Branching Strategies with Distributed Version Control in Agile Projects

David Arve

March 2, 2010

Abstract

Branching strategies play an integral part in traditional software development. In Agile projects however branching is seen as the opposite to embracing change and therefor given little attention.

This paper describes what branching strategies would support an agile workflow. It also describes how a distributed version control tool will fit better around these branching strategies then a centralized tool. This paper puts forth some original work on what branching strategies and team organization that fully takes advantage of a distributed version control tool. This paper aims to describe how a distributed version control can support agile teams with advanced branching strategies without slowing down or complicating the lightweight agile processes.

1 Introduction

Software development is a process that produces a massive amount of highly complicated content. All of this content is produced by teams that range from just a few developers up to several thousands. It is obvious that this process needs to be highly coordinated. The coordination of this content is a part of Software configuration management (SCM). Traditionally this has been done by a centralized version control tool. [2] Branching strategies have not gained a lot of popularity in agile teams. [3]

In recent years several new version control tools have emerged. These new tools break with the standard model of centralized version control (CVC) made popular by tools like CVS [9] and Subversion[6]. The new tools are based on a distributed model. Recent tools include Arch [5], Bazaar [8], BitKeeper [7] and Git [11].

As CVC only provides one central repository where developers can synchronize and merge their work it's easy for the developers and other users to understand the basic model. With distributed version control (DVC) the tools in them self does not provide a basic model in which they should be used. [4]

This paper presents why a DVC makes more sense for an agile team than traditional CVC tools. This paper also presents the workflows and responsibilities that are needed to fully take advantage of DVC in an agile team.

In section 2 the background of how these workflows have been tested is described. In section 3 the differences between DVC and CVC is outlined. In section 4 a simple workflow for a distributed system in described. In section 5 we describe some more advanced concepts and workflows. This is followed by a discussion in section 6 and conclusions in section 7.

2 Background

The methods outlined in the following sections were successfully implemented during a course in eXtreme programming at Lunds University. The course is given each year in two parts. Before christmas the students are given lectures and are taught the theory behind agile development in general and in particular eXtreme Programming. In the second part of the course the students have six weeks of simulated XP. Every Monday the students work eight hours in an agile setting. Later in the week they have a meeting with a teacher playing the role of a customer and they also have four hours of individual 'spike' time, where they investigate issues and techniques needed for the next Monday.

A total of eleven teams participate consisting of eight to ten developers and two coaches. The developers have very little experience with developing software in a group and have almost no experience with SCM tools. The developers have had a introductory lecture on CVS accompanied by a two hour lab where they can try out the tool.

Of the eleven teams ten use Subversion with the simple setup described above. These teams use little to no branching at all[1] . One team use Git with the advanced branching strategies described below. The team using git have started with a fairly simple workflow and advanced into more and more creative and flexible workflows.

2.1 Introducing Git to the developers

The team had a start-up meeting where the coaches described their role and what was going to happen during the next six weeks. A short lecture, ten to fifteen minutes, about version control and git was given. The simple workflows that are described below were outlined.

Some developers had heard about git before and were exited to use it, others had a milder interest. The scenario of a centralized model where developers commit unfinished stories to a repository was described. The problem of creating a release from this without freezing the code and without releasing something with half-implemented stories was outlined and understood by the developers.

The explanation of the distributed workflows below met some initial resistance with the developers. Some argued that this might give us bigger merge conflicts as we wouldn't integrate as often. Though without much prior experi-

[1]Some teams have used branches but only in a rudimentary way. One team i.e. used one branch to do refactoring but were never able to merge these branches.

ence with SCM tools or with developing software in a bigger group they trusted that the coaches knew what they were doing.

3 Differences between DVC and CVC

The traditional way of tracking software configuration items is the centralized model. This means that there is one central server that holds all the content and it's change history. A developer downloads a working copy on to a local machine where changes can be made and sent back to the central server.

This means that all changes to the projects configuration items will go through the central server. When a developer sends updated content to the server all other developers connected to this machine will receive the updated version when they update their local content from the server.

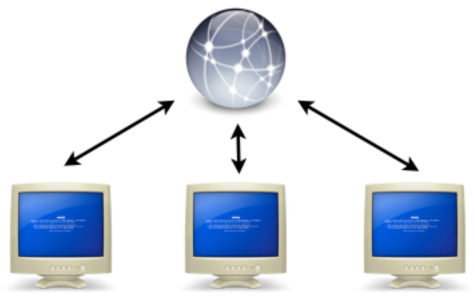

Figure 1: Traditional centralized version control.

