

**AccuRev<sup>®</sup>**

## **Technical Notes**

**Version 5.5  
June 2012**

*Revised 6-June-2012*

# Copyright

Copyright © AccuRev, Inc. 1995–2012

ALL RIGHTS RESERVED

This product incorporates technology that may be covered by one or more of the following patents:  
U.S. Patent Numbers: 7,437,722; 7,614,038.

**TimeSafe** and **AccuRev** are registered trademarks of AccuRev, Inc.

**AccuBridge**, **AccuReplica**, **AccuWork**, **Kando**, and **StreamBrowser** are trademarks of AccuRev, Inc.

All other trade names, trademarks, and service marks used in this document are the property of their respective owners.

# Table of Contents

<b>Converting to AccuRev from</b>	
<b>Directories Containing Baselevels .....</b>	<b>1</b>
Creating a Depot .....	1
Processing the First Baselevel .....	1
Recording the Baselevel with a Snapshot.....	2
Processing Subsequent Baselevels.....	2
Handling Additional Baselevel-to-Baselevel Differences .....	3
Cleaning Up .....	4
<b>Creating and Using a Maintenance Stream .....</b>	<b>5</b>
Creating a Snapshot .....	5
Creating a Stream Based on the Snapshot .....	5
Performing Maintenance Work.....	5
<b>Pathname Optimization:</b>	
<b>ACCUREV_IGNORE_ELEMS and .acignore .....</b>	<b>7</b>
Eligible “Whole-Workspace” Commands .....	8
GUI Counterparts to the “Whole-Workspace” Commands .....	8
Commands that Don’t Apply to the Whole Workspace.....	8
Values for ACCUREV_IGNORE_ELEMS.....	9
Examples.....	10
Specifying Directories and Their Contents.....	10
Setting ACCUREV_IGNORE_ELEMS on a UNIX/Linux System.....	11
Setting ACCUREV_IGNORE_ELEMS on a Windows System .....	11
Per-Directory Pathname Optimization — the .acignore File.....	11
<b>Techniques for Sharing Workspaces .....</b>	<b>13</b>
Accessing a Windows Workspace From Multiple Windows Clients .....	13
Universal Access to a Workspace Located on a Share .....	14
The ‘share_map.txt’ File.....	14
Workspace Location Entries .....	15
Fixing Workspace Location Entries .....	15
Fixing Machine Name Entries .....	16
Example: Samba Share .....	16
<b>What’s the Difference between Populate and Update? .....</b>	<b>17</b>
In a Nutshell .....	17
Example 1: Standard Update Scenario .....	18
Example 2: Restoring a Deleted File (“missing” by accident) .....	18
Example 3: Handling Active Elements.....	18
Example 4: A Tale of Two Files.....	19

Data Structures Used by Populate and Update .....	19
How the Data Structures Get Their Data .....	20
Backing Stream .....	21
Workspace Stream .....	22
Workspace Tree .....	24
The Update Algorithm .....	24
Advancing the Scan Threshold .....	26
Incomplete Updates .....	26
Incomplete Update: Command Interrupted .....	27
Incomplete Update: Checksum Failure .....	27
Performing the “Fixup” Update .....	27
<b>Using a Trigger to Maintain a Reference Tree .....</b>	<b>29</b>
<b>Notes for CVS Users .....</b>	<b>31</b>
AccuRev Workspaces vs. CVS Sandboxes .....	31
Common Operations .....	31
Obtaining a copy of the source files .....	31
Placing files under version control .....	31
Bringing others’ changes into your workspace/sandbox .....	32
Saving your changes .....	32
Finding the history of files .....	32
Finding the status of files in your workspace/sandbox .....	33
Removing files .....	33
Reverting changes to files .....	33
Moving files .....	34
Checking out files to edit .....	34
Comparing versions of files .....	34
<b>Version Control of Namespace-Related Changes .....</b>	<b>35</b>
Twin Elements and Stranded Elements .....	35
Preventing Creation of Twins in Workspaces .....	35
Reporting of Twins in Dynamic Streams .....	36
Ability to Reuse an Element Name after a Rename Operation .....	36
When a Purge Operation Causes an Element to Disappear .....	37
Detection of All Stranded Elements, Including “Twins” .....	37
Ability to Operate on Stranded Elements Using Element-IDs .....	38
More Sophisticated Analysis of Namespace-Related Changes .....	38
Change to Merge Algorithm for Namespace-Related Changes .....	38
Handling Stranded Elements .....	40
Defunct element obscured by element with same name .....	40
Resolving the Situation .....	41
Elements under a defunct parent .....	41
Resolving the Situation .....	41
Elements under an excluded parent .....	41
Resolving the Situation .....	42

Dangling directory elements .....	42
Resolving the Situation .....	42
Elements under a non-existent (purged) parent directory .....	42
Resolving the Situation .....	43
Elements under a stranded parent directory .....	43
Active element refers to a purged version .....	43
<b>Notes on Cross-Links.....</b>	<b>45</b>
Cross-Link Direction and Terminology.....	45
Cross-Links and Stream Namespaces.....	45
Source Stream: Workspace vs. Dynamic Stream .....	47
Multiple Cross-Links: Chaining .....	48
Double Vision: Seeing an Element Multiple Times in a Workspace .....	49
Double Vision and the ‘accurev name’ Command .....	50
Cross-Link Overlaps .....	50
<b>Notes on Promote-by-Issue .....</b>	<b>53</b>
Promote-by-Issue Basics.....	53
Promoting Issues to the Parent Stream .....	55
Cross-Promoting Issues to a Non-Parent Stream — Simple Case.....	56
Cross-Promoting Issues to a Non-Parent Stream — Patch Required .....	57
Changing the Way You Use the Change Palette .....	58
Promotion / Creating a Tracking Issue .....	60
Working with the Tracking Issue.....	61
If You Process Some Elements at the Stream Level, not the Workspace Level .....	63
Fixing Your Mistake .....	64
<b>Incomplete Change Packages .....</b>	<b>66</b>
Overview .....	66
An Example Scenario .....	66
File Purge (Revert to Backed).....	69
Reuse of Issues Across Streams.....	71
Promoting by File Instead of by Issue .....	72
Sample server_preop_trig rules .....	72
How to Troubleshoot Incomplete Change Packages .....	74
<b>Notes on Revert to ... and Diff Against...</b>	
<b>GUI Commands .....</b>	<b>76</b>
Overview.....	76
Diff Against... .....	77
Revert to.....	77
<b>Using Third-Party ITS Keys.....</b>	<b>79</b>
Modifying the schema .....	79

Using the Schema Editor .....	79
Editing schema.xml .....	80
Using Third-Party ITS Keys in the CLI.....	81
Commands That Return Third Party Issue Numbers.....	82
hist example output .....	83
cpkdepend example output .....	83
Using Third-Party Keys in the Java GUI.....	84

# Converting to AccuRev from Directories Containing Baselevels

Many development groups use source code provided by another group within the organization, or from another organization altogether. Typically, the code is imported on a periodic basis as a complete source tree, which we'll call a "baselevel". This note examines a scenario in which source-code baselevels are imported into AccuRev. Suppose each baselevel is stored as a directory tree:

```
D:\baselevels\gizmo1.0
D:\baselevels\gizmo2.0
D:\baselevels\gizmo3.0
```

It is easy to incorporate the multiple baselevels into AccuRev. Make sure you read these instructions all the way through before trying it out.

## Creating a Depot

First, create an AccuRev depot, where AccuRev permanently stores all of the data for a programming project. For example:

```
accurev mkdepot -p gizmo
```

This creates a depot called **gizmo**. It has a single stream, also called **gizmo**.

## Processing the First Baselevel

Next, populate **gizmo** with files from the first baselevel.

1. Go to the first baselevel directory:

```
cd \baselevels\gizmo1.0
```

2. Create a workspace to be used for importing files from the baselevel into AccuRev:

```
accurev mkws -w import -b gizmo -l .
```

Note that the command line ends with "dash-ell dot". This creates a workspace called **import**, which is based on stream **gizmo** (currently empty) in the current location.

3. Get a list of all of the files that AccuRev doesn't know about (which is all of them):

```
accurev stat -x > extfiles.list
```

The **-x** stands for external. View the resulting file. You may see many files that you don't want to put under version control: object files, executables, text-editor backup files, etc.

4. You can have AccuRev ignore such files, by specifying patterns (wildcards) that match their names as the value of an environment variable. (Note that case is important in this pattern-matching). For example:

```
set ACCUREV_IGNORE_ELEMS=*.exe *.obj *.lnk *.err *.map (and so on)
```

(The syntax for setting environment variables varies among operating systems and command-line processors. For more on ACCUREV\_IGNORE\_ELEMS, see *Pathname Optimization: ACCUREV\_IGNORE\_ELEMS and .acignore* on page 7.)

5. Try the preceding two steps again, keep repeating this process until the **stat -x** command lists exactly the set of files that you want AccuRev to keep track of.
6. Create initial versions of these files in the depot:

```
accurev add -c "initial file add" -x
```

This creates versions in the **import** workspace, but has not yet made them available to others working on the **gizmo** project. The **-c** option allows you to associate a comment with the transaction.

7. To make these new files public, promote them to the **gizmo** stream:

```
accurev promote -c "initial file promote" -k
```

## Recording the Baselevel with a Snapshot

At this point, you can create a snapshot of the **gizmo** stream. A snapshot is a special kind of stream, whose contents can never change. (Hence, a snapshot is also called a “static stream”, distinguishing it from a standard dynamic stream.) In this case, the snapshot will contain the versions in the first baselevel, because that’s exactly what the **gizmo** stream contains at the current time.

(The **gizmo** stream itself will change, as you incorporate additional baselevels. But any snapshot you create is guaranteed to be frozen forever!)

Use the **mksnap** command to create the snapshot:

```
accurev mksnap -s gizmo1.0 -b gizmo -t now
```

At any time in the future, you can use snapshot **gizmo1.0** to see the contents of the first baselevel. And if you need to fix a bug that existed at this baselevel, you can create a maintenance stream below the snapshot. See *Creating and Using a Maintenance Stream* on page 5.

Note: AccuRev does not implement snapshots with “version labels”, as do branch-and-label SCM systems. Since there’s no need to attach a label to each version in the baselevel, creating a snapshot is virtually instantaneous!

## Processing Subsequent Baselevels

Now, you need to “layer” the files in the next baselevel on top of the files that you’ve already placed under AccuRev control.

1. Change the definition of the **import** workspace:

```
cd D:\baselevels\gizmo2.0
accurev chws -w import -l .      (again, “dash-ell dot”)
```



In effect, you've moved the workspace to where the files are, instead of moving the files into the workspace! The files fall into several categories.

2. Make sure that all files in this baselevel have timestamps that are later than the timestamps in the preceding baselevel:

```
accurev touch -R .
```

3. Process the files that changed from **gizmo1.0** to **gizmo2.0**.

To AccuRev, these files appear to be modified versions of the **gizmo1.0** files that you *add*'ed and *promote*'d in the preceding section. You can list all these "modified" files:

```
accurev stat -m
```

And you can keep the new versions of the files:

```
accurev keep -m -c "my comment"
```

4. Process the files that didn't change from **gizmo1.0** to **gizmo2.0**.

You don't need to do anything about these files. In particular, you don't need to *keep* new versions.

5. Process the files that didn't exist in **gizmo1.0**, but do exist in **gizmo2.0**.

These files are external, because AccuRev hasn't seen them before. (Just as *all* the files were external when you placed the first baselevel under version control.) Add the external files to the depot, just as you did in the preceding section:

```
accurev add -x
```

As above, you may want to use *stat -x* and the `ACCUREV_IGNORE_ELEMS` environment variable to filter out unwanted files before entering the *add* command.

6. Promote the new files and changed files:

```
accurev promote -k
```

You've now placed two baselevels under AccuRev control. Layering the third baselevel, **gizmo3.0**, on top of the second one is exactly the same as layering the second one on top of the first. Just repeat the steps in this section.

## Handling Additional Baselevel-to-Baselevel Differences

In the discussion above, we broke a baselevel's "new layer" of files into three categories. This was a bit oversimplified — there are additional categories to consider.

- Files that existed in one baselevel, but were deleted in the next baselevel.

You can make such files disappear from the new baselevel by defuncting them:

```
accurev defunct <filenames>
```

- Files that were renamed from one baselevel to the next.

This will appear to be (1) a file that existed in one baselevel, but was deleted from the next baselevel, along with (2) a new file that didn't exist in the preceding baselevel. If you know that file **oldname.c** in the preceding baselevel was renamed to **newname.c** in the next baselevel, use this series of commands to make the connection:

```
accurev pop oldname.c
ren newname.c SAVEME (UNIX/Linux: use the mv command)
accurev move oldname.c newname.c
ren SAVEME newname.c
accurev keep newname.c
```

Now, AccuRev knows that the element formerly known as **oldname.c** is henceforth to be known as **newname.c** (until the next name change, that is!).

## Cleaning Up

Finally, deactivate the **import** workspace:

```
accurev rmws import
```

# Creating and Using a Maintenance Stream

Many software development organizations have two main streams of development: work towards the next release, and maintenance of the previous release. Other SCM systems use a “branch based on a label” paradigm to accomplish this. AccuRev uses snapshots (static streams).

## Creating a Snapshot

At the time of the release (say, “WidgetSoft Release 1.0”), create a snapshot:

```
accurev mksnap -s widget1.0 -b widget -t now
```

This creates a new snapshot called **widget1.0**. The snapshot contains whatever versions the **widget** stream contained at the time the *mksnap* command is executed. Subsequently, the **widget** stream can change as new versions are promoted to it, but the **widget1.0** snapshot never changes. Instead of *now*, you can specify any time in the past, such as **2005/05/18 10:10:24**.

## Creating a Stream Based on the Snapshot

For maintenance work on this release, create a new dynamic stream based on the snapshot:

```
accurev mkstream -s widget_maint -b widget1.0
```

Initially, **widget\_maint** will be identical to **widget1.0**, but it will change as people promote changes to it.

## Performing Maintenance Work

Maintenance developers use workspaces based on the **widget\_maint** stream. For instance, to make a maintenance fix, Mary might create a workspace like this:

```
accurev mkws -w widget_maint -b widget_maint -l <wherever>
```

When Mary promotes her maintenance work, the changes will go to **widget\_maint**.

All maintenance work is isolated from the main development stream, and vice-versa. Developers working on the next release create their workspaces off the development stream, not the maintenance stream. For example:

```
accurev mkws -w widget -b widget -l <wherever>
```

Changes promoted from **widget\_justine** will go to the main development stream, **widget**. The changes won't appear in the **widget\_maint** stream.

The Change Palette in the AccuRev GUI makes it easy to migrate changes back and forth between a main development stream (**widget**) and a maintenance stream (**widget\_maint**).



## Pathname Optimization: ACCUREV\_IGNORE\_ELEMS and .acignore

Like most command-line programs, AccuRev's CLI tool, **accurev**, accepts one or more filename/pathname specifications as command arguments:

```
accurev keep base.h
```

```
accurev promote intro.doc chap*.doc
```

Such arguments cause the command to be invoked on a particular set of files.

But several **accurev** commands are capable of operating on *all* the files in the current workspace. For example, this command searches the entire workspace containing the current working directory, and lists the files that have not been placed under version control (external files):

```
accurev stat -x
```

Such “whole-workspace” commands are very powerful and useful, but they can be time-consuming. If your workspace contains many hundreds or thousands of files, you must wait while all the names are transmitted to the server machine, the AccuRev Server process determines the status of each file, and information on the matching files is returned to the client machine.

This large list of files to be processed may contain a significant number of “don't care” files. For example, a search for external files is probably intended to locate source files (with suffixes like **.c** or **.cc** or **.java** or **.bas**) that you've forgotten to place under version control. You probably don't care about program-generated files with suffixes like **.exe** (executables built in the source directory), **.bak** (editor backup files), **.msg** (copies of mail messages, and so on — because you don't intend to place them under version control).

You can use the environment variable **ACCUREV\_IGNORE\_ELEMS** to specify up to 50 patterns (or even individual filenames/pathnames). When it executes certain “whole-workspace” commands, the **accurev** tool ignores all external files that match this specification. For example, setting **ACCUREV\_IGNORE\_ELEMS** to the following value causes the **stat -x** command to ignore all **.exe** and **.bak** files:

```
*.exe *.bak
```

**ACCUREV\_IGNORE\_ELEMS** is also used — in a slightly different way — by certain commands that process a particular set of files, instead of the whole workspace.

AccuRev now supports specification of objects to be ignored on a per-directory basis, using configuration files, in addition to the global specification in environment variable **ACCUREV\_IGNORE\_ELEMS**. The following sections explain the details of using the environment variable and the configuration files.

## Eligible “Whole-Workspace” Commands

The following **accurev** commands use the value of `ACCUREV_IGNORE_ELEMS` to filter names:

Command	Description
<b><i>stat -x</i></b>	List external (non-version-controlled) files
<b><i>stat -m</i></b>	List files that have been modified
<b><i>stat -n</i></b>	List files that have been modified, but are not active from AccuRev’s viewpoint (that is, are not in the workspace’s default group)
<b><i>stat -p</i></b>	List files that are pending promotion — files that have been modified, along with files with <b>(kept)</b> status
<b><i>add -x</i></b>	Place external files under version control
<b><i>update</i></b>	Update the workspace by incorporating versions from the backing stream (only if <code>USE_IGNORE_ELEMS_OPTIMIZATION</code> is set to <code>TRUE</code> )

These commands apply to the entire workspace if you *don’t* specify any filenames/pathnames or wildcard patterns on the command line:

```
accurev stat -n          (“whole-workspace” command)
accurev stat -n *.doc    (pattern specified; not a “whole-workspace” command)
```

When applying these commands to an entire workspace, the **accurev** tool, running on the client machine, uses `ACCUREV_IGNORE_ELEMS` to filter the list of filenames *before* sending the list to the AccuRev Server process. This can significantly reduce the amount of network traffic, and also reduce the amount of file-status computation the Server process must perform.

Note: by default, the ***stat*** command also uses a timestamp-based optimization to reduce the number of files it must process. For details, see [Optimized Search for Modified Files: the Scan Threshold](#) on page 248 in the *AccuRev CLI User’s Guide*.

## GUI Counterparts to the “Whole-Workspace” Commands

The AccuRev GUI also uses the `ACCUREV_IGNORE_ELEMS` environment variable. Working in the searches pane of the File Browser corresponds to using the various “whole-workspace” forms of the **accurev stat** command. Thus, `ACCUREV_IGNORE_ELEMS` is used by these search criteria:

- External (***stat -x***)
- Modified (***stat -m***)
- Non-member (***stat -n***)
- Pending (***stat -p***)

## Commands that Don’t Apply to the Whole Workspace

The ***stat*** and ***add*** commands above accept filename/pathname specifications, in several forms:

- Individual filename: **chap01.doc**
- Individual pathname: **doc/chap01.doc** or **./widgets/doc/chap01.doc**

- Wildcard pattern: **\*.doc** or **docs/\*.doc**
- list-file: **-l my\_list\_of\_files**

Such specifications restrict the command to processing a certain set of files, not the whole workspace (even if you also specify **-x**, **-m**, **-n**, or **-p**). Similarly, the **files** command processes a certain set of files, not the whole workspace.

For these commands, the **accurev** tool still uses ACCUREV\_IGNORE\_ELEMS — but in a way that is less efficient than for whole-workspace commands:

1. Send the list of files that match the filename/pathname specification to the AccuRev Server.
2. Use ACCUREV\_IGNORE\_ELEMS to filter the data that the Server returns. Only names of external files are filtered out; files that are AccuRev elements remain in the listing, even if they match the value of ACCUREV\_IGNORE\_ELEMS.

This may produce similar, or even identical results as a whole-workspace command, but the fact that an unfiltered list is sent to, and processed by, the Server means that overall performance won't be as good.

## Values for ACCUREV\_IGNORE\_ELEMS

The value of the ACCUREV\_IGNORE\_ELEMS environment variable must be a SPACE-separated list of filenames, pathnames, and wildcard patterns. Some examples:

```
*.exe
*.exe *.doc
manuals/*.doc
*.doc README.html
```

You can use any of these wildcards:

- **?** matches any one character
- **\*** matches any sequence of characters (including a zero-length sequence — see note below)
- **\*\*** specifies recursion down through the directory structure
- **[aekz]** matches **a**, **e**, **k**, or **z**
- **[a-e]** matches **a**, **b**, **c**, **d**, or **e**
- **{one,two,seven}** matches **one**, **two**, or **seven**

Additional rules to keep in mind include:

- You can use Windows style directory separators (**\**) or \*nix style (**/**) in your rules — they all get converted to **"/**" internally.
- Any instances of **"/**" get converted to **"/**".
- Starting a path spec with **"/**" makes that path absolute. Anything else causes AccuRev to assume that the path is relative.

## Examples

A simple wild card pattern such as “\*.doc” matches any of these names:

```
chap01.doc
manuals/chap01.doc
widgetproj/src/manuals/usergd/chap01.doc
```

The pattern **manuals/\*.doc** matches any of these names:

```
manuals/chap01.doc
manuals/chap02.doc
```

... but not these names:

```
manuals/usergd/src/chap01.doc
widgetproj/src/manuals/usergd/chap01.doc
```

However, using “\*\*” to specify recursion as in **manuals/\*\*/\*.doc** or **manuals/\*\*chap\*.doc** will match any occurrence of \*.doc or chap\*.doc in any directory underneath any instance of a **manuals** directory.

You can use the following pattern to ignore all items in the directory tree(s) named **usergd** (including **usergd** itself):

```
usergd*
```

## Specifying Directories and Their Contents

A typical application of ACCUREV\_IGNORE\_ELEMS is to have **stat -x** (“list all external files”) ignore temporary build directories. That is, you want the listing to exclude both the directories themselves and all the files within those directories. If the build directories are named **build\_001**, **build\_002**, etc., you might be tempted to use this pattern:

```
*/build_???/* or build_???/*
```

But this pattern matches only the *contents* of the directories, not the directories themselves. **Note:** a directory matching the pattern that is a subdirectory of another matching directory will be excluded. For example, in structure like **build\_001/build\_002**, **build\_002** will be excluded, but **build\_001** will not.

Instead, use the following value for ACCUREV\_IGNORE\_ELEMS:

```
*/build_??? */build_???/*
```

(The single pattern **\*/build\_???\*** would match both directories and their contents. But it also might coincidentally match names of some source files, such as **lib/build\_end.c**.)



