

A Bug's Life

Using Perforce to Propagate and Track Bug Fixes in Evolving Software

(c) 2003 Perforce Software, Inc.



Overview

- Codelines, branches, and evolving software
- How Perforce *jobs* and *fixes* work
- Recommended practices for better bug fix tracking

(c) 2003 Perforce Software, Inc.



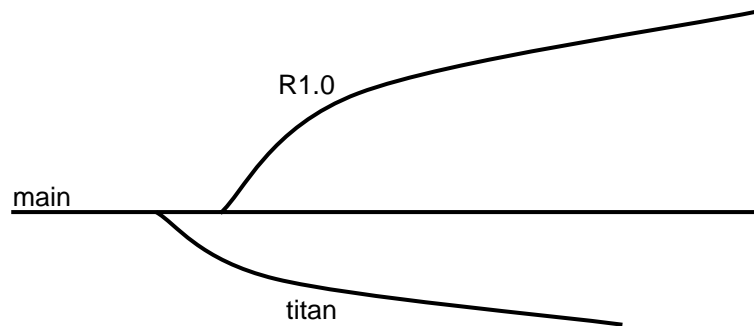
This presentation explores the challenge of figuring out which bugs are fixed where during the software development and release cycle. Perforce has features to help you meet this challenge, but there are some conditions that impact their effectiveness.

First, to lay the groundwork for this discussion, we'll review the familiar concepts of codelines and branches, and the purpose they serve in software evolution.

Then, we'll go over some Perforce-specific constructs: changelists, file revisions, jobs, and fixes. Understanding how these work in Perforce is essential to understanding the gist of this presentation.

With that groundwork, we'll itemize some ways -- some best practices we can use -- to make sure that the way we use Perforce is consistent with the conditions under which Perforce can effectively track bug fixes.

Codelines and Branches



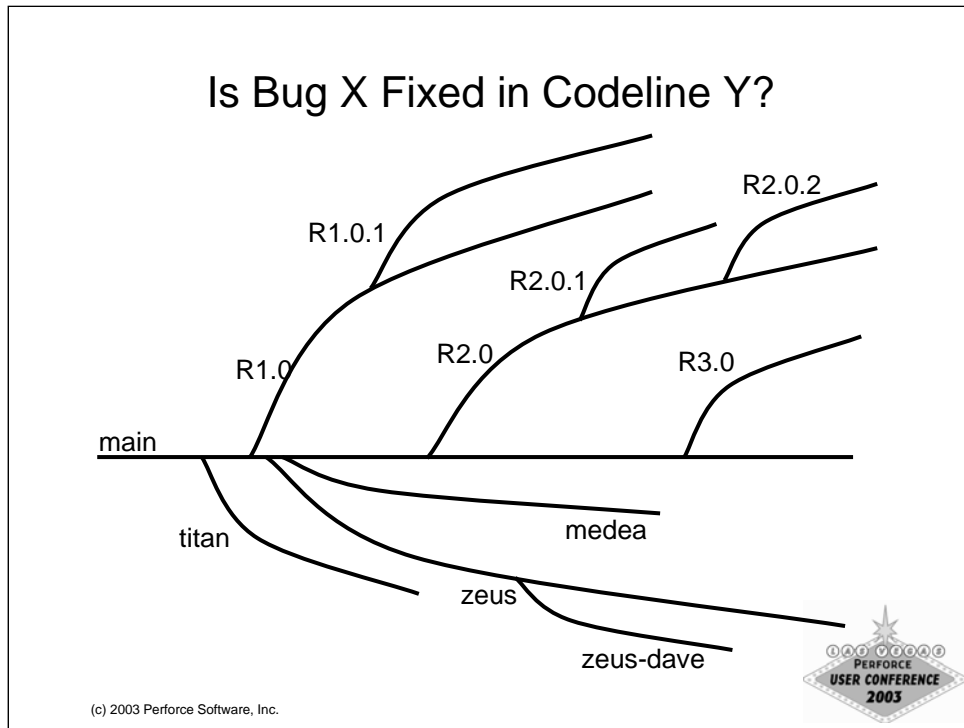
(c) 2003 Perforce Software, Inc.



You are all familiar with the concepts of codelines and branches.

