Revision Control Methodologies

Adam Pasztory
12/19/07

Revision control is one of the primary components of the change management process for system administrators.(1) Any project that has more than a couple of team members involved should have a revision control strategy in place. This strategy may include commercial or open source software tools such as Vault or SubVersion. The benefits of revision control software tools are straightforward: *they make it possible to revert to any version of a file that has been committed to a database (repository) of files*. These tools serve as a de facto backup system for a project, reducing the risk of information loss due to modification or deletion of files. In addition, the tools can compile a history of every file in a project, which serves as an audit trail that can be used to identify how and when problems were introduced. Finally, they mitigate the risk of change by allowing any user to revert to a “snapshot” of a project at any point.

Given such compelling selling points, it is curious that the use of revision control software often comes as an afterthought for development teams. In my personal experience, I have been surprised on a number of occasions where they could have provided an obvious benefit, but were not used. For example, at SF State, through the entire course of the Computer Science Major program, these tools were barely mentioned. And in a class like Operating Systems, where there are literally hundreds of files involved in a semester-long project, a revision control tool would have been invaluable. I have seen the same thing happen in industry. Sometimes companies simply do not do the appropriate planning to implement a standardized revision control solution.

The goal of this paper is to elucidate the methodology behind these software tools, and to show that at their core they perform very simple tasks. In addition, I will survey some of the most popular options that are available. It is my hope that a better understanding of the software at a high level can lead to more familiarity and comfort when working with it on a day-to-day basis.

Fundamentally, revision control software merely automates actions a person would perform manually when faced with the problem of tracking multiple versions. For example, if I had to track a file called “file.txt”, I might begin with a simple strategy of appending a number to the end of each filename: e.g.: file_1.doc, file_2.doc, file_3.doc, etc. A slightly more sophisticated solution might be to append dates rather than numbers (e.g. “file_29_march_2001”), thus providing a critical piece of identifying information within the filename. I could even use simple tool such as WinDiff to help track differences between versions and merge them. Obviously, a normal human can only maintain such a system with a limited number of files that do not change often. But a computer program does not face such limitations. As it turns out, the simplistic system I’ve just described is a baseline revision control methodology (though it is not commonly used)(2) where each new version of a file is stored independently, in its entirety. In this paper, I will refer to this methodology as **Complete Versions**.

Clearly, Complete Versions is impractical in terms of storage space, let us consider more space-efficient solutions... What if all changes were kept in a single master file? One could use meta tags within the file to indicate which revision a particular set of characters belonged to. This technique is called **The Weave** or **Merged Deltas** (the deltas being the changes). If you’ve ever look at a Microsoft Word Document in “Track Changes” mode (see figure 1), you’ve seen what such a master file might look like (though in Word, the changes are indicated by colors rather than meta tags). As with the first methodology, the Weave method becomes impossible for a human to manage once there are more than a handful of revisions. Again, this is where the power of computers to automate repetitive tasks becomes critical. Using Merged Deltas, a correctly written accessor program can output any revision from the master file in a constant time that is relative to the size of the file; the number of deltas has almost no impact on how quickly the file can be processed.(3)

The final, and most widely-used methodology is called **Separate Deltas**.(4) As with the other methods, it is not difficult to devise the algorithm for it simply by considering how one might solve an instance of the problem manually. Let us return to the concept of having one file that corresponds to each revision, as we had with the Complete Versions method. The initial version of a file will be committed to the file repository. However, each subsequent version of the file will contain only the deltas from the previous file. For example, say I checked a my term paper as an initial revision. Then, while proofreading, I notice that I'd forgotten to include my last name on the title page. Assuming there were no other changes in the document, my repository of revisions would now contain 2 files: the initial version, and second version that only contained my last name plus the meta data to help the accessor program figure out where the new text belonged. To retrieve version 2 of the file, the revision control software would have to reconstruct it by adding the deltas from the second file to the original version.

