



AccuRev®

Technical Notes

Version 7.1

Revised 15-September-2017

Copyright and Trademarks

Copyright © Micro Focus 2016. 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; 8,341,590; 8,473,893; 8,548,967.

AccuRev, **AgileCycle**, and **TimeSafe** are registered trademarks of Micro Focus.

AccuBridge, **AccuReplica**, **AccuSync**, **AccuWork**, **Kando**, and **StreamBrowser** are trademarks of Micro Focus.

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

Table of Contents

1. Converting Baselevel Directories to AccuRev1

Creating a Depot	1
Recording the Baselevel with a Snapshot.....	2
Handling Additional Baselevel-to-Baselevel Differences	3

2. Creating and Using a Maintenance Stream5

Creating a Snapshot	5
Creating a Stream Based on the Snapshot	5
Performing Maintenance Work.....	5

3. Specifying Ignore Patterns for External Objects 7

What Is an Ignore Pattern?.....	7
Example	7
Global versus per-Directory Use of Ignore Patterns.....	8
Ignore Pattern Precedence	9
Specifying an Exception to an Ignore Pattern	9
The (ignored) Status	9
Where You Can Use .acignore	10
Specifying Ignore Patterns in .acignore.....	11
One Pattern per Each --ignore	11

4. Techniques for Sharing Workspaces 15

Accessing a Windows Workspace From Multiple Windows Clients.....	15
Universal Access to a Workspace Located on a Share.....	16
The 'share_map.txt' File.....	16
Workspace Location Entries	17
Example: Samba Share	18

5. What's the Difference Between Populate and Update?..... 19

In a Nutshell	19
Example 1: Standard Update Scenario	20
Example 2: Restoring a Deleted File ("missing" by accident).....	20
Example 3: Handling Active Elements.....	20
Example 4: A Tale of Two Files.....	20
Backing Stream.....	22
Workspace Stream	23

Workspace Tree	25
Incomplete Updates	27
Incomplete Update: Command Interrupted	28
Incomplete Update: Checksum Failure	28
Performing the “Fixup” Update.....	28
6. Using a Trigger to Maintain a Reference Tree.....	29
7. Notes for CVS Users	31
AccuRev Workspaces vs. CVS Sandboxes.....	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.....	33
Checking out files to edit.....	34
Comparing versions of files.....	34
8. Version Control of Namespace-Related Changes	35
Defunct element obscured by element with same name.....	40
Elements under a defunct parent.....	40
Elements under an excluded parent	41
Dangling directory elements.....	41
Elements under a non-existent (purged) parent directory.....	42
Elements under a stranded parent directory.....	42
Active element refers to a purged version	42
9. Notes on Cross-Links	43
Cross-Link Direction and Terminology.....	43
Cross-Links and Stream Namespaces.....	43
Double Vision: Seeing an Element Multiple Times in a Workspace	46
Cross-Link Overlaps.....	48
10. Notes on Revert to ... and Diff Against... GUI Commands	49
Overview.....	49

1. Converting Baselevel Directories to AccuRev

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 in the `.acignore` file. For example:

```
*.exe
*.obj
*.lnk
```

`*.err`

`*.map` (and so on)

(See [Specifying Ignore Patterns for External Objects](#) 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 ignore patterns 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
```


2. 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**).

3. Specifying Ignore Patterns for External Objects

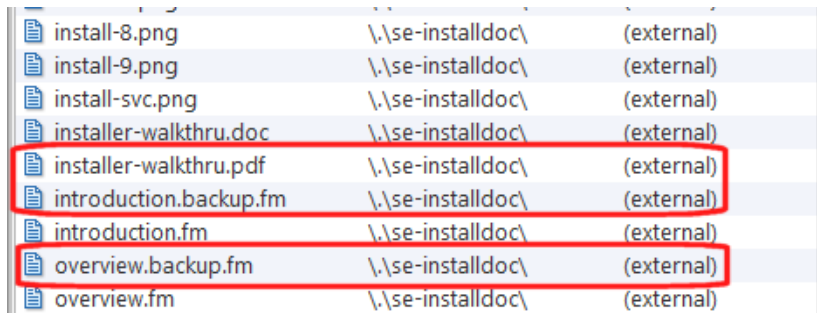
Software development projects can result in the addition of hundreds of new files to a workspace. This chapter describes how to create and use *ignore patterns* that help AccuRev distinguish files you care about (source files with suffixes like `.c` or `.cc` or `.java`, for example) from those you do not (program-generated executables (`.exe`) and backup files (`.bak`), for example).

What Is an Ignore Pattern?

An *ignore pattern* is a value AccuRev uses to filter external objects (and only external objects) from various AccuRev operations and displays. For example, if your workspace is routinely cluttered with numerous backup files, you can create an ignore pattern to filter them out so they do not appear in the File Browser's Detail pane or are not listed when you run the CLI `stat -x` command.

Example

Imagine identifying the external objects in a workspace. Most are source files that you want to get under AccuRev control, but there are several files (files with the `.backup.fm` and `.pdf` extensions in this example) that you do not.



install-8.png	\\se-installdoc\	(external)
install-9.png	\\se-installdoc\	(external)
install-svc.png	\\se-installdoc\	(external)
installer-walkthru.doc	\\se-installdoc\	(external)
installer-walkthru.pdf	\\se-installdoc\	(external)
introduction.backup.fm	\\se-installdoc\	(external)
introduction.fm	\\se-installdoc\	(external)
overview.backup.fm	\\se-installdoc\	(external)
overview.fm	\\se-installdoc\	(external)

Since you know you never want to add files with a `.backup.fm` or `.pdf` extension to AccuRev version control, you specify an ignore pattern that includes these extensions. For example:

```
*.backup.fm  
*.pdf
```

In this example, we specify the ignore pattern in an AccuRev file called `.acignore`. Now, if you add this file to the `\\se-installdoc` directory and perform the External search again, AccuRev filters out the files in that directory with the extensions you specified in the `.acignore` file:

install-7.png	\\se-installdoc\	(external)
install-8.png	\\se-installdoc\	(external)
install-9.png	\\se-installdoc\	(external)
install-svc.png	\\se-installdoc\	(external)
installer-walkthru.doc	\\se-installdoc\	(external)
introduction.fm	\\se-installdoc\	(external)
overview.fm	\\se-installdoc\	(external)

In addition to using a **.acignore** file, you can also specify ignore patterns using CLI command options. See [Where You Can Specify Ignore Patterns](#) on page 8 for more information.

Where You Can Specify Ignore Patterns

You can specify ignore patterns in AccuRev using:

- One or more **.acignore** files
- The **--ignore** option for the **add**, **files**, and **stat** CLI commands

All of these methods accept the ignore patterns described in [Wildcards in Ignore Patterns](#) on page 12. This section describes differences in their use, and the precedence when ignore patterns have been specified in multiple places.

Global versus per-Directory Use of Ignore Patterns

A *globally* specified ignore pattern is one that affects more than a single directory -- all of a user's workspaces, or all of the directories within a workspace, for example. An ignore pattern specified in a directory affects only that directory. The following table summarizes the effect of the ignore patterns you can specify in AccuRev.

Ignore Pattern Method	Effect	For More Information
.acignore	Per-directory or global, depending on where it is specified.	See Using .acignore on page 10.
--ignore	Global	See Using the --ignore Option for CLI Commands on page 11.

Examples

A simple wild card pattern such as **"*.doc"** that is specified globally matches any of these names:

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

The pattern **manuals/*.doc** specified in **docs.acignore** matches any of these names:

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

... but not these names:

```
docs/manuals/usergd/src/chap01.doc
docs/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 the **docs/manuals** directory. See [Wildcards in Ignore Patterns](#) on page 12 for more information on using ******.

Ignore Pattern Precedence

As mentioned previously, you can specify ignore patterns using one or more **.acignore** files and the **--ignore** option for the **add**, **files**, and **stat** CLI commands. If more than one ignore pattern matches a given file or directory, AccuRev enforces those patterns in the following order:

1. **--ignore** command line option takes precedence over any other ignore patterns
2. **.acignore** in any workspace directory
3. **.acignore** in **%WS_ROOT%/accurev**
4. **.acignore** in **%USERPROFILE%/accurev**

In addition, note that:

- A file or directory is automatically considered to be ignored if one of its parent directories is already ignored. In this case, AccuRev does not perform matching.
- If multiple pattern matches are found in the same **.acignore** file or set of **--ignore** options, the *last match* takes precedence over other matches in the same file or command line option.

Specifying an Exception to an Ignore Pattern

You can use the exclamation mark (**!**) to specify an exception to an ignore pattern. For example, imagine you have specified a global ignore pattern, ***.exe**, so that AccuRev ignores all external objects in your workspace with a **.exe** extension. If you wanted, you could specify **!*.exe** locally in an individual workspace directory, in which case AccuRev would ignore all external objects ending with **.exe** *except* those in the directory you specified. Similarly, specifying **!generate.exe** (**stat -x --ignore=!generate.exe**, for example) would include the **generate.exe** file in the list returned by the **stat** command.

The (ignored) Status

When you specify an ignore pattern, any external objects matching that pattern are given an (ignored) status (in addition to the (external) status they have because they are external objects). Ignored objects are not displayed when navigating using the Workspace Explorer -- because you have explicitly told AccuRev to ignore them.

You can display ignored objects in the File Browser (as shown in the following illustration) by choosing the **Include Ignored Objects** preference on the **General** tab of the AccuRev Preferences dialog box.

Name	Status
overview.fm	(external)
overview.backup.fm	(external)(ignored)
introduction.fm	(external)
introduction.backup.fm	(external)(ignored)
installer-walkthru.pdf	(external)(ignored)
installer-walkthru.doc	(external)

(Ignored objects are never displayed in the File Browser when using the External search filter, regardless of how the **Include Ignored Objects** preference is set.)

See [Controlling the Display of External Objects](#) on page 60 of the **AccuRev On-Line Help Guide** for more information.

Using .acignore

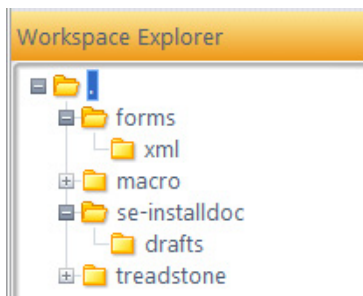
This section describes considerations for using the **.acignore** file to specify ignore patterns.

Where You Can Use .acignore

The following table summarizes where you can use **.acignore** files and what effect that has on the external objects AccuRev ignores.

