PERFORCE
Version everything.

High-Level Best Practices for Software Version Management

Abstract

When deploying software version management and version control tools, implementers sometimes focus on perfecting fine-grained activities, while unwittingly carrying forward poor, large-scale practices from their previous jobs or previous tools. The result is a well-executed blunder. This paper promotes some high-level best practices that reflect our experiences with software version management.

Table of Contents

INTRODUCTION

"A tool is only as good as you use it," the saying goes. As providers of software version management tools and as consultants to software companies, we are often asked for sound advice on software version management best practices—that is, how to deploy software version management software to the maximum advantage. Answering these requests has created a bounty of direct and indirect version management experience from which to draw. The direct experience comes from having been developers and codeline managers ourselves; the indirect experience comes from customer reports of successes and failures with our product (Perforce) and other version control tools.

The following sections cover lists six general topics of software version management deployment and some coarse-grained best practices within each of those areas.

1. THE WORKSPACE

The workspace is where developers edit source files, build the software components they're working on, and test and debug what they've built. Most version control or software version management systems have some notion of a workspace; sometimes they are called "views", as in ClearCase and Perforce. In distributed version control systems (DVCSs) like Git, an entire repository acts as a workspace. Changes to managed repository files begin as changes to files in a workspace, or when DVCSs push, pull, and merge entire repositories.

The best practices for workspaces include:

- **Don't share workspaces.** A workspace should have a single purpose, such as an edit/build/test area for a single developer, or a build/test/release area for a product release. Sharing workspaces confuses people, just as sharing a desk does. Furthermore, sharing workspaces compromises the version management systems' ability to track activity by user or task. Workspaces and the disk space they occupy are cheap; don't waste time trying to conserve them.
- **Stay inside of managed workspaces.** Your version control system can only track work in progress when it takes place within managed workspaces. Users working outside of workspaces are beached; there's a river of information flowing past and they're not part of it. For instance, version management systems generally use workspaces to facilitate some of the communication among developers working on related tasks. You can see what is happening in others' workspaces, and they can see what's going on in yours. If you need to be absent for an emergency, your properly managed workspace may be all you can leave behind. Use proper workspaces.
- **Don't use jello views.** A file in your workspace should not change unless you explicitly cause the change. A "jello view" is a workspace where file changes are caused by external events beyond your control. A typical example of a jello view is a workspace built upon a tree of symbolic links to files in another workspace—when the underlying files are updated, your workspace files change. Jello views are a source of chaos in software development. Debug symbols in executables don't match the source files, mysterious recompilations occur in supposedly trivial rebuilds, and debugging cycles never converge—these are just some of the problems. Keep your workspaces firm and stable by setting them up so that users have control over when their files change.
- **Stay in sync with the codeline.** As a developer, the quality of your work depends on how well it meshes with other peoples' work. As changes are checked into the codeline by others, you should update your workspace and integrate those changes with your own changes.
  As a software version management engineer, it's good to make sure this workspace update operation is straightforward and unencumbered with tricky or time-consuming procedures. If developers find it fairly painless to update their workspaces, they'll do it more frequently and integration problems won't pile up at project deadlines.
- **Check in often.** Integrating your development work with other peoples' work also requires you to check in your changes as soon as they are ready. Once you've finished a development task, check in your changed files so that your work is available to others.
  Again, as a software version management engineer, you should set up procedures that encourage frequent check-ins. Don't implement unduly arduous validation procedures, and don't freeze codelines (see Child Codelines, below). Short freezes are bearable, but long freezes compromise productivity. Much productivity can be wasted waiting for the right day (or week, or month) to submit changes.
- **Use task branching.** In contrast to the long, public life of codelines, task branches generally support short-lived, private tasks, such as hot fixes or local feature development. Their primary task is to help ready a set of changes for propagation into the main codebase. Once the contribution has been delivered, they can be deleted or forgotten. Task branching brings the power of versioning to your local workspace while keeping your main codebase clear from metadata clutter.

