Versioning Style Guide

Martin Baute

April 19, 2016

Preface

I could have written down a number of rules. Actually, I did so at first, but the resulting paper was a dry, depressing piece of legalese. So I wrote a tutorial instead, starting with the first source file and extending into maintaining multiple development branches of a complex product. Credit goes to the engineers at Amiga Inc. (1982-1984), as this system was strongly inspired by AmigaOS.

Some of the following goes deeper than just versioning, into the realm of release politics. At some points I will ramble a bit about things, at other points I just quietly depict procedures that might feel "strange" at first without saying much on the "why". In those places, I wouldn't get to the end of it if I'd start rambling.

Pre-1.0 versions aren't for a wider audience, for example, and I have to really bite my tongue not to get started about the stupidity of public betas that are not even alpha quality in my book, or those eternally-pre-1.0 products that are considered "stable enough" by their maintainers...

(Breathe. Relax. Breathe. Focus. Yes, that's better. ;))

1 Software != System

When you are reading a paper on versioning, chances are you want to use some versioning control system (VCS). CVS, Subversion, PVCS, ClearCase, Mercurial, git... all of them being great *tools*, but none of them a *system* by itself. None of them enables you to tell which versions of which source files went into building a given executable. To achieve this, you need a *system* in addition to the *software*.

Actually, you need not one, but *three* versioning systems:

- for the *sources* (exe/main.c, lib/my_func.c, doc/chapter5.xml);
- for the *components* (MySoft.exe, MySoft_Manual.pdf, MySoft.lib);
- for the *product* (the MySoft package).

In the following chapters, I will describe an integrated system that, I hope, makes maintenance easier for users, testers, and developers alike.

2 Beginnings

You have written a small program for your personal use. It consists of a couple of source files. After going through several revisions, you realize it is time to put your sources under version control.

Perhaps you already have VCS software of some sort running on your system, and adding another repository is quite easy to do. Perhaps you open up another SourceForge project. Perhaps version control is all new to you, and you have to read up on the basics - in which case I, personally, would recommend looking into Subversion[1]. It has a "traditional" interface, is almost as well-supported as the venerable CVS but easier to use, and is much superior to CVS in handling branches and tags while still being of low complexity.

Some version control systems allow for *keyword expansion*. Some keyword (like *Id*, placed in a source comment, gets expanded to a status string usually containing file version, file name, date, and username of last editor whenever you check out that file from the repository. This can help you to link files "flying around" on your hard drive or (as hardcopy) on your desk, with a given version of that file in the repository.

For our version control *system*, we will just ignore that Id string in any case. Its format, and the format of the file version used in that string, is vastly different from software to software, and many of the newer, distributed VCS do not support it at all, which makes the concept useless for any software-independent system.

But you have your files under version control now, and you can use the features of your VCS to do... whatever your VCS allows you to do. (Read the manual!) You made the first step.

3 Components

A number of source files makes up a *component*. Multiple C files and a makefile make up an executable, or library. Multiple LaTeX files and a makefile make up a documentation. Given the component, we must be able to identify which source files in which version were used to build the component; and we must be able to retrieve all files required to build the component from the repository, with a single command.

The file versions added automatically by our VCS are worthless for this. They differ from VCS to VCS, and they cannot be retrieved from the component. So, instead of pushing the individual file versions into the component somehow, we do it the other way around: We give a *component version* to the component, and then tag that component version on the source files used.

When you compile your first release of your project under version control, every component is assigned a 1.0 version. Now, you have two problems:

- how do you get that version number into the component binary so that it can be retrieved?
- how do you get that version number into the individual source files so that them - and only them in their current version - are marked as constituting the component 1.0?

[1] http://subversion.tigris.org

The answer is, of course, "it depends", and it depends on the VCS you are using. Virtually every version control software supports "tagging" of source files, and that is what we will use at this stage. When you compile version 1.0 of MySoft.exe, every contributing source file should be tagged "MySoft.exe 1.0". Find out how you can tell your VCS to "check out all source files tagged MySoft.exe 1.0". You will have to do some manual browsing here to find out how your VCS of choice ticks.

