Chapter 3

Revision Control

We begin our journey into software engineering before we write a single line of code. Revision control systems (RCSes) such as Subversion or CVS are astoundingly useful for single-developer projects, and essential for even the smallest group projects. These systems allow deveopers to obtain a copy of the current source code, work on it, and then "check in" their changes. The systems permit you to go back to any earlier version, and see the changes that were made between revisions. A good revision control system provides several benefits:

- **Super-undo**: You can go back to arbitrary saved versions of your source code.
- **Backups**: Using a revision control system means that you can keep the source code master copy on a safe, well-maintained machine, and edit/compile/develop on an arbitrary machine (such as your laptop).
- **Tracking changes**: The RCS can give you a list of all of the changes that affected a particular file (or project). This can be very useful in figuring out when something broke—and who broke it.
- **Concurrent access**: Many RCSes are designed to allow concurrent, safe access to the source code. More about this later.
- **Snapshots**: Is your code at a particular good working point? (e.g., a release, or in the case of 15-441, ready for a checkpoint.) An RCS should let you "snapshot" the code at that point with a memorable name so that you can easily access it later.
- **Branches**: A branch allows you to commit changes to an "alternate" copy of the code without affecting the main branch. A good example is the optimization contest in the second 15-441 assignment. With a branch, you can create a separate, derived tree for your code where you play around with ideas, knowing that they won't break the main, safe branch. You don't have to use branches, and they're a mildly "advanced feature," but if you're putting serious effort into the optimization contest, you might want to consider using them.

As a random aside, note that revision control is useful beyond just code. I (Dave) use subversion to store papers that my research group is working on, my c.v., my web pages, the configuration files for some of my machines, and so on. I use it exactly as I mentioned above in "backups" to keep all of my documents and projects on a server, while being able to edit them on my laptop.

Original file	**Modified file**

`This is a test file`
`It started with two lines`
	`This is a test file`
	`It no longer has two lines`
	`it has three`

The "universal" diff (`diff -u`) between the two files is:

```
--- file        Mon Aug 28 11:31:53 2006
+++ file2       Mon Aug 28 11:32:07 2006
@@ -1,2 +1,3 @@
 This is a test file
-It started with two lines
+It no longer has two lines
+it has three
```

Figure 3.1: Diff of two small files.

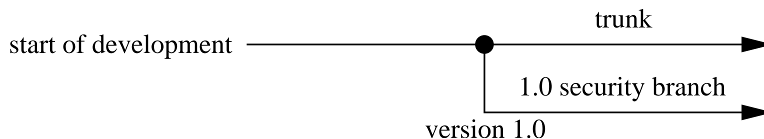

Figure 3.2: Two branches and a tag marking version 1.0.

3.1 Revision Control Concepts

Concept 1: The master copy is stored in a repository. Revision control systems store their data in a "repository" that is separate from the copy you're editing. In many cases, the repository may be a database or a set of binary files. To obtain a copy of the repository, you *check out* a copy. Doing so creates a local copy of the source code tree. The changes you make to files are only saved to the repository when you explicitly choose to do so by performing a *commit* operation. The copy of the code that you're working on is often called a *working copy*.

Concept 2: Every revision is available. When you commit to the repository, the changes you've made since your previous commit (or since check-out) are all saved to the repository. You can later see each of these revisions; perhaps more importantly, you can view a *diff* between arbitrary versions, showing only what changed. Diffs are most commonly expressed in unix `diff` format (Figure 3.1).

Concept 3: Multiple branches of development. Most RC systems support the idea of multiple lines, or "branches" of development. For example, consider a project that has a main line of development (which we will call the "trunk"). At some time, this project makes a public release of version 1.0. Afterwords, it releases only critical security updates to version 1.0, while continuing to add features to the "trunk" in preparation for version 2.0. (Figure 3.2).

Branches are a great idea, but they also add complexity for the developer, because someone must often ensure that desirable changes to one branch get propagated to the other branch, which may have somewhat different code. This process is called *merging*, and is similar to resolving conflicts (see

below).

Concept 4: Concurrent development. The final benefit from an RC system is that it lets multiple developers work on the code concurrently. Each checks out his or her own copy of the code, and begins editing. Developers commit their changes independently, and can *update* to receive changes that have been committed to the repository since they checked out the code or last updated.

This works perfectly if the developers are working on different parts of the code, but what if they are editing the same file at the same time? The answer depends. If the diffs are reasonably separate (different parts of the file, for example), then the system will usually be able to *merge* them automatically. If, however, the updates touch similar lines of code, then the updates will *conflict*, and the revision control system will report this to the user. The user must then merge the changes manually.

