Workspace awareness in an eXtreme Programming context

F. Gullstrand (dt07fg3@student.lth.se)
&
R. Simko (ada09rsi@student.lu.se)
Lund Institute of Technology, Lund Univeristy (LTH)

March 3, 2014

Abstract

This article consists of two parts, both analyzing how to improve workspace awareness in an eXtreme Programming (XP) project. The first part consists of a theoretical survey of the current state of workspace awareness. It covers some of the available tools, their advantages and disadvantages, and tries to make some conclusions as to which one would work the best in an XP project.

The second part is an empirical study conducted at Lund Institute of Technology, analyzing the workspace awareness tool Crystal in the context of an XP project. We find that Crystal is hard to set up and does not really reduce the amount of conflicts compared to a reference team, making it an unnecessary investment for the team. It also provides some developers with a false sense of security, increasing the risk of conflicts.

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 definitely 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 comparative 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.

2 Problem statement

This paper analyzes the occurrence of merge conflicts in agile software development in general as well as how these could be prevented. The main focus of the paper is finding what tools can be used to avoid or prevent conflicts, how these can be used and what effect they have. Because of the limited time available to perform the study it is divided in to two parts with two separate problem statements.

The first part takes a theoretical approach on the problem, analyzing previous research in an attempt to find a good tool from a theoretical point of view. This part tries to find what tool would be best suited for an eXtreme Programming project.

The second part, which is the focus of the study, consists of empirical research analyzing a specific tool. Here we try to find how well this tool works in this specific context, a course given at LTH simulating an agile development environment. The main focus in this part will be on actual saved time compared to simply handling the conflicts, answering how much this specific tool improved the effectivity of the team.

Part I
Theoretical study

This part covers some different tools available for workspace awareness and tries to draw some conclusions as to which one would theoretically be best suited for an eXtreme Programming project.

3 Background

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

3.1 The occurrence 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.

3.2 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 behavior 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.

4 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.

4.1 Prevention

4.1.1 Comunication

One way to prevent software merge conflicts in your agile software development team is to improve the communication 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.

4.1.2 Update/Pull often

Many merge conflicts occur 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 verbally call out "commit" to make other developers aware that they have to update/pull the latest version from the repository.

4.1.3 Commit/Push often

Many merge conflicts occur because developers keep their code far to long without pushing or committing 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. The easy solution is to create small tasks that developers can finish in a short amount of time and then commit.

4.1.4 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 separate 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.

4.2 Mitigation

4.2.1 Comunication

A conflict has happened because two different teams/programmers have made changes that do not agree with each other. These two teams/programmers have to communicate a joint possible solution to the problem. By making sure that the teams and programmers either sit in close proximity or by giving them good communication methods preferably with webcams the communication process is made allot easier.

4.2.2 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 all development until the problem has been identified. During this time the entire project team should work towards solving the merge conflict. This is to enable development to start up as quickly as possible.

5 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.

5.1 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]

5.2 Palantír

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

From what the developers describe it works in a more preventive way than Crystal, warning developers immediately 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 each other.

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]

Palantír is in a way a tool which supports all the methods covered in Section 4.2. Since it encourages developers to synchronize their changes with each other while developing, as opposed to doing it after commits through resolving merge conflicts.

5.3 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 Palantír 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]

6 Conclusions

Based on the theoretical data provided above, if a tool was to be used Palantír seems to be the best in this case. The integration with Eclipse provides developers with direct knowledge without the need to switch between programs. It also analyzes code as it's being written, as opposed to analyzing what has been committed like SVN Notifier or Crystal does. While Crystal could also be a working solution if the local repository is used heavily it still requires the installation and configuration of a separate program.

Part II
Empirical study

This part 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.

7 Background

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

7.1 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.

7.1.1 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 focusses on software development in relatively small teams (By industry standards, although large compared to what the students have normally worked in previously) using 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 accurate 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.

7.1.2 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.

