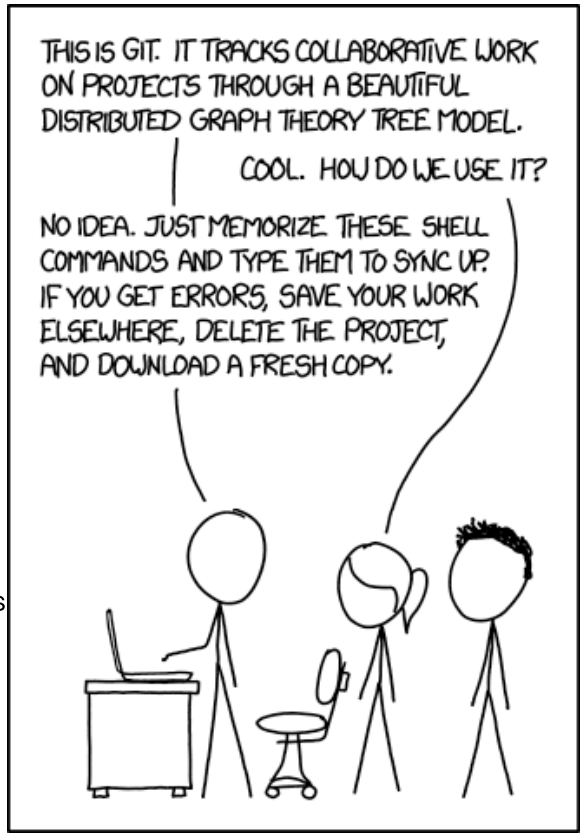
Version control concepts and best practices

by Michael Ernst



September, 2012
Last updated: April 8, 2024

This document is a brief introduction to version control. After reading it, you will be prepared to perform simple tasks using a version control system, and to learn more from other documents that may lack a high-level conceptual overview. Most of its advice is applicable to all version control systems, but its examples mostly use Git for concreteness.

This document's main purpose is to lay out philosophy and advice that I haven't found elsewhere in one place. It is not an exhaustive reference to the syntax of particular commands. This document covers basics, but it does not go into advanced topics like branching, nor does it discuss the ways in which large projects use version control differently than small ones.

Contents:

(Also see How to create and review a GitHub pull request.)

Introduction to version control

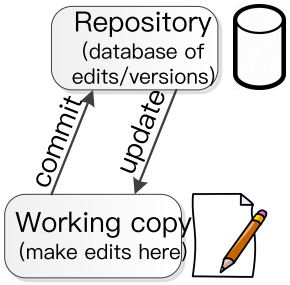
If you are already familiar with version control, you can skim or skip this section.

A version control system serves the following purposes, among others.

- Version control enables multiple people to simultaneously work on a single project. Each person edits his or her own copy of the files and chooses when to share those changes with the rest of the team. Thus, temporary or partial edits by one person do not interfere with another person's work.
  Version control also enables one person to use multiple computers to work on a project, so it is valuable even if you are working by yourself.
- Version control integrates work done simultaneously by different team members. In most cases, edits to different files or even the same file can be combined without losing any work. If two people make *conflicting edits* to the same line of a file, then the version control system requests human assistance in deciding what to do.
- Version control gives access to historical versions of your project. This is insurance against computer crashes or data lossage. If you make a mistake, you can roll back to a previous version. You can reproduce and understand a bug report on a past version of your software. You can undo specific edits without losing all the work that was done in the meanwhile. For any part of a file, you can determine when, why, and by whom it was ever edited.

Repositories and working copies

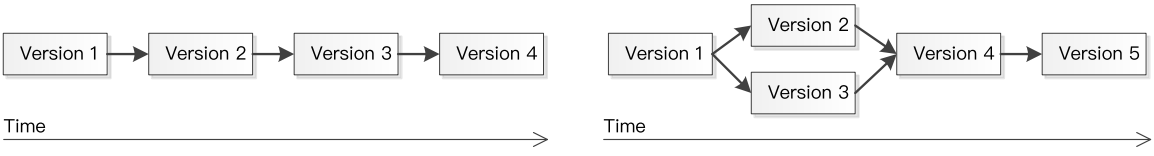
Version control uses a *repository* (a database of program versions) and a *working copy* where you edit files.