## Setting ACCUREV\_IGNORE\_ELEMS on a UNIX/Linux System

When setting the ACCUREV\_IGNORE\_ELEMS environment variable on a UNIX or Linux system, be sure to single-quote or double-quote the value, in order to protect any wildcard characters from being expanded by the shell:

```
export ACCUREV_IGNORE_ELEMS="*.exe *.doc"    (Bourne shell family)
setenv ACCUREV_IGNORE_ELEMS "*.exe *.doc"    (C shell family)
```

To determine the current value of ACCUREV\_IGNORE\_ELEMS, use either of these commands:

```
env | grep ACCUREV_IGNORE_ELEMS              (or a shorter 'grep' pattern)
echo "$ACCUREV_IGNORE_ELEMS"                 (don't forget the quotes!)
```

## Setting ACCUREV\_IGNORE\_ELEMS on a Windows System

On a Windows system, you can set the ACCUREV\_IGNORE\_ELEMS environment variable in the System applet (on the Control Panel). Alternatively, use the *set* command in a Command Prompt window:

```
set ACCUREV_IGNORE_ELEMS=*.exe *.doc        (no quotes!)
```

Don't use quote characters, even if the value includes SPACES.

To determine the current value of ACCUREV\_IGNORE\_ELEMS in a Command Prompt window, use either of these commands:

```
set
echo %ACCUREV_IGNORE_ELEMS%
```

## Per-Directory Pathname Optimization — the .acignore File

AccuRev's pathname optimization facility provides for faster performance by allowing certain objects to be ignored during various commands — notably (**external**)-status objects during whole-workspace searches. The value of environment variable ACCUREV\_IGNORE\_ELEMS is a pathname pattern — or several patterns, separated by SPACES; objects whose pathnames match a pattern are ignored by the various commands.

AccuRev now supports specification of objects to be ignored on a per-directory basis, in addition to the global specification in ACCUREV\_IGNORE\_ELEMS. One or more directories in a workspace can contain a text file named **.acignore**. An **.acignore** file works the same way as ACCUREV\_IGNORE\_ELEMS, with these exceptions:

- In an **.acignore** file, multiple patterns must appear on separate lines — not SPACE-separated on a single line.
- An **.acignore** file applies only to its own directory. In particular, it does *not* apply recursively to lower-level directories.



# Techniques for Sharing Workspaces

This note describes two techniques for accessing the same workspace from multiple machines.

## Accessing a Windows Workspace From Multiple Windows Clients

Multiple AccuRev users, on Windows client machines, can share a workspace that is physically located on a Windows machine. (Or maybe there's just one user, who wants to access a workspace from multiple Windows client machines.)

1. Designate a directory that is *above* the top-level directory of the workspace tree as a Windows “shared directory”. For example, if the workspace tree for workspace **widget\_maint\_derek** on machine **derekpc** is located at **C:\widget\workspaces\maintdrp**, you could set the shared directory as follows:

```
net share widgwork=C:\widget\workspaces
```

Note: the workspace tree's top-level directory itself (**maintdrp** in the example above) cannot be designated as the shared directory.

2. Determine how AccuRev records the workspace tree location, using the command **accurev show wspaces**. (The pathname will always use forward slashes, even if it's a Windows pathname.)

- If the workspace tree location incorporates the share name ...

```
widget_maint_derek    /widgwork/maintdrp    derekpc    ...
```

... skip to Step 3.

- But if the workspace tree location appears as an absolute pathname ...

```
widget_maint_derek    C:/widget/workspaces/maintdrp    derekpc    ...
```

... you must use the **chws** command to change the recorded location to a pathname that incorporates the share name. This involves mapping a network drive to the shared directory:

```
> net use K: \\derekpc\widgwork
The command completed successfully.
```

```
> K:
```

```
> cd \maintdrp
```

```
> accurev chws -w widget_maint_derek -l .          (“dash-ell dot”)
Changed location.
Changed machine name.
```

```
> accurev show wspaces
```

```
...
widget_maint_derek      /widgwork/maintdrp      derekpc      ...
```

3. Now, all users on Windows client machines can access the workspace tree by mapping a network drive to the shared directory. Even the user on the machine where the workspace tree is located (**derekpc** in our example) must use a network drive to access the workspace tree.

```
> net use P: \\derekpc\widgwork
The command completed successfully.
```

```
> P:
```

```
> cd \maintdrp
```

```
> accurev info
```

```
...
Workspace/ref:  widget_maint_derek
Basis:          widget_maint
Top:            P:/maintdrp
```

Users on different machines can map the shared directory to different drive letters, and access the workspace as, for example, **Y:\maintdrp** or **R:\maintdrp**.

## Universal Access to a Workspace Located on a Share

A workspace whose workspace tree is network-accessible through a share, can be accessed from any client machine — running UNIX/Linux or Windows. The share can be configured through Samba/SMB or some other network file system. It must make the actual storage location available through a machine name and a simple “share name”: a name that looks like a single pathname component.

### The ‘share\_map.txt’ File

The technique in *Accessing a Windows Workspace From Multiple Windows Clients* on page 13 relies only on Windows operating system facilities. But the “universal workspace access” technique requires the maintaining of a pathname-mapping file for use by the AccuRev Server. If a “share” (that is, shared directory) has an entry in the pathname-mapping file, any workspace located on that share can be used on all AccuRev client machines capable of accessing the machine where the share resides.

The pathname-mapping file is a text file, **share\_map.txt**, which must be located in the AccuRev **site\_slice** directory on the AccuRev Server machine. It maps share names to absolute pathnames. Each line of **share\_map.txt** consists of three TAB-separated fields, describing one share. For example:

```
jupiter      accwks      /public05/accurev_workspaces
```

This entry says, “a share named **accwks** is physically located on machine **jupiter**, at absolute pathname **/public05/accurev\_workspaces**”. More generally:

- The first field (**jupiter**) names a machine where one or more workspace trees are (or will be) located.
- The second field (**accwks**) specifies the share name.
- The third field (**/public05/accurev\_workspaces**) indicates the absolute pathname of the share on the machine. On a Windows machine, this includes the drive letter — for example, **C:/Public Directories/AccuRev Workspaces**.

Notes:

- All pathnames in **share\_map.txt** must use forward-slash characters ( / ), even Windows pathnames.
- The field separator in the **share\_map.txt** file must be single TAB character — don't use SPACES. If a specification (e.g. a share name) includes a SPACE character, do not enclose the specification in quotes.

## Workspace Location Entries

The **accurev show wspaces** (or the GUI's **View > Workspaces**) command displays the locations of existing workspaces in the repository. The pathnames always use forward slashes, even if they are Windows pathnames.

AccuRev can record the location as an absolute pathname on its machine:

```
C:/wks/light24/mnt_john           (Windows)
/bigdisk/home/john/widget_devel   (UNIX/Linux)
```

Alternatively, it can record a location that incorporates a share name:

```
/accwks/wks_john                 (Windows or UNIX/Linux)
```

Universal workspace access requires that a workspace's location be recorded as an absolute pathname in the workspaces table. (Note that the sharing technique described in [Accessing a Windows Workspace From Multiple Windows Clients](#) on page 13 has the opposite requirement: workspace locations must incorporate the share name.) In addition, the "Host" name listed in this table for a workspace must exactly match the first field in some **share\_map.txt** entry. Beware of domain name discrepancies — for example, **jupiter** vs. **jupiter.mycorp.com**.

## Fixing Workspace Location Entries

If a workspace located on a share has the "wrong kind" of entry in the workspaces table, fix it to enable universal client access:

1. Make sure that **share\_map.txt** has a valid entry for the share.
2. On any client machine that can "see" the workspace, go the top-level directory of the workspace tree.
3. Use the **chws** command to change the location recorded in the workspaces table to an absolute pathname:

```
> accurev chws -w <workspace-name> -l .           ( "dash-ell dot" )
```

You must fix each workspace location entry in this way individually.

## Fixing Machine Name Entries

If there's a discrepancy between a machine's name in a **share\_map.txt** entry (say, **jupiter**) and its name in the workspaces table (say, **jupiter.mycorp.com**), change the workspace table entry:

```
accurev chws -w <workspace-name> -m jupiter
```

## Example: Samba Share

Here's an example of how it can all work in a Samba environment, elaborating on the scenario above:

1. The organization decides that a directory, **/public05/accurev\_workspaces**, on UNIX host **jupiter** will be a location where users can create workspaces that can be shared across platforms.
2. The system administrator on **jupiter** turns that directory into a Samba share, named **accwks**. Here's the relevant excerpt from the Samba **smb.conf** file on host **jupiter**:

```
[accwks]
    comment = All users
    path = /public05/accurev_workspaces
    browseable = yes
    guest ok = yes
    writeable = yes
```

3. The AccuRev administrator makes this entry in the **share\_map.txt** file, in the AccuRev Server's **site\_slice** directory.

```
jupiter    accwks    /public05/accurev_workspaces
```

4. User **john**, working on a Windows machine, wants to create a workspace on the share. First, he makes the share accessible as a network drive:

```
net use T: \\jupiter\accwks
```

5. Then **john** creates his workspace on this network drive:

```
accurev mkws -w shrwks_john -b dvt_stream -l T:\wks_john
```

A **show wspaces** command indicates that the AccuRev Server uses an absolute pathname to record the new workspace's location on **jupiter**:

```
shrwks_john    /public05/accurev_workspaces/wks_john    jupiter    ...
```

6. **john** can now use this workspace from any client machine that can access the machine where the share resides.

# What's the Difference between Populate and Update?

AccuRev users sometimes confuse the two commands *Populate* and *Update*. These commands seem similar because they both bring new data into your workspace. But they are quite different, both in their usage pattern — most people use *Update* far more often — and in what they accomplish. Understanding the difference between these two commands will enable you to choose the right command at the right time (always useful!), and will deepen your knowledge of how AccuRev really works.

Note: the AccuRev CLI command **accurev pop** corresponds to the GUI's *Populate* command.

This note starts with a brief statement of the difference between *Populate* and *Update*, along with a few examples. Then, we present a full discussion of the data structures and mechanisms involved in these commands.

## In a Nutshell ...

The essential difference between *Populate* and *Update* concerns time. Roughly speaking, your workspace contains an informal “baseline” (the contents of the shared backing stream, at a particular moment) plus “changes” (the modifications that you make to some of the files). The *Update* command advances the workspace's baseline from the time of the workspace's last update to the present moment. This incorporates into the workspace data recently placed in the backing stream by other team members.

Note: AccuRev actually tracks the workspace's baseline in terms of transactions, not timestamps.

The *Populate* command doesn't advance a workspace's baseline at all, but leaves it “stuck in the past”. Instead, *Populate* simply restores the appropriate “old” version of one or more elements that are currently missing from the workspace.

The two commands also differ in their scope: *Update* always processes the entire workspace; *Populate* processes just a selected set of elements or directory subtrees.

The capsule description above uses imprecise language, such as “advancing the workspace's baseline” and “old version”. The following description is more precise, using AccuRev-specific terminology. The terms are explained fully in section *Data Structures Used by Populate and Update* on page 19 below.

The *Update* command changes both the workspace stream and the workspace tree:

- It advances the workspace stream's update level to the depot's most recent transaction — say, from current update level 32155 to new update level 34002. This allows a new set of versions — in this case, some or all the versions created by transactions 32156 through 34002 — to flow into the workspace stream from the backing stream.
- It copies the contents of the workspace stream's new versions from the repository's file-storage area to the workspace tree.

By contrast, the **Populate** command changes the workspace tree only, not the workspace stream. In particular, it doesn't change the workspace stream's update level. **Populate** merely fixes a discrepancy between the workspace stream and the workspace tree: a certain version of a file is in the workspace stream, but there is no actual file in the workspace tree — that is, the file's status is **(missing)**. To fix this situation, you invoke **Populate**, which copies the version currently in the workspace stream to the workspace tree.

Note: it would be incorrect to conclude that **Update** *never* processes **(missing)** elements, and that **Populate** *only* processes **(missing)** elements. Examples 3 and 4 below show that exceptions exist for both these “rules”.

## Example 1: Standard Update Scenario

You've just finished a coding project, so you're not actively working on any files in your workspace. Other team members create new versions of files **red**, **white**, and **blue** in their workspaces, then promote those versions to the team's backing stream. You invoke the **Update** command, which copies the most recent versions of **red**, **white**, and **blue** from the backing stream to your workspace.

## Example 2: Restoring a Deleted File (“missing” by accident)

Since you have complete control over the files in the workspace tree, it's easy to accidentally delete a version-controlled file with an operating-system command or a third-party tool. If you do this, AccuRev knows that the file *should* be there, because a version of the element still exists in the workspace stream. Thus, the File Browser continues to list the element, but shows it as **(missing)** from the workspace tree. You select the element and invoke **Populate** to fix the accidental deletion.

## Example 3: Handling Active Elements

**Update** and **Populate** differ in how they handle elements that are active (are in the workspace's default group). The **Update** story is simple: it *never* overwrites the file in the workspace tree. **Populate** usually doesn't overwrite the file, but there are a couple of cases to consider.

- It doesn't *need* to overwrite a file that you've kept and *not* subsequently edited, because the active version in the workspace stream is identical to the file in the workspace tree.
- But if you have subsequently edited the file in the workspace tree, so that the element status is **(modified)(member)**, then you can order **Populate** to overwrite the file and clobber those subsequent edits. This can also happen with a file that you've edited, but never kept, so that its status is **(modified)**.

Be careful — **(modified)** files will also be overwritten if you invoke **Populate** with both the **Recursive** and **Overwrite** options on a directory that directly or indirectly contains the active element.





## Example 4: A Tale of Two Files

Let's see how *Update* and *Populate* differ in this situation:

You have a workspace that is completely up to date. You delete two files, named **blue** and **green**. Someone creates a new version of **blue** in another workspace, and then promotes it to your workspace's backing stream.

If you select both **blue** and **green** in the File Browser, then invoke *Populate*, the two files that you deleted are restored to the workspace tree. This does *not* bring in the new backing-stream version of **blue**, because that version is not in the workspace stream — it's too new, having been created after your workspace's most recent update.

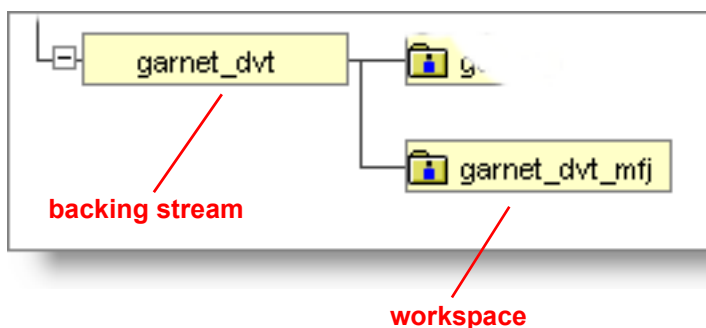
If you invoke *Update* instead of *Populate*, the workspace tree gets the new version of **blue**. No version of **green** is copied to the workspace tree, because *Update* only handles new versions — ones that enter your workspace stream as a result of advancing its update level.

## Data Structures Used by Populate and Update

The simplicity of AccuRev's day-to-day usage model stems, in large part, from the fact that you don't need to worry about the "big picture" of your organization's development scheme and process. Instead, you only need to concern yourself with:

- the workspace in which you maintain your private copies of version-controlled files
- the workspace's backing stream, a "data switchboard" that organizes the sharing of files' changes with other members of your development team

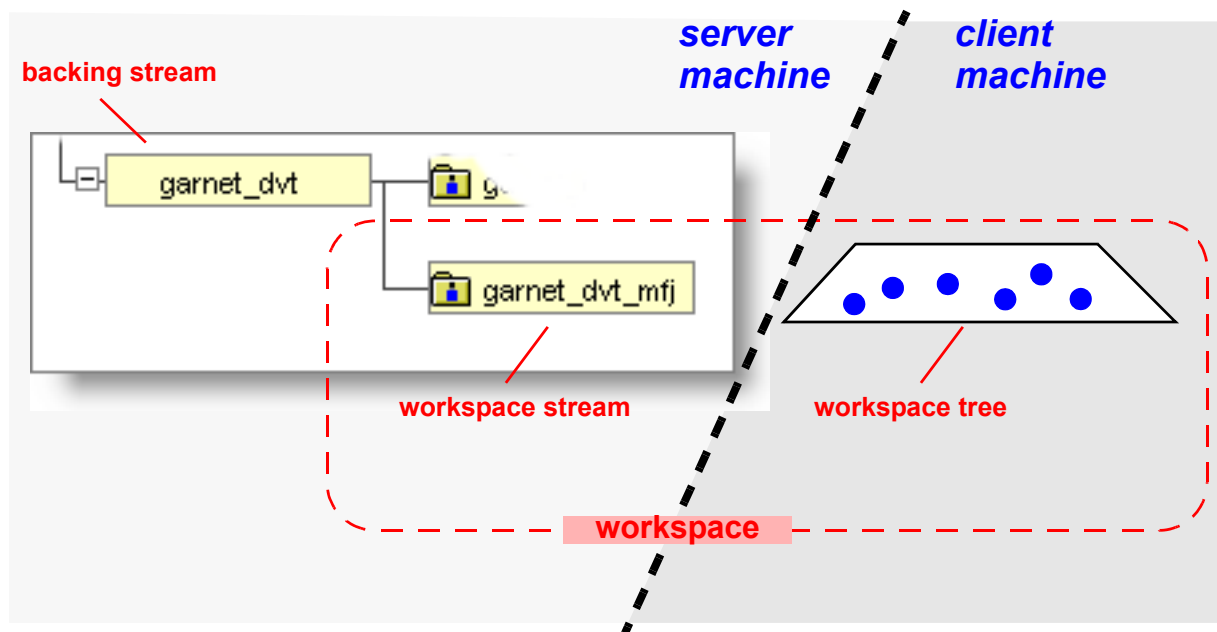
The illustration below shows how a workspace and its backing stream typically appear in the AccuRev StreamBrowser:



But a workspace actually consists of two parts:

- the workspace tree, an ordinary directory tree ("just a bunch of files")
- the workspace stream, which contains all of the workspace's configuration management information

So the picture looks more like this:



The above illustration shows one important difference between a workspace’s two parts: the workspace tree lives in “AccuRev client space”, while the workspace stream lives in “AccuRev Server space”. The following table summarizes all the important differences.

Workspace Stream	Workspace Tree
Resides in an AccuRev depot, located on the AccuRev server machine	A standard directory tree, located on your client machine (or in other user-accessible storage)
Managed by the AccuRev Server process	Managed by you, the individual user
Contains all version control and configuration management information for the workspace, such as version-IDs	Contains no version control or configuration management information
Contains no actual files, just <u>version</u> objects that point to files in permanent storage	Contains <i>only</i> files and directories, which you can edit, compile, copy, etc.
Operating system commands and tools never change data here	Operating system commands and tools can change data here

The sections below expand on these differences.

## How the Data Structures Get Their Data

Each of the data structures introduced above — backing stream, workspace stream, and workspace tree — is different in the way it gets changes (i.e. new data) from other parts of the development environment.

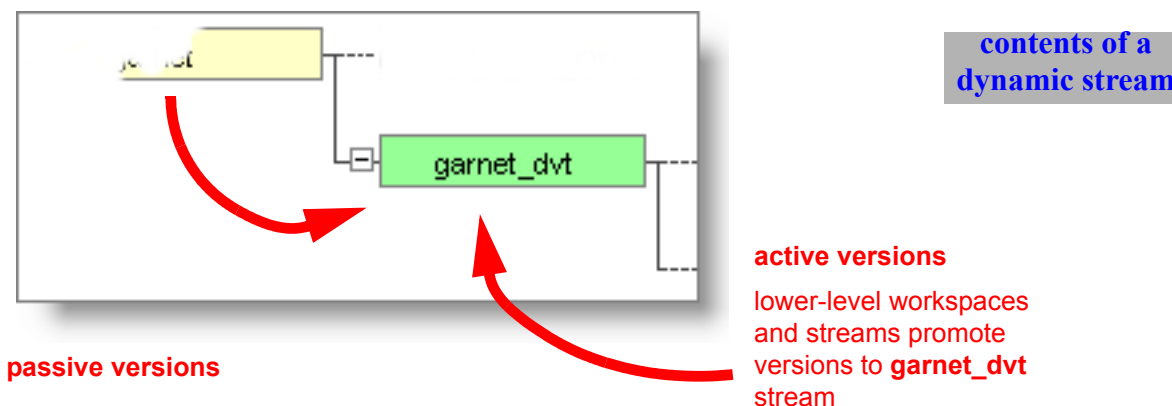
## Backing Stream

The backing stream is, in most cases, a dynamic stream. (It can also be a snapshot or a time-based stream.) A dynamic stream is a changing configuration of its depot. At any given moment, it (logically) contains a simple table that indicates particular versions of a set of elements. For example:

Element	Version-ID
doc	garnet_dvt\1
doc\chap01.doc	garnet_dvt\5
doc\chap02.doc	garnet\3
doc\chap03.doc	garnet_dvt\2
src	garnet\1
src\garnet.c	garnet_dvt\12
src\commands.c	garnet_dvt\7
tools	garnet\3
tools\start.sh	garnet_dvt\6
tools\end.sh	garnet\2

At any given moment, a dynamic stream's versions fall into two categories:

- **passive versions:** versions that the stream inherits from its parent stream. Inheritance is automatic and instantaneous: as soon as a new version enters the parent stream, it is inherited at once by the child stream.
- **active versions:** versions that have been *Promoted* to the stream, (usually) from lower-level workspaces and substreams. This set of versions constitutes the stream's default group.



In the configuration table above, all the **garnet\_dvt\...** version-IDs indicate active versions in the **garnet\_dvt** stream. All the **garnet\...** version-IDs indicate passive versions, inherited from the parent stream, named **garnet**.

## Workspace Stream

The workspace stream is the “behind the scenes” part of your workspace. It is the information contained in the database about changes made to your workspace. In many ways, it resembles a dynamic stream:

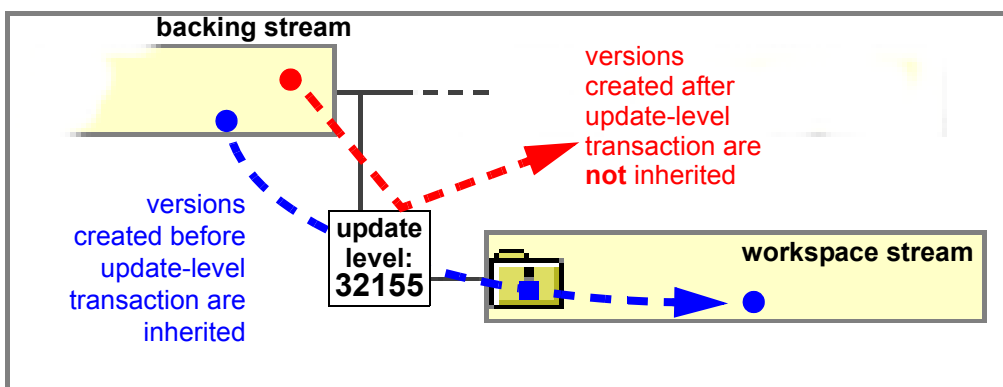
- It’s a changing configuration of one of the repository’s depots.
- It logically contains a particular version of some or all of the depot’s elements.
- It doesn’t contain actual files, but is logically just a table of elements and version-IDs.

Note: when creating a new version of a file, the **Keep** command copies the file to the repository’s file-storage area, not to the workspace stream itself. The workspace stream just gets a version-ID for the new version; the version-ID serves as a pointer to the file in the file-storage area.

