PhUSE 2017

Paper CT10

Git with It and Version Control!

Carrie Dundas-Lucca, Zencos Consulting, LLC., Cary, NC, United States
Ivan Gomez, Zencos Consulting, LLC., Cary, NC, United States

ABSTRACT

It is a long-standing coding best practice to leverage version control systems to support two goals: 1) ensure previous versions of programs are accessible and 2) a more structured means of capturing meaningful code comments as programs evolve. While some barriers to implementation must be overcome, we believe that the benefits far outweigh the costs of leveraging such tools in the clinical research field. In this paper, we bridge the gap between an established best practice and the traditional approach of achieving a similar result that exists in clinical research today. We start with an overview of Git – a free and open-source code versioning tool. Next, we provide guidance on assessing the common Git workflows to select the best fit for your organization's needs. Finally, we outline a real-world example demonstrating the process to integrate this highly beneficial best practice that meets regulatory requirements all without creating significant overhead.

INTRODUCTION

It is a longstanding coding best practice(1) to leverage version control systems to meet two key goals: 1) maintain historical copies of code and 2) capture comments describing the changes made to code over the course of its lifecycle. Throughout the remainder of this paper, this pair of goals will be referred to as code versioning goals. In its simplest and most generic definition, a version control system (VCS) is any systematical approach to record and track changes to files allowing users to revert to a previous version of a file as well as capturing its evolution over time (2). Even if you are unfamiliar with tools specifically designed for VCS, you have most likely implemented some sort of version control system at one point in your career.

In clinical research software development, it is common for version control systems to be required tools in the software development lifecycle to support compliance with 21 CFR Part 11 requirements. However, SAS programming teams are typically not required to use version control systems; in lieu of a VCS, it is an industry standard operating procedure that programmers track their updates and version history within the program itself via comments. This practice only addresses one of the two key benefits that version control systems can support. The second benefit being availability of copies of each version of their program over the course of its development cycle. While snapshot copies of programs may technically be available to SAS programmers via file and folder system backup procedures, that functionality does not retain a comprehensive collection of code updates.

A VCS can not only meet our code versioning goals but is also capable of linking these assets (copies of code and comments describing the updates to the code) to one another in a meaningful way. It is our belief that SAS programming in clinical research would benefit greatly by implementing a simple version control system to support compliance with company policies and procedures and the FDA's regulatory requirements.

This paper provides an overview of a simple version control tool called Git. We also provide an assessment of Git's workflows with guidance as to how they can be implemented to meet your organization's code versioning goals. And finally, we dive into how Git can be quickly and simply implemented within your organization to meet 21 CFR Part 11 requirements.

A GLANCE AT VERSION CONTROL

Implementing version control of any sort is in the best interests of the programmer, project team, and ultimately the organization as well. These practices can reduce rework and provide historical context of the project. While many organizations have some formal and informal version control practices, it is our experience that providing a simple yet robust tool specifically designed to support these objectives can increase efficiency at very little cost to the organization.

SOFTWARE DEVELOPMENT VS. SAS PROGRAMMING

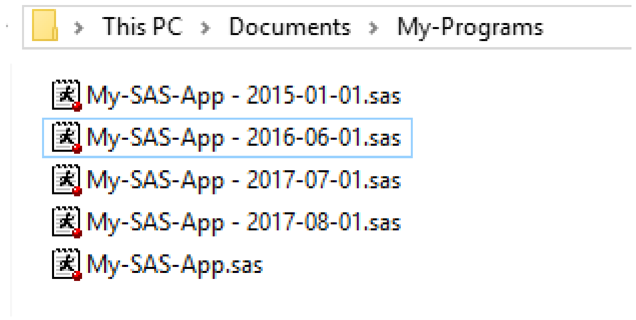
As mentioned previously, it is common for software development teams working in the clinical research industry to comply with standard operating procedures requiring them to leverage a code source control system to meet 21 CFR Part 11 requirements. These tools are used to ensure there is a historical record of changes made to code over the course of the development process. These tools also help to support change control procedures for future releases of their development projects.

