Science and Technology Facilities Council

Technical Report
DL-TR-2020-003

Software Outlook Version Control Systems: overview and best practices

B Mummery

October 2020

Enquiries concerning this report should be addressed to:

Chadwick Library
STFC Daresbury Laboratory
Sci-Tech Daresbury
Keckwick Lane
Warrington
WA4 4AD

Tel: +44(0)1925 603397
Fax: +44(0)1925 603779
email: librarydl@stfc.ac.uk

Science and Technology Facilities Council reports are available online at:
https://epubs.stfc.ac.uk

ISSN 1362-0207

Software Outlook

VERSION CONTROL SYSTEMS

Overview and Best Practices

Dr Benjamin Mummery
STFC Hartree Centre
8-1-2020

Contents

1 Why Use Version Control?

Version control can and should be a part of every developer's toolkit. Version Control Systems (VCS) automate the perennial tasks of keeping track of changes both big and small, and maintaining the ability to undo those changes when necessary, while also providing the ability to centrally manage and synchronise multiple parallel development tasks between separate people, groups and goals. If you've ever had to search your hard drive for that file you're *pretty sure* is called "working_version_final.c" because "working_version__final_final_latest.c" turns out to be less functional than the name would suggest and *final.c* was the last time you used it for this specific task and it's *probably* in one of these subfolders **somewhere**...then you should be using a VCS. If you have ever emailed your code to a colleague (or even worse, to yourself), then you should be using a VCS.

Using a VCS means that the complete history of changes for every file, including what change was made, when, by whom, and for what reason, is retained for every alteration. This means that the VCS allows you to easily keep track of and manage updates to a project over time, and provides the ability to quickly and easily turn back the clock, compare versions, and fix mistakes with a minimum of disruption. Multiple developers can work on the project simultaneously safe in the knowledge that conflicts are easy to spot and resolve, and that the nature, purpose, and content of their updates are clearly evident to the rest of the team. In other words, using VCS makes it easier to collaborate on your project, while protecting your work from both major disasters, minor mistakes, and human absent-mindedness.

In addition to this, all major VCSs provide the ability to *branch* and *merge*. Separate copies of the repository can be created to handle multiple streams of work without worrying about conflicts. Once branches are ready to be merged, they can easily be examined for conflicts and resolved.

There are also several secondary advantages. While VCSs explicitly should *not* be used as a backup system, it does contain a copy of your source code, and can therefore provide a backstop when disaster strikes - even if you lose *everything* else, that copy will still exist. Furthermore, the repository can be accessed remotely, and thus provides not only a shared space for both remote and local collaborative contributions, but also a convenient way to share your code. Many VCSs can also integrate with project management and bug-tracking software such as Jira, providing a convenient workflow for bug and issue fixing.

2 What is it and How Does it Work?

At its heart, a Version Control System (VCS) is a database of changes made to one or more file. The basic element of version control is the *commit*. A commit is a bundle of edits, accompanied by a message from the developer explaining their nature and purpose. The VCS will tag commits by date and user, meaning that it is completely transparent who has done what, when, and why.

By and large, any VCS will provide the following features:

- **Version Control** - Store and organise all versions of your files.
- **Configuration Management** - Associate version of each file with appropriate versions of other files.
- **Concurrency** - Allow multiple people to work on the same files, toward a common goal or release.
- **Branching** - Allow groups of people to work on substantially the same files, but each group towards its own goal or release.
- **Release Management** - Recover, at any time, a coherent configuration of file versions that correspond to some goal or release, either for investigation or extension like bug fixing

3 How and When to Commit

As commits are the basic building block of VCS usage, getting into good habits is essential. What those habits actually *are*, however, depends to an extent on context.

As a general rule, good commits are:

- small
- frequent
- complete
- testable
- semantic

To understand why, let's run through these requirements and examine the effect they have.

3.1.1 Small

The smaller your commit, the easier its changes are to understand, and the more granular the change history will be. This makes it significantly easier to trace problems and resolve conflicts.

3.1.2 Frequent

This naturally occurs as a side effect of keeping commits small, but also means that other contributors have an up-to-date copy of your work. This makes it easier to avoid merge conflicts and prevent duplication of effort.