Location	Effect	Description
Workspace directory	Per-directory	Sets the ignore pattern for that directory only.
%WS_ROOT%/.\accurev	Global	Sets the ignore pattern for the specified workspace. For example: c:\accurev\workspaces\ws_beta\accurev\acignore
%USERPROFILE%/.\accurev	Global	Sets the ignore pattern for all workspaces. For example: c:\users\dfoster\accurev\acignore

Note that patterns specified in per-directory **.acignore** files are not inherited unless they are specified using recursion syntax (**). For example, if you add a **.acignore** file to the **\forms** directory with the pattern ***.bak**, external objects with a **.bak** extension in the **\forms\xml** directory are not filtered.



Similarly, you could specify different patterns using separate **.acignore** files in the **\se-install\drafts** and **\forms\xml** directories. See [Wildcards in Ignore Patterns](#) on page 12 to learn about recursion syntax.

Specifying Ignore Patterns in .acignore

Each ignore pattern in an **.acignore** file must be on its own line. For example, you can specify this:

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

but not this:

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

Using the --ignore Option for CLI Commands

You can specify ignore patterns using one or more **--ignore** options for the **add**, **files**, and **stat** CLI commands. Ignore patterns specified using **--ignore** options take precedence over patterns specified using either the **.acignore** file.

Command	Description
stat -x	Lists external objects. (In the AccuRev GUI, performing an External search in the File Browser Outgoing Changes mode executes a stat-x command.)
add -x	Adds external objects to the AccuRev depot, placing them under AccuRev control.
files	Lists all elements and external objects.

(Note that the **stat -x** command is also used by the External search in the AccuRev GUI.)

When applying **stat -x** and **add -x** commands to an entire workspace, AccuRev uses any applicable ignore patterns to filter the list of objects *before* sending the list to the AccuRev Server process. You can, however, instruct AccuRev to include objects that would otherwise be ignored because they match an applicable ignore pattern, as described in the following section.

One Pattern per Each --ignore

The **--ignore** option can be used to express only a single pattern, however, you can use multiple **--ignore** options for a single CLI command. For example, if you want AccuRev to ignore all **.bak** and **.txt** files, for example, you might enter the following:

```
accurev stat -x --ignore=*.bak --ignore=*.txt
```

You can also specify a file containing ignore patterns and reference that file using **--ignore**. For example:

```
accurev stat -x --ignore="@c:\my_files\ignore.txt"
```

Patterns specified in this file (**ignore.txt**, in this example) must follow the same rules as the **.acignore** file.

See the *CLI User's Guide* or command line help for more information on the **--ignore** option for these commands.

Wildcards in Ignore Patterns

In addition to explicit filenames and paths (`dev_notes.doc`, `/temp/notes/` and `/pubs/design_offsite.txt`, for example) AccuRev ignore patterns support wildcards, as summarized in the following table.

Wildcard	Description	Examples
<code>?</code>	Matches exactly one character, which can be anything other than the path separator (typically <code>/</code>).	<code>pa?t.txt</code> matches <code>pact.txt</code> and <code>part.txt</code> , but not <code>pat.txt</code>
<code>*</code>	Matches zero or more characters, which can be anything other than the path separator (typically <code>/</code>).	<code>pa*t.txt</code> matches <code>pat.txt</code> , <code>pact.txt</code> , <code>part.txt</code> , and <code>paint.txt</code>
<code>[]</code>	Matches exactly one character from the list, which can be anything other than the path separator (typically <code>/</code>). If the list begins with an exclamation mark (<code>!</code>), matches any character <i>not</i> in the list. Use a dash (<code>-</code>) to specify a range of characters.	<code>[aekz]</code> matches <code>a</code> , <code>e</code> , <code>k</code> , or <code>z</code> <code>[a-e]</code> matches <code>a</code> , <code>b</code> , <code>c</code> , <code>d</code> , or <code>e</code> <code>.*ba[kt]</code> matches any file with either a <code>.bak</code> or <code>.bat</code> extension <code>file[!12].cpp</code> matches the <code>file3.cpp</code> and <code>file4.cpp</code> but not <code>file1.cpp</code> and <code>file2.cpp</code>
<code>**</code>	Matches zero or more nested directories. The wildcard must be specified as if it were a directory or a filename itself. That is, it must occur: <ul style="list-style-type: none">At the beginning of the pattern or be preceded by a path separatorAt the end of the pattern or be followed by a path separator	<code>pubs **/*.doc</code> matches any file with a <code>.doc</code> extension in any directory under the <code>pubs</code> directory: <code>pubs/overview.doc</code> and <code>pubs/ui/preferences.doc</code> , for example.
<code>{ }</code>	Matches exactly one word from the comma-separated list. Words in the list may contain other wildcards, including nested <code>{ }</code> wildcards) as well as path separators.	<code>n_{one,t??,s*}.txt</code> matches <code>n_one.txt</code> , <code>n_two.txt</code> , <code>n_ten.txt</code> , <code>n_six.txt</code> and <code>n_seven.txt</code> ,

Specifying Directories and Their Contents

A typical use of an ignore pattern is to have `stat -x` (or the External search in the File Browser) ignore temporary build directories. That is, you want the list of external objects that AccuRev returns to ignore both the directories themselves and all the files within those directories. If the build directories are named `build_001`, `build_002`, and so on, you might be tempted to use this pattern:

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

But this pattern matches only the *contents* of the `build_???` directories and not the directories themselves. (Note that in these examples, a directory matching the pattern that is a subdirectory of another matching directory will be excluded. For example, in a structure like `build_001/build_002`, the `build_002` directory will be excluded, but the `build_001` directory will not.)

