2. Create the new generic by selecting the **Administration/Create Generic** menu option. Select the radio button, **By Generic** or **By Release,** based on step 1 above and click on the ***Add Generic*** button.

3. Select the **Administration /Modify Generic** menu option. This will take you to the **Modify Generic** Screen.



Project: Jaguar

Generic Name: Gen_2.0

Description:

Branch for 2.0 development

Lock Mode:

Check out preference: ○ Common [✓ ● Uncommon]

4. Determine whether you want to leave individual common/uncommon edit decisions to the relevant developer or mandate by clicking on the **lock icon** on this screen. The default mode (Unlocked) leaves the choice with the developer since they would best know the source issues.
5. The new branch is now created. Be aware that creating a new generic will lock all releases on the predecessor generics.

To start editing files under this branch, create the CR (s) under this generic by selecting on the Generic in the Project Tree window and clicking on the **Change Request / Create** menu option or by assigning existing CRs for work under this generic. Note that CRs that have already been worked under one generic cannot be reassigned to another generic.

11.3 Merge and recommon

Merging allows two branches to synchronize the contents of the files and still retain the separate development paths. Merging in SpectrumSCM is the act of copying the changes made in one version of a file into another version of the file using the SpectrumSCM Merge Editors. All changes need not be made identical in both files.

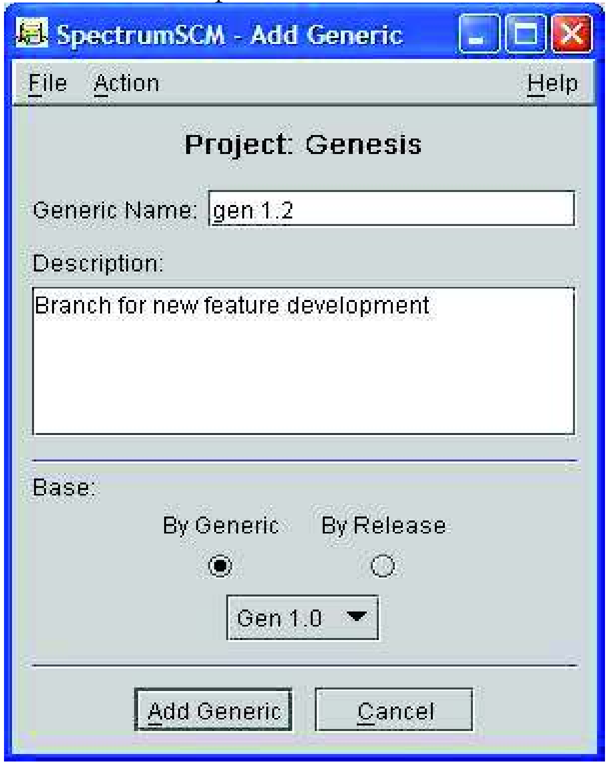
For Example, Team 1 may have made fixes to a file while Team 2 was making different changes to the same file as part of the mainline development. The changes made by Team 1 need to be incorporated (merged) into Team 2's version of the file.

Using the SpectrumSCM Merge Editor for Merge and Recommon
The Split Screen Editor is used for the Merge and Recommon functions. For these functions, modules must be checked out for Merge or Recommon.

In both the Merge and the Recommon processes, the files selected will be put in the split screen **Merge Editor** to allow the changes between the two versions to be identified and reconciled. This will produce a single version of the file that is then made common to the two generics in the case of a Recommon or reconciled and kept separate in the case of a Merge.

<u>**NOTE:**</u> When doing a recommon, the contents of both files in the editor must match when the check-in button is pressed. The editor will tell you if this is not the case. Merging does not have the same restriction.

Merge
To start the merge activity, from the SpectrumSCM Main Screen use the **Extract/ Check out to desktop** menu item and select both files to be merged and check them out with the **Merge** option. Open one on each side of the Merge Editor. This allows use of the SpectrumSCM Merge Editor to see and reconcile the differences between the two files and complete the merging. In this example, Genesis generic gen1.2 is created to develop a new feature. It is based on Gen 1.0.


SpectrumSCM - Add Generic

Project: Genesis

Generic Name: gen 1.2

Description:

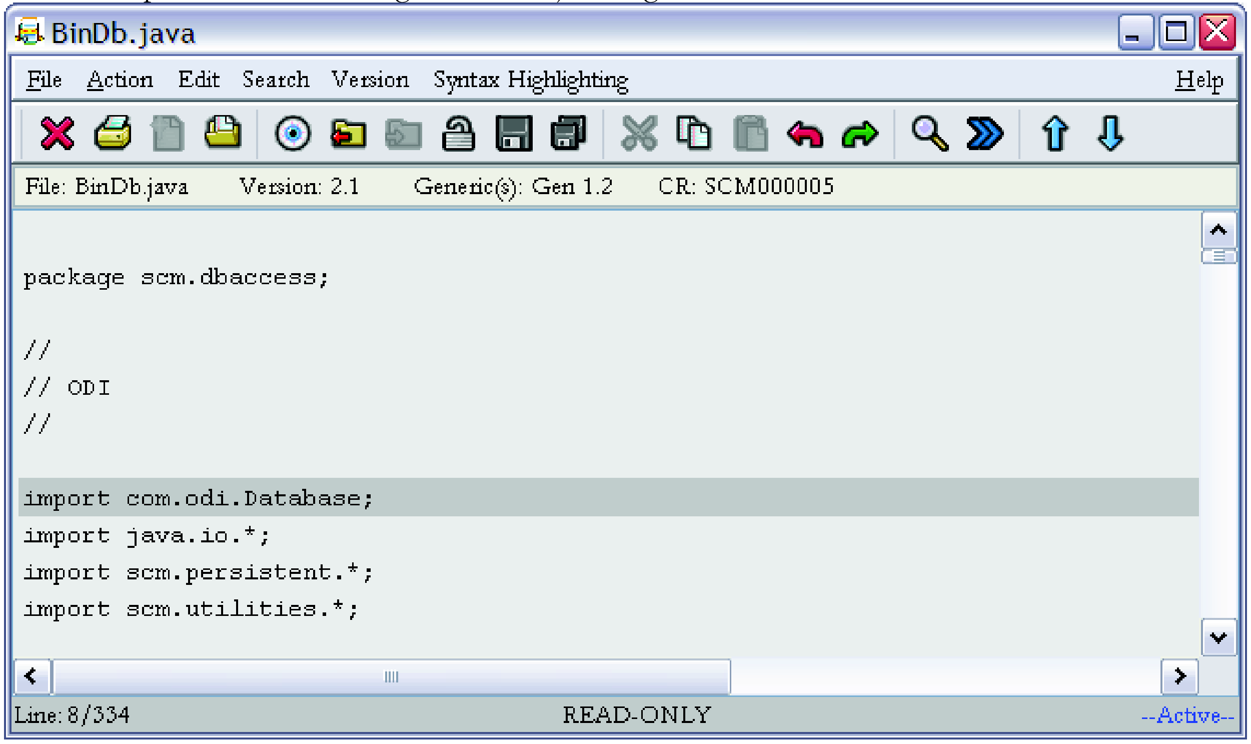
Branch for new feature development

Base:

By Generic [✓] By Release [ ]

Gen 1.0

The developer has made a change to BinDb.java in generic Gen 1.2:

