What are the basic clearcase concepts every developer should know? [closed]

Asked 15 years, 9 months ago Modified 5 years ago Viewed 101k times

65

votes

As it currently stands, this question is not a good fit for our Q&A format. We expect answers to be supported by facts, references, or expertise, but this question will likely solicit debate, arguments, polling, or extended discussion. If you feel that this question can be improved and possibly reopened, visit the help center for guidance.

Closed 12 years ago.

Locked. This question and its answers are locked because the question is off-topic but has historical significance. It is not currently accepting new answers or interactions.

What are the core concepts of the Clearcase version control system every developer should know?

clearcase

Share

edited Feb 19, 2013 at 15:03

George Stocker
57.9k ● 29 ● 180 ● 238

asked Mar 13, 2009 at 23:49

user32262
8,830 ● 23 ● 66 ● 78

2 To any moderator, **please don't delete this question** (pretty please). I understand it isn't a good fit to the current SO standards, but this represents 12 years of experience on that tool (ClearCase) and should bring lasting added value for the poor souls stuck with that VCS. – VonC Feb 1, 2013 at 8:53

Comments disabled on deleted / locked posts / reviews

7 Answers

Sorted by: Highest score (default)

144 votes

Core concepts?

- **centralized(-replicated) VCS**: ClearCase is halfway between the centralized VCS world (one or several "centralized" repos or VOBS - Version Object Bases - every developer must access to commit) and the distributed VCS world.
  But it also supports a "replicated" mode allowing you to replicate a repo in a distant site (MultiSite ClearCase), sending deltas, and managing ownership. (the license fees attached with that is quite steep though)
  This is not a true "decentralized" model, since it is does not allow for parallel *concurrent* evolutions: The branches are mastered in one VOB or another; you

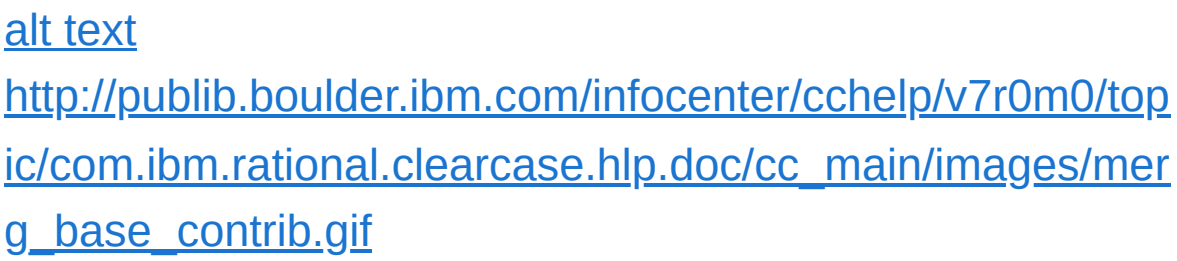
can only check-in to the master VOB for the branches mastered there, though you have readonly access to any branch at any replica.

- **linear version storage**: each file *and directory* has a linear history; there is no direct relationship between them like the DAG VCS (Directed Acyclic Graph) where the history of a file is linked to the one of a directory linked to a commit.
  That means
  - when you compare two commits, you have to compare the list of all files and directories to find the delta, since commits are not atomic across files or directories, meaning there is no single name for all the changes to all the files that make up a logical delta.
  - That also means a **merge** must find a common *base contributor* (not always the same as a common ancestor) through history exploration (see next point).

    (Git is at the opposite end of that spectrum, being both decentralized and DAG-oriented:

    - if a node of its graph has the same "id" as a node of a different commit, it does not have to explore further: all the sub-graphs are guaranteed to be identical
    - the merge of two branches is actually the merge of the content of two nodes in a DAG:

recursive and very quick for finding a common ancestor)



- **3-way merging**: to merge two versions, ClearCase must find a common based contributor in their linear history, which can be fairly long for complex version tree (branch/sub-branch/sub-sub/branch, ...), and the basic ClearCase merge command merges a file or directory, but it is not recursive. It only affects a singe file, or a single directory without its files (`ct findmerge` is recursive)
- **file-centric** (as opposed to the other recent VCS more repository centric): that means the commit is file by file, not "set of modified files": the transaction is at the file level. A commit of several files is not atomic. (Almost every other *modern* tool is "repository centric", with an atomic commit transaction, but first-generation systems like RCS, SCCS, CVS, and most other older systems do not have that feature.)
- **id-managed**: each file and directory has a unique id, meaning they can be renamed at will: their history will not change since the id remains for the "element". Plus a directory will detect in its history any addition/suppression of file. When a file is "removed" (`rmname`), it does not know it: only the directory is

notified and creates a new version in its history, with a list of sub-elements not including the file removed.

