Version Control Standards for Senior Design Project

Reference:
https://homes.cs.washington.edu/~mernst/advice/version-control.html#best-practices

Version control best practices

The advice in this section applies to both centralized and distributed version control.
These best practices do not cover obscure or complex situations. Once you have mastered these practices, you can find more tips and tricks elsewhere on the Internet.

Use a descriptive commit message

It only takes a moment to write a good commit message. This is useful when someone is examining the change, because it indicates the purpose of the change. This is useful when someone is looking for changes related to a given concept, because they can search through the commit messages.

Make each commit a logical unit

Each commit should have a single purpose and should completely implement that purpose. This makes it easier to locate the changes related to some particular feature or bug fix, to see them all in one place, to undo them, to determine the changes that are responsible for buggy behavior, etc. The utility of the version control history is compromised if one commit contains code that serves multiple purposes, or if code for a particular purpose is spread across multiple different commits.
During the course of one task, you may notice another issue and want to fix it too. You may need to commit one file at a time — the commit command of every version control system supports this.

- Git: git commit *file1 file2* commits the two named files.
- Alternately, git add *file1 file2* “stages” the the two named files, causing them to be committed by the next git commit command that is run without any filename arguments.
- Mercurial: hg commit *file1 file2* commits the two named files, and hg commit . commits all the changed files in the current directory.
- Subversion: svn commit *file1 file2* commits the two named files, and svn commit . commits all the changed files in the current directory.

If a single file contains changes that serve multiple purposes, you may need to save your all your edits, then re-introduce them in logical chunks, committing as you go. Here is a low-tech way to do this; each version control system also has more sophisticated mechanisms to support this common operation.

- Git: Move *myfile* to a safe temporary location, then run git checkout *myfile* to restore *myfile* to its unmodified state (same as whatever is in the repository).
- Git contains more sophisticated ways to do this, such as staging some but not all of the changes in a given file to the index (also known as the cache), or stashing some of your changes. Once you are more comfortable with Git, you should learn about these mechanisms.
- Mercurial: hg revert *myfile* copies the current *myfile* to *myfile*.orig and restores *myfile* to its unmodified state (same as whatever is in the repository).
- Subversion: Move *myfile* to a safe temporary location, then run svn update *myfile* to restore *myfile* to its unmodified state (same as whatever is in the repository).

Sometimes it is too burdensome to separate every change into its own commit. However, aiming for (and often achieving) this goal will serve you well in the longer term.

Avoid indiscriminate commits

As a rule, I do not run git commit -a (or hg commit or svn commit) without supplying specific files to commit. If you supply no file names, then these commands commit every changed file. You may have changes you did not intend to make permanent (such as temporary debugging changes); even if not, this creates a single commit with multiple purposes.
When I want to commit my changes, to avoid accidentally committing more than I intended, I always run the following commands:
For Git:

```
 # Lists all the modified files
  git status
  # Shows specific differences, helps me compose a commit message
  git diff
  # Commits just the files I want to
  git commit file1 file2 -m "Descriptive commit message"
```

For Mercurial:

```
 # Lists all the modified files
  hg status
  # Shows specific differences, helps me compose a commit message
  hg diff
  # Commits just the files I want to
  hg commit file1 file2 -m "Descriptive commit message"
```

Incorporate others' changes frequently

Work with the most up-to-date version of the files as possible. That means that you should run git pull, git pull -r, hg fetch, or svn update very frequently. I do this every day, on each of over 100 projects that I am involved with.

When two people make conflicting edits simultaneously, then manual intervention is required to resolve the conflict. But if someone else has already completed a change before you even start to edit, it is a huge waste of time to create, then manually resolve, conflicts. You would have avoided the conflicts if your working copy had already contained the other person's changes before you started to edit.

Share your changes frequently

Once you have committed the changes for a complete, logical unit of work, you should share those changes with your colleagues as soon as possible (by doing git push or hg push). So long as your changes do not destabilize the system, do not hold the changes locally while you make unrelated changes. The reason is the same as the reason for incorporating others' changes frequently.
This advice is slightly different for centralized version control such as Subversion. This advice translates to running svn commit, which both commits and shares your changes, as often as possible. However, be careful because you cannot make private commits that do not affect your teammates.

