VCS Evaluation

At National Australia Bank Wholesale Banking

1 Introduction

An important part of Software Delivery is managing source code. Allowing multiple versions of an application to co-exist in environments, while managing enhancements to each version, requires mature source control processes. The tools to help development teams perform these tasks are called Version Control System (VCS) [1] tools. There are many VCS tools currently on the market and the variation in functionality within those tools is also quite big. This makes choosing a VCS tool for your organisation a potentially daunting task.

National Australia Bank (NAB) is one of Australia's 'big four' banks providing retail banking, business banking, and wealth management services to customers in Australia, New Zealand, the United Kingdom, America and Asia. Wholesale Banking, a division of National Australia Bank (NAB WB) [2] has made strategic decisions in regards to VCS tools. This article describes the community driven process NAB WB followed in selecting a new VCS tool including the gathering of requirements, evaluation of tools, recommendations, implementation, and the results of the process.

It is hoped that the exposition of this process, and the resulting outcome, would be an inspiration to others in both large and small scale organisations who may benefit from similar approaches. The evaluation process described was undertaken during 2010, and the implementation of tools was undertaken during 2011, so some of the information that was used in the decision making process may have since changed. In making your own evaluation you would need to ensure you have the latest information.

2 A Community-Driven Process

There are three steps in the community-driven process [3]. They are:

- Gathering requirements
- Evaluating the options
- Writing the recommendation

Gathering requirements before discussing tools increases the objectivity of the evaluation. This prevents politics from getting in the way and helps ensure that the requirements cannot be written towards a particular tool. This also makes the end decision more defensible, and means that new tools released in the future can be appropriately evaluated without repeating the whole process.

After the requirements are agreed upon, the second step is to create a list of possible tools. If there are many options, create an elaborate list and narrow down the options by roughly comparing the product features to the list of requirements. Generally, it's fairly obvious whether or not a tool will be an appropriate candidate. The resulting shortlist can then be evaluated in more detail.

Run a proper evaluation by installing the various options in identical (or as identical as possible) environments and actually perform the actions as if you were a user. Don't trust vendor white papers! Surprisingly, this often leads to the realisation that the actual rating is lower than one would expect given the sales pitch. By evaluating only against the agreed requirements, maximum objectivity is preserved.

Throughout the evaluation of each tool, score the results of each requirement on a pre-defined scale and enter the scoring in a spread sheet for easy comparison. When all the comparisons are complete, review each score to double check whether the results are comparable. This should give a theoretical ranking of tools.

The last step is to translate the theoretical ranking into practical recommendations for the organisation. This is where costs, and unfortunately politics, become part of the discussion. The purpose of this recommendation is to inform management of the findings of the evaluation, and recommend implementation of the most appropriate tool. The organisation may still hold licenses for the current VCS, and an implementation schedule would need to be outlined. Would the migration be a "big bang" or a phased approach? What is the ROI? etc. These things need to be taken into account when creating a implementation plan.

The following sections of this article will lay out the journey that NAB WB has undertaken using the above process.

3 Gathering Requirements

Gathering requirements involved much time, effort and vigorous debate. It's fascinating how passionate people get about their favourite tools! This clearly showed the importance of VCS tooling to developers, and the importance of getting the user base on board with the selected tool.

A list of agreed requirements was gathered over the course of a 4 month period through workshops and brainstorming sessions. The requirements were divided into the following categories:

- Security
- Backup, Reliability and Robustness
- Support
- History / Reproducibility
- Scalability
- Performance
- Functionality

Each individual category was discussed in a separate session and requirements for that category agreed by all involved (40+ people). Through these group discussions, the clarity and necessity of the requirement was defined, so that ambiguous requirements were appropriately corrected.

A total of 57 functional and non-functional requirements were defined and three real life user scenarios were outlined to be used during performance testing.

Interestingly, the requirements defined did not come out of the latest and greatest thinking, nor did they promote the shiniest tool with the most bling to come out as number one. In fact, all of these requirements could have been written 20 years ago!

To close the requirements phase, multiple requests for feedback were sent via Email across the organisation. If anyone had objections to the criteria, they were encouraged to respond, or accept the outcome by their peers!

The resulting list of requirements can be found as attachment A and is periodically revised on the instant-alm website[4].

4 Tool Evaluation

Once all participants had agreed on the requirements, the team proceeded with the second of the three phases: the evaluation.

Based on the requirements listed, the VCS tools that made it into the shortlist were:

- Mercurial [5]
- TFS [6]
- GIT [7]
- Perforce [8]
- Subversion [9]
- ClearCase [10]
- AccuRev [11]

Also considered were:

- RTC [12] for various reasons, external to NAB WB, we were unable to fully evaluate.
- CVS [13] was considered however it lacked some of the requirements like atomic commit operations.

