# Table of Contents

1 [DRAFT] SVN vs GIT - Technical Aspects  
1.1 Licensing ................................................. 1  
1.2 Clients .................................................. 1  
1.3 Type ..................................................... 1  
1.4 Tree Model .............................................. 1  
1.5 Branching ............................................... 2  
1.6 Merge Process ......................................... 2  
1.7 Changes storage ....................................... 2  
1.8 Commit structure ...................................... 2  

2 [DRAFT] SVN x GIT - Practical Aspects  
2.1 Centralized VS Distributed ............................... 3  
2.2 Space Consumption ..................................... 5  
2.3 Repository structure ................................... 5  
2.4 Branches ............................................... 5  
2.5 Tags ..................................................... 5  
2.6 Security ................................................ 6  
2.7 Commits ............................................... 6  
2.8 Main workflow advantages ............................. 6  
2.9 Workflow examples [WIP] .............................. 6  

3 [DRAFT] Version Control Workflows  
3.1 Current Repository Structure .......................... 7  
3.2 Some attention points ................................ 7  
3.3 What to do? ............................................. 8  
3.4 What to do? ............................................. 8  
3.5 What to do? ............................................. 8  

4 Changes  
4.1 First suggestions ...................................... 9  
4.2 New branch structure .................................. 9  
4.3 References ............................................. 10  

5 References .................................................. 11  

6 Indices and tables ........................................ 13  

Bibliography ................................................ 15
1.1 Licensing

• Both are opensource

1.2 Clients

• SVN - Has the command line interface and some GUI clients like Tortoise SVN and Visual SVN.
• GIT - Also has the command line interface and lots of GUI clients like Git Extensions, Source Tree, GitHub, Git Kraken and Tortoise Git.

1.3 Type

• SVN - CVCS (Centralized Version Control System) - There is just one central repository where all operations are performed and from where every user gets the files. This allows a top-down access control and change locking features.
• GIT - DCVCS (Decentralized Version Control System) - Every user get a full copy of the repository on it's machine and, as a consequence, almost every operation is performed locally. There can be a central remote repository for collaboration to and from where the users will push and pull modifications. It also allows multiple remote repositories.

1.4 Tree Model

• SVN - Linear Tree (aka Stream) - SVN uses a linear tree as it’s data structure which means that every single commit will be represented as a node on that tree (or line). Every merge from other branches results on a single new commit on the tree like any other modification.
• GIT - DAG (Directed acyclic graph) [SINK1] [WIKI1] - The GIT tree has a Graph where every commit (graph node) has reference to, at least, one parent commit. This is specially helpful when you have branches and merges happening on your repository, this structure allows the user to find where each modification happened and in which branch it occurred. This structure maintains the whole commit history, even if some commits are not referenced by a branch or tag, they are still maintained on the RefLog and can be accessed.
1.5 Branching

- SVN - Branches on SVN are treated as normal folders on the repository. As you have only the central repository every branch is automatically remote and public (every user with access to the repository will have access to the branch) \(^1\). On the other hand, one can checkout the branch on a different folder on it’s file system independent of having the trunk of the repository checked-out. Another option is also start a branch on justa a subfolder of the repository. Branches on SVN are basically copies of the affected folders and this copy can be made locally and then commited to the central repo or remotly. On the remote copy it is made a Cheap Copy of the targeted files, this means that it is only a reference for the original files and not a actual file, this way the repository stay as lean as possible \(^2\).

  - “Subversion has no internal concept of a branch—it knows only how to make copies. When you copy a directory, the resultant directory is only a “branch” because you attach that meaning to it. You may think of the directory differently, or treat it differently, but to Subversion it’s just an ordinary directory that happens to carry some extra historical information... Subversion’s branches exist as normal filesystem directories in the repository.” [COLLINS1]

- GIT - Branches are pointers to specific commits on the repository. They are created locally and can be pushed/pulled to/from a remote repo or not.

1.6 Merge Process

- SVN
- GIT

1.7 Changes storage

- SVN - Deltas
- GIT - Files Snapshots

1.8 Commit structure

- SVN
- GIT

References

---

\(^1\) CONFIRM THIS INFO

\(^2\) Review text
2.1 Centralized VS Distributed

2.1.1 Centralized

- The repository is located on a remote server where all operations are performed.
- When the user checks out a directory from the repository it gets only the files, with minimal versioning information.
2.1.2 Distributed

- Every developer got a full copy of the repository stored locally where most of the operations are performed;
- The local repository can point to one or more remote repositories where the work can be shared with other developers.
2.2 Space Consumption

Knowing that when one developer clone a GIT repository it actually gets a full copy of the repository, a concern may arise in terms of local disk space consumption. Well, while you may have more data stored in your disk all the GIT database is compressed and highly optimized which means that the local repository will occupy less space in disk than one may think.