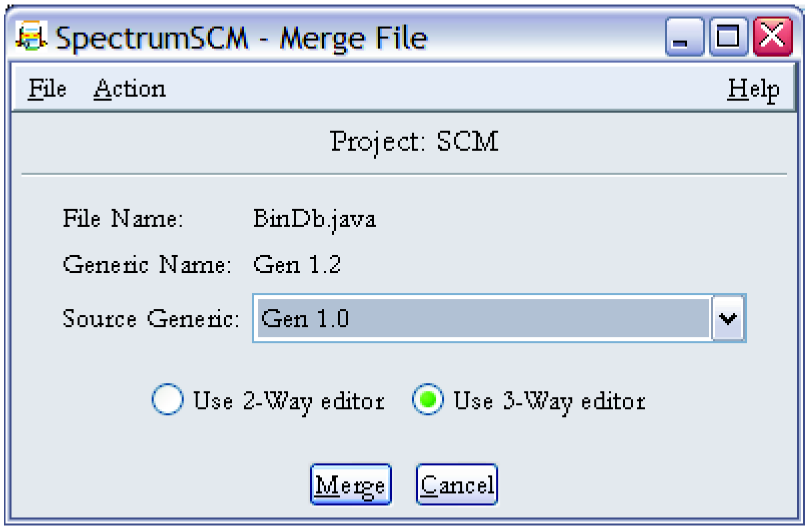


File: BinDb.java Version: 2.1 Generic(s): Gen 1.2 CR: SCM000005

```
package scm.dbaccess;

//
// ODI
//

import com.odi.Database;
import java.io.*;
import scm.persistent.*;
import scm.utilities.*;
```

Line: 8/334 READ-ONLY --Active--

The same developer has changed the same file in generic Gen 1.0. He would like to merge the changes between the two generics. To do so, he selects the file BinDb.java from the later generic (1.2) and checks it out to the desktop to merge options using **Extract / Checkout to desktop / Merge** menu options. A confirmation screen is shown. Click Merge to continue.

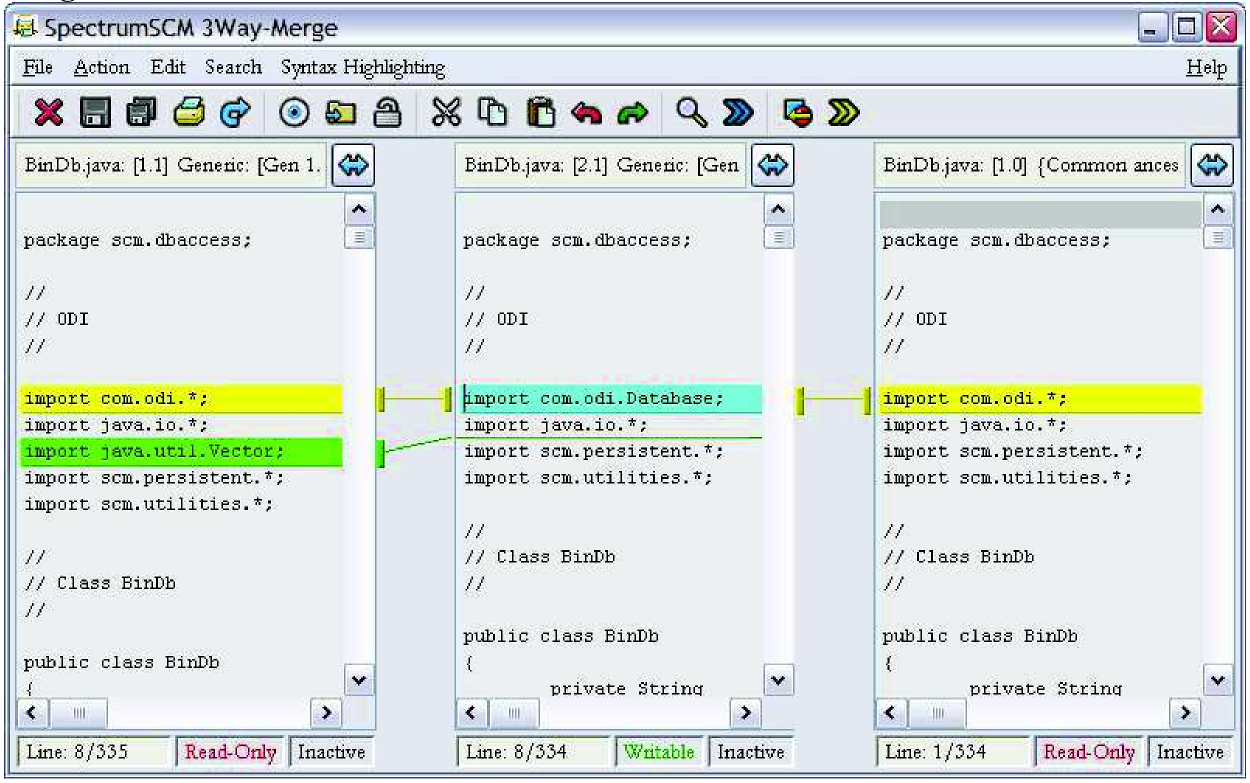


Project: SCM

File Name: BinDb.java
Generic Name: Gen 1.2
Source Generic: Gen 1.0

Use 2-Way editor [✓ Use 3-Way editor]

The choice of merge editors will default to the merge editor type selected by the user in the user's preferences options. Once the editor has been started both versions of the file are displayed in the **Merge Editor**.



BinDb.java: [1.1] Generic: [Gen 1.

```
package scm.dbaccess;

//
// ODI
//

import com.odi.*;
import java.io.*;
import java.util.Vector;
import scm.persistent.*;
import scm.utilities.*;

//
// Class BinDb
//

public class BinDb
{
```

Line: 8/335 Read-Only Inactive

BinDb.java: [2.1] Generic: [Gen

```
package scm.dbaccess;

//
// ODI
//

import com.odi.Database;
import java.io.*;
import scm.persistent.*;
import scm.utilities.*;

//
// Class BinDb
//

public class BinDb
{
        private String
```

Line: 8/334 Writable Inactive

BinDb.java: [1.0] {Common ances

```
package scm.dbaccess;

//
// ODI
//

import com.odi.*;
import java.io.*;
import scm.persistent.*;
import scm.utilities.*;

//
// Class BinDb
//

public class BinDb
{
        private String
```

