inedo

White Paper

Source Control Done Right:

Utilizing a Devops Methodology to Manage Your Source Control

Contents:

As professional philosophies shift towards more Devops mindsets, the technology and methods that are used have to shift with them. A great example of this is the way we deal with source control. Today's training in source control mostly revolves around the systems used, rather than the concepts that the systems use. It's common for an introduction to source control to be simple instructions on logging in, and file access, and nothing more. By and large this is an acceptable practice, as most developers are familiar with many of the key concepts that govern source control. Understanding the nature of files and directories makes an easy bridge for understanding the concepts of checking-in and checking-out; eventually even using repositories, merging, and committing become second-nature.

This simple understanding of source control allows for navigation of the system, but at the same time there is much more to the process below the surface. This especially holds true with the latest breed of distributed version control systems. While it's true that certain operations are now even easier to accomplish, not properly understanding when to and when not to use them means that it's that much easier to make mistakes. By better understanding source control, a team can dramatically increase its performance, as well as avoid the time consuming, costly, and common pitfalls that come with poor source control practices.

The Basics of Source Control

Any explanation of source control can be crippled without establishing a baseline of knowledge, since modern professionals approach source control from a variety of levels of expertise. It follows that in order to gain, or communicate a properly informed prospective on source control it is necessary to reintroduce the subject from the ground up. This means reexamining the basics, even in their most rudimentary forms.

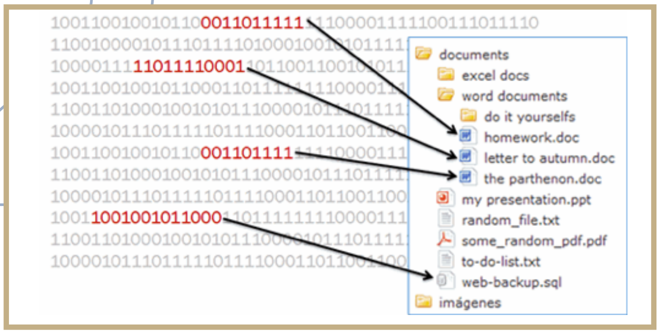
DIMENSION 1: BITS

```
1001100100101100011011111111100001111100111011110
1100110100010010101110000101110111111011110001101
1000010111011111101111000110110011001001011000110
1001100100101100011011111111100001111100111011110
1100110100010010101110000101110111111011110001101
1000010111011111101111000110110011001001011000110
1001100100101100011011111111100001111100111011110
1100110100010010101110000101110111111011110001101
1000010111011111101111000110110011001001011000110
1001100100101100011011111111100001111100111011110
1100110100010010101110000101110111111011110001101
1000010111011111101111000110110011001001011000110
1001100100101100011011111111100001111100111011110
1100110100010010101110000101110111111011110001101
1000010111011111101111000110110011001001011000110
1001100100101100011011111111100001111100111011110
1100110100010010101110000101110111111011110001101
1000010111011111101111000110110011001001011000110
1001100100101100011011111111100001111100111011110
1100110100010010101110000101110111111011110001101
```

Stripped to its basest elements all software, all code break down to bits. A stream of 1s and 0s, and in this context they are unfathomably large and for human purposes virtually endless. Whether it's the 16,000,000 on a 3.5" floppy disk or the 4,000,000,000,000 on a desktop hard drive, bits represent a line. Bits are the first dimension and, in and of themselves, they are entirely meaningless.

To introduce meaning there is a need for additional context, additional dimensions. Next we'll discuss file systems.

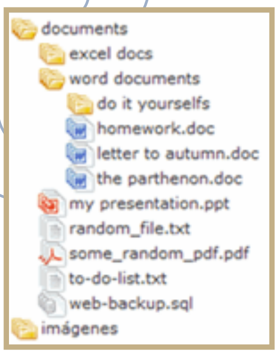
DIMENSION 2: THE FILE SYSTEM



The second dimension is the file system. The file system very simply intersects this near-infinite line of bits to create a path. By creating paths, the file system groups and defines strings of bits in order to give them meaning and functionality. An example could be explained something like this; bits 5,883,736 through 6,269,928 are grouped into a path which represents C:\Pictures\Kitty.jpg. The following 4,096 bits assigned metadata understood by the file system, such as "date created" and "last modified by user."