This setup, see figure 1, gives a conceptually simple model for how developers should coordinate their work. There are several issues with setup though. One developer could upload a version with broken content, such as code that doesn't compile or code with tests that does not pass. This will propagate out to all other developers as soon as they ask the server for the latest updates. This situation will unfortunately only be discovered once the damage has already been done.

A second issue with the central approach is that a one developer may be working on a experimental feature. The developer does not want to submit this until the feature is in a stable state as this might break functionality of the content. In the simplest form of centralized SCM the developer would need to keep the changes in the local workspace. This would mean that the developer would lack all the support a version control tool can give you, e.g. history and reseting to a working version. There is a bigger problem though. Here there is no way for this developer to coordinate the changes with other developers. If the task grew in size and the developer needs other developers to help out this can not be done with a simple centralized setup.

This is where branches come in. One main use of branches is to solve the above problem. A branch in a central SCM tool is server side construction that lets developers choose where to upload their changes.

With a distributed version control tool there is no central repository. Each developer owns a complete repository with full history. The version control tool itself doesn't require a hierarchy between these developers. Each developer changes the content and publishes these changes, any other developer can pull these changes if the changes are good, but is never required to. [4]

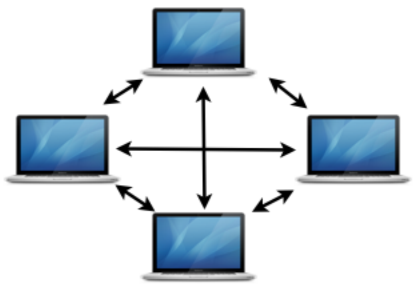

Figure 2: In distributed version control all repositories can sync with each other.

This is often seen as a complex setup where it would be hard, nearly impossible, to know who has what changes and what state the overall project is in figure 2. This is true, a distributed version control tool gives you enormous flexibility in how a team is organized, including ways that may be confusing. The flexibility of a distributed version control tool makes it imperative that the team has a clear organization around the version control tool . Or even better, the version control tool is able to fit around the organization of the team instead of the other way around. [10]

This paper describes how you relate a distributed version control tool to agile workflows. With the right usage of a distributed version control tool an agile team will have more control over the state of their project and the team will have an increased awareness of project changes.

These workflows have been successfully tested in a group programming course for second year master students in software engineering.

4 Simple workflow with a distributed version control tool

The first techniques that were incorporated into the team was a combination of two very powerful branching patterns; Branch per developer and Staged integration lines. [1]

The coaches had made a very basic first iteration of the project which every developer cloned into their own local repository. This repository is completely private to the developer, no other team member has read or write access to this repository. The developer then creates a public clone of their local repository. This repository gives read access to other team members, but the developer remains the only one with write access to his or her public repository.

The basic workflow for the team is this. A developer [actually a pair] starts to work on a story. When the developer has implemented a part of the functionality the developer commits changes to the local repository. This is done, not to communicate the changes to other developers, but to save changes to create a fine grained history of the development.

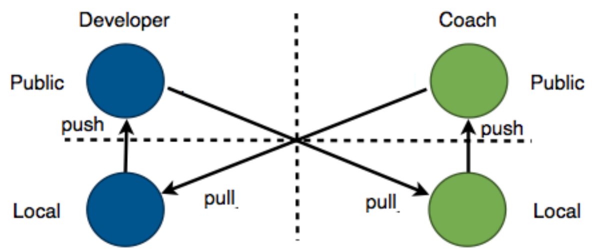

Figure 3: How git is setup for each developer to share their changes (push) and incorporate other developer's changes (pull).

When the developer is finished with the story the developer publishes the changes to their public repository. The developer then notifies the coach acting as a 'gate keeper' into the release integration line. This coach then downloads, or pulls, the changes from the developers public repository to his local repository. Here whomever is responsible for the release integration line will test the changes so that they meet the quality standards needed for this integration line. This would normally be to compile the project and run unit tests and acceptance tests. If these do not pass the changes are rejected and the developer needs to go back and fix this before these changes are accepted.

Once a developer's changes are accepted the 'gate keeper' publishes the developer's changes to his public repository. He then notifies the rest of the team that new functionality has been published and that they should download, or pull, these changes and merge with their local changes.

This means that in the simplest case the developers only care about what the 'gate keeper' publishes. Which means that there is no problem with awareness for the developers. There is only one place where they look for new changes.