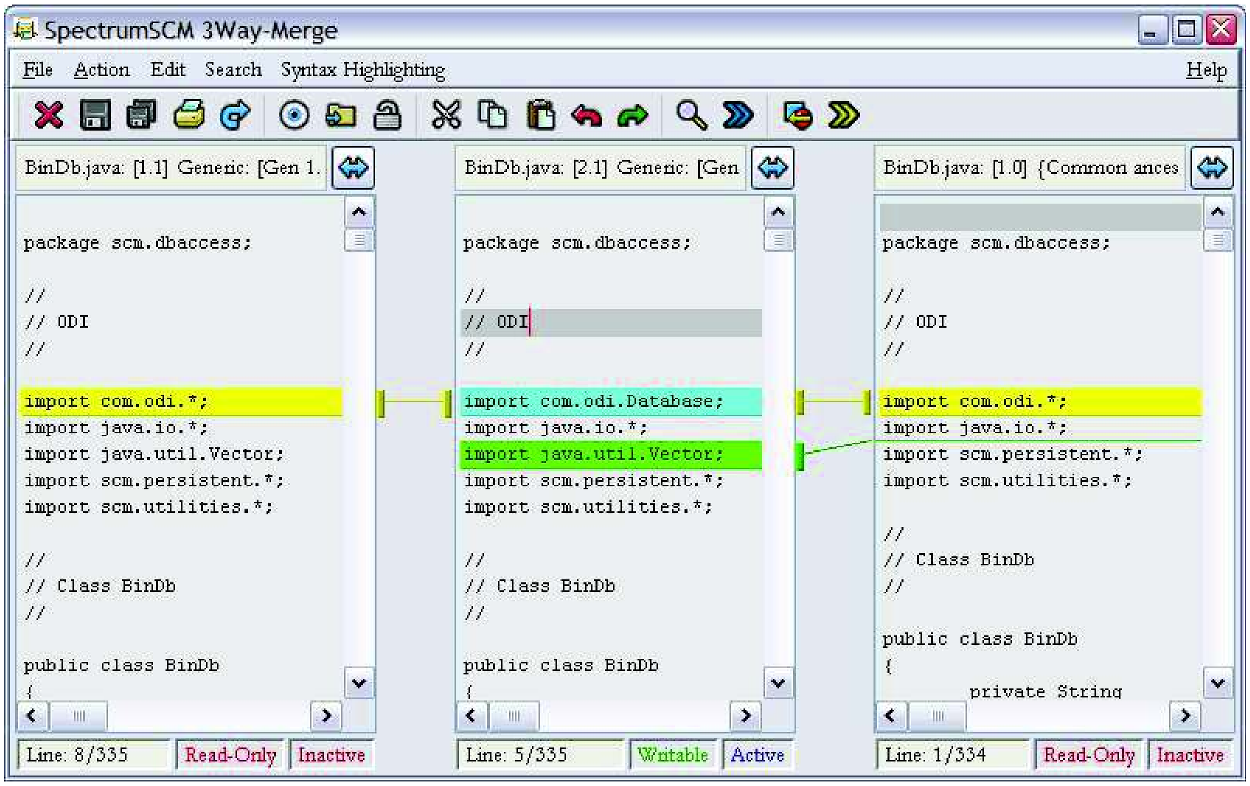
Line: 1/334 Read-Only Inactive

In the three way diff/merge tool (displayed) all diffs are done left to right relative to the common ancestor file on the far right hand side. That is, the branched file on the far left is compared relative to the second branch file in middle pane and then finally the second branch file is compared to the common ancestor on the far right. The user can choose to swap the position of the two branch files so that the comparison can be done against the first branch file and the common ancestor.

In the 2-way diff/merge tool, the user can run the diff analysis in either direction, which will have an affect on how the actual difference is displayed to the user. For example an insert in one file will be a delete in the other or vice-versa, depending on which "direction" the user chooses to run the diff.

The Merge Editor(s) highlight the differences between two versions of a file and the common ancestor (3-way). Inserts show up in green, changes in yellow and deletes in red.

In this case the first branch file has added a single line relative to the second branch file and the second branch file has a single line difference (conflict) with the first branch file. The insert from the first file to the second is highlighted with a green bar and the conflict between the two editors is highlighted with a yellow change bar. Note that the change line in the second branch file is actually color coded cyan. This is an indication that this change is an actual conflict between the changes relative to the common ancestor and must be resolved by the user manually.

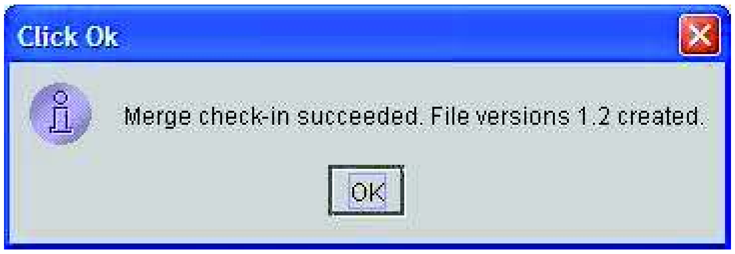

SpectrumSCM 3Way-Merge
File Action Edit Search Syntax Highlighting Help
BinDb.java: [1.1] Generic: [Gen 1.
BinDb.java: [2.1] Generic: [Gen
BinDb.java: [1.0] {Common ances
Line: 8/335 Read-Only Inactive
Line: 5/335 Writable Active
Line: 1/334 Read-Only Inactive

```
package scm.dbaccess;

//
// ODI
//

import com.odi.*;
import java.io.*;
import java.util.Vector;
import scm.persistent.*;
import scm.utilities.*;

//
// Class BinDb
//

public class BinDb
{
```

```
package scm.dbaccess;

//
// ODI
//

import com.odi.Database;
import java.io.*;
import java.util.Vector;
import scm.persistent.*;
import scm.utilities.*;

//
// Class BinDb
//

public class BinDb
{
```

```
package scm.dbaccess;

//
// ODI
//

import com.odi.*;
import java.io.*;
import scm.persistent.*;
import scm.utilities.*;

//
// Class BinDb
//

public class BinDb
{
        private String
```

In this case the developer has resolved the change by selecting the change bar in the first branch file and applying that change to the second branch file with a mouse click. He then selects **Check-in** button and since only the file from gen 1.2 was changed, only one file was checked in.



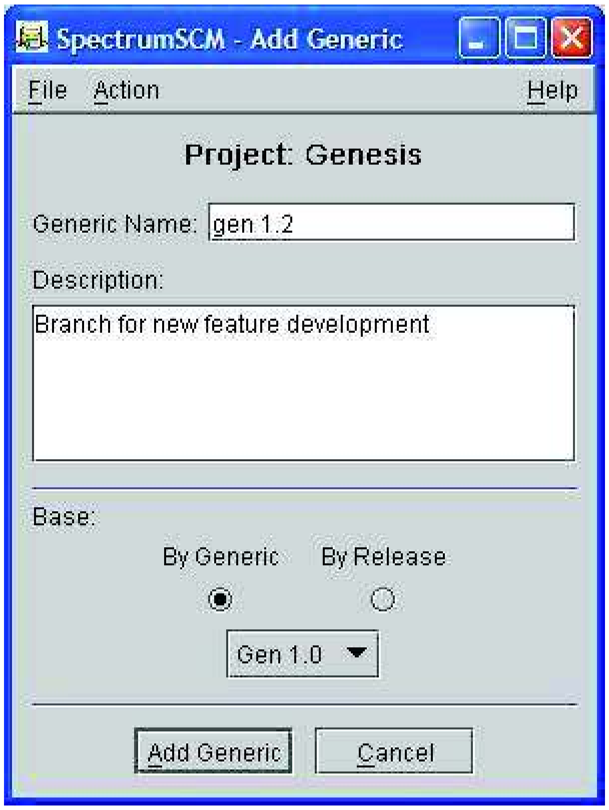
Merge check-in succeeded. File versions 1.2 created.

Generic 1.0 still has its version 1.1 file unchanged.

Recommon

Recommoning brings two versions of the same file (in two different generics) back into one version shared between the two generics (makes them common again). This is useful when a parallel development effort on a project is brought back together to create one code path.

Recommon requires that the later generic has been set up with a previous generic as a basis and that commonality to the previous generic is allowed. In this example, Genesis generic gen1.2 is created to develop a new feature. It is based on Gen 1.0.



Project: Genesis

Generic Name: gen 1.2

Description:

Branch for new feature development

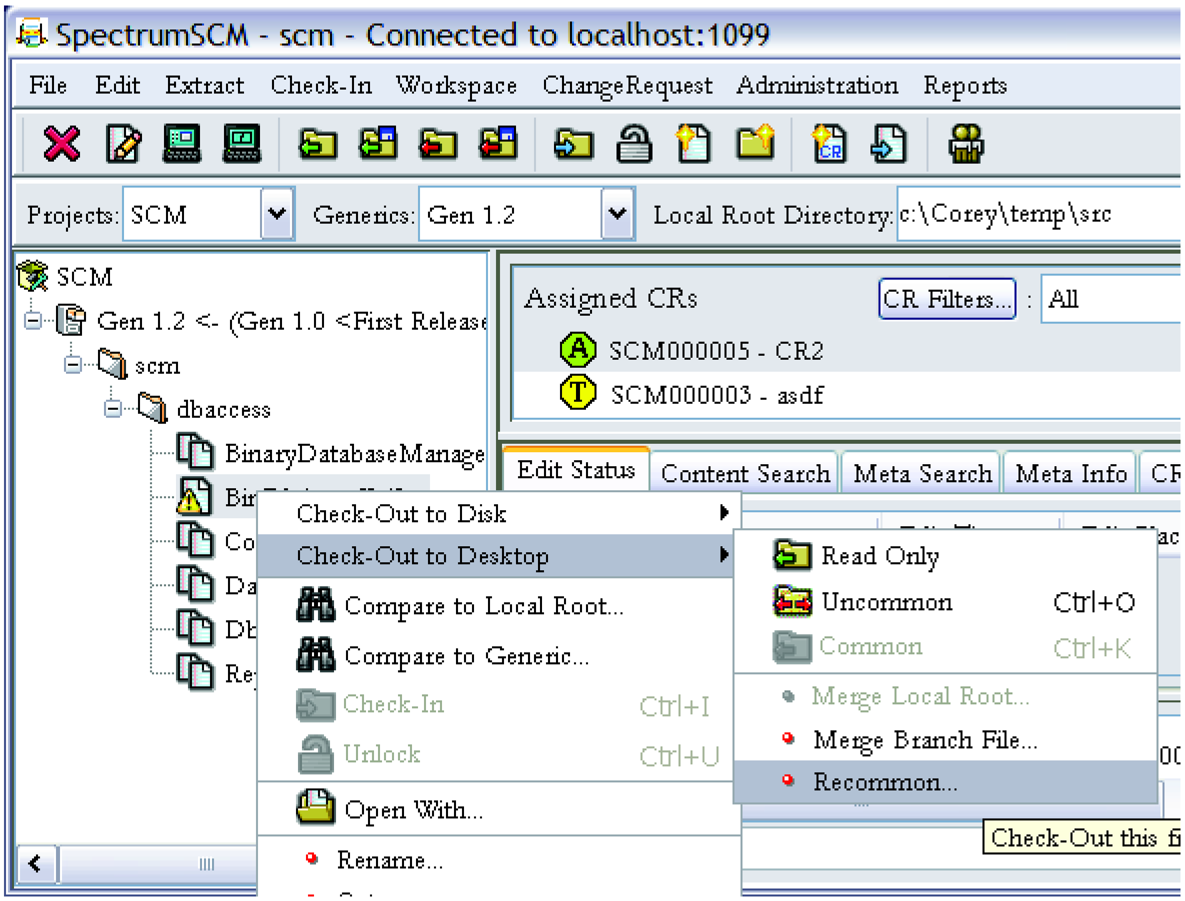
Base:

By Generic [✓] By Release

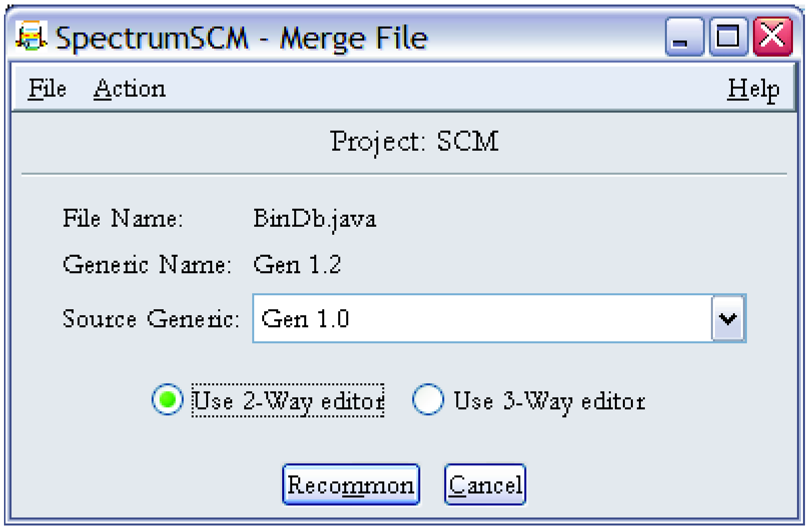
Gen 1.0

Recommoning is the act of bringing the divergent files back together into the same physical disk image. For example, recommoning could occur when part of the development team is charged with developing advanced features for the editor in parallel with the current work. The new work could be done in a separate generic on un-commoned files. Once the work has been completed and tested, the new features for the editor can be rolled into (recommoned with) the main development code stream.

Continuing from the previous merge example, the user has chosen to re-common the file BinDb.java in generic Gen1.2 with the same file in Gen1.0.


SpectrumSCM - scm - Connected to localhost:1099
File
Edit
Extract
Check-In
Workspace
ChangeRequest
Administration
Reports
Projects: SCM
Generics: Gen 1.2
Local Root Directory: c:\Corey\temp\src
SCM
Gen 1.2 <- (Gen 1.0 <First Release
scm
dbaccess
BinaryDatabaseManage
Assigned CRs
CR Filters...
All
SCM000005 - CR2
SCM000003 - asdf
Edit Status
Content Search
Meta Search
Meta Info
Check-Out to Disk
Check-Out to Desktop
Compare to Local Root...
Compare to Generic...
Check-In Ctrl+I
Unlock Ctrl+U
Open With...
Rename...
Read Only
Uncommon Ctrl+O
Common Ctrl+K
Merge Local Root...
Merge Branch File...
Recommon...
Check-Out this fi

To start this activity, from the SpectrumSCM main screen use the context sensitive menu on the tree view and select **Check-Out to Desktop->Recommon….** SpectrumSCM displays the source generic (gen 1.2) and the base generic (Gen 1.0). When the re-common action has been completed, the file from these two generics will be recommoned.



Project: SCM

File Name: BinDb.java

Generic Name: Gen 1.2

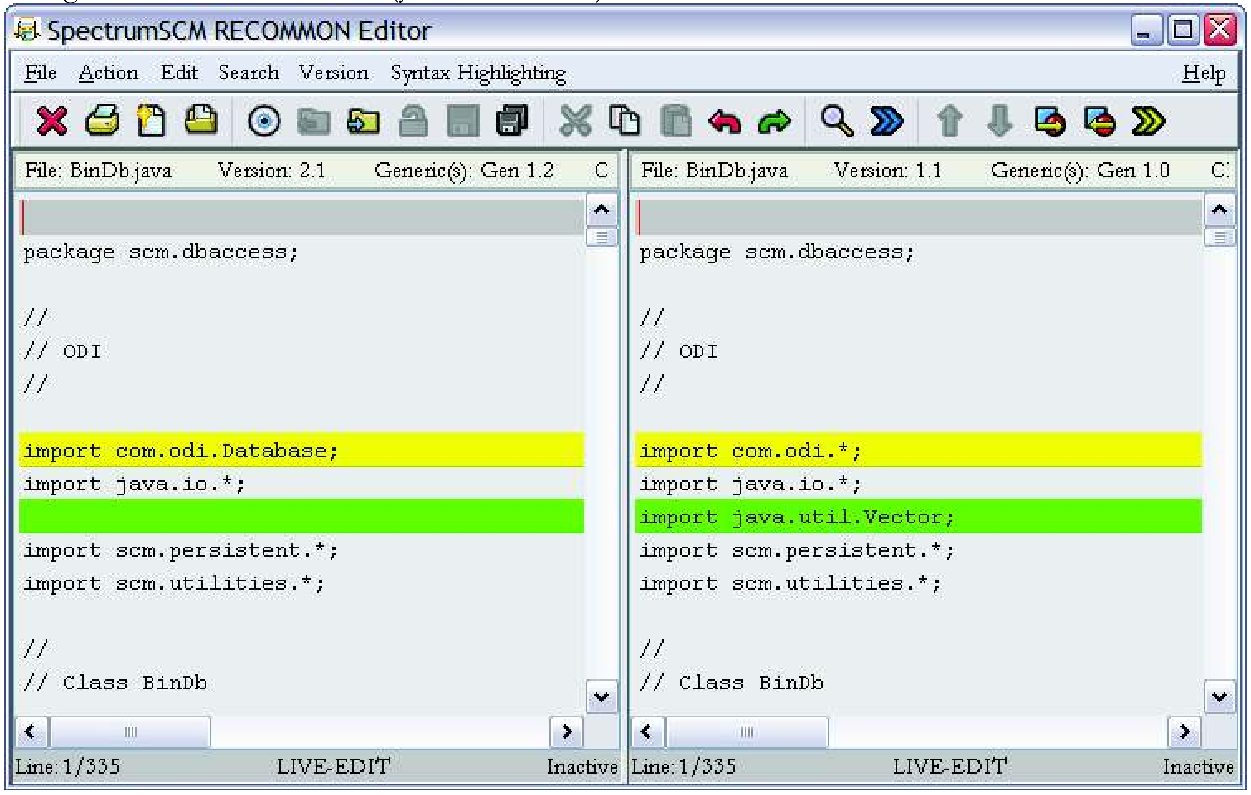
Source Generic: Gen 1.0

[✓ Use 2-Way editor] Use 3-Way editor

In this case the user has chosen to use the 2-way diff/merge tool to perform the re-commoning operation. The BinDb.java file from Gen 1.0 and Gen 1.2 are brought up in the Merge Editor.

