When You Make Sausages, Do You Really Want to Know What Is Inside?

Randall S. Becker >> **President** >> Nexbridge Inc.

Version control technologies have evolved significantly since the 1970s, particularly as a result of workstation-based development. Processes and policies built around trying to manage how software flows to and from workstations is challenging. With flexible process enablers, like git, different approaches to this problem are available, which has unfortunately led to conflicting development cultures. In the git sub-culture, there is a core question that boils down to whether you really want to know what is in your sausages (change history) or not. In concrete terms, there are process questions of how much interaction developers have with central repositories, including when and why, and then what library managers do with what is delivered by developers. This article describes why you need to look at this question and how to choose which approach to use to manage your software. It also will discuss some roles and responsibilities you need to consider that may change when using git.

Preamble

Bob and Steve are good friends and colleagues. Every Thursday, the two of them go out to the food truck at the corner. Bob always orders a sausage with the works. Steve likes his simple: just brown mustard. While Bob joyfully eats his in large bites, juices rolling off his chin, Steve savers every morsel and flavour that comes from the tasty indulgence. Bob often looks at Steve with a bit of disdain, thinking that he overthinks his lunch, while Steve wonders how Bob does not get indigestion.
Back at their desks, side by side, they continue working on their shared project. Just as he ate his sausage, Bob prefers to check-in his code only in big bites, once everything is tested and perfect. Steve, on the other hand, saves every change he makes, recording lengthy dissertations about the purpose of each change, even if it does not work yet. Bob thinks Steve is a bit obsessive about his change history, while Steve thinks Bob is being obsessive about not wanting to show any work that does not test cleanly. To each, this is a matter of pride. Their philosophical disagreements have even become legendary, to the point that the library manager, Jan, has had to break up some rather heated arguments over who is working the right way.
Jan has become an expert in eye-rolling at the two colleagues. At the end of the day, when it comes time to release code, both developers deliver the same high quality delicious sausage. The code moves cleanly through the deployment process and rarely, if ever, is there an issue, so why the arguments? This question of how to work seems to mean so much to both Bob and Steve, but when Jan gets the code, it all seems so nonsensical.

What you have just read was mostly fiction, based on true events, with the names changed because our legal department told us we had to – you know who you are. The question of Sausage Making, as the git community calls it, is a core philosophical disagreement in many organizations, and you will probably encounter it. The root cause of this disagreement calls into question how people fundamentally view their jobs, the type of communication they have with others, and the pride in their own work. For the author's part, the key advice here is not to ignore or minimize the impact of this question. In order to understand it, in a NonStop context, we need to look back in time, into our own history, and look at how code has been managed historically.

A Brief History of Culture

In the beginning, we had EDIT and TAL, but no tools. Our code sat in groups of Guardian sub-volumes that we fixed in place. As time passed, and we patched code by compiling it in production, we realized we needed to keep official copies and work in progress. Deployment became people typing and later scripting "FUP DUP". In the early 1980s, someone had this idea to try to port some standard SCM tools. Then came CONTROL and later RMS, which both did things differently, but we finally had version numbers, change logs, and central repositories. At the same time, the outside world was evolving through SCCS, RCS, PVCS, CVS, and then Subversion. Still, version numbering and central repositories reigned. The cultural impact was not entirely subtle, but important: quality == good; clutter == bad.

In the beginning, we were motivated only to record finished products.

In order to reduce the massive numbers of versions that would result from normal developer activities, we, as a culture, kept our interactions with the central repository to a minimum. It was natural. That and no one likes SCM systems anyway, right?

Then came the Enterprise Toolkit (ETK) and everything went into the proverbial loo. The idea of checking out code, making a change, and checking it back in became impractical for the NonStop-based repositories. Tools soon emerged to deal with that but not before customers had moved their code to off-platform repositories like CVS and Clear Case™. Suddenly, NonStop was isolated. Interactions with the NonStop repositories became even more infrequent (and painful). Some customers stopped storing code on the NonStop and just pushed objects.

But by 2007, the World as a whole had moved on and left NonStop behind, which is unfortunately where we still are today. In the Wider World, people were playing with Linux-style approaches to code management that had theoretical roots back in the late 1980s, but would not see production for some time – the Distributed Version Control System (DVCS), like git and Mercurial. These systems greatly improved workstation-based development by allowing history to be shared and changes to be added without constant contact with the central repository. This was actually

revolutionary – not technologically, systems like ClearCase™ had some of that, but because of its cultural impact. The DVCS allowed two developers to commit their code in isolation, and then combine the code later. Branches became lightweight, easy to create, deliver, and destroy when no longer needed. Culturally, this change allowed developers who were previously frustrated by the need to always be connected to the central repository, to keep their own private change histories – some of us call those research notes. These private histories became a trace or footprint of your work.