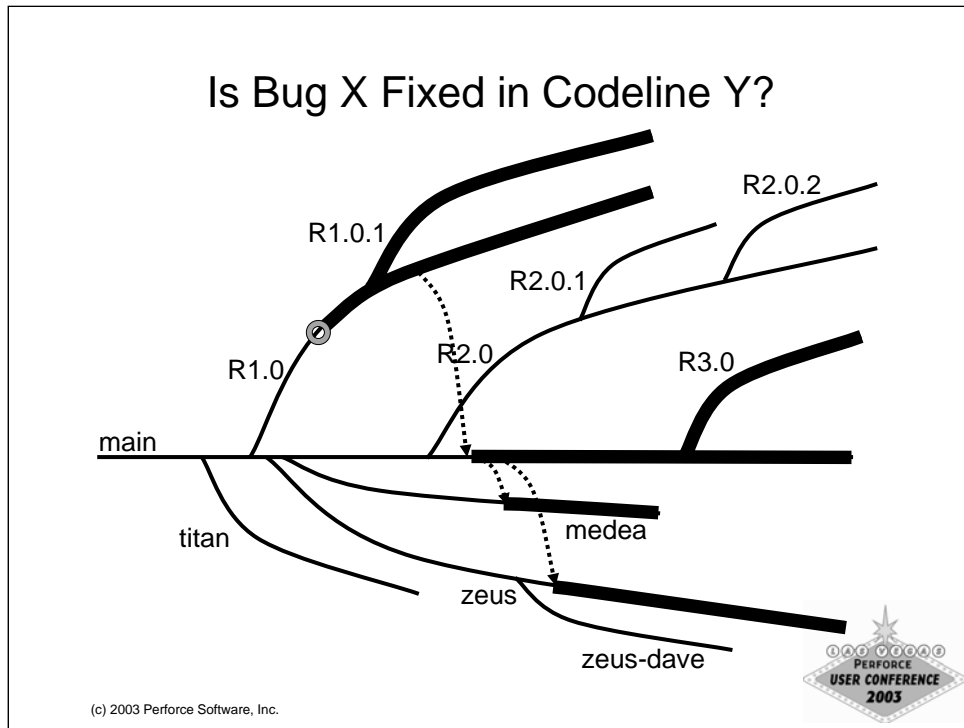
Codelines are collections of files that evolve together. What makes a codeline a codeline is that it contains files that are consistent with each other with respect to their evolution toward a particular goal in the development cycle. Branches are copies of codelines. Codelines are branched so that more than one phase of software development can take place concurrently.

For example, the R1.0 codeline is the collection of files that are being tested and patched for release as Release 1.0 of a product. The Titan codeline is the collection of files that are being enhanced to support a new feature code-named Titan.



Every time a codeline is branched, a new variant of the software it contains is born. Variants start out identical, but as work proceeds, changes are submitted independently to each codeline and, depending on whether the changes are propagated between codelines, the variants start to diverge. Eventually the software embodied by each codeline takes on its own characteristics of quality and functionality. Software in a release codeline, for example, might be very robust, with few known bugs, whereas software in the development project codelines might offer more features but less stability.

In a production software development environment the number of active codelines can grow large. Eventually the one question that comes up is: "Is bug X fixed in codeline Y?" In a development environment where there are many codelines, this question can be difficult to answer.



In some cases, simply diagramming the codeline lineage can reveal the answer. It's easy to see that if a bug is fixed in one codeline at some point in time, any codeline subsequently branched from that codeline will contain the same bug fix. All you have to do is look at the codeline branching diagram to see which codelines contain a bug fix that was made before the codeline was branched.

But what about the case where a bug fix is propagated to the codeline it was branched from? What if the fix was back-ported to an unrelated or distantly related codeline? What if several bug fixes are back-ported at once? In these cases, although branching diagrams could conceivably show the progress of bug fixes through the codelines they affect, there would be so many relationships to diagram that the end result would be very difficult to interpret.

Perforce, it turns out, is innately capable of answering the question "Is bug X fixed in codeline Y?" without the aid of diagrams or recursive analysis.

Commands That Reveal Bug Fixes

➤ Changes

```
p4 changes -i [codeline]
```

➤ Jobs

```
p4 jobs -i job=[bug#] [codeline]
```

➤ Fixes

```
p4 fixes -i -j [bug#] [codeline]
```

(c) 2003 Perforce Software, Inc.



There are three Perforce commands that are capable of reporting whether a particular bug fix has been propagated to a particular codeline. The commands that do this are **changes**, **jobs** and **fixes**. All three commands have an optional form, **-i**, that cause them to report the presence of bug fixes that have been integrated to a particular codeline. In a moment we'll look at the logic behind them so we can understand what they *actually* do.

For now it is worth noting that these commands can detect a bug fix that has been integrated from one codeline to another to affect the target codeline. An integrated bug fix can be detected regardless of which codeline the fix originated in, how many intermediate codelines it passed through, when it was propagated to each of the intermediate codelines, and the path it took before reaching the target codeline.