(Create two files with the same size and content, they will get the same id in Git -- a SHA1 key -- and will be stored only once in the Git repo! Not so in ClearCase.
Plus, If two files with the same path and name are created in two different branches, their id being different means those two files will never be merged: they are called "*evil twins*")

- **branches are first-class citizens**: most VCS consider a branch and a tag as the same: a single point in the history from which a new linear history can grow (branch) or from where a description is attached (tag).
  Not so for ClearCase, where a branch is a way to reference a version number. Any version number starts at 0 (just referenced in ClearCase) to 1, 2, 3, and so on. Each branch can contain a new list of version numbers (0, 1, 2, 3 again).
  This is different from other systems where the version number is unique and always growing (like the revisions in SVN), or is just unique (like the SHA1 keys in Git).
- **path-accessed**: to access a certain version of a file/directory, you need to know its extended path (composed of branches and versions). It is called an "extended path name":
  `myFile@@/main/subBranch/Version`.

(Git does refer to everything through id -- SHA1-based --: version [or commit], tree [or version of a directory] and blob [or version of a file, or rather of a *content* of a file]. So it is "id-accessed" or "id-referenced".
For ClearCase, an id refers to an "element": a directory or a file, whatever its version is.)

- **both pessimistic lock and optimistic lock**:
  (reserved or unreserved checkouts in ClearCase):
  even a pessimistic lock (reserved checkout) is not a true pessimistic one, since other users can still checkout that file (albeit in "unreserved mode"): they can change it but will have to wait for the first user to commit his file (checkin) or cancel the request. Then they will merge their checkout version of that same file.
  (Note: a "reserved" checkout can release its lock and be made unreserved, either by the owner or the administrator)
- **cheap branching**: a branch does not trigger a copy of all files. It actually triggers nothing: any file not checkout will stay in its original branch. Only modified files will have their new versions stored in the declared branch.
- **flat-file storage**: the VOBs are stored in a proprietary format with simple files. This is not a database with an easy query language.
- **local or network workspace access**:

    - local workspace is through checkout to the hard-drive ("update" of a snapshot view).
    - network workspace is through dynamic view, combining versioned files and directories access through network (no local copy, instant access) and local files (the ones which are checked out or the private files). The combination of distant (versioned) and local (private) files allows a dynamic view to appears like a classic hard drive (whereas actually any file "written" is stored in the associated view storage).
- **centralized deported storage**: [view] storage is there to keep some data and avoid some or any communication with the central referential.
  a workspace can have:
    - a scattered storage: like the `.svn` subdirectories all over the place
    - a centralized storage: like the view storage in ClearCase, they contain information about the files displayed by the view, and that storage is unique for a view.
    - a deported storage: the storage is not part of the view/workspace itself, but can be located elsewhere on the computer or even outside on the LAN/WAN

(Git has no "storage" per se. Its `.git` is actually all the repository!)

- **meta-data oriented**: any "key-value" can be attached to a file or a directory, but that couple of data is not historized itself: if the value changes, it overrides the old value.

(meaning the mechanism is actually weaker than the "properties" system of SVN, where properties can have an history;
Git on the other end is not too keen on meta-data)

- **system-based protection**: the owner and the rights associated with a file/directory or repositories are based on the rights-management of the underlying system. There is no applicative account in ClearCase, and the group of users are directly based on the Windows or Unix existing group (which is quite challenging in an heterogeneous environment, with Windows clients and a Unix VOB server!)

(SVN is more like "server-based" protection, where the Apache server can get a first level of protection, but must be completed with hooks to have a finer grain of rights.
Git has no direct rights management and must be controlled by hooks during push or pull between repositories)

- **hooks available**: any ClearCase action can be the target of a hook, called trigger. It can be a pre or post operation.
- **CLI managed**: cleartool is the Command Line Interface from which all actions can be made.

