Workspace awareness in an eXtreme Programming context

F. Gullstrand and R. Simko
[dt07fg3 | ada09rsi]@student.lth.se
Lund Institute of Technology, Lund Univeristy (LTH)

Abstract—We would like feedback on the overall structure of the report, the outline, as well as if all sections are sufficiently detailed. Can a reader understand the background, why are we using this tool, what does it do and do you know (enough) about how it works after reading this paper?

Index Terms—Workspace awareness, SCM, conflict avoidance, eXtreme Programming, Crystal

1 Introduction

MERGE conflicts, the bane of every larger software development project. The fact that the project is an agile one definitly does not improve the situation, with shared code ownership, no strict rules and simultaneous development the possibility to make a mess is significant. However, several tools are available to solve this problem, saving the developers headache and hours of fixing conflicts.

This paper will try to outline the problem and a few possible solutions, testing one of them practically in a comparitive study done on a team of students taking the course EDA260 - Software Development in Teams (Programvaruutveckling i Grupp in swedish or PVG for short) given at Lund Institute of Technology (LTH). The paper is part of an in-depth study in the course EDA270 - Coaching of Programming Teams also given at LTH.

February 17, 2014

1.1 Problem statement

This paper analyzes the occurence of merge conflicts in agile software development in general as well as how these could be prevented. Initially we find different tools and practices which can be used to reduce or even completely eliminate the occurance of merge conflicts.

The authors also performed an empirical study on a course given at LTH simulating an agile development environment and some conclusions will be drawn based on this. The main focus in this part will be on actual saved time compared to simply handling the conflicts.

2 Theoretical background

This section focuses on analysing the occurance of merge conflicts in software development projects in general and how this can be prevented.

2.1 The occurence of conflicts

Brun et al. analyzed how common merges and conflicts were across a multitude of large open-source projects including Git, Perl and jQuery. [BHDN11] The conclusion was that, in these projects, 19% of merges were so-called textual conflicts. The definition of this being that the tool could not merge one or several lines of code because simultaneous edits had been made. The remaining 81% were clean merges, i.e. merges made automatically by the version control tool. This does however not mean that the conflicts resulted in runable code but simply that they were resolved from an SCM point of view.

2.2 Different kinds of conflicts

Based on Brun et al., conflicts can easily be divided in to several categories. First is the textual merge, a merge which is the result of two developers changing the same line in the same file, preventing the tool from merging it. The second type is a build conflict, a conflict which is resolved textually but still does not compile. An example could be one developer changing a method's signature resulting in a failure to compile for another developer. The third one is a test conflict, where the text is merged, the code builds but tests fail to run, perhaps because of some changed behaviour in the code. The two latter kinds of conflicts are what Brun et al. call higher-order conflicts and are relatively rare. [BHDN11]

The first of these conflicts can be fixed through better tools which do a better job of quickly and automatically merging text files. A good example of this is the three-way-merge implemented in, among many other SCM-tools, Git. It is however not always possible to perform an automatic merge and in that case good information must be presented to the developer in order to simplify the manual merging. The result of better SCM tools and better manual merging tools (Through more modern UIs) is that this type of merge is the easiest to solve but it's also the most common one. Deeply rooted conflicts (I.e. where merging has been postponed for various reasons) can, even though they are only textual, result in complex merges later on.

3 Possible solutions to merge conflicts

This section will discuss possible solutions to prevent conflicts and how to reduce the impact if a merge conflict happens.

3.1 Prevention

3.1.1 Comunication

One way to prevent software merge conflicts in your agile software development team is to improve the comunication in your team. A stand up meeting is a good time to communicate what changes are planned in what packages. Merge conflicts could be a non problem if communication between teams are done properly.

3.1.2 Update/Pull often

Many merge conflicts occure because developers forget to use the latest version of the code from the repository. The solution is very simple but also very difficult to adhere to. The development team should try to have a method to their work. If you work together, in the same room, developers that commit code can call out "commit" to make other developers aware that they have to update/pull the latest version from the repository.