It is also common for version control procedures to be defined in Biostatistical and SAS Programmer SOPs that aim to address our previously defined code versioning goals. In contrast to software development teams, these goals are oftentimes addressed through manual steps rather than being achieved through automated processes. Common practices for SAS programmers include: saving off a copy of “gold standard” code, having a working version of their program, and to track updates to their programs in their program header via comments. These practices, while an attempt at replicating the functionality provided by version control systems, they cannot effectively be enforced thereby leaving room for simple, yet devastating, losses to be experienced.

It is our belief that these code versioning goals, commonly addressed by manual steps, can be more simply and effectively met through the implementation of a version control system, saving programmers and IT teams a significant time... and a great number of headaches too!

Let’s look at the example of a simple SAS program composed of a single file and compare the program’s lifecycle without a VCS and then with a VCS, such as Git. Over the course of a project, a programmer will create, test, revise, and retest a program to produce the appropriate deliverable in its desired state. As changes are made to their program, a cautious programmer operating without a VCS may choose to create copy after copy of their program, distinguishing them from one another by leveraging a simplistic file naming convention such as appending the current date to the file name. As the program grows larger and more updates are made, the number of copies or snapshots also grows, as shown in Figure 1.

Figure 1: Version Control with file names and timestamps



This PC > Documents > My-Programs

My-SAS-App - 2015-01-01.sas
My-SAS-App - 2016-06-01.sas
My-SAS-App - 2017-07-01.sas
My-SAS-App - 2017-08-01.sas
My-SAS-App.sas

While this methodology may be appropriate for some situations, it does not provide the level of detail needed to answer key audit questions that clinical research project teams must be prepared to answer such as:

- Who made changes to a specific line of code and when?
- Is there a description of changes made?
- When was this file last modified?
- How does today’s version of this file compare to the version from two weeks ago?(3)

At first glance, providing responses to these questions seem a daunting task. However, VCS tools have been designed so that they can be answered easily. Over the course of the years, VCS have evolved with the emersion of different architectures, categories, and tools. Nowadays there are plenty of options to choose from, but there are basic concepts that are common across them all:

- Repository: the database that contains all the information about the files being tracked.
- Changeset: a set of changes made to one or more files. A changeset is the individual unit that is stored in the repository and includes the information generated after editing existing files, adding new files, as well as removing files that are no longer necessary.
- Commit: the process of taking a set of changes to create a changeset that is ultimately stored in the repository. Committed changes are accompanied by a corresponding commit message that briefly describes the changes introduced to the source code.

An in-depth explanation and comparison of VCS tools on the market is outside the scope of this paper. Instead, we have selected to focus on Git and its use case to support programming teams’ code versioning goals.

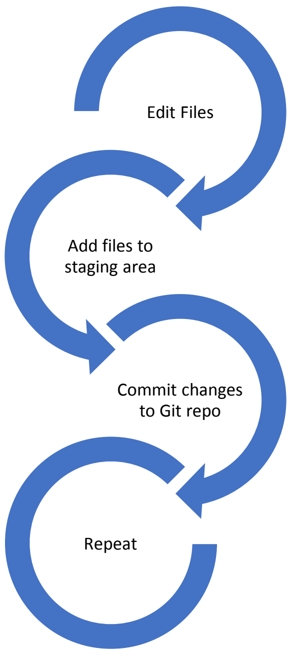
GIT WITH IT!

The formal definition of Git states that it is a "a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency"[4] our two previously defined code versioning goals.
Let's take this definition and break into pieces:

- Free and open source: Git can be downloaded and installed on any workstation or server without licensing or financial fees of any type. Furthermore, the entire source code that runs Git is available to anyone interested in looking at it or reviewing it to determine exactly how any specific functionality works. This provides full transparency to the users.
- Distributed version control system: Unlike other VCS tools, Git does not need any other servers, software applications, or network connectivity to achieve its goals. A Git repository is self-contained and it holds all the files that are part of that repository.
- Designed to handle everything from small to very large projects with speed and efficiency: Git's design makes it a great choice for small projects with few files all the way to projects with thousands of files that take up Gigabytes of storage.

In addition to the simple functionality described above, Git also supports multiple people working on a single set of files as a result of is its two-step process to commit changes to the repository. Unlike other tools, Git has a "staging area" that temporarily holds the contents that will eventually become the next commit. Even if multiple files have been modified, Git allows users to be very selective about what files and what changes they want to capture and commit to the repository. Through this iterative process, illustrated in Figure 2 is how Git repositories grow over time.

