The PracTEX Journal, 2007, No. 3
Article revision 2007/08/17

Subversion and TextMate: Making collaboration easier for LaTeX users*

Charilaos Skiadas, Thomas Kjosmoen, and Mark Eli Kalderon

Abstract This article focuses on the Subversion version control system, and in particular its integration into the TextMate text editor. TextMate is a text editor, for the Mac OS X operating system, that has excellent support for working with LaTeX documents and projects, and its seamless Subversion integration makes it very easy to add version control to your workflow.

Version control systems have been used by computer programmers for many years for work on projects involving multiple authors. They have been invaluable in keeping track of the contributions of each party to the project, the changes that took place and when they happened, and so on. Subversion is one of the most popular version control systems today, and it is integrated into many text editors and online collaboration sites.

1 Introduction

We often have to work on LaTeX documents with other people. There are a number of problems with such a collaboration:

- We need to exchange files with our collaborators and keep track of the date of each file, to avoid working on old files.
- We need to communicate with our collaborators what changes we have made to the current draft.
- We would like to keep a track record of all changes made to the document, and which person made each change, for future reference.
- We need some system to keep backups of the files, to avoid data loss.
- We may need to work on the same files on multiple computers, for instance at home and at work.

*Thanks to Francisco Reinaldo for his help and encouragement.

Most of us deal with these problems in awkward ways. E-mailing is probably the most usual way of exchanging versions of our files. Since each file has an associated *modification date*, we also have an elementary way of keeping track of when a file was last changed. We also often tend to name the files according to the date on which they were last modified. To communicate the changes made, we might write comments on the margins via `\marginpar`, or just comments in the LaTeX text, or simply a descriptive e-mail.

For many years programmers have battled with these problems, and a number of excellent solutions have emerged; the so-called *Revision Control Systems* or *Version Control Systems*. *Subversion*[1] is a popular and freely available version control system. The article by Mark Eli Kalderon in the current issue [2] offers a comprehensive introduction to the Subversion system and its benefits when working on LaTeX projects. TextMate as a LaTeX editor has also been discussed by the first two authors in another article in the current issue [3]. In this article, however, we will focus on TextMate's Subversion capabilities, and also discuss more advanced Subversion concepts such as branches and tags, and the process of merging.

In Section 2 we review the basic concepts of Subversion, discuss the problem of simultaneous commits and how Subversion addresses this, introduce the extremely useful concepts of branches and tags, and discuss the `svn merge` process. In section 3 we talk about TextMate's Subversion menu and the various commands in it. Section 4 discusses another very useful feature of TextMate, the TODO bundle, which allows you to keep track of tasks in a simple and convenient way.

2 Revision Control and Subversion

Version control systems are divided in two categories, depending on their decision of where to keep all the data. *Centralized* version control systems follow a client-server model, where the history of the project is stored in some central server, known as the *repository*. Changes made by an author must be committed to this central repository, and most of the operations asking for information about the history of the project need to access this server. Subversion is the primary example of free centralized version control systems, and it has to a large extent

1. http://subversion.tigris.org

supplanted the more traditional CVS[2]. This paper will focus on a discussion of Subversion, though most of the general concepts apply to other centralized version control systems as well.

Distributed version control systems on the other hand are based on the idea that each author has a local copy of the complete history of the project. The author can then go on to work on the project, and at some point his changes can be synchronized with the changes from the other authors. One of the main advantages of distributed version control systems is that they allow a lot more offline operations. Examples of distributed version control systems are Git[3], Mercurial[4] and SVK[5].[6]

2.1 The Repository and Working Copies

The central location where data is stored is known as the *Repository*. It is usually in a remote server[7] or in your local machine[8]; the data is stored in a database (*e.g.* Berkeley DB in Subversion) or as files (*e.g.* FSFS in Subversion), and could potentially be encrypted. This can offer a safe backup location, as servers are typically daily backed up. There are several good free services on the Internet offering access to servers that have Subversion set up.[9]

Let us say that the central repository has been set up and has the recommended repository layout (with trunks, branches, and tags—more on these in section 2.3). In order to protect the repository, users are not given direct access to it. Instead, they work with snapshots of the repository, known as *Working Copies*.

2. http://www.nongnu.org/cvs
3. http://git.or.cz
4. http://www.selenic.com/mercurial
5. http://svk.bestpractical.com/view/HomePage
6. For more information about the philosophy of these distributed version control systems, please see http://speirs.org/2007/07/19/a-subversion-user-looks-at-git and http://www.selenic.com/mercurial/wiki/index.cgi/UnderstandingMercurial. An extended list of version control systems can be found here: http://en.wikipedia.org/wiki/List_of_revision_control_software
7. http://www.assembla.com/space/2007-3PTJ_charilaos of this paper, or svn://tug.org/pracjourn/trunk of the PracTeX Journal.
8. file:///Users/Thomas/SVNrepository/Documents/trunk
9. Please see http://www.subversionary.org/hosting/hosting-services for a list of such services.

The working copy is a directory of files that the author gets from the repository for the first time in a process known as a *Check Out*. This is done *once*.

After the working copy has been checked out, the work process typically consists of the following steps:

- *Update* the working copy from the repository. This gets the newest version of the files from the repository, which includes the changes made by your coauthors.
- *Change* the working copy—either by editing individual files or by changing the directory structure by adding, deleting, or renaming files and directories.
- *Review* the changes made to the working copy.
- *Commit* the changes you made to the working copy. This sends the changes you made, usually along with a comment on the changes, into the repository, so that your coauthors can access them.
- If in the meantime another author has checked in their changes, then you have to update your copy and *merge* your changes with the other author's after *resolving* any conflicts (unless Subversion was able to resolve the conflicts itself).

2.2 The Problem with Simultaneous Commits

Collaboration is based on multiple people working on the same files, and this can cause some problems depending on the order in which these people read to and write from the files. Suppose, for example, that you started to work on a file from the repository, and made some changes to it. You then attempted to commit those changes to the repository. In the meantime, one of your coauthors made some changes to the same file, and committed his changes before yours. The version of the file that you were working on knew nothing of these changes, so when you commit your version of the file, with your changes in it, then his changes will be lost for ever.

There are more than one way to solve this problem. A method used by many systems is a "locking" mechanism. Using this mechanism, you *lock* the files you are going to change before making the changes. You then check the files out,

make the necessary changes, and subsequently commit the changes you made to the repository. After you are done, you unlock the file. The problem with this system is that nobody else can work on the file while it is locked. For a LaTeX document this might work reasonably well, but it is still a significant drawback.

