Simplifying and unifying SCM for novices

Max Åberg mat09mab@student.lu.se
Jacob Burenstam Linder ada09jbu@student.lu.se

March 4, 2014

Abstract

This report gives a brief introduction to the basics of our own developed tool called git-story [1]. We have had the opportunity to coach a group of 8 students during spring 2014 in agile software development. We used this occasion to test Git in practice in a somewhat small project, with the goal of finding out Git's usability and learning curve with the help of git-story. We wanted to find out whether or not Git with a work flow tool is a good alternative to SVN as a first encounter with software configuration management tools. Since git-story is our invented experiment, feedback was of importance. This feedback and git-story's modifications, in consideration of the feedback, is described and analyzed. Finally a conclusion is made whether git-story was a good alternative to Subversion.

Contents

1 Introduction

This study is a part of a course at the Faculty of Engineering, Lund University where students in groups of eight to ten develop a project in an agile manner. We, the authors, are acting as coaches to one of these groups. Our role as coaches will change during the project, considering that we want to experiment if development tooling can benefit in agile means.

This report will explore the potential efficiency benefits of using customized development tooling for new XP development projects, built by inexperienced developers. This is achieved by using tools for Software Configuration Management (SCM), Continuous integration (CI), consistent developer environment and build tools (Maven). The tools used for each of these practices will be described later in the report.
Each integration is tested by a CI server. Feedback from test runs and other reports will be available to developers about the state of the code.

Our goal is to empower and simplify each developers work flow, so that they can focus on writing good, maintainable software while providing good traceability and consistency.

2 Problem

Extreme programming (XP) is now a very common software development method, since it provides a framework for code changes during a project's lifetime, which is notoriously difficult.
XP is based around the five values: simplicity, communication, feedback, respect and courage[2]. These values have resulted in a number of methods, work flows, metaphors and other ideas based upon the five values. We focused on improving Software Configuration Management (SCM) from an XP perspective. We developed an SCM tool, built on top of Git, where we put emphasis on the XP values: simplicity and feedback.

Most of the other 2014 EDA260 teams use Subversion as well as almost all other teams the year we studied EDA260, using only one master branch which all developers pushed to. This increases the probability of a lot of merge conflicts. Furthermore the commit history becomes very hard to read since all commits are displayed in one consecutive list. This leads to a commit history where small and large commits are at a glance indistinguishable. Our experience using only Subclipse and the server provided by the Faculty of Engineering, Lund University was that nobody ever checked the history log or reviewed commit diffs.

3 Hypothesis

We argue that by using software development tools, it could significantly increase developer productivity. By simplifying the standard work flow for developers in agile teams, they can focus on the code and the task at hand. It will also yield (in our variant) a more consistent development process.

4 Background

The basic goal of any Version Control System (VCS) tool is to keep a record of file changes over time. For develoment projects, it often means collaborating with other developers and keeping track of each individual's changes.

4.1 Central VCS

A central VCS tool has a central repository were each tracked file's entire history is kept[3]. To add your changes or checkout other's changes you need to communicate with the central server. Examples of central VCS tools are Subversion, Peforce and CVS.

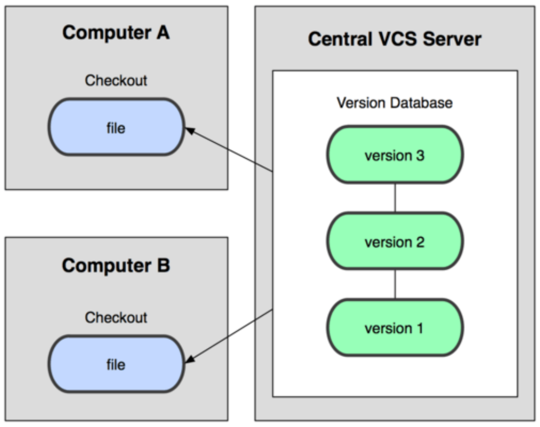


Figure 1: Central VCS diagram

4.2 Distributed VCS

Using a distributed VCS tool each machine has the repositories entire history. All changes and branches are tracked locally, no central server is required. Developers can synchronize their work directly between them. However it is very common to have some sort of central repository that developers push to. Example of distributed VCS tools are Git and Mecurial.