Changelists

#	12346
User	andy
Client	ws-andy
Date	2002/12/29
Status	submitted
Description	Enable resize button on...

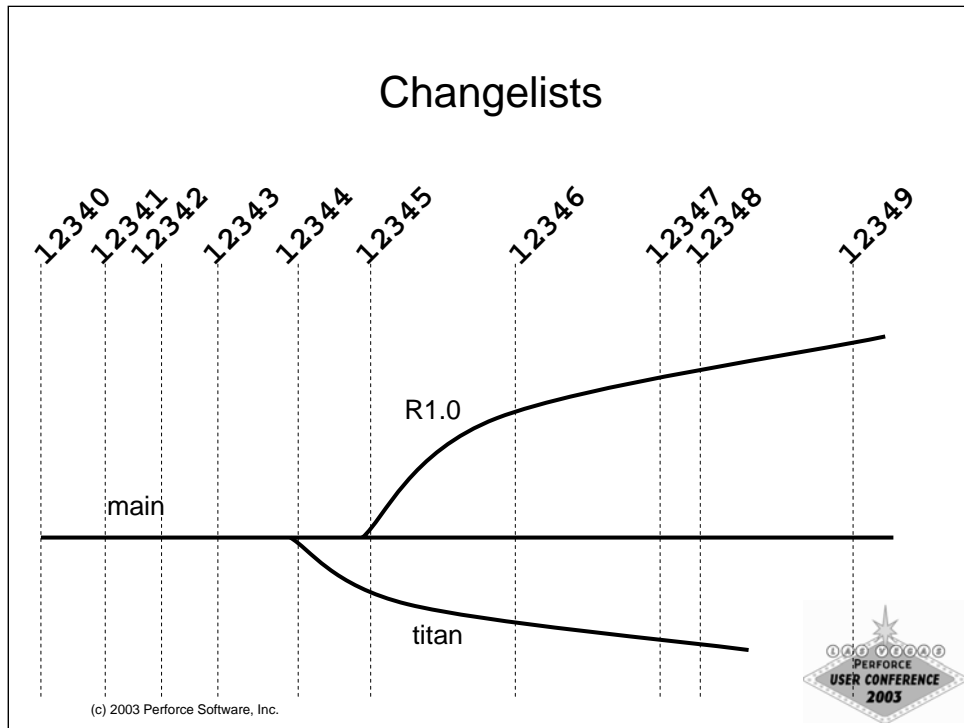
(c) 2003 Perforce Software, Inc.



Before looking at the logic behind Perforce's bug fix tracking ability, let's take a look at some of the objects modeled in the Perforce database.

We'll start with the "changelist" object. 'Submitted changelists' are created in the Perforce database when users submit files. A submitted changelist identifies an update that occurred in the Perforce repository. (The database also records 'pending changelists', which are subtly different from submitted changelists in ways that aren't that relevant here so I won't go into them.)

Changelist records in the Perforce database are keyed on changelist number.



By the way, the number-ID of a changelist is more than just an identifier -- it indicates the order in which the changelist was submitted relative to other changelists. As you know, Perforce updates files one atomic transaction per changelist. So every changelist number indicates an identifiable state of the repository -- that is, a version of the repository.

Files and File Revisions

File Name	<code>//depot/main/img/cursor.png</code>
Revision	<code>4</code>
Type	<code>binary</code>
Action	<code>delete</code>
Changelist	<code>12345</code>

(c) 2003 Perforce Software, Inc.



A file in Perforce is just a name -- the name of the file relative to the root of the repository.

A file revision is an object in the Perforce database schema. It identifies a file at a particular revision. File revision records are uniquely identified by the file's name and a revision number. File revision records also hold an updating action ("add", "edit", "branch", etc.) a changelist number, file type information, and repository location information (used by the Perforce Server to fetch the contents of that file revision).

Integration Records

		start	end
"From" File	//depot/R1.0/img/cursor.png	2	3
"To" File	//depot/main/img/cursor.png	7	7
How Resolved	copy into		
Changelist	12179		
"From" File	//depot/main/img/cursor.png	7	7
"To" File	//depot/R1.0/img/cursor.png	2	3
How Resolved	copy from		
Changelist	12179		

(c) 2003 Perforce Software, Inc.



Perforce stores integration records when users submit files opened for branch or integrate. For the purpose of this discussion, integration records associate triplets of file revisions: the starting rev of the “from” file, the ending rev of the “from” file, and the “to” file rev. Integration records also store a changelist number and a “resolve action”. The changelist number is the changelist related to the update that created the integration record.

Astute observers familiar with relational database normalization will wonder why an integration record stores a changelist number, when file revisions have changelist numbers. The reason is that Perforce always creates two integration records at a time: one in each direction. The second (psuedo) record enables Perforce to determine what revisions to skip when a user attempts to integrate changes in the reverse direction.