3.1.3 Complete

Committing half-finished code is an easy way to create confusion and discord. It means that when somebody is tracing a problem there are multiple commits for one section of code, possibly even with later commits modifying changes made in earlier commits. The code you commit should be the final version (at least as far as you are concerned at that point in development).

3.1.4 Testable
If it isn't tested, it doesn't work. Arguably, this should be seen as part of **complete**, since no responsible developer would ever consider their code to be complete without thoroughly testing it. However, since we are all imperfect humans who don't always act responsibly, it's worth making the point. Additionally, considering testing to be a requirement aids in breaking down large tasks in order to keep the commits small: the answer to the question "How do I decide what constitutes a suitable sub-task?" is "Find the smallest chunk of work that can produce a testable output."

It is also worth mentioning that some UI extensions to VCSs such as GitLab include tools to automatically run a testing pipeline for each new commit. This allows less pressure to be placed on the pre-commit testing, but cannot eliminate it entirely, especially since these pipelines typically need the code to be in the repository before testing. It does, however, make it much easier to routinely test the full project in the context of each new commit, allowing you to catch potential problems much earlier.

3.1.5 Semantic
This is arguably the most important rule. All the edits in a commit should be in service of one goal - be that adding a feature, fixing a bug, or making a specific functional change. This is essential for keeping the change history clear and understandable, and making it easy to roll back when needed. If your commit fixes two bugs, it should be two commits.

3.2 When to Ignore these Rules
The requirements given above are not absolute, and there are circumstances in which actually applying them all will be overly restrictive, and actually impede your workflow. As an example, the **complete** requirement can be too restrictive in certain situations - you may wish to share work-in-progress to ask for assistance, or you could want to "clean" your head to help clarify your current status before beginning a new burst of dev work. The benefits of commit properties like **complete** and **testable** are most prominent when the commits affect large groups, so when considering an individual's local repository (if using a distributed VCS), or a branch worked on exclusively by small groups (~3 people is a good upper limit), they can usually be safely dropped. It is also worth noting that "Tested" van mean very different things in different contexts. Depending on the pipeline you are using, testing can be anything from unit-testing every single function, to "does this thing print what I expect?".

As working with larger groups inevitably leads to more difficult communication, and less flexibility, these requirements grow more and more important as the number of people who may need to deal directly with your commit grows. As such, it is wise to use squash merges to condense partially completed commits into a single complete, testable, and semantic package when moving your work onto the next stage towards deployment (as long as proper merge etiquette is followed).

4 How and When to Branch

Branches are a VCS staple, and an incredibly useful tool. A new branch is a new copy of the repository, complete with the full change history up to the point where it was branched, but independent of the original. This allows a developer, or team of developers, to make changes freely without running the risk of conflicts. As such, they can try new ideas, test out large-scale changes, or just develop a complex feature in a self-contained sandbox where they will not interfere with other workflows. Crucially, branches can be merged once their content is stable and tested.

For an illustration of why branching is useful, consider the following. You have completed, tested, and deployed *Feature X* before beginning work on *Feature Y*. You are making regular commits (all of which are small, frequent, complete, tested, and semantic as we discussed above of course) when you discover a problem with *Feature X* that needs an immediate fix. You will need to track down the root cause of the problem, which means you need to roll back your work on *Feature Y*, fix the problem, then re-introduce your changes. All of this is a recipe for losing work and creating even more bugs.

Using a separate branch to develop *Feature Y* means that you can commit to your heart's content without ever worrying about *Feature X*. You can create a new branch from the *Feature X* deployment, find and fix the problem, then merge those changes. This gives you less confusion, a cleaner file history, and less additional effort.

4.1 Branching best practices

4.1.1 Stage Branches

It is common practice when branching to have a "Main" or "Trunk" branch. This is the default working branch of your repository, and the core of the branching system. In most cases, it is useful to have at least two additional branches. These three branches together can be considered as "Development", "Release Candidate" and "Release".

The Development branch is where new features and bugfixes are created, i.e. where all active development takes place. As such, it is inherently unstable and transitive - new changes are occurring all the time, and conflicts and bugs are likely to arise.