Share

edited Jun 28, 2012 at 15:30

eykanal

27k ● 19 ● 85 ● 114

answered Mar 14, 2009 at 11:10

VonC

1.3m ● 558 ● 4.7k ● 5.6k

19

votes

ClearCase is a beast to use. Slow, buggy, and expensive. Some things that I have done to cope with using CC are:

1. Always put good comments when you check in.
2. Use a common config spec and don't change it very often.
3. Never try to use CC over a VPN or slow network connection.
4. Turn off the loading off CC doctor on startup.
5. Don't move files around to different directories.
6. Schedule at least 2 min per file for checkin.
7. Snapshot views are slow, but dynamic views are slower.
8. Make a development habit of checking in early and often because the reserved files and merges are painful.
9. Have all the developers check out files in unreserved by default.

Share

answered Mar 14, 2009 at 0:01

Joshua
26.7k • 22 • 79 • 107

6 I disagree on the beast comment. I think what you're missing is a CC administrator that knows what they're doing. Yes, CC is a complicated system and we've had troubles with it, but not since we hired someone that knows it well. It's not something I'd use for casual source control. – paxdiablo Mar 14, 2009 at 0:06

+1 for item number 1, even though it is not CC specific ;-)
– Treb Mar 14, 2009 at 0:30

If your checkins take 2 minutes per file, well, you've got serious problems with your setup. That is grotesquely out of whack!
– Jonathan Leffler Mar 14, 2009 at 4:52

3 BTW, we have a full time CC admin. 10 years ago, this would have been acceptable. Based on the tools today, not so much.
– Joshua Mar 14, 2009 at 14:25

The reason it takes so long for checkins is the VERY inefficient protocol, it is connected to the networks LDAP, and it needs to talk to the ClearQuest server to verify the issue report, and then also apply the CQ labels. – Joshua Mar 14, 2009 at 14:27

16 votes

We've been using CC for just over fifteen years now. It has a lot of good features.

All our development is done on branches; I created a couple today, for a couple of different sets of changes. When I'd checked into the branch, I got a colleague to review the changes, and then merged back into /main/LATEST - which happens to be where my work

needed to go. If it had been for an older release on a branch, it wouldn't have been any harder.

The merges from my temporary branches were fully automatic; no-one had changed the files I worked on while I had them checked out. Although by default checkouts are reserved (locked), you can always unreserve the checkout later, or create the checkout unreserved. When the changes take multiple days, the resynchronization of my temporary branch with the main branch is easy and usually automatic. The mergetool is OK; the biggest problem for me is that my server machine is 1800 miles or so from my office (or home), so that X over that distant is a bit slow (but not intolerably so). I've not used a better mergetool, but that may not be saying much since I've not used any other graphical mergetool.

Views (dynamic views) are fast on our setup. I've not used snapshot views, but I don't work on Windows when I can help it (our team uses snapshot views on Windows; I'm not clear why). We have complex branching systems, but the main development is done on /main/LATEST, and the release work is done on a branch. After GA, maintenance work is done on a release specific branch, and merged forward to /main/LATEST (via any intermediate versions).

CC does need good administrators - we have them and are fortunate in doing so.

CC is not trivial to use, though at the moment, I find 'git' as daunting as CC is to those who've not used it. But the basics are much the same - checkout, change, checkin,

merge, branch, and so on. Directories can be branched - cautiously - and certainly are version controlled. That is invaluable.

I don't see the office switching from CC any time.

Embedded Version Numbers - Good or Evil?

I wrote:

> The biggest problem I have with CC is that it does not embed version numbers into the source files - a problem that git has too, AFAICT. I can half see why; I'm not sure I like giving up that trackability, though. So, I still use RCS (not even CVS) for most of my personal work. One day, I may switch to git - but it will be a jolt and it will take a lot of work to retool the release systems configured around (SCCS and) RCS.

In response, @VonC notes:

> We always considered that practice as evil (mixing meta-data information into data), introducing "merge hell". See also How to get Clearcase file version inside a Java file. Of course, you can use a trigger for RCS keyword substitution (Clearcase

> Manual: Checkin Trigger Example) provided you use an appropriate merge manager.

There are several issues exposed by this discussion, and they all get mixed together. My views verge on the archaic, but have a rationale behind them, and I'm going to take the time to write them down (messed up by life - it may take several edits to complete this).

Background

I learned SCCS back in 1984, about the time RCS was released (1983, I believe), but SCCS was on my machine and the internet was nascent at best. I moved from SCCS to RCS reluctantly in the mid-90s because the SCCS date format uses double-digits for years and it was not clear whether SCCS would be universally fixed in time (it was). In some respects, I don't like RCS as much as SCCS, but it has some good points. Commercially, my employer used SCCS up to mid-1995, but they started to switchover to Atria ClearCase from early 1994, tackling separate product sets one at a time.

