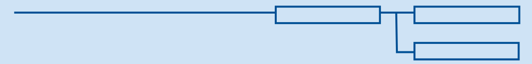
AccuRev



How To Know You've Outgrown Subversion

AccuRev

How To Know if You've Outgrown Subversion

Software Configuration Mangment (SCM) tools have a profound effect on the day-to-day life of a developer. These types of systems have either helped or hindered development teams deliver software. SCM systems are like the "hub" of a development team. It's where teams artifact important work, integrate changes, save important ideas and add features for customers. It's the center of our development universe!

Over the last few years, Subversion (SVN) has become a popular choice for some development teams. We could go as far as to say that SVN is a great basic SCM tool. It accomplished it's goal "CVS Done Right". But after teams have been using SVN for a certain amount of time, it's limitations become apparent.

Teams often struggle when they reach a certain size that's somewhere in the range of 20 developers. They will start to find signficant problems in the way that SVN handles branching, merging and integrating code changes. The core problem is SVN's architecture which is based on files, directories and whole baseline trunk level commits.

Software Development is About Innovation.... But How Do You Know if You've Outgrown Your Current Solution?

It's all about the developers. They need to be free to innovate and get changes out the door quickly and on time. But they can't if they are stifled by tools that get in the way. Tools need to be able to ENHANCE the software development process. Many people think that source control is just a place to checkin / checkout code; but it's more than that, it's where the software development process comes to life. If the SCM system isn't up to the task of a complex development process, developers can't innovate.

Sometimes it's hard to understand that you have a tooling problem, even if it's staring you in the face. Think of an old pair of trusty sneakers that you have at your house. We all have a pair. They are many years old, beat-up, dirty, torn… but we still keep them. Our feet hurt when we wear them, but for some reason, we refuse to get rid of these old sneakers. Until one day (usually after a sprained toe) we decide to buy a brand new pair and.. WOW our feet feel great. Why did I keep the other pair so long?

Software tools are often like this. There is an "if it ain't broke don't fix it" attitude. We often keep tools too long after their effective date. You'll hear it from your development team, moaning about the pains of merging code, switching workspaces, checking out … it's enough to make you cringe. But still we don't change.

This guide was written tto help you understand if you've outgrown your tools. Subversion is the sneaker, and collectively as a group, you and your team have a hard time recognizing when your feet hurt.

Sign#1: You Have A Complex Development Process

Modern software development practices are centered around delivering code quickly and getting rapid feedback. If you're doing Agile, XP or a hybrid approach, customers demand that you deliver changes immediately. The development process usually consists of different states as you deliver those changes.

The complexity lies within coordinating the workflow for code changes started with developers. If we think of the different states of our issues in a software project, they may go from "In Development" to "QA" and then move to "Production". This is a simple workflow that many Issue Tracking Systems (ITS) follow or that a Project Management tool will have.

The problem with SVN is that we continue to work with modern software development practices that push the limits of how quickly we can deliver value, but SVN supports a very slow traditional workflow. While it may be possible to create and enforce a process with SVN using branches and tags, it's not easy.

With SVN, creating branches is very simple. Branches are copies--there is no internal concept of a branch. Subversion does use “cheap copies” creating a new directory entry pointing to an existing tree instead of duplicating data. This limits the growth of the repository. Branches are typically created in a “branches” subdirectory of the project in the repository, individual file, subdirectory, or the entire source tree can be copied to a new branch. SVN does not provide any visibility into the overall branching structure, so teams typically must rely on naming conventions in order to keep track of the relationship between branches.

The repository browser displays the folder structure, but this only shows what branches exist, listing each as a folder in the branches directory. The revision graph will show the complete branching structure of an individual file or folder. There is no out-of-the box mechanism to understand the overall branching structure and history. Most of the time this is often written down on a whiteboard, or a diagram and sent out to the development team.

To manage the process, you'll have to support; maintenance releases, customer one-offs, parallel mainline efforts. Think of starting out with trunk. You will need to release 1.0, you'll need 2 tracks for the maintenance release, and a 2.0 release branch to work in parallel. Trunk will also always have to be stable for the next release.

To make this even more complex, you'll probably need to have team, or feature branches to support those types of projects that might not be immediately included into a release line.

In this diagram (Figure 1), you can see the amount of branching and merging that would have to be done just to manage a simple process. The remote teams will have to branch back and forth between all project teams, and some refactor efforts will be on a single branch that will have to merge with TRUNK.