Tracking revisions becomes more complicated when more than one user is creating and modifying files in a project. It becomes crucial to restrict individuals from accessing certain files at certain times; this mechanism is called **File Protection**.(5)   You have probably seen File Protection in action if you've ever worked on a shared document that lives on server. When another user is accessing a document, you may not be allowed to modify or make changes to it; the file is **locked** to you (see figure 2).

There are two File Protection methods typically employed by revision control software: **Strict Locking** and **Optimistic Locking**. Under Strict Locking, only one user can modify a file at a time, as in the shared document scenario described above. In practice, Strict Locking can be cumbersome and inefficient. For example, Strict Locking can cause headaches for the administrator if a user forgets to unlock file before going on vacation. Though merges may be possible, in some cases, Strict Locking will prevent multiple users from editing different parts of the same file simultaneously.(6)

Optimistic Locking, on the other hand, allows users to work in parallel. Each user has their own “working copy” of the project, in which they make changes. When they commit files, their changes are merged with changes committed by other users. If the revision control software is unable to merge the changes, the user receives an alert. At this point, the user must reconcile the files manually. Doing so may even require consultation with other users who have also changed the file recently. While Optimistic Locking may sound chaotic, in practice conflicts are rare.(7)

Other aspects of revision control that come into play when multiple users are modifying the same files are **Identification** and **Documentation**.(8) Identification refers to the application of version numbers to files, folders, and entire releases. Revision control software makes Identification easier by automatically generating a version number for each revision. Most software also supports comments that can serve as Documentation. Every time a user commits a file, they are prompted to describe their change in plain language. These comments are stored in the database, and can be critical when other users are reviewing changes. Regardless of how sophisticated revision control software may be, human communication will always be a key part in the system. A solid revision control process depends on a logical organization of files, clearly-written comments, and pro-active conflict resolution.

There are now a multitude of revision control tools available. I have chosen to spotlight four that are illustrative of revision control methodologies. Aside from the first, which is obsolete, the others are all open source, and are readily available.

Source Code Control System (SCCS) - This program was created by Bell Labs in 1972. It is striking that its feature set was essentially the same as what is provided by modern tools: storage, protection, identification and documentation.(9) SCCS uses the Weave methodology to track revisions and Strict Locking is employed to prevent conflicts (it keeps an internal list of locked resources along with the programmer's name). Password protection and messaging is also implemented. In 1975, one of the developers, Marc Rothkind, referred to SCCS as "radical" method of source control.(10) An open source version (CSSC) is still active on SourceForge.

Revision Control System (RCS) - RCS is a revision control system that was developed for the UNIX operating system. You will find it pre-installed on most UNIX-based systems today, including OSX Macs. It uses the Separate Deltas and Strict Locking method. RCS is best for tracking individual files, rather than large-scale projects. Since it's ready to go out of the box, it may be appropriate if you don't want to hassle with installing and configuring one of the other tools.

Concurrent Versioning System (CVS) - CVS uses the Separate Deltas and Optimistic Locking methods. CVS provides mechanisms for **branching** a codebase, for example to support a Release and Development branch of a project.(11) Branches can be given tags -- identifiers that make it easier to track the points where the codebase was forked.(12)