Early ClearCase Experiment with Triggers - and Merge Hell

Our project migrated later, when there was already some experience with CC. Partly because I insisted on it, we embedded version control information in the source files via a check-in trigger. This lasted a while - but only a while -

because, as VonC states, it leads to merge hell. The trouble is that if a version with the tag /main/branch1/N is merged with /main/M from the common base version /main/B, the extracted versions of the files contain a single line which has edits in each version - a conflict. And that conflict has to be resolved manually, rather than being handled automatically.

Now, SCCS has ID keywords. ID keywords take two formats, one for files being edited and one for files that are not being edited:

```
Edit          Non-Edit
%I%           9.13
%E%           06/03/09
%Z%           @(#)
%M%           s.stderr.c
```

If you attempted a 3-way merge of the editable versions of SCCS files (with the %x% notation), then there would be no conflicts on the lines containing metadata unless you changed the metadata on those lines (e.g. by changing from US-style %D% dates to UK-style %E% dates - SCCS does not support ISO-style 2009-03-15 dates as standard.)

RCS also has a keywords mechanism, and the keywords also take two formats, though one is for files which have not yet been inserted into RCS and the other is for those that have:

```
Original       After insertion
$Revision$     $Revision: 9.13 $
```

```
$Date$          $Date: 2009/03/06 06:52:26 $
$RCSfile$       $RCSfile: stderr.c,v $
```

The difference is between a '$' following the keyword and a ':', space, text, space and finally a '$'. I've not done enough merging with RCS to be sure what it does with keyword information, but I note that if it treated both the expanded and 'contracted' notations as equivalent (regardless of the content of the expanded material), then merging could take place without conflict, leaving the contracted notation in the output of the merge, which would be appropriately expanded when the resulting file is retrieved after checkin.

The ClearCase problem is the absence of an appropriate merge manager

As I've indicated in my discussion of SCCS and RCS, if 3-way merging is done treating the keywords in the correct (contracted or editable) formats, then there is no merge conflict.

The problem with CC (from this viewpoint - clearly, the implementors of CC disagree) is that it lacks a system for handling keywords, and therefore also lacks an appropriate merge manager.

If there was a system for handling keywords and an appropriate merge manager, then:

- The system would automatically embed the metadata into files at appropriate markers.

- On merge, the system would recognize that lines with the metadata markers do not conflict unless the markers changed differently - it would ignore the metadata content.

The downside of this is that it requires either a special difference tool that recognizes metadata markers and treats them specially, or it requires that the files fed to the difference tool is canonicalized (the metadata markers are reduced to the neutral form - $Keyword$ or %K% in RCS and SCCS terms). I'm sure that this little bit of extra work is the reason why it is not supported, something I've always felt was shortsighted in such a powerful system. I've no particular attachment to RCS or SCCS notations - the SCCS notations are easier to handle in some respects, but they're essentially equivalent - and any equivalent notation could be used.

Why I still think metadata in the file is good

I like to have the metadata in the source code because my source code (as opposed to my employer's source code) is distributed outside the aegis of the source code control system. That is, it is mostly open source - I make it available to all and sundry. If someone reports a problem in a file, especially in a file they've modified, I think it is helpful to know where they started from, and that's represented by the original metadata in the source file.

Here, SCCS has an advantage over RCS: the expanded forms of the SCCS keywords are indistinguishable from

regular text, whereas the RCS keywords continue to look like keywords, so if the other person has imported the material into their own RCS repository, their metadata replaces my metadata, a problem that does not happen with SCCS in the same way (the other person has to do work to overwrite the metadata).

Consequently, even if someone takes a chunk of my source code and modifies it, there are usually labels enough in it to identify where it came from, rather than leaving me to speculate about which version it is based on. And that, in turn, makes it easier to see what parts of the problem are of my making, and what parts are of their making.

Now, in practice, the way open source works, people don't migrate code around as much as you might think. They tend to stick with the released version fairly closely, simply because deviating is too expensive when the next official release is made.

I'm not sure how you are supposed to determine the base version of a piece of source code that originated from your work and has been revised since then. Finding the correct version, though, seems key to doing that, and if there are fingerprints in the code, then it can be easier.