4.1 Evaluation Environment

All shortlisted tools were evaluated against the requirements and scored between 0-3 where 0 indicated that the tool did not support the requirement at all, and 3 indicated that the tool supported the requirement very well.

Every tool was evaluated by its “champion”, to ensure the user operated the tool correctly. The evaluators sometimes paired up to evaluate, but mostly did the evaluation individually.

A series of performance tests were conducted on the tools using 3 different real code sets from NAB WB applications that had been built internally. Three identical states were produced, a parent state, and 2 child states representing work from 2 different developers.

For performance testing, servers were setup for each tool on a virtualisation platform in our server environment, and clients were actual developer desktops in the office. The virtualisation platform helped ensure similar conditions across all the tools and enabled us to spin these environments up quickly and throw them away at the end.

We did not check for exceptional high/low loads on the physical server even though that might have impacted the performance tests. We accepted the risk in deviation and mitigated some of that by performing sanity checks on the performance results. There was no indication of those deviations.

We also assumed that the tools under evaluation could scale to the number of concurrent users we expected if put into the correct environment. We did perform geographical distance testing as NAB WB has globally distributed users.

Performance was measured in 9 steps to represent real development scenarios.

1. Initial import of parent state of code from the client to the server
2. Creation of two local developer work spaces or streams
3. Copying and Committing a new child state over developer stream A (note: only the committing of changes was timed, not the method of making the changes on the stream, a file copy)
4. Delivering the changes made on developer stream A to the server
5. Copying and Committing a new child state over developer stream B (note: only the committing of changes was timed, not the method of making the changes on the stream, a file copy)
6. Pulling changes from the server that were previously delivered from developer stream A onto stream B
7. Merging changes pulled from server with changes made locally.
8. Delivering the merged changes made on developer stream B to the server
9. Pulling changes from the server that were delivered from stream B onto developer stream A

These steps were performed on each tool, on 3 real code sets within our organisation, and the times were averaged across the 3 code sets to produce comparable results. Timing of operations was performed using stopwatches or scripted where the timing was too fast to be accurate using a stopwatch.

In this evaluation, accuracy was only of high importance once you were measuring a time under approximately 30 seconds. Relative performance was actually more important to us than the end number. It matters greatly for a developer whether a simple check-in takes 2 seconds rather than 2 minutes. Whether it's 2 minutes flat or 2 minutes and 5 seconds however, is not highly relevant.

4.2 Aligning Outcomes

Once the individual evaluations were completed, all the evaluators, and other key contributors, got together in a short series of comparison meetings to level the scores across tools.

In the comparison meetings the attendees went through each requirement and compared the result for each tool. We discovered many instances where the individual ratings changed as they heard of better or worse results from other tools, which helped them more appropriately rate the tool which they had evaluated. In these cases we adjusted the results accordingly.

Despite carefully defined scales, many were adjusted during these meetings given the results at hand. This showed quite clearly how much people learn when actually evaluating tools instead of simply reading white papers!

The outcome from those meetings was an agreed matrix (spread sheet) that scored the tools against the agreed requirements.

The outcomes of this evaluation can be found as Attachment B and periodically updated on the instant-alm website [14]

5 Recommendations for our Organisation

With the results matrix, it was then possible to have an evidence-based discussion around the options for our organisation, taking into account both the results of this evaluation and the current priorities of the organisation.

It is worthy to note that it is possible to use more than one tool in an organisation as large as NAB WB, due to the diverse activities being undertaken. Being a large enterprise organisation, the scenarios in which these tools are used in are quite varied, from dedicated Microsoft tooling, Java development, to Pearl or Python scripting. Thus having two or three tools that are the best tool for each scenario is far more important than mandating the use of only one tool to the detriment of some use cases. Developers are typically savvy enough to be able to use the correct tool for the job, and switch tools as required. The use of a couple of tools should not create training or code sharing issues. However, having 10 different tools would not be beneficial or cost effective either, as increasing licensing, installing and knowledge issues would all become significant overheads.

The guideline should be:

- To use as few tools as possible, as long as it doesn't significantly negatively impact efficiency. It's the pattern that matters. Multiple tools might be suitable to assist in performing a particular task. These tools can each excel depending on their contexts. By limiting ourselves to one tool per purpose, it could potentially limit the efficiency and frustrate users. One tool might not be enough to service the users, but we are unlikely to need 10…
- To use the same pattern within each tool. In our case, we have VCS policies that apply to all chosen tools.