The second dimension is the way the most users experience their software. But there is more depth added when considering the third dimension, the delta.

DIMENSION 3: THE DELTA



A delta represents the differences between sequential data rather than entire files. Delta revision affords the user the ability to revert to any state at any time. When taken as a whole this three dimensional system is known as revision control.

Revision Control Vs. Source Control

There's one key distinction between revision control and source control: the latter is a specific type of the former. With that, we can arrive at fairly broad yet concise definition.

> Source Control [sawrs kuhn-trohl]: change management for source code; noun

This definition is specifically pointed; source control is for source code. Source control systems can be and are commonly used for a variety of situations that fall outside of these definitions. While small infractions can be harmless enough, learning to rely on a source control system as a golden repository can have far reaching implications and result in a number of unnecessary problems.

SOURCE CONTROL OPERATIONS

There are dozens of different source control systems on the market, but all must implement at least two core operations.

- GET: retrieve a specific set of changes (files, directories, etc) from the repository
- COMMIT: add a specific set of changes to the repository

Many source control systems will also implement a locking mechanism that prevents a Commit against a specific set of files. This yields two additional operations:

- CHECK-OUT: Locking a specific set of files and then Getting the latest revision of those files
- CHECK-IN: Unlocking a specific set of locked files and then Committing changes against those files

Avoiding Source Control Abuse

While there are a near infinite number of ways to improperly use a source control system, the top three misuses tend to be:

- **Document management (such as SVN For HR, or even developer documentation)**
- **Storing compiled binaries**
- **Littered with gigabytes of video and picture files**

Minor misuse of a source control system is a small problem, and generally one not worth addressing. The true problem is the ethic that arises from repeated misuse, it is important to remember what source control is and is not supposed to be. It is not a document manager, an artifact (build output) library, a backup solution, a configuration management tool, etc. Source control is for source code.

This notion of locking files has created two distinctly different philosophies for maintaining code:

- CHECK-OUT + EDIT + CHECK-IN: before making any changes to a file, a developer must check-out the file, and while it's checked-out, no other developers can make changes against it until the file is checked-in
- EDIT + GET/MERGE + COMMIT: before committing a change to the repository, a developer must get the latest version of the file and merge any changes made since the last time he retrieved it.

The debate over which practice is better has become mainly partisan: for every advantage one technique holds over the other, there's an equal disadvantage. Rather than simply choosing a side and always drawing the same conclusion, most projects are will be better served by a subjective approach. Some projects are going to be more suited for Edit + Get/Merge + Commit, while others work best for Check-out + Edit + Check-in. It's really about team preference.

The Fourth Dimension

So far the distinctions drawn between revision control and source control have been semantic. The concepts of organizing bits into file (second dimension) and maintaining changes in these bits (third dimension) apply to data of all types.

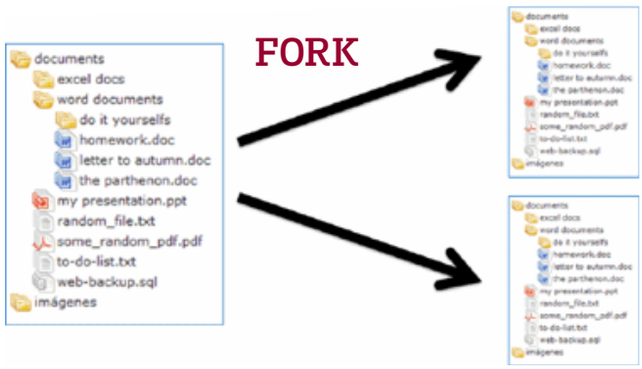
Source code however, is a special kind of data: it represents a codebase, or the living blueprint for an application that's maintained by a team of developers. It's this key distinction that makes source control a special case of revision control, and it becomes necessary to include an additional dimension for managing changes in source code.

DIMENSION 4: BYTES + PATH + CHANGES + ALTERNATES

While a three-dimensional system manages a serial set of changes to a set of files, the fourth dimension enables parallel changes. The same codebase can receive multiple changes without having any impact whatsoever on another.

FORK OPERATIONS

The quickest way to access the fourth dimension is through an operation called Fork.