Once you have that in place, you're done with the hardest part. Promise. ;)

4 Pre-Release

What was said in the previous chapter leaves you free to check in source files more often than tagging them. If more than one person works on the project, the modus operandi should be to outright ignore any file versions that are not tagged. That's work-in-progress of somebody else; the latest "visible" version is the latest "tagged" one. If you are working in a team, work out some MO to handle concurrent development (multiple developers working on the same file, potentially conflicting edits and two developers attempting to use the same tag). As this again depends on the features of the VCS used, you'll have to find out how yourself, but it would probably be a good idea to "soft lock" a component being worked upon, so that you will be notified on checking out sources if someone else is already working on the same component.

Whenever your work-in-progress has reached another stable point, tag it again. Don't be afraid to tag often, especially in the pre-release phase. Too many changes between two tags render any error-seeking `diff` useless. A typical changeset should address a single problem, be describable with a single sentence (the check-in comment), and include an update to the documentation where applicable (i.e., don't postpone documentation updates till later, since later never comes). It also should compile without warnings and pass the regression tests (which you have automatted to run on each check-in, didn't you?).

If a check-in comment doesn't fit all files touched in the changeset, or does not describe all changes made, then your changeset isn't a good one. This requires experience. If you discover problem B while working on problem A, make a note, and create a new changeset for problem B *after* you finished the one for problem A.

Sorry for the rambling, but this is *important*. If you don't task yourself here, the useability of a version control system degenerates sharply.

These frequent "intermediate" tags always increment the *minor* version number of the component - 1.1, 1.2, 1.3 and so on.

5 Product Build

So now you have a collection of components, and you put them together to form the larger product for the first time. While your components have undergone individual testing (you *are* using automatted test drivers to ensure correctness of your components, aren't you?), the product as a whole is untested. Hence, while the components are versioned 1.x, the product is versioned 0.1.

This is an important point in our version control *system*. MySoft.exe 1.27, MySoft_Manual.pdf 1.8 and MySoft.lib 1.31 constitute MySoft 0.1. Most likely, you will continue working on MySoft. There will be a MySoft 0.2 some day. So, when you tag your sources the first time *after* a product release, you bump their *major* version number and reset the *minor* version number to 1.

The first step towards MySoft 0.2 are thus MySoft.exe 2.1, MySoft_Manual.pdf 2.1, and MySoft.lib 2.1. The minor version will probably iterate a couple of times before you are ready to build MySoft 0.2, but you just established an "information link" between the product version 0.1 and the 1.x component versions. A 2.x component does not belong to a 0.1 product. If someone approaches you with a bug report and states that MySoft 0.1 doesn't work because MySoft.lib 2.14 throws an exception, you know that the problem is probably a corrupt update or somesuch.

In case that a component doesn't change at all between MySoft versions, you still have to bump the major version number of the component - 2.**0** is reserved for this case. This is a somewhat tricky and implicit "information encoding", but I find the idea appealing. If you don't like it, don't do it. The information content ("unchanged since last MySoft version") is marginal anyway.

Using this scheme, you can work comfortably all through the pre-release phase of MySoft. Things get a bit more complicated once you "go public" with MySoft 1.0.

6 Alpha - Beta - Release - Patch

We'll take a jump into the future. MySoft has been released to the public. Version 1.6 just hit he shelves (consisting of components versioned 30.x), and it's a stunning success. You acquired a couple of co-developers helping you in the development.

As usual, once a new product release has been made, you bump the components' major version on their next tagging, to 31.x. People using MySoft 1.6 intensively enough to care for component versions will know that 31.x sources or components belong to a future version, even if they come across them in isolation. Your co-developers know that, too.

After a couple of changes, you are satisfied with the result. You build a new MySoft package. Of course, the next logical version number is 1.7, but you haven't tested that one yet. How to avoid people jumping on what they consider the next release of their favourite software package, and running into any number of undiscovered bugs?

