

Version control systems and git

Understanding Version Control Systems and Learning Git in Detail



August 3, 2018

Elisheva strauss

Maalot Capstone Senior Thesis

Table of Contents

[Intro to Version Control Systems 2](#_Toc521070951)

[What it is 2](#_Toc521070952)

[Types 2](#_Toc521070953)

[Local 2](#_Toc521070954)

[Centralized 3](#_Toc521070955)

[Distributed 4](#_Toc521070956)

[Comparing Version Control Software and How To Use Them 5](#_Toc521070957)

[Local 5](#_Toc521070958)

[Centralized 5](#_Toc521070959)

[Distributed 6](#_Toc521070960)

[An Example of a Popular VC: Git and GitHub in Detail 8](#_Toc521070961)

[Installing Git 8](#_Toc521070962)

[Creating repositories 8](#_Toc521070963)

[Understanding the Different Statuses of the Repository 11](#_Toc521070964)

[Adding and Committing 12](#_Toc521070965)

[Comparing Files 15](#_Toc521070966)

[Reverting 16](#_Toc521070967)

[Cherry-picking 18](#_Toc521070968)

[Rebasing 18](#_Toc521070969)

[Branching 19](#_Toc521070970)

[Merging 20](#_Toc521070971)

[Deleting branches 21](#_Toc521070972)

[Remotes 21](#_Toc521070973)

[Using GitHub 23](#_Toc521070974)

[Pull Requests 23](#_Toc521070975)

[Forking 24](#_Toc521070976)

[Full Version Control Plan 24](#_Toc521070977)

[Testing 24](#_Toc521070978)

[Conclusion 25](#_Toc521070979)

[Glossary 26](#_Toc521070980)

[Bibliography 27](#_Toc521070981)

# Intro to Version Control Systems

## What it is

Version control systems regulate changes so that developers can keep track of their own work and collaborate with others on the same project.

It allows developers to know who made what changes and when, so that everything is organized and controlled. Another benefit is that developers can go back and redo a project or file from any point in ‘history’. Additionally, version control systems allow developers to compare different versions and not override other developer’s code too easily. [[1]](#footnote-1)

## Types

### Local

On every PC there is a directory where the files are saved every so often. This is not ideal to use for big projects, because it is only available on one computer, and it is very easy to make mistakes.[[2]](#footnote-2)

### Centralized



From Pro Git

Centralized version control systems have one central server with all the versions saved on it. Each developer gets the versions that they need from the main repository. This is called ‘checking-out’ the version. After the developer finishes updating the code, he send it back to the main repository. This is called ‘checking-in.’[[3]](#footnote-3)

Advantages of a centralized system are that there is communication between developers. Everyone can see what others are doing because there is one centralized server. Additionally, administrators have more control over the project. All these benefits lead to an easier system to use than a local system.

One disadvantage is that if the central server goes down or is compromised, all data could potentially be lost. [[4]](#footnote-4) This can be fixed if the server is backed up regularly in either the cloud or a different server in a different location.

### Distributed



From Pro Git

While using distributed version control systems, developers download all the versions (the entire history) when they first get the repository onto their local machine. To help developers work together, there is usually a central server that all developers can access. That central server is the version kept most up to date and is where all the developers send their code when they have updated it. Each developer can work locally, make changes, and test their work, and then just send it to the central server for the other developers to access.[[5]](#footnote-5)

There are many advantages to using a distributed system. Firstly, they are much faster to use because everything is stored on the developer’s machine. Secondly, developers don’t need to use the internet or be connected to any network.[[6]](#footnote-6) Additionally, if a server goes down, developers can copy their own code back to the server because each developer has all the code. Lastly, developers can connect to more than one repository at once.[[7]](#footnote-7)

There are not many disadvantages to using a distributed system over centralized or local but, in some situations, distributed systems can take up more space and checking out and checking in can be more complex.