A fork copies a three-dimensional repository, creating two equal but distinct repositories. A commit performed against one repository has no impact on the other, which means the codebases contained within will become more and more different, and eventually evolve into different applications altogether.

While forking is an important aspect of open source projects, the operation offers little benefit for a team maintaining a single application. For this type of development, the parallel repositories must come together at some point through merging.

MERGING

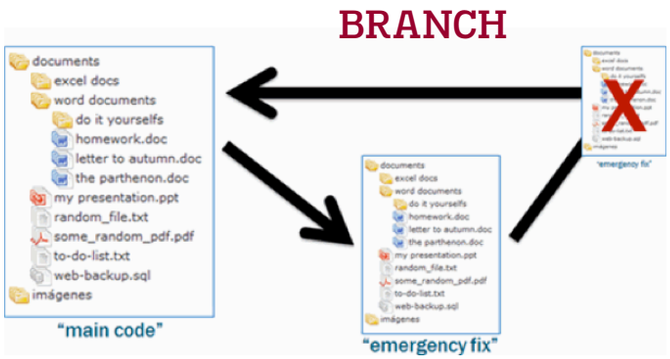
Merging is a concept that is often, but unnecessarily discussed in an over complicated fashion. When we're looking at things from a fourth dimensional point of view, there are three lower dimensions to merge:

- BITS: a relatively easy task for text files; lines can be added, changed, or deleted, and a smart diff program will help identify these for human inspection
- PATHS: files can be moved, renamed, copied, deleted, etc., ideally with the same smart diff program and same human inspection
- DELTAS: these are the least important to merge, and in some cases source control systems simply won't merge them. They are only a history of changes, so they can get automatically shuffled together, not unlike a deck of cards

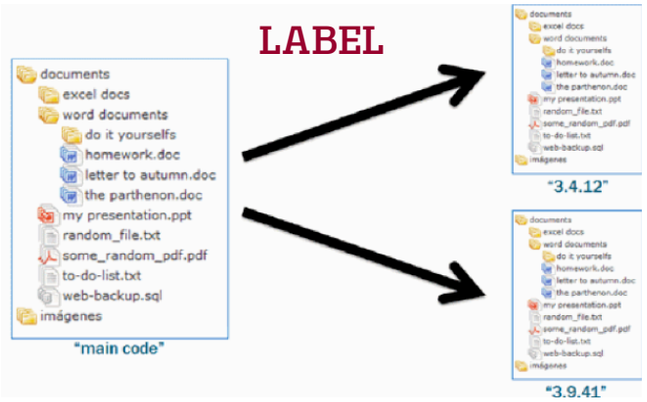
No matter how smart a diff program may be, if there have been changes in both repositories, merging should always be verified by a human. Just because the diff program reported "no conflicts" and was able to add a line here, and delete a file there, that doesn't mean the resulting codebase will work properly or even compile.

WHY MERGE?

While understanding how merging functions are helpful, it doesn't dictate why parallel repositories needing merging in the first place. The most common reason for this is one of the special types of fork operations called a branch.

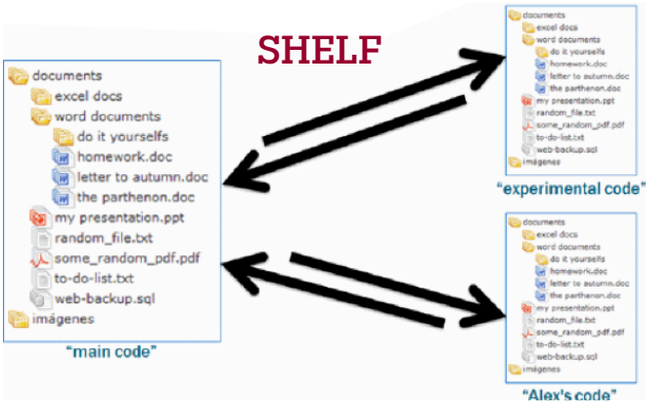


Though its name doesn't quite imply it, a branch is simply a temporary fork. Changes made to a branch are generally merged back in, eventually the branch is closed. If it didn't it wouldn't be temporary, and would be a basic fork.