Subversion uses a different method, which involves *merging* of files. The idea is quite simple. Authors check files out as usual, make the changes they want, and try to commit their changes. If, in the mean time, another author has made some changes, Subversion will check to see if those two sets of changes conflict with each other, namely whether both authors have made changes to the same line. If the changes do not conflict with each other, Subversion will *merge* the two sets of changes and update the file accordingly. Otherwise, it will inform the committing author of the problem and allow them to look at the conflicts and decide how best to fix them. Sections 2.4 and 3.4 discusses these issues more.

2.3 Revisions and Branches

Each time you commit something to the repository, the resulting state of the repository is called a *revision*. For instance the first commit made to the repository is revision 1, the second commit becomes revision 2 and so on. These revisions contain the entire history of your project. They each can contain a *commit message*, a useful and brief description of what was committed, or any other comments the author wants to make regarding the commit. The commit messages allow the other authors to get a quick summary of the various stages in the project. For this reason, it is in general advisable that each commit be "atomic", namely that it deals with only one change to the document. For instance, once you finish revising a particular section, you can commit those changes. This allows for more information to be stored in the revision.[10] Since revisions are "cheap", there is no reason not to have many of them. A classical motto is: "Commit early, commit often."

All these revisions allow the authors a lot of flexibility. For instance, you can ask Subversion to show you all the differences between two revisions: Which files were changed, and what changes were made to them. You can also ask it to show you the state of the repository at any revision you want.

10. As an example, this paper is in a Subversion repository, and has seen over 130 revisions in 4 days.

While revisions have a sequential structure, Subversion offers another way to proceed with the project, using what is known as *branches*. A branch, as its name indicates, is a new direction in the repository; very useful for doing major changes to files (Figure 1). A branch, then, is a directory of the repository representing

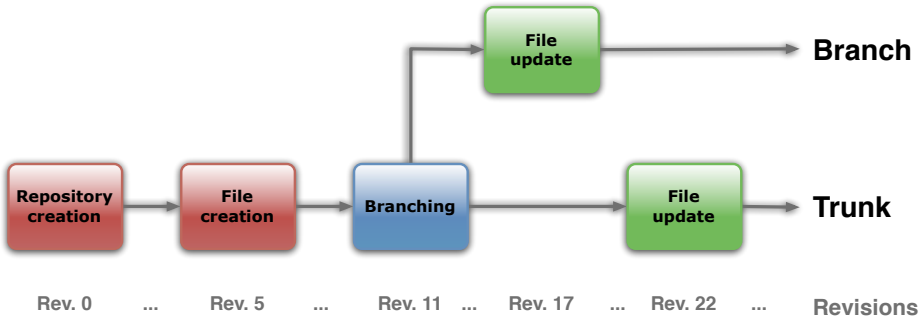

Figure 1: A branch is a new direction of development, moving parallel to the trunk.

a line of development of your project independent of its main or current line of development. In contrast, the main or current line of development resides in a distinct directory called the *trunk*. To create a branch of your document simply copy it into a branch directory and commence with its alternative line of development. If, at some point, you are satisfied that the branch supersedes the trunk as it stands, the changes to the branch can be merged into the trunk.

Suppose, for example, that you are working on some part of the document, on which you are planning to do major changes, which will take many steps and quite a lot of time to complete. At the same time though, your coauthors will need to work on some other parts of the same document, and your work really should be kept separate from their changes. The best solution then is to create a new branch of the repository, and make your changes in that branch, probably in many revisions. In the meantime, your coauthors make their changes into the trunk. Once you are finished with your changes, you can tell Subversion to *merge* the branch into the trunk (Figure 2).

However a branch doesn't have to end when its changes are merged into the trunk. It can continue to exist parallel to the main trunk, with merges happening

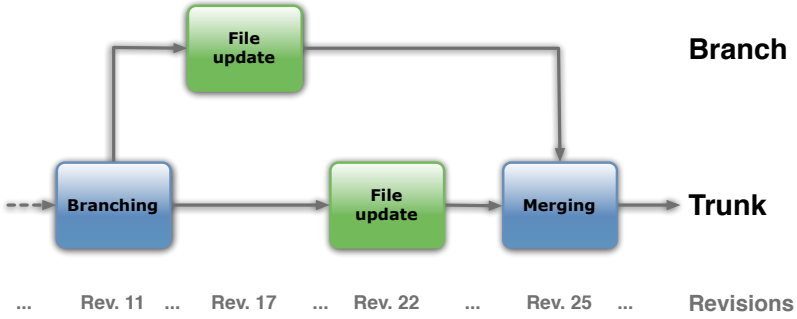

Figure 2: An author may create a temporary branch, to work on a file without interrupting the work of other authors on that file.

either from the trunk to the branch or the other way around (Figure 3). Imagine, for example, working on a textbook. Then the main trunk could correspond to the student's edition, and a branch could correspond to the teacher's edition.

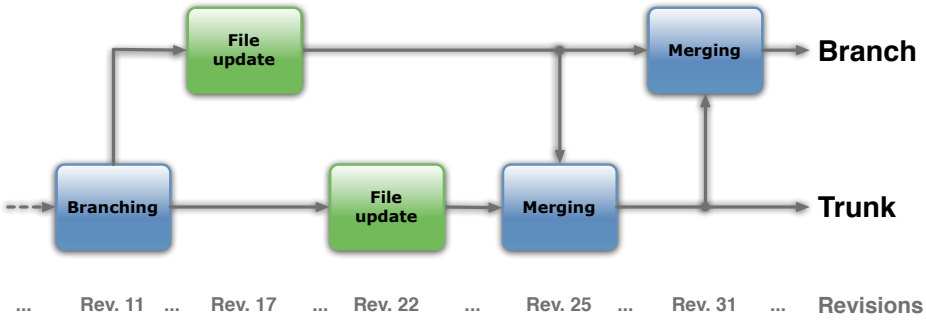

Figure 3: A branch can continue parallel to the trunk, and merging between the two can occur many times.

Another concept, possibly not of much use to LaTeX users but quite useful in, for instance, manuals or other business documents, is the concept of *tags*. A tag is simply a snapshot of the repository at a particular time, and it is meant to not be edited again. So it is just like a revision, except that it is *tagged* with a more meaningful name, and it is thought of as a "finished product". Tags can be used for instance for the various *versions* of a manual. If we were talking about a software, a tag would correspond to a stable release.