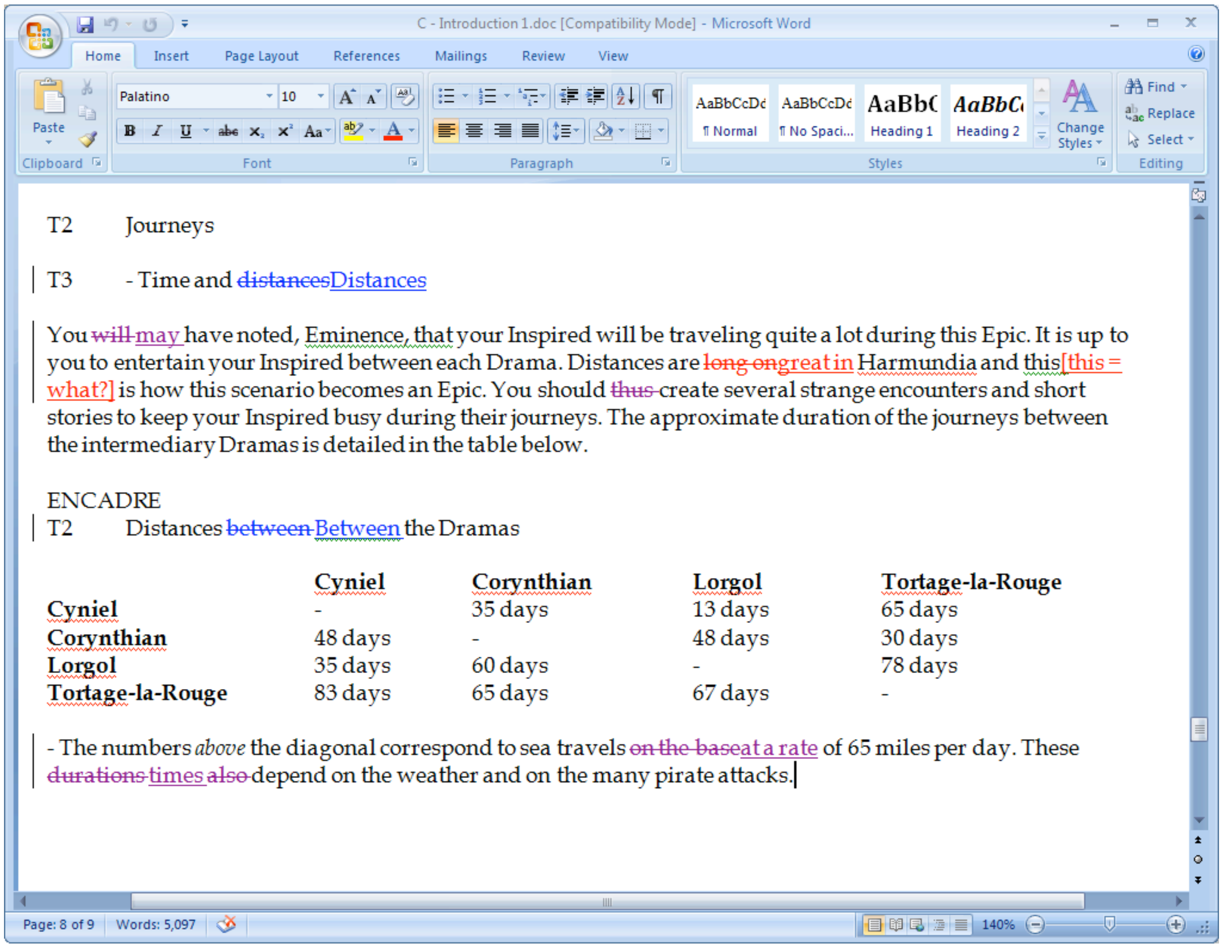
SubVersion (SVN) - SVN is the spiritual successor to CVS, though it is built on a new codebase. Like CVS, it uses the Separate Deltas and Optimistic Locking methods. SVN is notable because it tracks version information not just for files, but for the entire directory structure of the project. SVN supports operations such as Copy and Rename that are not supported in CVS. (In CVS, to rename a file, you must first delete it from the repository, and then add it back in under a new name.) As a result, the version history aggregated by SVN is closer to the "true" history of the project. SVN adds a mechanism called **atomic commits**, which helps maintain the integrity of the repository by disallowing incomplete modifications. (13) For example, if your computer crashes in the midst of uploading files to the repository, the entire transaction is cancelled.

Revision control solutions are now easier to use and more readily available than ever before. In addition to the tools discussed above, many programs, such as Microsoft Office, Adobe DreamWeaver and Eclipse provide features that help manage different versions of files. Online services like CVSDude.com even offer remote hosting of SVN and CVS repositories at affordable prices (their basic plan is free), allow configuration via a simple browser-based UI, and they provide deluxe features like web-based repository browsing.