The “resolve action” tells how the content of the “from” file end revision relates to the content of the “to” file revision. For instance, if it was merged without user intervention, if it was merged and edited by the user, if it is identical (“copied”), or if it was deliberately ignored.

By the way, the Perforce commands that show integration records are **filelog**, **integrate**, and **integrated**.

The **changes -i** Command

➤ **p4 changes -i //depot/dev/titan/...**

```
Change 1234 on 2003/03/20 by bruce
Change 1220 on 2003/03/18 by ann
Change 1200 on 2003/03/14 by henry
Change 1106 on 2003/02/23 by tracy
Change 1095 on 2003/02/12 by kip
(etc.)
```

```
Change 1200 on 2003/03/14 by henry
Files:
//depot/R1.0/conf/db.cfg#4
//depot/R1.0/db/foo.c#3
//depot/R1.0/db/bar.c#9
```

(c) 2003 Perforce Software, Inc.



Knowing what we know now, we can take a look at how the **changes -i** command works. Let's say we know we fixed a bug by submitting changelist 1200. Now we want to find out if that bug fix is in the Titan development project codeline. (The Titan codeline is everything in the `//depot/dev/titan` path.) We run this command:

p4 changes -i //depot/dev/titan/...

If we see changelist 1200 listed in the output, we know the bug fix is in the Titan codeline. But when we examine changelist 1200 we see that it refers only to files in the R1.0 codeline.

In other words, this bug was fixed in the R1.0 codeline. Somehow, **changes -i** has determined that changelist 1200 (and thus the fix we want) was propagated to the Titan codeline. How did it figure that out?

Well, as you saw in the previous slides, Perforce keeps a record of integration history. By following the trail of integration records emanating from revisions in the target codeline (`//depot/dev/titan`) it can identify the changelists that have contributed content to it.

Jobs

Job	Bug90
Description	Icons very jaggy
Reported By	ron
Date Reported	2003/01/15
Fix By	BETA2
Owner	henry

(c) 2003 Perforce Software, Inc.



Before getting back to the actual topic of tracking bug fixes, we should look at two more Perforce database objects, jobs and fixes.

Jobs are different from other Perforce objects in that their attributes can be custom-defined. Also, their use is entirely optional. However, they can be useful in tracking bug fixes, as we'll see.

All jobs have as an attribute a unique identifier. That is the only attribute of jobs that is relevant at the moment.

Fixes

Change 10713 on 2003/01/14 by pdjames

Files:

```
//depot/r1.0/conf/db.cfg  
//depot/r1.0/img/site.png  
//depot/r1.0/img/user.png
```

Jobs:

```
Bug451 # Wrong background colors...
```

Job	Bug451
Changelist	10713

```
p4 fix -c 1764 Bug135
```

Job	Bug135
Changelist	1764

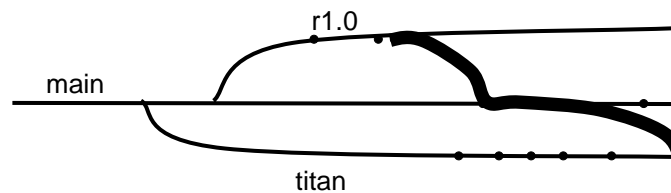
(c) 2003 Perforce Software, Inc.



Perforce can store associations between jobs and changelists. Those relationships are known as “fixes”. Fix records record a job ID and a changelist number. They can be created in the database by submitting a changelist -- there’s a field in the changelist form that accepts job numbers. They can also be created explicitly, after the fact, using the **fix** command.

fixes -i and jobs -i

- `p4 fixes -i -j Bug451 //depot/dev/titan/...`
Bug451 fixed by change 10713 on 2003/01/14
- `p4 jobs -i -e "color*" //depot/dev/titan/...`
Bug451 on 2003/01/14 "Wrong background colors"



(c) 2003 Perforce Software, Inc.