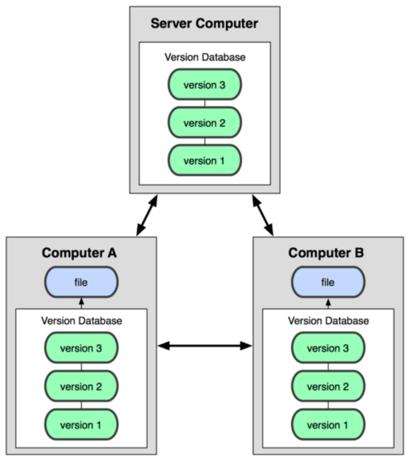


Figure 2: Distributed VCS diagram

5 Suggested solution

To increase developer productivity we decided to experiment with tools. Tools that will result in empowerment and knowledge and at the same time making development as easy as possible. Instead of only using already developed tools we also created our own tool, extending Git. This tool, called git-story, enforces the usage of the GitHub flow branching model. It provides a simple user-friendly command line interface and automates good VCS practices.

5.1 git-story

git-story is a shell script that uses Git under the hood. It performs various checks inorder to preempt common misstakes. Further it is possible to set version controlled commands for running tests and making releases. It requires the usage of a central server, inorder to use all of git-story's functions.

We were inspired by the branch model described in Scott Chacons blog post GitHub Flow [4]. GitHub Flows branching model provides a clear and easy to read history where each feature is developed in its own branch and then merged using GitHub pull requests. The model provides excellent traceability since each branch is logically named (i.e task8.1, fileprinter bugfix) and an aggregated diff can be displayed on GitHub. Each branch has its own commit log where each commit can be reviewed separately.

GitHub flow main principles:

1. master branch is always releasable
2. create each new feature from master branch
3. create a pull request on GitHub to merge with master

git-story design goals:

1. As simple as subversion
2. Unified development process
3. Run tests before each commit[5]
4. Integrate with the project's build and test process

The main design goal was to simplify the teams work flow and they should feel comfortable using it in five minutes. This was accomplished with a shell script utility. The script can be used with only two commands to achieve the standard development work flow: `gs dev` and `gs done`.

Before each commit the project's entire test suite is runned. The developer gets prompted to answer if the tests pass and can choose to abort the commit or ignore the failed test and commit it anyway. `gs dev` sets up a new working branch based of the latest version of master (or specified) branch in the remote repository. `gs pull` synchronizes the remote master branch with the current working branch. `gs done` "commit message" commits changes and synchronize changes from remote master (or specified) branch and pushes the committed code to the remote.

Along side this, GitHub will be used (though there are several alternatives CodebaseHQ etc.) in order to provide a simple and intuitive interface for reviewing commit diffs, branch merges and

general project activity.
On GitHub we added a “service hook” [6] to Travis-CI, which signals Travis-CI that a push has been made to the repository. Travis-CI will then run the project’s test suite and report back the result to GitHub.
In order to get your code merged to master a pull request has to be made from the GitHub web interface. In the pull request view, feedback from Travis-CI is displayed (OK, error, fail etc.) and a pull request should only be merged to the master branch if Travis-CI has run all tests successfully. If any errors occur or if any test fails, rinse and repeat.

5.1.1 git-story configuration

Configuring git-story is done by defining certain variables at the project root in a file named *.gitstoryrc*. All project specific configurations are defined in that file. Available configuration options are pre-commit hook, pre-commit checklist message and various prompt flags.

5.2 Maven

We decided to setup the project with Maven [7] since it is a very mature and free project that is very commonly used in the Java community, though there are several alternatives that achieves the same functionality. The rationale for using Maven, was that we wanted each development environment to be as similar as possible, so that if any dependencies where added all developers would use the exact same version of the dependency etc. Maven also has the ability to build jar-files, so that releases can be build with ease by any developer in the team.

5.3 Travis-CI

We choose to use Travis-CI [8] as our continuous integration server. There are several alternatives such as Drone.io or you could roll your own Jenkins server. We chose Travis-CI because of their pre-built integration with GitHub. Each time the commit is pushed to the project’s GitHub repository GitHub signals Travis-CI that there is new code to test. Travis-CI then downloads the commit, builds and tests using Maven and sends back a signal to GitHub whether all tests pass.