When you checkout a folder from a subversion repository you end up with 2 uncompressed versions of the folder locally, your working copy and the snapshot of the last revision, the one you checked out, of the folder. SVN does that to allow the developer to make diffs between the working copy and the last revision locally without needing to go to the server.

We can compare the two options to see which has the biggest size, that can be done importing an existing SVN repository to a GIT repository using the GIT-SVN tool and then compares sizes of the folders.

For this example it was considered the Phoenix repository and a common operation:

- Clone of the imported git repository;
- Checkout of the trunk from SVN.

_IDK if this part will be featured on the final presentation, I've found some problems with the repo import. I'll redoing the process to double check the data_ <<<EXAMPLE - PHOENIX REPO>>> 1.04 GB VS 781 MB

2.3 Repository structure

On subversion you have a tree model with just one branch where the revisions are stored sequentially.

GIT uses a Graph structure where each commit is a node that knows every of its parents. The commit is the central piece on the git repository.

2.4 Branches

For SVN branches are just another directory on the repository with its own history information. You can merge revisions between branches but they are still treated as separated entities. Because of the centralized nature of SVN all branches are remote and public. The administrator can restrict the access to the branch to a certain group of developers, but still, the branch is located on the server.

On git a branch is basically a named pointer for a specific commit on the tree, because of that nature branches are very cheap for the system. Branches can be only local or can be published to a remote repository for collaboration with other developers. Every new copy of the repository, cloned from a remote repository, is treated as a new local branch of the main repository, that means that every developer repository is an isolated entity and the work on that local repository does not interfere with other developers work until it is pushed to the shared repository and pulled by the others.

2.5 Tags

Tags and branches are very similar in both systems. For SVN a TAG is just another folder in the main repository, that shouldn’t receive any update and remain static, that can be achieved using access restrictions on the new tag.

On git a tag is a static pointer to a specific commit. Unlike the branch pointer, the tag pointer is static, which means that it cannot be moved to another commit.
2.6 Security

SVN offers a more granular security control, you can have different access rights for each folder on your repository, remember that tags and branches are also folders, which is perfect for a top-down management.

GIT security is repository-wise, you can allow or revoke access of a user to the whole repository only. You can apply read-only access to certain branches also, but for the whole repository, never to a single folder or file.

2.7 Commits

Every commit on SVN creates a new revision to the remote repository, that means that to be able to commit some changes to the code the developer need to have access to the main repository.

On git the commit is happening locally, so the developer does not need to have access to the remote. The work can be committed locally and then when the developer gains access to the remote repository, it can push all the commits at once.

2.8 Main workflow advantages

Some of the main advantages of the development workflow that git can provide are:

- Commit often - Due to the distributed version of git it makes possible for a developer to commit his work as often as he likes without risking breaking other developers code and maintaining a more detailed history of the work performed.

- Branching workflow - Since branches are lightweight and can be only local the developer may work on the code on a separate branch and then solve all the merge conflicts locally on a complete separated context from other developers code. Most of the git workflows take advantage of its branching model using separate and/or temporary branches for different kinds of work.

- Pull requests - Git offers the option to create a pull request between branches, this means that the work will only be merge on the destination branch if it gets approved. This feature helps a lot with code reviews and release of new versions of the application.

- Peer-To-Peer - Every clone of the repository is a full repository, so you can add any copy of the repository as a remote for your local copy. This allows developers to share branches (e.g.: Experimental Branches) without the need to push it to the server.

2.9 Workflow examples [WIP]

2.9.1 Feature Branch

2.9.2 GitFlow

2.9.3 Forking
3.1 Current Repository Structure

- **Trunk**
  - Dev workline - where almost every work is done
- **Stable**
  - QA testing branch - revisions that are made and tested on **Trunk** should be merged in this branch
  - Bug fixes related to the code located here should be fixed first on **Trunk** and then merged here.
- **Release**
  - Created by QA after a version on **Stable** is ready for release
  - Only hotfixes are allowed here
- **Private**
  - Temporary branch that should be used for work that can break the **Trunk** - e.g.: New Features, experimental work.