Once a required set of features and bug fixes are complete and tested in isolation, the Development branch can be merged to the Release Candidate branch. This branch is where the "complete" version of the project can be tested in its entirety, ideally by deploying it to test versions of the environment(s) in which it will ultimately be used. Doing this on a separate branch means that it is independent of ongoing development - testing and bugfixing of the Release Candidate can occur in parallel with ongoing feature development.

Finally, when the Release Candidate has been tested to satisfaction, it can be merged to the Release branch. This keeps the history of Release as the history of released versions, makes it trivial to refer back to previous version as and when issues may arise, and gives you a specific location from which to source deployments, separate from active development or testing.

Keeping at least these three branches and maintaining the principal of separating development from testing from deployment leads to well organised, easily understandable repositories.

Branching also provides the ability to maintain separate branches for separate platforms - rather than try to write one version of your project that works on anything, you can keep the core code platform agnostic and specialise in independent branches without worrying about conflicts between, for example, the Windows 10 1909 version and the Android 7.1.2 version.

4.1.2 Create Feature and Hotfix Branches for New Feature or Hotfix
Feature branches should be created when you begin any non-trivial piece of work. This might be a new feature, a bug fix, or an experimental idea. Doing this provides a sandbox in which to develop, separate from the rest of your repository. Once your feature or bugfix is complete, the branch can be merged back into the main development branch from which it was created.

4.1.3 Use the Minimum Number of Branches
New branches should be created any time you begin non-trivial work, or for new staging or environments. Consequently, branches can propagate rapidly during active development. Fewer branches makes for fewer merges, fewer conflicts, and less confusion. Branches should be merged and deleted as soon as is feasible. For example, Short-lived feature branches are an excellent way of gaining the benefits of branching without confusion.

4.1.4 Merge Regularly
Merge conflicts tend to be the result of parallel development causing branches to diverge too far. Where possible, branches should be regularly merged (but not deleted), ensuring a degree of parity while still maintaining the separate sandboxes.

4.1.5 Maintain Clear Demarcation
As each branch should have a specific purpose, so the areas of the repository that they will be changing should be clearly understood. In an ideal world, this would mean that parallel development can continue merrily and simply be slotted together at the end, secure in the knowledge that everything should still work as expected. In practice, however, there are frequently unexpected issues that require modification of the repository outside the envisioned scope of the branch. In these cases, where shared components are being changed, it is essential that other developers and/or teams be made aware. Regular merges can help with this, but more useful is a clear understanding of what is, and is not, expected to be modified by each line of development.

4.2 Example branching scheme
For most developments, it is possible to define five distinct types of branch:

4.2.1 Release Branch
- A series of tagged releases. The "Public Facing" aspect of the repo. Deployments source from this branch exclusively.
- Merges from **Release Candidate Branch**.
- **Hotfix Branches** can be created from, and merged to, this branch when necessary.
- No commits should be made to this branch.
- This branch should not be deleted.

4.2.2 Release Candidate Branch
- A series of candidates for release. Used as a staging area for testing potential releases.
- Merge to **Release Branch** once tests and hotfixes are complete.
- Merge to **Development Branch** regularly while hotfixes are being performed.
- **Hotfix Branches** can be created from, and merged to, this branch.
- No **Feature Branches** should be created from, or merged to, this branch.

- Commits should be made to this branch only in the case of trivial bugfixes. If in doubt, create a **Hotfix Branch**.
- This branch should not be deleted.

4.2.3 Development Branch

- The main branch for active development. Collects all features and any bugfixes for those features.
- Merge to **Release Candidate Branch** when a pre-agreed set of features have been completed.
- **Hotfix** and **Feature Branches** can be created from, and merged to, this branch.
- Commits should be made to this branch only in the case of trivial bugfixes. If in doubt, create a **Hotfix Branch**.
- This branch should not be deleted.

4.2.4 Feature Branch

- Branch for developing a specific feature.
- Created from **Development Branch**, or an encompassing **Feature Branch**, whenever work on a new feature or sub-feature begins.
- Merge to **Development Branch** once feature is complete.
- **Hotfix Branches** can be created from, and merged to, this branch.
- Commits to this branch should pertain solely to the specified feature.
- This branch should be deleted after final merge to **Development Branch**.