6 Analysis

Many larger software companies have dedicated “tooling teams”. Their only responsibility is to maintain and build development tools for other developers within the company. Other companies use “hack weeks” in order to let developers build their own tools [9].

After each iteration we asked the team to evaluate the tools we had chosen for them.
During the first iteration there was a fair amount of confusion regarding the tools we choose to use. Even though we gave all of them an hour long spike[1] to get acquainted with git-story. The confusion was mostly a result due to inexperience with our predefined work flow in collaboration with git-story. The most common reason that resulted in trouble was that the developers forgot to branch and developed directly in the local master branch. The development environment used, Eclipse, also caused a lot of problems. In collaboration with the Maven environment the JUnit dependency didn’t work as expected, this due to that Eclipse didn’t take these dependencies to account. On some of the developers logins, Eclipse couldn’t find/define the different class

[1]Homework of ca. 4 hours

paths and errors were distributed through all the code. This was taken care of manually by us coaches through the build path option. However it wasn't long before all programming pairs was comfortably using git-story and later in the afternoon almost no questions arose.
Other problems that were unrelated to our study also surfaced. We experienced a bottleneck in the afternoon, this happened due to a distributed denial of service attack (DDOS) on GitHub. Groups not using Git didn't experience this problem but did instead experience other problems and more often with their version control tools, which in some cases took hours to resolve[2].
Switching pairs was also a difficult practice for the team to apply. We as coaches needed to step in and enforce them to switch pairs.
Something that really surprised and impressed us was their will to refactor their code in the first iteration. The developers initiated a smaller meeting amongst themselves, on their initiative, and discussed possible future problems with their existing structure and counter actions as solutions.

Unfortunately these counter actions that were discussed were very specific and didn't leave much room for bigger changes in the code. Although this the developers did manage to develop a lot, for a limited amount of time.
Before the 4th iteration we predicted that architecture problems would arise, and right we were. This because of the previously refactoring wasn't general refactorization, the team had only done refactorizations that temporary treated problems in the nearby future (next coming story). We decided to step aside a little, since we previously had stated the importance of refactoring, and let the team discover this for themselves. Unfortunately they discovered this in the afternoon and not much was successfully delivered. The customer had prioritized technical document and the focus was mostly placed on that, but even this task wasn't done effortlessly.
Sadly the technical documents were stored in the Maven target folder. After a pull, Maven runs all tests and outputs to the target folder and therefore the files were erased. This folder is git-ignored and changes were not tracked. The issue was resolved by using the Eclipse history-feature.
That Git didn't track the target folder was expected behaviour, the problem occurred because the developer didn't know about the gitignore. This might have been naive from us and at the same time bad practice from us. We should also have emphasized the importance about gitignore and made it clear which paths/folders that are ignored.

7 Discussion

As expected several problems and issues evolved during the project's life cycle. Both issues related and unrelated to our developed tool git-story. Since we want to focus on the impact of our tool in software development, we will only discuss problems and solutions related to git-story.

In the initialization phase the developers were not comfortable and inexperienced with git-story, which was expected. The branching problem mentioned in section 6 was directly due to a neglection when implementing the tool. This was immediately taken care of by rewriting the `gs dev` command in git-story. The new implementation resulted in an extra parameter to the command (`--force`) that, as the name states, forces a branching even if uncommitted changes are present [10]. Early on duplication issues arose due to pairs editing the same code. This could not be prevented with our tool, but the manual revert was annoying. To accomplish even more simplicity we implemented another command in git-story (`gs stash`) , that pushes the newest changes on to a local stack and removes it from current working directory.

[2]Group 2 in computer room Alfa

An issue that recurred a few times was "red code"[3] in the repository. Not a big problem, but should not have happened. The problem was that we as coaches hadn't emphasized and explained the steps in a review of a task and therefore the developers overlooked these issues. By simply explain this to the developers made the problem obsolete for future pushes.

These issues mentioned are the only ones that occurred during the project's lifetime, that were related to our tool git-story. A lot fewer problems than we hade expected. But to ensure that we are not biased, we established a questionnaire regarding the features and benefits of git-story [11]. A summary of the developers responses can be seen in Appendix A [12].
To ensure the reader that a non bias study has been performed, statistics and comparisons have been made and are presented below.