3.1.3 Commit/Push often

Many merge conflicts occure because developers keep their code far to long without pushing or commiting it to the repository. While one should not commit "red" code it is also very bad to work for a couple days and not commit. The code may very well be so outdated, and cause so many merge conflicts, that you have to rewrite everything.

3.1.4 Software and Tools

Another solution is to use some kind of software or tool. This can be in the form of a plugin for integrated development environment IDE or in the form of a separet program. The idea is that the program should warn the developers if a merge conflict is possible in case a certain change is made. This warning should be enough to stop the problem before it becomes a huge problem.

3.2 Mitigation

3.2.1 Comunication

A conflict has happened because two different teams have made changes that do not agree with each other. These two teams have to comunicate a joint possible solution to the problem.

3.2.2 Stop Development

If development is continued when there is a marge conflict the problems can spread, it is a good idea, if the problem is very large, to temporarily stop development until the problem has been identified.

4 Assistance from SCM-tools

Several tools exist which attempt to prevent merge conflicts. This section will attempt to cover some of the tools which exist in order to mitigate the problem with merge conflicts.

4.1 Crystal

Crystal is an SCM awareness tool, which attempts to prevent merge conflicts in projects. As their website states "The Crystal tool informs each developer of the answer to the question, Might my changes conflict with others' changes?". [VC12]

The tool analyzes each developer's local Git repository and compares it to the other developers'. With the help of this it generates status for your own repository compared to the other's as well as master. Further technical details are beyond the scope of this paper but can be found in the user manual. [VC12]

4.2 Palantr

Palantr is a tool which attempt to prevent conflicts though awareness of simultaneous changes. Unlike Crystal which monitors each developer's commits, Palantr monitors the developer's workspaces in order to create awarenes between developers to promote communication.

From what the developers describe it works in a more preventive way than Crystal, warning developers imediatly when conflicting changes are made. It does not "predict" the outcome of future commits and merges like Crystal but rather encourages developers to solve conflicts before they even arise from an SCM-tool point-of-view. Since developers are informed at all times about any changes conflicting with the changes they are currently making it creates the urge to contact the other developer and communicate the changes being made and how they affect eachother.

The great advantage of this is that it can prevent build conflicts as well. For instance, if one developer changes the public signature of a method, this change can be compared to every other developer's changes. The other developer is quickly informed that a change to the method might be impending and can act accordingly. [SRvdH12]

Palantr is in a way a tool which supports all the methods covered in Section 3.2. Since it encourages developers to synchronize their changes with eachother while developing, as opposed to doing it after commits through resolvment of merge conflicts.

4.3 SVN Notifier

SVN notifier has a somewhat different take on the problem compared to the two previously mentioned tools. It is basically a "port" of `cvs watch` from CVS. It allows a developer to be notified whenever someone makes a change to a specific set of files. The big difference compared to Crystal and Palantr is that it does not analyze current on-going work but rather complete commits, notifying other developers that a new version is available. While this does not prevent conflicts as such it encourages team communication and avoids situations where developers go a long time without updating their workspace. [Tig]

5 Emperical study – Background

This section will provide some background related to the emperical study which was performed as a part of writing this paper.

5.1 Courses

The research done in this paper is based on the two courses Software Development in Teams and Coaching of Programming Teams. This section aims to give some background as to what the courses are and what students are taking them. Both of the courses are given as part of engineering studies.

5.1.1 Software Development in Teams (PVG)

The PVG course is given mainly for computer science students as a mandatory course in their second year as well as for some other programmes in their forth year. The course forcuses on software development in relatively small teams (By industry standards, although large compared to what the students have normally worked in previously) usint agile

methodology, eXtreme Programming to be specific. The course revolves around developing software for measuring time during Enduro[1] races, as well as sorting these times to provide an acurate leaderboard. The requirements for the software develop gradually through stories, in alignment with the XP methodology. The role of the customer is played by a professor or graduate student at the institution.