So, that's a moderate summary of why I like to embed the version information in the source files. It is in large part historical - SCCS and RCS both did it, and I liked the fact that they did. It may be ancient relic, something to be bidden farewell in the era of DVCS. But I'm not yet wholly convinced by that. However, it might take still more of an

essay to explain the ins and outs of my release management mechanism to see why I do things as I do.

One aspect of the reasoning is that key files, such as 'stderr.c' and 'stderr.h', are used by essentially all my programs. When I release a program that uses it, I simply ensure I have the most recent version - unless there's been an interface change that requires a back-version. I haven't had that problem for a while now (I did a systematic renaming in 2003; that caused some transitional headaches, but Perl scripts allowed me to implement the renaming pretty easily). I don't know how many programs use that code - somewhere between 100 and 200 would be a fair guess. This year's set of changes (the version 9.x series) are still somewhat speculative; I haven't finally decided whether to keep them. They are also internal to the implementation and do not affect the external interface, so I don't have to make up my mind just yet. I'm not sure how to handle that using git. I don't want to build the library code into a library that must be installed before you can build my software - that's too onerous for my clients. So, each program will continue to be distributed with a copy of the library code (a different sort of onerous), but only the library code that the program needs, not the whole library. And I pick and choose for each program which library functions are used. So, I would not be exporting a whole sub-tree; indeed, the commit that covered the last changes in the library code is typically completely unrelated to the commit that covered the last changes in the program. I'm not even sure whether git should use one repository for the library and another for the programs that use it, or a common

larger repository. And I won't be migrating to git until I do understand this.

OK - enough wittering. What I have works for me; it isn't necessarily for everyone. It does not make extraordinary demands on the VCS - but it does require version metadata embedded in the files, and CC and Git and (I think) SVN have issues with that. It probably means I'm the one with problems - hangups for the lost past. But I value what the past has to offer. (I can get away with it because most of my code is not branched. I'm not sure how much difference branching would make.)

Share

edited Jun 20, 2020 at 9:12

Community Bot

1 ● 1

answered Mar 14, 2009 at 5:07

Jonathan Leffler

752k ● 145 ● 946 ● 1.3k

"it does not embed version numbers into the source files" ??? We always considered that practice as evil (mixing meta-data information into data), introducing "merge hell". See also cmcrossroads.com/component/option,com_fireboard/func,view/… – VonC Mar 15, 2009 at 11:33

(Big-big thank you for all the edit in my post, by the way. The depth of your knowledge and experience shown in this edits/corrections is impressive) – VonC Mar 15, 2009 at 11:34

Of course, you can use a trigger for RCS keyword substitution (ibm.com/developerworks/rational/library/4311.html#N1046A) provided you use an appropriate merge manager

(ibm.com/developerworks/rational/library/05/1213_diebolt/…)
– VonC Mar 15, 2009 at 11:39

Whoa... I gonna read all this slowly, but in the meantime... +1!
– VonC Mar 15, 2009 at 18:00

Basically, I do not believe in metadata being fetched outside the the VCS, except for a big README (an its associated checksum), with a unique id in it labelling the all set of file. That way, the files can be modified over and over, I know where they come from. – VonC Mar 15, 2009 at 18:11

4

votes

I worked with clearcase for the better part of 6 years and generally found it tolerable. It does have a certain learning curve but once you get used to the quirks you can pretty much work smoothly with it. A very competent CC admin that knows what he's doing is essential for anything but trivial setups. Unless you have one, people are going to run into problems and soon enough there will be talk about the "ClearCase" problem. Then management will have to intervene by switching to something else causing only waste of time for everyone involved. CC is not a bad product, It's just sometimes poorly understood.

Here are few concepts I found important, some of these are not entierly CC only oriented -

- A check-out is unlike the regular CVS-like notion of a check-out. When you check out you lock the file until you check-in it in.
- There is no problem with moving files. infact this works flawlessly.

- Version trees are essential to understanding what has been happening to a file. They can get quite messy for active files but when you get used to watching them it becomes a very useful tool and One that is very lacking in other source control tools such as SVN (to some extent).
- Do not use dynamic views under any circumstances. its not worth it.
- before making a new branch, stream or project, advise with your admin to make sure that what you create is really what will serve you best. When starting a new code-base, make sure you get the streams and projects layout right from the start by planning ahead. changing it later is a real head-ache if even possible.
- Fine tune the privileges of users and set up triggers for common events to prevent common mistakes or enforce policies. The server is very configurable and most for problems you encounter there is probably a reasonable solution.
- educate the developers on anything from basic concepts to advance operations. A power-user that can find what the problem is using cleartool lowers the load on the admin.
- Don't leave dangling streams and views. When a developer leaves the project have someone to remove all the views he had on his machine and delete all his private streams. Not keeping your server clean will result in... it being dirty and over time, slow. When you do a "find all checkouts" on all streams and views you

should not see files that are checked-out by people who no longer exist.