In summary, local systems have one copy on one machine, centralized systems have one copy that can be accessed from many machines, and distributed systems have many copies on many machines.[[8]](#footnote-8)

# Comparing Version Control Software and How To Use Them

## Local

The standard local version control system is RCS. RCS saves just patches so if a developer wants a specific version the computer combines the patches.[[9]](#footnote-9) This takes up less space than saving the entire file version but can take longer because the computer needs to combine all the patches to produce the desired version. To improve efficiency, the computer occasionally saves a file with all the patches and continues saving from that version.

## Centralized

Centralized version control system are like RCS in the way that they store data. They are file based and therefore save each file change using patches. In centralized systems, developers only use:

**<version system> add**

to add a file the first time. After that, they just commit their changes[[10]](#footnote-10)

**<version system> commit**

This is because the files are saved once, when they are ‘added’, and only patches are saved after that by ‘committing.’

#### CVS

CVS was one of the first centralized version control systems. Using CVS, developers can ‘rollback’ and completely delete past commits.[[11]](#footnote-11) This is beneficial if a change was made that needs to be undone, but is much more hazardous than other systems that do not completely rollback, but keep the reverted changes in the history. Using CVS, once a developer rolls-back, he can never get back that version. It is removed from the history.

One con of CVS is that it does not allow easy storage of any type of file and metadata[[12]](#footnote-12)

Subversion

Subversion, also known as SVN, is very commonly used today. One drawback of SVN is what was mentioned earlier, that a developer can’t ‘rollback’ meaning he can change code back but it will always be in the history.[[13]](#footnote-13)

## Distributed

Git

Git has become very popular over the last few years. One of its many benefits is that it is open-sourced which means it is free and constantly updated. Another benefit is that Git never deletes history, it is unique in that it uses pointers and moves the pointers to a new place.

Git takes snapshots of the repository which saves the entire repository, not just patches. This is done at every commit. ‘git add’ is done whenever developers make changes, in addition to being used to add a file to the repository to start controlling it. In most other systems, the add command is used just for adding files to the repository and only patches are saved after that, using the commit command. Unlike in other systems, the command add in Git means that the developer wants a new change to be included in the next commit/ snapshot. ‘git commit’ produces new snapshots with different hash codes for each new snapshot. Using Git, developers can compare different versions of the same file because they are comparing snapshots.

Lastly, when Git sends files, it stores the files as changes, deltas, not as whole files. This minimizes the content being sent over networks thereby maximizing speed and efficiency.[[14]](#footnote-14)

#### Mercurial

Mercurial is also an open source software. One advantage of Mercurial, is that it allows developers to add their own features. They can write shell scripts that will be read by the command line and make their own commands that will be run by the command line as well (called aliases).

One disadvantage of mercurial is that it does not allow the developer to redo history. It only allows rolling back one commit.[[15]](#footnote-15)

# An Example of a Popular VC: Git and GitHub in Detail[[16]](#footnote-16)

## Installing Git

Git can be installed in a few ways, I will be using the command prompt which can be accessed in all three installation methods.

Developers have various options to use. They can use Git Bash, a version of the command prompt with git already preconfigured, Git, the actual source code which can be used from the command prompt, or GitHub Desktop, a graphical interface. Github Desktop works with the local repository to track changes and differences between the local repository and the repository stored on GitHub. It is easy to use and comes preconfigured with a command prompt that has git installed as well.

When the developer has a command prompt with git installed open, he should set up his name and email by using the following,

**git config--global user.name "your name"**

**git config--global user.email "your email"**

## Creating repositories

Once the developer has git installed, he should start keeping track of files. He can either clone an existing repository or create a new one.

#### Clone an existing repository

GitHub is a website that stores repositories, allows the developers to do all git functions, and much more. Developers use GitHub like a central repository, in that it is the most up to date branch. All developers can access it and update their local repositories from it when necessary. They also merge their own updates to the repository stored on GitHub.

To clone an existing repository, the developer should open the command prompt and move to the directory where he wants his local repository to exist.