2.4 Svn Merge and Svnmerge

So far we have seen merging used to resolve conflicts and in branch management. In the case of conflicts, merging needs to be done by hand, since it requires the application of human judgement and consensus with your coauthors and neither human judgment nor consensus can be automated.[11]. But in the case of branch management, Subversion provides a useful tool, the `svn merge` command. Let's take a closer look at the `svn merge` command and its limitations. Unfortunately, for LaTeX users, these limitations are severe. Fortunately, these limitations can be overcome, in part, by the script `svnmerge.py` that can be integrated with TextMate's Subversion bundle. Moreover, these limitations of merge tracking are being addressed in the next major release of Subversion in a similar manner to `svnmerge.py`.

The `svn merge` command does two things, it compares the differences between two files or directories of files and applies these differences to the working copy. `svn merge` thus takes three arguments. The first two arguments can be repository urls or working copy paths (the locations of the two files or directories to be merged) and the third argument is a working copy path (the location where these differences are to be applied).

Let's consider a simple example to illustrate this. Suppose you are maintaining a list, say, of invitations, that you decide to branch. Perhaps you are considering inviting some more people, but you are unsure whether to invite them or not. Suppose your initial list of invitations, recorded in `trunk/list.txt`, is as follows:

```
Bill
Peter
Mary
```

And suppose your branched list, recorded in `branch/list.txt`, also includes Sally and John :

```
Bill
Peter
Mary
Sally
John
```

11. See the Subversion manual [1] for detailed instructions about resolving conflicts by hand

Suppose you decide that it is OK after all to invite Sally and John (they have been arguing with Bill, but they have made up). You now want to merge the changes you made in the branch into your trunk. This can be done as follows:

```
$ svn merge http://myrepository.com/svn/mylist/trunk/list.txt
http://myrepository.com/svn/mylist/branch/list.txt
mylist/trunk/list.txt
```

The file in your working copy, `mylist/trunk/list.txt`, now contains Sally and John as well. To propagate these changes to your repository, be sure to commit these changes.

Sounds easy. There are, however, limitations to the `svn merge` command that can create complications. The problem arises when you repeatedly merge changes. It can happen that you merge the same change twice over. Sometimes this is OK. If Subversion notices the file has the relevant change already it does nothing. But suppose the change has been made *and has been further modified*. We now have a conflict. Consider the list example again. Suppose John prefers to go by "Jonathan" and this change is made to `mylist/trunk/list.txt` and not to `mylist/branch/list.txt`. The branched list now has names of additional people that should definitely be invited. But it also has John, which svn does not identify as the same as Jonathan. Hence, running the above `svn merge` command again will produce a conflict. The problem arises because Subversion doesn't remember what changes have already been merged into the trunk and so attempts to merge these again.

Another limitation is the inability to exclude a previous revisions from being merged. Suppose you are writing a paper and have been invited to give a talk based on that material. It is natural to create a branch. It is also natural to eliminate some things from the talk like footnotes. You will do this with some commit, that corresponds, say, to revision 20. But suppose in writing the talk you hit upon things that should be in the paper. These are saved in some other revision, say, revision 21. You would now want to merge these changes into the paper, which is in the trunk. However, you cannot tell `svn merge` to merge only those changes, and to not merge the changes you made at revision 20. So with `svn merge` you would end up losing the footnotes in the original paper.

A final limitation is that the changes made when a merge occurs are attributed to the person that did the merge. So, for example, if there were more than one

person working on the branch, and one of those two ended up merging the branch to the trunk, then all the resulting changes are attributed to that person, even though some of them were made by someone else.

So to summarize, some of the main limitations of the `svn merge` command are:

- Subversion doesn't remember merged changes.
- Subversion can't exclude a change set from being merged.
- When a merge occurs, Subversion attributes all changes to the branch to the person merging.[12]

`svnmerge.py`[13] is a tool for automatic branch management that addresses these problems. It remembers which changes have already been merged and does the right thing by only merging the new changes. It lists changes available for merging so that some, but not all, of these can be merged. When merging, it includes the log of all the changes in the branch in the commit message so that changes are appropriately attributed.

To initialize the merge tracking support run the following command:

```
$ svnmerge.py init path/to/my/branch/
```

This only needs to be done *once* for each branch that you are managing. The command:

```
$ svnmerge.py avail
```

lists the revisions that are available for merging. And the command:

```
$ svnmerge.py merge
```

can be used to merge some, but not all, of the available revisions. One note of caution: Be sure that your commits are truly "atomic" (see section 2.3), since if a revision contains two changes, only one of which should be merged into the trunk, then this will need to be done by hand. After these changes have been merged, you can then commit them using `svn commit`.

12. For trenchant criticism of these and related limitations see Linus Torvalds' talk to Google about the advantages of Git and the distributed, as opposed to centralized, model of version control http://www.youtube.com/watch?v=4XpnKHJAok8.
13. http://www.orcaware.com/svn/wiki/Svnmerge.py

3 TextMate's Subversion menu

In this section we will discuss TextMate's Subversion menu, that allows you to do most Subversion-related tasks without ever having to leave TextMate. All Subversion commands in TextMate use the same shortcut, so one is typically presented with the options in Figure 4. The numbers next to the first ten commands mean that those commands can be accessed by simply pressing the corresponding number key. We will go in some detail through each command, but if you still have questions, the "Help" command in the menu will likely answer some of them. In particular it will provide details on the various environment variables you can set to customize how Subversion works.

The commands are divided in groups, depending on their functionality. The first group has to do with making and receiving changes in the repository. The second group has to do with reviewing information about the repository. A third group has to do with looking at the changes between revisions, and a fourth group consists of commands to deal with conflicts.

3.1 Working with Files

The first set of commands is all about working with files. The first command allows you to add files to the repository, and it essentially calls the `svn add` command for you. It will schedule for addition all the files that are selected in the project drawer. Similarly, the second command will schedule the selected files for removal from the repository. These actions will only take place at the next commit attempt, which we will discuss shortly.

The third option allows you to revert the file to its most recent repository state. What this means is that you will lose any local changes you've made to the file but had not had time to commit yet. Hence, TextMate warns you first before doing it (Figure 5).

The next command in the Subversion menu is used to update the working copy to the newest revision. It will only work on selected files, so you can update only part of the working copy if you prefer, or select the root folder and update the entire directory.