Instead, to exclude build directories such as `build_001`, `build_002` and their contents, specify the ignore pattern as follows:

```
build_??*/
```

(The pattern `*/build_??*` would match both `build_001` and `build_002` directories and their contents, but it also might coincidentally match names of some source files, such as `lib/build_end.c`.)

When using path separators, keep the following rules in mind:

- You can use Windows style backslashes (\) or UNIX/Linux style slashes (/) as path separators in your patterns. However:
 - If a pattern contains at least one slash (/), then / is treated as the path separator, regardless of platform, and \ (if any) is treated as an escape character. For example, \? matches ?.
 - If a pattern contains only backslashes (\), AccuRev assumes a Windows path and \ is treated as the path separator. There is no provision for escape sequences in this situation.
- When you start a pattern with a slash (/), AccuRev applies that pattern to that directory only. For example, if you specify an ignore pattern of `/*.exe` at the workspace root, `stat -x` returns all external objects in the workspace; only files with a `.exe` extension *at the workspace root* would be excluded. On the other hand, if you specified an ignore pattern of `*.exe` (no slash) at the workspace root, `stat -x` would exclude files with a `.exe` extension in *any* workspace directory, including those at the workspace root (the same as if you specified recursion by prepending `/**/` to the pattern).
- A slash after a directory name (`daily_build/`, for example) matches the directory itself and its contents.

Using Filenames, Masks, and Lists

The `stat`, `add`, and `files` commands 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`

Using such specifications has two effects:

- The command is restricted to processing a certain set of files, not the whole workspace (even if you also specify `-x`, `-m`, `-n`, or `-p`)
- The command includes *all* specified objects, even if they match an ignore pattern

4. 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 15 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 15 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.

## 5. 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 chapter 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 21 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.



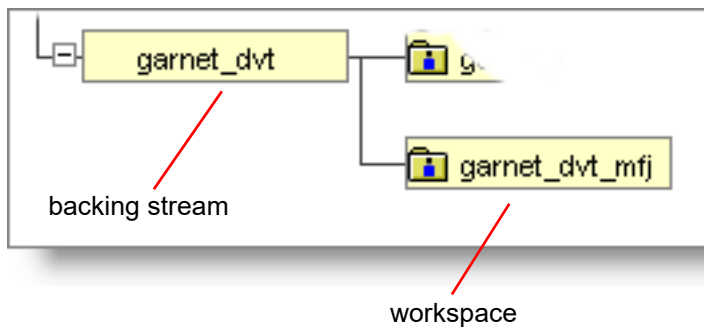
If you invoke *Update* instead of *Populate*, the workspace tree gets the new version of **blue**. No version of **green** is copied to the workspace tree, because *Update* only handles new versions — ones that enter your workspace stream as a result of advancing its update level.

## 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

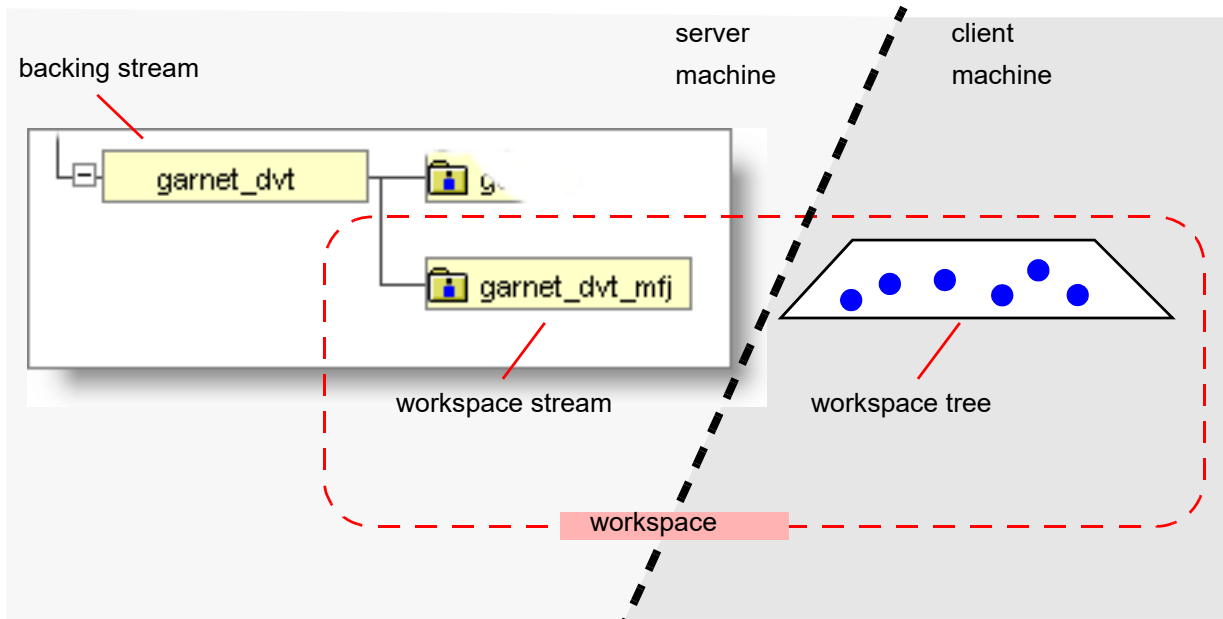
The illustration below shows how a workspace and its backing stream typically appear in the AccuRev StreamBrowser:



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 following sections 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.