7.1 Statistics

The number of pull request remained fairly stable the first two iterations. It indicated that developers SCM work flow didn't drastically change after using git-story a while. The third iteration had a spike in number of commits, due to some bad-smells in the code base, which caused a number of commits fixing thoose bugs. During the forth iteration a larger refactor branch was created, which increased the number of branches. Before the fifth and sixth iteration we put emphisis on writing documentation and updating the manual as part of each task. This had previously been done after commiting code and tests changes, hence the commit-counts were noticeably lower.

Iteration	Pull requests	Branches	Commits	Builds
1	21	18	67	90
2	23	21	73	101
3	20	20	126	103
4	28	21	86	116
5	18	18	60	100
6	13	9	87	103
Total	123	107	499	613

Table 1: Statistics regarding the code for every iteration

Overall the team produced a rough average of 5 commits per branch. Each task or story is contained in it's own branch and are therefore easier to track than a flat history of five hundred commits.

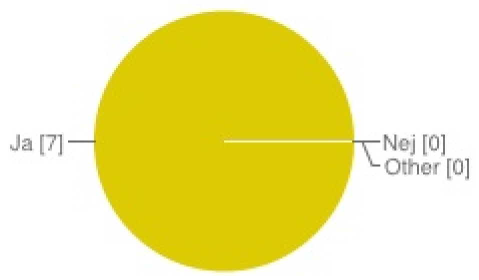
8 Conclusions

Even though problems arose, these problems would have happened with any other tool. Our tool is not in anyway comprehensive but has evolved during its lifetime after receiving developers feedback. git-story is open-source [1] and is therefore customizable to suit developers applications.
The feedback from the questionnaire speeks for itself and summarizes a result that fulfill our initial hopes, that git-story has been a problem solver and served it's purpose regarding version control in agile development teams.

[3] Code that can't be compiled

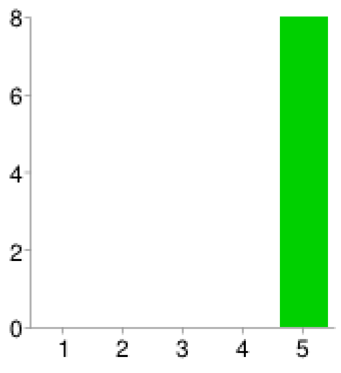
With the help of git-story, in collaboration with Git, traceability and simplicity has been a lot easier for both us coaches and the developers. Even though many developers in our team were inexperienced using VCS tools.
There are several ways in which git-story can improve. Inorder to be used full scaled, as Git and Subversion are, it would need a lot of work. git-story shows however that there are several advantages for such a tool, especially working with an inexperienced team.
git-story is also being tested at a small startup consisting of both designers and experienced developers. git-story was used to enable the designers to work using the same branch and test strategy as the rest of the development team. The feedback has been positive, requests has mostly been better feedback and error messages.
All in all we think that git-story is a viable substitute to Subversion for the PVG course.

Är git-story enklare än svn?



Ja	7	100%
Nej	0	0%
Other	0	0%

git-story har varit ett bra verktyg för PVG kursen



1	0	0%
2	0	0%
3	0	0%
4	0	0%
5	8	100%

References

[1] GitHub. Git-story. https://buren.github.io/git-story.

[2] Chromatic. *Extreme Programming Pocket Guide*. O'Reilly Media, July 2013.

[3] Scott Chacon and Junio C Hamano. *Pro git*, volume 288. Springer, 2009.

[4] S. Chakon. Github flow. http://scottchacon.com/2011/08/31/github-flow.html, August 2011.

[5] Lou Kosak. Testin at airbnb. http://nerds.airbnb.com/testing-at-airbnb/.

[6] Webhooks. http://developer.github.com/v3/repos/hooks/.

[7] Maven. https://maven.apache.org/.

[8] Travis. https://travis-ci.org.

[9] Joakim Sunden. Organizing a hack week. http://labs.spotify.com/2013/02/15/organizing-a-hack-week/, February 2013.

[10] GitHub. Issue 14. https://github.com/buren/git-story/issues/14.