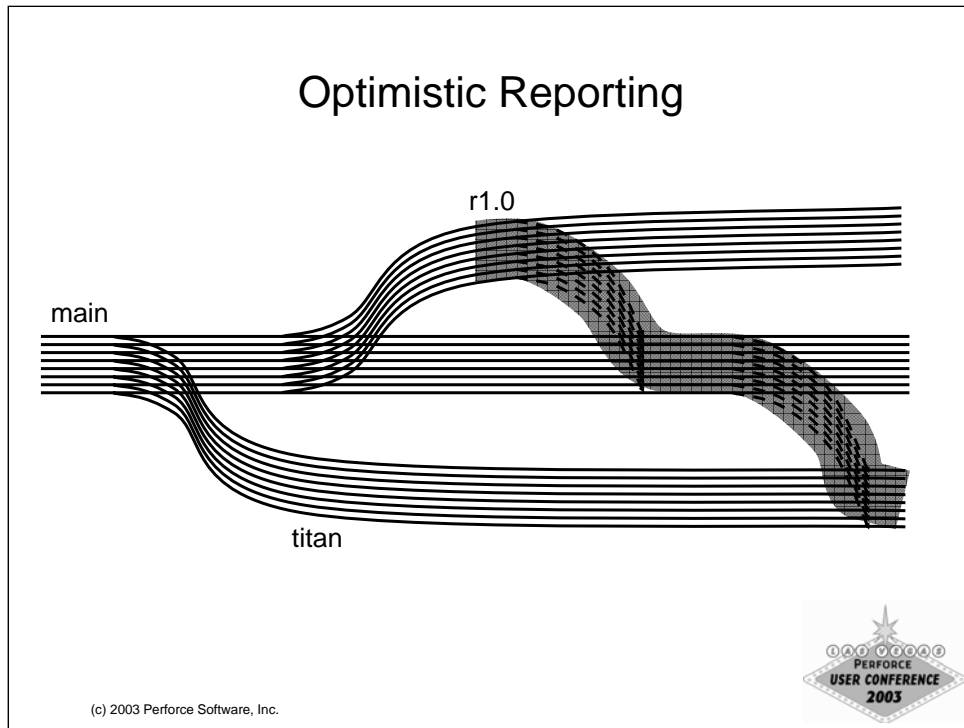


If you have jobs associated with changelists you can do better than the **changes -i** command. **fixes -i** and **jobs -i** can be used to search for specific bug fixes. Whereas **changes -i** outputs all the changelist numbers it finds, **fixes -i** can limit its output to that relevant to a bug ID you supply, and **jobs -i** can limit its output to fixes for jobs matching search parameters you supply.

The **jobs -i** and **fixes -i** commands use the same logic as the **changes -i** command to report results. Starting with the list of all file revisions in the codeline path you supply, they follow the trail of integration records looking for references to file revisions associated with the changelist (or changelists) that are associated by fix records with the job ID you supplied.

So, just knowing a bug ID allows you to find out if that bug is fixed in any of your codelines. No matter where the bug was fixed, if the fix has made its way into the codeline you are curious about, you can find the answer.

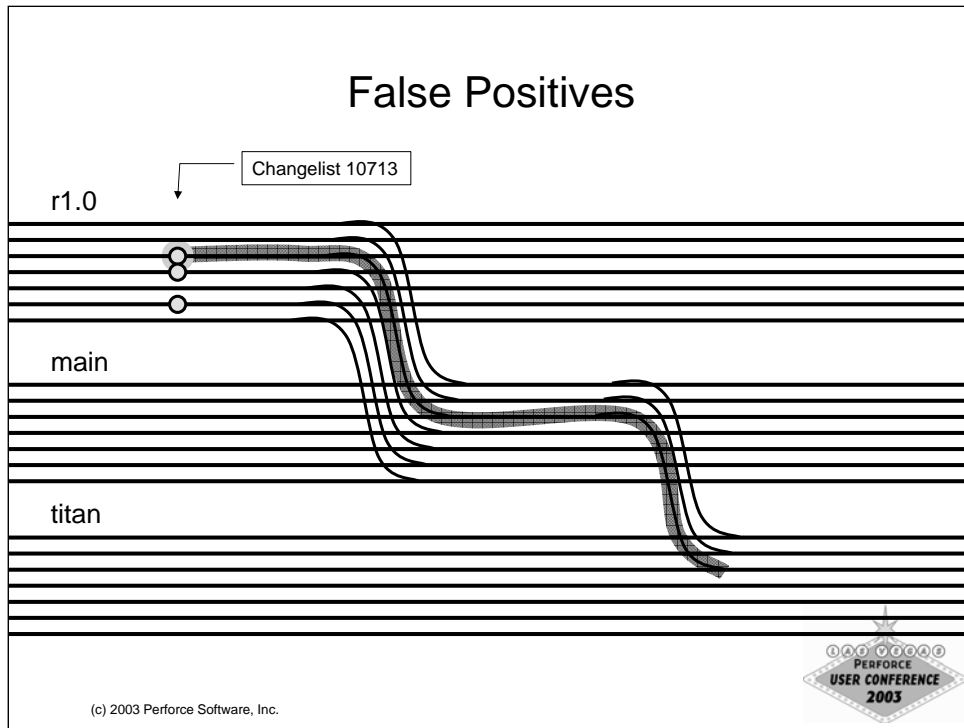
Optimistic Reporting



But what is Perforce actually telling you?