Figure 2: Steps to commit changes to the repository



GIT WORKFLOWS

As stated before, Git is "designed to handle everything from small to very large projects". This means that Git is also flexible and can be leveraged in almost any digital environment. At one extreme, Git can support more than 18 thousand developers collaborating seamlessly in the ongoing development of the Linux Kernel[5], while at the same time it is an excellent manager for small projects such as the writing of this paper. It can even be used for such simple tasks as keeping track of single-user activities such as to-do lists or grocery shopping lists.

This flexibility is achieved through the adoption of a "Git Workflow", which in essence, is a set of guidelines[6] followed by all the contributors to a repository to ensure that changes to files are introduced in a controlled manner. These workflows vary in complexity depending on the need, ranging from centralized linear flows that simply stack commits over time to decentralized workflows that can support multiple copies and branches of the same repository to maintain and develop several streams of code in parallel.

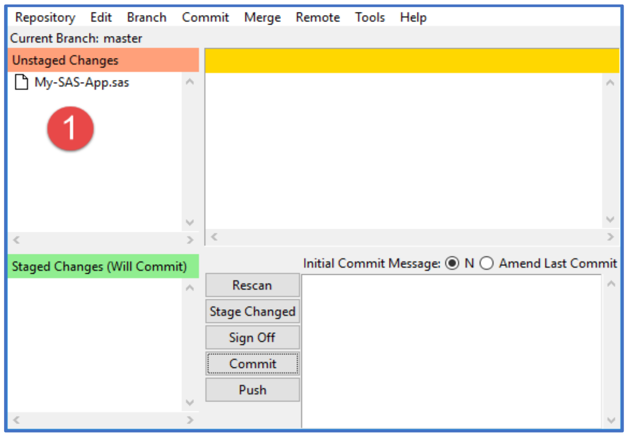
REAL-WORLD EXAMPLE

Here we present one of Git's simplest workflows that can be adopted to leverage the power of Git without significant overhead. This workflow automates the manual processes currently used in clinical research. This particular workflow will function regardless of the number of developers working on a folder, whether the new changes consist of modifying existing files or files being added or removed:

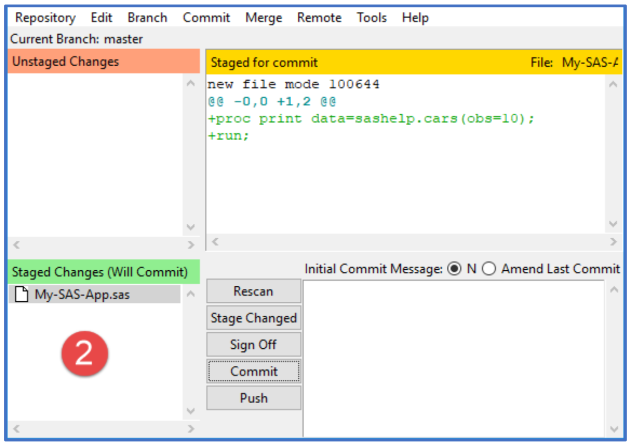
1. A programmer/statistician works on a SAS file and makes changes to it.
2. When the programmer/statistician is satisfied with the set of changes, they are moved to the staging area. At this point the programmer/statistician may review the changes introduced to confirm only the desired files are committed.
3. The programmer/statistician provides a descriptive commit message and permanently stores the changes in the repository.

Using the example of My-SAS-App.sas from above, let's see how through this workflow, a programmer/statistician would commit the changes after making edits to this file.