- It contains active versions, created by explicit user commands: **Add**, **Keep**, **Rename**, **Defunct**, etc. The new versions in the repository preserve the changes that you’ve made to files in your workspace tree. (There’s only one way to create an active version in a dynamic stream: the **Promote** command.)
- It also contains passive versions, inherited from its parent stream, the workspace’s backing stream.

The last item is where the crucial difference between workspace streams and dynamic streams comes into play. The versions inherited by the workspace stream are not the ones *currently* in the backing stream, but the versions that *were* in the backing stream when the workspace was last updated. This is called the workspace’s update level. More precisely, AccuRev records the number of the depot’s most recent transaction — say, transaction #32155 — as the workspace stream’s update level. So we can rephrase the principle:

*The workspace stream inherits from the backing stream versions that were created in transactions up to and including the workspace’s **update level**.*



Note: the update level of a workspace stream is very much like the optional basis time of a dynamic stream. Both mechanisms restrict the flow of versions to a child stream from its parent stream, based on a point in the depot's development history.

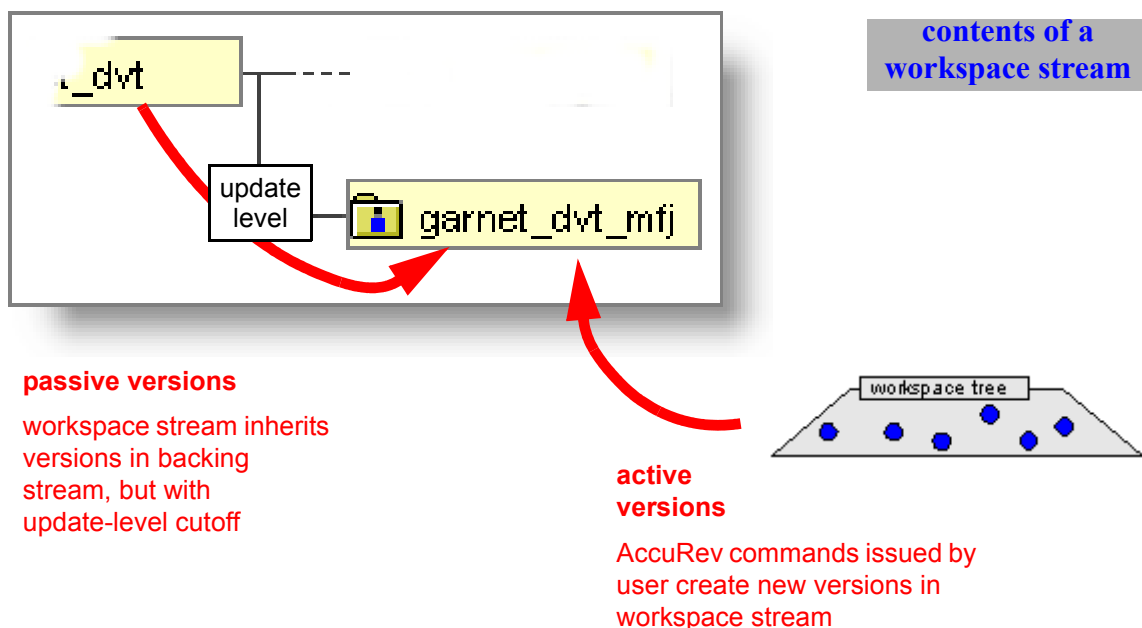
Thus, a workspace stream is not updated dynamically when changes occur to its parent stream. It gets new versions from the backing stream only when you issue an **Update** command. This is how AccuRev implements the workspace's user-controlled "privateness", isolating it from the changes regularly being recorded in the backing stream by other team members.

To summarize: at any given moment, a workspace stream contains:

- a set of passive, inherited versions, created in transactions that do not exceed the workspace's update level. (A file that you've **Promote**'d since the last update is an exception. The version is passive, but was created after the workspace's update.)
- a set of active versions, which you've created in that workspace with such AccuRev user commands as **Add**, **Keep**, **Rename**, and **Defunct**.

Roughly speaking, the set of versions in the workspace stream indicates what data currently *should* be in your workspace tree. Examples:

- The workspace stream contains active version **garnet\_dvt\_mfj\9** of file **commands.c**, which you created with the **Keep** command. This means your workspace tree should contain a file **commands.c** that matches the repository file referenced by version-ID **garnet\_dvt\_mfj\9**.
- The workspace stream contains passive version **garnet\_dvt\6** of file **start.sh**. This means your workspace tree should contain a file **start.sh** that matches the repository file referenced by the version in the backing stream, **garnet\_dvt\6**.

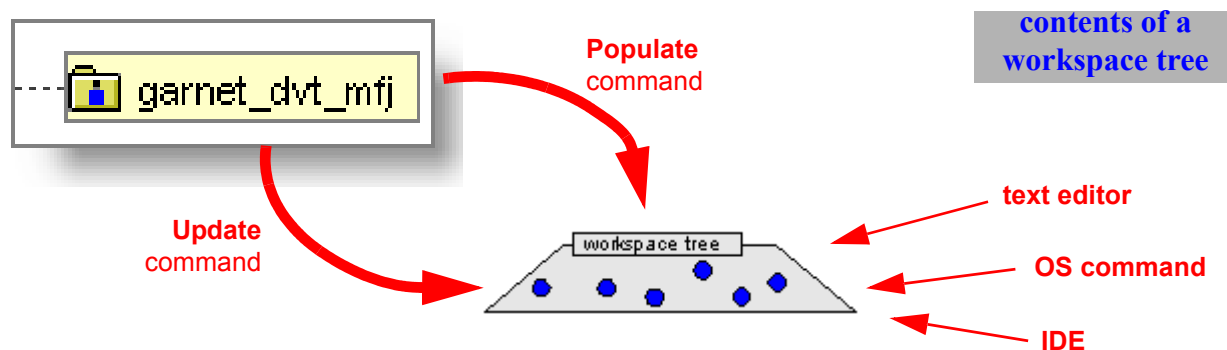


If you modify a file without **Keep**'ing it (or modify it again after **Keep**'ing it), the file in the workspace tree does not exactly match the version in the workspace stream. AccuRev indicates this by reporting the file's status as **(modified)**.

## Workspace Tree

The workspace tree is an ordinary directory tree, located in your personal disk storage. You can modify the contents of the workspace tree in two basic ways:

- By invoking operating system commands, text-editing tools, IDEs, etc.
- By invoking AccuRev commands to copy existing versions from the repository to the workspace tree. This can either overwrite existing files in the workspace tree or add new files. Both the *Populate* and *Update* commands copy versions from the workspace stream to the workspace tree. (So do a couple of other commands, such as *Send to Workspace*.)



## The Update Algorithm

**Note:** The following discussion includes a description the traditional AccuRev Timestamp Optimization algorithm. AccuRev Release 5.4 introduced an new, optional, client-oriented approach to timestamp optimization. This new algorithm is discussed in the Java GUI on-line help (*AccuRev On-Line Help Guide*), at *Timestamp Optimization: Controlling the Determination of (modified) Status* on page 93.

This section describes the processing of the *Update* command in detail. It's not as simple as “get all the new versions”, because AccuRev takes care not to overwrite version-controlled files that you have changed, but whose changes have not yet been preserved in the repository.

1. AccuRev first searches the workspace tree for such “at-risk” files, by performing a CLI *stat* (file status) command on your workspace:
  - It uses the *-n* option to *stat*, which restricts the search to version-controlled files that you have modified but are not in the workspace's default group — the status of such “non-member” files includes the **(modified)** indicator but not the **(member)** indicator. It's safe to ignore default-group files, because these are never affected by an *Update*.
  - It uses timestamp optimization to speed the file search: it ignores files whose timestamps precede the workspace's scan threshold (the time that the workspace was most recently updated or otherwise searched for modified files). If you modify a file by overwriting it with a file with an old timestamp, the file will be ignored in this step. This can cause problems in Step 5 below.

- If environment variable `USE_IGNORE_ELEMS_OPTIMIZATION` is set to `TRUE` (the value is case-insensitive), then it uses the value of environment variable `ACCUREV_IGNORE_ELEMS` to ignore certain pathnames in the file search. If an ignored file actually has **(modified)** status, an error will occur in Step 5 below. Make sure that `ACCUREV_IGNORE_ELEMS` matches *only* external elements when doing an update. Otherwise, **Update** will fail when it encounters a modified element that has been ignored because it matches the `ACCUREV_IGNORE_ELEMS` setting during the non-member search. (Note that even though the status of these files change from “modified” to “backed” in this scenario, the modifications in the workspace are not lost.)
  - To make sure that a file with a recent timestamp has actually been modified, it compares the file with the version currently in the workspace stream by performing a checksum. (This means that simply modifying a file’s timestamp with a *touch* command won’t prevent the file from being overwritten by **Update**. You have to make a real change to the file.)
2. If the preceding step found any “non-member” files — modified, but not in the default group — the **Update** command in AccuRev releases prior to Version 4.6 terminated immediately, without updating any file. Starting in Version 4.6, the **Update** command can sometimes proceed, even in the presence of such files:
    - A non-member file that is not due to be updated, because there is no newer version in the backing stream, does not prevent the update from proceeding.
    - A non-member file that *is* due to be updated has **(modified)(overlap)** status. By default, the presence of one or more such files causes the update to terminate immediately, without updating any file. But you *might* be able to enable the update to proceed if you invoke an update option — **update -m** in the CLI, user preference **Update Resolves Trivial Merges** in the GUI. With this option invoked, the update proceeds only if a trivial merge can be performed for each file with **(modified)(overlap)** status. (The backing-stream version can be merged with the workspace-tree file automatically, because there are no conflicts that require manual intervention.)
  3. AccuRev notes the number of the repository’s latest transaction, and sets that number as the workspace’s target update level.
  4. AccuRev decides which recently created versions in the repository should be delivered to the workspace tree. A version is a candidate for delivery if it became visible in the workspace’s backing stream by one of the transactions between the workspace’s *current* update level and the newly set *target* update level.
  5. AccuRev attempts to deliver all those versions to the workspace tree. Each time it is about to overwrite a file in the workspace tree, AccuRev first makes sure it won’t be “clobbering” unpreserved changes: it checksums the existing file to confirm that it matches the current version in the workspace stream (the version at the current update transaction level).

Note: it’s sometimes OK for the workspace tree to already contain the new version (the version at the target update level). See *Incomplete Updates* below for an explanation.

If a file about to be overwritten fails this checksum step, AccuRev reports a “crc mismatch” and the **Update** command terminates immediately. The most common cause of this error is

your having overwritten the file in such a way that it gets an old timestamp. Such a file escapes detection in Step 1, but gets caught here — just in the nick of time to avoid being clobbered.

If the checksum succeeds, the file is safe to overwrite, so AccuRev updates it — finally! The update can be a replacement of the file’s contents, a change in its pathname, or both.

If AccuRev is processing files with **(modified)(overlap)** status (see Step 2 above regarding *update -m*), it automatically merges the backing-stream version with the file in the workspace tree, instead of simply overwriting the file. No checksum of the workspace-tree file is performed for such elements.

6. If *all* the recently created versions identified in Step 4 were successfully delivered to the workspace:
  - The target update level becomes the workspace’s current update level, indicating a successful, complete update.
  - The workspace’s scan threshold, used for the *stat -n* timestamp optimization (Step 1), is set to the time that this successful update began.

## Advancing the Scan Threshold

The timestamp optimization is most effective when it enables AccuRev to ignore as many files as possible. Accordingly, AccuRev takes advantage of opportunities to validly advance the scan threshold to a later time:

- If a workspace is completely up-to-date, the *Update* command advances the scan threshold anyway, setting it to the time the command began. (Thus, an unnecessary *Update* doesn’t “do nothing”.)
- The CLI command *stat -n* advances the scan threshold to the point in time just before the *earliest* timestamp among the files it finds (modified files that are not members of the workspace’s default group). This preserves the validity of the *timestamp optimization principle*: for file elements that are not in the workspace’s default group, the timestamp of a modified file is later than the workspace’s scan threshold.

If *stat -n* does not find any non-member modified files, it advances the scan threshold to the time that the command began.

If *stat -nO* finds non-member modified files whose timestamps precede the scan threshold, it moves the scan threshold *backward*, in order to preserve the timestamp optimization principle described above.

## Incomplete Updates

The *Update* command is not implemented as an atomic operation, and it is not recorded as an AccuRev transaction. Transactions are used to organize and serialize changes to the repository, not to the workspace trees that implement user “sandboxes”. An *Update* can take a significant amount of time, and is sometimes interrupted before it completes — by user request, by network failure, by loss of telephone connection, etc.



For purposes of discussion, assume that your workspace, *talon\_dvt\_mary*, has a current update level of 84, that the highest transaction in the repository is 109, and that 45 files in your workspace would be involved in a complete *Update*. You can monitor transaction levels using the CLI command *show wspaces*:

- Before the update the output of *show wspaces* might include:

```
talon_dvt_mary      c:\wks\talon_dvt      xlnr      13 84 84 1 0
```

The first **84** indicates the workspace's target update level; the second **84** indicates the current update level.

- The *Update* command sets the target update level to **109**, then proceeds. If *Update* processes all files successfully, it raises the current update level to the target:

```
talon_dvt_mary      c:\wks\talon_dvt      xlnr      13 109 109 1 0
```

But if the *Update* does not complete successfully, the target update level remains unchanged. A subsequent *show wspaces* reveals that the target update level differs from current update level:

```
talon_dvt_mary      c:\wks\talon_dvt      xlnr      13 109 84 1 0
```

The differing update levels — target vs. current — is the telltale sign that the most recent *Update* did not complete successfully.

## Incomplete Update: Command Interrupted

Consider the case in which *Update* was interrupted — say, after it had processed 29 out of the 45 files to be updated. When a subsequent *Update* command is issued:

- The checksum of the 29 files will succeed, because those files are already at the target transaction level.
- The checksum of the 16 files will fail, because those files are still at the current update transaction level.

(See Step 5 above for a discussion of the checksum process.)

## Incomplete Update: Checksum Failure

Now consider the case of an incomplete *Update*, due to one or more “crc mismatch” errors. Suppose that only 42 out of 45 files are updated, because 3 files fail the checksum match. You must fix the problem before issuing another *Update*.

If those three files had been overwritten by mistake, you can restore the proper versions using the *Revert to Backed Version* command (CLI: *purge*). Then, a second *Update* brings the new versions of those 3 files into the workspace.

## Performing the “Fixup” Update

When it begins executing an *Update* command, AccuRev determines whether the preceding update of the workspace completed successfully or not:

- The ***Update*** completed successfully if the target and current update levels are the same.
- The ***Update*** was incomplete if the target and current update levels differ.

If the preceding update was incomplete, AccuRev performs two updates at once. First, it performs a “fixup” update that completes the preceding update; then it performs an additional update (if necessary), to process changes made to the backing stream after the incomplete update. During the “fixup” update, AccuRev avoids the unnecessary work: it does not retransfer files that were successfully delivered to the workspace during the incomplete update.

Example:

- Your workspace’s current update level is 84, and the highest transaction in the repository is 109.
- You issue an ***Update***, but it fails to complete. At this point, the workspace’s target update level is 109, but its current update level is still 84.
- You wait until after lunch break to reissue the ***Update*** command. At this point, the highest transaction in the repository is 137.
- AccuRev performs a “fixup” update, which brings the current update level to the original target update level, 109. Then, it advances the target update level to 137 and performs another update. If this update succeeds, it advances the current update level to 137.

## Using a Trigger to Maintain a Reference Tree

Reference trees allow you to have a physical copy of the most recent sources for a stream. They are available for reference, thus the name reference tree. Snapshots never change, so they only need to be updated once using **update -r** and then you can forget about them.

To create a reference tree, use the **mkref** command. To keep a reference tree up to date with its associated stream, you need to run **update** on the reference tree every time versions are promoted to the stream.

AccuRev supplies the following trigger scripts to automate this procedure:

### **server\_post\_promote.pl**

A general-purpose script, which can be used to perform various tasks after completion of every **promote** command. In this case, we're going to have it call the **update\_ref.pl** script.

### **update\_ref.pl**

A script that invokes the **update** command to update the files in a reference tree. On a UNIX/Linux machine, this script must be setUID-root.

The indirection is necessary for security purposes.

To enable the automatic updating of one or more reference trees, follow these steps:

1. Make sure the following Perl scripts are installed in some directory on the search path of the AccuRev Server process's user identity:

```
server_post_promote.pl
update_ref.pl
```

See *Operating-System User Identity of the Server Processes* on page 9 of the *AccuRev Administrator's Guide*.

2. Edit both the **server\_post\_promote.pl** and **update\_ref.pl** scripts, and follow the step-by-step instructions contained within them.
3. Windows only: convert the Perl scripts to Windows batch files:

```
pl2bat server_post_promote.pl
pl2bat update_ref.pl
```

4. Tell AccuRev to run the **server\_post\_promote** script after every **promote** command:

```
accurev mktrig server-post-promote-trig server_post_promote.pl (UNIX/Linux)
accurev mktrig server-post-promote-trig server_post_promote (Windows)
```

For more information, see the descriptions of **mkref**, **mktrig**, and **show triggers** commands.



# Notes for CVS Users

This note contains information that will be helpful for CVS users who are migrating to AccuRev.

## AccuRev Workspaces vs. CVS Sandboxes

Each directory in a CVS sandbox has a subdirectory named **CVS**. This subdirectory stores metadata: where the versions were checked out from and the version number of each file. Only these directories record the relationship between files in the sandbox and the repository. If you move a sandbox, CVS doesn't care because you are simultaneously moving the **CVS** subdirectories.

With AccuRev, the relationship between a workspace and the repository is tracked by the AccuRev Server. No metadata is stored in the workspace itself. AccuRev tracks the client machine where each workspace resides and the pathname of its top-level directory. If you move a workspace to a different location, you must inform AccuRev of the move using the **chws** command.

## Common Operations

This section lists common version-control operations, and describes how to perform them with CVS, with the AccuRev CLI, and with the AccuRev GUI.

### Obtaining a copy of the source files

#### CVS

```
cvs checkout <module>
```

#### AccuRev CLI

```
accurev mkws -w <workspace-name> -b <backing-streamname> -l <workspace-location>
```

#### AccuRev GUI

*File > New > Workspace*

### Placing files under version control

#### CVS

```
cvs add <file(s)>
```

#### AccuRev CLI

```
accurev add <file(s)>
```

#### AccuRev GUI

Select files, right-click, *Add to Depot*

## Bringing others' changes into your workspace/sandbox

### CVS

1. `cvs update -dP`
2. Edit any merge conflicts.

### AccuRev CLI

1. `accurev update`
2. `accurev merge -o` (edit any merge conflicts for each file)

### AccuRev GUI

1. Click Update button.
2. Choose Overlap search.
3. Select files, right-click, *Merge*.

## Saving your changes

### CVS

`cvs commit`

### AccuRev CLI

`accurev keep -m`  
`accurev promote -k`

### AccuRev GUI

1. Choose Modified search.
2. Select files, right-click, *Keep*.
3. Choose Kept search.
4. Select files, right-click, *Promote*.

## Finding the history of files

### CVS

`cvs history [ <file(s)> ]`

### AccuRev CLI

`accurev hist [ <file(s)> ]`  
... or ...  
`accurev hist -a`

## AccuRev GUI

1. Select file, right-click, *History* > *Show History*.  
... or ...
1. *Admin* > *Depots*
2. Right-click depot, *History*.

## Finding the status of files in your workspace/sandbox

### CVS

cvs status <file(s)>

### AccuRev CLI

accurev stat <file(s)>  
... or ...  
accurev files <file(s)>

### AccuRev GUI

Automatically displayed in File Browser

## Removing files

### CVS

1. cvs remove <file(s)>
2. cvs commit

### AccuRev CLI

1. accurev defunct <file(s)>
2. accurev promote <file(s)>

### AccuRev GUI

1. Select files, right-click, *Defunct*.
2. Select files, right-click, *Promote*.

## Reverting changes to files

### CVS

cvs unedit <file(s)>

### AccuRev CLI

accurev purge <file(s)>

## AccuRev GUI

Select files, right-click, *Revert to > Backed Version*.

... or ...

Select files, right-click, *Revert to > Most Recent Version*.

## Moving files

### CVS

1. `cp <old-name> <new-name>`
2. `cvs remove <old-name>`
3. `cvs add <new-name>`

### AccuRev CLI

1. `accurev move <old-name> <new-name>`
2. `accurev promote <new-name>`

### AccuRev GUI

1. Select file, right-click, *Rename*.
2. Select file, right-click, *Promote*.

## Checking out files to edit

### CVS

`cvs edit <file(s)>`

### AccuRev CLI

not necessary; just start editing the file

### AccuRev GUI

not necessary; just start editing the file with right-click, *Edit*.

## Comparing versions of files

### CVS

`cvs diff -r <rev1> -r <rev2> <file>`

### AccuRev CLI

`accurev diff -v <rev1> -V <rev2> <file>`

### AccuRev GUI

1. right-click file, *History > Browse Versions*.
2. right-click version, *Diff Against > Other Version*, click other version



# Version Control of Namespace-Related Changes

AccuRev SCM includes both management of changes to the *contents* of files and changes to the *pathnames* of files and directories (folders). During the course of development — and in particular, during periodic “refactoring” of the source code base — developers may make several kinds of namespace-related changes to the pathnames of version-controlled elements:

- Changing the names of files (for example, from **framework.java** to **gizmo\_arch.java**)
- Changing the names of directories (for example, from **src** to **gizmo\_src**)
- Moving files and directories to different locations in the source tree (for example, moving file **commands.java** from directory **gizmo\_src** to a subdirectory named **gizmo\_src/lib**)

AccuRev records each change to the pathname of a file or directory element as a new version of that element. As with content changes, all such namespace-related changes originate in workspaces, and are subsequently promoted up the stream hierarchy.

Version 3.8 introduced significant improvements to the handling of namespace-related changes. These improvements make AccuRev more flexible and intuitive, and they reduce the likelihood of creating “twins” unintentionally.

## Twin Elements and Stranded Elements

The improvements affect these areas of namespace-related functionality:

- **Handling of “twin” elements** — Two or more distinct elements are described as twins if they have the same pathname within a depot. (Some SCM environments use the term “evil twins”; we won’t make that judgment.) AccuRev now tracks element names consistently, provides better information about the existence of twins, and provides tools for resolving unintended twin situations.
- **Detection of “stranded” elements** — an element is stranded in a particular workspace or stream if (1) it is active in that workspace or stream, but (2) does not have a pathname in that workspace or stream. AccuRev includes significant improvements to the detection and reporting of stranded elements.

These two areas are related. If a stream contains a set of twins at a particular pathname, only one of those elements is visible at that pathname, for most purposes. The other twin(s) are stranded.

The following sections describe the changes to namespace-related functionality in detail.

### Preventing Creation of Twins in Workspaces

Two checks performed by the **Add to Depot** command (CLI command: **add**) help to prevent twins from being created:

- If the user’s workspace currently contains a defunct element at the same pathname, the **Add to Depot** command is cancelled.