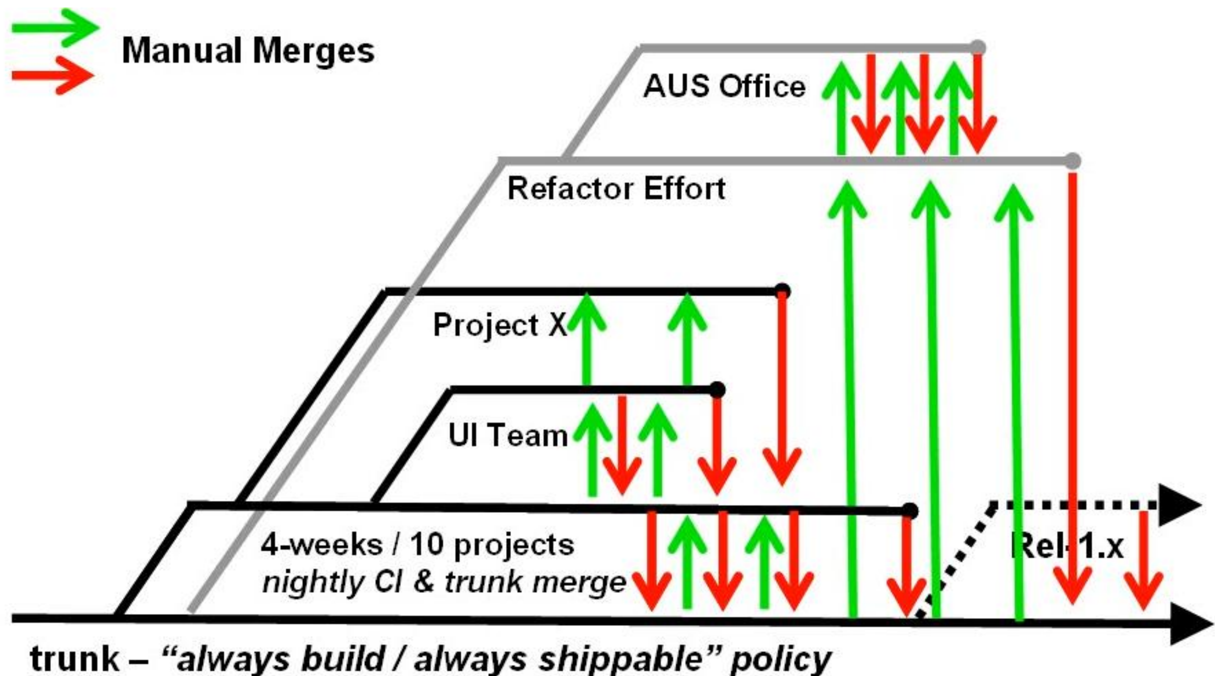


Figure 1

The problem with SVN is that branches are the only tools that are available to help manage a process. Branches themselves are simple and allow you to work in parallel to a certain point. But the dynamic nature of software development is not rigid in the way that branches are structured.

In order to work around this, you'll have to manage the process outside of SVN. This means that the process will either be documented in some sort of way (usually written on a whiteboard), or through spreadsheets and emails. Keeping track of this process is usually cumbersome and difficult to manage.

The truth is that the development process looks something more like this (fig2 below).