Remember that codelines represent collections of files, each file having its own revision and integration history. If any of the revisions in the codeline you're inspecting are related by integration to the bug fix you're looking for, Perforce will report a positive result.

What Perforce is reporting with fixes -i and jobs -i is the optimistic answer.



So if a bug was fixed by a changelist of 3 files, and, for one reason or another, only one of those files is related by integration history to the codeline you are examining, you may be looking at a false positive.

Perforce's optimistic reporting is based on a couple of premises. First, content diverges between codelines. Because of that, not all revisions involved in a bug fix will apply to codelines the bug fix is being integrated to. Second, changes are always cumulative. Because of that, integrations always involve revisions from all preceding changes. So whether you want them or not, those changes are being integrated too.

Recommended Practices

- Codeline scope is succinct and unambiguous
- Developers submit well-organized changelists
- Changelists are integrated in their entirety
- Changelists are integrated in order
- Fixes are really fixes

(c) 2003 Perforce Software, Inc.



There are a number of things you can do to make sure all your **jobs -i** and **fixes -i** outputs are correct. If you follow these recommendations, you can reduce the likelihood of getting false positives when you look for bug fixes.

Succinct Codeline Scope

➤ Codeline can be represented with a single Perforce file pattern

- Succinct:

Main codeline: `//depot/main/...`
Release 1.0: `//depot/r1.0/...`
Project Titan: `//depot/dev/titan/...`

- Not succinct:

Main codeline: `//depot/main/gui/...`
`//depot/main/db/...`

Release 1.0: `//depot/src/r1.0/...`
`//depot/tests/r1.0/...`

(c) 2003 Perforce Software, Inc.



The first recommendation is that codeline scope is succinct enough that it can be identified with a single Perforce file pattern. (A *file pattern* is a Perforce file syntax expression that is, for the purpose of operations and commands, a single argument. Although the file pattern can refer to more than one file, it itself is a single argument.)

Unambiguous Codeline Scope

➤ A file is a member of at most one codeline

- Unambiguous:

Main codeline: `//depot/main/...`

Project Titan: `//depot/dev/titan/...`

- Ambiguous:

Main codeline: `//depot/main/...`

Project Titan: `//depot/main/titan/...`

- What codeline is this file in?

`//depot/main/titan/doc/index.html`



(c) 2003 Perforce Software, Inc.

In the same vein, codeline scope should also be unambiguous.

If any file in the repository could qualify as a member of more than one codeline, those codelines are defined with ambiguous scope. Ambiguity arises when codelines share a depot path hierarchy.

Given these definitions, is

```
//depot/main/titan/doc/index.html
```

a member of the main codeline? It certainly qualifies, given the file pattern that defines the main codeline, but the answer is ambiguous when the definition of the Titan codeline is taken into account.

Note that the number of levels in the path have nothing to do with ambiguity.

Well-Organized Changelists

➤ Good:

Change 12347 on 2003/01/15 by ann

Fix outdated copyright templates

Change 12330 on 2003/01/14 by henry

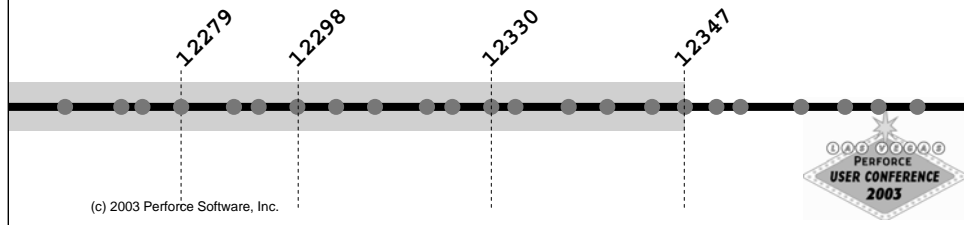
Fixed the jaggies on the file icons

Change 12298 on 2003/01/12 by ann

Fix crash on the file-not-found case

Change 12279 on 2003/01/11 by ann

Refactor .h files for found/not-found logic



The next recommendation is that developers should submit well-organized changelists.

In this example, Ann and Henry have been careful to organize the files they submit into changelists that reflect points of integrity in their work. Each changelist they submit introduces a known condition into the repository: first some refactoring is completed, next a crash is fixed, next some icons are improved, and finally some copyright dates are updated.