7.2 Tools

7.2.1 Software Configuration Management Tools

The course normally uses Subversion (SVN) as configuration management tool. Our team however uses Git as it was the only option which was compatible with

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

the conflict awareness tool which we intended to test. While they differ in many ways, including the fact that Git is distributed, meaning that each developer holds his own copy of the entire repository, for the intents and purposes of this study they are quite similar. Throughout the course, Git was used in the same manner as SVN would be, doing a `git commit -a` followed by a `git push`, immediately sending the commit to the remote server, negating the local repository function.

There is however one major difference, and it is that of automatic merging. Git uses the change-set model meaning that each commit is tracked individually. [Fei91] This also means that merges are tracked between branches and since each developer's repository is considered a branch, merges are tracked between developers as well. The end result is that the same merge will never have to be made twice, which could be the case in SVN. However, regardless of which merge tool works the best, the fact that the teams use different tools will be a source of error.

7.2.2 Choice of Crystal

The motivation for choosing crystal is that it was the most readily available tool when the study was planned. It was also 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).

While the study was ongoing, it was discovered that Palantír 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.

8 Results

8.1 Crystal setup

The setup of crystal was quite complex. It required a large amount of configuration which took several hours 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.

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.

8.1.1 Computer environment

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.

8.2 Occurrence and severity of conflicts

Overall the Crystal team had more numerous and more severe conflicts than the other team. (Table 1 and 2) Initially the Crystal team implemented more stories than the reference team, perhaps resulting in the longer and more common conflicts. As a result it was decided to record a few averages based on the number of implemented stories. (Table 3 and 4). The initial thought was that this would lead to the Crystal team improving when comparing the average results. However, as can be seen in the tables, this was not the case.

Another experiment which was made during the study was to skip the usage of Crystal during two weeks for the Crystal team in order to find if this made a difference. As can be concluded from Table 3 this does not appear to have made a large difference in the resolution time per story. The results of the first week may be skewed, however, because the teams where getting to know each other. The average resolution time per story for Week 2 and 3 is 3.57 minutes and the time for Weeks 4 and 5 is 8.71 minutes, a more than two-fold increase. If the first week is included the average for Crystal is raised to 6.67 minutes, while still being lower than when not using Crystal the difference is much smaller.

8.3 The developers' opinion

Towards the end of the study the developers using Crystal were asked to voice their opinion on how it worked. Overall people were not happy with how it worked; with replies ranging from that it did not help, to that it actually made conflicts worse. This because Crystal would tell the developer that it was OK to pull changes when in fact they would cause a conflict, hence the tool was not always corrent. While this neither increases nor decreases the number of conflicts (The merge would have to be made either way) the developers' perception was that Crystal made it worse.

They did however say that it might had worked better if the team was distributed, since in that case direct communication becomes harder. It would also require developers to be more active with their local repository, committing more often without pushing. The developers in this team usually did commit and push simultaneously, in the same way one would had done in a Subversion repository. They also viewed Crystal as a nice reminder that it is important to update regularly, however one of the developers stated that a regular alarm clock would had filled the same role.

8.4 Observation of the team

The authors observed the Crystal team during their development in order to build an outside view of how it was used. The observed usage was that Crystal filled the function of `git status` more than anything else. Developers used it to check if they were up to date or not. This function should of course not be filled by a tool intended to prevent conflicts but instead by the SCM tool's build in function.

9 Discussion

Limited sample size causes some problems with the results. It is not possible to make a definitive assessment of the program. More research and testing is needed in the future with different groups using crystal under longer periods of time. Five 8-hour work days was not enough to get used to using Crystal.

Another issue is the low team maturity, which may increase the amount of confusion and thereby conflicts. While both teams are equally immature, the Crystal team had no preparation using neither Git nor Crystal, whereas all teams were prepared to use Subversion. The combination of having to use two new tools of course makes the issue worse which could explain why the reference team saw fewer and less severe conflicts.

9.1 Crystal setup

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 no 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.