A label is nothing more than a read-only fork. Depending on the source control system labels are given different names; tags, snapshots, checkpoints, etc. The concept is always the same: it's a convenient way to label the codebase at a certain point in time. It's almost like creating a backup or a ZIP file of a directory, except it has a history of all of the changes as well.

The third, least used, special type of fork operation is called shelf. It's an unofficial in progress, or working, fork that's used to manage or share incomplete changes. Changes committed to a shelf may be merged into the main codebase, or vice versa.



Some organizations give each developer their own shelf so that he can easily share/merge changes with other developers. Others teams may set-up shelves as some sort of quality gateway; a development shelf, an integration shelf, etc. This is to ensure that only approved changes find their way in real application's codebase. There's only one rule about shelves: they should never be used to create builds that are intended for production deployment.

In total, there are a minimum of four forth dimensional operations:

1. FORK: a parallel repository
2. BRANCH: a temporary fork
3. LABEL: a permanent, read-only fork
4. SHELF: a "personal" or "working" fork

```
/mainline
 /labels
  /2.5.8
  /2.6.1
  /2.6.2
  /2.6.3
  /2.7.1
 /branches
  /2.6
  /2.7
 /shelves
  /stages
   /development
   /integration
  /users
   /alexp
   /paula
  /groups
   /offshore
   /vendor993
```

The most basic implementation is to simply support a "fork" operation and rely on the end-user to maintain sub-repositories. Subversion (and others) create default sub-repositories called /trunk, /tags, and /branches, but the convention is not concrete and the same implementation could just as easily use names like the examples seen in the sidebar.

Without proper oversight, these forks could take up an excess of of disk space, but the same forks could use an indexing system and require hardly any resources at all. Other source control systems will not only optimize the implementations, but they will hide implementation details and give them special names:

- ACCUREV branches are called "streams"
- CA HARVEST: labels are called snapshots
- SOURCE SAFE: labeling is its own operation
- TEAM FOUNDATION SERVER: shelving is its own operation

How these concepts are implemented behind the scenes is less essential than understanding how to use them.

Source Control and Application Development

Understanding the differences between builds and releases are as fundamental to application development as debits and credits are to accounting, so as a quick refresher:

- RELEASE: represents a planned set of changes to an application. The release could be planned far in advance and require tens of thousands of developer hours to implement, or it could be a single line change rushed to production in an emergency.
- BUILD: represents an attempt at implementing the requirements of a particular release. It also serves as a snapshot of an application's codebase that is tested throughout the application's environments before going into being "released" (i.e. deployed to production, shipped to the customer, etc.)

Builds are immutable, which is aligns them well with the special fork function label. Creating a new Build should always correspond to creating a new label. Not only will that help enforce the code immutability, but by knowing which build is in which environment, allows for a better understanding of what code is where.

BRANCHING STRATEGIES

A branch is a temporary creation, so it isolates a particular set of changes to the application's codebase. That's basically the definition of a release, which is why the concepts of branching and releases are so intertwined. Branches are a way to isolate the changes in one release from another.

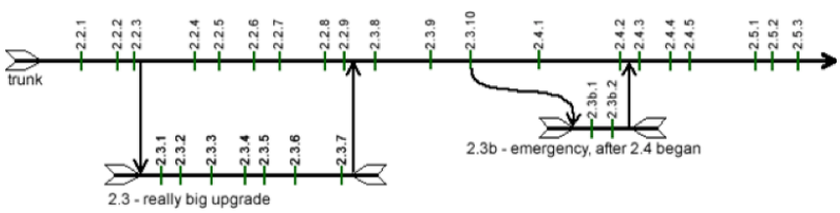
This function is easy to overuse. There are two proper uses for branching; branching by rule and branching by exception. Both are related to isolating changes in releases, and because of this, a branch should always be identified by its corresponding release number.

This rule comes from a simple place, there's only one trunk, and the code under that trunk is either what's in the production now (the latest deployed release)or what will be in he production later (the next planned release). Which means that there is either always branching or only branching sometimes.

BRANCHING BY EXCEPTION

This is by far the easiest strategy to follow. In fact, teams that have never branched anything before, then still can claim to follows this strategy. As the name implies, a branch is only created for exceptional releases, and the definition of exceptional is defined entirely by the development team.