It is the responsibility of developers and system administrators to evaluate and implement some sort of revision control process. In addition, many other types of users could benefit. In my testing, I've found that it is incredibly easy to compare and merge revisions of almost any kind of Microsoft Office document using Tortoise SVN, a powerful SubVersion client for Windows. At this point it is primarily a lack of understanding of revision control tools that prevents more widespread adoption. Before researching this paper, I was concerned about using SVN for a lot of my documents because I was afraid it would eat up my hard disk space. But now that I understand the efficiency of SVN's Separate Deltas algorithm, my fears have been assuaged.

It amazes me that revision control programs have been available since 1972, yet so few people use them. Perhaps there is even a business opportunity for Google or some other intrepid company, to introduce a revision control solution that is tailored to everyday computer users rather than developers.

Figure 1: Track Changes Mode in a Microsoft Word Document



T2 Journeys

T3 - Time and ~~distances~~Distances

You ~~will~~ may have noted, Eminence, that your Inspired will be traveling quite a lot during this Epic. It is up to you to entertain your Inspired between each Drama. Distances are ~~long on~~great in Harmundia and this[this = what?] is how this scenario becomes an Epic. You should ~~thus~~ create several strange encounters and short stories to keep your Inspired busy during their journeys. The approximate duration of the journeys between the intermediary Dramas is detailed in the table below.