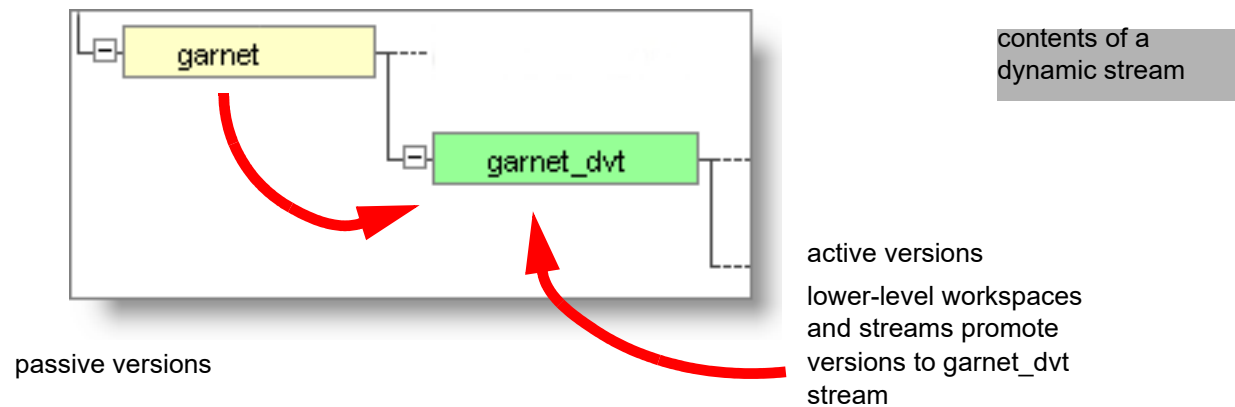
### Backing Stream

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    |
|--------------------------------|---------------|
| <a href="#">doc</a>            | garnet_dvt\1  |
| <a href="#">doc\chap01.doc</a> | garnet_dvt\5  |
| <a href="#">doc\chap02.doc</a> | garnet\3      |
| <a href="#">doc\chap03.doc</a> | garnet_dvt\2  |
| <a href="#">src</a>            | garnet\1      |
| <a href="#">src\garnet.c</a>   | garnet_dvt\12 |
| <a href="#">src\commands.c</a> | garnet_dvt\7  |
| <a href="#">tools</a>          | garnet\3      |
| <a href="#">tools\start.sh</a> | garnet_dvt\6  |
| <a href="#">tools\end.sh</a>   | 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:

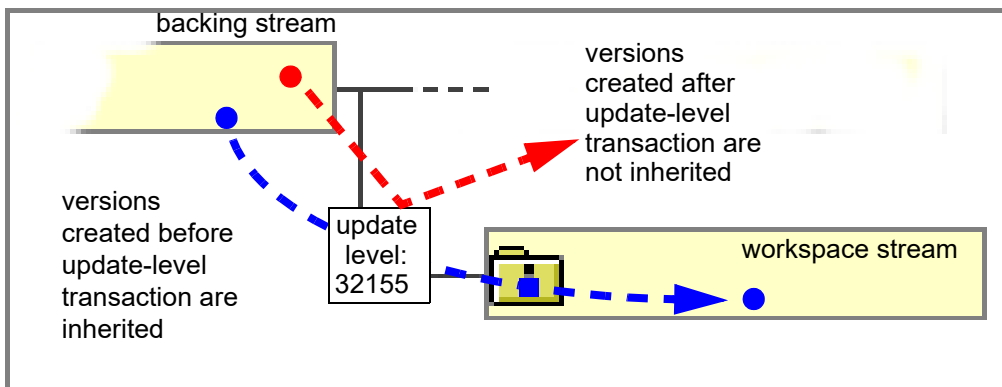
- It's a changing configuration of one of the repository's depots.
- It logically contains a particular version of some or all of the depot's elements.
- It doesn't contain actual files, but is logically just a table of elements and version-IDs.