- If an element with the same pathname currently appears in the workspace’s parent stream — and the element does *not* have **(defunct)** status in the parent stream — the *Add to Depot* command is cancelled. If the element is **(defunct)** in the parent stream, an *Add to Depot* command succeeds.

In previous releases, the first check was performed for renamed elements as well as for defuncted elements. This check no longer takes place — see *Ability to Reuse an Element Name after a Rename Operation* below.

## Reporting of Twins in Dynamic Streams

The checks described in the preceding section apply to workspaces only, not to dynamic streams. There are various ways to create twins in dynamic streams — for example:

Defunct an existing element in a workspace, and promote the change to the parent stream. Then create a new element at the same pathname (in the same workspace or in a sibling workspace), and promote the new element to the parent stream. The parent stream now contains two elements at the same pathname — one is defunct, the other is “live”.

If a set of two or more twins exists in a dynamic stream (or snapshot), the File Browser shows only the one “live” element in the folders view and in the results of a Default Group or Defunct search. (If *every* element in a set of twins has **(defunct)** status, the most recently defuncted element is deemed to be “live”.)

In this situation, all the other twins in the set have **(stranded)** and **(twin)** status. These elements are displayed by the File Browser’s Stranded search, and by the AccuRev CLI command *stat -i*:

```
> accurev stat -s tin_dvt -i
\\.doc\new.doc e:20 tin_dvt\2 (4\2) (defunct) (member) (stranded) (twin)
```

For more information, see *Detection of All Stranded Elements, Including “Twins”* below.

## Ability to Reuse an Element Name after a Rename Operation

AccuRev now offers improved support for “refactoring” operations over previous releases. Now, all element renamings and directory hierarchy overhauls can be completed and tested in the developer’s workspace, before any changes are promoted to the parent stream.

This flexibility stems from the fact that after an element *Rename* operation (CLI command: *move*), the element’s former name becomes available for reuse immediately. (Previous releases “reserved” the former name, in case the change was purged from the workspace — see *When a Purge Operation Causes an Element to Disappear* below.)

Here’s a simple refactoring example:

```
> accurev move brass.h util.h
Moving \.\src\brass.h to \.\src\util.h

> copy c:\temp\temp_new_brass.h .\brass.h
1 file copied

> accurev add brass.h
```

```
Added and kept element \.\src\brass.h
```

```
> accurev promote util.h
Validating elements.
Promoting elements.
Promoted element \.\src\util.h
```

```
> accurev promote brass.h
Validating elements.
Promoting elements.
Promoted element \.\src\brass.h
```

Note that **util.h** (formerly named **brass.h**) must be promoted first, to “free” the name **brass.h** in the parent stream. Then the new element named **brass.h** can be promoted.

### When a Purge Operation Causes an Element to Disappear

The preceding section describes new flexibility for refactoring. But it does introduce a complication: what happens if you rename an element, create a new element at the same pathname, then invoke ***Revert to Backed*** (CLI command: ***purge***) on the renamed element?

The renamed element cannot revert to its old pathname, because there’s a new element at that pathname. Accordingly, the original element simply disappears from your workspace. This element does *not* assume (**stranded**) status — the purge operation makes the element inactive in the workspace, and (**stranded**) status applies only to active elements.

Note that at this point, your workspace contains a new element at the given pathname, and the parent stream contains the original element at that pathname. Attempting to promote the new element would produce a “name already exists in parent stream” error. These steps remove the original element from the parent stream: (1) defunct the original element in the workspace, using ***defunct -e***; (2) promote this change to the parent stream.

### Detection of All Stranded Elements, Including “Twins”

The File Browser’s Stranded search, and equivalently, the AccuRev CLI command ***stat -i*** now detect all known cases of stranded files:

- Defunct elements in same workspace/stream as a “live” element at the same pathname (“twins”)
- Elements located in a directory that is, itself, stranded
- Elements located in a directory that is defunct
- Elements located in a directory that is excluded from the workspace/stream
- Elements located in a directory that has been purged from the workspace/stream
- Elements with a “pathname cycle”

Stranded files are reported with the status flag (**stranded**). If a stranded file happens to be a twin of another element, it is also reported with the status flag (**twin**).

The final case, “pathname cycle”, occurs when two sibling workspaces make contradictory changes to the depot’s directory hierarchy, then promote the changes to the common parent stream. For example, one workspace might move directory **src** under directory **util**, while another workspace moves **util** under **src**. When both the changes are promoted to the parent stream, AccuRev won’t be able to determine the correct pathname for these directories and the elements under them. The two directories assume (**stranded**) status, and the elements under these directories become inaccessible.

### Ability to Operate on Stranded Elements Using Element-IDs

The CLI now provides improved tools for relieving situations involving stranded elements. The **move**, **defunct**, and **undefunct** commands now support the **-e** option, which enables you to specify an element by its element-ID, rather than by its pathname. This is necessary for situations in which the desired element does not *have* a pathname in the workspace or stream.

See *Handling Stranded Elements* on page 40.

### More Sophisticated Analysis of Namespace-Related Changes

AccuRev now distinguishes between these two changes to the pathname of an element:

- Renaming of the directory in which the element resides. This is not considered a change to the element itself; it is a change to the parent-directory element.
- Moving of the element from its current directory to another directory in the depot. This *is* considered a change to the element itself (and not a change to either of the parent-directory elements).

For example, suppose that file element **commands.java** resides in directory **cmd\_interface**. A colleague changes the name of the directory to **cli** in her workspace, then promotes the change to the parent stream. When you update your workspace, the pathname of the file changes from **.../cmd\_interface/commands.java** to **.../cli/commands.java**, as a “side effect” of the change to the parent directory. Note that this is not a change to the **commands.java** file element itself.

On the other hand, if the colleague moves file **commands.java** to another directory, say **.../cmd\_interface/utills/commands.java**, this *is* a change to the file element. When you update your workspace, the pathname of the file changes accordingly (unless you have made a namespace-related change to the file, in which case a merge is required).

AccuRev implements this scheme by tracking an element’s parent directory by its element-ID (which never changes), rather than by its name (which can change, and can vary from stream to stream).

### Change to Merge Algorithm for Namespace-Related Changes

The algorithm used by the **Merge** command (both in the GUI and the CLI) now uses the more sophisticated analysis described in the preceding section. **Merge** may perform two separate namespace-related steps:

- **Element name merge** — required when the simple name of the element being merged differs in the two contributor versions.

- **Path merge** — required when the parent directory of the element being merged differs in the two contributor versions.

Often, either or both of these steps will be performed automatically by **Merge**. If only *one* of the two contributors differs from the versions' closest common ancestor, then that contributor's change is applied automatically.

(Note that **Merge** may also need to perform a third step — a content merge — for a file element.)

The following example of the CLI **merge** command involves both kinds of namespace-related changes: (1) a file's simple name has been changed by two users, **john** and **mary**; (2) each user has moved the file to different sibling directory.

```
> accurev merge file01.mary
Current element: \.\dir02\sub03\file01.mary
most recent workspace version: 4/2, merging from: 5/5
common ancestor: 5/3
```

*Both "path" and "element name" conflicts must be resolved manually, but the contributors' contents can be merged automatically.*

```
Path merge will be required.
Element name merge will be required.
Automatic merge of contents successful. No merge conflicts in contents.
```

```
Path conflict for \.\dir02\sub03\file01.mary
```

*John moved the file from directory sub02 to directory sub01 (which has element-ID 68).  
Mary moved the file from directory sub02 to directory sub03 (which has element-ID 82).*

```
Resolve path conflict by choosing path from:
```

```
(1) common ancestor: \.\dir02\sub02\ [eid=75]
(2) backing stream : \.\dir02\sub01\ [eid=68]
(3) your workspace : \.\dir02\sub03\ [eid=82]
```

```
Actions: (1-3) (s)kip (a)bort (h)elp
```

```
action ? [3] 2
```

*John renamed the file from file01.txt to file01.john.  
Mary renamed the file from file01.txt to file01.mary.*

```
Resolve name conflict by choosing name from:
```

```
(1) common ancestor: \.\dir02\sub01\file01.txt
(2) backing stream : \.\dir02\sub01\file01.john
(3) your workspace : \.\dir02\sub01\file01.mary
```

```
Actions: (1-3) (s)kip (a)bort (h)elp
```

```
action ? [3] 3
```

*The content merge is performed automatically.*

```
Actions: keep, edit, merge, over, diff, diffb, skip, abort, help
```

```
action ? [keep]
```

```
Moving \.\dir02\sub03\file01.mary to \.\dir02\sub01\file01.mary  
Kept element \.\dir02\sub01\file01.mary
```

## Handling Stranded Elements

As described above, an AccuRev workspace or stream can contain one or more elements that are stranded. An element is stranded in a particular workspace or stream if it is a member of the default group, but cannot be accessed because there is no pathname to the element in that workspace or stream. An element can be stranded in one stream but not be stranded in other streams.

In the AccuRev GUI, stranded elements are listed in the File Browser's **Stranded** filter. In the CLI, the command **stat -i** lists stranded elements. A stranded element is listed by its element-ID, along with a pathname that was once (but is not currently) valid in that stream.

The sections below describe the ways in which elements can become stranded, along with procedures for handling each situation.

*Defunct element obscured by element with same name*

*Elements under a defunct parent*

*Elements under an excluded parent*

*Dangling directory elements*

*Elements under a non-existent (purged) parent directory*

*Elements under a stranded parent directory*

*Active element refers to a purged version*

## Defunct element obscured by element with same name

This occurs in the parent stream of two workspaces if:

- The user in workspace #1 defuncts an element, then promotes this change to the parent stream.
- The user in workspace #2 updates the workspace (to incorporate the defuncting), creates a new element with the same name, then promotes the new element to the parent stream.

At this point, the defuncted element is stranded in the parent stream. It cannot be promoted to the “grandparent” stream by name, because it doesn’t have a name in the parent stream. The new element cannot be promoted to the grandparent stream at all, because the name in the grandparent stream belongs to the defuncted element.

Note: through repeated *add-promote-defunct-promote* cycles, it’s possible to have multiple elements with defunct status in the parent stream, all of which were created at the same pathname.

## Resolving the Situation

To get the defuncted element out of the way, promote it by element-ID to the grandparent stream: *promote -e <eid> -s <parent\_stream>*.

To recover the defuncted element in workspace #1, use *undefunct -e <eid>* on the defuncted element. This has the side effect of making the new element inaccessible in workspace #1. Depending on your needs, use *defunct -e* or *move -e* on the new element.

## Elements under a defunct parent

This occurs in the parent stream of two workspaces if:

- The user in workspace #1 defuncts a directory element, then promotes this change to the parent stream.
- The user in workspace #2 modifies a file element in that directory, then keeps and promotes it to the parent stream.

At this point, the file element is stranded in the parent stream. In addition, the user in workspace #1 cannot access the file element by name.

## Resolving the Situation

To propagate the file element’s change to the grandparent stream, promote it by element-ID: *promote -e <eid> -s <parent\_stream>*.

To access the file’s contents, use its element-ID: *cat -e <eid> -v <version-ID>*.

The only way to work with the file element in workspace #1 is to first *undefunct* the directory, which makes the file visible again.

## Elements under an excluded parent

This occurs in the parent stream of two workspaces if:

- The user in workspace #1 modifies a file element in that directory, then keeps and promotes it to the parent stream.
- The user in workspace #2 sets a rule that excludes a directory element from the parent stream (*excl -s <parent-stream> <directory-name>*).

At this point, the file element is stranded in the parent stream. In addition, the user in either workspace cannot access the file element by name (after updating the workspace).

## Resolving the Situation

To propagate the file element's change to the grandparent stream, promote it by element-ID: ***promote -e <eid> -s <parent\_stream>***.

To access the file's contents, use its element-ID: ***cat -e <eid> -v <version-ID>***.

The only way to work with the file element in either workspace is to first remove the exclude rule (***clear*** command) from the parent stream, and then update the workspace. This makes the file visible again.

## Dangling directory elements

This contradictory situation — a particular directory seems to be both above and below another directory — occurs in the parent stream of two workspaces if:

- The user in workspace #1 moves directory A under directory B, then promotes directory A.
- The user in workspace #2 moves directory B under directory A, then promotes directory B.

At this point, both directories are stranded in the parent stream. An update of workspace #1 causes directory B to be removed; an update of workspace #2 causes directory A to be removed.

## Resolving the Situation

The only way to untangle this knot of inconsistency is to checkout (***co*** command) a previous version of each directory that has the “correct” (that is, consistent with the other directory) pathname, then promote these old versions to the parent stream.

The simplest way to do this is to specify the transaction that created the directory at its correct pathname: ***co -t <add-transaction-number>***. But this method can be “messy” if the ***add*** transaction also created other elements, such as the files within the directory.

Another method is to use a workspace under a time-based stream to see the relevant directories with their correct pathnames. Checkout the “old” directory versions, promote these versions from the workspace to the time-based stream, then use ***promote -s <time-based-stream> -S <parent-stream>*** to promote to the parent stream.

Note: with either method, you'll probably need to use the ***-O*** option to the ***promote*** command, in order to avoid the need to merge the “old” directory versions.

## Elements under a non-existent (purged) parent directory

This occurs in the parent stream of a workspace if:

- The user creates a new directory and file within the new directory, and promotes both new elements to the parent stream.
- The user purges (GUI: ***Revert to Backed***) the new directory from the parent stream.

At this point, the new file is stranded in the parent stream.



## Resolving the Situation

You cannot propagate the file element's change to the grandparent stream, because the new directory never existed in that stream.

To access the file's contents, use its element-ID: *cat -e <eid> -v <version-ID>*.

The only way to work with the file element is to first checkout (*co* command) the version of the directory that was originally created in the workspace. The simplest way to do this is to specify the transaction that created the directory: *co -t <add-transaction-number>*. But this method can be “messy” if the *add* transaction also created other elements.

Another method is to use a workspace under a time-based stream to see the directory before it was purged from the parent stream. Checkout the directory, promote it from the workspace to the time-based stream, then use *promote -s <time-based-stream> -S <parent-stream>* to promote to the parent stream.

Note: with either method, you'll probably need to use the *-O* option to the *promote* command, in order to avoid the need to merge the “old” directory version.

## Elements under a stranded parent directory

To access an element under a stranded parent directory, restore the accessibility of the parent, as described in the sections above. This restores the accessibility of the element in question.

## Active element refers to a purged version

This occurs when an element (file, directory, or link) is active in a dynamic stream. The dynamic stream's virtual version is a reference to a version that has been purged, so that there is no active version of the element in a higher-level stream (and no version in the depot's root stream).

The only known scenario involves promoting an active element from a dynamic stream to one of its child streams, then purging the element from the original stream.



# Notes on Cross-Links

This note clarifies and supplements the basic documentation of AccuRev's cross-link feature, introduced in Version 4.5.

## Cross-Link Direction and Terminology

A cross-link is created in a workspace by the *Include from Stream* command (CLI: *incl -b*). The command name implies that a connection is being established *from* a specified backing stream *to* the workspace. But an existing cross-link is listed by the CLI command *lsrules* like this:

```
xlink <pathname> from <workspace> to <backing-stream>
```

That is, the direction of the cross-link “arrow” is the opposite of the direction implied by the “include from” command name. When describing a cross-link, we use this terminology:

- The workspace (or stream) where the cross-link has been created is the cross-link's source stream.
- The designated backing stream is the cross-link's target stream.

In CLI messages, “cross-link” is abbreviated to “xlink”.

## Cross-Links and Stream Namespaces

Each AccuRev stream (including snapshot streams and workspace streams) provides a namespace: a set of pathnames to some or all of the depot's elements. For example:

```
\\.doc
\\.src
\\.tools
\\.doc\chap01.doc
\\.doc\chap02.doc
\\.src\commands.c
\\.src\topaz.c
\\.src\topaz.h
\\.tools\cmdshell
\\.tools\perl
\\.tools\python
\\.tools\tools.readme
\\.tools\cmdshell\bash
\\.tools\cmdshell\csh
\\.tools\cmdshell\bash\end.sh
\\.tools\cmdshell\bash\start.sh
\\.tools\cmdshell\csh\end.csh
\\.tools\cmdshell\csh\start.csh
\\.tools\perl\add_cr.pl
\\.tools\perl\remove_cr.pl
```

```
\\.tools.python.setup.py
\\.tools.python.vars.py
```

Since this set of depot-relative pathnames defines a hierarchy, it's often clearer to list the pathnames component-by-component, like this:

```
\\.
  doc
    chap01.doc
    chap02.doc
  src
    commands.c
    topaz.c
    topaz.h
  tools
    tools.readme
    ...
```

To locate an element, AccuRev interprets its specified pathname component-by-component (just like the operating system does). The cross-links facility provides a way to make AccuRev switch namespaces in the middle of the pathname-interpretation process.

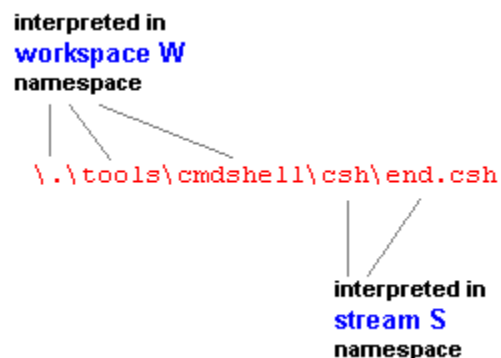
Note:

For example, consider this pathname:

```
\\.tools.cmdshell.csh.end.csh
```

And suppose you've created a cross-link at subdirectory **cmdshell**, with workspace **W** as the source stream and stream **S** as the target stream. AccuRev will process the pathname, component-by-component, as illustrated here:

- Pathname components up to *and including* the cross-linked component, are interpreted in the original (source stream) namespace.
- Additional pathname components, if any, are interpreted in the new (target stream) namespace.



Note that in workspace **W**, you continue to access the cross-linked element, subdirectory **cmdshell**, through its “local” name in the workspace’s namespace. It’s quite possible (but you don’t need to know) that this element has a different name — even a different pathname — in the target stream:

```
\\.tools.shell_scripts
\\.tools.common/scripts\
\\.scripting
... etc.
```

Pathname components below “cmdshell” are interpreted in the namespace of stream **S**, the target stream. For example, if script **end.csh** has been renamed in stream **S** to **topaz\_exit.csh**, then that’s the name you must use in workspace **W**, as well:

```
\\.tools\cmdshell\csh\topaz_exit.csh
```

The File Browser and the CLI commands **stat** and **files** make this namespace-switching transparent: AccuRev shows you the element names and pathnames that will enable you to access the data from your current workspace or stream context.

### Source Stream: Workspace vs. Dynamic Stream

The example in the preceding section uses a workspace as the “source stream”. The same pathname-interpretation principles apply if the source stream is a dynamic stream.

But the basic difference between workspace streams and dynamic streams affects the way cross-links work in them:

- In a dynamic stream, the **Include from Stream** command incorporates all changes from the target stream immediately. This reflects the fact that a dynamic stream inherits versions from its backing stream automatically and instantly.
- In a workspace, the **Include from Stream** command respects the workspace’s update level. That is, it incorporates only those changes that occurred in the target stream before the workspace’s most recent update. A subsequent **Update** command will bring in the more recent changes from the target stream.

Example: to see how cross-links work with a workspace’s update level, suppose that the following changes have been made in stream **topaz\_mnt**:

- directory element **\\.tools\cmdshell\cmd** has been **Defunct**’ed
- directory element **\\.tools\cmdshell\csh** has been renamed to **\\.tools\cmdshell\c\_shell**
- file element **\\.tools\cmdshell\c\_shell\start.csh** has been edited

You use the **Include from Stream** command to create a cross-link from your workspace to stream **topaz\_mnt**, at pathname **\\.tools\cmdshell**. The immediate change to your workspace depends on its update level:

- If the changes in stream **topaz\_mnt** occurred after your workspace’s most recent update, you won’t see the changes immediately in your workspace: directory **cmd** will still exist, directory **csh** won’t be renamed to **c\_shell**, and you won’t see the edits to file **start.csh**. But the status of these elements includes the **(stale)** indicator, showing that the changes are in the backing stream, waiting to be incorporated:

```
\\.tools\cmdshell\cmd    topaz\1 (9\1) (backed) (xlinked) (stale)
\\.tools\cmdshell\csh    topaz\1 (9\1) (backed) (xlinked) (stale)
\\.tools\cmdshell\csh\start.csh  topaz\2 (9\4) (backed) (xlinked) (stale)
```

At this point, performing an **Update** will bring the changes into the workspace.

- If the changes in stream **topaz\_mnt** occurred before your workspace’s most recent update, all those changes will be brought into the workspace immediately.

The procedure in the first bulleted paragraph can be described as “Include then Update”; the second bulleted paragraph’s case can be described as “Update then Include”. The final result is the same in both cases: the changes to the cross-linked elements in their new backing stream are incorporated into your workspace. We consider the second case to be an AccuRev best practice:

***Best Practice:***

***Update*** your workspace before performing an ***Include from Stream*** command

If you ***Update*** first, other backing-stream changes won’t be “mixed in” with the ***Include from Stream*** changes during the next workspace update. Moreover, fully establishing the link from your workspace to the target stream will involve a single step (Include), rather than two steps (Include then Update).

Note: because it respects — but does not change — your workspace’s update level, ***Include from Stream*** more closely resembles the ***Populate*** command than the ***Update*** command.

### Multiple Cross-Links: Chaining

AccuRev can traverse two or more cross-links in the same pathname. For example, you might use this pathname in workspace **W**:

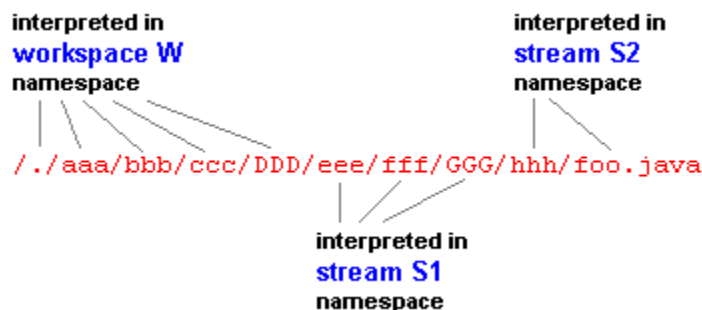
```
./aaa/bbb/ccd/DDD/eee/fff/GGG/hhh/foo.java
```

And suppose there are two cross-links:

- At subdirectory **DDD**, a cross-link from workspace **W** to stream **S1**
- At subdirectory **GGG**, a cross-link from stream **S1** to stream **S2**

As AccuRev traverses the pathname component-by-component, it interprets the components as illustrated here. As it progresses down the pathname, AccuRev also traverses a “chain” of cross-links:

- start in workspace **W**, then ...
- cross-link to stream **S1**, then ...
- cross-link to stream **S2**



“Chaining” of cross-links can continue to any number of levels. The same principle applies repeatedly: a cross-linked pathname component is interpreted in the *source* stream’s namespace; subsequent non-cross-linked components are interpreted in the target stream’s namespace.