Figure 3 (series of 3 images): Committing changes to the Git repository

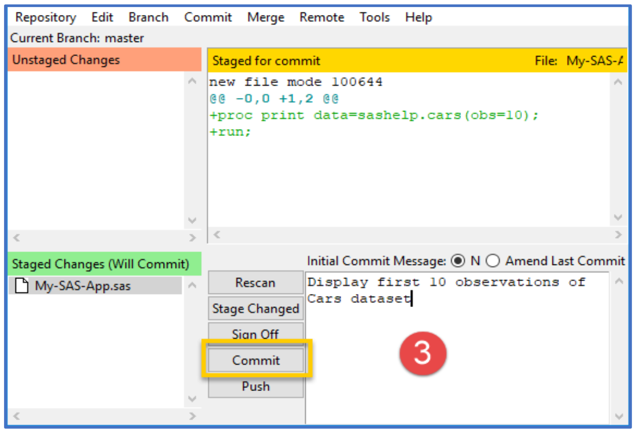

Repository Edit Branch Commit Merge Remote Tools Help
Current Branch: master
Unstaged Changes
My-SAS-App.sas
1
Staged Changes (Will Commit)
Initial Commit Message: N Amend Last Commit
Rescan
Stage Changed
Sign Off
Commit
Push

In this first screenshot, the programmer/statistician has made changes to the file My-SAS-App.sas. At this point, Git detects that the file has been changed since it was last committed to the repository and thus the file is listed as an "Unstaged Change".


Repository Edit Branch Commit Merge Remote Tools Help
Current Branch: master
Unstaged Changes
Staged for commit
File: My-SAS-A
Staged Changes (Will Commit)
My-SAS-App.sas
2
Initial Commit Message: N Amend Last Commit
Rescan
Stage Changed
Sign Off
Commit
Push

```
new file mode 100644
@@ -0,0 +1,2 @@
+proc print data=sashelp.cars(obs=10);
+run;
```

The next step in the workflow is to move the program into the Staged Changes (Will Commit) area. During this step, the programmer/statistician can look at the changes (presented under "Staged for commit") to confirm that the desired edits are about to be saved to the repository

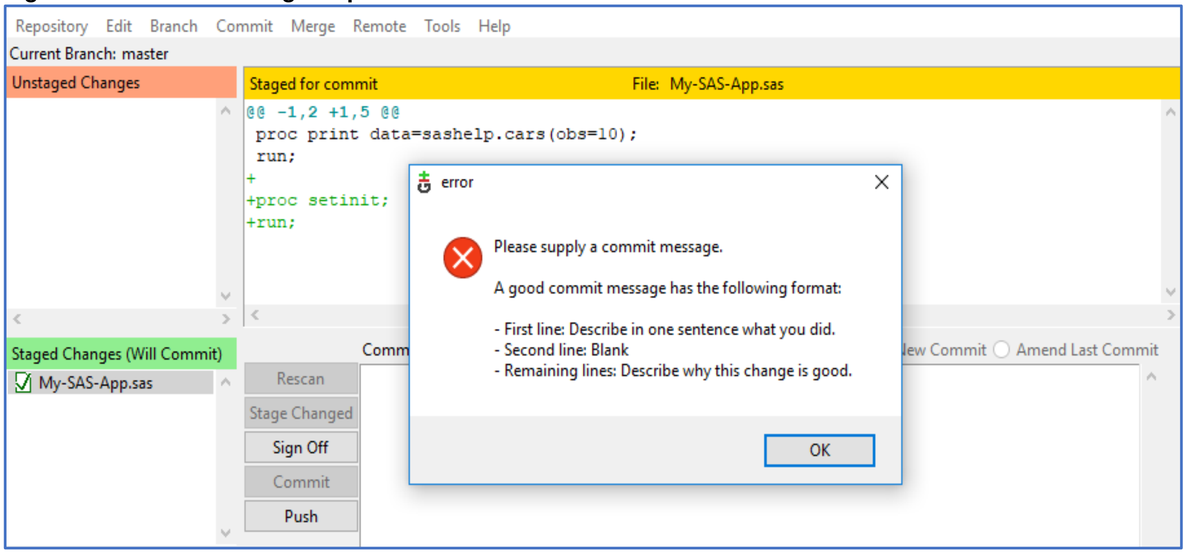

Repository Edit Branch Commit Merge Remote Tools Help
Current Branch: master
Unstaged Changes
Staged for commit
File: My-SAS-A
Staged Changes (Will Commit)
My-SAS-App.sas
Initial Commit Message: N Amend Last Commit
Rescan
Stage Changed
Sign Off
Commit
Push
3

```
new file mode 100644
@@ -0,0 +1,2 @@
+proc print data=sashelp.cars(obs=10);
+run;
```

```
Display first 10 observations of
Cars dataset
```

The final step is to add a descriptive commit message explaining the changes introduced to the repository. Thus, the changeset introduced to the repository contains information about who, when, and why changes were made to files.