The Merge Editor highlights the differences between two versions of a file.

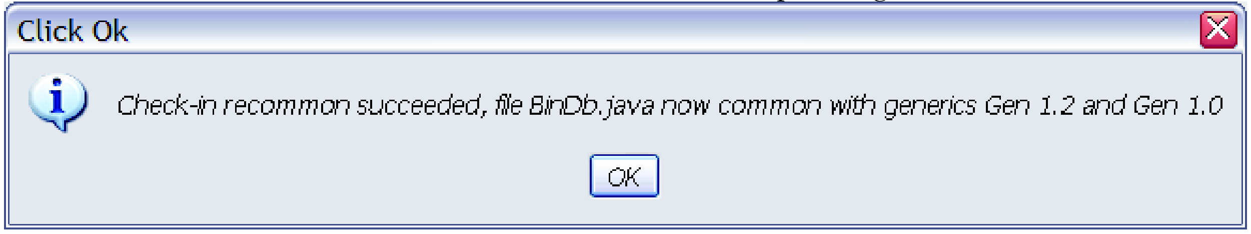
Use the buttons to base the difference from either right to left or left to right. Inserts show up in green, changes in yellow and deletes in red. Depending on which direction you base the differences, an insert (green) one way will be a delete (red) in the other.
In this case, a single line was added to the file in Gen1.0 (green color bar) and a single line was changed in the file on Gen 1.2 (yellow color bar).


SpectrumSCM RECOMMON Editor
File Action Edit Search Version Syntax Highlighting Help
File: BinDb.java Version: 2.1 Generic(s): Gen 1.2
File: BinDb.java Version: 1.1 Generic(s): Gen 1.0
Line: 1/335 LIVE-EDIT Inactive
Line: 1/335 LIVE-EDIT Inactive

```
package scm.dbaccess;

//
// ODI
//

import com.odi.Database;
import java.io.*;

import scm.persistent.*;
import scm.utilities.*;

//
// Class BinDb
```

```
package scm.dbaccess;

//
// ODI
//

import com.odi.*;
import java.io.*;
import java.util.Vector;
import scm.persistent.*;
import scm.utilities.*;

//
// Class BinDb
```

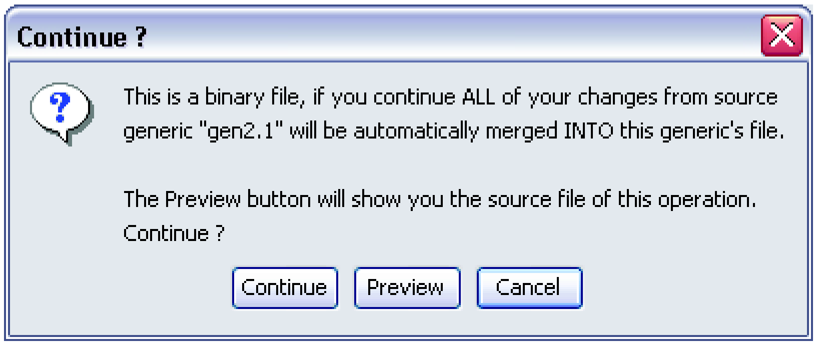
Use the mouse to select each change and then apply the change in the direction that the diff was executed. Use the buttons to continue to check for differences in both directions. Then use the **Check-in** function to recommon them into both releases. When doing a recommon, the contents of both files in the editor must match at the time that check-in is selected. The editor will tell you if this is not the case and check-in recommon will be blocked until the files are identical.
When the files are identical, both will be checked into their respective generics.



Check-in recommon succeeded, file BinDb.java now common with generics Gen 1.2 and Gen 1.0

Merging and Recommoning binary files

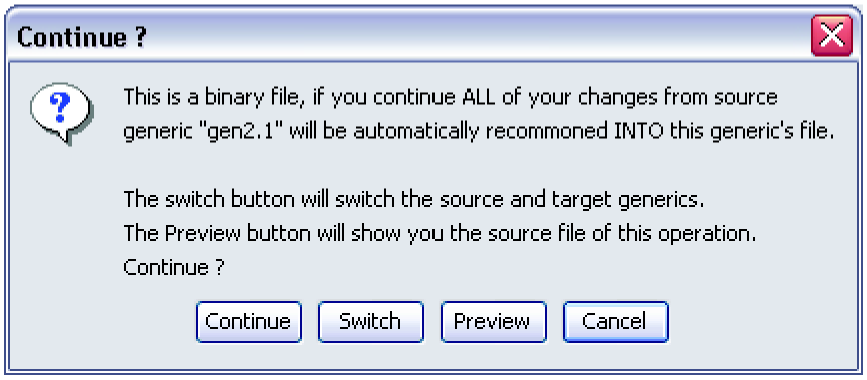
Since the SpectrumSCM dual editor can only be used with text files, the merge/recommon operations for binary files proceed differently.
With a merge operation you will be presented with the following popup –


Continue ?

This is a binary file, if you continue ALL of your changes from source generic "gen2.1" will be automatically merged INTO this generic's file.

The Preview button will show you the source file of this operation.
Continue ?

The source generic will be the one you selected at the start of the merge operation. Selecting the **Preview** button will open your custom editor on the version of the file under the source generic. *See Chapter-5 for details on how to define custom editors.* This will be the version that will be merged into the current file in the generic where you are performing the edit.
The **Continue** button will proceed and automatically perform the merge, overlaying the current version of the file with the contents from the other generic. Since all the changes are versioned, you can still step backwards through the version history if necessary.
With a recommon operation, the popup is slightly different –


Continue ?

This is a binary file, if you continue ALL of your changes from source generic "gen2.1" will be automatically recommoned INTO this generic's file.

The switch button will switch the source and target generics.
The Preview button will show you the source file of this operation.
Continue ?

Specifically, because the edit is occurring on both generics, there is a "**Switch**" button which allows you to choose which file should be the source of the recommon operation. Use the **Preview** button as before to see the version of the file that will become the head revision.
Once the **Continue** button is pressed, the recommon operation will complete automatically.

11.4 Using Branching Patterns for Configuration Management*

*Based on *Advanced Branching Techniques for SpectrumSCM, a White Paper written by William C. Brown, 05/14/2002*

The technical execution of creating a generic, creating, merging and recommoning files is simple compared to the task of determining how best to create branches to support the specific needs of the system development effort. Over the years, developers and system engineers have developed many unique branching techniques to solve difficult configuration management problems. The purpose of this section 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 more 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.

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. The use of a CM system should not become a burden by adding to the workload 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 and maintain after years of activity.

SpectrumSCM 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.

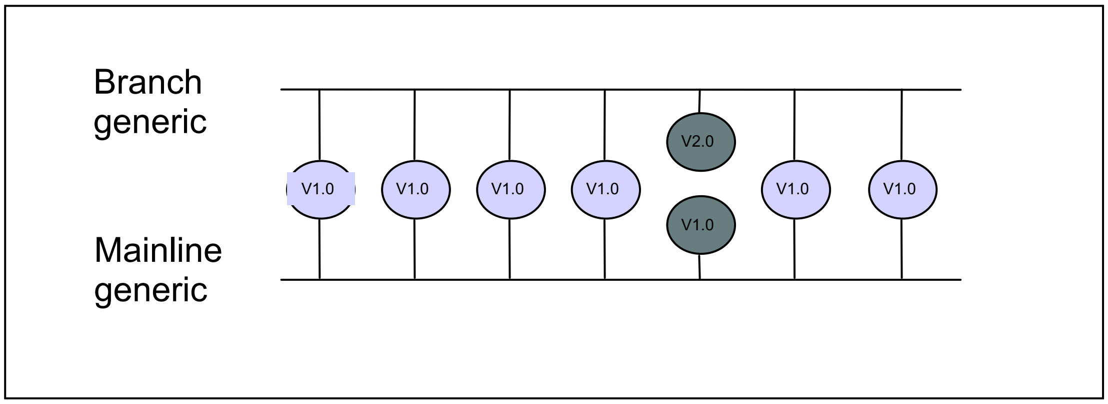
- **The Classic branching design pattern**
- **Parallel Development pattern**
- **The Sandbox pattern**
- **The Promotion (Repository) Pattern**
- **The Patch Pattern**

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. The classic pattern allows individual developers to individually create alternate branches of code extended from the mainline development stream. Without a strong CM system such as SpectrumSCM, the existence of such a branch is not immediately obvious to the other users of the system.