Your *working copy* (sometimes called a *checkout* or *clone*) is your personal copy of all the files in the project. You make arbitrary edits to this copy, without affecting your teammates. When you are happy with your edits, you `commit` your changes to a *repository*.

A repository is a database of all the edits to your project. Equivalently, it is a database of all historical versions (snapshots) of your project. It is possible for the repository to contain edits that have not yet been applied to your working copy. You can `update` your working copy to incorporate any new edits or versions that have been added to the repository since the last time you updated. See the diagram at the right.

In the simplest case, the database contains a linear history: each change is made after the previous one. Another possibility is that different users made edits simultaneously (this is sometimes called “branching”). In that case, the version history splits and then merges again. The picture below gives examples. In the timeline on the right, Version 4 is called a "merge".



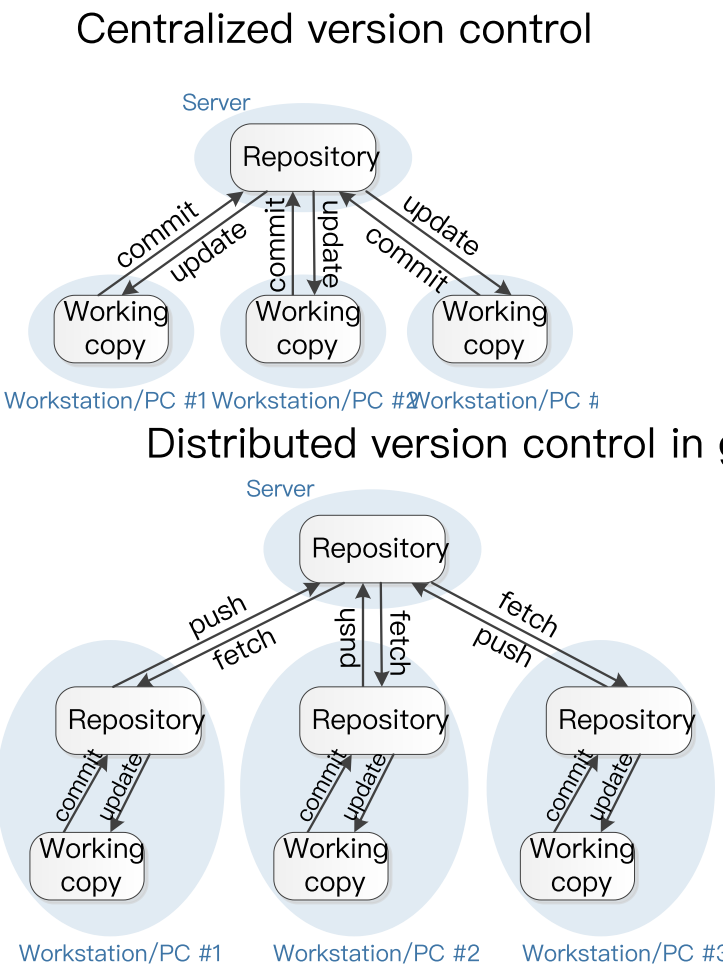
Distributed and centralized version control

There are two general varieties of version control: *centralized* and *distributed*. Distributed version control is more modern, runs faster, is less prone to errors, has more features, and is more complex to understand. Centralized version control is rarely used nowadays.

The most popular version control system is Git (distributed). In practice, most projects use Git. Other version control systems are Mercurial (distributed) and Subversion (centralized).

The main difference between centralized and distributed version control is the number of repositories. In centralized version control, there is just one repository, and in distributed version control, there are multiple

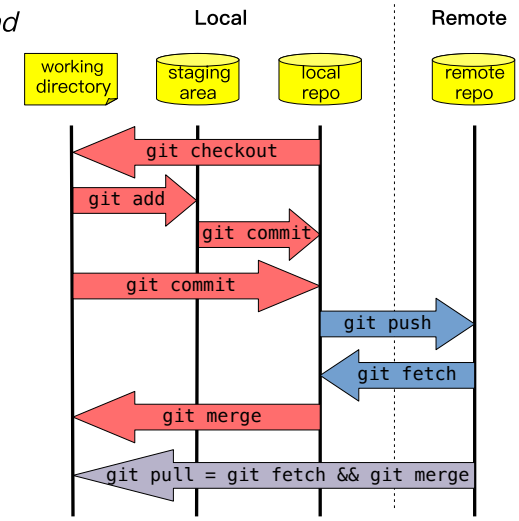
repositories. Here are pictures of the typical arrangements:



In **centralized version control**, each user gets their own working copy, but there is just one central repository. As soon as you commit, it is possible for your co-workers to update and to see your changes. For others to see your changes, 2 things must happen:

- You commit
- They update

In **distributed version control**, each user gets their own repository *and* working copy. After you commit, others have no access to your changes until you push your changes to the central repository. When you update, you do not get others' changes unless you have first fetched those changes into your local repository. For others to see your changes, 4 things must happen:



- You commit
- You push
- They fetch
- They update
  (Note that `git pull` does both `fetch` and `update`, in one git command.)

Notice that the commit and update commands only move changes between the working copy and the local repository, without affecting any other repository. By contrast, the push and fetch commands move changes between the local repository and the central repository, without affecting your working copy.

The `git pull` command is equivalent to `git fetch` then `git update`. This is handy because you usually want to perform both operations. But, this makes Git's terminology a bit confusing, since `push` and `pull` are not

complements of one another. (Mercurial's naming is more logical: Mercurial's `pull` operation is like git's `fetch`, and Mercurial's `fetch` operation is like git's `pull`; that is, `hg fetch` performs both `hg pull` and `hg update`.)

The diagram at right shows which operations affect the working copy (red), which affect the repository (blue), and which affect both (purple). Merging, `git add`, and the staging area are explained later.

Conflicts

A version control system lets multiple users simultaneously edit their own copies of a project. Usually, the version control system is able to merge simultaneous changes by two different users: for each line, the final version is the original version if neither user edited it, or is the edited version if one of the users edited it. A *conflict* occurs when two different users make simultaneous, different changes to the same line of a file. In this case, the version control system cannot automatically decide which of the two edits to use (or a combination of them, or neither!). Manual intervention is required to resolve the conflict.

"Simultaneous" changes do not necessarily happen at the exact same moment of time. Change 1 and Change 2 are considered simultaneous if:

- User A makes Change 1 before User A does an update that brings Change 2 into User A's working copy, and
- User B makes Change 2 before User B does an update that brings Change 1 into User B's working copy.

In a distributed version control system, `push` and `fetch` never cause a conflict. These operations can cause the repository to contain multiple different (perhaps mutually exclusive) histories that coexist. There is an explicit operation, called `merge`, that combines simultaneous edits by two different users. Sometimes `merge` completes automatically, but if there is a conflict, `merge` requests help from the user.

Usually you don't have to run `merge` explicitly, because it is automatically run by the `update` command (recall that `git pull` runs `git fetch` then `git update`).

It is better to avoid a conflict than to resolve it later. The best practices below give ways to avoid conflicts; for example, teammates should frequently share their changes with one another.

Conflicts are bound to arise despite your best efforts. It's smart to practice conflict resolution ahead of time, rather than when you are frazzled by a conflict in a real project. You can do so in this tutorial about Git conflict resolution.

If you run `git config --global merge.conflictStyle zdiff3` on every computer where you use git, then git's merge conflicts will be more informative because they will show not just the differences between the two parent commits of the conflict, but also the common ancestor of the two parents. Knowing the common ancestor is often essential to resolving the merge conflict correctly.

Merging changes

Recall that `update` changes the working copy by applying any edits that appear in the repository but have not yet been applied to the working copy.

In a centralized version control system, you can `update` (for example, `svn update`) at any moment, even if you have locally–uncommitted changes. The version control system merges your uncompleted changes in the working copy with the ones in the repository. This may force you to resolve conflicts. It also loses the exact set of edits you had made, since afterward you only have the combined version. The implicit merging that a centralized version control system performs when you `update` is a common source of confusion and mistakes.