For security managers and intellectual property policy writers, workstation-based development was the stuff of nightmares that continues to this day – but that is another lengthy topic to be discussed in the next article.

In git, you have control of your own footprint.

For managers, being able to see all of the research notes was a boon to track what people were doing and whether there were potential quality issues slipping in. As you might guess, this was not received well by some developers. This negative reception was actually unnecessary but by the time git got to it, it was too late, culturally. In git, this footprint is only visible if the developer chooses to share it. The impact pushed further down the path of minimal interactions with the repository; but in a DVCS, this is rather like not saving your work in the editor. Resist committing at your own risk, at least according to Steve. Bob was in the other camp, wanting only to commit work that was perfect.

The DVCS Divide

So here we are, in the present, facing a question of sausage making. It is your development group who really needs to decide, do you want to know what is in the sausage - do you want to see the developer footprints, or do you just want to see the final product. In git, you can do either, or both. This is mostly a matter of culture and process. In my previous article, we saw how you can use merge-squash to make the final deliveries immutable – the git community calls it atomicity if you are Googling this. The same technique applies to changes being pulled into your integration branch. Let's take a look at how the sausage makers work.

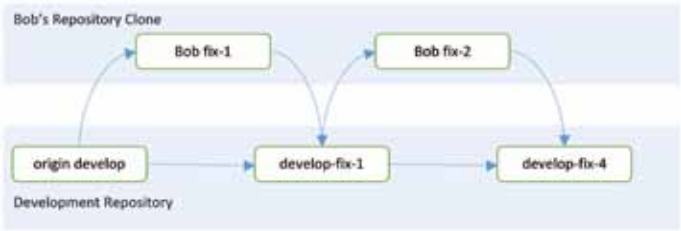

Figure 1 Bob's Ideal View of the World

From Bob's perspective, the repository where he works contains the development branch and his own topic branches. Topic branches are the light-weight branches where you make changes for a specific unit of work, from a bug-fix to a project. Bob's topic branches are very small, containing only one commit for the work he has completed. Once he is ready, Bob pushes his topic branches to the development repository where Jan merges them into the main development branch. It is actually up to Jan how to sequence Bob's changes into the integrated development branch. Bob can continue to work on the second fix either off the same origin point as his original change, or build on top of it. That is actually dependent on the nature of the fix.

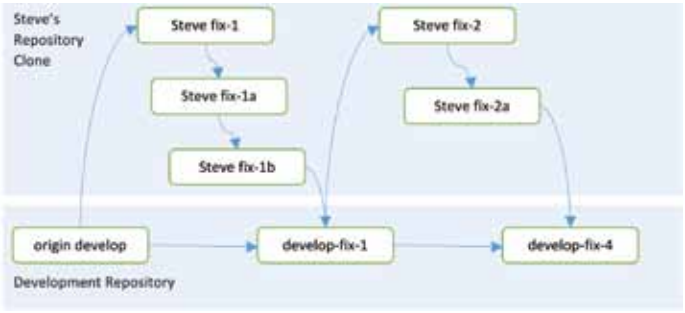

Figure 2 Steve's Ideal View of the World

As with Bob, from Steve's own perspective, the repository where he works contains the development branch and his own topic branches. Steve's topic branches are longer than Bob's because they contain his intermediate changes though the final commit, at which point, he pushes his topic branches to the development repository where Jan merges them into the main development branch. Jan can choose to perform a merge-squash to treat Steve's work as a single commit, or can take the entire history. Like Bob, Steve also can continue to work on the second fix either off the same origin point as his original change, or build on top of it. If you are a small shop, this is probably the extent of the process and decisions you need to look at initially.

Working Together on a Project

However, if Steve and Bob are part of a larger team, you will need to add a layer to the picture when the two work on a project together, because things become a bit more complicated as they share code. In fact, the roles and responsibilities are pretty much the same, but unless Jan wants to stay in the middle of the arguments between Steve and Bob, the two of them are going to have to learn put on Jan's proverbial hat – that also means that the team maintains proficiency and backup skills for the merge process, something that is essential in an environment with many branches.