All the tools met the security, scalability, backup, reliability, and robustness criteria. It was agreed that the simpler tools (that is, with less moving parts or installed software) would have a greater chance of supporting uptime requirements and disaster recovery failover ability. Further, tools that had inherited redundancy (i.e. distributed design) were considered even better, because when servers inevitably become unavailable due to scheduled (or unscheduled) maintenance, it is particularly good that developers are able to continue working with all local operations unaffected, and workarounds available for some remote operations (like synchronising direct to the developer sitting next to you rather than having to go through the central server).

In terms of support requirements, Mercurial and Git have far bigger online support communities that may actually deliver support faster than a paid organisation at times.. It was not considered possible to create a "One page getting started guide" for TFS and Perforce.

All tools had the mandatory history related requirements i.e. tracked history of each commit, and you can see diff's between prior versions in various ways. However TFS, subversion, and ClearCase didn't have the ability to rewrite history. That is the ability to adjust the history when necessary. It could be argued that rewriting history is a bad thing, however quite regularly this is exactly what is needed. For instance when a file was left out of a commit for a finished feature, or someone has left in a configuration setting (password for instance) that should not have gone into the VCS.

In terms of performance, there was a wide variety of results from the shortlisted tools. The typical operations of a developer during the day ranged from 20 seconds to 35 minutes. Tools that were consistently under 40 seconds were deemed acceptable. These results can obviously vary quite significantly depending on your infrastructure and network. At NAB WB, most developers are in a geographically different location (and therefore on a separate network) than the servers. Therefore, in these circumstances server based tools might have a slight disadvantage compared to the distributed tools. This evaluation however is specific to our enterprise environment and this "disadvantage" would actually reflect our end-state.

NAB WB uses a wide variety of technologies including Java, Microsoft based and more, on Windows, Linux, and UNIX, so a cross platform tool is important. TFS (and some of the other tools) do contain more than just VCS including team collaboration tools, (Unit) Test Management, Progress Tracking, Modelling, extensibility through web services and customisable ALM workflow engine. However, this was of limited benefit as NAB WB is not a purely Microsoft development shop (where TFS would shine) and already has numerous other tools for these functions.

Licencing was never a big factor in our evaluation and comparison. Users who wanted to use TFS already had an MSDN subscription which covers licensing cost; the other tool chosen, Git, is free for the client software. NAB WB purchased GitHub Enterprise [15] for use as the Git Server platform. Other alternatives considered were Gitorious [16] and Gitolite [17] +GitWeb [18] combination. GitHub was chosen for its collaboration features (Social Coding) and enterprise support for local installations. At the time, Gitorious had no such formal offering. The Gitolite+GitWeb combination has great management features for Git Repositories with better fine grained centralised access control over the repositories but was not an acceptable choice due to the lack of collaboration features (Social Coding is proving quite beneficial for shared components). Current offering or other alternatives should be checked when making a decision.

Cost of server infrastructure was similarly not considered, as NAB WB already had a high cost of server infrastructure on our existing platform, and it wasn't expected that this would make enough of a difference when compared to the benefit of choosing the right tool.

The decision between GIT and Mercurial was hard to differentiate because of the small differences in their nature and evaluation ranking (at times it seemed they had a rat race style feature list). It came down to deciding between the slightly easier useability of Mercurial (which focuses on ease of use with power features available in the background) versus the native in your face branching capabilities of GIT. Again both features are not black and white for either tools, but rather very close shades of grey. Mercurial likewise has some more advanced features in recording history of branches which Git currently lacks. There is an inevitable leapfrog effect of features between the two.

GIT had a slightly more complex feel to commit your sources. However, its advanced and flexible branching features including the performance when creating and switching branches (particularly important when the need arises to quickly switch and fix a high severity defect), coupled with its unparalleled ability to edit history, but also secure history as immutable with baked in tamper protection is very hard to beat. Essentially the specific set of features offered by Git at the time was the better combination for NAB. Windows support and useability of Git was fast becoming as good as Mercurial's with the excellent development efforts of the Git Extensions [19] team meaning that any useability issue of Git on Windows ended up being more of a perception issue than anything else.

For these reasons, the advice for NAB WB given its circumstances was:

- GIT is the default central Version Control System for all technologies including Microsoft technology based development.
- However, if a Microsoft technology based project/team intends to use the additional features of TFS that would also be setup and supported

6 Implementation and results

Management acknowledged the communities' decision to implement GIT and TFS by creating a working group [20] that implemented the recommendations. Both GitHub Enterprise (installed within our network, not the public instance) and TFS were setup, including disaster site recovery/failover ability.

Developer/Tester productivity and general satisfaction has increased since the new tools, chosen through the community-driven process, were implemented. Productivity in continuous delivery environments was the most improved, where a developer delivering code, and a continuous integration server checking out the new code in some teams has been cut from 30 minutes each to approximately 20 seconds each. Other teams reported time savings of ½ hour per day per developer (6.25%).