The simple setup outlined above gives a natural workflow that has been much appreciated by the participants. Though one should remember that the setup above would be considered extremely complex in normal agile teams. The setup is actually equivalent to having 12 branches, one for each developer and two for the coaches. As this is a eXtreme Programming setting the developers work in pairs leaving only five branches being worked on in parallel, together with one release branch. The team normally do between 10 to 15 non trivial merges between the branches every iteration. There are also numerous automatic merges that do not require any manual resolving of merge conflicts.

It is important to note here that the fact that a coach was playing the role of 'gate keeper' for the release repository is not a necessity. Everyone is a 'gate keeper' of their own repositories and anyone could do a release. So there is nothing special with the coaches repository from the point of view of the DVC. There is one important difference though. A developers code is only done when the story is done. Most stories are only done when the customer have accepted it. This means that the customer is, or should be, the 'gate keeper' for the release branch. As the customer isn't always on site and there are other qualities like unit test coverage it is a good idea to have a coach as the 'gate keeper' for the release branch.

5 Advanced workflow with a distributed version control tool

Even though the above workflow uses some advanced SCM concepts they are quite easy in their setup and conceptually easy for the developers to understand without any previous experience with any SCM tools.

Going further then the above simple setup there are mainly two concepts that bring additional value to the team. The first is the addition of integration lines that correspond to dependencies between stories splitting the team into sub teams. The other is the use of local branches and in particular the branch per task pattern which gives the individual developers more control over their workspace.

5.1 Staged integration lines revisited

In the simpler approach one staged integration line is used. This is the release line which always keeps a releasable version of the project. That means that only when a story is completely done and ready for release it is integrated into this line.

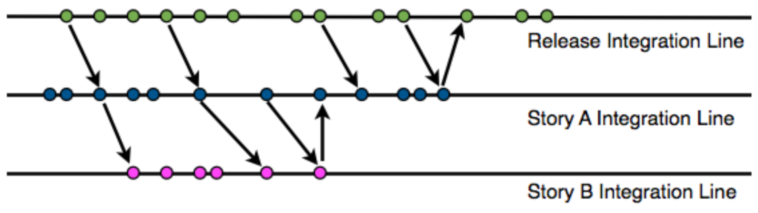

Figure 4: Staged integration lines.

There are other situations though where there is a need for coordination between different teams before stories can be considered ready for release. One such situation is when a story is divided into parallel tasks. The story is not done and ready for release until all the tasks are completely finished. With a distributed SCM this situation becomes trivial. All the developers that are working with the story pull changes from each other. If the story is split among many pairs one of the pairs will act as the 'gate keeper' for the story integration line. That pair will delegate tasks from the story to other pairs and will pull in their changes when the pairs are done. Only when all the pairs have finished their tasks and integrated the changes into the story integration line will the story be integrated into the release integration line.

This is also extremely powerful when we have dependent stories. It is not uncommon that we have stories that depend on each other. This seldom means that one story has to be completely done for other stories to be started. One might need to have the basic data structure of one story in place to start with another, but the first story doesn't have to be completely finished.

In figure 4 a typical scenario is depicted where Story B depends on Story A. After Story A has done some work another pair can start with Story B. They pull the content from the pair working on Story A and add the features needed for Story B. They continue to pull from the other pair to stay in sync with their continued development. When they are done, they notify the other pair and the pair working on Story A will pull the changes that make up Story B. When Story A is finished, both Story A and Story B will be integrated into the Release line.

5.2 Local branching

A powerful feature that is has become very popular with distributed version control tools is the possibility for local private branches. These branches are lightweight, easy to create and extremely powerful.

The typical workflow is that a developer will create a branch for the current story they are working on, see figure 5.

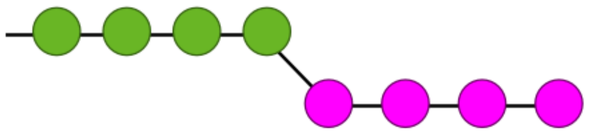

Figure 5: Working on a branch for a story.

New changes from the release integration line will be integrated into their master branch, se figure 6.

Instead of merging the two branches as seen in figure 7 the story branch will be rebased on top of the new changes, see figure 8.