Working together means having your own integration branch.

When you decide to have two members of your team work on a change together, and the change takes more than a few days – like an ATM key management enhancement – they are probably going to have to have an integration branch for that change. There is really no difference compared to the main development branch managed by Jan, except it only contains changes for the specific fix that Bob and Steve are working on. Bob can keep his sausage making to himself, while Steve can continue to write a lot of history. When it comes time to sharing changes with each other, Bob can merge his changes onto their **ATM_Key_Enhancement_1234_Branch**, while Steve rebases to pick those up, or Steve can merge his changes and Bob can rebase, or both. That is how they keep in sync with each other on their ever-growing common branch. Pretty cool? Actually, mostly for Steve. Bob probably might think he won't like this approach much because it does mean that their common branch may contain more than one commit. However, if they agree, collectively, that only working changes must be shared, then Bob actually can be the happy one, while Steve will have to wait to rebase or merge until he is at a stable point. It is still pretty cool, because the history is kept intact for both of them. If Bob insists

that he does not want to see Steve's history, then Steve can use a merge-squash to – remember what Jan can do? – publish his changes, so their joined history is also clean. By the time Jan gets it, all conflicts are gone – both in code and interpersonal. And better still, the development branch still only contains functionally consistent commits. And that makes the quality of delivered code higher.

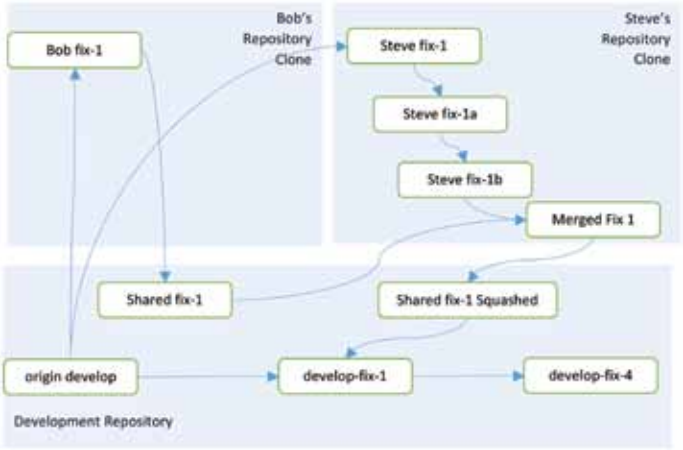

Figure 3 Three Repositories Working Together

In the above diagram, Bob and Steve are collaborating on a fix. Either Bob or Steve could have done the rebase merge and delivery, but in this example, Steve was chosen. Bob's repository contains an abridged image of the history, much to Bob's delight, although if he chooses to fetch all branches, he can see Steve's work. Technically, the **Shared fix-1** commit is actually the same as Bob's initial fix, and the squashed commit could be done either in Steve's repository or on the server, depending on who is performing that operation.

Having a good process design and cookbook of work instructions is important so that everyone knows and agrees on what they are supposed to do and when. The branch pointer, **ATM_Key_Enhancement_1234_Branch**, would first be on **Shared fix 1**, and then move to **Shared fix 1 Squashed**. Jan would use that branch pointer, to merge the change into the **develop** branch.

The team can even can even choose to drop Steve's and Bob's branches as part of regular maintenance, which will orphan the commits along their branches and cause them to be marked for removal during a git garbage collection. But Steve still has the option to retain his own history in his repository for his own research notes.

Production should never see the insides of a sausage.

But none of the details of these changes should ever make it to production. As the previous article discussed, when the final code was delivered, the detailed footprints disappear into the atomic immutable commit that represents the release itself.

Conclusion

The details of sausage making are of concern when thinking about how developers work and keeping them productive. Often, the repository manager is the arbiter of the philosophical disputes discussed here, but their primary role is to ensure that quality code deliveries make it to production. These footprint discussions are valuable because, ideally, they improve how developers work together and how both options can co-exist. Consider carefully whether footprints are essential to your business, particularly if you are conducting audited research. But mostly, just make sure that the sausages taste good by the time they are finally served to your customers.

Randall S. Becker is a speaker, author, and consultant on Policy and Process that delivers continuous availability. He is an expert in Software Configuration and Change Management since 1989 and has spoken at many NonStop and community events. Randall can be contacted at: +1.416.984.9826 or rsbecker@nexbridge.com.