Exceptional could mean "an emergency, one-line fix that needs to be rushed to production" or it could mean "some experimental changes to 3.0." The main tenet is that you generally use the trunk for developing code and creating builds that will be tested and deployed.



The diagram shows how this strategy works in practice. If we follow the trunk line, which represents a series of changes over time, we'll see that the story starts with the development being in the midst of Release 2.2. The green, vertical hashes represent labels, corresponding to to a codebase with a build number starting at 2.2.1 and ascending whenever a new build was created.

At some point, work on Release 2.3 was done in parallel with the codebase. In this case this was due to an exceptionally big release, so code was forked to a branch called 2.3 and created builds against both releases. Once Release 2.2 was deployed, the changes were merged from Release 2.3 into the trunk, deleted the branch, and continued creating and testing builds. Eventually, Build 2.3.10 was deemed the a higher quality, so they commenced work on Release 2.4 and even created a preview build, 2.4.1.

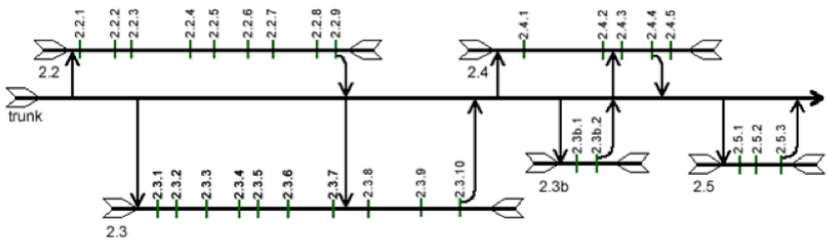
Following the trunk line it becomes apparent that during work on 2.4, the business delivered some bad news, a serious mistake in production, Build 2.3.10, which needed to be addressed immediately. By forking the codebase, using the label instead of using the trunk, which was undergoing work for Release 2.4, the emergency change, 2.3b, was realized pushed through to production after two builds, and finalized by merging the fix into Release 2.4, and deleting the branch.

BRANCHING BY RULE

If every release seems to be exceptional, or if there's a real need to isolate changes between releases, branching by rule may be worth exploring. In this case, every release gets its own branch, and changes are never committed to the trunk. Since the trunk represents the deployed codebase, once a release has been deployed, its changes should immediately be applied, not merged, to trunk, and the branch should be

collapsed.

This doesn't mean that merging doesn't need to happen. On the contrary, there needs to be extreme caution exercised hen working with parallel branches. As soon as a release is



deployed, the changes applied to trunk should be merged into all branches.

The diagram uses the same conventions as the branch by exception, and so needs less explanation than the last, but there are a few important things to take note.

- Trunk is never labeled; builds are never created from it, so there's no point in labeling
- A branch itself is never automatically merged into trunk; instead, the label that represents the deployed build (e.g. 2.3.10) is automatically merged
- Changes committed after a label (e.g. changes in Build 2.4.5 of Release 2.4) could be lost forever if they're not merged to another branch

```
/
 /DEV
  helloworld.c
  hiworld.c
 /QA
  helloworld.c
 /PROD
  helloworld.c
```

- When changes are automatically merged into trunk (e.g. Build 2.2.9), they're manually merged (to resolve conflicts, etc) into each open branch (Release 2.3)

BRANCHING BY SHELF

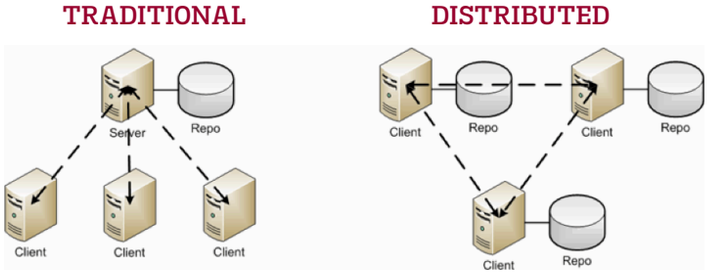
One of the reasons that shelves are the least common of the fork operations are that they're often misunderstood and misused as branches. Often branching can be misrepresented like in this sidebar.

There are multiple shelves, each designed to represent a certain stage of testing, and one for production. As features are tested and approved, the changes that represent those features are "promoted" to their appropriate environment through merging. When deploying to an environment, code is retrieved from the corresponding repository, compiled, and then installed.