The "Commit" command is perhaps the most important command in the

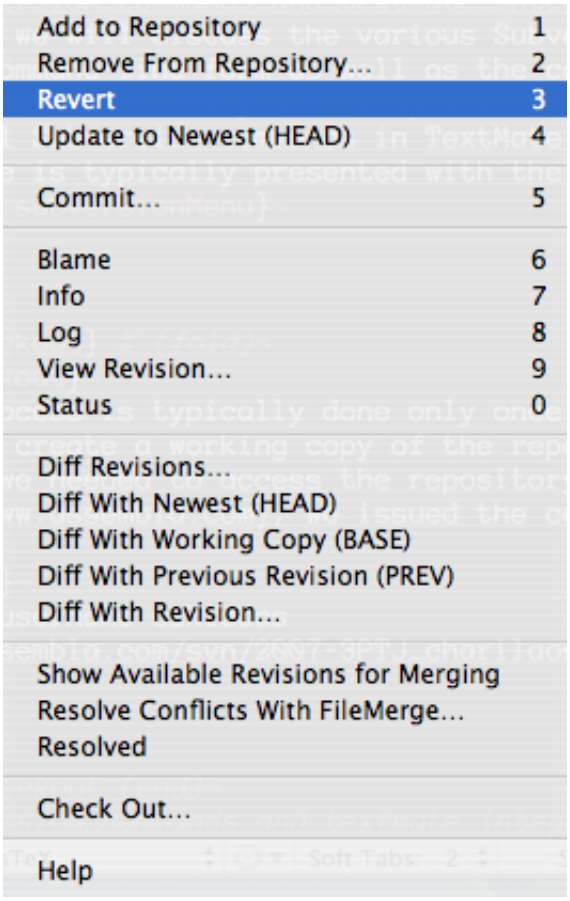

Figure 4: TextMate's Subversion menu offers quick access to all Subversion commands.

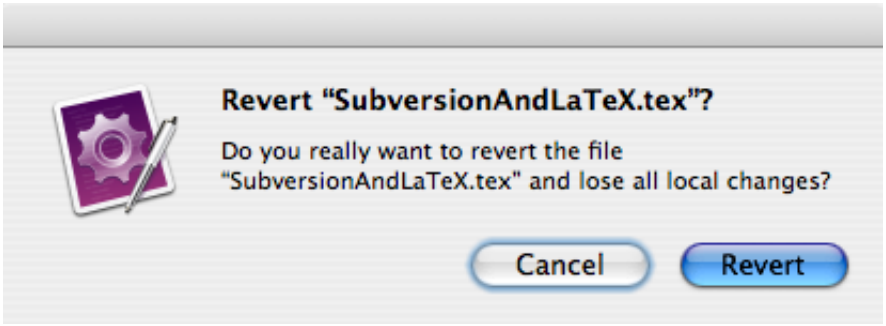

Figure 5: TextMate warns you before doing anything destructive!

menu. It is after all the command you would use to commit your changes to the repository, and thus move forward with the project. It has an elaborate menu, that allows you to review the changes you have made and decide what you want to commit, and what you don't want to commit yet. Figure 6 shows how this window looked like at some point in the progress of this paper.

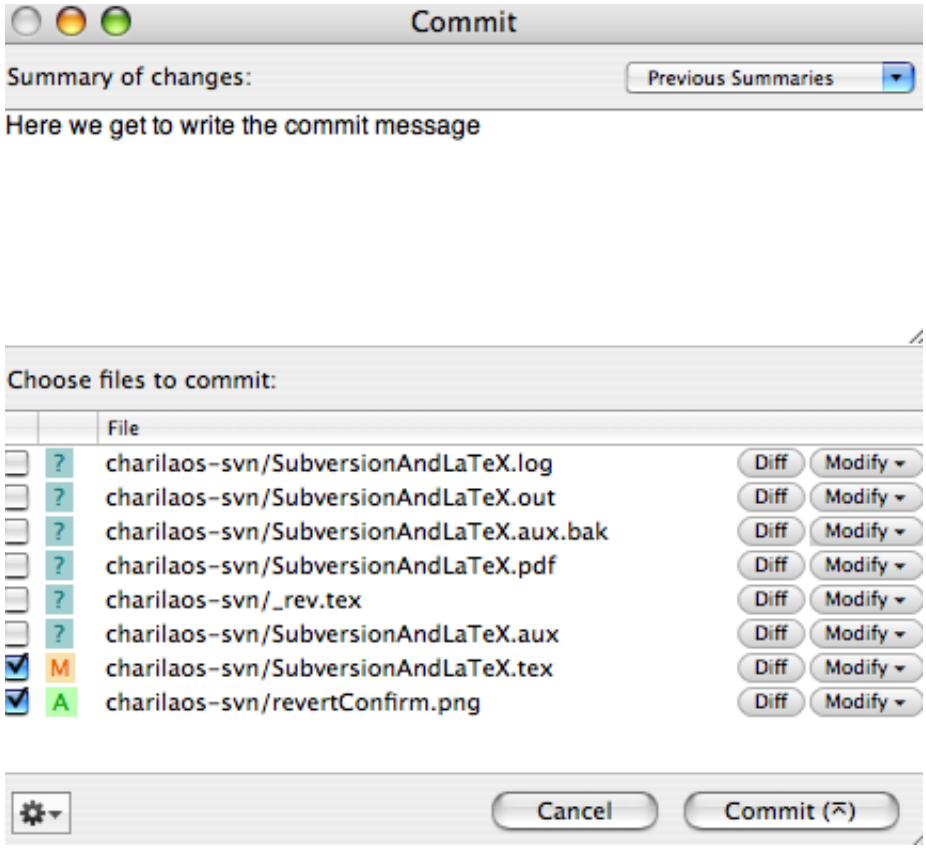

Figure 6: The Commit Window allows you to review your changes, and decide what to commit.

The top of the Commit Window contains an area where you write the commit message. The bottom part of the window is the most interesting part. Here you are provided with a list of all files related to your commit. Each file has a checkbox on the left, where you can decide whether you want this file to be committed or not. Next to the checkboxes, a colored character indicates the state

of the file. A question mark indicates that this file only exists in the working copy, and the repository knows nothing about it. In our case, these are files generated from the LaTeX file, hence we don't want them in the repository. The "M" next to the file `charilaos-svn/SubversionAndLaTeX.tex` indicates that this file has local changes, and the checked box indicates that we are about to commit our changes. Finally, the "A" next to the file `charilaos-svn/revertConfirm.png` indicates that this file is not yet in the repository, but we have scheduled for it to be added in the next commit.

Buttons on the right side of the window allow us to view changes, and to modify the state of the file. The "Diff" button is useful mainly for the modified files, and provides a screen with the differences between our copy of the file, and the copy in the repository that we are about to replace (Figure 7).[14] The "Modify" button provides you with options depending on the status of the file. For files unknown to the repository (those marked with a "?") it offers the option of adding them, for those that are modified the option to revert the changes, and so on.

3.2 Getting Information

The next set of commands in the Subversion menu deal with obtaining information from the repository, regarding the status of files, their history, etc. The first of these is the "Blame" command, which provides a view of the current document, with each single line marked according to when that line was last modified in the file, by whom and on which revision (Figure 8).