**cd <local/repository/URL>**

He clones the repository by

**git clone <URL**>

He should get back something like the following:

**Cloning into '<Name of Repository>'...  
remote:  Counting objects:  97, done.  
remote:  Compressing objects:  100 % (10 / 10), done.  
remote:  Total 97 (delta 1), reused 0 (delta 0), pack - reused 87  
 Unpacking objects:  100 % (97 / 97), done.**

#### Create a new repository

To create a new repository instead of cloning, the developer should open the command prompt and move to the directory where his new repository is stored. (As a reminder he needs to use the “cd” prompt.)

Then he should use git init to initialize the repository as a git-controlled repository. This will produce a .git folder within his repository.

**git init**

Add and commit all the existing files to his repository. See adding and committing later. (git add . and git commit). Now he should have a version-controlled repository on his local computer.

To clone the new repository to GitHub, he should open the command prompt and cd into his local repository.

In the command prompt, he should add the GitHub remote repository as a remote repository and name it. In this example origin is the name of the new remote repository. Developers usually use origin as the name of the remote repository.

**git remote add origin git@ github.com: < username > /<repository name>.git**

Then he should push his current repository (here master) to the remote repository (called origin)

**git push origin master**

The developer will be asked for his username and password for his GitHub account. Keep in mind that when entering the password, the command prompt will appear as if nothing is being entered. The command prompt will look very similar to this:

**fatal:  HttpRequestException encountered.  
  An error occurred**

while**sending the request.  
Username**

for**'https://github.com': < username >   
Password**

for**'https://<username>@github.com':   
Counting objects:  53,  done.  
Delta compression using up to 4 threads.  
Compressing objects:  100 %  (51 / 51),  done.  
Writing objects:  100 %  (53 / 53),  55.79  KiB  |  5.58  MiB / s,  done.  
Total  53 (delta 2), reused 0 (delta 0)  
 remote:  Resolving deltas:  100 %  (2 / 2),  done.  
To https: //github.com/elishevastrauss1/SchoolWork.git  
\* [**new**branch]      master -> master**

As was mentioned earlier, when cloning and passing files, git compresses them by saving the changes (called Deltas) to maximize speed. As shown in the previous lines that are reiterated below.

**Delta compression using up to 4 threads.  
Compressing objects:  100 %  (51 / 51),  done.**

## Understanding the Different Statuses of the Repository

The working directory refers to the files currently in the workspace. These can be altered and saved.

The index, also known as the staging area, refers to the files that were added to the next commit.

The local repository is on the local computer. When the developer adds, commits, reverts, etc. these commands are only done locally to that developer’s local repository.

The remote repository is usually in a different location, like github. After developers commit to their local repository, they can push their changes to the remote repository so that the changes can be viewed and accessed by other developers.

## Adding and Committing



(Steele)

There are many steps to change and save changes to a git repository.

First the developer should find out what is the status of each file in his repository to find out what was changed.

**git status**

The output will be:

**On branch master  
 Your branch is up to date**with**'origin/master'.  
  
Changes  not staged**

for**commit:    (use  "git add <file>..."**

**to update what will be committed)  
  (use  "git checkout -- <file>..."**

**to discard changes**in**working directory)  
  
         modified:    < file name >**

The first two lines are saying that the local repository, master is “up to date” with the remote repository, origin. If the other developers changed the remote repository, the developer will have to update his local repository before trying to save his commits back to the remote. (See updating the local repository.)

The rest of the lines tell which files in the working directory have changed since the last commit. They are not staged so will not be included in a commit unless they are added to the staging area before committing.

The next step is that the developer needs to add the changed files that he wants to commit to the staging area to be committed.

**git add <file name>**

or

**git add .**

or

**git add --p**

‘git add’ is called staging a commit. Those files are now staged and ready for committing. They have not been saved to the local repository and the developer can still change the files. ‘git add <file name>’ stages just one file, while ‘git add .’ stages all changed files to be committed. ‘git add –p’ will allow him to choose which items to commit in a given file. This allows just parts of files to be committed.

Now there will be an additional section when the developer does ‘git status’

**Changes to be committed:**

**(use "git reset HEAD <file>..." to unstage)**

**modified: < file name>**

These lines show which files are in the staging area, meaning they are ready to be committed.