5.1.2 Coaching of Programming Teams (Coaching course)

The coaching course is given as an optional course in the fourth year for computer science students. The course is intended as a compliment to the PVG course, enabling students to take the role of coaches for the teams in the PVG course. The coaches try to contribute with some experience as well as guide the students through development but without stepping in or acting as a project manager. As a part of the course, coaches also perform an in-depth study in a field of their choosing, the result being a paper like this one.

5.2 Tools

5.2.1 Software Configuration Management Tools

The course normally uses Subversion (SVN) as configuration managment tool. Our team however uses Git as it was the only option which was compatible with the coflict awareness tool which we intended to test.

This means that the team using Crystal had a different version control tool and hence different merge tools compared to the reference team.

5.3 Choice of Crystal

The motivation for choosing crystal is that it was the most readily available tool when the study was planned. It also came recommended by our supervisor, as a part of the suggested study. Since it's Java based it was guaranteed to work in all environments used during the course (Mainly Linux and Mac OS X).

1. `http://en.wikipedia.org/wiki/Enduro`

While the study was ongoing, it was discovered that Palantr was also Java based as well as an Eclipse plugin and usable with SVN. Since Eclipse is the IDE being used on the course and SVN was the standard versioning tool this would had been a better choice with regards to the arguments presented for Crystal.

6 Empirical study

This secion covers the study of Crystal on two PVG teams, detailing the results and the difference in the number of merge conflicts and how long time each team spent resolving them.

6.1 Crystal setup

The setup of crystal was quite complex. It required a large amount of configuration which took several ours to figure out and set up on each developer's workstation. Crystal also requires access to each developer's local git repository, which means it has to have read access in the `.git` folder in all workspaces. Fortunately LTH's computer system uses a networked file system where all user files can be accessible by all other users provided they have the correct permissions. However, Git refuses to respect file permissions in the `.git` directory, resetting the permissions on the index file after each action (Commit, Pull, Push etc.). As a result, the team was forced to make aliases for each Git command where a `chmod` is executed after the command to restore the correct permissions on the `.git` folder.

Apart from all this, Crystal is also quite poor at providing usable feedback in terms of what's wrong with the configuration which meant that setting it up correctly required a lot of guesswork.

Since Crystal is somewhat old (It has not been developed since 2011) and primarily developed for Mercurial, Git was added as a part of some of the last commits on the projects, it is also somewhat lacking in terms of support for Git. Several bugs were discovered during the setup which had to be fixed by the authors (Fortunately Crystal is completely open-source). There are also features mentioned in the manual and documentation which are not available when using Git as the backing version control tool.

6.2 Crystal team

Long setup time, big investment

Merge conflicts were maybe worse overall for Crystal team, but fewer conflicts per story.

7 Conclusions

Crystal's long setup time, combined with the lack of features and support as well as the risk of more bugs means that it's a large and risky investment for a team. In our case there are not economical stakes, the point of the course is to learn about the potential risks and pitfalls in an agile software development project. As such, none of these issues affected the project negatively, however if the project was a real project, it would have to be a long one where a return on investment is given on the long setup time through the avoidance of long and difficult-to-solve conflicts.

Team maturity is low, may have affected the result. Processes are not adopted to using crystal.
Write something about Crystal not really doing much, observations of our team shows that they are not using it a lot. When they use it, they use it more or less like "git status" to check if they are in sync or not. If Crystal shows a conflict there is nothing you can do, you will get the conflict, hence it's more or less useless. Palantir might have been a better choice, as it encourages people to communicate *before* they commit or develop stuff, which is kind of the point with the PVG course.

References

[BHDN11] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Not. Proactive detection of collaboration conflicts. In *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2011.

[SRvdH12] Anita Sarma, D F. Redmiles, and Andre van der Hoek. Palantir: Early detection of development conflicts arising from parallel code changes. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 38(4):889–908, July/August 2012.

[Tig] Tigris.org. Svn notifier. `http://svnnotifier.tigris.org`.

[VC12] Crystal VC. Crystal user manual. `https://code.google.com/p/crystalvc/wiki/CrystalUserManual`, 2012. Retreived 2013-01-27.