- Mandate an "always rebase before deliver" policy for child branches to avoid people "breaking the integration stream" when delivering code that conflicts with recent changes.
- Continuous integration - don't let the integration stream stagnate while each developer or team work on their own branch. Mandate once every X time everyone has to atleast rebase to the most recent integration baseline if not to deliver their stable changes. This is indeed very difficult to do, especially with large projects but the other alternative is "integration hell" where at the end of the month no one does anything for 3 days while some poor sod tries to make all the changes fit together

Share

answered Mar 14, 2009 at 0:35

shoosh

78.8k ● 57 ● 213 ● 331

1 - I seriously argue with the comment about Dynamic views. How else would you get derived objects? – Spedge Apr 14, 2009 at 18:31

You will not. anything you need to derive should be able to be built locally. – shoosh Apr 14, 2009 at 22:27

how about cleartool commands when u need to automate and be in the view context. I am talking about clearttol, .bat and ANT scripts where a lot of customization/automation is happening and u need to be in the view context by STARTVIEW, or can't wait to update view or ... lot of other

scenarios. IBM might have made CC complex,but almost everything has a purpose and so has dynamic views
– Pulak Agrawal Nov 8, 2011 at 6:18

1 -1: Requiring a dedicated CC admin is a negative. – sashang Dec 22, 2011 at 0:42

---

4 votes

How to use git on top of ClearCase!

- Git Backstage, Under the Covers and on the Down Low
- A Clearcase for Git, git-cc bridge

Share

edited Dec 17, 2019 at 17:52

Jean-François Fabre ♦
140k ● 24 ● 177 ● 241

answered Mar 14, 2009 at 4:51

Matt Curtis
23.6k ● 8 ● 62 ● 63

---

1 vote

I've worked on a number of medium to large projects successfully using both Clearcase and SVN. Both are great tools but the team using them need documented processes. Create a process that describes how you will use the version control system.

1) find or create a best practices document for your Version Control System. Here's one for subversion, adapt it to your Clearcase process. All developers must adhere to the same game plan.

Basically decide if you are going to 'always branch' or 'never branch'.

Never Branch Scheme:

- The never branch scheme is what SourceSafe uses where files are locked during checkout and become available during checkin. This scheme and is okay for small (1 or 2 developers) team projects.

Always Branch Scheme:

- The always branch scheme means developers create branches for each bugfix or feature add. This scheme is needed for larger projects, projects that have a lead (buildmeister) who manages what changes get allowed into /main/LATEST in Clearcase or /trunk in SVN.
- The always branch scheme means you can checkin often w/o fear of breaking the build. Your only opportunity to break the build is only after your bugfix or feature is complete and you merge it to /main/LATEST.

'Branch when needed' is a compromise and may work best for many projects.

2) With Clearcase (and Subversion) you must learn to merge -- merging is your friend. Learn to use the merging capabilities of Clearcase or use a tool like Beyond Compare or emacs-diff. If your project is well modularized (many small decoupled files), you will benefit with fewer (or no) conflicts during merging.

3) Enjoy.

Share

answered Mar 14, 2009 at 2:27

thompsongunner
1,160 ● 1 ● 6 ● 6

1

vote

If you are using ClearCase, make sure you use the UCM that comes with it and Composite Components.

It makes all of your branching/merging effortless. I am talking about major re-org branches that run for 6 months involving tens of thousands of changes include directory renaming, file renaming, etc that automatically resolve 99.9% of the deltas.

Also, we only use SnapShot views, not dynamic views. Our snapshot views load faster than you can drag and drop (Windows) the same source tree from a network drive.

The only gripe I have about UCM is that history cannot span components. If you split up a component into multiple new components, each new component starts at /main/0.

Share

answered Jan 5, 2011 at 20:05

Englebart
11 ● 1

1 I often make deliver/rebase using a *dynamic* view, because the merges there are much faster and not blocked because of a snapshot view not *fully* updated. *Then* I update my snapshot view with the result of said merge and go on working. – VonC Jan 5, 2011 at 20:19