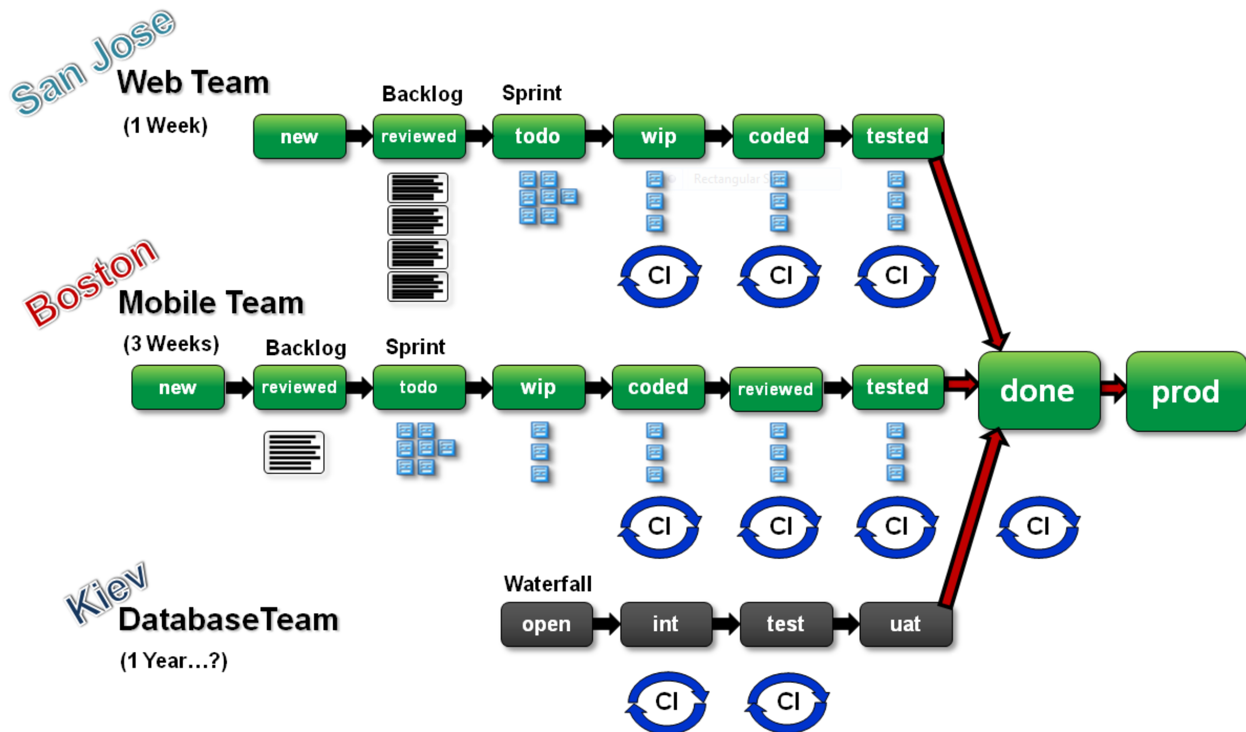


Figure 2

Think of all the different states your code goes through as it completes the development lifecycle. You'll have development code that isn't ready for Quality Assurance (QA), code that was tested by QA and ready to ship to User Acceptance Testing (UAT), then maybe share that code with other teams before finally shipping it to production. This process might be different depending on the methodologies you're using (like Agile or Waterfall), but the need to support those states remain the same.

The crutch of the process management in SVN is that it doesn't support these states. Code that might not be ready to test is still on the release line, or the project branch where it orginially resided.

With SVN branching structures; you are prone to polluting your mainline. This is mixing code in different states to TRUNK or your project branches, polluting those codebases with unfinished work. Simply put, SVN manages the isolation of code in branches, but doesn't help with a complex development process.

Sign#2 You Spend too much time with Branching and Merging

Developers don't want to merge code. Nobody wants to merge code. Development is about problem solving and innovation. Merging is a disruption to the brilliant work that developers accomplish on a daily basis.

Remember when you had to merge your current release branch to TRUNK? Think of the amount of time and pain that caused for your development team. And what about all those scenarios where a merge wasn't even possible, and you had to throw away weeks worth of code!

What happens when you have to maintain two releases? Or even three?

Subversion is designed with file-and-directory based merging. The problem with this is that while it's great for isolating code changes, it isn't great at managing the process that brings them back together. Teams are often confused understanding when and how merges will or should take place.

Since the introduction of basic merge tracking in Subversion 1.5, the ability to track merge history is much improved in Subversion. However, there are several shortcomings. First, understanding how Subversion merging works requires every end user to be proficient in the merge internals. The moment a simple merge has a conflict (which is a very common thing), they will need to understand the internals and how they affeect other developers.

Subversion does maintain history for most merges. However there are several important cases, such as multi-branch merges, where history is not preserved. This makes it difficult to establish traceability and auditability throughout the development cycle, it is also error-prone. It becomes impossible to trace what happened when trying to understand where and from to.

Some modern commercial SCM systems do not have these problems. They supports full merge history for all files and directories, regardless of their ancestry. Ultimately, merging in SVN should not be taken lightly.

Aside from merging, parallel development requires full namespace versioning, so that code refactoring preserves the history of any moved files and directories. Subversion does not provide full namespace versioning. The move command is simply a combination of SVN copy and SVN delete, so when refactored code needs to be merged to another branch, the moved files are treated as separate entities. The net result is that previous history of the moved files is lost, rather than being merged into the target branch.The Subversion manual recommends that "until Subversion improves, be very careful about merging copies and renames from one branch to another"

Sign#3 Your Developers Don't Have a Private Place to Check-in

Committing early and often is an SCM best practice. Over the years, developers have been told that if it's not in source control, it never happened. Typically, many teams require developers to check in everyday so there is no work that's lost.

Seems simple right? Check in your work so you don't lose it…

There is a dark side to this practice though, and it comes from the way SVN handles commits and reverts. Committing to TRUNK or your project branch pretty much guarantees that you've shared this code with everyone, whether it was finished or not.

Consider a scenario that you're about 100 lines of code into a particular project, but you've become stuck on a particular function. Sometimes it's good to switch gears and move on to some other work. But there is no way to privately check in that work to the SVN repository. There is no difference between "committing" and "publishing"…they are one in the same.