4.2.5 Hotfix Branch

- Branch for fixing a specific bug.
- Created from any branch except **Release** when a bug is identified.
- Merge to the creating branch once the bug is fixed.
- No other branches should be created from this branch.
- Commits to this branch should pertain solely to the specified bug.
- This branch should be deleted after final merge to its creating branch.

These five categories give a hierarchy of restrictions that grow more stringent as code gets closer to release. Branches have specific purposes that separate out development, testing, and deployment, and help to minimise overlap by making the goal and scope of different branches clear. Aside from the three permanent branches, all new branches will be either Feature or Hotfix branches and therefore have an explicit expectation of being deleted once their goal is achieved, thus minimising the number of branches. The rigid structure makes divergence less likely, allowing for easier, more frequent merges between feature branches to maintain parity and further reduce conflicts.

This structure is not an absolute as it is almost guaranteed to be overkill for small teams or narrowly scoped projects, would be directly detrimental for rapid prototyping, and may clash with existing workflows. It is also only capable of supporting the single latest release, and is therefore unsuitable for scenarios where multiple supported releases are required, in which cases a rather more involved model is mandated. However, it is a good illustration of how to apply best practices, and can serve as a foundation for a scheme that serves the specific needs of your working environment, workflow, and team(s).

5 Choosing your Version Control System

Which version control system is right for you depends on your project needs and workflow, with no one option being right for everyone. Below, we highlight the major differences in functionality between different systems and give an overview of the most commonly used options.

The major differences between VCSs lie in the *Repository Model* and the *Concurrency Model*.

Repository Model refers to the relationship between different copies of the source code repository. In a *centralised* model (also referred to as *client-server*), the only complete copy (i.e. with the full version history) is kept on a server and accessed via a client. Local versions typically hold only a working copy of a project tree. Code changes made in a local working copy must be committed to the remote repository on the server before they can be propagated to other users

Conversely, in a *distributed* model (git, other examples), local repositories act as peers and typically have the complete version history available by default.

centralised offers better access control, can check out specific sub-trees or files, and tend to be backed-up more often. Everyone sharing the server also shares everyone's work, which can be a downside if somebody commits buggy code. There is no concept of a 'local repository' in the centralised model, only the remote repository accessed via a server, and the user's working copy. As such, centralised VCSs are dependent on access to the remote server.

Distributed means that you do not need a network connection in order to change revisions or add changes to work. They are considerably faster and easier to work with when merging branches, and more flexible with respect to individual workflows. However, they are rather heavier on the user-side as local repositories contain the entire version history and one cannot selectively checkout specific files or folders.

Concurrency Model refers to the mechanism by which the system handles competing changes and avoids simultaneous edits. In the *Lock* system, changes are allowed only once the user has requested and received an exclusive lock on the file in the master repository. *Merge* systems on the other hand, permit users to edit files freely and commits to the repository are checked for conflicts (usually referring back to the user to decide what to do where conflicts arise). centralised VCSs can make use of either system, while Distributed VCSs tend towards Merge systems.

5.1 Overview of Common Version Control Systems

5.1.1 Git

- Repository Model: Distributed
- Concurrency Model: Merge
- The current most widely used VCS. Its maturity and market share mean that it is stable and well supported, and while Windows support has historically been an issue, this has been largely resolved in recent years.
- Services such as GitLab and BitBucket can be used to add features such as centralised hosting, issue tracking, integration with systems like Jira and Trello, automated CI/CD, etc.
- Distributed allows for faster operations and offline working.
- Large set of commands provides highly specific usability but can be esoteric.
- Staging area allows for extreme granularity of commits.

5.1.2 Subversion (SVN)

- Repository Model: centralised
- Concurrency Model: Merge, however lock can be enabled on a per-file basis.
- Per folder security
- Gentle learning curve

5.1.3 Concurrent Versions System (CVS)

- Repository Model: Distributed
- Concurrency Model: Merge
- the oldest open source version control system. Still widely used, but diminishing and no new features being added.
- Has largely been replaced by Subversion, but can be necessary where compatibility with older systems is required.

5.1.4 GNU Bazaar

- *Repository Model:* Mixed - either or both model can be utilised.
- *Concurrency Model:* Merge.
- Generally simpler to use than competitors.
- Smaller command set limits potential workflows.

