Evaluation Framework for Workspace Awareness Tools

Sebastian Nyberg, Christian Olsson

March 6, 2015

Abstract

Conflicts arising from merging different versions of a shared project can be very costly and can in some cases persist for a long time. With that in mind it is logical to try prevent merge conflicts and to lessen their severity. Workspace awareness tools can help here by providing information about what is happening as it is happening so that developers can take actions as early as possible. Using a medium sized team of fairly inexperienced programmers in a small XP project and existing studies, we design an evaluation framework for workspace awareness tools. The team used the awareness tool Crystal to experience such tools first hand and provide us with valuable feedback.

1 Introduction

In the modern software development industry, one of the most challenging — if not *the* most challenging — aspect is that of coordination of work. Coordination involves not only the assignment of work but also keeping everyone aware of the constantly changing codebase. Awareness is often limited to the private workspace, and with private workspaces we get the double maintenance problem [3] where things risk to grow apart more and more as time passes. This makes coordination painful. Unfortunately there isn't a "standard time before coordination is needed" — so what we need is *global awareness*.

In order to assess these awareness difficulties, workflow models have been proposed and some of them are now implemented into SCM tools. In addition to the revision control systems, there are other SCM tools such as Travis CI [2] and gitk [1] which facilitates concurrent work by providing graphical feedback on the status of a project. In short, these tools aim to keep the individual developers up-to-date with the progress of the rest of the team.

But even if the current status of the project is readily available for all developers, there must be traceability of modules. Without this, there is no way for the developers to effectively communicate with whoever made the change. And more importantly, what ultimately causes this need for workspace awareness is the risk of conflicts. The early detection of conflicts is key to avoid major merge problems when introducing new functionality.

This is where workspace awareness tools come into play. Not only direct conflicts caused by conflicting entries in the same section of code, but also indirect conflicts caused by dependencies between modules need to be detected as soon as possible. The most thoroughly tested of these tools, Palantir [7], showed a statistically significant increase in productivity by providing this functionality.

Crystal was introduced in 2011 [5] — providing realtime access to conflict-assessment by comparing the local workspace to that of others. In essence, it refreshes the other developers'

workspaces and tests for direct (merge) conflicts. In addition, Crystal can also be set up to show when a merge would cause a project to fail to build or to fail unit tests. When a potential merge conflict is present, both developers are presented a visual cue in the Crystal application window. The window also provides additional information such as when the local workspace is in need of an update.

By introducing Crystal to the course project group (henceforth referred to as the PVG group), we gain knowledge of the effects of using such a tool. Because of the varied backgrounds of the team members and limited time-scope, we have performed a qualitative study where each individual member was interviewed for their experience with using such a tool. We combined this information with the existing studies to design a framework for evaluating workspace awareness tools. Specifically, the results of the study helped us focus on the *presentation* of data.

In the next section, we give an overview of the fundamental concepts of workspace awareness; addressing the current means of detecting and solving conflicts as well as other methods of improving awareness. We then present the results of using Crystal in the PVG project, combining our observations with the result of the interviews. Lastly, we present our evaluation framework and discuss future work that can be done on the subject.

2 Background

The first piece of workplace awareness is the preemptive minimization of potential conflicts. There are many factors that come into play when minimizing risks: Impact analysis, division of code into independent modules, expandable architecture, XP practices such as continuous integration, pair programming and collective code-ownership. This study is however focused on the awareness of *what* went wrong, *where* it went wrong and from *whom* the conflict originated.

Conflicts are normally divided into two groups: Direct and indirect conflicts. In order to illustrate these, we have the two developers Alice and Bob who are working on the same project described outlined in Table 1. Their project contains five methods, m1, ..., m5, which have the methods they are calling in the reference set. On the right side are some tasks that will result in changes for the methods in their change sets.