6.1 Alpha

The answer is "1.7 alpha 1". *Alpha* tells people that this is an internal development snapshot, no warranties, hands off unless you're a MySoft developer yourself. The number is there so that you are prepared to go through a number of minor patches before 1.7 is ready for release.

Your co-developers give 1.7 alpha 1 some testing, and discover several minor flaws (or come up with better suggestions). Changes are applied (with the

components' minor version bumped accordingly), and a new version of MySoft is built - "1.7 alpha 2".

6.2 Beta

That version is better, your co-developers give it a thumbs-up after some weeks of testing. Without applying any changes, you tag 1.7 alpha 2 as 1.7 beta 2 (note that the number after the alpha/beta part is *not* reset, to keep the "information link" that alpha 2 and beta 2 are, in fact, identical). You release the software to a selected group of beta testers (and early adopters), to put some stress on it in real-life conditions, and to have a higher chance at discovering the more obscure bugs.

Personally I advocate against "public betas", as they don't do the term "beta tester" any justice. A beta tester is much more than someone using the software on his system and looking if it breaks. A beta tester puts a software through all kinds of possible and impossible situations, tracks down misbehaviour to the smallest set of actions required to reproduce the error, and files a report to the project's bug tracker. That is a skill set in itself, in no way inferior to a code developer. Good bug reports actually help you developers to identify and fix the problem in a timely fashion, and *immensely* improve product quality. Good beta testers are an asset to any project. "Public beta" implies that anyone is good enough to be a beta tester; that is not the case, and if your bug tracker either remains empty or overflows with reports that are of little use, don't say I didn't tell you.

If a beta tester comes up with a flaw or bug that keeps 1.7 beta 2 from being released, you just apply more changes to your sources, bump the minor number of the affected components, and release 1.7 alpha 3. If the changes are *really* minor, like correcting typos, you might go right for 1.7 *beta* 3 instead and bypass the alpha phase - but be careful, every beta release means lots of tiring work for your beta testers.

6.3 Release

Some day, a beta will get the thumbs-up from your beta testers, and you will tag that beta as 1.7, to be released to the public.

If you aren't doing Open Source, all your users will ever see is a major.minor version number. If you *do* Open Source, they will know to stay away from anything that has more than that in its version label.

6.4 Patch

Then there is the issue of maintenance. Just telling people to upgrade to the latest version isn't a nice way of handling issues with older ones, at least not with complex products like operating systems, compilers etc. – there might be some downward compatibility issues, or the update might be too costly to be viable to the customer. Bottom line, you might want to provide updates to 1.6 of your product even after 1.7 came out.

Here, again, the feature set of your respective VCS decides how you handle things on the tool level. You might want to create a branch whenever you start working on a new release. As for the versioning *system*, well, the 30.x

components are still there, so that is what you apply any subsequent patches to. It might be a good idea to apply a "MySoft 1.6" tag to the 30.x components involved, so you don't have to look up the individual component versions involved. You still need a way of keeping the *component* tags intact, or you wouldn't know that it's MySoft.lib 30.14 you are applying the patch to (so you wouldn't know you have to apply a 30.15 tag when you are done).

As for *versioning* your patches... AmigaOS provided a `SetPatch` binary (a component of its own), which combined all patches for all versions of the operating system. Regardless whether you were using AmigaOS 2.1 or 3.0 or whatever, as long as you had the latest `SetPatch` run at boot time, you had the latest OS patches installed. It was the one component you would use despite its major version number not matching your other components. Personally I liked this scheme very much, as it not only made maintenance (and looking for new patches) much easier, it also meant that developers were mindful of how many changes could be squeezed into a patch and what required a new version.

You could copy this approach, or introduce a per-version "patch component" if you consider a one-size-fits-all uberpatch to be too limiting / demanding. If you don't want to use this approach at all, you have to denote the "patch level" of MySoft as a whole - perhaps as "MySoft 1.6 patch 3" or somesuch. Since it is quite common for even large-scale projects to abandon older releases as soon as a newer one comes out anyway, I won't go into more depth here.

