Apps Factory

Development Operations – VCS

Motivation

The aim of the DevOps series of proposals is to help formalize discussions on the various areas of development operations that can help the Apps Factory tackle issues that are prevalent with rapid-development teams. While the bias of these proposals will evidently be towards lightweight tooling and processes, we do intend to share our thoughts and experiences with other teams in hopes it provides value. This initial proposal will briefly discuss Version Control Systems(VCS) and what steps need to be taken in order to effectively implement a VCS. An analysis will be provided to showcase differences and highlight opportunities that different version control systems offer as well. While the recommendation of this proposal will focus on Git for the Apps Factory, it is encouraged to look at this from a critical standpoint given the important place a VCS has within any development team.

Holistic Perspective

Before we begin and detail what logical steps your team needs to follow in determining what VCS is appropriate (or any system for that matter!), there are two fundamental perspective that need to be considered in any decision making process:

- External: What type of software is your team building? What type of release structure(how fast) does your team need to be able to meet the needs of its stakeholders/clients? Who are your stakeholders? What are their preferences?

- Internal: What is the chemistry of your team chemistry? Are your development resources focused on one product at a time? Is everyone involved in the review and approval process? Is there a need for a centralized way of decision making? Is the team distributed? How much traceability do you require?

While these views might seem self-intuitive, often times teams pick development processes based more on technical merit (“This is the industry way of doing it”) rather then also considering the implications to the product(“I need a heavy centralized system just-because even if it forces clients to wait longer then usual”). In a way, think of these two views as pillars of a house: if there is an

inconsistency in the material or the way it was designed, you will have weakness in that foundation and once the house gets bigger it will crumble because it doesn't have a strong/consistent enough foundation to hold it together. Too frequently do teams implement some or all of the above systems in an inconsistent way and end up struggling with scale and growth issues downstream.

As the discussion on VCS will highlight, the critical nature of this system will help shape the decisions needed to determine what other solutions your team will require. More specifically:

- Product management systems like issue tracking and backlog management
- Testing systems and approaches such as continuous integration, test-driven development, and etc.
- Code Review Mechanisms
- Branching strategies

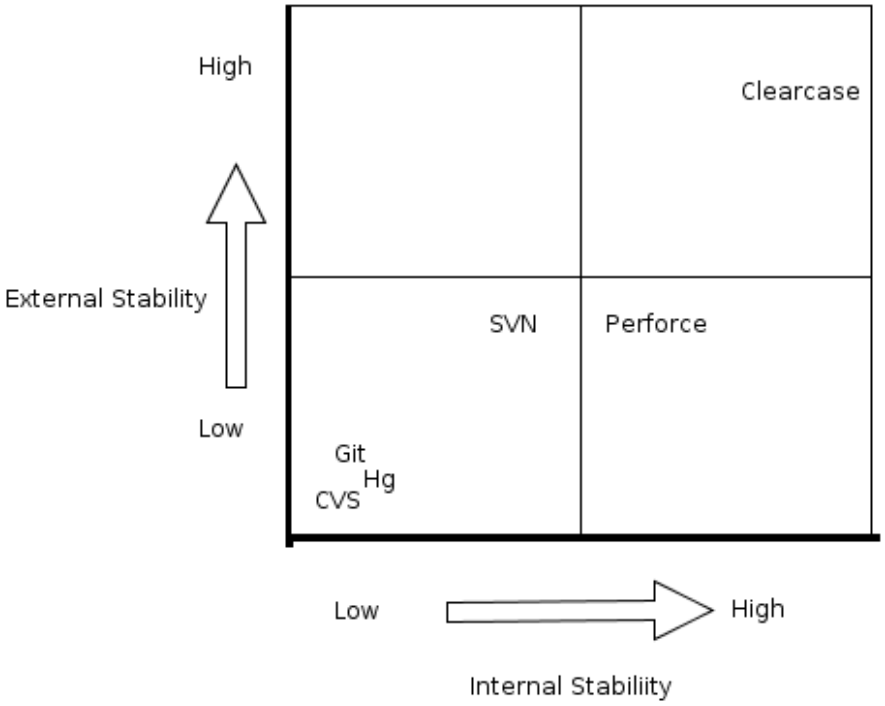
We can view all these systems as pieces of a puzzle and how "well" they fit is determined by the external(business) environment.

So what is VCS? Which do I choose?

A VCS is a concept that adheres to a view that all changes done to a code structure can be represented by revisions. A VCS organizes and keeps track of these revisions in a logical way and it's because of this simple, yet powerful, method that allows a team to manage multi-developer projects. While it's easy to pick one form of VCS, a common industry practice is to integrate many systems together based on the business environment and the constraints that those teams are dealing with. This goes back, again, to the analogy of viewing VCS as a puzzle. The rest of this article will go in a step-by-step fashion and suggest ways a team can implement a VCS.

