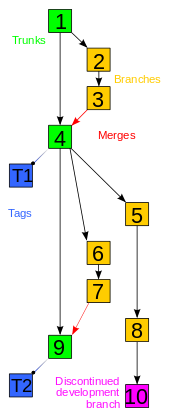
**Version Control**

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. It manages the changes to documents, computer programs, large web sites, and other collections of information. Changes are usually identified by a number or letter code, termed the "revision number," "revision level," or simply "revision." For example, an initial set of files is "revision 1." When the first change is made, the resulting set is "revision 2," and so on. Each revision is associated with a timestamp and the person making the change.

 Revision control allows for the ability to revert a document to a previous revision, which is critical for allowing editors to track each other's edits, correct mistakes, and defend against vandalism and spamming.

In computer software engineering, revision control is any kind of practice that tracks and provides control over changes to source code. Software developers sometimes use revision control software to maintain documentation and configuration files as well as source code.  Bugs or features of the software are often only present in certain versions (because of the fixing of some problems and the introduction of others as the program develops). Therefore, for the purposes of locating and fixing bugs, it is vitally important to be able to retrieve and run different versions of the software to determine in which version(s) the problem occurs.  It may also be necessary to develop two versions of the software concurrently (for instance, where one version has bugs fixed, but no new features (branch), while the other version is where new features are worked on (trunk).

Example history graph of a revision-controlled project; trunk is in green, branches in yellow, and graph is not a tree due to presence of merges (the red arrows).

**Features of VCS**

* **Backup and Restore.** Files are saved as they are edited, and you can jump to any moment in time. Need that file as it was on Feb 23, 2007? No problem.
* **Synchronization.** Lets people share files and stay up-to-date with the latest version.
* **Short-term undo.** Monkeying with a file and messed it up? (That’s just like you, isn’t it?). Throw away your changes and go back to the “last known good” version in the database.
* **Long-term undo.** Sometimes we mess up bad. Suppose you made a change a year ago, and it had a bug. Jump back to the old version, and see what change was made that day.
* **Track Changes**. As files are updated, you can leave messages explaining why the change happened (stored in the VCS, not the file). This makes it easy to see how a file is evolving over time, and why.
* **Track Ownership.** A VCS tags every change with the name of the person who made it.
* **Sandboxing**, or insurance against yourself. Making a big change? You can make temporary changes in an isolated area, test and work out the kinks before “checking in” your changes.
* **Branching and merging**. A larger sandbox. You can **branch** a copy of your code into a separate area and modify it in isolation (tracking changes separately). Later, you can**merge** your work back into the common area.

Most version control systems involve the following concepts, though the labels may be different.

Basic Setup

* **Repository (repo)**: The database storing the files.
* **Server**: The computer storing the repo.
* **Client**: The computer connecting to the repo.
* **Working Set/Working Copy**: Your local directory of files, where you make changes.
* **Trunk/Main**: The primary location for code in the repo. Think of code as a family tree — the trunk is the main line.

Basic Actions

* **Add**: Put a file into the repo for the first time, i.e. begin tracking it with Version Control.
* **Revision**: What version a file is on (v1, v2, v3, etc.).
* **Head**: The latest revision in the repo.
* **Check out**: Download a file from the repo.
* **Check in**: Upload a file to the repository (if it has changed). The file gets a new revision number, and people can “check out” the latest one.
* **Checkin Message**: A short message describing what was changed.
* **Changelog/History**: A list of changes made to a file since it was created.
* **Update/Sync**: Synchronize your files with the latest from the repository. This lets you grab the latest revisions of all files.
* **Revert**: Throw away your local changes and reload the latest version from the repository.