Methods		**Tasks**	
name	reference set	name	change set
$m1$	$\{m2\}$	$t1$	$\{m3, m4\}$
$m2$	$\{\emptyset\}$	$t2$	$\{m1, m2\}$
$m3$	$\{m4, m5\}$	$t3$	$\{m1\}$
$m4$	$\{\emptyset\}$	$t4$	$\{m4, m5\}$
$m5$	$\{m2\}$		

Table 1: Alice and Bob's project setting. The reference set includes all references made to other methods, which could also be seen as the *dependencies* for a method. The change set is the set of methods that will be changed during the implementation of a task

2.1 Direct conflicts

A direct conflict can occur in **Scenario 1:** Alice picks up task **t2** and Bob task **t3**. This because both **t2** and **t3** includes changes to the method $\{m1\}$:

1. Bob's changes are less extensive and once he is done, he commits and pushes to the remote repository.
2. Once Alice is done with her task, she attempts to push aswell, but is stopped by the SCM tool and asked to update her workspace. This requirement to sync the local workspace with the remote repository is the first line of defense against direct conflicts and overwritten simultaneous updates.
3. The SCM tool reports that there is a merge conflict (a direct conflict) since both Bob and Alice have written code in the same place ($m1$).
4. Alice uses a merge tool, marks the conflict as resolved, and pushes her changes.

In order to detect a merge conflict, the SCM tool compares the locally committed changes to those pulled from the repository on a line-by-line basis. A workplace awareness tool that has the ability to detect direct conflicts pre-commit does exactly the same, only before the commit has been made. This means that a realtime comparison must be made between each developer's workspace and the remote repository.

2.2 Indirect conflicts

Indirect conflicts arise from dependencies between methods. For this description we exclude indirect conflicts caused by external dependencies such as imported libraries.

Scenario 2: Alice picks up task **t3** = $\{m1\}$ $(\underbrace{m1 \rightarrow \{m2\}}_{dependency})$, Bob picks up task **t4** = $\{m4, m5\}$ ($m4 \rightarrow \{\emptyset\}$, $m5 \rightarrow \{m2\}$).

1. As Bob is implementing task $t4$ he realises that changes need to be made in $m2$ in order for his new functionality to work in $m5$ which has the reference set $\{m2\}$. He happily makes these changes, finishes up the work in $m4$, commits and pushes his changes.
2. Alice quickly finishes her task by implementing the listed changes in $t3$ which are in $m1$. She then commits and tries to push her work.
3. The revision control system prompts her to pull the updates, and after doing so, the build fails.

At this point, there is nothing the revision control tool could've done to realise that Bob's changes in $m2$ made Alice's new functionality in $m1$ result in a bug. This happens because $m1$ remains intact between the two revisions.

There are two ways to detect indirect conflicts: by a failed build or a failed integration test. A failed build is the most common way of being alerted of a newly introduced bug. The second way — integration testing — relies on extensive and thorough testing, something that is implemented in Test-Driven Development (TDD), but often overlooked.

In this example Alices's changes was only in one method. It is then inherently easy to track down where her problems originated from by looking at the reference set. But let's

say that instead there were a hundred methods that had been changed by Alice, and at least ten of those had new changes introduced in their reference sets. There would be a high risk of indirect conflicts, even with continuous integration.

This is where more complex workspace awareness tools come into play. By using syntactic differencing, static analysis and semantic analysis, indirect conflicts can be detected early.

When the tool discovers a potential indirect conflict — that is, a remote uncommited change that when combined with the local changes would result in a build- or integration test failure — it presents this information to the subject in the form of a visual cue. This way, effective communication is possible before-the-fact as both the developers will notice that their changes are colliding. Detecting an indirect conflict early on saves a lot of time and headaches for both parties.

2.3 Difficulties with detecting indirect conflicts

The whole idea of detecting conflicts early relies on being able to compile and run tests on the combined workspaces of the developers. Depending on the amount of developers working within the same scope e.g. within a certain branch or module, and the size of the codebase, this could require a lot of computation for each workspace involved.

In order to alleviate this issue, the tool should perform less computation-heavy static analysis of the projects. By keeping an up-to-date structure of modular, class- and method-specific dependencies, the locally made changes and their dependencies can be compared with the remote delta for potential conflicts. When a possible conflict emerges, the build or integration testing takes place to determine whether this change actually introduced a conflict or not.

2.4 Project Setting

This study was conducted in conjunction with the course "Software Development in Teams - Project". That course takes a group of inexperienced programmers (in our case 14 programmers) and tries to simulate an eXtreme Programming project from start to finnish. Almost all the programming is done during "long labs" where they program from 8-17 once a week exluding lunch and small breaks. In addition, they have four hours of weekly unscheduled "spike" time that are used for acquiring useful knowledge, trying new technology, and other things which are useful to the team. The project they are tasked with is to make a program that can measure times during Enduro competitions and new requirements are added weekly.