The next command is the "Info" command, which is just a wrapper for the `svn info` command. More useful information is obtained from the "Log" command, which shows a list of all the revisions where the current file was affected, the commit messages on each revision, and lots of other useful information (Figure 9).

The "View Revision..." command allows you to select one of the previous revisions of the current file, and have that version of the file open in a new window (Figure 10). Finally, the "Status" command shows a window with options

14. Diff files will be discussed more in section 3.3.

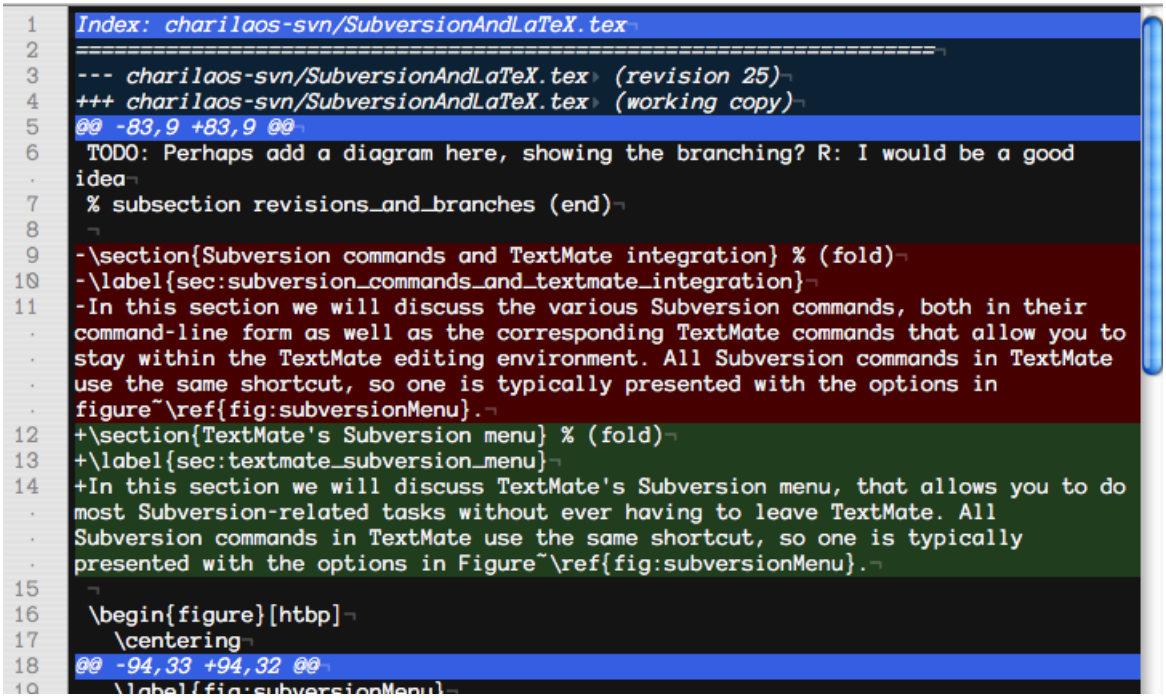

Figure 7: Subversion allows you to see the differences between the working copy of a file and the copy in the repository.

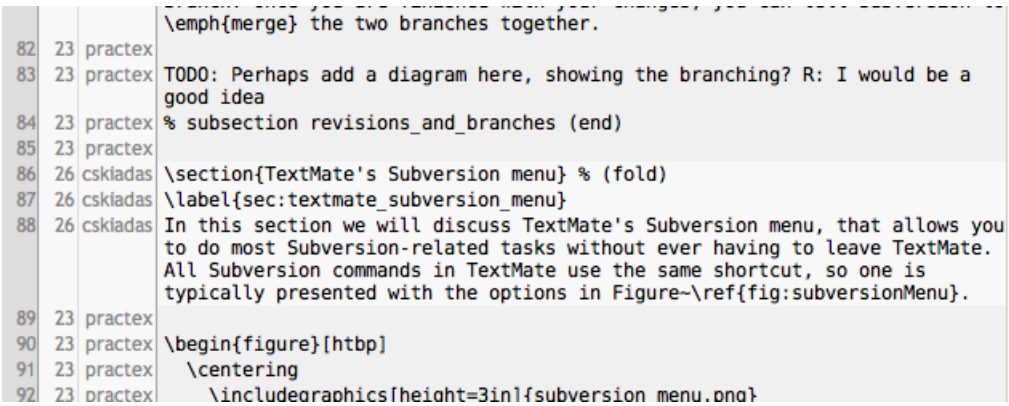

Figure 8: A portion of the Blame Window. Some lines were added by the author named `practex` in revision number 23, while others were added by author `cskiadas` in revision number 26. Hovering over a revision number makes the corresponding commit message for that revision appear.