Advanced Actions

* **Branch**: Create a separate copy of a file/folder for private use (bug fixing, testing, etc). Branch is both a verb (“branch the code”) and a noun (“Which branch is it in?”).
* **Diff/Change/Delta**: Finding the differences between two files. Useful for seeing what changed between revisions.
* **Merge (or patch)**: Apply the changes from one file to another, to bring it up-to-date. For example, you can merge features from one branch into another.
* **Conflict**: When pending changes to a file contradict each other (both changes cannot be applied).
* **Resolve**: Fixing the changes that contradict each other and checking in the correct version.
* **Locking**: Taking control of a file so nobody else can edit it until you unlock it. Some version control systems use this to avoid conflicts.
* **Breaking the lock**: Forcibly unlocking a file so you can edit it. It may be needed if someone locks a file and goes on vacation (or “calls in sick” the day Halo 3 comes out).
* **Check out for edit**: Checking out an “editable” version of a file. Some VCSes have editable files by default, others require an explicit command.

**Centralised vs Distributed VCS**

Centralized Version Control Systems (CVCSs) were developed to help the developers collaborate with each other. Such CVS have a single server that contains all the versioned files, and a number of clients that check out files from that central place. For many years, this has been the standard for version control.

However, this setup also has some serious downsides. The most obvious is the single point of failure that the centralized server represents. If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they’re working on. If the hard disk the central database is on becomes corrupted, and proper backups haven’t been kept, you lose absolutely everything – the entire history of the project except whatever single snapshots people happen to have on their local machines.

This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS such as Git, clients don’t just check out the latest snapshot of the files: they fully mirror the repository. Thus if any server dies, and these systems were collaborating via it, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data.

Furthermore, many of these systems deal pretty well with having several remote repositories they can work with, so you can collaborate with different groups of people in different ways simultaneously within the same project. This allows you to set up several types of workflows that aren’t possible in centralized systems, such as hierarchical models.

**Version Control System-GIT**

**Git** is a widely used version control system for software development. It is a **distributed revision control system** with an emphasis on speed, data integrity, and support for distributed, non-linear workflows. Git was initially designed and developed by Linus Torvalds for Linux kernel development in 2005. Every Git working directory is a full-fledged repository with complete history and full version-tracking capabilities, independent of network access or a central server.

**Characteristics**

**Strong support for non-linear development**

Git supports rapid branching and merging, and includes specific tools for visualizing and navigating a non-linear development history. A core assumption in Git is that a change will be merged more often than it is written, as it is passed around various reviewers. Branches in Git are very lightweight: A branch in Git is only a reference to a single commit. With its parental commits, the full branch structure can be constructed.

**Distributed development**

Git gives each developer a local copy of the entire development history, and changes are copied from one such repository to another. These changes are imported as additional development branches, and can be merged in the same way as a locally developed branch.

**Compatibility with existing systems/protocols**

Repositories can be published via HTTP, FTP, or a Git protocol over either a plain socket, or ssh. Git also has a CVS server emulation, which enables the use of existing CVS clients and IDE plugins to access Git repositories. Subversion and svk repositories can be used directly with git-svn.

**Efficient handling of large projects**

Torvalds has described Git as being very fast and scalable, and performance tests done by Mozilla showed it was an order of magnitude faster than some version-control systems, and fetching version history from a locally stored repository can be one hundred times faster than fetching it from the remote server.

**Cryptographic authentication of history**

The Git history is stored in such a way that the ID of a particular version (a *commit* in Git terms) depends upon the complete development history leading up to that commit. Once it is published, it is not possible to change the old versions without it being noticed.

**Toolkit-based design**

Git was designed as a set of programs written in C, and a number of shell scripts that provide wrappers around those programs. Although most of those scripts have since been rewritten in C for speed and portability, the design remains, and it is easy to chain the components together.

**Pluggable merge strategies**

As part of its toolkit design, Git has a well-defined model of an incomplete merge, and it has multiple algorithms for completing it, culminating in telling the user that it is unable to complete the merge automatically and that manual editing is required.

**Garbage accumulates unless collected**

Aborting operations or backing out changes will leave useless dangling objects in the database. These are generally a small fraction of the continuously growing history of wanted objects. Git will automatically perform garbage collection when enough loose objects have been created in the repository. Garbage collection can be called explicitly using git gc –prune

Biblography

* 1. [www.wikipedia.com](http://www.wikipedia.com)
  2. http://betterexplained.com/articles/a-visual-guide-to-version-control/