**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.

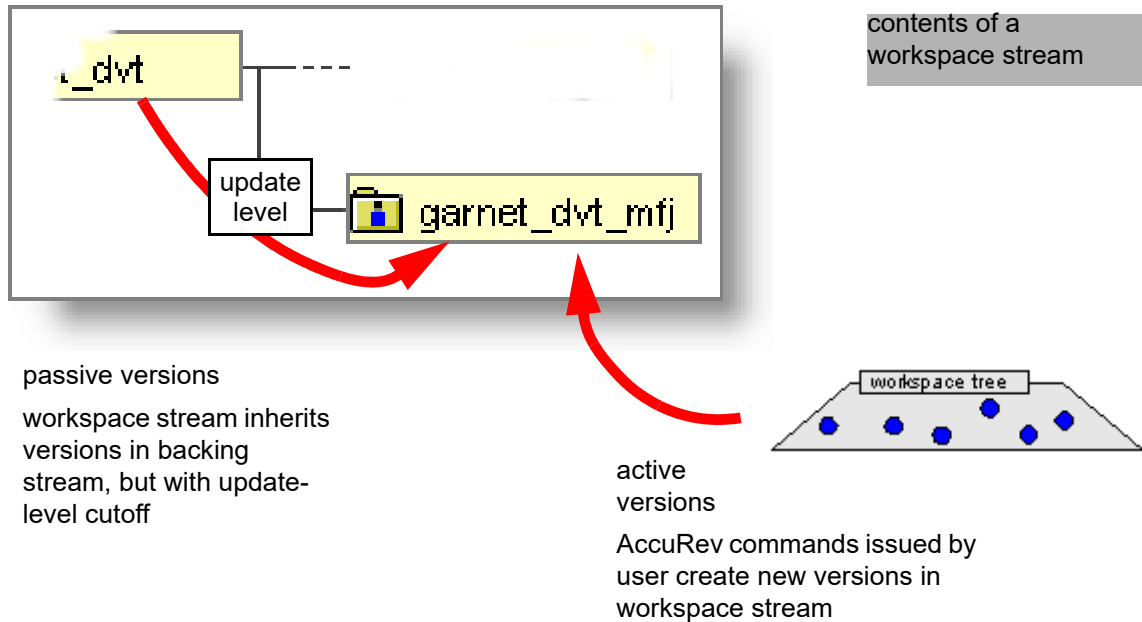
Thus, a workspace stream is not updated dynamically when changes occur to its parent stream. It gets new versions from the backing stream only when you issue an **Update** command. This is how AccuRev implements the workspace’s user-controlled “privateness”, isolating it from the changes regularly being 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**.

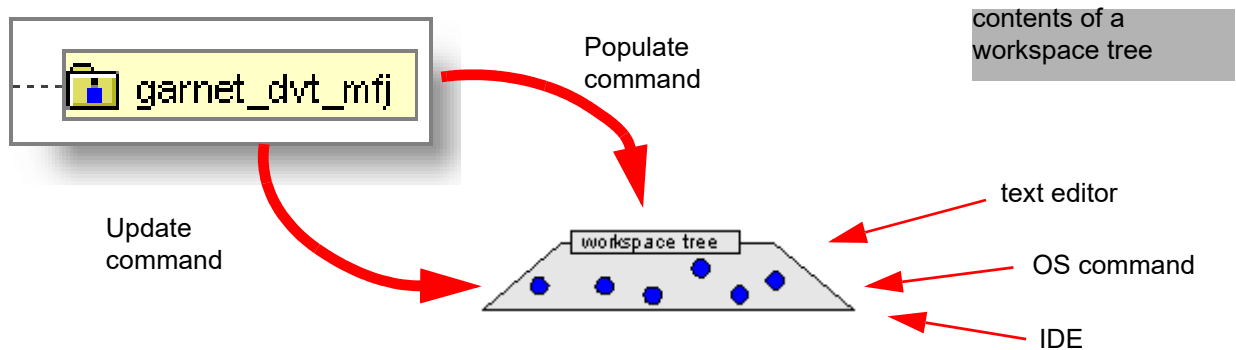


If you modify a file without **Keep**'ing it (or modify it again after **Keep**'ing it), the file in the workspace tree does not exactly match the version in the workspace stream. AccuRev indicates this by reporting the file's status as **(modified)**.

## 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

The processing of the *Update* command is 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 (TSO) to speed its search for files that have been modified in your workspace: it does not consider files whose timestamps have not changed since the last time the workspace was updated or searched for modified files.
  - To make sure that a file with a recent timestamp has actually been modified, AccuRev 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 is 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 local TSO cache will be updated for all files that are updated.

## 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 xlnt 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 xlnt 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 xlnt 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 Basis* 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.



## 6. 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 13 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.



## 7. 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, *Keep*

## 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. Go to the File Browser Incoming Changes mode.
2. If necessary, resolve any elements with (**overlap**) status (select and click the **Merge** button).
3. Click the **Update** button.

## Saving your changes

### CVS

`cvs commit`

### AccuRev CLI

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

### AccuRev GUI

1. Go to the File Browser Outgoing Changes mode.
2. Select the files you want to save and share with others.
3. Click the **Promote** button.

## 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

cvcs 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

cvcs unedit <file(s)>

### AccuRev CLI

accurev purge <file(s)>

### AccuRev GUI

Select files, right-click, *Revert to > Basis 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.

## 8. 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.

### Twin Elements and Stranded Elements

Namespace-related changes :

- “**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”.) AccuRev tracks element names consistently, provides better information about the existence of twins, and provides tools for resolving unintended twin situations.

*Tip:* AccuRev provides a wizard and other features in the AccuRev GUI to help you identify and resolve elements with a (**twin**) status. See [Resolving \(twin\) Status](#) on page 121 in the *AccuRev On-Line Help Guide* for more information.

- “**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 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 namespace-related functionality in detail.

#### Preventing Creation of Twins in Workspaces

Two checks performed by the **add** command help to prevent twins from being created:

- If the user’s workspace currently contains a defunct element at the same pathname, the **add** 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** command is cancelled. If the element is (**defunct**) in the parent stream, an **add** command succeeds.

## 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. (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 Stranded search in the File Browser Outgoing Changes mode, 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.

*Tip:* Twin elements are displayed in the File Browser Conflicts mode.

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

AccuRev supports “refactoring” operations, so that 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 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 Basis** (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 Stranded search in the File Browser Outgoing Changes mode, and equivalently, the AccuRev CLI command *stat -i* 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**).

*Tip:* Twin elements are displayed in the File Browser Conflicts mode.

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 provides tools for relieving situations involving stranded elements. The *move*, *defunct*, and *undefunct* commands 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 39.