2. THE CODELINE

In this context, the codeline is the set of source files required to produce your software. Codelines are branched into child codelines to support specific variants of the codeline, such as development and release codelines.

The best practices with regard to codelines are:

- **Have a mainline.** A "mainline," or "trunk," is the branch of a codeline that evolves forever. A mainline provides an ultimate destination for almost all changes—both maintenance fixes and new features—and represents the primary, linear evolution of a software product. Release codelines and development codelines are branched from the mainline, and work that occurs in branches is propagated back to the mainline.

Figure 1 shows a mainline (called "main"), from which several release lines ("ver1", "ver2", and "ver3") and feature development lines ("projA", "projB", and "projC") have been branched. Developers work in the mainline or in a feature development line. The release lines are reserved for testing and critical fixes, and are insulated from the hubbub of development. Eventually all changes submitted to the release lines and the feature development lines get merged into the mainline.

An adverse approach is to "promote" codelines: for example, to promote a development codeline to a release codeline, and branch off a new development codeline. Figure 2 shows a development codeline promoted to a release codeline ("ver1") and branched into another development codeline ("projA"). Each release codeline starts out as a development codeline, and development moves from codeline to codeline.

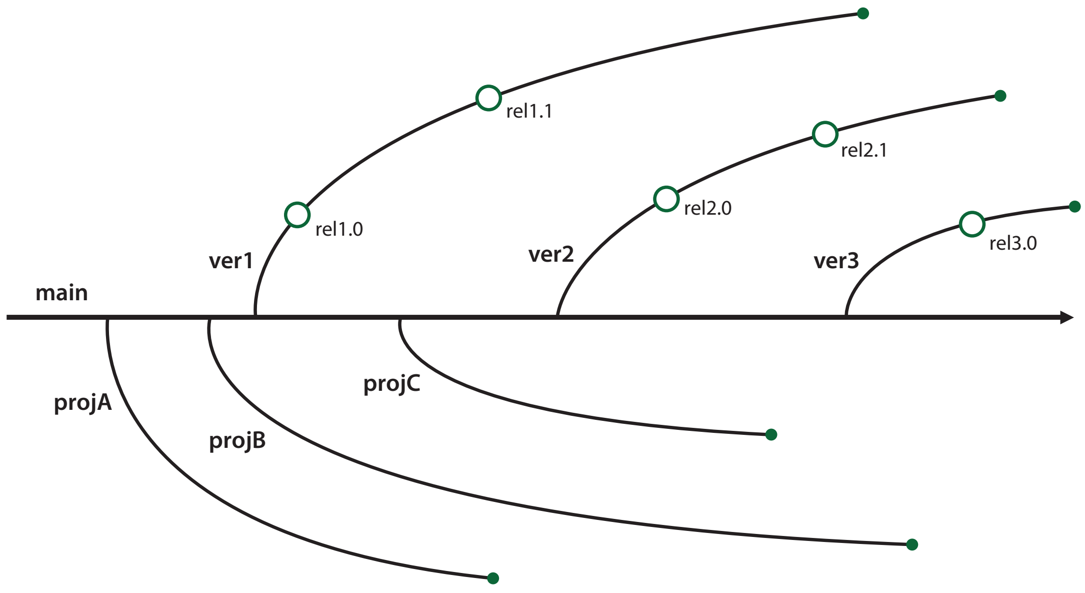

Figure 1: The Mainline Model

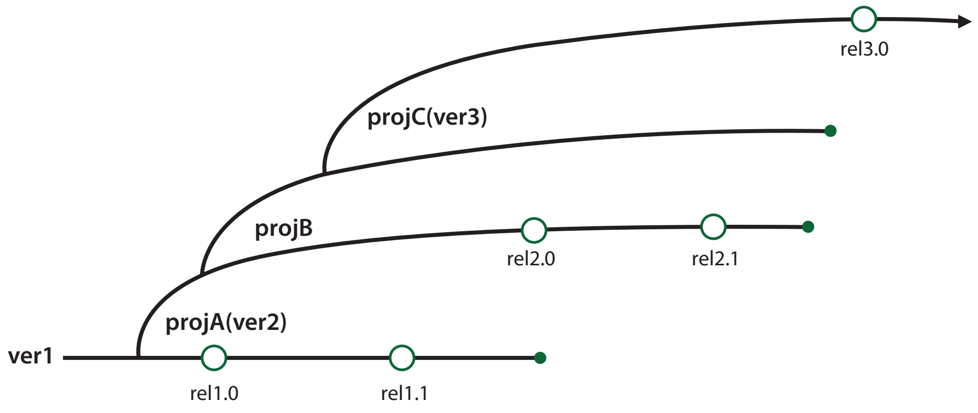

Figure 2: The Promotion Model

The promotion scheme suffers from two significant drawbacks: (1) it requires the policy of a codeline to change, which is never easy to communicate to everyone; (2) it requires developers to relocate their work in progress to another codeline, which is error-prone and time-consuming. With the promotional model, 90% of legacy SCM "process" is spent enforcing codeline promotion to compensate for the lack of a mainline.

Process is streamlined and simplified when you use a mainline model. With a mainline, contributors' workspaces and environments are stable for the duration of their tasks at hand, and no additional administrative overhead is incurred as software products move forward to maturity.