Traditionally, 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 embedded in the version number of the file, for example in systems based on RCS (Revision Control System) where the third digit in the file version number is greater than "0" (file version 1.3.1.2 might mean that a branch for this particular file was formed at version 1.3 and is now at version 1.2 of the new branch). Creating a release with the proper file versions can be difficult at best.

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:



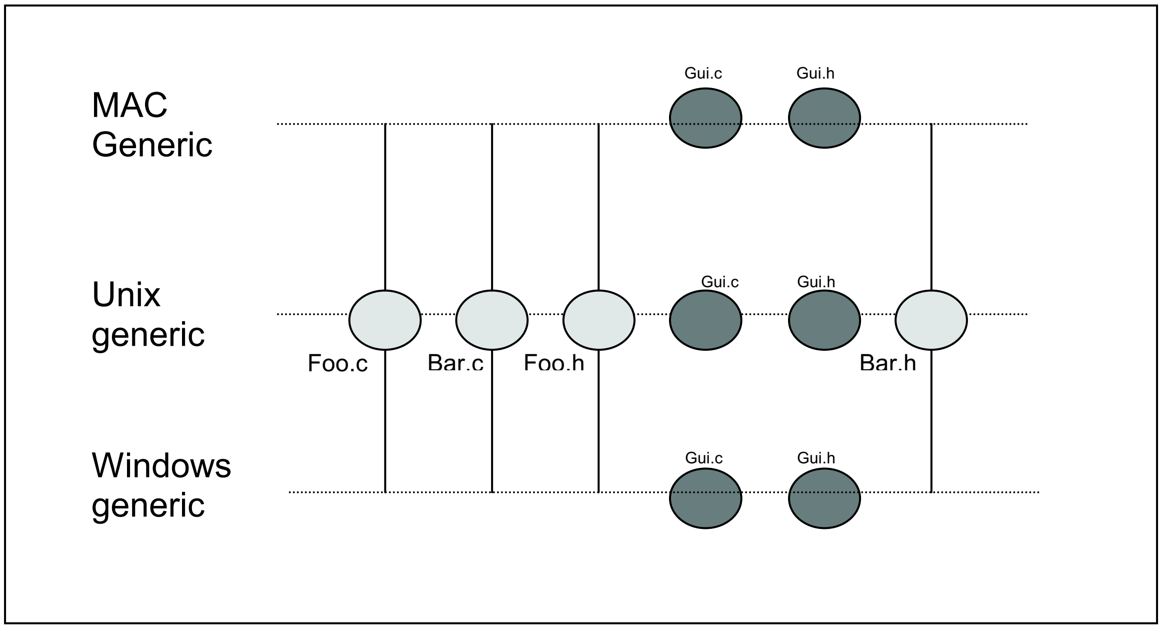
In this example, the dark circles represent a single uncommoned file. Each branch contains a separate physical instance of this file and shared instances of all the other files. When the files are ***recommoned***, both branches will again share a single file instance.

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.

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.

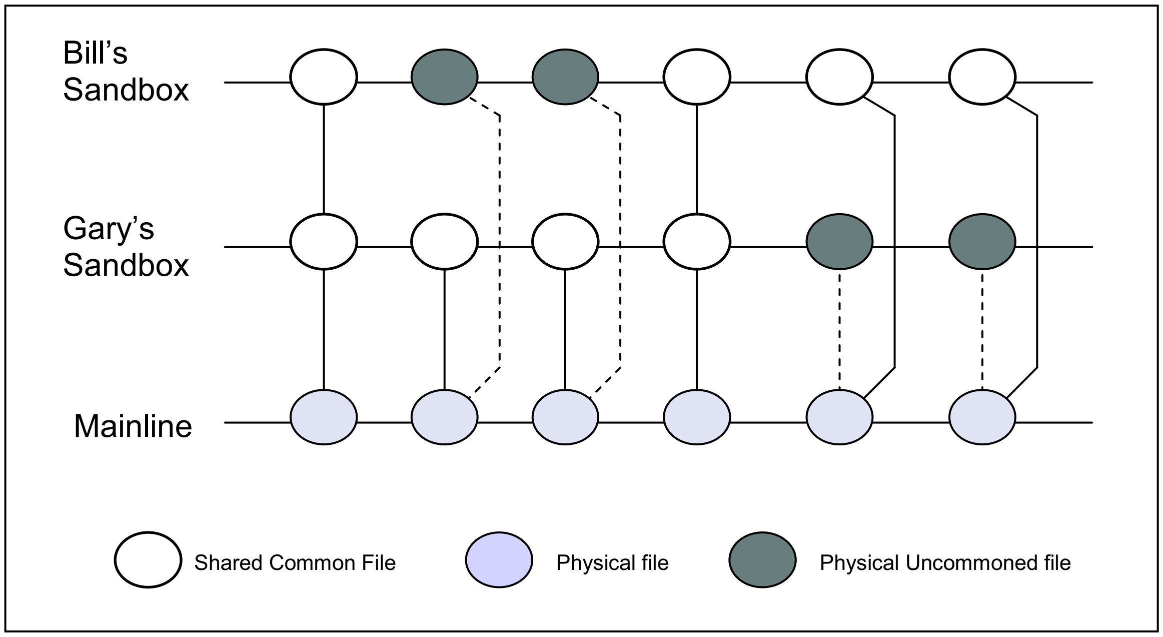
Parallel Development Pattern
The parallel development pattern is very similar to the classic pattern in that two or more branches are created, but in the parallel pattern, some files are never merged or recommoned back with the mainline. The parallel development pattern might be used during the development 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.*

The Sandbox Pattern

In the Sandbox 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 that 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 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.

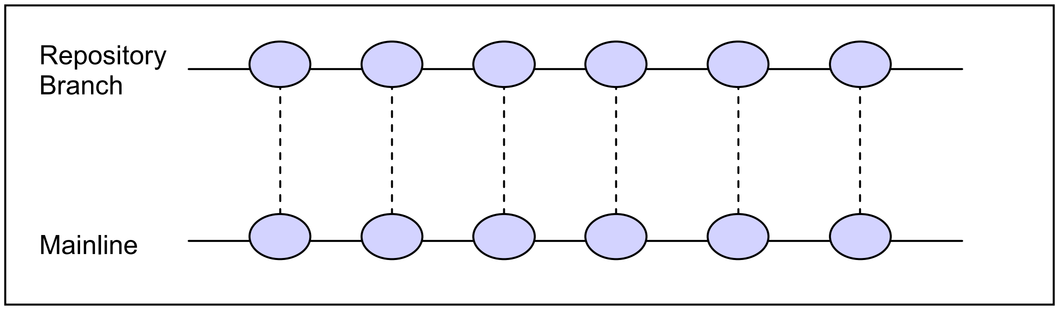
SpectrumSCM may be the only CM system that properly supports this pattern.

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.