This source control anti-pattern leads to any number of problems. Worse still, these problems grow exponentially and become exponentially difficult to solve as thecodebase grow.

The Distributed Revolution

Traditionally, source control systems have worked on the client/server model: developers perform various source control operations (get, commit, etc.) using a client installed on their local workstation, which then communicates and performs those operations against the server after some sort of security verification.

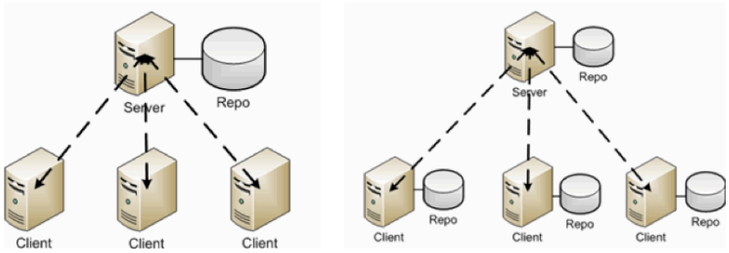


But distributed source control systems are quite a bit different.

There's no central server, every developer is the client, the server, and the repository. Source code changes are committed as per normal, but remain isolated unless a

developer shares those changes with another repository through push and pull operations. It's a paradigm shift, not unlike the BitTorrent shift in the file-sharing world.

The main problem with this shift is that applications, especially of the proprietary in-house variety, can't be developed peer-to-peer. An application needs to have a central codebase where its code is maintained and from which builds are created. This means that the traditional vs. distributed diagram should look more like this.



There's still one big difference: each client has their own fork of the repository. The obvious downside to this is that a repository can grow to be pretty big, and storing/transferring all that history to a local workstation may be time consuming on an initial load. But there's another subtle disadvantage. In order to add a change to the central repository, developers must perform two operations, a "commit" to his local repository, then a "push" to the central. This may not seem important, but for some developers, it's twice the reason to avoid source control altogether.

That's not to say that having a local repository is not advantageous, but many of the purported benefits- frequent commits, easy merging, sharing code, patching, etc.- can be achieved with individual shelves.

Of course, with all the new terminology and the learning curve of the paradigm shift, the notion of proper branching strategy is being left by the wayside. With the ease of forking, the simplicity of merging, and allure of pulling, the low hanging fruit of branching by shelf seems more and more appealing. Yet, as earlier stated, this methodology eventually creates more problems than it solves, and eventually costs more hours than it saves.

Distributed control versions are still useful, but in different arenas. In fact, it's become the de facto standard for open source projects, because open source projects are developed vastly differently than proprietary applications. Community support doesn't function the same for professional developers. They simply don't submit patches, hoping the business, or another developer will accept their fix to the bug. They are assigned the task of fixing the bug, and complete the task as part of their job.

Technique over Toolsets

The obvious conclusion to this is that most of the challenges in source control arise from methodology rather than poorly functioning source control systems. When used properly to manage source code changes; labeling for builds, branching by exception, etc. Even the poorest rated systems will far outperform an industry leader set-up with haphazard commits and pushes.

Understanding how to use source control, not just a specific source control system, will allow any team to grow and add value to their work. Source control systems shouldn't be platforms for niched partisan debates, or software brand loyalty. They are simply tools, tools the help to properly manage source code changes.

inedo

Inedo, founded in 2003, leads the charge for higher quality software and best practices within software development and deployment with its distinctive software product BuildMaster. Established as a boutique software development shop, Inedo saw and attacked the many challenges inhibiting small and medium size businesses from fully exploiting the latest software technologies. With over 16 years of experience in a variety of industry segments including banking, education, insurance, retail, healthcare, government and manufacturing, the Inedo team noticed that so many software development organizations failed, or were unable to fully exploit software development and ALM best practices. Consultative services were added to help these businesses optimize and streamline their software development processes and deliver higher quality software more quickly. Alex Papadimoulis, Inedo's founder, quickly realized that their extensive knowledgebase could and should be implemented as a software toolset that companies could use to automate and control their software development with increased control and transparency with lower risk and cost.

BuildMaster is the result.

Inedo, LLC
64 Front Street, 2nd Floor • Berea Ohio 44017
440.243.6737 • **inedo.com**