- **Give each codeline a policy.** A codeline policy specifies the fair use and permissible check-ins for the codeline, and is the essential user's manual for software version management codeline control. For example, the policy of a development codeline should state that it isn't for release; likewise, the policy of a release codeline should limit changes to approved bug fixes[1]. The policy can also describe how to document changes being checked in, what review is needed, what testing is required, and the expectations of codeline stability after check-ins. A policy is a critical component for a documented, enforceable software development process, and a codeline without a policy, from a software version management point of view, is out of control.
- **Give each codeline an owner.** Having defined a policy for a codeline, you'll soon encounter special cases where the policy is inapplicable or ambiguous. Developers facing these ambiguities will turn to the person in charge of the codeline for workarounds. When no one is in charge, developers tend to enact their own workarounds without documenting them. Or they simply procrastinate because they don't have enough information about the codeline to come up with a reasonable workaround. You can avoid this trap by appointing someone to own the codeline, and to shepherd it through its useful life. With this broader objective, the codeline owner can smooth the ride over rough spots in software development by advising developers on policy exceptions and documenting them.

3. CHILD CODELINES

The mainline is typically branched into child codelines specialized to support a variety of different tasks. The collection and relationships of the mainline and its child codelines can be visualized on a flow graph diagram.

1 *Some sensible codeline policies:* Development codeline: interim code changes may be checked in; affected components must be buildable. Release codeline: software must build and pass regression tests before check-in; check-ins limited to bug fixes; no new features or functionality may be checked in; after check-in, branch is frozen until entire QA cycle is completed. Mainline: all components must compile and link, and pass regression tests; completed, tested new features may be checked in.

Here are best practices for organizing and creating new child codelines:

- **Organize codelines according to the Tofu Scale.** Much like tofu in the supermarket that comes in different measures of firmness, codelines also fall somewhere on the firmness scale of increasing stability. Release codelines appear above the mainline and seek minimal change. Development codelines are arranged below the mainline and seek additional change.

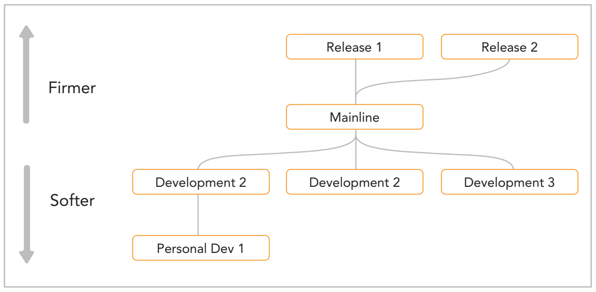

Figure 3: The Tofu Scale for codeline firmness.