Because there is less time spent in operating the VCS, the developers don't get distracted. To quote a project manager: "They stay in the zone and can just get on with it". It is a comment to which many developers (including the authors) can relate.

Because the previous version control system was disliked, teams had found ways around the "problem" while still satisfying governance. Such teams would use non-standard VCS tools on a day-to-day basis and only periodically publish to the central system. However, since the introduction of the new VCS, this has disappeared entirely. Some might say that more governance would also have solved this issue, and that is true. However, that would not have helped our organisation deliver better and/or faster. Teams started using the new VCS from the moment we achieved agreement. A fantastic result! The majority of teams are all using the same standard now, without the organisation spending time and money on governance.

The support team that manages our VCS has seen a significant drop in support calls, which clearly indicates that the new tools are much easier to use.

There was peer to peer training sessions organised and run to assist teams getting up to speed quickly with the new tools. Despite this, the learning curve of the new tools did cause resistance in some teams. However, all the good news stories from other teams quickly convinced more and more to use these new tools.

Many teams that have been using the new central VCS also reported a shift in the team behaviour. The new VCS allow them to share their code very easily, which means (integration) is now happening more frequently. This has resulted in more cooperative development practices, so a better VCS actually changed the team dynamics! The organisational benefit is that code integration issues are found much earlier and are therefore cheaper to fix.

Another unexpected result is that people are sharing much more code than ever before. All sorts of code snippets are making its way into the central code repository to the point that processes had to be created around the sharing of code snippets. Some teams have literally sent out emails to encourage all their members (including testers, architects etc) to move anything useful on their hard drive and their private network folder into VCS. This is again a clear indication of a tool that is working FOR the users and not holding them back. What this delivers to NAB WB as an organisation is the ability for people to search for a solution to a problem and having an ever increasing chance to find that solution and/or a name of a colleague to ask for more information.

To summarise all the above, one of the solution architects put it like this:

“Migration of our large development projects has boosted productivity by 15 - 30 minutes per developer per day, decreased the turn-around time getting code changes built and deployed to test by several minutes per build, and greatly improved the morale and engagement level of the development team. The only downside to the introduction of GIT/TFS has been that the many legacy components still in the old system are now even less attractive to a developer than ever!”

7 Outcomes

The most important part of this initiative for NAB WB was the community-driven process used (being open and transparent) to evaluate the tools we use. At the end of the day, the internal NAB WB Development community have realised this decision and fully embraced the new tools as a direct result of their ownership of the decision.

The process followed these steps:

- Creating a list of requirements by the community before considering the solution.
- Evaluation of solutions against those requirements
- Recommendation to management by the community

It was crucial to have management backing right from the start so that the participants understood that they were enabled to drive change.

For these reasons, the advice to management of NAB WB in its circumstances was:

- GIT is the default central Version Control System for all technologies including Microsoft technology based development.
- However, if a Microsoft technology based project intends to use the additional features of TFS, then we support that too.

The chosen tools significantly improved development team performance and behaviour, exceeding our expectations in adoption.

Even though the evaluation has been performed in an enterprise size organisation, many (if not all) requirements may also be applicable for small and medium size IT businesses and departments.

8 References

[1]	"Version Control System (VCS)" http://www.wikipedia.com/
[2]	"Wholesale Banking, a division of National Australia Bank" http://www.nab.com.au/WholesaleBanking
[3]	"evaluation process" www.instantpii.org/doku.php?id=processes:evaluate_a_tool
[4]	Requirements on instant-alm website http://www.instantalm.org/doku.php?id=tools:vcs:requirements
[5]	Mercurial http://mercurial.selenic.com/
[6]	TFS http://msdn.microsoft.com/en-us/vstudio/ff637362
[7]	GIT http://git-scm.com/
[8]	Perforce http://www.perforce.com/
[9]	Subversion http://subversion.apache.org/
[10]	ClearCase http://www.ibm.com/developerworks/rational/products/clearcase/
[11]	Accurev http://www.accurev.com/
[12]	RTC http://www-01.ibm.com/software/rational/products/rtc/
[13]	CVS http://www.tortoisecvs.org/
[14]	Evaluation results http://www.instantpii.org/doku.php?id=processes:evaluate_a_tool#tool_evaluation
[15]	GitHub Enterprise https://enterprise.github.com/
[16]	Gitorious http://gitorious.org/
[17]	Gitolite http://wiki.github.com/sitaramc/gitolite/
[18]	GitWeb http://sourceforge.net/apps/trac/sourceforge/wiki/GitWeb%20repository%20browser
[19]	Git Extensions http://sourceforge.net/projects/gitextensions/
[20]	working group http://www.instantpii.org/doku.php?id=roles:roles#twg_group