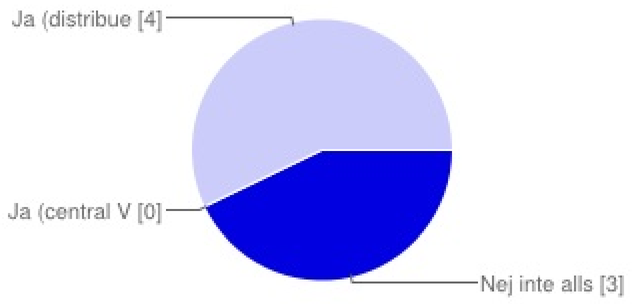
[11] Questionnaire. https://docs.google.com/forms/d/1GgRrwFOEseIx8oKEanqbl9Ua7eInX78LMb-EVv_7prk/viewform.

[12] Questionnaire response summary. https://docs.google.com/forms/d/1GgRrwFOEseIx8oKEanqbl9Ua7eInX78LMb-EVv_7prk/viewanalytics.

9 Apendices

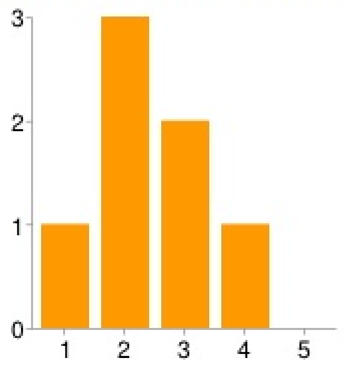
9.1 Appendix A - Summary of feedback form 1

Hade du tidigare erfarenheter av VCS innan pvg-kursen?



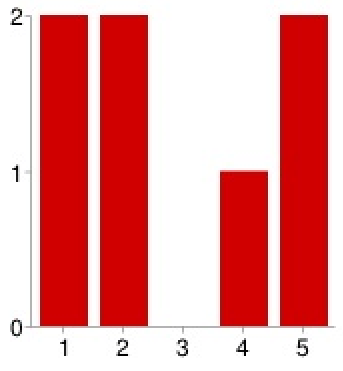
Nej inte alls	**3**	43%
Ja (central VCS tex svn, cvs)	**0**	0%
Ja (distribuerat VCS tex git, mecurial)	**4**	57%

Hur bra koll har du på något VCS sen innan?



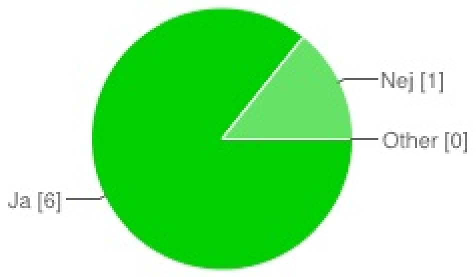
1	**1**	14%
2	**3**	43%
3	**2**	29%
4	**1**	14%
5	**0**	0%

Hur enkelt är git-story att använda?



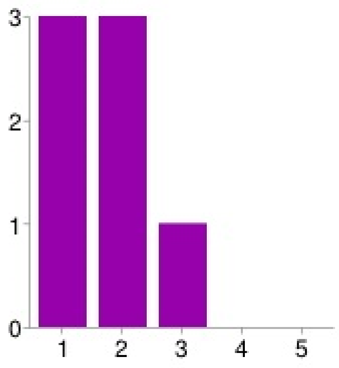
1	**2**	29%
2	**2**	29%
3	**0**	0%
4	**1**	14%
5	**2**	29%

Jag rekommenderar git-story för utvecklare med lite erfarenhet



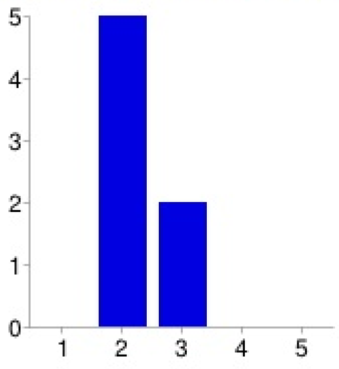
Ja	**6**	86%
Nej	**1**	14%
Other	**0**	0%

Att använda terminalen med git-story försvårar saker jämfört med att ha ett användargränssnitt