3.2 A practical introduction to Subversion

In 15-441, you'll use Subversion for source control. This section explains some of the basic subversion commands and how to make your way around the system.

1. **Check out the repository**
   The course staff will have created a repository for you to use. It will initially be empty.

   ```
   svn checkout https://moo.cmcl.cs.cmu.edu/441/441proj1-group1
   ```

   This will create a local directory called 441proj1-group1 (please replace with your own group name as appropriate), and fill it with the latest version of your source code.

2. **Add a file to the repository**
   Create the file you wish to add (or copy it) to the repository. Put it in your checked out source code tree wherever you want. Then

   ```
   svn add <filename>
   ```

   Note that this step will *schedule* the file to be added, but until you commit, it will just be in your local copy.

3. **Add a directory**
   You can add a directory to the repository by either making it using mkdir and adding it, or by using `svn mkdir`. We suggest using the latter, because it means that subversion is immediately aware that the directory exists, and can use it as a target for move operations.

   ```
   svn mkdir bar
   ```

4. **Commit your changes**

   ```
   svn commit
   ```

You can commit from either the top of your directory tree (in which case all of your outstanding changes will be committed), or from a sub-directory, or just for a particular file by naming it after the commit (e.g., `svn commit foo`). This will send your local changes to the repository. When you commit, svn will prompt you for a log message to describe, for the benefit of you and your fellow developers, the changes you've made. Don't leave this blank—good log messages are very useful for debugging and to help coordinate with your partner.

If the version of the file you were editing is not the latest one in the repository, `commit` will fail and let you know. At this point, you'll need to update (5) and perhaps resolve any conflicts between your edits and the previous ones (Section 3.2.1 below).

5. **Update to get the latest changes**

   ```
   svn update
   ```

   This will check out the latest changes from the repository. Subversion will print messages indicating what files were changed. For example, if the update added a new README file and changed the Makefile, subversion would indicate this as:

   ```
   A  src/trunk/README
   U  src/trunk/Makefile
   ```

6. **Make a snapshot of your code to hand in**

   Subversion uses the "copy" command to create both tags and branches. (Internally, they're the same thing, but humans treat them differently by not committing to a tag). By convention, subversion tags are stored in a top-level "tags" directory. We'll use this for checking out a copy of your projects so that you can keep developing without clobbering the handed-in version.

   ```
   svn copy trunk tags/checkpoint1
   ```

7. **See what changed in a file**

   To see the changes you've made since your last checked out or updated a file, use:

   ```
   svn diff file
   ```

   You can also see the difference between arbitrary revisions of the file. For example, to see the difference between revision 1 and revision 2 of a file:

   ```
   svn diff -r 1:2 file
   ```

You can also use `svn log` to see what changes have been recorded to a file.
Table 3.1 lists other commands you may find useful:

`svn remove`	Remove a file or directory
`svn move`	Move or rename a file or directory
`svn status`	Show what files have been added, removed, or changed
`svn log`	Show the log of changes to a file (the human-entered comments)
`svn blame`	Show each line of a file together with who last edited it (also `svn annotate`).

Table 3.1: Other useful Subversion comments

3.2.1 Resolving conflicts

Eventually, you'll have conflicting edits to a file. Fortunately, resolving them in subversion isn't too hard. Let's say that you and your partner both edit the file `testfile`. Your partner commits her changes first. When you update, you'll see that your edits conflict with the version in the repository:

```
~/conflict-example> svn up
C    testfile
Updated to revision 17.
```

If you look in the directory, you'll see that subversion has put a few copies of the file there for you to look at:

```
~/conflict > ls
testfile        testfile.mine   testfile.r16    testfile.r17
```

The file `testfile` has the "conflict" listed. The text from the two revisions is separated by seven "<" "=" and ">" markers that show which text came from which revision, such as:

```
This is a test file

It has a few paragraphs,
to which we'll be making some
conflicting edits later in this
<<<<<<< .mine
This line was added on machine 1
=======
This line was added on machine 2 - hah, I got it committed first!
>>>>>>> .r17
exercise.

Yes, a few paragraphs it has.
Wooo, that's cool.  I like paragraphs!
```

You can see that one version has a line that says "This line was added on machine 1" and the other version has a line that says "This line was added on machine 2 - hah, I got it committed first!" You have a few options for resolving the conflict:

1. **Throw out your changes**. If you just want your partner's changes, you can copy her file on top of yours. Subversion conveniently supplies you with a copy of the latest version of the file; in this case, it's `testfile.r17`.

   `cp testfile.r17 testfile`.