Well-Organized Changelists

➤ Bad:

Change 12345 on 2003/01/15 by ann

Oops, some more .h files to fix crash

Change 12333 on 2003/01/14 by ann

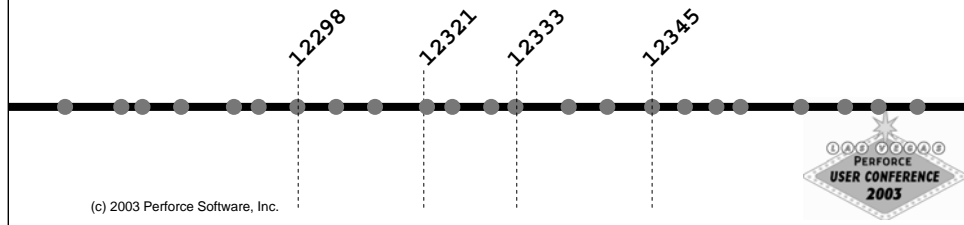
One more copyright template

Change 12321 on 2003/01/13 by henry

Fixed the jaggies on the file icons

Change 12298 on 2003/01/12 by ann

Work on crash and fix some copyright templates



Here is an example of poorly organized changelists, taken from a parallel universe. In this universe, Ann is bad; she has not been careful about organizing her changelists.

To use any of her changelists to refer to a point of integrity would not be particularly useful, as none of her changelists refer to a point at which anything she worked on is complete. Even using Henry's changelist number to identify a point of integrity is impossible because his changelist was submitted between two of Ann's half-baked submissions.

Changelist Etiquette

- Break up large tasks into smaller logical changes, each of which can be completed separately.
- Make sure all the files involved in a particular logical change are opened and submitted in the same changelist.
- Use separate changelists, or even separate workspaces, to submit concurrent work on unrelated or independent logical changes.
- Submit a changelist only when the logical change it addresses is complete.

(c) 2003 Perforce Software, Inc.



While Bad Ann seems to use changelists to submit random files at random intervals, Good Ann observes this etiquette when submitting changelists.

Integrating Entire Changelists

```
Change 12340 on 2003/01/14 by henry
```

Files:

```
//depot/r1.0/conf/data.cfg  
//depot/r1.0/img/fopen.png  
//depot/r1.0/img/fclosed.png
```

Jobs:

```
Bug90 # Icons very jaggy
```

- Integrating by subdirectory gets *partial* changelists:

```
p4 integ //depot/r1.0/img/... //depot/main/img/...
```

- Integrating by codeline is best:

```
p4 integ //depot/r1.0/... //depot/main/...
```



(c) 2003 Perforce Software, Inc.

Another recommendation is to always integrate entire changelists.

This rule is easy to violate. For example, consider this changelist submitted by Henry to fix Bug90. It affects files in two subdirectories of the R1.0 codeline.

Now someone comes along and does an integrate between R1.0 and main, but only integrates one subdirectory. While perfectly legal in Perforce, this operation destroys the integrity of Henry's bug fix. The main codeline contains the updated file icons associated with Henry's bug fix, but it does not contain the `data.cfg` file Henry updated when he submitted his changelist.

If you were to run **fixes -i** or **jobs -i** on `//depot/main` looking for Bug90, you'd get a false positive. There are some files in `//depot/main` that are related by integration to Bug90, but those commands can't tell you that some revisions related to Bug90 have *not* been integrated.

Integrating by codeline is the best way to make sure you integrate entire changelists.

Integrating in Order

```
Change 12347 on 2003/01/15 by ann
```

```
Files:
```

```
  //depot/r1.0/conf/copyright.xml  
  //depot/r1.0/conf/data.cfg
```

```
Jobs:
```

```
  Bug150 # Copyright date is wrong
```

- Integrating by changelist can get out of order:

```
p4 integ //depot/r1.0/...@12347,12347 //depot/main/...
```

- Integrating with no revision range is best:

```
p4 integ //depot/r1.0/...@12347 //depot/main/...
```

(c) 2003 Perforce Software, Inc.