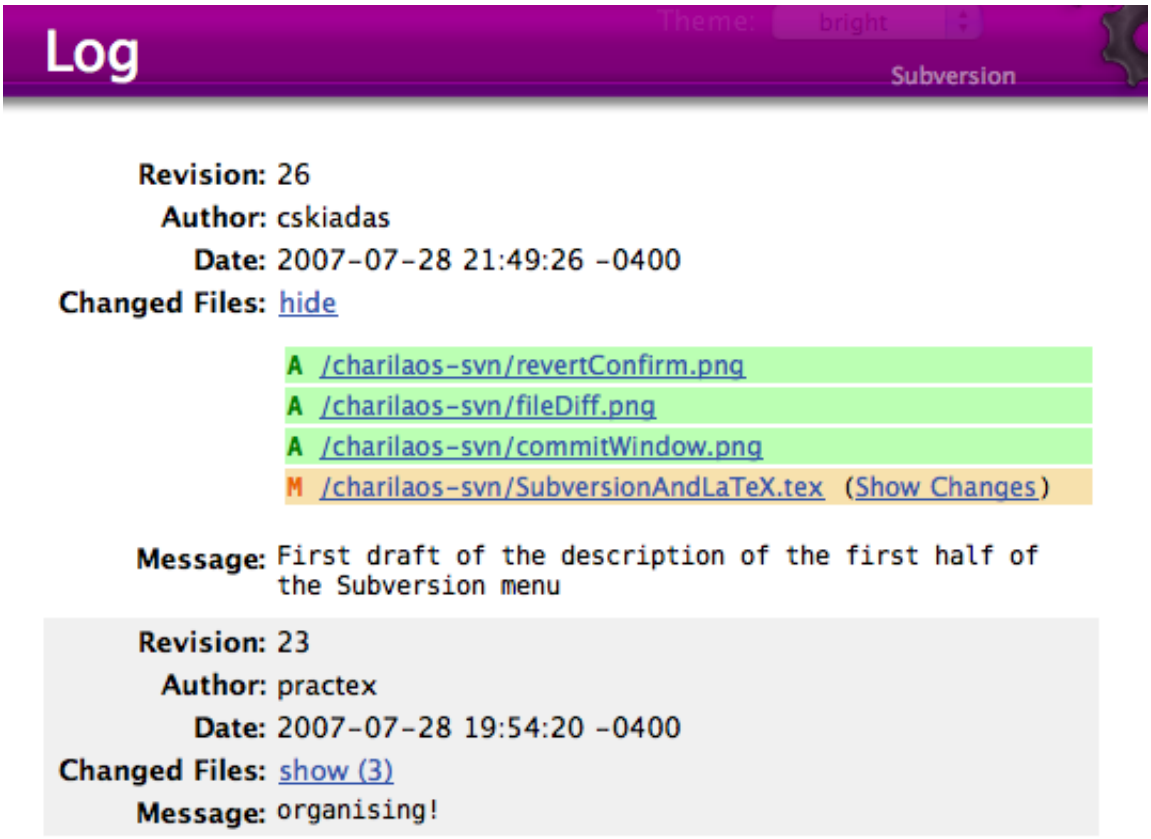

Figure 9: The Log Window gives us information on the history of the current file.

very similar to the ones shown in the Commit window. There are options for seeing the changes you would commit, options for adding files to the repository or removing files from it, and even an option to commit your changes.

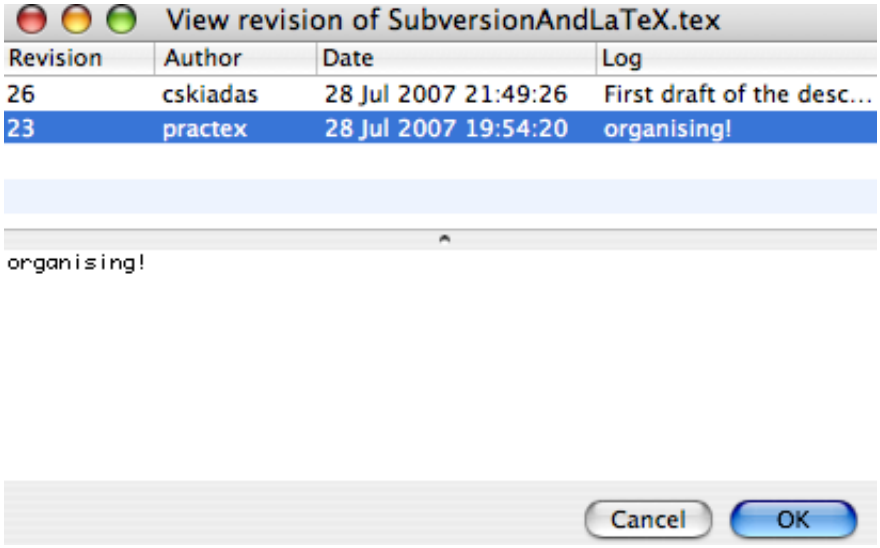

Figure 10: The View Revision window shows all previous revisions of a file, and allows you to open any one of those revisions in a new window.

3.3 Looking at Changes

The next set of commands in the Subversion menu are the five commands for producing various kinds of diff files. *diff* files are files that have a certain format, and their purpose is to show the differences between two other files. We already saw such a diff file in Figure 7. TextMate recognizes the particular format of diff files, and produces a colored output showing the differences between the two files. Unfortunately for LaTeX users, diff files look at *whole lines*, which means that if you compare two files that differ in a single character, then the diff file will indicate the entire line where that character is as the difference. In a LaTeX document, that line could happen to be a whole paragraph. Alternatives to this standard diff mechanism are discussed in [2], and TextMate can easily be made

to use those alternatives. In this section, we will talk about the diff commands assuming that the standard diff mechanism is used.[15]

All diff commands do essentially the same thing: They show you the differences in the selected files between two revisions. The commands differ as to whether and how those revisions are specified. They all produce a new diff file that contains the differences between the two revisions.

The "Diff Revisions..." command allows you to specify any two revisions from a simple and very informative list (Figure 11). The "Diff with Newest" command is perhaps the most often used of the diff commands. It shows the differences between your current version of file and the newest version of the file in the repository. This allows you to see the changes that you would make to the repository version if you were to commit the file right now.

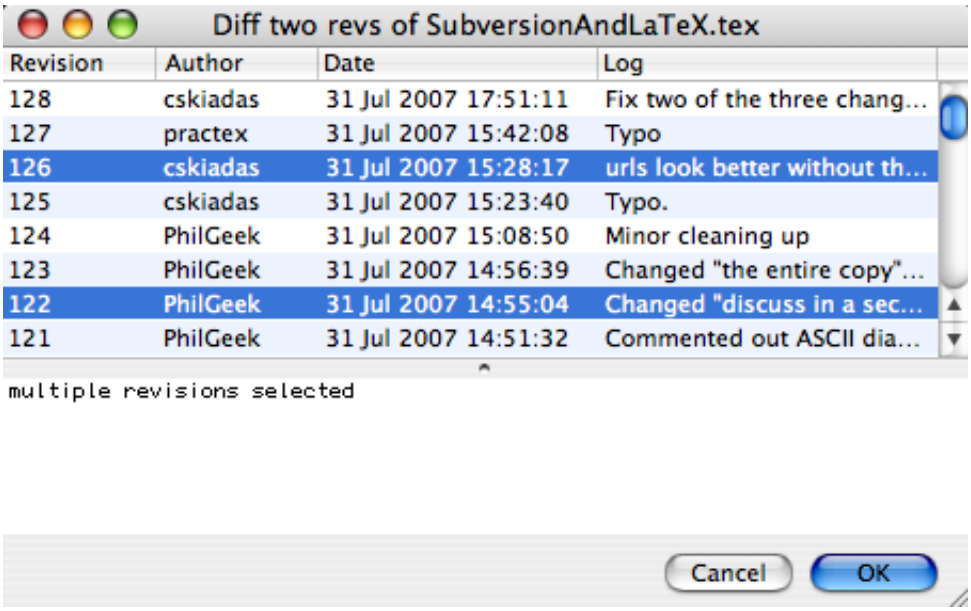

Figure 11: The Diff Revisions command allows you to select the two revisions whose differences you want to look at.