- **Don't copy when you mean to branch.** An alternative to using your software version management tool's branching mechanism is to copy a set of source files from one codeline and check them in to another as new files. Don't think that you can avoid the costs of branching by simply copying. Copying incurs all the headaches of branching—additional entities and increased complexity—but without the benefit of your software version management system's codeline support. Don't be fooled: even "read-only" copies shipped off to another development group "for reference only" often return with changes made. Use your software version management system to make a new codeline when you spin off parts or all of an existing codeline.
- **Branch on incompatible policy.** There is one simple rule to determine if a codeline should be branched: it should be branched when its users need different check-in policies. For example, a product release group may need a check-in policy that enforces rigorous testing, whereas a development team may need a policy that allows frequent check-ins of partially tested changes. This policy divergence calls for a codeline branch. When one development group doesn't wish to see another development group's changes, that is also a form of incompatible policy: each group should have its own codeline.
- **Branch late.** To minimize the number of changes that need to be propagated from one branch to another, put off creating a new codeline as long as possible. For example, if the mainline contains all the new features ready for a release, do as much testing and bug fixing in it as you can before creating a release codeline. Every bug fixed in the mainline before the release

branch is created is one less change needing propagation between codelines.

- **Branch instead of freeze.** On the other hand, if testing requires freezing a codeline, developers who have pending changes will have to sit on their changes until the testing is complete. If this is the case, branch the codeline early enough so that developers can check in and get on with their work.

4. CHANGE PROPAGATION

Branching codelines creates the task of propagating file changes between codelines. The propagation of changes throughout a flow graph is known as the flow of change. Here are some things you can do to keep it manageable.

- **Use the baseline protocol.** Stabilizing change flows continually to softer codelines. Changes from the softer codelines flow up to the baseline at points of completion. Changes do not flow up from the baseline.
- **Propagate early and often.** When it's feasible to propagate a change from one codeline to another, do it sooner rather than later. Postponed and batched change propagations can result in stunningly complex file merges.
- **Merge down, copy up.** Create safer merges by merging changes down to the softer codeline, and copying up to a firmer codeline. The softer codeline can better accommodate the risk of merging because (1) instability is more acceptable there and (2) code in the softer codeline is further from the release date. A typical workflow would be:
    1. Merge from the baseline to the softer codeline.
    2. Test the merged result.
    3. Copy from the softer codeline to the baseline, now that we are ensured a successful merge.

Merge down, copy up minimizes the risk of introducing destabilizing changes into the firmer/more stable baseline.

- **Choose the correct codeline for original changes.** You may have dozens of codelines in your codeline graph. How do you decide where to make an original change? Make original changes intended to stabilize a codeline above the mainline, and make all other changes below the mainline.
- **Get the right person to do the merge.** The burden of change propagation can be lightened by assigning the responsibility to the engineer best prepared to resolve file conflicts. Changes can be propagated by (a) the owner of the target files, (b) the person who made the original changes, or (c) someone else. Either (a) or (b) will do a better job than (c).

5. BUILDS

A build is the business of constructing usable software from original source files. Builds are more manageable and less prone to problems when a few key practices are observed:

- **Source + tools = product.** The only ingredients in a build should be source files and the tools they are put into. Memorized procedures and yellow stickies have no place in this equation. Given the same source files and build tools, the resulting product should always be the same. If you have manual setup procedures, automate them in scripts. If you have manual setup steps, document them in build instructions. And document all tool specifications, including operating systems, compilers, include files, link libraries, make programs, virtual machines, and executable paths. Version all of the resulting materials.
- **Version all original source.** When software can't be reliably reproduced from the same ingredients, chances are the ingredient list is incomplete. Frequently overlooked ingredients are makefiles, setup scripts, build scripts, build instructions, and tool specifications. All of these are the source you build with. Remember: source + tools = product.
- **Separate built objects from original source.** Organize your builds so that the directories containing original source files are not polluted by built objects. Original source files are those you create "from an original thought process" with a text editor, an application generator, or any other interactive tool. Built objects are all the files that get created during your build process, including generated source files. Built objects should not go into your source code directories. Instead, build them into a directory tree of their own. This separation allows you to limit the scope of software version management-managed directories to those that contain only source. It also groups the files that tend to be large and/or expendable into one location, and simplifies disk space management for builds.
- **Use common build environments.** Developers, test engineers, and release engineers should all use or have access to identical build environments. Much time is wasted when a developer cannot reproduce a problem found in testing, or when the released product varies from what was tested. Remember: source + tools = product.
- **Build often.** Continuous integration (CI) with automated regression testing safeguards your development through continuous response to the greatest source of defects: new check-ins. You will
    1. Detect build failures created by check-ins (don't break the build).
    2. Find product functional problems created by check-ins (don't break the product).

A beneficial side effect is that CI and testing produce link libraries and other built objects that developers can use. Every codeline should be subject to regular, frequent, and complete builds and regression testing, even when product release is in the distant future.

- **Keep build logs and build outputs.** For any built object you produce, you should be able to look up the exact operations (e.g., complete compiler flag and link command text) that produced the last known good version of it. Archive build outputs and logs, including source file versions (e.g., a label), tool and OS version info, virtual machine configuration, compiler outputs, intermediate files, built objects, and test results, for future reference. As large software projects evolve, components are handed off from one group to another, and the receiving group may not be in a position to begin builds of new components immediately. When they do begin to build new components, they will need access to previous build logs in order to diagnose the integration problems they encounter.

6. PROCESS

It would take an entire paper, or several papers, to explore the full scope of software version management process design and implementation, and many such papers have already been written. Furthermore, your shop has specific objectives and requirements that will be reflected in the process you implement, and we do not presume to know what those are. In our experience, however, some process concepts are key to any version management implementation:

- **Track change sets.** Even though each file in a codeline has its revision history, each revision in its history is only useful in the context of a set of related files. The question "What other source files were changed along with this particular change to `foo.c`?" can't be answered unless you track change sets, or sets of files related by a logical change. Change sets, not individual file changes, are the visible manifestation of software development.
- **Track the flow of change.** One clear benefit of tracking change sets is that it becomes very easy to propagate logical changes (e.g., bug fixes) from one codeline to another. However, it's not enough to simply propagate change sets across codelines; you must keep track of which change sets have been propagated, which propagations are pending, and which codeline branches are likely donors or recipients of propagations. Otherwise you'll never be able to answer the question "Is the fix for bug X in the release Y codeline?" You should never have to resort to "diffing" files to figure out if a change set has been propagated between codelines.
- **Distinguish change requests from change sets.** "What to do" and "what was done" are different data entities. For example, a bug report is a "what to do" entity and a bug fix is a "what was done" entity. Your software version management process should distinguish between the two, because in fact there can be a one-to-many relationship between change requests and change sets.
- **Give everything an owner.** Every process, policy, document, product, component, codeline, branch, and task in your software version management system should have an owner. Owners give life to these entities by representing them; an entity with an owner can grow and mature. Ownerless entities are like obstacles in an ant trail—the ants simply march around them.
- **Use living documents.** The policies and procedures you implement should be described in living documents; that is, your process documentation should be as readily available and as subject to update as your managed source code. Documents that aren't accessible are useless; documents that aren't updateable are nearly so. Process documents should be accessible from all of your development environments: at your own workstation, at someone else's workstation, and from your machine at home. And process documents should be easily updateable, and updates should be immediately available.
- **Learn and use integrations to supporting tools.** Software version management often sits within a dizzying array of supporting tools in an application lifecycle management (ALM) stack. Learn all the options for integrating your software version management system into tool sets that manage requirements, features, projects, tests, releases, and more.