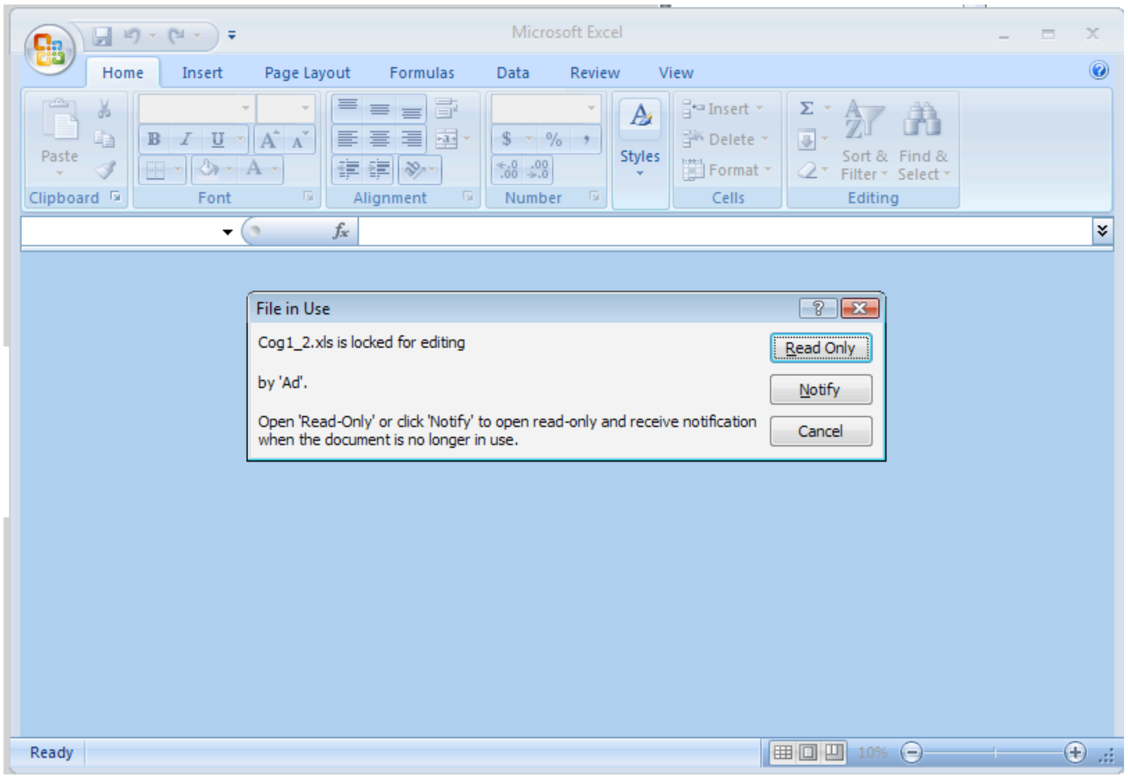
ENCADRE

T2 Distances ~~between~~ Between the Dramas

	Cyniel	**Corynthian**	**Lorgol**	**Tortage-la-Rouge**
Cyniel	-	35 days	13 days	65 days
Corynthian	48 days	-	48 days	30 days
Lorgol	35 days	60 days	-	78 days
Tortage-la-Rouge	83 days	65 days	67 days	-

- The numbers *above* the diagonal correspond to sea travels ~~on the base~~at a rate of 65 miles per day. These ~~durations~~ times ~~also~~ depend on the weather and on the many pirate attacks.

Figure 2: Strict Locking in Microsoft Office



File in Use

Cog1_2.xls is locked for editing

by 'Ad'.

Open 'Read-Only' or click 'Notify' to open read-only and receive notification when the document is no longer in use.

Read Only | Notify | Cancel

Footnotes

1. Limoncelli, Thomas A., Christina J. Hogan, and Strata R. Chalup. The Practice of System and Network Administration. 2nd ed. Upper Saddle River: Addison Wesley, 2007: 416.
2. Hudson, Greg. "Notes on Keeping Version Histories of Files." Web.Mit.Edu/Ghudson/. 24 June 2006. M.I.T. 16 Dec. 2007 <http://web.mit.edu/ghudson/thoughts/file-versioning>.
3. Tichy, Walter F. "Design, Implementation, and Evaluation of a Revision Control System." International Conference on Software Engineering Archive (1982): 58-67.
4. Hudson
5. Rochkind, Mark J. "The Source Code Control System." IEEE Transactions on Software Engineering. 1975: 364.
6. Collins-Sussman, Ben, Brian W. Fitzpatrick, and C. Michael Pilato. Version Control with Subversion. 1st ed. Sebastapol: O'Reilly Media, Inc., 2004: 11.
7. Thomas, David, and Andrew Hunt. Pragmatic Version Cotrol Using CVS. 1st ed. Raleigh: The Pragmatic Programmers, LLC, 2003: 22.
8. Rochkind 364
9. Rochkind 364.
10. Rochkind 368
11. Thomas, et al. 16
12. Thomas, et al. 93
13. Collins-Sussman, et al. 3

Works Cited

Collins-Sussman, Ben, Brian W. Fitzpatrick, and C. Michael Pilato. Version Control with Subversion. 1st ed. Sebastapol: O'Reilly Media, Inc., 2004.

Collins-Sussman, Ben. "The Subversion Project: Building a Better CVS." Linux Journal 94 (2002): 3.

Hudson, Greg. "Notes on Keeping Version Histories of Files." Web.Mit.Edu/Ghudson/. 24 June 2006. M.I.T. 16 Dec. 2007 <http://web.mit.edu/ghudson/thoughts/file-versioning>.

Limoncelli, Thomas A., Christina J. Hogan, and Strata R. Chalup. The Practice of System and Network Administration. 2nd ed. Upper Saddle River: Addison Wesley, 2007.

Magnusson, Boris, Ulf Asklund, and Sten Minor. "Fine-Grained Revision Control for Colalborative Software Development." Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering (1993): 33-41.

Rochkind, Mark J. "The Source Code Control System." IEEE Transactions on Software Engineering. 1975. 364-370.

Thomas, David, and Andrew Hunt. Pragmatic Version Cotrol Using CVS. 1st ed. Raleigh: The Pragmatic Programmers, LLC, 2003.

Tichy, Walter F. "Design, Implementation, and Evaluation of a Revision Control System." International Conference on Software Engineering Archive (1982): 58-67.