Coordinate with your co-workers

The version control system can often merge changes that different people made simultaneously. However, when two people edit the same line, then this is a *conflict* that a person must manually resolve. To avoid this tedious, error-prone work, you should strive to avoid conflicts.
If you plan to make significant changes to (a part of) a file that others may be editing, coordinate with them so that one of you can finish work (commit and push it) before the other gets started. This is the best way to avoid conflicts. A special case of this is any change that touches many files (or parts of them), which requires you to coordinate with all your teammates.

Remember that the tools are line-based

Version control tools record changes and determine conflicts on a line-by-line basis. The following advice applies to editing marked-up text (LaTeX, HTML, etc.). It does not apply when editing WYSIWYG text (such as a plain text file), in which the intended reader sees the original source file.
Never refill/rejustify paragraphs. Doing so changes every line of the paragraph. This makes it hard to determine, later, what part of the content changed in a given commit. It also makes it hard for others to determine which commits affected given content (as opposed to just reformatting it). If you follow this advice and do not refill/rejustify the text, then the LaTeX/HTML source might look a little bit funny, with some short lines in the middle of paragraphs. But, no one sees that except when editing the source, and the version control information is more important.
Do not write excessively long lines; as a general rule, keep each line to 80 characters. The more characters are on a line, the larger the chance that multiple edits will fall on the same line and thus will conflict. Also, the more characters, the harder it is to determine the exact changes

when viewing the version control history. As another benefit to authors of the document, 80-character lines are also easier to read when viewing/editing the source file.

Don't commit generated files

Version control is intended for files that people edit. Generated files should not be committed to version control. For example, do not commit binary files that result from compilation, such as .o files or .class files. Also do not commit .pdf files that are generated from a text formatting application; as a rule, you should only commit the source files from which the .pdf files are generated.

- Generated files are not necessary in version control; each user can re-generate them (typically by running a build program such as make or ant).
- Generated files are prone to conflicts. They may contain a timestamp or in some other way depend on the system configuration. It is a waste of human time to resolve such uninteresting conflicts.
- Generated files can bloat the version control history (the size of the database that is stored in the repository). A small change to a source file may result in a rather different generated file. Eventually, this affects performance of the version control system.
- This is a particular problem when the generated file is binary. Version control systems can concisely record the differences between two versions of a textual file (usually the differences are much smaller than the file itself). However, version control systems have to store each version of a binary file in its entirety.

To tell your version control system to ignore given files, create a top-level .gitignore or .hgignore file, or set the svn:ignore property.

Understand your merge tool

The least pleasant part of working with version control is resolving conflicts. If you follow best practices, you will have to resolve conflicts relatively rarely.
You are most likely to create conflicts at a time you are stressed out, such as near a deadline. You do not have time, and are not in a good mental state, to learn a merge tool. So, you should make sure that you understand your merge tool ahead of time. When an actual conflict comes up, you don't want to be thrown into an unfamiliar UI and make mistakes. Practice on a temporary repository to give yourself confidence.
A version control system lets you choose from a variety of merge programs (example: Mercurial merge programs). Select the one you like best. If you don't want an interactive program to be run, you can configure Mercurial to attempt the merge and write a file with conflict markers if the merge is not successful.

Obtaining your copy

Obtaining your own working copy of the project is called "cloning" or "checking out":

- git clone *URL*
- hg clone *URL*
- svn checkout *URL*

Use your version control's documentation to learn how to create a new repository (hg init, git init, or svnadmin create).

Distributed version control best practices

Typical workflow

The typical workflow when using Git (or Mercurial) is:

- git pull (or hg fetch)
- As many times as desired (but usually very few times):
    - Make local edits
    - Examine the local edits: git status and git diff (or hg status and hg diff
    - git commit (or hg commit)
    - git pull or git pull -r (or hg commit)
- git pull or git pull -r (or hg commit)
- git push (or hg push)

Note that an invocation of git pull or hg fetch may force you to resolve a conflict.