Thanks to the file system, the issues with Crystal requiring read permissions in all developer's repository was solved through use of the networked file system. Crystal offers a few other ways of sharing your repository with co-workers in their manual, Dropbox and HTTP sharing. [VC12] This however presents its own issues, Dropbox has no read-only sharing, meaning that another developer can modify your workspace or cause conflicts. Add to this the obvious issue of putting one version controlled repository into another type of version control tool, using them both at the same time, and the result is most likely a mess. The option of HTTP sharing is convoluted at best and requires a HTTP server to be running on each developer's machine and for it to be accessible to from every other developer.

For obvious reasons the only option if one were to use Crystal in an industry project would be to have a network file system and having each developer set the correct permissions on their local repository.

9.2 Developer's usage of Crystal

As mentioned when observing the team, it was found that developers do not user Crystal in the way it was intended. The result is that the data wasn't necessarily representative of how Crystal worked as it may not have been used in the way it was intended.

In retrospect, Palantír may have been a better choice as it is not only easier to use in this case (Because of the Eclipse integration). It also seems to encourage people to communicate *during* development as opposed to after a commit is made as is the case with Crystal.

9.3 Data analysis

While the amount of conflicts increased when the Crystal team stopped using Crystal it is hard to draw any conclusions from this because of the limited sample size. The compared iterations are only two sessions of eight hour programming with Crystal and two equal sessions without. The variation seen may as well come from variations between sessions, such as some of them containing a release process and similar external events.

10 Conclusion

Firstly more research has to be done to get a clear definitive answer as to the benefits of using programs such as crystal.

However with the opinions of the developers in mind we can conclude that Crystal may be beneficial in very specific cases but is mostly no benefit at all, and in some cases may even lure the developers into thinking that they are safe, with no risk of merge conflicts, where there in fact will be large merge conflicts in case there is a push to the repository. The data also confirms this, with fewer merge conflicts in the reference team, meaning that there must be other factors affecting the number of conflicts.

We can, therefore, not recommend using crystal at this time. Instead the team should try to focus on a good work method and good communication since this not only solves merge conflicts, but may also be beneficial in other cases.

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.

[Fei91] Peter Feiler. *Configuration Management Models in Commercial Environments*. Technical report, Software Engineering Institute, Carnegie Mellon University, 1991.

[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.

Apendix

	1 min	2 min	5 min	20 min	40 min	1 h	2 h
Week 1	3	1	1	0	0	1	0
Week 2	2	0	1	0	0	0	0
Week 3	1	1	0	2	0	0	0
Week 4 (Without Crystal)	1	0	2	0	1	0	0
Week 5 (Without Crystal)	1	2	1	0	0	0	0

Table 1: Crystal team resolution time per week

	1 min	2 min	5 min	20 min	40 min	1 h	2 h
Week 1	3	0	0	0	0	0	0
Week 2	4	0	0	0	0	0	0
Week 3	0	0	1	0	0	0	0
Week 4	0	0	0	0	0	0	0
Week 5	0	0	0	0	0	0	0

Table 2: Reference team resolution time per week

	Number of completed stories	Number of conflicts	Numer of conflicts per story	Total resolution time	Resolution time per story	Average resolution time
Week 1	4	6	0	70 min	17.5 min	11.67 min
Week 2	7	3	0.43	7 min	1 min	2.33 min
Week 3	7	4	0.57	43 min	6.14 min	10.75 min
Week 4 (Without Crystal)	5	4	0.8	51 min	10.2 min	12.75 min
Week 5 (Without Crystal)	2	4	2	10 min	5 min	2.5 min
Total	25	21	0.84	181 min	7.24 min	8.62 min

Table 3: Crystal team average results

	Number of completed stories	Number of conflicts	Numer of conflicts per story	Total resolution time	Resolution time per story	Average resolution time
Week 1	1	3	0	3 min	3 min	1 min
Week 2	6	4	0.67	4 min	0.67 min	1 min
Week 3	2	1	0.5	5 min	2.5 min	5 min
Week 4	3	0	0	0 min	0 min	0 min
Week 5	7	0	0	0 min	0 min	0 min
Total	19	8	0.42	12 min	0.63 min	1.5 min

Table 4: Reference team average results