But you must take care when “chaining” cross-links in this way. It is possible to create ambiguous configurations, which AccuRev handles by removing the affected elements. See [Cross-Link Overlaps](#) on page 50.

A special case of cross-link chaining occurs when you create a configuration in which two or more cross-links occur at the *same* pathname component. For example, consider again this pathname:

```
\\.tools\cmdshell\csh\end.csh
```

And suppose there is a chain of two cross-links at the same pathname component:

- At subdirectory **cmdshell**, a cross-link from workspace **W** to stream **S1**
- At subdirectory **cmdshell**, a cross-link from stream **S1** to stream **S2**

In workspace **W**, the subdirectory will continue to have its “original” name, **cmdshell**. But the subtree under the subdirectory will come from the stream **S2** namespace. By extension, you could chain any number of cross-links at the **cmdshell** component: **W** > **S1** > **S2** > **S3** > **S4** ... As above, the directory retains its “original” name in the workspace, and the workspace sees the directory’s subtree as it exists in the final target stream.

## Double Vision: Seeing an Element Multiple Times in a Workspace

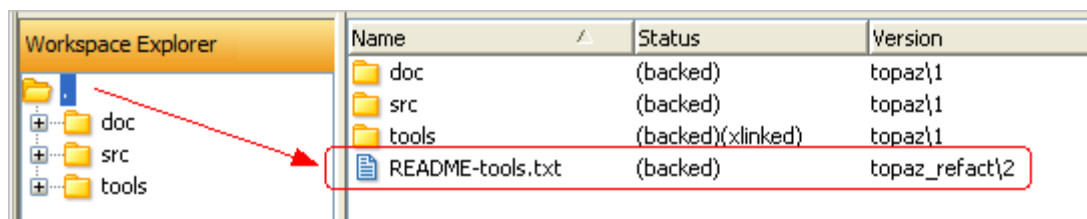
One consequence of AccuRev’s cross-link facility is that two (or more) different versions of the same element can appear at different pathnames in the same workspace or stream. We call this phenomenon double vision. This is not an error — at least, not from AccuRev’s perspective. Seeing the same element twice might be exactly what you intended, or it might signify that you’ve left some refactoring work unfinished.

Here’s an example: suppose you are tasked with doing some cleanup on the Topaz project’s development tree:

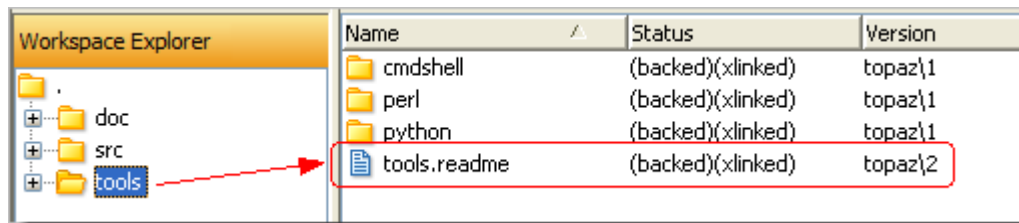
- Flatten out the subdirectories under **tools**.
- Move file **tools.readme** to the depot’s root directory, and rename it to **README-tools.txt**.
- Improve the source file comments in the **src** directory.

You perform this work in your workspace, named **topaz\_refact**. But when the dust settles, you find that the programs in the **tools** subdirectory no longer work. You are not sure whether the problem is in the **tools** directory or the **src** directory. So you decide to “back out” your refactoring of the **tools** directory, by cross-linking to the known-to-work version of the **tools** directory in snapshot stream **topaz\_2.3.9**.

Now, you have two different versions of the “README” element in your workspace! In your refactoring, you created a new version in your workspace, at pathname **\\.README-tools.txt**:



But your workspace now cross-links to the Release 2.3.9 version of the **tools** subdirectory, which contains the Release 2.3.9 version of the same element, at pathname `\\.tools\\tools.readme`:



This case of double-vision is clearly an error, reflecting the fact that your refactoring work is still ongoing. In other cases, you might want two (or more) versions of a commonly used source file, say **topaz.h**, to appear in a workspace. Perhaps several different versions of the file are required, in order to build different executables using that file. Version skew is the executables' other dependencies might mandate the different versions of **topaz.h**.

### Double Vision and the 'accurev name' Command

The *accurev name* command lists the pathname for a given element (specified by element-ID) in your workspace. It can also list the pathname for a specific version of an element, or the version in a specific stream:

```
accurev name -e 28
accurev name -v topaz_mnt -e 116
```

In a double vision situation, the *name* command can list all of an element's pathnames in a workspace or stream:

```
> accurev name -e 28 -v topaz_refact
\\.tools\\tools.readme
\\.README-tools.txt
```

### Cross-Link Overlaps

Section *Multiple Cross-Links: Chaining* on page 48 describes how a set of cross-links can define a "chain" of backing streams to be used at different components in a pathname:

```
/. /aaa/bbb/ccccc/DDD/eee/fff/GGG/hhh/foo.java
```

start interpreting  
pathname in  
**workspace W**

switch from  
**workspace W**  
to **stream S1**

switch from  
**stream S1**  
to **stream S2**

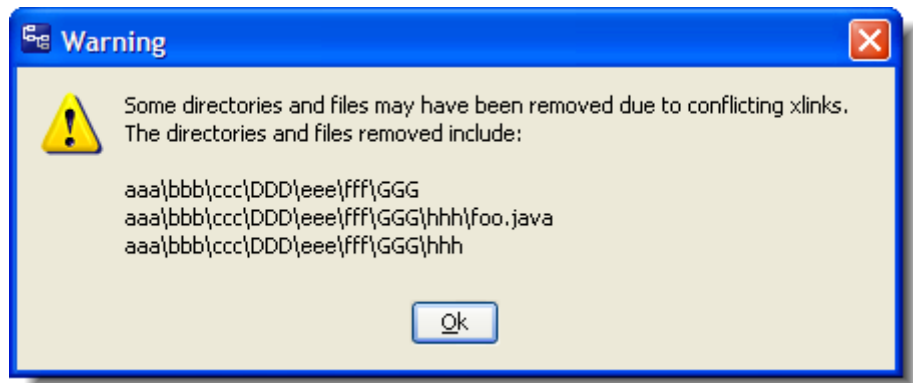
Chaining works correctly if each switch to the next link in the chain occurs at the same pathname component or at a lower component. But here's a situation that violates this rule:



`/. /aaa/bbb/ccc/DDD/eee/fff/GGG/hhh/foo.java`  
 |  
 start interpreting  
 pathname in  
**workspace W**  
 |  
 switch from  
**stream S1**  
 to **stream S2**  
 |  
 switch from  
**workspace W**  
 to **stream S1**

In this case, the second link in the cross-link chain (**S1** > **S2**) occurs at a *higher* pathname component, **DDD**, than the first link (**W** > **S1**, at component **GGG**). AccuRev recognizes this situation as a cross-link overlap.

When a workspace that has a cross-link overlap gets updated, AccuRev removes the subtree below the component where the first link was created.





# Notes on Promote-by-Issue

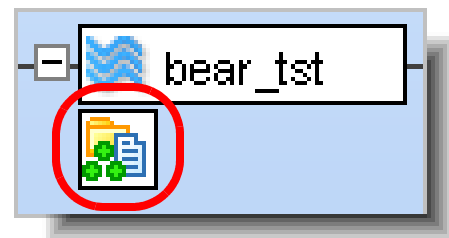
This technical note describes AccuRev's **promote-by-issue** feature, both reviewing the basic concepts and presenting several usage scenarios. This information supplements and updates the basic documentation of promote-by-issue in the AccuRev GUI's help screens. The most important points are:

- In some cases, promote-by-issue requires you to create an additional version of one or more elements, using the **Patch** or **Merge** command. You must create a new AccuWork issue, called a tracking issue, to enable AccuRev to account for the additional version(s).
- For certain usage scenarios, you must use the Change Palette in a way that might differ from your previous usage procedure.

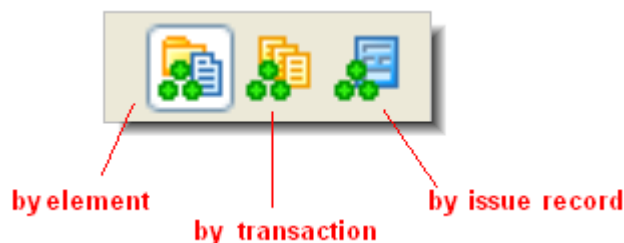
## Promote-by-Issue Basics

(This section paraphrases the information in section “Viewing a Stream's Current Development Activity” of the “Stream Browser” help topic for the AccuRev GUI.)

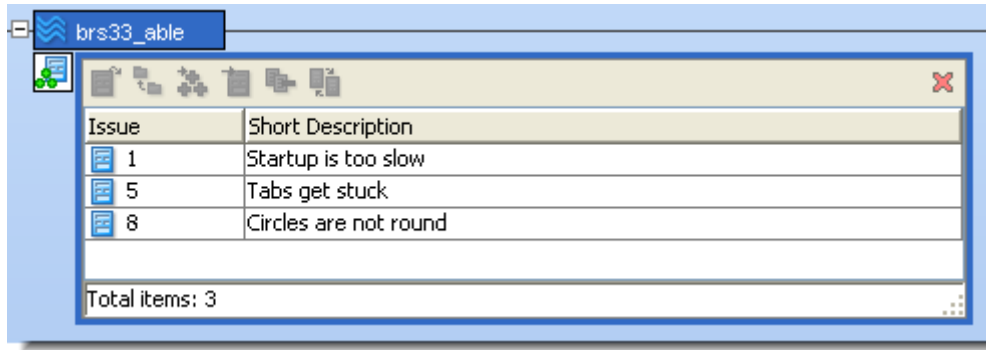
The Stream Browser can show the development activity currently taking place in each stream or workspace. A control below the stream or workspace opens or closes a subwindow that displays the details of the development activity. The activity details can appear in several ways — by element, by transaction, or by issue record.



Use the development-activity mode controls at the right side of the Stream Browser toolbar to determine how the activity details will be displayed. (The icons on the controls below the streams and workspace change accordingly.) You can change modes either before or after opening a development-activity subwindow.



When displaying a stream's activity by issue record, the subwindow displays the issue records that are in a particular stream:



(This display is the same as the results of a **Show Active Issues** or **Stream Diff by Issues** command. In this note, all the examples use the Stream Browser’s development activity subwindow, but you can perform promote-by-issue operations in these other contexts, also.)

Promote-by-issue involves selecting one or more issue records and performing any of the following operations:

- Invoking the **Promote** command from the subwindow’s toolbar or the context menu of the issue(s).
- Drag-and-drop’ing the issue record(s) onto another stream or workspace in the Stream Browser display. The destination can be the parent (backing) stream, another dynamic stream, or one of your workspaces. This invokes a **Promote** or **Send to Workspace** operation on all the elements in the selected issue record(s).

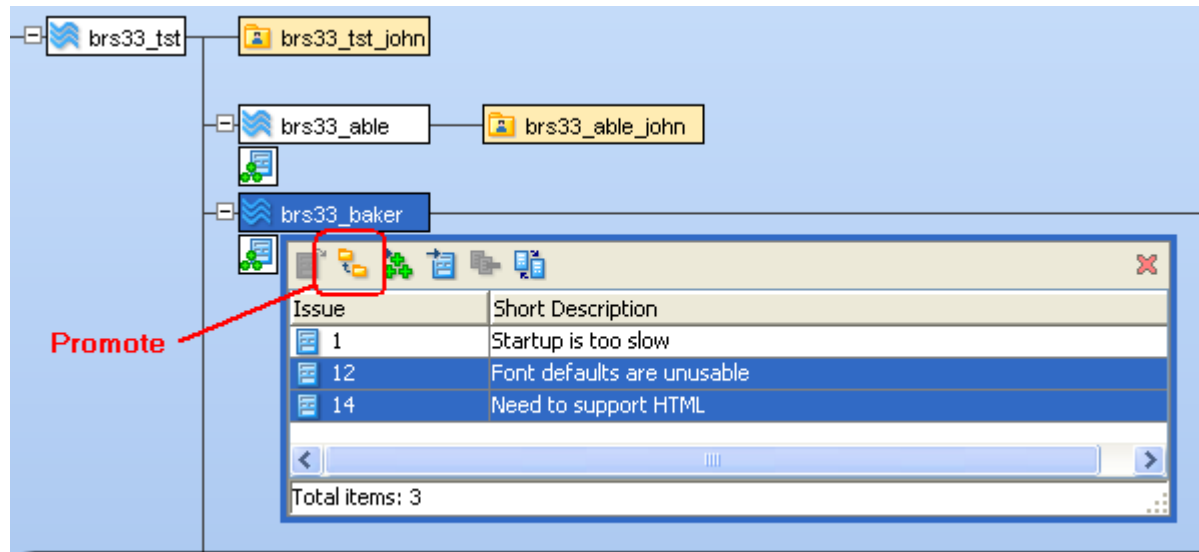
You can also drag-and-drop the subwindow control to another stream. This performs the **Promote** or **Send to Workspace** operation on all the issue records in the subwindow (even if only some of them are currently selected).

The following sections explore several promote-by-issue scenarios.

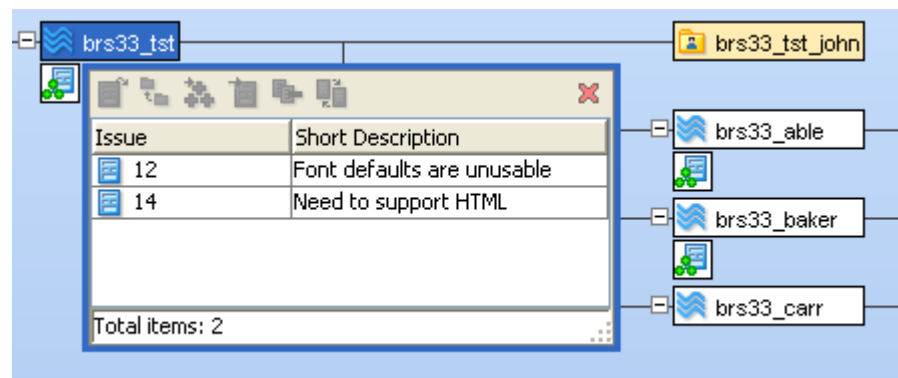
Note: in the remainder of this technical note, we shorten the term “issue record” to “issue”.

## Promoting Issues to the Parent Stream

In many cases, promote-by-issue is simple and easy. You promote one or more issues to the parent stream ...



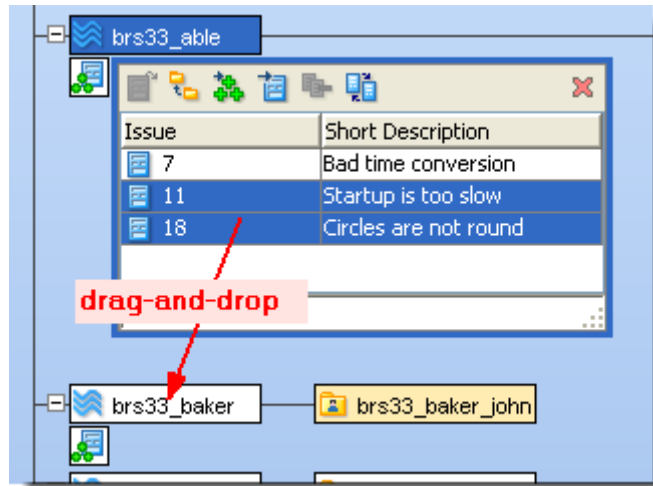
... and the issues simply move into the parent stream:



Note: if the **Promote** command fails with a “merge required” error, you must perform a merge for one or more elements in the source workspace (or a workspace below the source stream). When promoting the merged version(s), assign them to the same issues as the original version(s).


## Cross-Promoting Issues to a Non-Parent Stream — Simple Case

Similarly, in many cases cross-promoting from one dynamic stream to another dynamic stream proceeds without complication. First, you perform a drag-and-drop operation:











This populates the Change Palette with the versions in the selected issues. If the **Promote** command is enabled when you select all the elements, you can proceed to propagate all the issues (and their versions) to the destination stream:

Destination Stream:









Source Stream:





Element	Version	Basis Version	Trans.	Status	Count
 \.tools\cmdshell\csh\end.csh	6\2	6\1			1
 \.tools\cmdshell\csh\start.csh	6\2	6\1			1

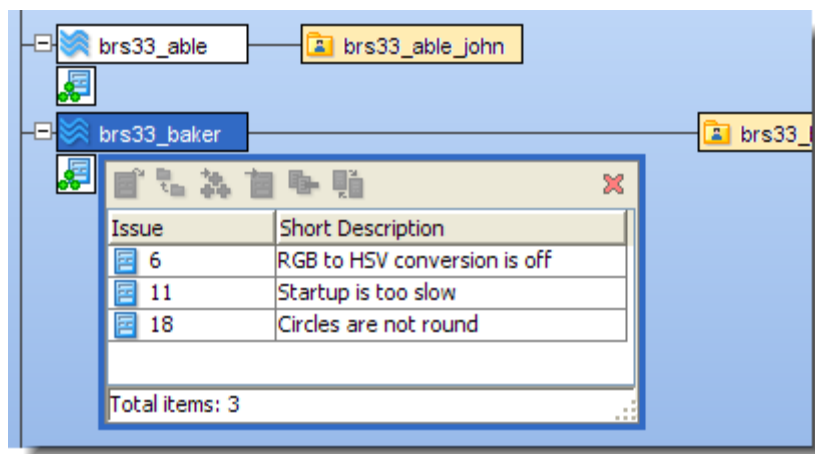
Destination Stream: brs33\_baker

Source Stream: brs33\_able



Element	Version	Basis Version	Trans.	Status	Count
 \\.\tools\cmdshell\csh\end.csh	6\2	6\1			1
 \\.\tools\cmdshell\csh\start.csh	6\2	6\1			1

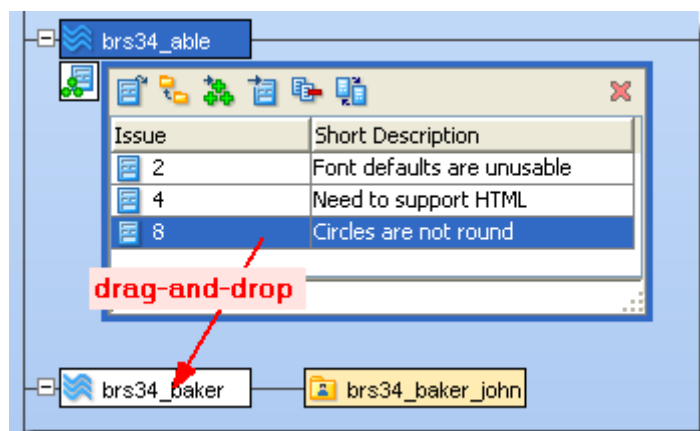
(In this case, issue #11 consists of element **start.csh**, and issue #18 consists of element **end.sh**.)  
Back in the Stream Browser, the promoted issues now appear in the destination stream:



## Cross-Promoting Issues to a Non-Parent Stream — Patch Required






In this scenario, your intention is the same as in the preceding one — to cross-promote the versions in one or more issues from one dynamic stream to another. Before proceeding, please see definition of the following terms in the *AccuRev Glossary* chapter of the *AccuRev Concepts Manual*: direct, indirect, incomplete, cyclical, and coalesce.

You start the same way, with a drag-and-drop operation from stream **brs34\_able** to stream **brs34\_baker**:






As before the elements in the issue(s)' change packages are loaded into the Change Palette. But in this scenario, one or more of the elements is listed with **(patch)** status:

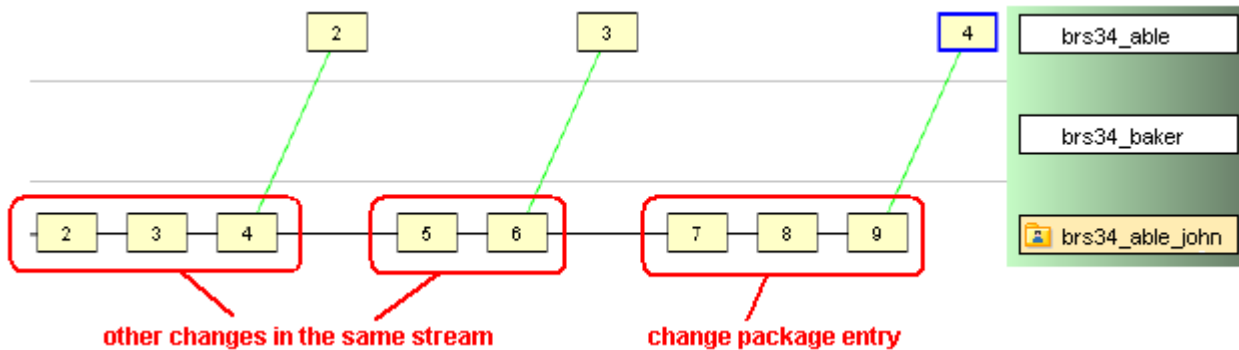
Destination Stream:



Source Stream:

Element	Version	Basis Version	Trans.	Status	Count
 \.\src\brs34.c	6\8	6\1			1
 \.\tools\cmdshell\bash\start.sh	6\9	6\6		(patch)	1
 \.\tools\tools.readme	6\5	6\1			1

This occurs when the element's change-package entry does not contain the complete set of changes to the element that (1) are in the source stream and (2) have not yet been promoted to the destination stream:



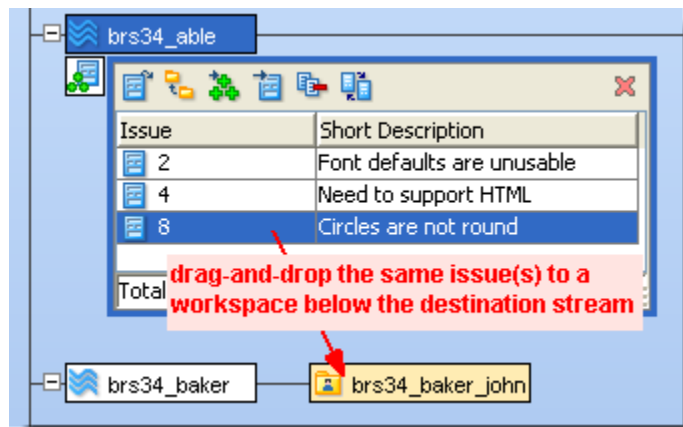
## Changing the Way You Use the Change Palette

In the situation described in the preceding section, you might be accustomed to dealing with the **(patch)**-status element separately from the other elements. But do not proceed in this manner! Instead, close the Change Palette tab without propagating any of the changes to the destination stream.