The next command is the "Diff with Working Copy" command. This shows you the differences between your version of the file and the version in the latest

15. What the commands do is essentially the same regardless of what diff mechanism is used.

revision that you have in your working copy (which might not be the latest revision in the repository if you haven't updated recently). This is particularly useful if you have to often work offline, since it doesn't need access to the repository. "Diff with Previous Revision" is helpful when you have just used `svn up` to update your working copy, and want to see the latest changes made to the file. It will show you a diff file between the two newest revisions. Finally, the "Diff with Revision..." command shows you the diff file between your current version of the file and the version in one particular revision.

The standard diff mechanism is useful, but it has its limitations. As we mentioned, one big limitation for LaTeX users is that it only displays *line* differences. The problem is that paragraphs are long lines and so the standard diff mechanism won't discriminate multiple differences within a line. What would be more useful is a diff mechanism that displayed *word* differences. Another limitation of the standard diff mechanism is that it represents differences with textual output in diff format. It would be useful for these to be visually displayed. As we have seen, TextMate addresses this by helpfully providing syntax coloring for the diff output. Another alternative is to use a diff mechanism with a GUI interface.[16] One such alternative that addresses both these limitations is available on Mac OS X. If you install the developer tools that comes with Mac OS X[17], one useful tool that you will have installed is FileMerge. FileMerge is a GUI diff program that visually displays differences between files. And while FileMerge displays line differences, it also highlights word differences. To use FileMerge as your diff mechanism from within TextMate, you need to download and install the script, `fmdiff`.[18] Then, in TextMate's preferences, under Advanced, Shell Variables assign `fmdiff` as the value of the variable `TM_SVN_DIFF_CMD`. Wow. That's a lot of work. But here's the payoff. When you call any of the diff commands discussed in this section, word differences will be visually displayed in a nice GUI. (See Figure 12.)

16. See [2] for further discussion of these limitations and the alternatives.
17. These are on the CD that installs Mac OS X and can be downloaded by members of the Apple Developer Connection (membership is free) at: `http://developer.apple.com/tools/download`.
18. `http://ssel.vub.ac.be/ssel/internal:fmdiff`

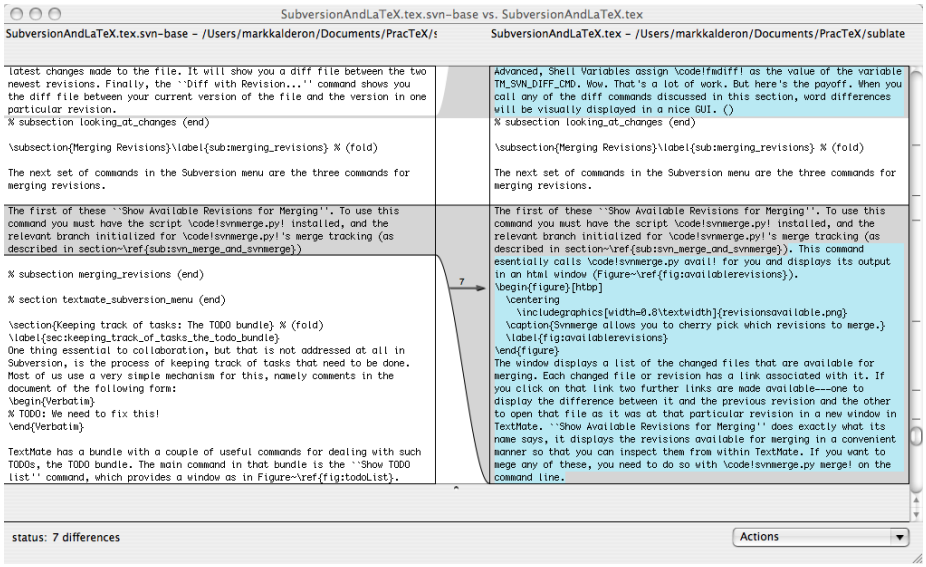

Figure 12: FileMerge graphically represents word and line differences.

3.4 Merging Revisions

The next set of commands in the Subversion menu are the three commands for merging revisions.

The first of these is "Show Available Revisions for Merging". To use this command you must have the script `svnmerge.py` installed, and the relevant branch initialized for `svnmerge.py`'s merge tracking (as described in section 2.4). This command essentially calls `svnmerge.py avail` for you and displays its output in an html window (Figure 13). The window displays a list of the changed files that are available for merging. Each changed file or revision has a link associated with it. If you click on that link two further links are made available—one to display the differences between it and the previous revision and the other to open that file as it was at that particular revision in a new window in TextMate. Note that "Show Available Revisions for Merging" does exactly what its name says, it displays the revisions available for merging in a convenient manner so that you can inspect them from within TextMate. If you want to merge any of these, you need to do so with `svnmerge.py merge` on the command line.

The next command is "Resolve Conflicts with FileMerge". To use this com-

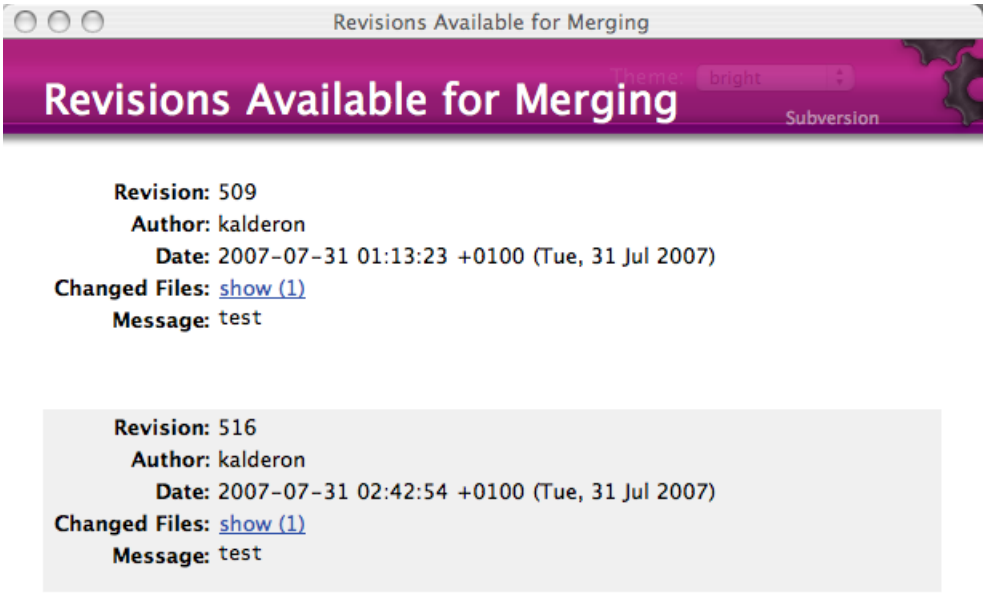

Figure 13: Svnmerge allows you to cherry pick which revisions to merge.

mand you must install Apple's developer's tools and the script `fmdiff` (as described at the end of section 3.3). As we noted in section 2.4, merging changes between a file in a repository that conflicts with the version of that file in your working copy must be done by hand. It cannot be done automatically since it requires the application of human judgement and, in a collaborative setting, consensus among coauthors.[19] In manually resolving conflicts, it would be useful for the conflicting changes to be explicitly displayed. "Resolve Conflicts with FileMerge" visually displays these conflicting changes in FileMerge's GUI. (See Figure 12.)