In a distributed version control system, if you have uncommitted changes in your working copy, then in some cases you cannot run `update` (or other commands like `git pull` that themselves invoke `update`). The reason is

that it would be confusing and error-prone for the version control system to try to apply edits, when you are in the middle of editing. You will receive an error message such as

```
abort: outstanding uncommitted changes
```

Before you are allowed to `update`, you must first `commit` any changes that you have made (you should continue editing until they are logically complete first, of course). Now, your repository database contains *simultaneous* edits — the ones you just made, and the ones that were already there and you were trying to apply to your working copy by running `update`. You need to `merge` these two sets of edits, then `commit` the result. The reason you need the commit is that merging is an operation that gets recorded by the version control system, in order to record any choices that you made during merging. In this way, the version control system contains a complete history and clearly records the difference between you making edits and you merging simultaneous work.

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 to the main branch should have a single purpose and should completely implement that purpose. For example, a commit should not contain a bug fix, a new feature, and a typo fix; you should make three different commits instead. Many commits to the main branch are made via pull requests from a different branch. Therefore, this advice can also be stated as: each branch should have a single purpose and, at the time it is merged into the main branch, 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 something else you want to change. You may need to commit one file at a time — the `commit` command of every version control system supports this.
In Git, `git commit` *`file1 file2`* commits the two named files.
Alternately, `git add` *`file1 file2`* "stages" the two named files, causing them to be committed by the next `git commit` command that is run without any filename arguments.

You can also stage part of a file. This is useful if you want to exclude local changes, such as diagnostics. Version control tools support interactive staging, in which tool asks the user, for each changed hunk, whether to stage that hunk or not. It can take a little while to get used to these tools and to the staging area more generally, but you will eventually find them quite useful.

Avoid indiscriminate commits

It is very easy to commit more changes than you intended to. Double-check the changes before you make a commit. Here are some commands that you should likely run before each commit.

```
# Lists all the modified files
git status

# Shows specific differences, helps me compose a commit message.
# If I am using the staging area:
git diff --staged
# Whether I am using the staging area or not:
git diff

# If I am using the staging area:
git commit -m "Descriptive commit message"
# If I am not using the staging area, commit just the files I want to:
git 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` very frequently. I do this every day, on each of the hundreds of projects that I am involved with. (I use the program multi-version-control.)

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`). So long as your changes do not destabilize the system, you should not hold the changes locally while you make unrelated changes. The reason is the same as the reason for incorporating others' changes frequently.

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 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. Examples include a wide-scale renaming or other code reorganization.

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/justify paragraphs. Doing so changes every line of the paragraph and makes merge conflicts likely. Refilling paragraphs also makes it hard to determine, later, what part of the content changed in a given commit, and 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 about 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 by running a build program such as `make`, `gradle`, `mvn`, `ant`, etc.
- 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 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 `.gitignore` at the top level of your repository, or create a `.gitignore-global` directory in your user home directory.

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. You can do so in this tutorial about Git conflict resolution.

A version control system lets you choose from a variety of merge tools; to see Git's list, run `git mergetool --tool-help`. Select the one you like best. Many of the merge tools start an interactive GUI program. If you don't want that, you can configure your version control system to attempt the merge and write a file with conflict markers if the merge is not successful. Then, you can use your favorite editor, or a tool that is not a git mergetool.

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. Example commands include `git init`, `hg init`, and `svnadmin create`, but you will more often create a repository at a hosting site such as GitHub, then clone it locally.

Distributed version control best practices

Typical workflow

A typical workflow when using Git is:

- On the main branch: `git pull`
- `git branch` *`NEW-BRANCH-NAME`*
- `git checkout` *`NEW-BRANCH-NAME`*
- As many times as desired:
  - Make local edits.
  - Examine the local edits: `git status` and `git diff`
  - `git commit`, or `git add` then `git commit`
  - `git pull`
- Ensure that tests pass.
- `git push`
- Make a pull request for branch *NEW-BRANCH-NAME*

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

That's pretty much all you need to know, besides how to clone an existing repository.

In Mercurial, use `hg fetch`, not `hg pull`

(This tip is specific to Mercurial. In Git, just use `git pull`. Git's `pull` acts similarly to Mercurial's `fetch`.)

I never run `hg pull`. Instead, I use `hg fetch`. It is the most effective way to get everyone else's changes into my working copy. The `hg fetch` command is like `hg pull` then `hg update`, but it is even more useful. The actual effect of `hg fetch` is:

- `hg pull`
- If merging is necessary:
  - `hg merge`
  - `hg commit`
- `hg update`

To enable the `hg fetch` command, add the following to your `$HOME/.hgrc` file or equivalent:

```
[extensions]
fetch =
```

There is nothing after the "`=`" in "`fetch =`".

Don't force it

Git or Mercurial occasionally refuses to do a particular action, such as pushing to a remote repository when you have not yet fetched all its changes. For example, Mercurial indicates this problem by outputting:

```
abort: push creates new remote heads!
(did you forget to merge? use push -f to force)
```

In the second line of the message, Mercurial makes two suggestions:

- You can merge the changes — the best way to do this is with `hg fetch` (not `hg merge`), then you can try again to push.
- You can perform the operation by using the `-f` command-line option, which stands for "force" and can also be written as `--force`.
  Never use `-f` or `--force`: doing so is likely to cause extra work for your team, such as making multiple people perform the same merge.

Don't rewrite history

`git rebase` is a powerful command that lets you rewrite the version control history. Rebasing can change a commit, change commit messages, reorder commits, squash multiple commits into one, split one commit

into multiple commits, delete commits, and more.
Never use `rebase`, including `git pull -r`. (Until you are more experienced with git. And, then still don't use it.)

Rewriting history is ineffective if anyone else has cloned your repository. Your changes to history will be added to the existing history in all remote repositories, which is not the effect you wanted. Rewriting history frequently causes difficult merge conflicts, and it may force those merge conflicts on multiple users. Rewriting history makes it more difficult to understand the actual development history, and recording the true development history is a main purpose of a version control system.

If you want to keep your development history clean, there are better ways than rewriting history, such as squash-and-merging GitHub pull requests.

Merging when you have uncommitted changes

As explained above, you cannot `update` until you `commit` and `merge`. You will see an error message like

```
abort: outstanding uncommitted changes
```

But, sometimes you really want to incorporate others' changes even though your changes are not yet in a logically consistent state and ready to commit to your local repository.

A low-tech solution is to revert your changes with `hg revert` or the analogous command for other version control systems. Now, you can `git pull` or `hg fetch`, but you will have to manually re-do the changes that you moved aside. There are other, more sophisticated ways to do this as well (for Git, use `git stash`; for Mercurial, see the Mercurial FAQ).

More tips

Caching your password

Subversion

SVN (Subversion) automatically caches your password. You have to type the password only the first time.

Git

GitHub and other hosting services give you multiple URLs of the main repository (the one on the GitHub servers) that you can clone.

- If you use a ssh URL, such as `git@github.com:owner/repo.git`, then you can use an SSH key and the `ssh-agent` program, and you will never need to type a password. See GitHub's instructions.
- If you use a HTTPS URL, such as `https://github.com/owner/repo.git`, then you can install a credential manager, or the GitHub CLI, so you only have to type your credentials once. See GitHub's instructions.

Mercurial

Here are two ways to have Mercurial remember/cache your password so you don't have to type it every time.

1. `hg clone https://michael.ernst:`*`my-password-here`*`@jsr308-langtools.googlecode.com/hg/ jsr308-langtools`
2. In your global `.hgrc` file (which should not be world-readable!), add this section:

```
# The below only works in Mercurial 1.3 and later
[auth]

googlecode.prefix = code.google.com
googlecode.username = michael.ernst
googlecode.password = my-password-here
googlecode.schemes = https
```

```
dada.prefix = dada.cs.washington.edu/hgweb/
dada.username = mernst
dada.password = my-password-here
dada.schemes = https
```

Email notification

It's a good idea to set up email notification. Then, every time someone pushes (in distributed version control) or commits (in centralized version control) all the relevant parties get an email about the changes to the central server.

If you are using a hosted service such as GitHub or Bitbucket, it's easy to set up email notification on their website; here are the GitHub instructions.

Back to Advice compiled by Michael Ernst.

Michael Ernst