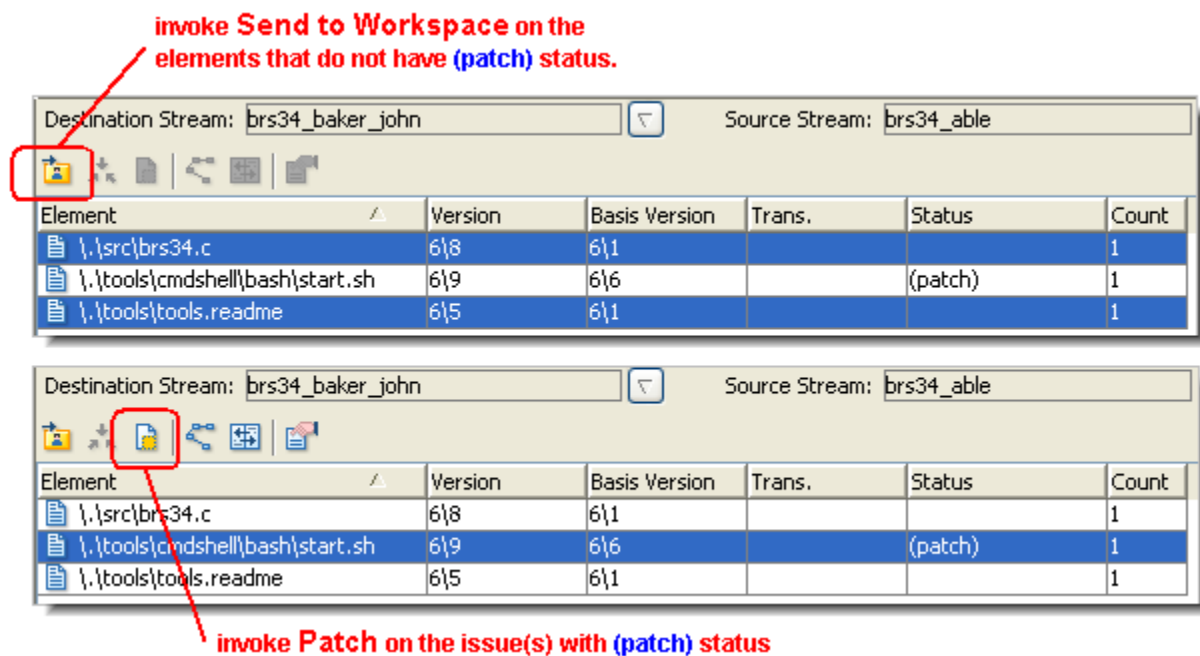
Note: What happens if you proceed to use the Change Palette in the traditional way, processing the **(patch)**-status element separately? See *If You Process Some Elements at the Stream Level, not the Workspace Level* on page 63 for an explanation.

Now, populate another instance of the Change Palette with a drag-and-drop operation of the same issue(s) from the source stream to a workspace below the destination stream:



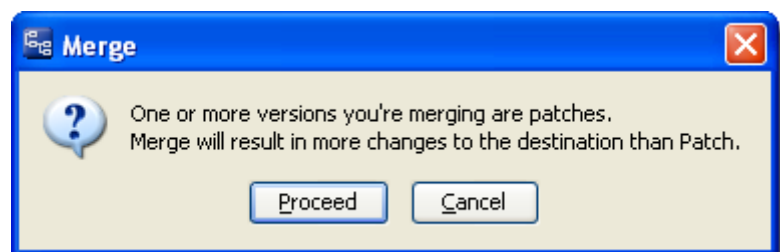


The new Change Palette display is similar to that of the previous instance. Process the elements as indicated below:



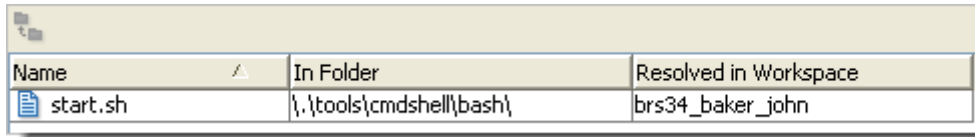
**What about (overlap) status?** If you have made changes to one or more of these elements in this workspace, some of the elements will have **(overlap)** status. For such elements, you can use **Merge** instead of **Send to Workspace**.

If an element has both **(patch)** and **(overlap)** status, use the



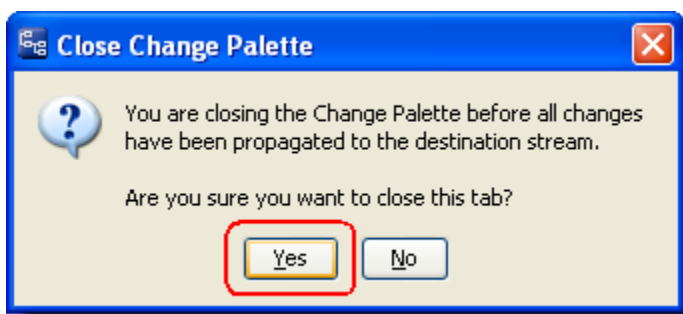
**Patch** command, not the **Merge** command. AccuRev warns you if you attempt to merge. In this case, click **Cancel** and invoke the **Patch** command instead.

After you have processed all the elements in the issue(s), each Patch'ed element is listed at the bottom of the Change Palette:

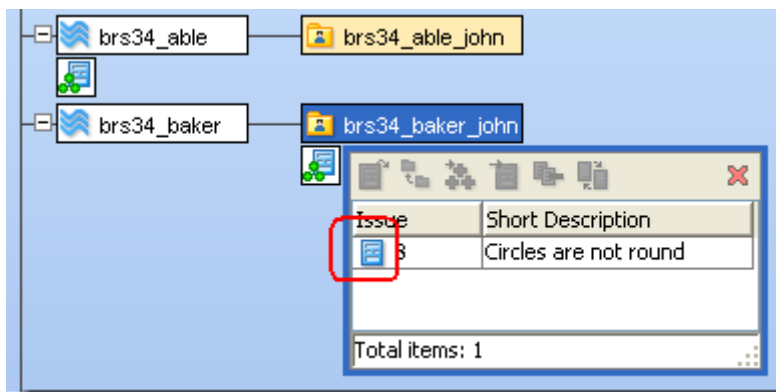


Name	In Folder	Resolved in Workspace
start.sh	\\.\\tools\\cmdshell\\bash\\	brs34_baker_john

Although you might be accustomed to promoting Patch'ed and Merge'd elements from the Change Palette, do not proceed in this manner. Instead, close the Change Palette tab without invoking **Promote**.

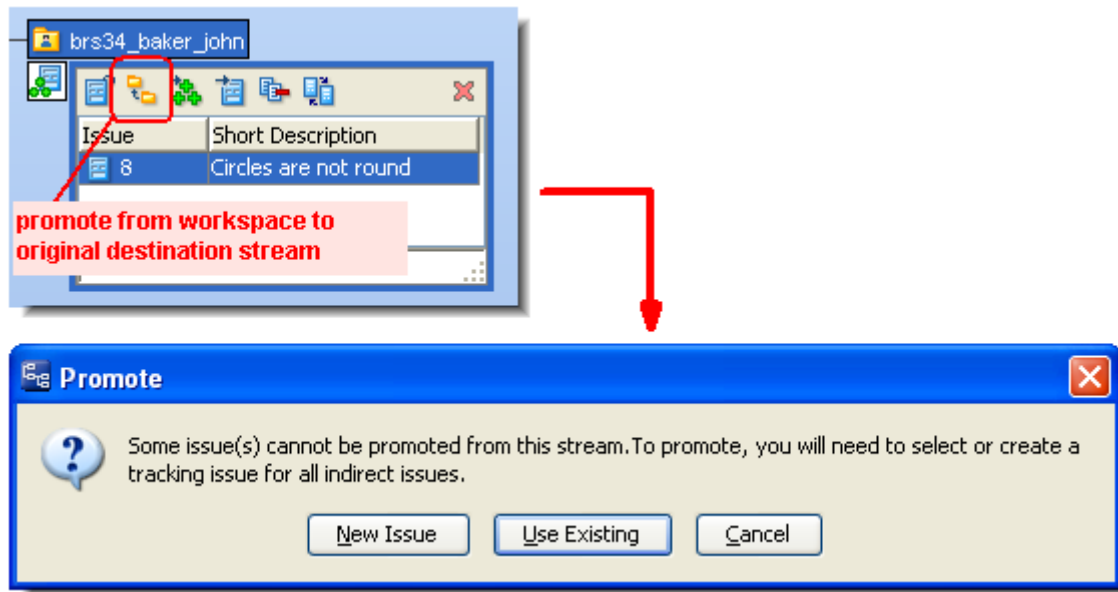


At this point, the selected issue(s) have been propagated to the workspace below the destination stream. Each issue is listed in the workspace's development activity subwindow with a special variant of the issue icon:



## Promotion / Creating a Tracking Issue

When your work on the issue is finished, you must use promote-by-issue to send your changes to the workspace's backing stream (the original destination stream of the cross-promote). Don't use promote-by-element or promote-by-transaction, which would defeat AccuRev's ability to track changes related to the selected issue(s). When you invoke the **Promote** command, AccuRev prompts you to specify an additional issue, called a tracking issue. This is the mechanism that AccuRev uses to record the changes made in this workspace to the original issue(s)' elements.



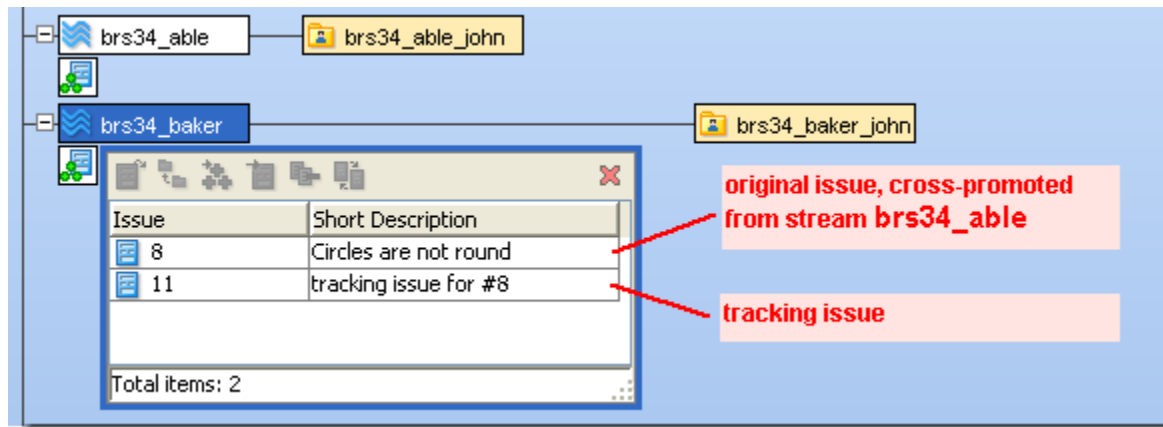
A single tracking issue can keep track of the additional changes for any number of original issues. Thus, it makes sense to select **New Issue** the first time you need to specify a tracking issue, and select **Use Existing** on subsequent uses of promote-by-issue in the same workspace. You might also need to select **Use Existing** if you're not allowed to create new issues — for example, if you've integrated a third-party issue-tracking system with AccuWork.

Either way, an AccuWork edit form appears, in which fill in the fields of the new or existing tracking issue. As usual, click the edit form's **Save** button when you are finished filling in the fields.

AccuRev automatically proceeds with your original command, **Promote**: it prompts you for a comment, then sends the versions from the workspace to the original destination stream. The cross-promotion of the original issue(s) is now complete — after just a bit of a detour!

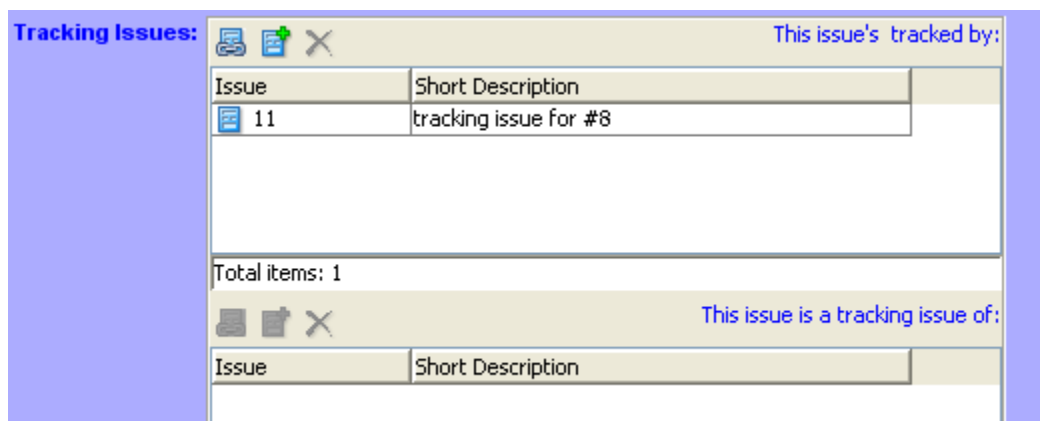
## Working with the Tracking Issue

The Stream Browser shows that the original issue (in our example, #8) has been promoted from the source stream (**brs34\_able**) to the destination stream (**brs34\_baker**). The tracking issue (#11) records the additional changes to issue #8's elements that have been propagated to the destination stream.

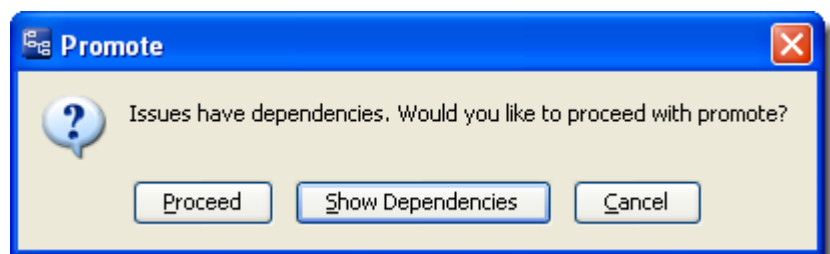


AccuRev Version 4.6 introduced change package dependency tracking, whose goal is to ensure that a **Promote** operation sends a self-contained, consistent set of changes to the destination stream. A tracking issue and the corresponding original issue(s) are connected by a change package dependency: the tracking issue depends on the original issue(s).

The best way to monitor such connections is with a Relationship field whose type is **Track**. For example, when issue #8 is viewed in an edit form, its connection to issue #11 might be displayed like this:



As with all change package dependencies, AccuRev warns you if you attempt to promote a tracking issue without its dependencies, the original issue(s). In this situation, don't click **Proceed**! It's important always to promote both the original issue(s) and the tracking issue at the same time. So the correct procedure is to **Cancel**, select both the original and tracking issues, then invoke **Promote** again.



As you propagate issues up (or across) the stream hierarchy, you must continue to obey this same rule:

Always promote original issues and the corresponding tracking issues at the same time.

If you attempt to **Promote** an original issue alone, without including its tracking issue, AccuRev prompts you to promote the tracking issue as well.

## If You Process Some Elements at the Stream Level, not the Workspace Level

Section *Changing the Way You Use the Change Palette* on page 58 describes a new way to use the Change Palette, but does not describe what occurs if you continue to use the old way. This section provides the details.

Suppose you have populated the Change Palette with a drag-and-drop operation, as illustrated below:

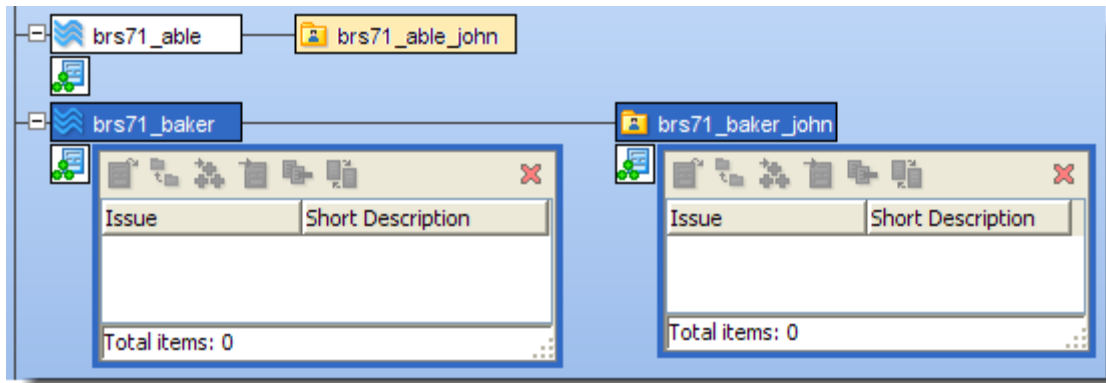
The top screenshot shows a workspace view with a table of issues. A red arrow points from the 'drag-and-drop, to cross-promote issue' text to the 'Issue 6' row. Another red arrow points from the 'in the Change Palette, one or more of the elements is listed with (patch) status' text to the 'Status' column in the Change Palette table.

The bottom screenshot shows the 'Change Palette' dialog box. The 'Destination Stream' is 'brs71\_baker' and the 'Source Stream' is 'brs71\_able'. The table below shows the elements being promoted:

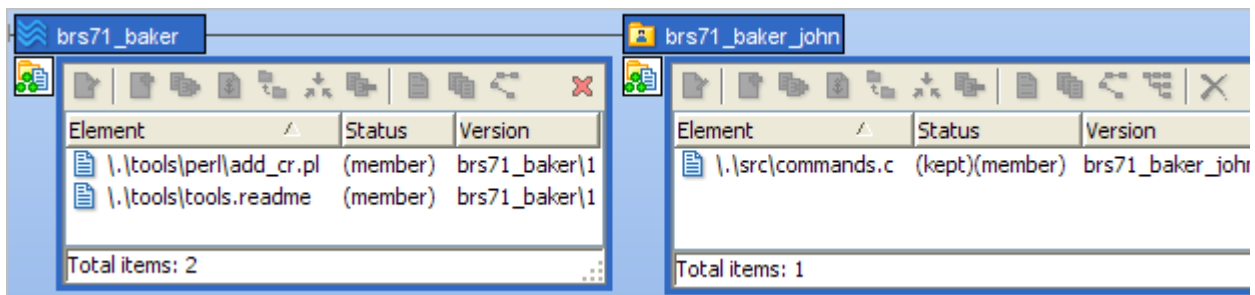
Element	Version	Basis Version	Trans.	Status	Count
\\src\commands.c	6\17	6\14		(patch)	1
\\tools\perl\add_cr.pl	6\8	6\1			1
\\tools\tools.readme	6\5	6\1			1

A red box highlights the 'Status' column, and another red box highlights the 'add\_cr.pl' and 'tools.readme' rows. A red arrow points from the 'elements that should not be Promoted in this way' text to the 'add\_cr.pl' and 'tools.readme' rows.

If you proceed to **Promote** elements **add\_cr.pl** and **tools.readme** to the destination stream and **Patch** element **commands.c** to a workspace below the destination stream, you create the following situation:



(This is a composite picture — the Stream Browser displays the development activity for only one stream at a time.) Issue #6 has been “taken apart” by the **Promote/Patch** sequence in the Change Palette. The issue is not completely “in” stream **brs71\_baker**, nor is it completely “in” workspace **brs71\_baker\_john**. Switching to the Stream Browser’s file mode confirms that issue #6 has been “taken apart”.



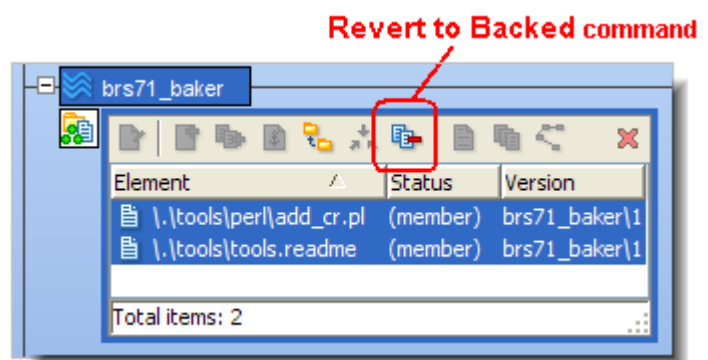
(Again, this is a composite picture.) Two of issue #6’s three elements have been sent to one destination, and the third element has been sent to another destination.

## Fixing Your Mistake

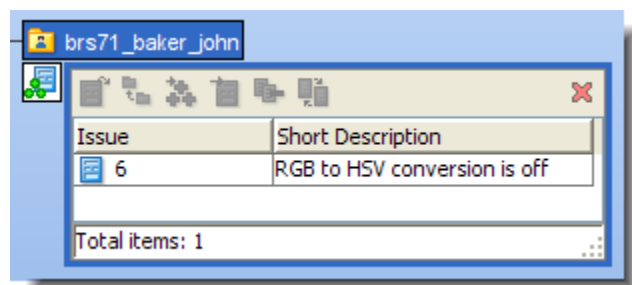
If you get yourself into this situation, what is the remedy? There are two options. **Revert to Backed**, or **Promote** the missing file.

**Revert to Backed:** First, use **Revert to Backed** to “undo” all the **Promote(s)**. Then, switch the Stream Browser back to issue mode, and proceed as described in section *Changing the Way You Use the Change Palette* on page 58. That is, drag-and-drop the original issue(s) to the workspace below the original destination stream. (In this example, drag-and-drop issue #6 to workspace **brs71\_able\_john**.)

**Promote** the missing file: From the workspace, select the file and **Promote** against an issue which will track the changes made for this issue.



You have now “reunited” the changes to all of the issue(s)’ elements at a single destination. The Stream Browser confirms this.



# Incomplete Change Packages

AccuRev change packages provide a powerful tool for dealing with a set of element versions as a group. However, if you do not institute and follow some best practice policies, you can encounter error messages warning of missing versions when you promote by issue. This section describes the three most common causes of these warnings, how to correct them, and how to avoid them in the future.

## Overview

The formal definition of an incomplete change package is "Some of the versions of elements associated with the change package are not part of the current stream."

But what does this really mean? How did this happen? And how do you fix it? Better yet, how do you avoid getting into this situation in the first place?

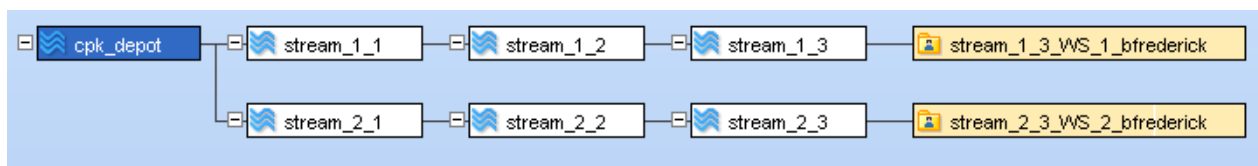
There are three typical scenarios that result in incomplete change packages:

- **Purge** ("**Revert to Backed**" in the GUI) operations on individual files.
- Reuse of issues across multiple streams.
- **Promote by File** (instead of **Promote by Issue**) operations.

All three of these result in the same situation: one or more of the files in your change package will not have the correct version in the current stream.