The next command is "Resolved". Manually resolving conflicts involves making the relevant changes to the file in your working copy. Suppose you have made these changes and that you and your coauthors are satisfied with them. Before you can commit these changes, you must tell Subversion that the conflict has been resolved. On the command line, this is done with the command:

```
$ svn resolved mymainfile.tex
```

19. See the subversion manual [1] for detailed instructions about resolving conflicts by hand

where `mymainfile.tex` is the file in your working copy that is in a conflict state. This command tells Subversion that the conflict has been resolved and that it is OK to commit these changes to the repository. "Resolved" is essentially a wrapper for `svn resolved`. Having made the relevant changes to `mymainfile.tex` within TextMate, there's no need to go to the command line—you can tell Subversion that the conflict has been resolved within TextMate with this command.

4 Keeping track of tasks: The TODO bundle

One thing essential to collaboration, but that is not addressed at all in Subversion, is the process of keeping track of tasks that need to be done. Most of us use a very simple mechanism for this, namely comments in the document of the following form:

```
% TODO: We need to fix this!
```

TextMate has a bundle with a couple of useful commands for dealing with such TODOs, the TODO bundle. The main command in that bundle is the "Show TODO list" command, which provides a window as in Figure 14. A list of all the TODOs shows up, along with links to the location in the document where they appear. The command will actually look for TODOs in the entire project, not just the current file. The command actually recognizes a variety of tags like FIXME and CHANGED, and you can add your own tags through the bundle preferences. In Figure 14, we have added a new tag for each of the authors, and those tags look for items like the following:

```
% TODO(cskiadas): This is assigned to Charilaos
% TODO(kjosmoen): This is assigned to Thomas
% TODO(markkalderon): This is assigned to Mark
```

Used in conjunction with Subversion, the TODO bundle, in effect, constitutes an efficient ticket system.

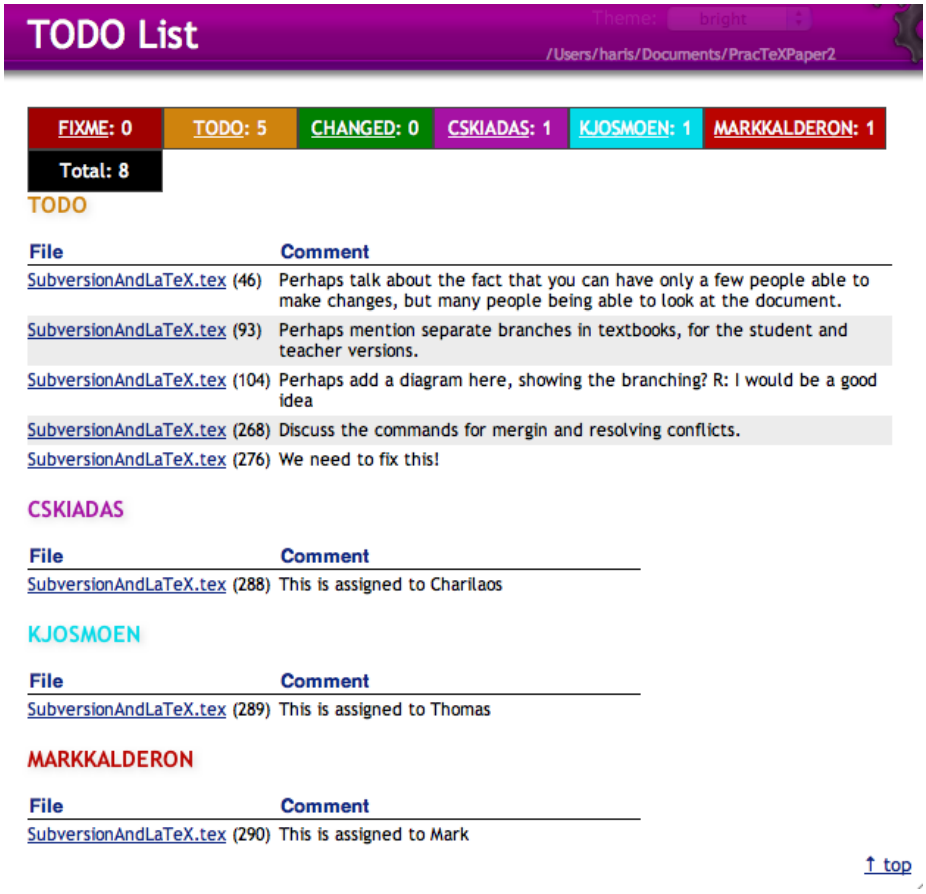

Figure 14: The TODO list window: A simple and efficient way to keep track of tasks.

5 Concluding Remarks

LaTeX is great for the construction of complex documents. Some form of version control is really indispensable for managing the development of your complex LaTeX document. This need will be especially apparent when that document is the collaborative effort of coauthors. Subversion, a free and open source version control system, can help meet this need. Keeping your LaTeX documents in version control with Subversion is made easier if your LaTeX-aware editor has Subversion integration. We have discussed the Subversion integration of one popular LaTeX-aware editor on the Mac OS X platform, TextMate. Other LaTeX-aware editors on Mac OS X with subversion integration include BBEdit[20], Aquamacs[21] and Carbon Emacs[22], to name a few. For more information about Subversion, consult the book, *Version Control with Subversion* [1]. This book is available free online in both pdf and html format.[23] Besides being free, it is a well written and user-friendly introduction to Subversion. Consult it, and consider using Subversion for your next collaborative LaTeX project. And for a really enjoyable collaborative LaTeX experience, try using TextMate.

References

[1] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion*. O'Reilly, 2004.

[2] Mark Eli Kalderon. LaTeX and Subversion. *The PracTeX Journal*, (3), 2007.

[3] Charilaos Skiadas and Thomas Kjosmoen. LaTeXing with TextMate. *The PracTeX Journal*, (3), 2007.

20. http://www.barebones.com/products/bbedit
21. http://aquamacs.org
22. http://homepage.mac.com/zenitani/emacs-e.html
23. http://svnbook.red-bean.com