The developer should then commit the changes to his local repository.

**git commit**

The developer can add all files and commit at the same time by

**git commit -a**

A message editor will pop-up to write a detailed message about the commit. He should then save the message and close the editor.

Another option would be to just add the message after the commit command by saying

**git commit -m "<message>"**

This will not open an editor and will just continue as if the editor was opened, saved, and closed.

Once the editor is closed, the command prompt will display something like:

**[master(root - commit)  < Commit Log Number > ]  < Your Message >   
40 files changed,  1136 insertions(+)  
 create mode 100644 .classpath  
 create mode 100644 .project  
 create mode 100644 .settings / org.eclipuse.jdt.core.prefs  
 create mode 100644  MusicLessons.html  
 create mode 100644  MusicNotes.jpg  
 create mode 100644  StudentLists.txt  
 create mode 100644 bin / dataStructures / LinkList.**class

## Comparing Files

To compare files, use the ‘git diff’ command. Pressing the letter ‘q’ will exit the diff.

‘git diff’ can be used in many ways. It can be used to see changes to the working directory by:

**git diff <file name>**

The file name is optional and if it’s not included, all differences will be output.

To output differences between the staging area and the local repository, meaning show the files that are ready to be committed only:

**git diff --cached**

To compare branches:

**git diff <branch> <other-branch>**

#### HEAD

HEAD is the pointer to the current branch. When the developer commits, the HEAD pointer moves forward in history along with the current branch. HEAD~<number> refers to the commit that is the specified number away from the current HEAD.

To compare different versions, use the HEAD pointer.

To compare the latest commit to the one before it:

**git diff HEAD^1 HEAD**

## Reverting

Many times developers want to undo something they’ve done, either in the working directory or already committed. Git allows for easy fixing of mistakes.

To undo changes that were not yet committed to the local repository, the developer can just checkout the specified file from the repository.

**git checkout < file name >**

This will get rid of all the working directory changes which were not committed. The developer should be careful, because there is no way to get those changes back.

In many situations, developers need to change back to a previous version in history. To do this, developers need to first find out what the hash of the specific snapshot is. They can find out by viewing the history with the commands:

**git log**

or

**git reflog**

Both commands display the previous commits and their hash codes. ‘git reflog’ displays all actions done including switching branches, merging, resetting, etc. It doesn’t just show commits. ‘git log’ shows the commits in greater detail, including the entire hashcode, author, date, and message. Developers should have written thorough messages which describe the changes made, so that the bug is easy to find.

To revert, developers need to copy the log number (hash code) and then

**git revert < log number >**

Only the first eight numbers of the hash should be used. This will make a new commit that does the opposite of the specified commit. In the diff, this means that all the ‘+’ become ‘–‘ and visa-versa. This only reverts that specific commit.

Developers can also reset back to a specific commit. This will make a new commit that reverts all the commits until the specified log number.

To unstage a file that was already added to the pending list:

**git reset HEAD < file name >**

This resets the HEAD pointer back to the last commit.

To undo the last commit and all changes since then:

**git reset HEAD^**

There are two options when resetting; keep the changes in the working directory after the reset, or completely delete them.

**git reset --soft < log number >**

This undoes the changes and saves them in the working directory.

**git reset --hard < log number >**

This undoes the changes and does not save them in the working directory.

The most basic reset would be to reset the working directory back to the state it was in after the last commit. This is done without the log number.

**git reset –hard**

## Cherry-picking

Cherry-picking is applying a specific commit to a branch.

**git cherry-pick <hash of commit>**

This is commonly done when a commit was mistakenly committed to the wrong branch. In that case, the developer would simply revert the commit in the mistaken branch, and then cherry-pick it into the correct one.

## Rebasing

Rebase resets the base of the branch so that if the master continued while the developer was working on a branch, he would rebase the branch to make it look like his branch merge came after the other fixes. For example, a developer branches off of the master and does some work while the master was updated. He may want to add his commit to master right after the updated commit and not have to worry about merging in the new changes. He wants to commit his changes straight after the master changes.