5.1.5 Mercurial

- Distributed
- designed with the goals like scalability, high performance, distributed & decentralised system in mind.
- Well documented and easy to learn.
- Less powerful out of the box than competitors.

5.1.6 Recommendations

As the two most widely used systems, Git and Subversion are likely to be the preferred options, if for no other reason than the wider user base makes it easier to find help. Distributed repository systems offer more flexibility with respect to workflows and network availability, however centralised systems allow for a greater degree of security and control. As such, where flexibility is required we would recommend Git, and where there is a larger need for control we would recommend Subversion.

5.2 VCS Hosting Services

A number of services have sprung up that offer centralised hosting (both for distributed and centralised repository systems). These commonly offer additional functionality such as automated testing and issue tracking as we've touched on above. These are too numerous and too subtly varied to cover in detail, however we can summarise the most common options.

5.2.1 GitHub

As the name suggests, GitHub provides cloud hosting for git projects. Historically it has been geared primarily towards open-source projects, offering free public repositories, although recently it has started providing unlimited private repositories as well.

Pros

- Unlimited public and private repositories.
- Widely used - stable and well supported.

Cons

- Limited storage on free plan (500 Mb)
- No native build system
- No built-in CI/CD provision
- Self-Hosting is only available on paid plans.

5.2.2 GitLab

GitLab is similar to GitHub in many regards, but differs in two key ways. First, it is open-source, allowing you to self-host for free if you don't want to rely on their cloud storage. Second, it provides built-in CI/CD automation facilities, along with shared facilities to run the pipelines.

Pros

- Unlimited public and private repository
- Generous storage allocation (10Gb/project)
- CI/CD automation and runners

Cons

- Less widely used and supported than GitHub

5.2.3 BitBucket

BitBucket is designed for use by professional teams, and as such offers a more complete suite of controls with respect to access and editing of your code. The main selling points are the support for Mercurial as well as Git, and the integration with Jira and Trello for issue and task tracking - branches can be created directly from Jira issues or Trello cards. Like GitLab, BitBucket boast native CI/CD automation tools.

Pros

- Mercurial support
- Jira and Trello integration
- Tighter access controls for improved security
- CI/CD automation and runners
- Workflow control to enforce a project or team workflow

Cons

- Lacks the analytics tools found in GitHub/Lab
- Smaller user community
- Tighter controls can be more restrictive - harder to integrate with 3rd-party tools

6 Summary

6.1 Use Version Control!

- Track changes
- Easily undo changes
- Synchronise between collaborators
- Reduce the overheads of keeping multiple versions of code

6.2 Commit Related Changes

- Each commit should be bound by a self-contained intent.
- If your commit fixes two different bugs, it should be two commits
- Makes it easier for others to understand what your commit does and roll it back if necessary.
- Depending on your VCS, staging can allow for *very* granular commits.

6.3 Commit Often

- Keeps commits small and easy to understand
- Allows for more frequent commits - your code is shared more frequently with others, so it is easier for everyone to integrate changes regularly and avoid having merge conflicts.

6.4 Commit completed, testable code where possible

- Split large features into small, self-contained, *testable* chunks, and commit these when they are finished and tested.
- Services like GitLab can automate tests for convenience.
- Can be less stringent with local repositories (if available) and personal branches, use squash merges to clean the history when merging.

6.5 Write Good Commit Messages

- Make it clear what your commit does.
- Start with a short summary for quick reference when looking through multiple commits (~50 characters is a good guideline).
- Give a more complete description below. Include the motivation for the change, and how the new version differs from the old implementation.

6.6 Make Use of Branches

- Use staging to separate development, testing, and deployment.
- New branches for new features, specific bug fixes, trying out ideas, etc. - keep development separate.
- Merge often to avoid conflicts.
- delete obsolete branches.

6.7 Version Control /= Backup

- Using a VCS comes with the added benefit of having your code backed up on a remote server, but it should not be a complete backup of your working directory
- commits should be small and semantic
- files that are necessary to *development* but not *deployment* (such as notes, design documents, test outputs, etc.) should not be in your repository.

Acknowledgements

This work made use of computational support by CoSeC, the Computational Science Centre for Research Communities, through its Software Outlook Activity.