3.2 Some attention points

- Cherry picking merges became the norm.
- **Trunk** and **Stable** have independent life cycles but are maintained by the same users.
- Inconsistency between both branches history and code.
- Fixing **Stable** bugs on **Trunk** can lead to a inconsistent fix.
- A code that works on **Trunk** may not work on **Stable** as is.
3.3 What to do?

Improve or change the workflow?

#HSLIDE

3.4 What to do?

- Subversion?
- Git?

#HSLIDE

3.5 What to do?

Adapt and then migrate

or

Migrate and then adapt
4.1 First suggestions

- Change the “merge to Stable” strategy
- New development Branch (Trunk) based on Stable
- Keep Trunk clean
- Every new release to QA is a full merge os the development branch to the Stable branch
- Encourage the use of branches for experimentation or breaking updates
- Branch-When-Needed strategy vs Feature-Branch
- Development needs + QA needs
- Feature Toggles vs Feature-Branch for breaking features
- Development to Stable stabilization

4.2 New branch structure

- QA requires a Stable branch for the testing releases, so the Stabel branch remains as is since it is the “Production Ready” branch.
- Trunk is too unstable and messy to be used as a base development branch, so it should be put aside in favor of a new development branch that should come out of the Stable branch and should return to it on every and each new release as a whole, with no more cherrypick of commits by the developers.
- It is each developer responsibility to guarantee the stability of the development branch by all means available. E.g.: writing and/or running tests and making sure that every critical test pass, make sure that the solution can be built without any problem before every check-in.
- If a developer breaks the build it is his/her responsibility to make it work again.
- The merge meeting focus is to ensure that the new version is working and pass all the tests, and make sure that all the tickets related to this new version are still working on the new stable version.
- The merge from the development branch to the stable one is made by the end of the development cycle/week and the full branch is merged back into the stable.
- The developers should use any means one sees fit the goal of not break the stability of the development branch and not introduce any new bug or unfinished feature on the new stable build. E.g.: New temporary development branch (aka Feature Branch), feature toggles to exclude any unfinished features from the build process.
• In the case of the developer opting to use a separate branch for his/her development (aka Branch as needed strategy) he/she is the sole responsible for maintaining the branch in sync with the mainline (development branch) and to solve any kind of conflict (of content or tree) before merging back to the mainline.

4.3 References

4.3.1 Feature toggle

• https://martinfowler.com/bliki/FeatureToggle.html
• https://martinfowler.com/articles/feature-toggles.html
• https://www.pluralsight.com/courses/dotnet-featuretoggle-implementing

4.3.2 Feature Branch

• https://martinfowler.com/bliki/FeatureBranch.html
• http://svnbook.red-bean.com/en/1.7/svn.branchmerge.commonpatterns.html

4.3.3 Branch by abstraction

• https://martinfowler.com/bliki/BranchByAbstraction.html

4.3.4 Continuous integration

• https://martinfowler.com/articles/continuousIntegration.html
• Backblaze and Subversion, Time to Change?
• Fowler - Version Control Tools
• Fowler - Feature Branch
• StackOverflow - How is SVN v 1.8 branching / merging compared to Git?
• StackOverflow - git cherrypick and Datamodel Integrity
• git merge after git cherry-pick: avoiding duplicate commits
• Merge History, DAGs and Darcs
• StackOverflow - how and or why is merging in git better than in SVN
• Git Wiki - Git SVN Comparison
• SoftwareSwirl - Git, Mercurial, and Bazaar—simplicity through inflexibility
• BoxySystems Inc. - 5 Fundamental differences between GIT & SVN
• WANDISCO BLOG - Subversion Best Practices: Repository Structure
• Git Best Practices
• Ben Collins-Sussman, Brian W. Fitzpatrick & C. Michael Pilato - Version Control with Subversion
• StackOverflow - What Git branching models work for you?
• Atlassian - GIT resources
• BeanStalck - Using Branches in Subversion
• Joanne Carr - Subversion workflow guide
• Beanstalk - Developing and Deploying with Branches
• BeanStalk - Guides
• Gitflow Cheatsheet
• Vincent Driessen - A successful Git branching model
• Michel Ernest - Version control concepts and best practices
• MS Developer Blog - What is a Cherry Pick Merge, and why do you recommend against them?
Indices and tables

- genindex
- modindex
- search
[SINK1] Eric Sink - Merge History, DAGs and Darcs
[WIKI1] Wikipedia - Directed acyclic graph