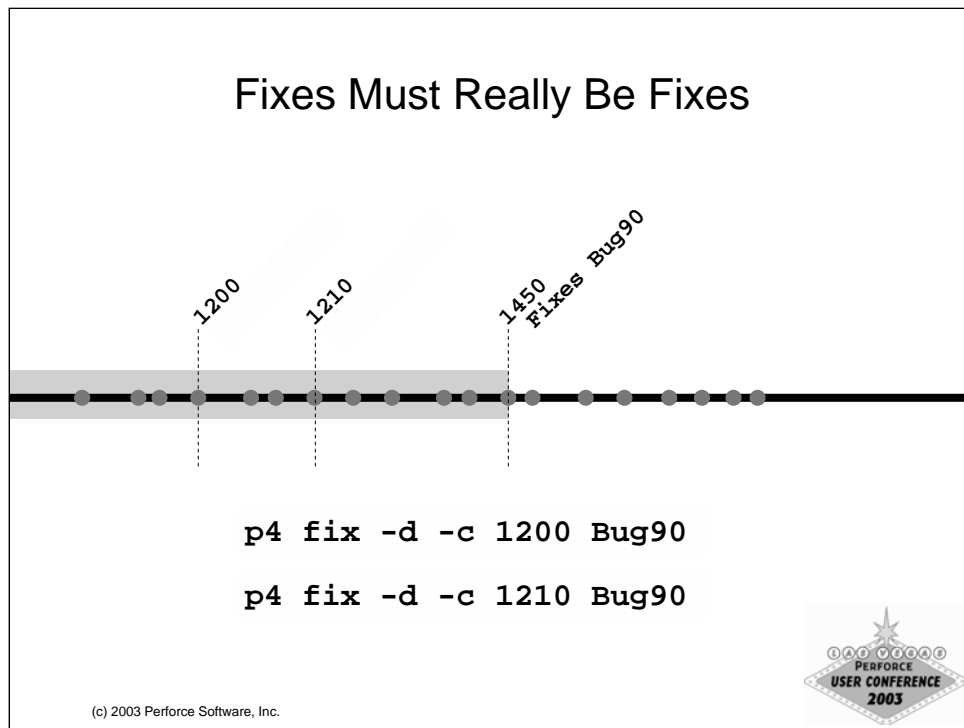
For the same reason, it's important to integrate changelists in order.

Consider this changelist submitted by Ann. It was submitted after Henry's change, and it affects the same configuration file Henry updated.

Say someone comes along and integrates *just* this change from R1.0 to main. (That is, some integrates using a revision range of one changelist.) Again, perfectly legal in Perforce, but again, it destroys the integrity of Henry's change.

As a rule, it is better to *not* use revision ranges when integrating bug fixes. You can integrate *up to* a changelist number without jeopardizing the integrity of bug fixes, but if you skip over changelists, and/or integrate them out of order, you introduce conditions under which **jobs -i** and **fixes -i** will report false positives.

Fixes Must Really Be Fixes



The last rule to remember is that when you fix a bug with a changelist, that changelist really has to fix the bug. In other words, that changelist must be the point of integrity with respect to that bug fix (in its originating codeline).

Say you fix Bug90 with changelist 1200. Then you realize you missed something so you submit changelist 1210, also to fix Bug90. Later someone else finds a flaw in your work and you fix it *again* with changelist 1450. That finally fixes it for good.

Perforce is perfectly happy to create three fix records for you. Each one associates the same job ID with a different changelist. But at what point in the life of the R1.0 codeline is Bug90 really fixed? As of changelist 1450.

For the purpose of tracking bug fixes, the fix records that associated Bug90 with changelists 1200 and 1210 are red herrings. If changes prior to 1450 are integrated from R1.0 to another codeline, Perforce will report Bug90 as being fixed in that codeline -- another false positive.

To prevent that from happening, you should remember to remove the fix records for fixes that aren't. You can do that with **fix -d**.

Unexpected Benefits

- Easier integration operations
 - Merging is easier when changes are integrated in order and in their entirety
- Better performance
 - Succinct codeline scope means tighter client views, branch views, etc.
- Simplifies planning
 - Codeline diagrams represent content evolution
 - No mystery about what changes end up where

(c) 2003 Perforce Software, Inc.



These recommendations are extreme; are they too extreme? Could you actually put these practices to use under production development and release assembly line conditions?

Yes you could. In fact, if anything, these practices *simplify* things.

First, it's easier to do integrations and merging if you don't have to cherry-pick changelists or subdirectories.

Second, your branch views (and maybe even your client views) will be simpler, because the scope of your codelines is so succinctly defined. That usually improves performance.

Finally, it's easier to plan branching and codeline management. For starters, just diagramming your branches is easier when you stick to the rules of integrating entire changelists and integrating in order.

This is not to say that exceptions won't arise, or that these recommendations work for every single development and release methodology. But they are consistent with good SCM in production environments.

Conclusion

- Perforce can tell you which bugs are fixed in which codelines
- Certain practices improve the accuracy of bug fix tracking
- The same practices also improve performance, simplify use, reduce complexity

(c) 2003 Perforce Software, Inc.



As you saw, Perforce can answer the question “Is Bug X fixed in Codeline Y?” In particular, the two commands you can use to get it to answer those questions are: **fixes -i** and **jobs -i**

There are certain recommended practices developers can follow to improve the accuracy of the information yielded by **fixes -i** and **jobs -i**. These practices have to do with how changes are submitted, how codelines are defined, and how changes are integrated between codelines.

It turns out that the best practices are also the easiest practices. Though they may seem draconian, their effect is to keep variance in check and reduce complexity. This results in better performance, simpler integrations, and easier planning.