1. Choose a VCS

Going back to the pillar concept, the following diagram will help to differentiate between the many VCSs that are in the industry and help guide the selection process:



Some notes:

- High External Stability/Low Internal Stability: This environment is characterized by waterfall management where project timelines are long and rate of change is quite low. This can be characterized by the banking/insurance industries. Normally, centralized VCSs such as SVN, Perforce and ClearCase will suffice. The focus here is on enterprise-by-design so concepts such as security, stable release and branch structures
- Low External Stability/Low Internal Stability: The focus here is on rapid-development and need for “Agile” methods is paramount. Startups, open-source and prototype development is a characteristic of this industry. Distributed systems such as Mercurial(Hg), Git, CVS and to a certain extent SVN are ideal. Such systems are characterized(not as much as SVN

which is more enterprise focused) by short branch life and lack of approval mechanisms.

    - A large focus of the proposals by the Apps Factory will be centered on tooling/workflows for this area as systems such as Git don't naturally advocate approval systems and enterprise concepts.

- Low External Stability/High Internal Stability: Usually characterized by organizations in transition. Given the low external situation where product lifelines are pressured to be shortened, systems that offer some form of cross-platform support yet maintain security and stability are ideal. Industry tends to point towards Perforce and SVN.
- High External Stability/High Internal Stability: Similar to first case, stable and prescriptive methodologies are ideal given the length of project cycles. Stable and rigid systems such as ClearCase are ideal.

Within the context of the Apps Factory, Git is the ideal candidate given the following scenario:

- Prototype development
- Distributed in nature
- Rotation of development resources is quite high

2. What goes in?

What goes into a VCS is largely depended on the concept of maintainability. Naturally, your software project should be the first thing that comes to mind (else, it defeats the purpose!). But what else? Here are some discussion points:

- Wikis – If a team tends to wiki/document meetings and processes, a system to help maintain versions is usually ideal; the phrase "A process is outdated once it's put on paper" comes to mind. Depending on the extent of collaboration within a team (think where on the stability scale), a VCS might be appropriate. Solutions such as Confluence offer similar revision systems

- DLLs/JARs: Thinking back to where your organization lies on the stability scale, this largely depends on who is managing these libraries. Some organizations have a team developing and managing these libraries while others maintain their own.

3. Structuring your system

Different VCSs have their own "best-practice" way of structuring their files and this largely depends on how such systems are implemented. While arguments may range in how/where you submit your revisions, one of the underlying themes of any VCS is the concept of a "main line" of development. This is where the key words "centralized" and "distributed" separate the various systems:

- Centralized: VCSs such as Perforce and ClearCase advocate the main line of development to be the central meeting place for developers with the onus there is a single central repository. Any changes/revisions you make need to go here before it gets shipped. Where/when that happens depends on details and concepts such as the TOFU[1] model help dictate best practice.
- Distributed: As the name implies, VCSs with this categorization tend to be flexible systems where the concept of a single "master" branch and a single repository that has all dependencies for a project is unheard off in a multi-developer environment. More details on branching strategies, which is an abstract way of structuring revisions, will be discussed in another article

4. Agree on conventions

VCSs, by design, are meant to support abstraction of code structure so there is always flexibility in how you use a system. At this point, it is important to agree on conventions to ensure your team utilizes the benefits of a VCS. Some topics that should be discussed are:

- How regular should developers update their code?
    - By not updating regularly, you are putting yourself in a situation where developers are working with old version of code which increases merge/integration conflicts down the road

[1] http://www.perforce.com/sites/default/files/flow-change-wingerd.pdf

- How big should the revisions be?
    - The more dependencies the revision contains or requires, the higher chance of conflicts downstream
- Commenting
    - Useful comments are good not just for other developers (to understand what the functionality is), but also when looking back at project history. Stress useful comments and not ones where (this author is a culprit of): “Fixed a bug”.

Fit with other topics

Now that you get a better understanding of what a VCS is and what steps to follow in deploying one, this section will briefly introduce the other topics that will be discussed and what relationship they have with revision systems:

- Branching/Tagging
    - As mentioned above, VCS allows teams to structure and organize their code to ensure proper management of code base. While VCS tends to organize things “physically”, branching/tagging is an abstract way of organizing your code to better utilize efficiency and ensure accurate execution of your release. The focus of this document will be on distributive branching strategies and looking at “best-practice” workflows in Git
- Product Management Systems
    - It is the view of this author that requirements (or stories in agile-speak) represent a contract between client and development/product-owner. While a VCS manages code, it is important to manage these requirements in a way which can fit well with the existing VCS.
- Testing and Code Review
    - While there are many ways to test and to perform code review, what the selection of a VCS will impact is the choice of tooling needed for adequate testing processes and code review systems.