As a software version management engineer, it's also essential to master how your system integrates with Agile development tools and processes. CI, automated testing, work flow, defect tracking, and code review tools are all likely integrations into a software version management system.

Remember, software version management is often one part of a larger ALM and/or Agile environment. Be sure you understand all the integrations available for how your software version management tool can support this larger environment.

7. CONCLUSION

Best practices in version management, like best practices anywhere, always seem obvious once you've used them. The practices discussed in this paper have worked well for us, but we recognize that no single, short document can contain them all. So we have presented the practices that offer the greatest return and yet seem to be violated more often than not.

We welcome the opportunity to improve this document, and solicit both challenges to the above practices as well as the additions of new ones.

8. REFERENCES

Berczuk, Steve. *Configuration Management Patterns*, Addison-Wesley, 1997.

Compton, Stephen B. *Configuration Management for Software*, VNR Computer Library, Van Nostrand Reinhold, 1993.

Continuus Software Corp., “Work Area Management”, Continuus/CM: Change Management for Software Development [PDF].

Dart, Susan. Spectrum of Functionality in Configuration Management Systems [PDF], Software Engineering Institute, 1990.

Jameson, Kevin. Multi Platform Code Management, O’Reilly & Associates, 1994.

Linenbach, Terris. *Programmers’ Canvas: A Pattern for Source Code Management*, Raven Publishing, 1996.

Lyon, David D. *Practical CM*, Raven Publishing, 1997.

McConnell, Steve. Best Practices: Daily Build and Smoke Test, *IEEE Software*, Vol. 13, No. 4, July 1996.

Van der Hoek, Andre, Hall, Richard S., Heimbigner, Dennis, and Wolf, Alexander L., Software Release Management, Proceedings of the 6th European Software Engineering Conference, Zurich, Switzerland, 1997.

Wingerd, Laura. The Flow of Change [PDF], Perforce Software, Inc., 2005.

Wingerd, Laura. *Practical Perforce*, O’Reilly & Associates, 2005.

Updated January 2012, April 2012, November 2012 James Creasy

MORE ABOUT HOW PERFORCE SUPPORTS THESE BEST PRACTICES

Workspaces

Stream views, Perforce Sandbox, Git Fusion

Use local task branching

Perforce Sandbox:
perforce.com/sites/default/files/pdf/p4sandbox-product-brief.pdf

Git Fusion
perforce.com/sites/default/files/perforce-git-fusion-product-brief.pdf

Codelines

Streams implement codelines

Perforce Streams:
perforce.com/product/product_features/perforce_streams

Child Codelines

Streams graph, default codeline policies, tofu scale

Use the Tofu Scale:
oreilly.com/catalog/practicalperforce/chapter/ch07.pdf

Change Propagation

Streams graph, visual tools, default codeline policies, enforcement of rules of Flow of Change

Agile Flow of Change:
info.perforce.com/Agile-Flow-of-Change-WP-Offer.html

Builds

Integrations with ALM:
perforce.com/product/integrations/thirdparty_software_integrations

Agile tools:
perforce.com/product/product_features/always_agile

Process

Perforce Software Version Management:
perforce.com/products/perforce

p e r f o r c e . c o m

PERFORCE
Version everything.

North America
Perforce Software Inc.
2320 Blanding Ave
Alameda, CA 94501
USA
Phone: +1 510.864.7400
info@perforce.com

Europe
Perforce Software UK Ltd.
West Forest Gate
Wellington Road
Wokingham
Berkshire RG40 2AT
UK
Phone: +44 (0) 845 345 0116
uk@perforce.com

Australia
Perforce Software Pty. Ltd.
Suite 3, Level 10
221 Miller Street
North Sydney
NSW 2060
AUSTRALIA
Phone: +61 (0)2 8912-4600
au@perforce.com