## An Example Scenario

Here is a very simple stream environment that we will use to illustrate a typical incomplete change package situation:

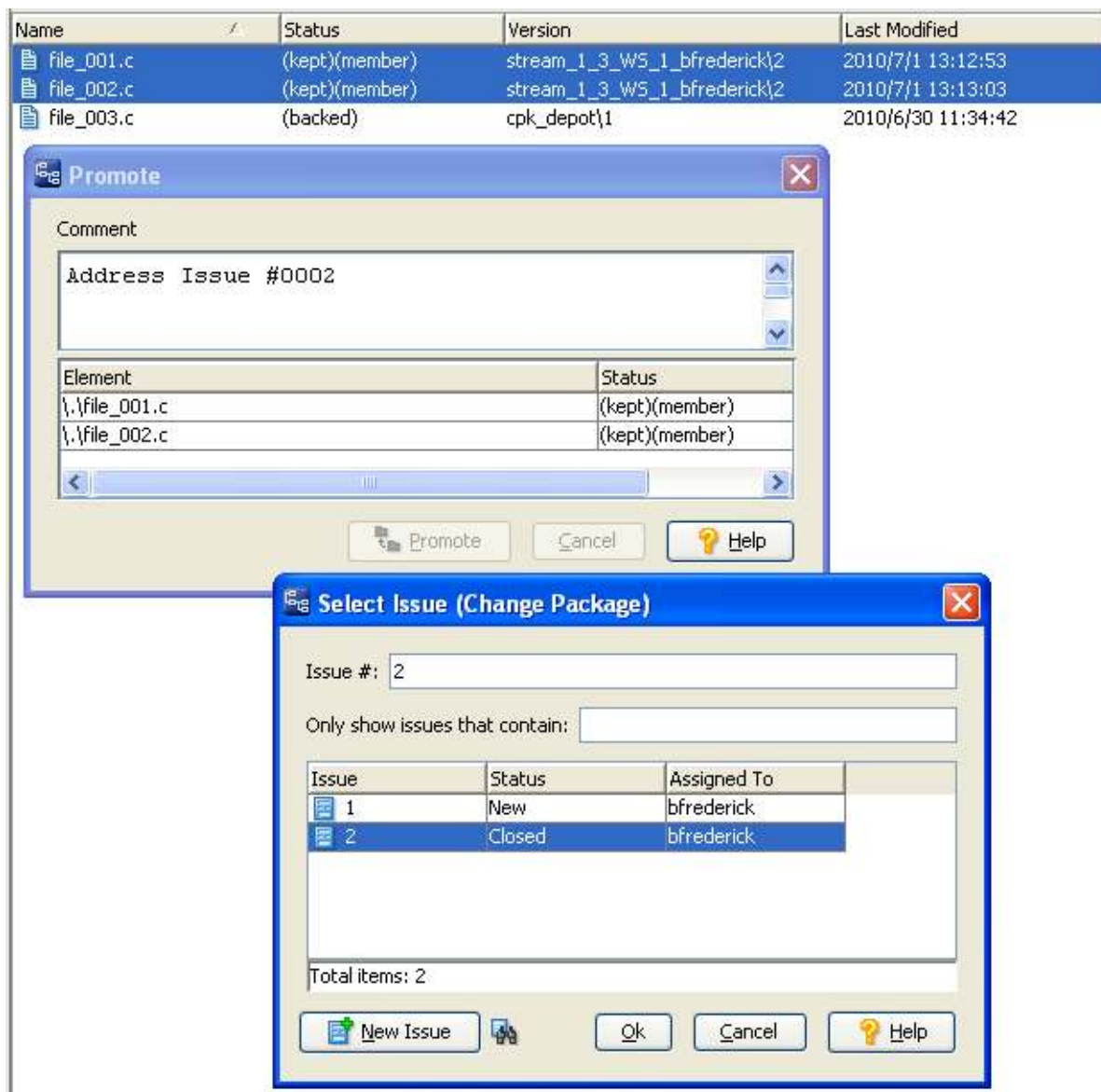


Assume that workspace stream\_1\_3\_WS\_1 contains three source code files:

Name	Status	Version	Last Modified
file_001.c	(backed)	cpk_depot\1	2010/7/1 12:17:10
file_002.c	(backed)	cpk_depot\1	2010/7/1 12:17:10
file_003.c	(backed)	cpk_depot\1	2010/7/1 12:17:10




You modify file\_001.c and file\_002.c to address issue #0002, and promote them from your workspace to the parent stream (stream\_1\_3).

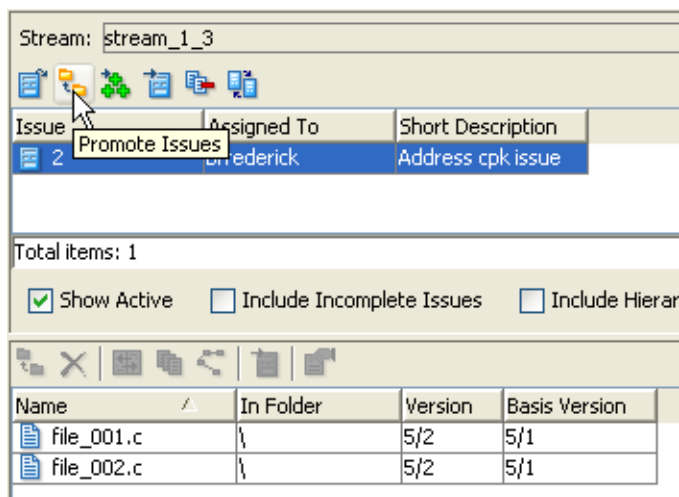


**Note:** You must have the change package integration enabled in the Schema Editor to be prompted for an issue when you promote from a workspace. If you don't see the Select Issue (Change Package) dialog when you promote, you need to enable the integration. See "Change-Package-Level Integration" in the *AccuRev Administrator's Guide* for details.

In addition, if you have implemented support for a third-party issue tracking system (ITS), the Select Issue (Change Package) dialog displays controls that let you choose whether to submit changes against the AccuWork issue number or the third-party ITS key. See *Using Third-Party ITS Keys* on page 79 for more information.

At this point, if you go to the stream browser and select **Show Active Issues** by right-clicking over stream 1\_3, you will see issue #2 and the files that you just promoted. Make sure that **Include**

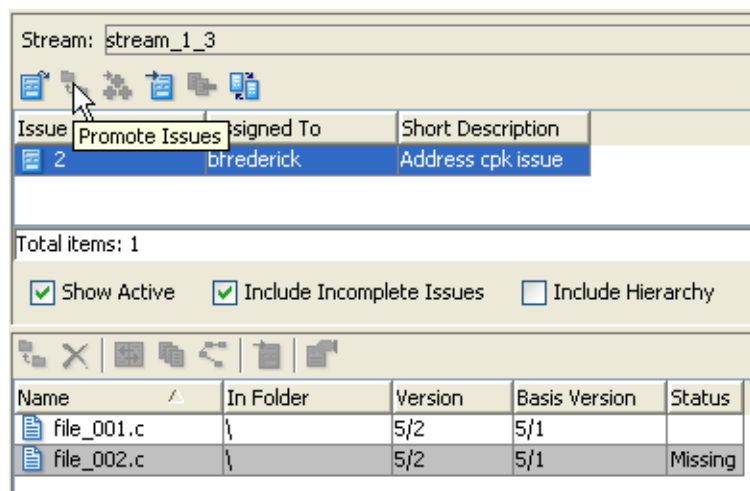
**Incomplete Issues** is not checked to confirm that this is a complete change package. If you were ready to continue promoting this package up the stream hierarchy, you could do so by selecting issue #2 and clicking the **promote** icon (  ).



This is the correct way to promote change package files higher up the stream hierarchy.

But what if somebody had done one of the three operations listed above, which caused, for example, only the backed version of file\_002.c to appear in stream\_1\_3?

This first thing you might notice is that the issue does not appear unless the **Include Incomplete Issues** checkbox is enabled. And when it is, the **Promote** icon is grayed out when you select issue #2, and file\_002.c appears with a gray band.



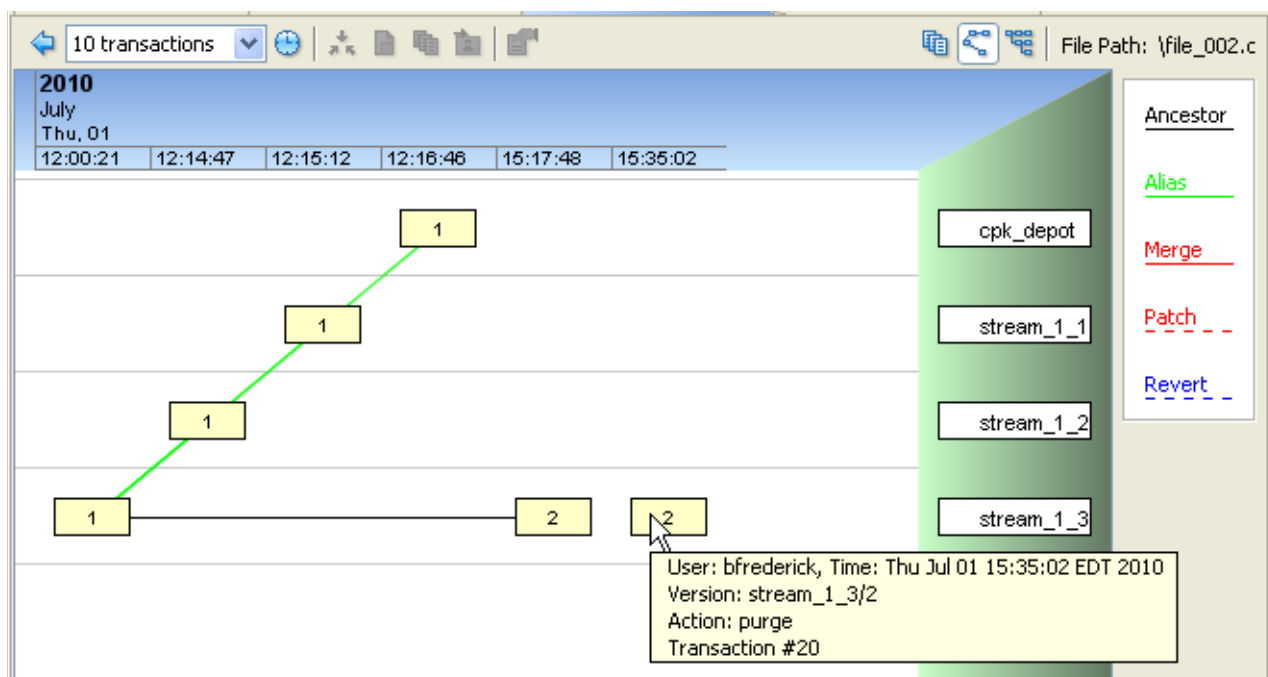
For steps on how to troubleshoot and identify the cause of your incomplete change package, see [How to Troubleshoot Incomplete Change Packages](#) at the end of this section. But first, here are details about each of the three main scenarios.

## File Purge (Revert to Backed)

### Scenario:

You fix an issue and then promote the affected files from your workspace to an integration stream. For some reason, somebody decides to perform a **purge (Revert to Backed)** on one of the files. When it's time to promote the change package from the integration stream to its parent stream, you discover that the change package is incomplete and the **promote** icon is grayed out. This is, in fact, exactly what happened in *An Example Scenario* above.

You can tell that this is the case by right-clicking on the “missing” file (file\_002.c) and selecting **Browse Versions**.



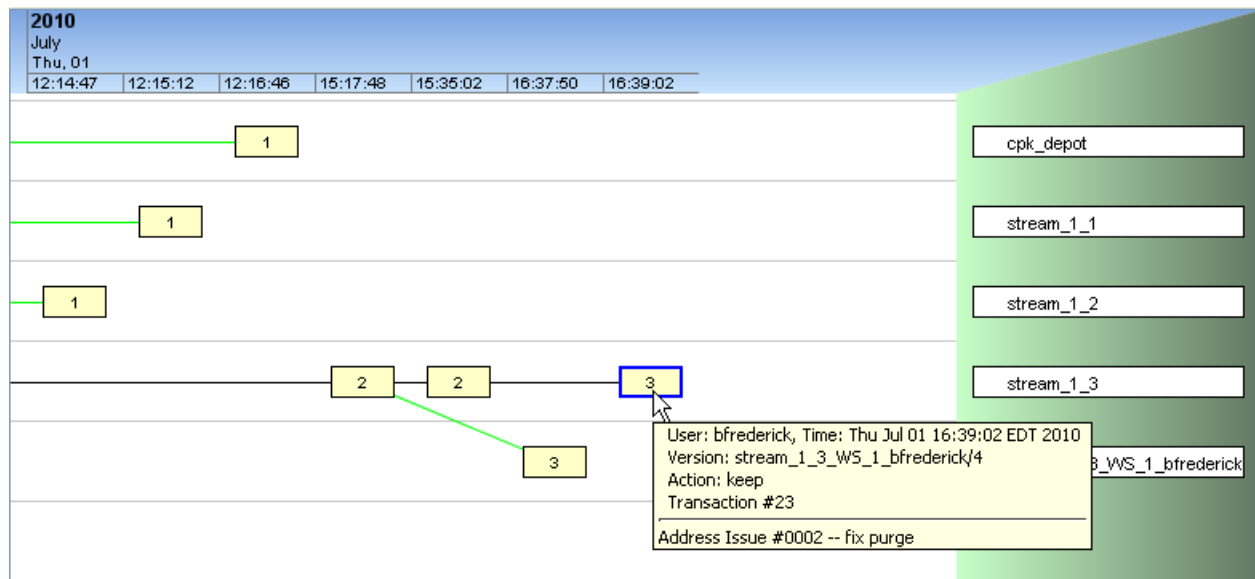
The second, disconnected box for version 2 is the tip-off that the file was purged, and the tooltip message confirms the action.

### How do you fix it?:

If you determine that the purge was a good idea, and that the change package doesn't require the purged file, your simplest fix is to remove it: go to the Changes tab for the issue, select the file and then click remove. Once you have done this, the issue will no longer appear when Include Incomplete Issues is checked, and you will be able to promote the change package further up the stream hierarchy.

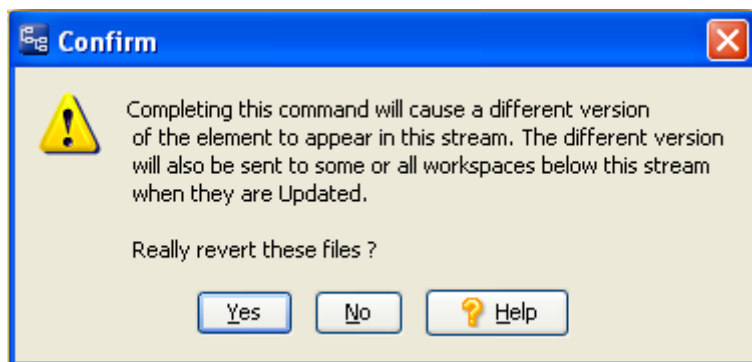
However, if it turns out that the **purge** was a mistake, you'll need to reverse it. Because AccuRev is TimeSafe, you can't just make the purge disappear like it never happened. But what you can do is send that version of the file to your workspace (right-click the version in the Version Browser and select **Send To Workspace** and re-promote it. When the file becomes active in your workspace, it will not have the same version number—in this example it will now be version “3” instead of the original version “2”, and will become version “4” when you **keep** it. After you

promote it, the change package in the parent stream will again be complete, and the version browser will no longer show the purged version as a disconnected box.




### How do you prevent this from happening in the future?:

If you need to purge a file, first make sure that it is not part of a change package. Pay attention to any error message that tries to warn you against making a mistake.



You typically want to click **No** in response to such a warning.

If the file is a part of a change package, and you still want to purge the file, then perform a **Revert by Change Package** instead: in the Stream Browser, right-click the stream from which you wish to revert the change package and click **Show History**. Then select the promotion operation you wish to reverse, and click the revert icon (  ). Make the adjustments you need in your workspace, and then re-promote the modified change package.

**Note:** As of Release 5.2, you now have two ways to **Revert by Change Package**:

- **Revert Change Package Using Workspace**
- **Revert Change Package Directly in Stream**

***Revert Change Package Using Workspace*** gives you the opportunity to test your changes before promoting them into a stream. ***Revert Change Package Directly in Stream*** puts the results of the revert into the stream immediately, so should be used for simpler reverts, or in streams that do not propagate untested changes to other users.

You can also prevent non-administrators from purging files in streams by customizing the `server_preop_trig` trigger. See [Sample server\\_preop\\_trig rules](#) below for more information.

## Reuse of Issues Across Streams

### Scenario:

In the previous sections, you worked on issue #0002 in workspace `stream_1_3_WS_1` and promoted the following files against it:

- `file_001.c`
- `file_002.c`

But what if this issue also applied to a different release being handled in a different stream -- in this case, the hierarchy under `stream_2_1`. In that stream, the fix needs to be made in a different set of files:

- `file_002.c`
- `file_003.c`

You perform the changes to these files in workspace `stream_2_3_WS_2`. Even worse, somebody else continues to work on issue #2 in the original stream and adds a new file named `file_004.c` which doesn't even exist in Stream 2. Both developers promote their changes against the same issue, one from workspace `stream_1_3_WS_1` and the other from `stream_2_3_WS_2`.

Now the change package is incomplete in both streams: the modified versions of `file_002.c` and `file_003.c` are not available under Stream 1, nor does the new `file_004.c` exist under Stream 2. If you do a Show Active Issues in both of the parent streams, you see two different incomplete change packages, each missing different files.

Stream: stream\_1\_3

Issue	Assigned To	Short Description
2	bfrederick	Address cpk issue

Total items: 1

☒ Show Active ☒ Include Incomplete Issues ☐ Include Hierarchy

Name	In Folder	Version	Basis Version	Status
file_001.c	\	5/2	5/1	
file_002.c	\	9/2	5/1	Missing
file_003.c	\	9/1	5/1	Missing
file_004.c	\	5/1	0/0	

Stream: stream\_2\_3

Issue	Assigned To	Short Description
2	bfrederick	Address cpk issue

Total items: 1

☒ Show Active ☒ Include Incomplete Issues ☐ Include Hierarchy

Name	In Folder	Version	Basis Version	Status
file_001.c	\	5/2	5/1	
file_002.c	\	9/2	5/1	
file_003.c	\	9/1	5/1	
file_004.c	\	5/1	0/0	Missing

### How do you fix it?:

Don't try to cross-promote or merge the missing files. Instead:

1. Create a new issue for Stream 2.
2. Open the original issue and select the files which were added when promoted into Stream 2.
3. Right click and select *Send to Issue*.
4. Select the new issue created in **Step 1** to send the selected file(s) to.

You will now see the original issue as complete in Stream 1, and the new issue as complete in Stream 2.

### How do you prevent this from happening in the future?:

Make it a policy to not use the same issue to address problems in multiple streams. If you need to fix the same issue in another stream, create a new issue for that stream and promote files against that issue.

## Promoting by File Instead of by Issue

### Scenario:

Similar to the previous examples, you work on issue #0002 in workspace stream\_1\_3\_WS\_1 and promote the following files against it:

- file\_001.c
- file\_002.c

Later, in the backing stream(stream\_1\_3), somebody promotes file\_002.c (but not file\_001.c) to its parent stream (stream\_1\_2). This leaves the change package for issue 2 complete in stream\_1\_3, but now it appears as an incomplete change package in stream\_1\_2.

### How do you fix it?

This one is easy: simply promote the entire change package from stream\_1\_3 to its parent stream\_1\_2. Since file\_002.c has already been promoted, it is ignored, but file\_001.c gets promoted, making the change package complete.

### How do you prevent this from happening in the future?:

Make it a policy to always promote by issue. Avoid promoting by file. You can enforce this policy by customizing the server\_preop\_trig trigger to disable promotion by file and only allow promotions to occur via issue. See [Sample server\\_preop\\_trig rules](#) below for more information.

## Sample server\_preop\_trig rules

You can use AccuRev triggers to help prevent some of the actions that can lead to incomplete change packages. For general information about these triggers, see the discussion of [server\\_preop\\_trig](#) in the Administrator's Guide, and the sample [server\\_preop\\_trig.pl](#) file in the AccuRev installation [examples](#) folder. The Perl snippets below take the examples provided in

the sample [server\\_preop\\_trig.pl](#) file one step further and show how you can prevent non-administrators from purging files from higher-level streams, and how to enforce promote-by-issue in non-workspace streams.

```
##### CUSTOMIZE ME
#### Add to (or replace) the example code below to
#### implement validation for the PROMOTE command.
#####

# EXAMPLE VALIDATION:
# only a user listed as an administrator can promote versions
# to a stream in the "admin_stream" list

# if ( defined($admin_stream{$stream2}) and `$:AccuRev ismember $principal "$admingrp"` == 0 )
{
# print TIO "Promoting to a stream identified as an 'admin stream' disallowed:\n";
# print TIO "server_preop_trig: You are not in the $admingrp group.\n";
# close TIO;
# exit(1);
#}

# EXAMPLE VALIDATION:
# only a user listed as an administrator can run the Promote
# command without entering a comment
if ( $comment eq "" and `$:AccuRev ismember $principal "$admingrp"` == 0 ) {
    print TIO "Empty comments for 'promote' command disallowed:\n";
    print TIO "server_preop_trig: You are not in the $admingrp group.\n";
    close TIO;
    exit(1);
}
# end of EXAMPLE VALIDATION

# This will prevent users from promoting or cross promoting individual files.
# Only users defined in the $admingrp group will be allowed to promote by file.
# This will prevent issues from becoming incomplete which can cause coalescing problems.

# foreach my $changepackage (keys(%{$$xmlinput{'changePackagePromote'}})){
# my @issues = (@{$$xmlinput{'changePackagePromote'}[0]{'changePackageID'}});
# my @noissue = (@{$$xmlinput{'changePackagePromote'}});
# foreach my $issue (@noissue) {
#     # foreach my $issue (@issues) {
#     # print "Array Issue num = $issue\n";
#     if ($issue == 0 and `$:AccuRev ismember $principal "$admingrp"` == 0 ){
#         print TIO "Promotion by file is disallowed.\n";
#         print TIO "You need to promote by issue, please select the issue which needs promotion and
#             promote\n";
#         print TIO "Only users in the $admingrp group are able to promote by file as this can cause
#             incomplete issues.\n";
#         close TIO;
#         exit(1);
#     }
# }
# }

# end of EXAMPLE VALIDATION

# no problems, allow command to proceed
close TIO;
exit(0);
}

#### end of validation for PROMOTE command
```

```

####
#### Validation for PURGE command
####

if ($command eq "purge") {
# at this point, the following variables will have meaningful values:
#   $hook Trigger name
#   $command AccuRev command being run
#   $principalUsername of person invoking command
#   $ip IP address of AccuRev client machine
#   $streaml Stream from which versions are being purged
#   $depotDepot name
#   $fromClientPromoteData passed from pre-promote-trig script
#   @elems Element list

##### CUSTOMIZE ME
#### Add to (or replace) the example code below to
#### implement validation for the PURGE command.
#####

# EXAMPLE VALIDATION:
# only a user listed as an administrator can promote versions
# to a stream in the "admin_stream" list

# if ( defined($admin_stream{$streaml}) and `$:AccuRev ismember $principal "$admingrp"` == 0 )
{
# print TIO "Purging from a stream identified as an 'admin stream' disallowed:\n";
# print TIO "server_preop_trig: You are not in the $admingrp group.\n";
# close TIO;
# exit(1);
#}
# end of EXAMPLE VALIDATION

# Prevent users from purging elements so AccuWork issue will not become incomplete in streams.
unless ( $streaml =~ /_/$principal/ or `$:AccuRev ismember $principal "$admingrp"` == 1 ) {
    print TIO "You can not perform \"Revert to Backed\" or purge operations in streams.\n";
    print TIO "This will prevent CR's from disappearing in streams due to them becoming
        incomplete issues.\n";
    print TIO "Only users in the $admingrp group will be authorized to preform this
        operation.\n";
    close TIO;
    exit(1);
}

# no problems, allow command to proceed
close TIO;
exit(0);
}

#### end of validation for PURGE command

```

## How to Troubleshoot Incomplete Change Packages

Use the following steps to identify incomplete change packages and their causes. Then you can use the information above to fix the situation and avoid it in the future.