## Sophisticated Analysis of Namespace-Related Changes

AccuRev 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/utils/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) uses the 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 Basis*) 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.

## 9. Notes on Cross-Links

This note clarifies and supplements the basic documentation of AccuRev's cross-link feature.

### 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.

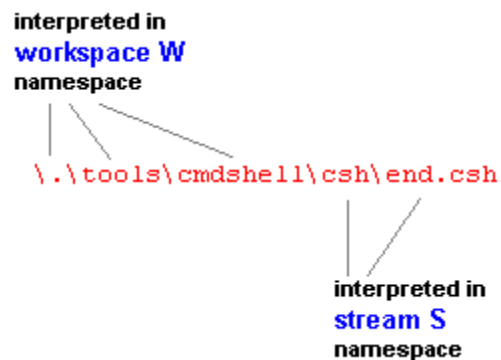
Note that in workspace **W**, you continue to access the cross-linked element, subdirectory **cmdshell**, through its “local” name in the workspace’s namespace. It’s quite possible (but you don’t need to know) that this element has a different name — even a different pathname — in the target stream:

```
\\.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**:

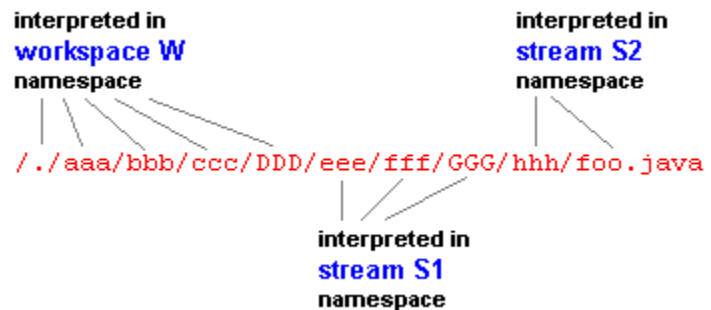
```
./aaa/bbb/cc/ddd/eee/fff/ggg/hhh/foo.java
```

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 48.

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

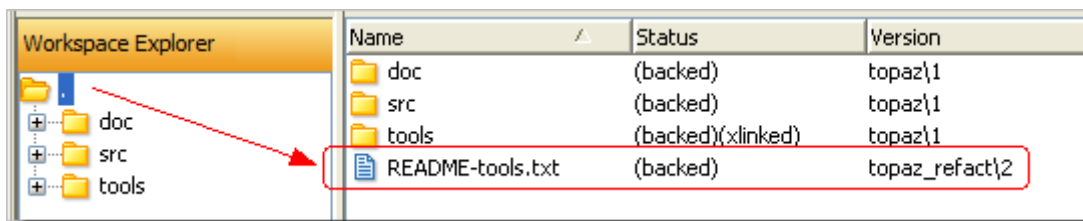
One consequence of AccuRev’s cross-link facility is that two (or more) different versions of the same element can appear at different pathnames in the same workspace or stream. We call this phenomenon double vision. This is not an error — at least, not from AccuRev’s perspective. Seeing the same element 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:

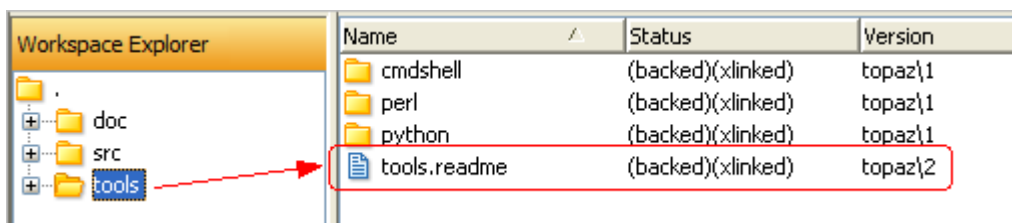
- Flatten out the subdirectories under **tools**.
- Move file **tools.readme** to the depot's root directory, and rename it to **README-tools.txt**.
- Improve the source file comments in the **src** directory.

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 in 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:

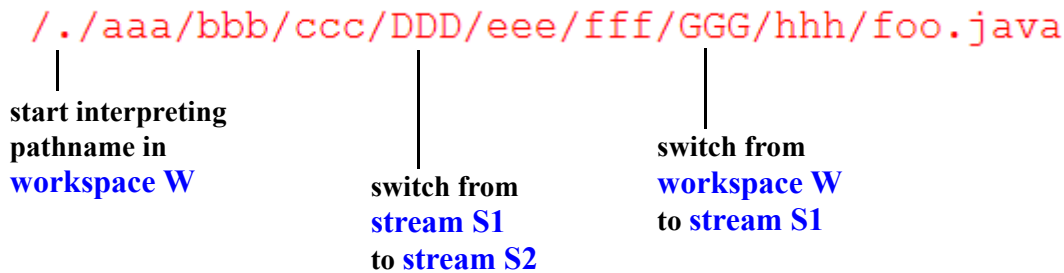
```
> accurev name -e 28 -v topaz_refact
\\.\\tools\tools.readme
\\.\\README-tools.txt
```

## Cross-Link Overlaps

Section [Multiple Cross-Links: Chaining](#) on page 46 describes how a set of cross-links can define a “chain” of backing streams to be used at different components in a pathname:

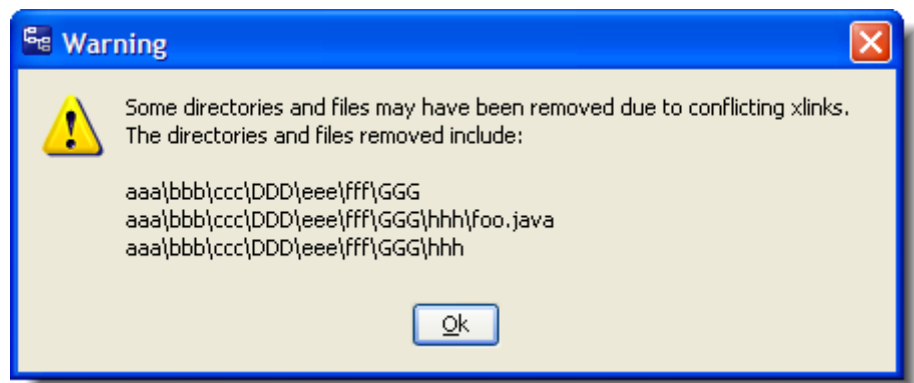


Chaining works correctly if each switch to the next link in the chain occurs at the same pathname component or at a lower component. But here’s a situation that violates this rule:



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.



# 10. Notes on Revert to ... and Diff Against... GUI Commands

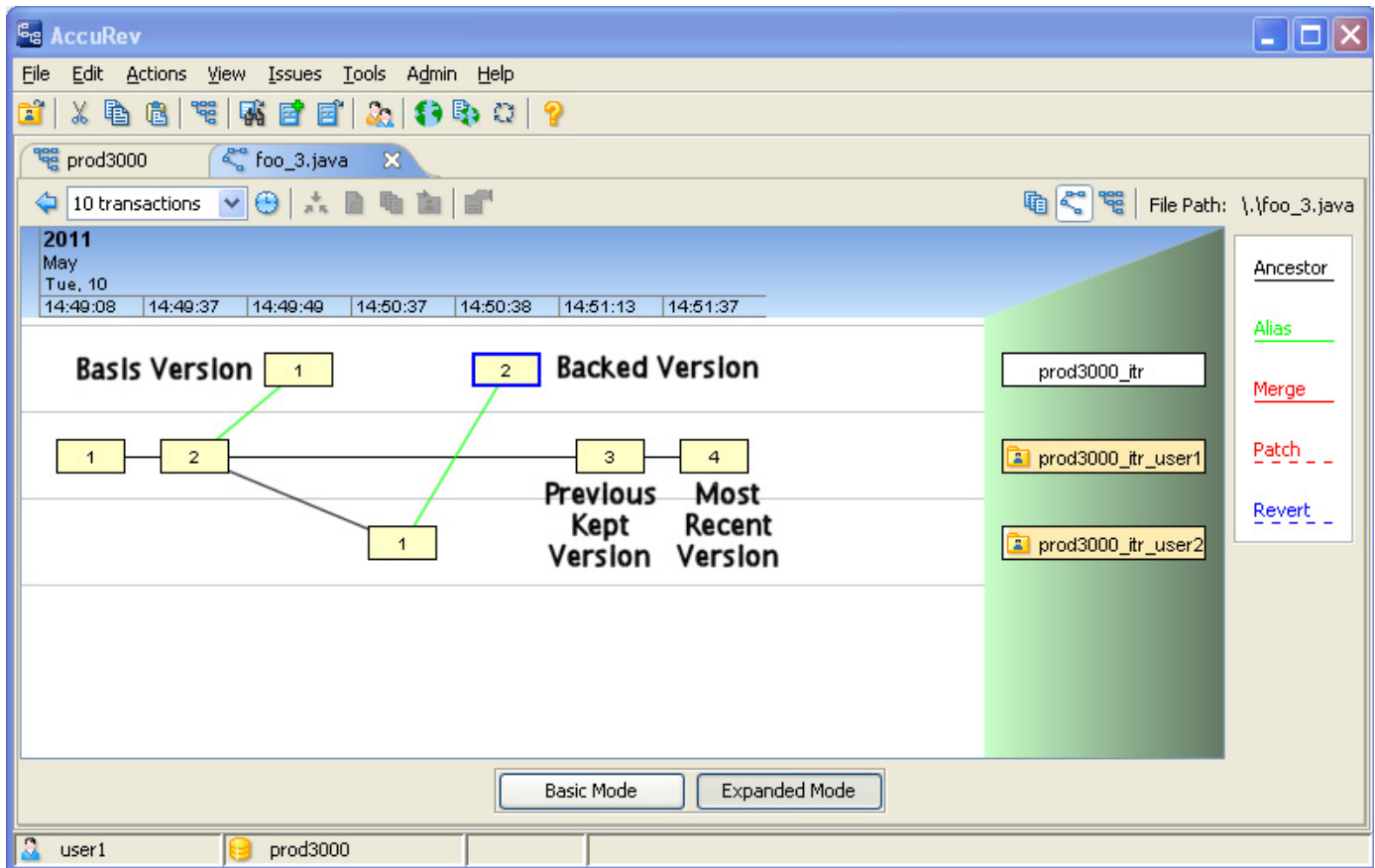
*Revert to Basis* is the GUI version of the CLI *purge* command. The name “Revert to Basis” is something of a misnomer, as the operation would more accurately be named “Revert to Last Update Version” (that is, replace the current workspace version with the version from the backing stream based on the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