The Promotion (Repository) Pattern

The Promotion (Repository) 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:

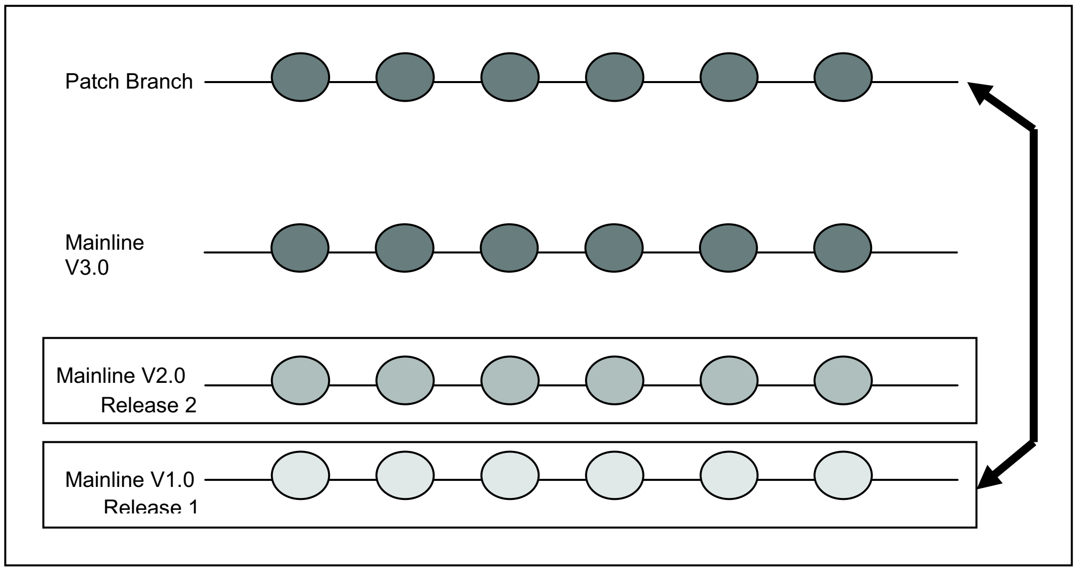


SpectrumSCM easily supports this pattern. After the repository branch is created, all files are checked out uncommon from the mainline. SpectrumSCM allows the project leaders to enforce this behavior by locking the repository generic. Locking guarantees that all files checked out or into the mainline 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.

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.

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.
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. Creating a new generic that is rooted in a previous release of the system does this. 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.
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.

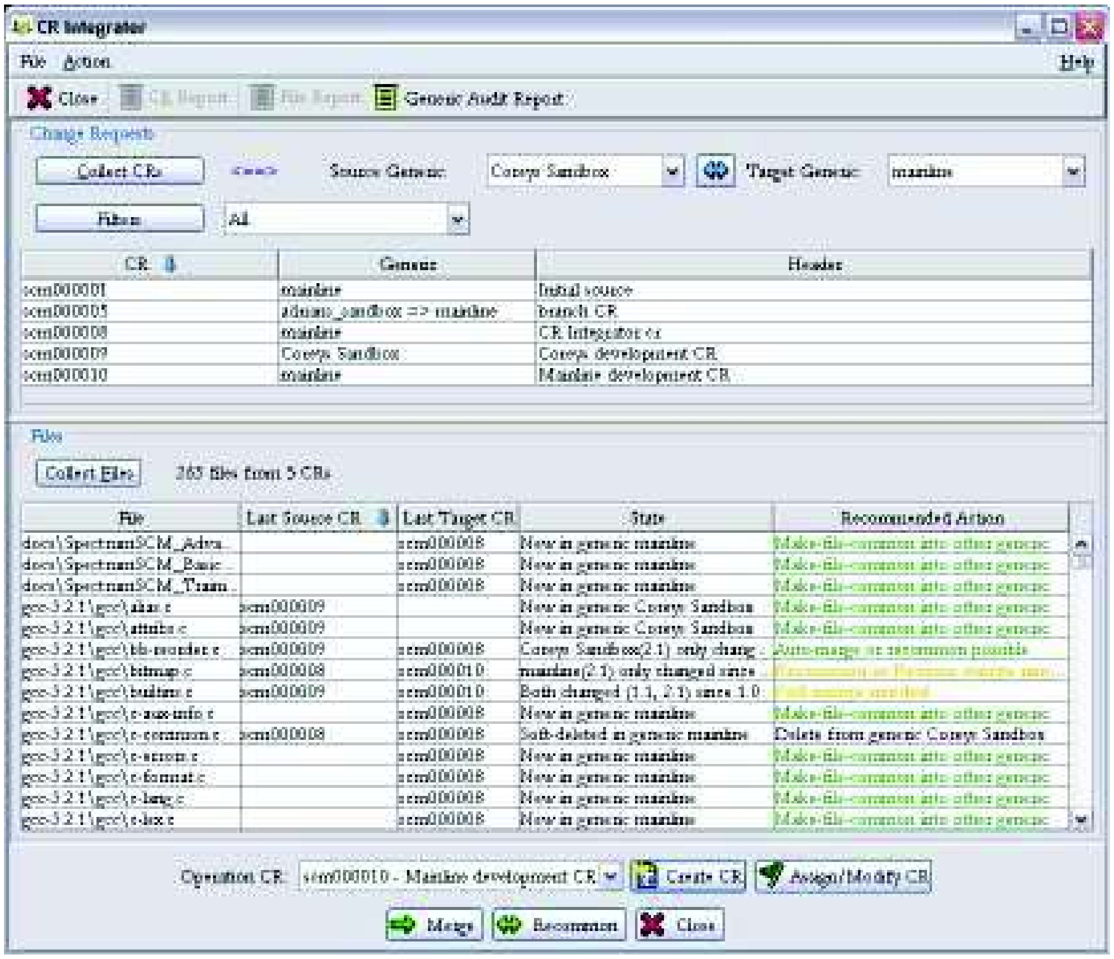
Conclusion

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.

11.5 CR Integrator

The CR Integrator is an interactive screen that can be used to compare, merge and recommon 2 generics or branches. The integrator is designed to help and automate (where possible) the process of merging or recommoning changes from one generic to another. This action frequently occurs as part of a number of branching patterns (e.g. the sandbox pattern). Under the sandbox pattern a developer (or team) perform their work in a sandbox generic, isolated from the main production line. Then when that work is complete they merge the work into the mainline ready for the next production release.

Some work-items can be automatically completed (seen in **green** in the screen-shot below) leading to huge productivity gains. For other items that require some scrutiny (seen in yellow), the user will be walked through the necessary steps. This in turn, not only gives you some productivity gains but also ensures the stability of your product.



Source Generic: Coreys Sandbox
Target Generic: mainline
Filters: All

CR	Generic	Header
scm000001	mainline	Initial source
scm000005	adams_sandbox => mainline	branch CR
scm000008	mainline	CR Integrator cr
scm000009	Coreys Sandbox	Coreys development CR
scm000010	mainline	Mainline development CR

263 files from 5 CRs

File	Last Source CR	Last Target CR	State	Recommended Action
docs\SpectrumSCM_Adva...		scm000008	New in generic mainline	Make-file-common into other generic
docs\SpectrumSCM_Basic...		scm000008	New in generic mainline	Make-file-common into other generic
docs\SpectrumSCM_Train...		scm000008	New in generic mainline	Make-file-common into other generic
gcc-3.2.1\gcc\alias.c	scm000009		New in generic Coreys Sandbox	Make-file-common into other generic
gcc-3.2.1\gcc\attribs.c	scm000009		New in generic Coreys Sandbox	Make-file-common into other generic
gcc-3.2.1\gcc\bb-reorder.c	scm000009	scm000008	Coreys Sandbox(2.1) only chang...	Auto-merge or recommon possible
gcc-3.2.1\gcc\bitmap.c	scm000008	scm000010	mainline(2.1) only changed since...	[illegible]
gcc-3.2.1\gcc\builtins.c	scm000009	scm000010	Both changed (1.1, 2.1) since 1.0	Full merge needed
gcc-3.2.1\gcc\c-aux-info.c		scm000008	New in generic mainline	Make-file-common into other generic
gcc-3.2.1\gcc\c-common.c	scm000008	scm000008	Soft-deleted in generic mainline	Delete from generic Coreys Sandbox
gcc-3.2.1\gcc\c-errors.c		scm000008	New in generic mainline	Make-file-common into other generic
gcc-3.2.1\gcc\c-format.c		scm000008	New in generic mainline	Make-file-common into other generic
gcc-3.2.1\gcc\c-lang.c		scm000008	New in generic mainline	Make-file-common into other generic
gcc-3.2.1\gcc\c-lex.c		scm000008	New in generic mainline	Make-file-common into other generic

Operation CR: scm000010 - Mainline development CR

11.5.1 Screen Flow

Select Generics - The first thing you want to do, is select your ***Source Generic*** and then your ***Target Generic***. Only generics that are related and can therefore be merged will be presented. The target generic is where the merge operation work is going to be performed.