Figure 4: Commit message required



```
@@ -1,2 +1,5 @@
 proc print data=sashelp.cars(obs=10);
 run;
+
+proc setinit;
+run;
```

Please supply a commit message.

A good commit message has the following format:

- First line: Describe in one sentence what you did.
- Second line: Blank
- Remaining lines: Describe why this change is good.

In this final screenshot we can see that Git requires a comment to be added that describes the updates to the file being committed.

It is important to note that Git also supports the scenario where a program is updated, those updates are committed, and then the programmer identifies the need to strategically retain updates most recently committed but having to reject updates added to an earlier commit. This is critical functionality to reduce loss of work but to also quickly remove content updates that are not of value.

THE ELEPHANT IN THE ROOM: 21 CFR PART 11 COMPLIANCE

The implementation of Git within your organization will likely require collaborating with your IT group to get the system validated before it is used for project work. The reason this is likely to be necessary is that 21 CRF Part 11 requires that if a manual process currently managed via standard operating procedures is replaced by a tool that automates execution of those tasks, it will have to be validated.

Validation of Git can be quite simple for programming teams that choose to implement the simple workflow described in the previous section. The validation project would be focused on two key tasks: capturing the installation and deployment of Git to the necessary users/machines and then validating that Git can effectively meet the two code versioning goals defined at the beginning of this paper. The latter objective would be met via simple user acceptance tests.

Additional requirements or functionality to support more complex workflows can be implemented but would likely lead to more complicated validation projects and may delay implementation of this useful tool. As such, we would advise starting with a simple implementation in place to support key requirements and determining if additional functionality is desirable in the future.

LIMITATIONS OF GIT

There are a handful of Git features that clinical research programming teams will need to consider if they add Git to their toolset. First and foremost, Git supports multiple individuals working on and committing updates to the same file. Given the nature of clinical research programming practices, this feature should not be used by programming teams. We would also recommend retaining guidance documentation, such as an SOP, that delineates the responsibilities of project team members in support of the organization’s double-independent-programming requirements.

And finally, while there are a great many benefits to leveraging a tool like Git, open source software may introduce a slightly higher level of risk to the organization that updates will be released on an irregular basis and there may be a delay in applying bug fixes that would be less likely with a paid solution.

CONCLUSION

It may have been the case years ago that the version control systems available on the market weren’t quite right for clinical research programming teams but that is no longer the case. Applications such as Git can provide clinical research programming teams with a light and effective tool to help them meet their code versioning goals and allow them to move away from having to rely on manual processes.

This paper has made the case for supplanting manual processes with a VCS tool like Git. Additionally, it has provided an overview how one of Git’s workflows can be implemented to meet your organization’s code versioning goals as well as outlining how Git can quickly and simply implemented within your organization to meet 21 CFR Part 11 requirements.

REFERENCES

(1) Andrew Hunt and David Thomas. (2000) *The Pragmatic Programmer.* Crawfordsville, IN: Addison-Wesley, pgs. 86 – 89.
(2) Git Getting Started: https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control
(3) Andrew Hunt and David Thomas. (2000) *The Pragmatic Programmer.* Crawfordsville, IN: Addison-Wesley, pg. 87.
(4) About Git: https://git-scm.com/
(5) Linux GitHub: https://github.com/torvalds/linux
(6) Atlassian Git Tutorials: https://www.atlassian.com/git/tutorials/comparing-workflows

ACKNOWLEDGMENTS

CONTACT INFORMATION

(In case a reader wants to get in touch with you, please put your contact information at the end of the paper.)
Your comments and questions are valued and encouraged. Contact the authors at:

Carrie Dundas-Lucca, MSIS
Zencos Consulting, LLC
1400 Crescent Green Dr., Suite 215
Cary, NC 27518
919-256-5979
cdundaslucca@zencos.com

Ivan Gomez, MSA
Zencos Consulting, LLC
1400 Crescent Green Dr., Suite 215
Cary, NC 27518
919-238-1297
igomez@zencos.com