   Then tell subversion you're done merging:

   `svn resolved testfile`

   and commit.

2. **Overwrite your partner's changes**. Just like the previous example, but copy `testfile.mine` onto testfile instead.

3. **Merging by hand**. Open `testfile` in an editor and search for the conflict markers. Then select which of the versions (if either) you want to preserve, and update the text to reflect that. When you're done, `svn resolved` and commit.

3.3 Thoughts on using revision control

A selection of thoughts from our experience using revision control.

- **Update, make, test, and then commit**. It's good to test your changes with the full repository before you commit them, in case they broke something.

- **Merge relatively often.** We come from the "merge often" school. See the discussion in the next section about breaking down your changes into manageable chunks.

- **Commit formatting changes separately.** Why? So that you can more accurately identify the source of code and particular changes. A commit that bundles new features with formatting changes makes it difficult to examine exactly what was changed to add the features, etc.

- **Check `svn diff` before committing.** It's a nice way to check that you're changing what you meant to. We've often discovered that we left in weird debugging changes or outright typos and have avoided committing them by a quick check.

- **Try not to break the checked in copy.** It's convenient for you and your partner if the current version of the source code always at least compiles. We suggest breaking your changes down into manageable chunks and committing them as you complete them.

  For example, in Project 1, you may start out by first creating a server that listens for connections and closes them immediately. This would be a nice sized chunk for a commit. Next, you might modify it to echo back all text it receives. Another commit. Then create your data structure to hold connection information, and a unit test case to make sure the data structure works. Commit. And so on.

  If you're going to make more invasive changes, you may want to think about using a branch. A good example of branches in 15-441 is the optimization contest for the second project: You may want to experiment with techniques that could break your basic operation, but you don't want to risk failing test cases just to make your program faster. Ergo, the "optimization"

RCS	An early source control system. Allows files to be locked and unlocked; does not address concurrent use and conflict resolution. Sometimes used for web pages and configuration files where changes occur slowly but revision control is useful.
CVS	The Concurrent Version System. CVS is built atop RCS and uses its underlying mechanisms to version single files. Very popular. Major users include the FreeBSD project and many other open source systems.
Subversion	Subversion is designed to fix many of the flaws in CVS while retaining a familiar interface. Adds a number of capabilities to CVS (e.g., the ability to rename files and directories) without breaking the basics. Quickly gaining popularity among open source projects.
Bitkeeper	A commercial distributed source control system. BitKeeper used to be used for the Linux kernel.
Git	A relatively new source control system used for the Linux kernel.
Visual SourceSafe	Microsoft's source control system.
Perforce	A popular, heavy-weight commercial revision control system.

Table 3.2: Popular revision control systems.

branch. You can make changes in this branch (and save your work and get all of the benefits of source control) without affecting your main branch. Of course, if you create a branch and like the ideas from it, you'll have to merge those changes back later. You can either use `svn merge`, or you can `svn diff` and then patch the files manually. The nice thing about using `svn merge` is that it records which change you're propagating.

- **Use meaningful log messages**. Even for yourself, it's great to be able to go back and say, "Ah-ha! That's why I made that change." Reading a diff is harder than reading a *good* log message that briefly describes the changes and the reason for them.

- **Avoid conflicts by good decomposition and out-of-band coordination**. Revision control is great, but conflicts can be a bit of a pain. To keep them to a minimum:

  – Make your program modular. If one person can work independently on, say, the user IRC message parsing code while the other works on the routing code, it will reduce the chances of conflicts—and it's good programming practice that will reduce your headaches in lots of other ways. Have this modularity reflected in the way you put what code in what file.

  – Coordinate out of band with your partner. Don't just sit down and start working on "whatever"—let your partner know what you're working on.

3.4 Other source control systems

Table 3.2 lists a number of other popular source control systems that you might encounter after this class.

3.5 Trying it out on your own

Subversion is installed on the Andrew linux machines. To create a repository, try:

```
mkdir svn
svnadmin create svn/test
svn checkout file:///afs/andrew.cmu.edu/usr23/dga2/svn/test
```

(Of course, replace the path to my repository with that to your repository!)

You'll be able to access this repository from any machine with AFS access. You can also access it via SSH by specifying the URL as:

```
svn checkout  \
 svn+ssh://you@linux.andrew.cmu.edu/afs/andrew....../svn/test
```

3.6 Recommended Reading

The best subversion book is "Version Control with Subversion." It's available in book form and for free online [1].

Bibliography

[1] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion*. O'Reilly and Associates. ISBN 0-596-00448-6. Available online at `http://svnbook.red-bean.com/`.