For example: if you are merging your work from your sandbox back into the mainline, then the source generic will be your sandbox and the target will be the mainline.

Note: The ⇔ button can be used to swap the source and target generics.

Collect CRs - Once the source and target generics have been selected, the *Collect CRs* button should be pressed. This performs the generic audit to identify which CRs have caused files to be different between the 2 generics.

A double click on a CR entry will trigger the Change Request Report. This is also available off of the toolbar and Action menu.

Collect Files - Once the CRs list is populated, you can filter them if desired OR you can just present the file differences by pressing the *Collect Files* button. The file differences show which files that were edited by the selected CRs, are currently different between the source and target generics.

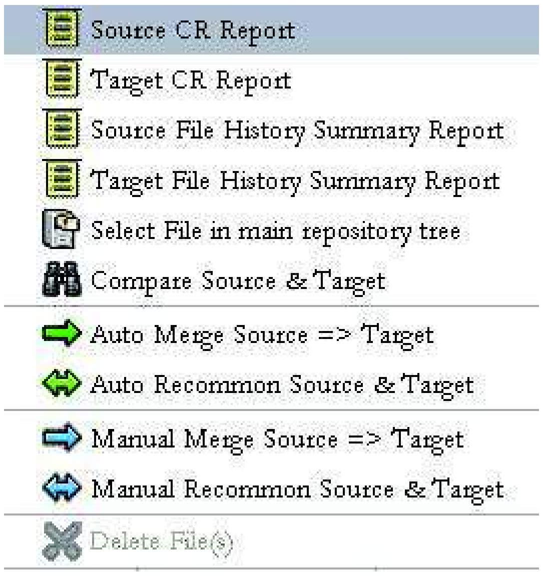
The Collected Files table presents the following information -

- The file.
- The last source CR - which CR was last used to edit this file in the **source** generic.
- The last target CR - which CR was last used to edit this file in the **target** generic.
- State - A summary line as to the state of the difference between the 2 generics for this file.
- Action - A recommendation as to which actions would generally make sense under this state.

At this point you can review the differences, or you can work the differences either individually or in bulk.

File lines shown in green are able to be automatically resolved.

File lines in yellow need manual assistance.


Source CR Report
Target CR Report
Source File History Summary Report
Target File History Summary Report
Select File in main repository tree
Compare Source & Target
Auto Merge Source => Target
Auto Recommon Source & Target
Manual Merge Source => Target
Manual Recommon Source & Target
Delete File(s)

There are also a number of options available on the right-mouse button, these operations are performed relative to the selected lines. Double-clicking on a file line will run the **File History Generic Comparison** report, which shows how the file versions have progressed in the source and target generics, version by version.

11.5.2 Running an Auto-Merge

Select the CR for this operation - If you perform an edit operation, it needs to be performed relative to a change request that is *assigned to you*. If this is a merge operation the CR needs to be assigned under the target generic. If this is a recommon operation, since both generics are affected, the CR can come from either the source or the target generic.

The **Operation CR** choice box allows you to select this CR. Target generic CRs are shown in **black** and can therefore be used for either merge or recommon operations. Source generic CRs are shown in blue and can therefore only be used for recommoning. When a specific CR is chosen, then that will be used for all the operations until it is changed.

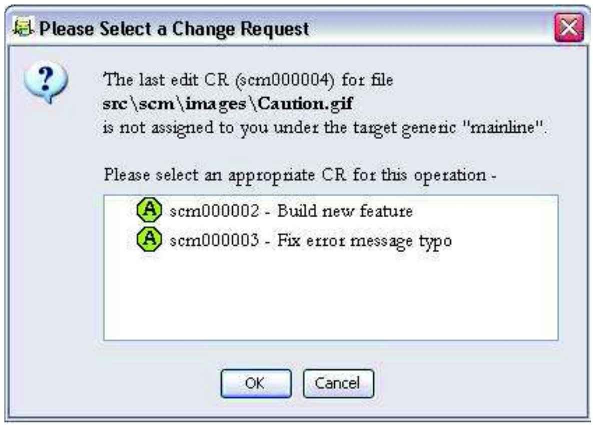
NOTE - Because merge operations have to be completed against a CR in the target generic, only **black** CRs will enable the merge button.

There are also a couple of special values ***Last Target CR*** and ***Last Source CR***.

If the *"Last Target CR"* option is chosen, the merge or recommon operation will be attempted against the CR shown in the *Last Target CR* column for each file. This is useful if you are merging the work of multiple CRs and you want the merge operation to be recorded against each of those CRs respectively.

Similarly, the *"Last Source CR"* will attempt to perform the requested operation against the CR shown in the *Last Source CR* column.

If there is no "Last ... CR", or it is no longer assigned to you, you will be presented with a warning and requested to select a valid CR.



The last edit CR (scm000004) for file
src\scm\images\Caution.gif
is not assigned to you under the target generic "mainline".

Please select an appropriate CR for this operation -

- scm000002 - Build new feature
- scm000003 - Fix error message typo

If no file lines are selected and either of the merge or recommon buttons at the bottom of the screen are pressed then ALL lines will attempt to be processed. If one or more lines are selected then only those lines will be processed. You can select on a column header of the file table to sort by those values. So, for example, if you want to sort the actions into automatable and manual tasks you can do so.

Manual Merge and Manual Recommon operations are the same as if they were executed from the main screen file tree/menu structure.

Auto-Merge will update the target generic with the source file version contents but leave the files as separate instances. Auto-recommon literally makes the two generics point to the same file (with the source file contents). Recommoning enables future common edits.

If an automatic operation is selected upon a number of file lines, only those lines that are auto-capable will be automatically completed. The dialog will walk you through performing the manual tasks as appropriate.

When file lines are processed the **Recommended Action** box is set as such. If you want to reprocess a particular line, simply select it and select the required operation. A popup dialog will confirm the reprocessing action.

11.6 Generic/Branching Reports

There are 2 specific reports useful for managing multiple generic/branch situations. The "Generic Audit Report" compares the 2 selected generics and reports on the differences (uncommon or new files). The "File History Generic Comparison Report" reports on a particular files history with respect to its branching, commonality or uncommon situations.

The example below shows how the file was introduced common under CR 10. The file was then edited common under CR 11 before being edited uncommon into the "Branch" generic by CR 12. Work under CR 12 continued in the "Branch" until it was recommoned back into the mainline with version 2.2.

File History Generic Comparison Report

Project :	scm
Source Generic :	Branch
Compare Generic :	Mainline
Filename :	alias.c
Filepath :	gcc-3.2.1\gcc

File type is: TEXT

Version History

Branch Version Number	**Edit CR**	**Edit Date**	**Common**	**Mainline Version Number**	**Edit CR**	**Edit Date**
2.2	scm000012	2008/01/08 16:18:41	Common	2.2	scm000012	2008/01/08 16:18:41
2.1	scm000012	2008/01/08 16:18:05				
2.0	scm000012	2008/01/08 16:17:54				
1.1	scm000011	2008/01/08 16:17:48	Common	1.1	scm000011	2008/01/08 16:17:48
1.0	scm000010	2008/01/04 12:01:19	Common	1.0	scm000010	2008/01/04 12:01:19