A rebase differs from a normal merge in that a normal merge will result in a node (commit) with two parents. A rebase keeps nodes with single parents resulting in a cleaner history which is easier to follow. One should note that a

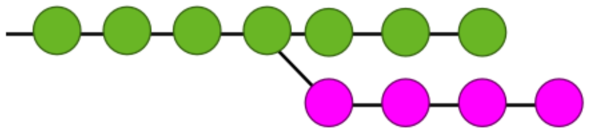

Figure 6: Development from other stories are integrated in to the main branch.

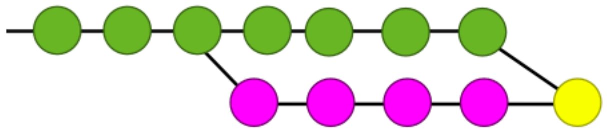

Figure 7: Merging two branches.

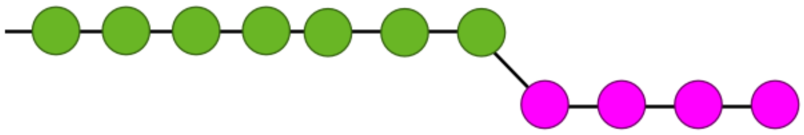

Figure 8: Rebasing a branch.

rebase also needs to merge changes. The nodes will have to be changed to be able to fit in their new position.

6 Discussion

This section discusses the results of implementing the above described workflows and branching strategies.

6.1 How successful was the use of git

At the last iteration the developers were asked to answer some questions regarding the usage of git. The developers were asked to rate how much they agreed, where 1 meant that they strongly disagreed and 5 would be that they strongly agreed. [2]

6.1.1 Did the lecture and exercise in CVS help you understand git?

The developers rated this at an average of 3.2 where all but two developers gave it a 3. It is hard to draw any specific conclusions from this. The author thinks

[2]The questionnaire was done in Swedish.

it is good to have some kind of general version control experience before joining an agile team that depends heavily on version control.

6.1.2 Do you feel that you can take over the responsibility of acting as the release 'gate keeper'?

The developers gave this question an average of 3.3. The answers were mixed between twos, threes and fours. It is possible that the higher results come from developers that have been responsible for a "staged integration line" (see section 5.1). The author would have hoped that this number would have been higher though it might be hard to get fives from developers that have never tried to be a 'gate keeper'.

6.1.3 If you were to start your own project with you use git as version control?

Everybody except one developer stated that they would use git. Some developers added that this was mainly because it was the version control tool that they had the most experience with after this course. Others also noted that they had heard horror stories from other teams about their usage of svn and had been discouraged.

6.1.4 How did git work in general during the iterations?

This was rated as 4.7 by the developers, scoring seven fives. The author takes it as a success that the developers were so happy with the version control tool and how it was used.

6.2 Other teams branching experience

There were some teams that tried using branches in Subversion. The main reason for these teams to use a branch were to freeze development in a release branch. They created a release branch in Subversion where they would only commit bug fixes for the release. One team the author talked to were able to merge this branch back using Subversion after the release was done. The other teams either did manual merges between branches or completely discarded one of the branches. All teams the author contacted that had used branches had abandoned the strategy as it was considered to cumbersome with little or no advantages.

6.3 A gatekeeper is not needed

In discussions with other coaches I've met the criticism that there is no need for a 'gate keeper' and that it is in fact a bad practice. The criticism is mainly on two points. First, a gatekeeper is often a coach or someone else in a leading role. This is seen as contradictory to the coach role of being as passive as possible and not active during the development. The other criticism is that by using a gatekeeper the team does not spread the responsibility of the code to the whole team.

The role of the coach is not carved in to stone. It is not an end goal that the coach should not be doing anything. Or if so, then there is place for more roles then just developer, coach and customer. The 'gate keeper' typically has two responsibilities. The first is to judge whether the functionality of the story really is in place or not. This is often best done by the customer, but some times this can be delegated to a coach or a developer. The other part is to make sure that the code that is submitted is of reasonable quality. This is best done by a coach or, better still, a developer. Here a story can be sent back if there is not enough test cases or that the design needs to be refactored.

The main advantage with a 'gatekeeper' is to split the development of a story and whether or not it is finished or not.

6.4 Branching contradicts continuous integration

A possible criticism is that branching can't be done in an Agile environment cause branching breaks the idea that a team should embrace change and integrate continuously. The point here being that when we branch the code diverges which is not good mainly for two reasons. It will build up massive merge conflicts and will reduce the awareness of the team.