Some might argue that you could simply create a private developer branch for every developer. But if you just finished reading the previous section (#2), You spend too much time branching and merging. We know this just isn't practical.

The amount of merging required for private branches is signification. SVN doies record the branch creation and revision, but being able to trace over time changes is a cumbersome manual process.

Sign#4 You Have Trouble Managing Distributed Teams

Collaborating and sharing code with distributed teams is more complex than ever. Teams routinely perform software development in many locations, and sometimes test or perform other tasks in yet a different location. This distribution of teams strains the development process. There are security, auditing, and integration problems throughout the process. In addition, there is no way to get around the fact that code bases can be massive… often containing hundreds of thousands of files. This becomes a burden when trying to maintain multiple versions of your codebase.

SVN's architecture natively supports distributed development over the WAN. While this works well when there are single users scattered throughout the world or for small remote teams needing only occasional access to the repository, connecting directly to a central repository is not optimal for large distributed development teams.

Since users must connect to the central repository even for read operations such as: seeing the history of a particular file or of the repository; listing the branches or tags in the repository; and viewing differences between two branches or tags, performing these operations can be bandwidth-limited, and developer performance is hindered by the lack of replicated metadata. Developers can experience even larger performance decreases over the WAN for write operations--every time a user updates a working copy, checks out from a branch, or exports files to disk for a build, they must wait for the data to be pulled over a WAN.

Subversion users who want to use replication to speed up this process must combine SVN with another version control system. A solution such as GIT or Mecurial could be connected (using git-svn). While these solutions provide the benefit of replication, they represent an additional cost and complexity from a technical, administrative and support perspective.

Sign#6 You Can't Visualize Or Keep Track Issues You've Delivered

To have an effective development process, the process needs to be highly visible. This gives teams the ability to communicate more effectively and self manage. Without visibility into the process, teams will not be able to make corrections during the course of the development cycle or accurately understand where bottlenecks are.

There are all types of visibility when it comes to SCM. For example, how do I know what issues are finished in a particular branch? What changes were delivered since the last release to TRUNK? How do I know what branch to merge from and then to?

Unfortunately for Subversion, the only place to store these changes are in the commit log and directory listings with revision numbers.

Since Subversion relies on revision numbers to track changes, there isn't a robust way to visualize change sets. If you're using an issue tracking system with SVN, when developers check in the code there is no linkage between the issue they are working on and the code. Typically SVN administrators will create a script that will require people to enter an issue number when committing.

This process can make it difficult to answer the following questions

- What issues have been delivered to between two different branches?
- What issues are not included in TRUNK?
- Which code belongs to issues which are not included in this release?
- How do I know which code was merged from one branch to another?

In SVN, it is difficult to work effectively by issue. This is especially true when the change set is created by more than one code change. For example, a user works on a change for user story 1 and commits the code in a branch as revision 4. Another user then commits a completely unrelated change for user story 2 to the same branch, creating revision 5. The first user, at some point, checks in more code for user story 1 again. If the user wants to merge that to trunk or a release branch they will now have to peform two different merge operations, the first user must merge changes from revision 3-4; and subsequently merge the changes from revision 5-6.

Additionally the history of those changes is now lost as the set of changes is merged from one branch to another. This makes it difficult to ascertain exactly where a particular set of changes have been applied. In most cases, users often cherry pick revisions from one branch to another, without knowing which changes have already been applied. This is both tedious and prone to mistakes. Fundamentally, SVN lacks any notion of change sets and issue-based development which would make merging these changes a lot easier.

To make this worse, SVN doesn't understand the relationship between branches. They are simply listed as a flat directory tree somewhere in a your repository. But that's just icing on the cake.

Conclusion

This is just the tip of the iceberg with Subversion. There are many other reasons why one might need migrate to another SCM system, including performance, lack of support, robust continuous integration support or advanced workflows.

But the real reason to move is to think of version control not as an inconvenience for your development team that you just have to tolerate… but a collaboration hub to improve the development process. It's really a framework to manage all of the innovation and work, which is the greatest asset of your development team.

SCM is an approach to help solve the workflow of change; enable the organizations development process; and facilitate team interactions. Is Subversion up to that task for you?

The stream based architecture of AccuRev is specifically built for today's fast paced software development processes. Streams form the architectural foundation for AccuRev to solve the fundamental problems associated with Subversion. While streams are analogous to branches, the workflow that streams provide allow you to allows you to optimize a flexible, powerful and easy to use software development process. AccuRev is the only tool that fuses development process to software assets, ensuring that your process is realized to it's full potential.