7 The 1.0 Release Rant

"Release early, release often" is the credo of the Open Source community. I believe this concept is fundamentally flawed.

While project maintainers might look on frequent public releases as a means of quicker evolution through higher user feedback, this concept waters down the message given to the user by the version number. On the one hand pre-1.0 versions are advertised as the solution to real-life problems (like OpenSSH and OpenSSL), and the user is expected to accept the concept of pre-1.0 software as critical part of his system. On the other hand, complains about buggy behaviour or functional shortcomings are frequently declined with a haughty "it's pre-1.0, what do you expect?".

Either a product is fit for the public (1.0 or later), or it shouldn't be packaged for public access outside a public beta. A public beta is not a solution to a productive problem, but a workload to the person employing it.

Some maintainers believe that releasing a 1.0 somehow requires all the features they ever dreamt of to be implemented. This is folly. Set yourself realistic goals for your 1.0 release. Make notes of which features you will reserve for later versions. Implement a functional subset of your dreams, and release your 1.0 with a "todo" list. Or better yet, release your 1.0 *without* a "todo" list, and surprise your 1.0 users with stunning feature after stunning feature in subsequent releases. That will make for much better PR than eternally chunking out pre-1.0 versions because in your mind you reserved 1.0 for your personal software utopia.

7.1 The 1.0 Warranty

Releasing version 1.0 of your product is a warranty given to the user: Your product does what it is advertised to do. There is documentation available that is up-to-date and complete. The software was reasonably tested and behaves correctly in the face of erroneous input. You intend to still provide support for this version for a reasonable amount of time should a later version be incompatible with it.

The user, on the other hand, knows that a pre-1.0 version isn't really intended for casual everyday use, and might misbehave spectaculary. He also knows that a 1.0 version still leaves things to be desired (and hopefully implemented in later versions). He knows he's using your pre-1.0 at his own risk.

"Waitaminute," I hear the OSS people cry, "OSS software comes without any warranty because I wasn't payed for it. They are using it at their own risk anyway!" To those people I can only say, you are a disgrace to software engineering and haven't grasped the concept that users will download your software because they expect it to work as advertised. They might not be inclined to sue you for not delivering, but they *are* inclined to give you a lip if you release shitty software.

If you're playing in a band playing in public places, not getting payed is no excuse for bad performance either, and people *will* run you off if your performance sucks. If you don't feel up to delivering quality, you're excercising the wrong hobby. Try ikebana as an artistic output that doesn't offend other people, or hole up in a basement, but spare other people the trouble of finding out how bad your performance actually is.

Sorry, ranting again, but this one came from the heart.

8 Miscellaneous

8.1 1.0 Release

The 1.0 release is somewhat special: It marks the transition from pre-release sequential numbering to the alpha / beta / release cycle. It is quite likely that you want to skip alpha, and possibly even beta phase completely for 1.0, since you did spend considerable amount of time testing and improving MySoft 0.1, 0.2, 0.3 etc. etc.

Ideally, your 1.0 should be the most well-tested and solid release ever.

8.2 Major Releases

At some point, there will be improvements on your todo-list that cannot be achieved without breaking downward compatibility. The new version of MySoft will offer much more functionality, but e.g. scripts written around previous versions of MySoft will have to be re-checked whether they work correctly with the new version.

That is why you will bump the *major* version number of MySoft with the next release. After 1.7, the next version will be 2.0. As to the alpha / beta cycle, everything remains the same.

The thing, however, is that there will be people refusing to update to 2.0. They might not want to go through the maintenance hassle of checking all

their scripts again. They might not consider your 2.0 much of an improvement. (Significant changes always bear the risk of significant loss in quality.) The price tag coming with your 2.0 might outweight the benefits.

Anyway, people might expect you to continue the 1.x product line for some time, not only with critical patches but some feature back-porting too. The more complex your product, the more seldom should you do a major release.

9 License