First of all Agile methodologies puts working code as the way to measure progress. This is why we shouldn't integrate code that doesn't compile or where the unit tests doesn't work. We do this for two reasons, if the code in the repository doesn't work we can't create a release to the customer and if we share code that doesn't work every developers workspace will be broken.

The big point here is that we shouldn't share code that doesn't work. But it's not enough that code compiles and the unit tests passes for it to work. If the code is in a state where it would not be releasable to the customer, then it's broken. If we want to commit partially working code we need branches. Branches is also needed for developers to share code that is not working between each other.

With a traditional Agile setup a developer merges with other code all of the time. The problem with this is that the code that the developer is merging is small and out of context. It is unclear to the developer that merges the code what the author of the code is trying to achieve. With branching strategies the entire thought process is done when a developer merges the code. This means that it is easier to follow why the changes has been made in the code. It also gives for a better opportunity for the authors of the code to explain what they have done, as they are actually finished.

6.5 Why is a distributed tool better then a centralized

From Configuration Management experts the author has heard the criticism that there is nothing new that is added by the distributed version control tools. There is nothing really different between Git and CVS for example.

All the branching patterns and workflows that we have described above you can implement with a centralized version control like CVS. There are some disadvantages though.

If we would use a centralized version control creating a branch per developer would in the course above amount to twelve branches (one per developer and the two coaches) together with one release branch. This means that the number of branches that a developer would be exposed to would be thirteen. This despite the fact that the developer, in the initial stages, is only interested in his or her own branch and the release branch. If the team is bigger the number of branches will soon be come completely unmanageable.

In a distributed version control tool the developer only needs to know about their own repository (which is their own branch) and the 'gate keepers' repository (i.e. the release branch).

A smaller problem is the fact that all the branches on a central server live on the same name space. This means that every branch must be named differently. This can easily give complicated branch names.

In contrast a distributed version control tool a developer can name remote repositories whatever he or she wants as this is done locally.

A bigger issue is commit rights. If we would create all these branches in CVS or Subversion there wouldn't be any restrictions on what branches a given developer can commit. This means that there is a risk that developers commit to the wrong branch. Together with naming issues this problem gets bigger.

WIth a distributed version control tool nobody has write access to a developers repository other then that developer. Therefor it is not possible for anyone by mistake to commit content to the wrong repository.

Another issue is that branches are hard. Branches in a centralized version control is hard and local branches on a distributed version control is hard. The big advantage with a distributed system is that the local repository is not seen as a branch by developers. So even if a developers repository in every sense is a branch this is conceptually hidden by a distributed version control.

7 Conclusions

This paper shows that branching strategies can help agile team be more agile. It shows that a distributed version control tool is far more adaptable to the agile workflow then the traditional tools. Agile teams have been unwilling to adopt advanced SCM practices as they are perceived as cumbersome. Distributed version control tools are far more adaptable and fit a lot closer to the agile workflow making adoption extremely easy.

The fact that ten students in their second year of their software engineering masters studies, without any previous experience with SCM tools or even software development in teams in general, have no problem of accepting and mastering advanced SCM techniques with a distributed version control tool, shows that this is not complicated to adopt.

References

[1] B. Appleton, S. Berczuk, R. Cabrera, and R. Orenstein. Streamed lines: Branching patterns for parallel software development. In *Proceedings of the 1998 Pattern Languages of Programs Conference, PLoP*, volume 98, pages 98–25. Citeseer.

[2] L. Bendix and T. Ekman. Software Configuration Management in Agile Development. *Agile Software Development Quality Assurance*, page 136.

[3] K. Braithwaite and T. Joyce. XP expanded: Distributed extreme programming. In *6th International Conference on eXtreme Programming and Agile Processes in Software Engineering*, pages 180–188. Springer.

[4] Ian Clatworthy. Distributed Version Control Systems Why and How. 2007.

[5] Free Software Foundation. Arch. `http://www.gnu.org/software/gnu-arch/`.

[6] The Apache Software Foundation. Subversion. `http://subversion.apache.org/`.

[7] BitMover Inc. BitKeeper. `http://www.bitkeeper.com/`.

[8] Canonical Ltd. Bazaar. `http://bazaar.canonical.com/en/`.

[9] Derek Price. CVS. `http://www.nongnu.org/cvs/`.

[10] T. Schummer and J. Schummer. Support for distributed teams in extreme programming. *Giancarlo Succi, Michele Marchesi:eXtreme Programming Examined, Addison Wesley*, 2001.

[11] Linus Torvalds. Git. `http://git-scm.com/`.