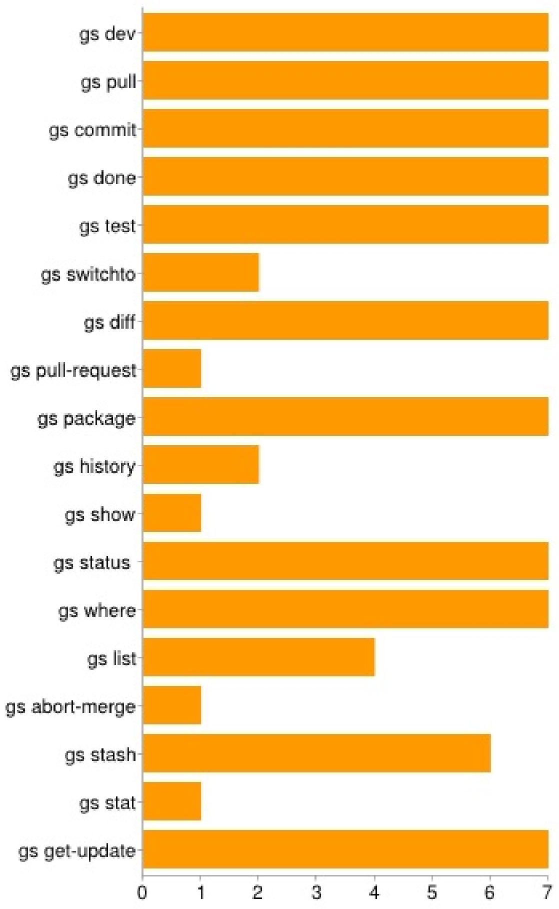
1	**3**	43%
2	**3**	43%
3	**1**	14%
4	**0**	0%
5	**0**	0%

När problem har uppstått (relaterat till git-story) har lösningen varit komplicerad



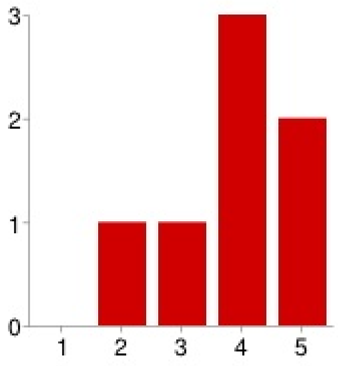
1	**0**	0%
2	**5**	71%
3	**2**	29%
4	**0**	0%
5	**0**	0%

Vilka kommandon känner du till och vet vad dem gör?



gs dev	7	8%
gs pull	7	8%
gs commit	7	8%
gs done	7	8%
gs test	7	8%
gs switchto	2	2%
gs diff	7	8%
gs pull-request	1	1%
gs package	7	8%
gs history	2	2%
gs show	1	1%
gs status	7	8%
gs where	7	8%
gs list	4	5%
gs abort-merge	1	1%
gs stash	6	7%
gs stat	1	1%
gs get-update	7	8%

Snabbguiden gör det lätt att förstå användningen av git-story

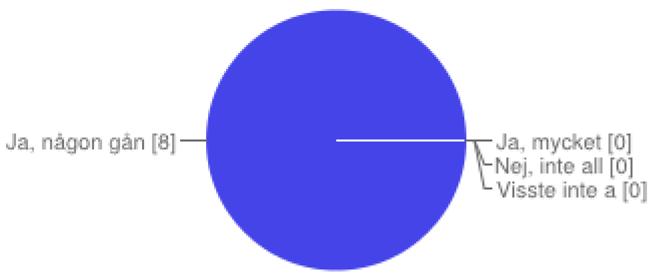


1	0	0%
2	1	14%
3	1	14%
4	3	43%
5	2	29%

Förbättringar till README

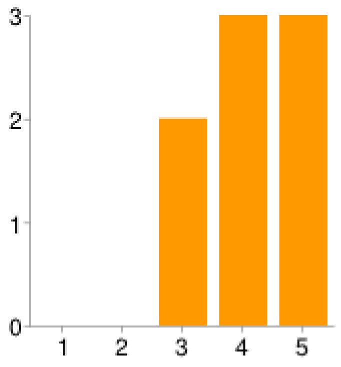
Jag är jävligt nöjd med gs, det är väldigt lätt att arbete med det och det är lätt att göra de saker som man vill göra. Det är lätt att göra de lite mer komplicerade sakerna (i förhållande) som git erbjuder, ganska så lätt. Den är fin. Mycket smidigt verktyg, förenklar mycket vilket leder till mer frekventa commits/push/pulls, vilket i sin tur ger färre mergekonflikter osv. Alla fixar som involverar att man måste kunna git-kommandon i sig självt borde det isåfall finnas mallar för (dvs de olika situationerna som kan uppstå). Jag vet faktiskt inte, jag har inte tittat i den. Jag ser inte så mycket som kan förändras. Det fungerade bra eftersom att vi fick en liten "walkthrough" innan.