**Note:** There is an excellent training video on the AccuRev web site covering these troubleshooting steps. Web addresses change over time, so go to [www.accurev.com](http://www.accurev.com) and search for “training videos”. This one is titled “Incomplete Change Packages”.



1. To see if an incomplete change package exists in a stream, go to the Stream Browser, right-click over the stream, and select Show Active Issues.
2. Make sure that ***Include Incomplete Issues*** is checked, and look for any grayed out issues.
3. If you see a grayed out issue, select it. This displays the change package files in the lower pane. At least one of these will be grayed out and have a status of “missing”.
4. Make note of the values in the “Version” and “Basis Version” columns. The first number (before the slash) identifies the stream and the second number (after the slash) identifies the file version that should exist in that stream.
5. To identify the stream names for these numbers, go to the Stream Browser and display the tabular mode. Look for the number under the column labeled “#”, and identify its name under the “Name” column.
6. To figure out why the file is missing, use the Version Browser:
  - If the version is simply missing from this stream, but exists in the child stream, then it is likely that one or more of the files from the change package were promoted individually (rather than by issue) and this file was not. (See *Promoting by File Instead of by Issue* above.)
  - If the version shows a disconnected box in the Version Browser, the problem is due to a purged file. (See *File Purge (Revert to Backed)* above.)
  - If the version is missing from this stream, and doesn’t exist in the child stream but appears to exist in a different stream hierarchy, it is likely that the same issue was used to fix a problem in two different streams. (See *Reuse of Issues Across Streams* above.)

**Note:** There is one additional incomplete change package scenario that you may encounter: if the issue contains only one file, the issue may not appear as incomplete when it should. You need to go to the ***Show Active Issues*** display and ensure that the ***Include Hierarchy*** checkbox is enabled for this situation to be displayed.

# Notes on Revert to ... and Diff Against...

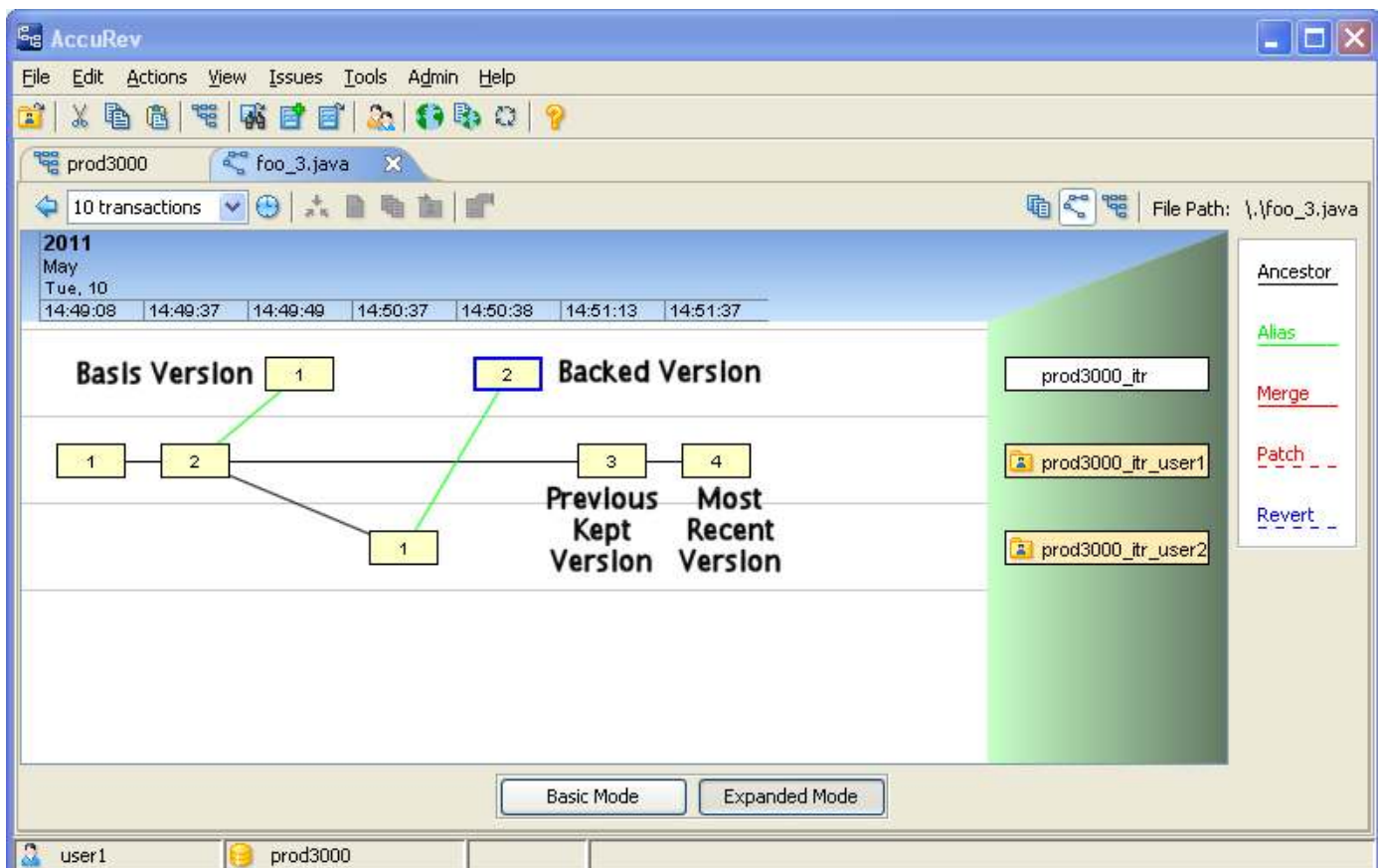
## GUI Commands

**Revert to Backed** is the GUI version of the CLI *purge* command. The name “Revert to Backed” is something of a misnomer, as the operation would more accurately be named “Revert to Last Update Version” (that is, replace the current workspace version with the version from the backing stream based on the last time the workspace was updated).

Since the **Diff Against...** GUI command offers both “Backed” and “Basis” options, and **Revert to...** also offers a “Most Recent Version” option, it is easy for new users to be confused by these different features. This section illustrates the differences between Backed and Basis versions of an element, and what a user can expect from various AccuRev GUI commands.

### Overview

The following screenshot shows the progression of file foo\_3.java. The file is initially created, edited, and promoted to stream prod3000\_itr by user1 in workspace prod3000\_itr\_user1. It is then edited and promoted to stream prod3000\_itr by user2 in workspace prod3000\_itr\_user2. Finally, user1 makes two more sets of changes in workspace prod3000\_itr\_user1, which she keeps, but does not promote. The screenshot is taken from an “Expanded Mode” Version Browser display (**History -> Browse Versions**) of foo\_3.java from the prod3000\_itr stream.



**Note:** For the discussions below, it is critical to have a firm understanding of the *AccuRev Glossary* terms *workspace stream* and *workspace tree*:

### workspace stream

The private stream that is built into a workspace. All new versions of elements are originally created in workspaces; AccuRev records these versions in workspace streams.

### workspace tree

The ordinary directory tree, located in the user's disk storage, in which the user performs development tasks and executes AccuRev commands.

## Diff Against...

In relation to the “Most Recent Version” of foo\_3.java in the prod3000\_itr\_user1 *workspace stream* (prod3000\_itr\_user1/4), **Diff Against...** can be used with:

- **Backed Version**—prod3000\_itr/2. For a **Diff** operation, this is always the current version in the backing stream.
- **Basis Version**—prod3000\_itr/1. In this example, this is the previous promoted version, and is the base version originally used to create Version 4. The basis version is the version you began your work with—usually the version from the backing stream downloaded from the server by your last update operation, or the last time you promoted the file, whichever came last.
- **Previous Version** —prod3000\_itr\_user1/3. The previous kept version in the *workspace stream*. (Note: **Diff Against...** the previous version is available from the **Other Version** option in the Version Browser.)

Similarly, foo\_3.java in the prod3000\_itr\_user1 *workspace tree* can be compared with the backed and basis versions from the prod3000\_itr backing stream using the rules above, and its most recent version, prod3000\_itr\_user1/4 in the *workspace stream*. It can also be compared to the previous kept version and any prior version using **Diff Against.. -> Other Version** from the Version Browser.

## Revert to...

In the context of a **Revert to...** operation on foo\_3.java in the prod3000\_itr\_user1 *workspace tree*:

- **Revert to Backed** would replace the version of foo\_3.java in the *workspace tree* (the current workspace version, which may or may not be the Most Recent Version) with the version from the backing stream based on the last time workspace was updated. In this example, this is its basis version, prod3000\_itr/1, *not the current version in the backing stream* (prod3000\_itr/2), *which could potentially cause your local build to fail*. Think of this command as “**Revert to Last Update Version**”.
- **Revert to Most Recent Version** would replace the modified version in your workspace tree with the last kept version (i.e., it would replace the version of foo\_3.java in the *workspace tree* with the latest version from the workspace stream, prod3000\_itr\_user1/4.)

A **Revert to Backed** operation on the current version in the prod3000\_itr dynamic stream (prod3000\_itr/2) would cause foo\_3.java to become inactive in the stream. This would result in the prod3000\_itr stream inheriting the version from its parent stream. (It is not replaced with the previous version in the stream, prod3000\_itr/1.) To "undo" a **promote** or **purge** operation, a **Revert** from the History Browser (or a CLI **revert** command) is issued specifying the transaction number. This creates new kept versions that are minus the set of changes promoted in the specified transaction. These kept versions must then be promoted to complete the undo (reverse merge) operation.

## Using Third-Party ITS Keys

If you use AccuRev with a third-party issue tracking system (whether through an integration, or just in parallel), you might find it convenient to use the third-party issue numbers (“keys”) along with or instead of AccuWork issue numbers.

To do this, you need to:

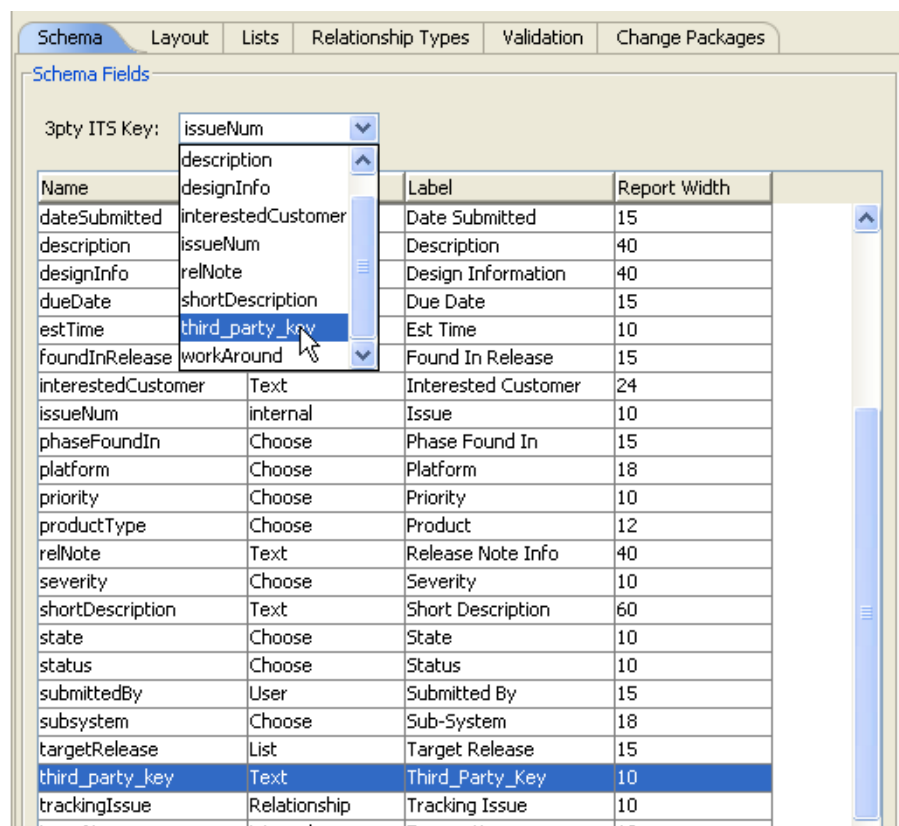
- Customize your AccuRev schema to define a new third-party issue field (or make use of an existing one).
- Specify a new `-3` switch with CLI commands that make use of third-party tracking numbers (for example, `promote -3 -I <3rd_party_ITS_key> <element_name>`).

## Modifying the schema

To make use of third party ITS keys, you must first modify the AccuRev schema. You can do this by either manually editing the `schema.xml` file, or by using the Schema Editor in the Java GUI (see [Schema Editor](#) on page 301 of the AccuRev [On-Line Help Guide](#)).

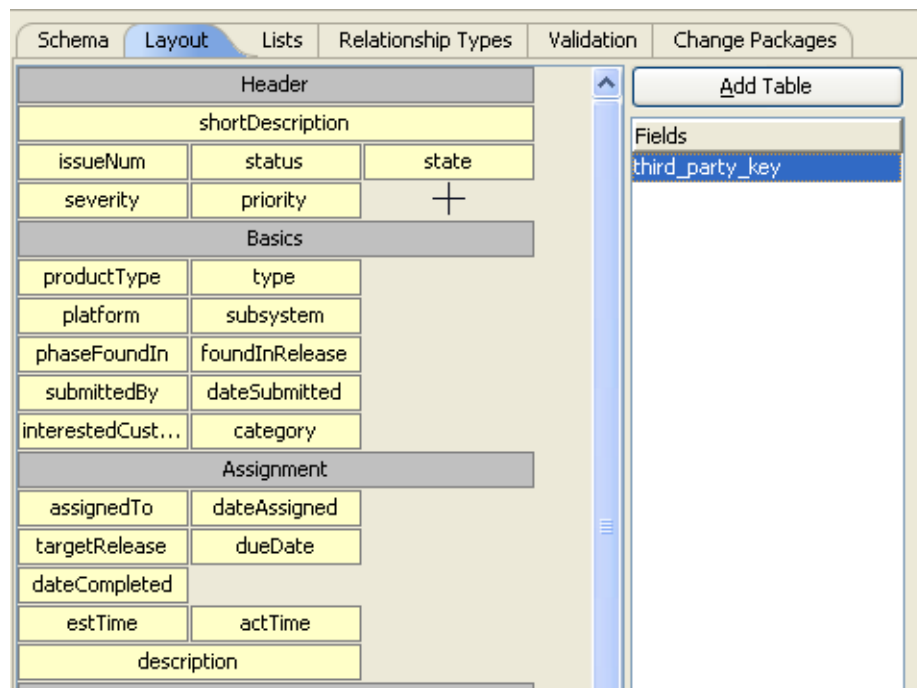
## Using the Schema Editor

If you do not wish to edit `schema.xml` directly, you can use the Schema Editor in the AccuRev Java GUI. Identify an existing key field that can be used, or define a new one using the **Add** button. Then assign this key field using the **3pty ITS Key** dropdown menu.



Note that some AccuRev schemas (like the schemas delivered with AccuSync) already have a third-party key defined, such as “jirakey”. If this is the third-party ITS key that you need to use, you just need to ensure that it is assigned to the **3pty ITS Key** dropdown menu as described above, and that it is positioned on the Issue form as described below.

Next go to the *Layout* tab and position this field in the Issue form (or confirm that it is already there). To position it, select the entry in the Fields column, and then click where you want it to appear in the left side of the tab (for example, in the Header section).



Save your changes and exit the Schema Editor. Now when you display the AccuWork Issue form, the third-party ITS key field will be displayed.

The screenshot shows the 'AccuWork Issue' form. The 'Short Description' field is at the top. Below it are several input fields: 'Issue:', 'Status:' (set to 'New'), 'State:' (set to '<none selected>'), 'Severity:' (set to '<none selected>'), 'Priority:' (set to '<none selected>'), and 'Third\_Party\_Key:'. At the bottom, there are tabs for 'Basics', 'Assignment', 'Misc', 'Attachments', 'Resolution', 'Relationship', 'Changes', and 'Issue History'. The 'Basics' tab is active, showing 'Product:' (set to 'Product1') and 'Type:' (set to 'defect').

## Editing schema.xml

You can also directly edit the schema XML files (schema.xml and layout.xml). In a text editor, open the schema.xml file for your depot:

```
<ac_install>/storage/depots/<depot_name>/dispatch/config/schema.xml
```

First, determine if you need to define a new field to handle the third-party ITS key. Note that all AccuRev schemas have an entry for AccuWork issue numbers. For example:

```
<field
name="issueNum"
type="internal"
label="Issue"
reportwidth="10"
fid="1">
```

Do not use this for third-party ITS keys. Define a new one (or determine if you already have one available in your schema) and make sure that it is specified in the `lookupField` as described below.

Some AccuRev schemas (such as the schemas delivered with AccuSync) already have a third-party key defined, such as:

```
<field
name="jiraKey"
type="Text"
label="JIRA Key"
reportwidth="10"
width="10"
fid="3">
</field>
```

If you need to define a new key, do so using the above key as an example, making sure to assign it an unused `fid` (field ID) number.

After identifying a key field, or defining a new one, modify the `fid` value of the `lookupField` entry to reflect the new key field (the `lookupField` entry is typically found near the top of `schema.xml`):

```
<lookupField fid="1"/>
```

For example, if you were modifying this entry to use the `jiraKey` field shown above, you would change the `fid` value to “3”.

If you will be using this convention in new depots that you create in the future, you should also make this change to the master copy of `schema.xml` in:

```
<ac_install>/storage/site_slice/dispatch/config
```

To position this field on the Issue form (or to confirm that its position is already defined), open `layout.xml` in a text editor. Use the existing entries in the file as a model for defining the third-party ITS key field.

## Using Third-Party ITS Keys in the CLI

Once the schema has been configured as described in the previous section, you use third-party ITS keys with a variety of AccuRev CLI input commands by specifying the new “-3” switch in addition to the traditional “-I” switch (for example “-3 -I US23407”). Note that you can also still use AccuWork issue numbers by using the traditional “-I” switch by itself, but you cannot specify both on the same command line.

Here are the AccuRev commands that accept the -3 switch:

- **cpkadd**
- **cpkdepend**
- **cpkdescribe**
- **cpkremove**
- **issuelist**
- **merge**
- **patch**
- **promote**
- **purge**
- **revert**

Here are some examples of using the **promote** command with the **-3** switch:

Promote a file and associate it with an issue using the third-party issue key:

```
accurev promote -I <3rd_pty_key> -3 <element_name>
```

Promote a file and associate it with several issues using third-party issue keys:

```
accurev promote -I "<3rd_pty_key_1> <3rd_pty_key_2> <3rd_pty_key_3>"
-3 <element_name>
```

You can also use the third-party keys with the **-Fx** switch for XML. For example, to promote a file using the **-Fx** switch and associate it with several issues using third-party issue keys:

```
accurev promote -Fx -l <xml_filename>
```

where the XML file has the following format:

```
<issues third_party="true">
<id>[3rd_pty_key_1]</id>
<id>[3rd_pty_key_2]</id>
<id>[3rd_pty_key_3]</id>
</issues>
```

## Commands That Return Third Party Issue Numbers

These commands return third-party key information in XML output (specified with the **-fx** switch):

- **hist**
- **cpkdepend**
- **xml -l**

**cpkdepend** also returns third-party key information in non-XML format. If you specify **-3** to query the dependency, then the values in non-fx will be displayed using third-party values. If specify standard issue numbers, they will be displayed in standard format. If you specify **-3**



without **-fx** and an issue does not have a third-party value, an asterick ('\*') appears before the issue number.

In the examples below, the third-party values are displayed in **bold**. "**thirdPartyName**" is the field name (not the label name) defined in the Schema Editor.

### hist example output

```
<version
  path="\.\file.cpp"
  eid="3"
  virtual="4/2"
  real="5/2"
  virtualNamedVersion="s25263_proj1_dev/2"
  realNamedVersion="ws25263_testuser1/2"
  elem_type="text"
  dir="no">
  <issueNum
    thirdPartyName="jiraKey"
    thirdPartyValue="J1002">2</issueNum>
</version>
```

### cpkdepend example output

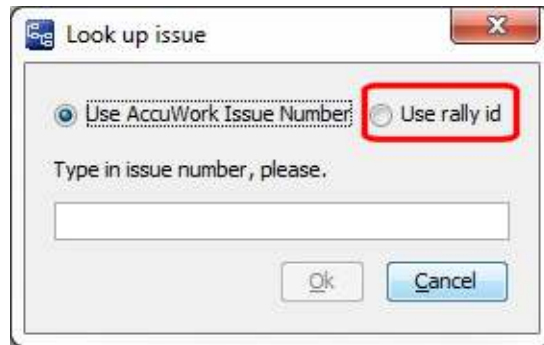
```
<acResponse>
  <issueDependencies>
    <issueDependency>
      <issues>
        <issue
          number="2"
          thirdPartyName="jiraKey"
          thirdPartyValue="J1002"
          incomplete="false"/>
      </issues>
    <dependencies>
      <issue
        number="1"
        thirdPartyName="jiraKey"
        thirdPartyValue="J1001"/>
      </dependencies>
```

```
</issueDependency>
</issueDependencies>
</acResponse>
```

## Using Third-Party Keys in the Java GUI

As of AccuRev 5.4.1, if you have enabled the use of third-party issue tracking system (ITS) keys in the AccuWork schema, the *Select Issue (Change Package)* dialog and the *Look Up Issue* dialog display additional controls that let you indicate whether you want to use AccuWork issue numbers or third-party ITS keys when specifying issues.

For example, when you bring up the *Look Up Issue* dialog, if third-party ITS keys are enabled AccuRev displays radio buttons that let you indicate that you want AccuRev to use the third-party key rather than the AccuWork issue number when looking up the issue.



The radio button for the ITS is labeled with whatever text is specified in the Schema Editor for the "Label" field (see [Using the Schema Editor](#) on page 79).