This can be accomplished by resetting the base of his commit to the updated master.

**git rebase origin/master**

on the branch that the developer wants to reset and it will make the base be the updated origin/master.

## Branching

Branches are copies of the repository that can be merged into other branches. There are many benefits to branching. They allow developers to do work that will not affect the master branch or any other work that is being done. Additionally, branches allow developers to do specific tasks separately. This leads to clarity in the code, in the history log, and clarity between developers.

When switching between branches, the developer’s working directory is completely changed to be identical to the specified branch.

To switch between branches, the developer first must determine what branches exist:

**git branch**

git will put a star next to the current branch.

To create a new branch:

**git branch <new branch name>**

To checkout a specific branch that was already created in the working directory:

**git checkout <branch>**

To make a new branch and switch to it at the same time:

**git checkout –b <new branch name branch>**

This is equivalent to ‘git branch <new branch name>’ and ‘git checkout <branch>’ together.

When switching branches, the developer’s working directory can’t have any changes that were not added or committed, meaning no pending changes or unstaged changes. To fix this problem, the developer can use the command

**git stash**

which will clean out the working directory and save his unstaged changes.

To get back previously stashed changes he should first use

**git stash show**

which will give a list of the branches and files that have stashed changes. Then

**git stash apply**

which will reapply stashed changes.

## Merging

Merging is adding changes from one branch to another branch.

To do so, the developer first checks-out the branch he wants to change.

**git merge <branch>**

will merge the branch specified into the current branch.

Conflicts can occur when trying to merge two files that changed the same area of code. The developer will be warned that there is a conflict and that the file can’t be completely merged. He must go to the file and resolve the conflict, which will be found in between “merge conflict markers” (<<<<< and >>>>>). The developer can either open a merge tool by

**git mergetool**

which will help him merge the conflict. To do this, the mergetool must already be preconfigured. He can also manually go to the file and resolve the conflict by hand.

## Deleting branches

If the branch was already merged and the developer wants to delete it:

**git branch –d <branch name>**

If the branch was not merged and the developer wants to forcibly delete it anyway. This can be because the branch is no longer needed or because it failed:

**git branch –D <branch name>**

## Remotes

A remote is a word that simply means the URL of another server with the specified git repository on it.

When the developer clones, he gets two pointers, master and origin. ‘master’ is the main local working directory where all the updates get merged to. The ‘master’ branch is usually what gets deployed in production. ‘origin’ points to where the remote repository was last when the developer cloned or last updated it.

After a developer successfully updates the software and commits it to his local repository, he now needs to ‘push’ the commits back to the remote to update that repository as well.

There are two scenarios that can occur. Firstly, if the remote’s file has not been changed since the developer last updated, then the developer can go straight to pushing the commits remotely.

**git push <remote url or alias for the remote url > <branch that the developer is pushing to in the remote server>**

Usually the developer will be pushing his branch to the origin by the command

**git push origin <branch>**

This will update the remote repository’s specified branch and will move the ‘origin/master’ pointer to point to where the developer just committed.

When trying to push the commit to the remote, a second situation can occur. If the remote has changed since the developer’s last update, then git will not allow him to push his file changes because they will be overwriting other changes. Therefore, he always needs to first ‘fetch’ the updated repository

**git fetch**

Fetching only pulls the changes to the developer’s local repository, not the working directory. Then he needs to merge the updated repository file with what he wants to commit. Usually the command will be

**git merge origin/master**

because he is merging the updated repository changes into his current branch. Now that the remote’s updated version is in this files history, the developer can push to the remote.

Another option is to use the command, which fetches and then automatically merges with the remote repository to the current branch. This can get confusing sometimes, so it is usually better to first fetch and then merge manually.