Jag har använt mig utav git-story dokumentationen



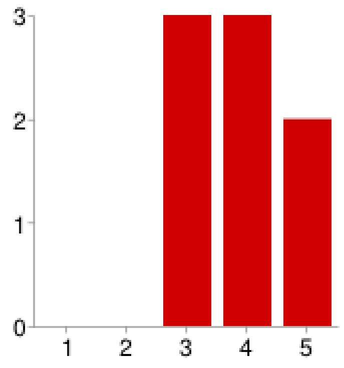
Ja, mycket	**0**	0%
Ja, någon gång	**8**	100%
Nej, inte alls	**0**	0%
Visste inte att det fanns	**0**	0%

Jag kommer vilja använda git-story i framtiden



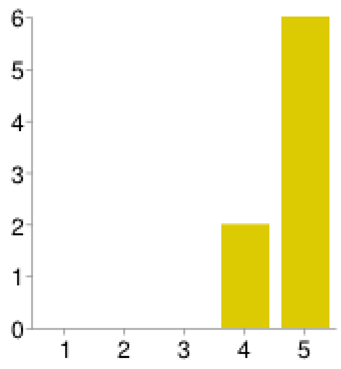
1	**0**	0%
2	**0**	0%
3	**2**	25%
4	**3**	38%
5	**3**	38%

Jag kommer fortsätta använda git-story efter PVG kursen



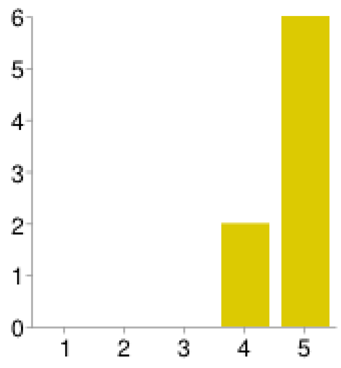
1	0	0%
2	0	0%
3	3	38%
4	3	38%
5	2	25%

Jag kommer vilja använda git i framtiden

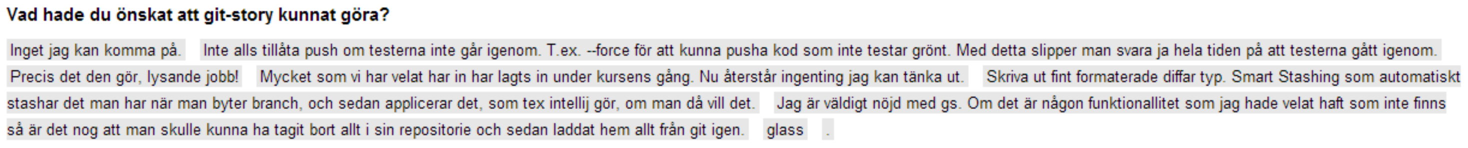


1	0	0%
2	0	0%
3	0	0%
4	2	25%
5	6	75%

Jag kommer vilja använda git i framtiden



1	0	0%
2	0	0%
3	0	0%
4	2	25%
5	6	75%



Vad hade du önskat att git-story kunnat göra?

Inget jag kan komma på. Inte alls tillåta push om testerna inte går igenom. T.ex. --force för att kunna pusha kod som inte testar grönt. Med detta slipper man svara ja hela tiden på att testerna gått igenom. Precis det den gör, lysande jobb! Mycket som vi har velat har in har lagts in under kursens gång. Nu återstår ingenting jag kan tänka ut. Skriva ut fint formaterade diffar typ. Smart Stashing som automatiskt stashar det man har när man byter branch, och sedan applicerar det, som tex intellij gör, om man då vill det. Jag är väldigt nöjd med gs. Om det är någon funktionallitet som jag hade velat haft som inte finns så är det nog att man skulle kunna ha tagit bort allt i sin repositorie och sedan laddat hem allt från git igen. glass .