**git pull**

## Using GitHub

Github is a place to store code and share it with others. Every github repository has at least one owner who has absolute control over the repository. Owners can make the repository public or private, delete the repository, invite collaborators, etc. Collaborators have less rights than owners, but they can still participate in a project by pushing to github, opening pull requests, and more.[[17]](#footnote-17)

### Pull Requests

When a member of a repository wants to make a change to the code, he pushes it to the github repository (as discussed in the Remotes section.) He then opens a ‘pull request’ from github. Pull requests are ways suggest changes and have someone else review them. The developer who opened the pull request will set a reviewer to review his code. The reviewer will get a notification that he has a pull request to review. If the reviewer approves, and he is an owner of the repository, he will approve the request and merge it. If he does not approve, he will either close the pull request, or he will comment on it. The developer who opened the pull request will get a notification whenever a change is done to his pull request.[[18]](#footnote-18)

### Forking

If someone is not an owner or collaborator, he cannot push or open a pull request in that repository; instead, he can fork the repository. A fork is a branch off the main code to produce an entirely new file which will not necessarily be merged back. It is now owned by that developer and does slight or big differences from the original.[[19]](#footnote-19) After making the fork and updating the changes, the developer can open a pull request from the forked project. To fork in github, the developer simply clicks on the ‘Fork’ button in the repository that he wants to fork.[[20]](#footnote-20)

# Full Version Control Plan

Besides for tracking a company’s own code, developers must keep in mind the code that they are dependent on. For example, a company that uses git must be aware of git software changes in versions aside from keeping track of their own code. They must update their dependent software and should try to make sure that they are never more than one version behind the current version of the software. This is because many companies stop updating their out of date software which allows for bugs and viruses to be introduced. A company should try to make sure to never use a deprecated version.[[21]](#footnote-21)

## Testing

If a new software or version is introduced to a company, there are a number of steps the company should follow. First, they should define the changes that will be implemented. Database changes should be given extra consideration.

After the changes are defined, the company should apply the changes in a testing environment. A testing environment is an environment where the code is the same as the production environment, but is not actually production. New features and software can be tested in this environment without affecting production.

After successfully deploying the changes to the test environment, some developers in the company should be trained to use the new software or version. They will test out the new code in the testing environment. The testers determine if there are any problems. If there is something wrong with the new code, the company will inform the producers. The company will meanwhile keep production running on the old software.

If the tests are successful and the new software works correctly in the test environment, the new software is released into production. The updated software in production is meticulously watched for a couple weeks. If all is still running smoothly and correctly, the change is accepted completely.[[22]](#footnote-22)

# Conclusion

In conclusion, version control systems can keep developers organized and prevent mistakes from causing severe damage. They help track code revisions, show history, enable multiple developers to work on the same code at once, support branching and merging of those branches, reveal conflicts, allow developers to fix those conflicts, and support going back in history. Version control systems are critical for developers.

# Glossary

Branches- divide from the trunk to attempt something new.[[23]](#footnote-23) Branching will completely change the working directory to be identical to the new branch.[[24]](#footnote-24)

Checkout- checkout a specific branch from the central repository and (git) to refresh a specific file’s code.

Commit-

1. In distributed systems: save new changes to original on the developer’s computer[[25]](#footnote-25)
2. In centralized systems: send changes to central repository[[26]](#footnote-26)

Fork- branch off the main code to produce an entirely new file. Now it is owned by that developer and does slight or big differences from the original.[[27]](#footnote-27)

Local repository- developer’s personal copy of the central repository.[[28]](#footnote-28)

Log- See who did what[[29]](#footnote-29)

Merge- merge working copy to central repository[[30]](#footnote-30)

Patch- A change done to one version that produces a new version.[[31]](#footnote-31)

Pull/ Update- update the current branch by ‘pulling’ from the shared central repository. Used when other developers made changes to central repository. Needs to be done before merging changes to central repository so that there is no conflicts.[[32]](#footnote-32) Does fetch and merge together.

Repository

* 1. A group of files that is under version control
  2. Other older version control systems it means the central code[[33]](#footnote-33), which there is no such thing in distributed systems.

Revert- changes to initial state and deletes pending changes[[34]](#footnote-34)

Status- shows new, changed, and deleted files[[35]](#footnote-35)

Trunk- main branch meaning the most up to date branch where all developers push to and pull from

Acronyms

CVS- Centralized Version System

PC- Personal Computer

RCS- Revision Control System

SVN- Subversion

VCS- Version Control System

# Bibliography

Allbery, Russ. *CVS*. n.d. <eyrie.org/~eagle/notes/CVS>.

Chellman, Joe. *Version Control for Everyone*. n.d. <Lynda.com>.

*CVS vs SVN*. n.d. <tartarus.org/~simon/cvs-vc-svn.html>.

*GitHub Contributing to a Project*. n.d. <https://git-scm.com/book/en/v2/GitHub-Contributing-to-a-Project>.

Mercer, Cole. *Version Control and Releasing*. n.d. <Lynda.com>.

*Mercurial vs Git Why Mercurial*. n.d. <https://www.atlassian.com/blog/software-teams/mercurial-vs-git-why-mercurial>.

*Permission Levels For a User Account Repository*. n.d. <https://help.github.com/articles/permission-levels-for-a-user-account-repository/>.

Rand-Hendriksen, Morten. *Mapping the Modern Web Design Process*. n.d. June 2018. <Lynda.com>.

Rosenbaum, Dr. Jerry. "Version Management -People Aspect." n.d.

*Soft Svn Vscvs*. n.d. <pushok.com/soft\_svn\_vscvs.php>.

Stansberry, Glen. *The Top 7 Open Source Version Control Systems*. September 2008. <smashingmagazine.com>.

Steele, Oliver. n.d. <osteele.com>.

Straub, Scott Chacon and Ben. *Pro Git*. Apress, n.d.

*SVN*. n.d. <abbeyworkshop.com/howto/misc/svn01>.

*SVN Tutorial*. n.d. June 2018. <tutorialspoint.com>.

*Version Control Systems*. n.d. 2018. <g2crowd.com/categories/version-control-systems>.

Vesperman, Jennifer. *CVS Intro*. 3 January 2002. 2018. <linuxdevcenter.com>.

1. (Straub) [↑](#footnote-ref-1)
2. (Straub) [↑](#footnote-ref-2)
3. (Ishai) [↑](#footnote-ref-3)
4. (Straub) [↑](#footnote-ref-4)
5. (Chacon) [↑](#footnote-ref-5)
6. (Chacon) [↑](#footnote-ref-6)
7. (Straub) [↑](#footnote-ref-7)
8. (Mercer) [↑](#footnote-ref-8)
9. (Straub) [↑](#footnote-ref-9)
10. (Chacon) [↑](#footnote-ref-10)
11. (Soft Svn Vscvs) [↑](#footnote-ref-11)
12. (Soft Svn Vscvs) [↑](#footnote-ref-12)
13. (Soft Svn Vscvs) [↑](#footnote-ref-13)
14. (Chacon) [↑](#footnote-ref-14)
15. (Mercurial vs Git Why Mercurial) [↑](#footnote-ref-15)
16. (Chacon) [↑](#footnote-ref-16)
17. (Permission Levels For a User Account Repository) [↑](#footnote-ref-17)
18. (GitHub Contributing to a Project) [↑](#footnote-ref-18)
19. (Mercer) [↑](#footnote-ref-19)
20. (GitHub Contributing to a Project) [↑](#footnote-ref-20)
21. (Rosenbaum) [↑](#footnote-ref-21)
22. (Rosenbaum) [↑](#footnote-ref-22)
23. (Chellman) [↑](#footnote-ref-23)
24. (Chacon) [↑](#footnote-ref-24)
25. (Mercer) [↑](#footnote-ref-25)
26. (Vesperman) [↑](#footnote-ref-26)
27. (Mercer) [↑](#footnote-ref-27)
28. (Mercer) [↑](#footnote-ref-28)
29. (SVN Tutorial) [↑](#footnote-ref-29)
30. (Mercer) [↑](#footnote-ref-30)
31. (Straub) [↑](#footnote-ref-31)
32. (Mercer) [↑](#footnote-ref-32)
33. (Chellman) [↑](#footnote-ref-33)
34. (SVN Tutorial) [↑](#footnote-ref-34)
35. (SVN) [↑](#footnote-ref-35)