3 Crystal

Crystal is an open source tool written in Java that can be used with git or Mercurial projects. It shows the relationship between your workspace and other team members' as well as the upstream repository using icons as can be seen in figure 3. Crystal gives information about what would happen if you pulled from each particular remote repository (or if they pulled from you). A green arrow would mean no merge necessary, yellow means a merge that can be solved automatically, and red means that the merge requires manual intervention. The shading of the icons indicates who can change the relationship. More info about what the icons mean be can found in the manual [4].

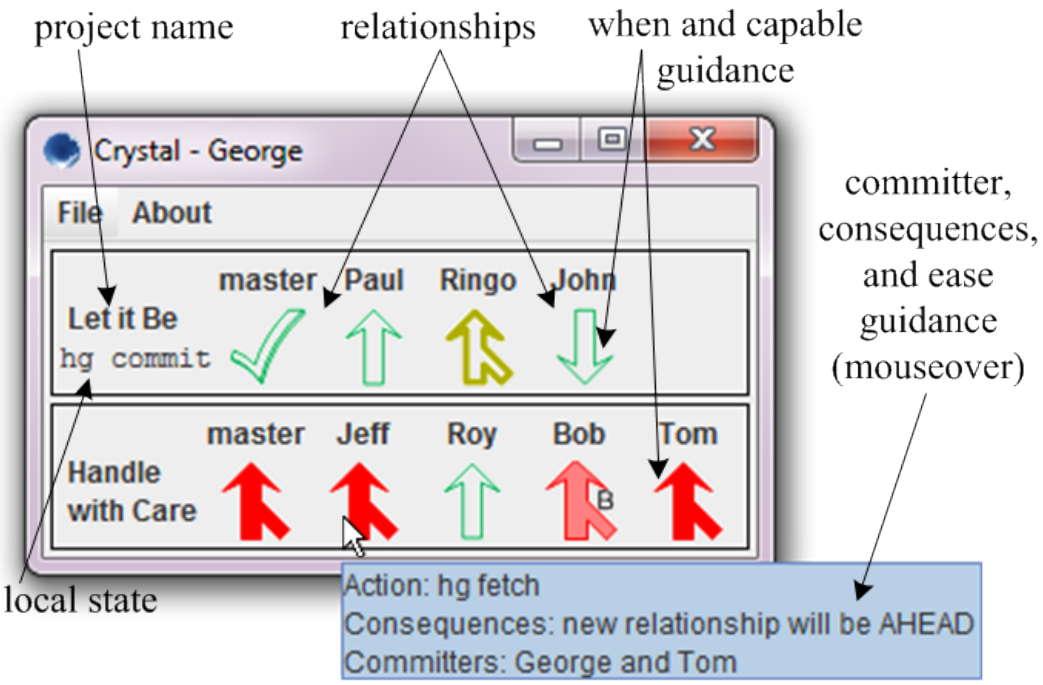

Figure 1: Screenshot of Crystal [4].

Figure 2: Another screenshot of Crystal taken during the study. The crosses mean that Crystal has encountered an error.

You can configure Crystal to use a build script and test suite which allows Crystal to check for indirect conflicts (assuming you have good test coverage) so you can see when a clean textual merge would result in build failures or when it would cause the code to not pass the tests. Crystal only looks at commited changes so the usefullness of Crystal depends on how often you commit.

3.1 Setup

The first spike we assigned for the team was to setup Crystal. We created a short list of steps needed so that the setup would go smoothly. This list was not entirely correct or complete due to the limited amount of time we had to prepare. We reasoned that only half of the developers needed a workspace with Crystal since they do pair programming so the rest could simply share. We then arranged a meeting where we would be available in case the team needed help. Because we had read "Workspace awareness in an eXtreme Programming context" [6] we were aware of the problem of git resetting the permissions for the repository on every action. That is to say, Crystal requires that each workspace's .git folder's permissions are set properly so the other team members can read it. To alleviate this we had each Crystal user create a script that ran chmod after every git action. Even with all the knowledge we had from reading manuals and Rickard's and Simko's work it still took a couple of hours to configure everything and make sure it worked.

3.2 Issues During the Project

Crystal stopped working for most people when most developers switched to another branch to do refactoring. This happened because Crystal keeps local copies of all the workspaces in order to test if automatic merges will succeed or not. These local files will then update themselves by their corresponding remote. The problem occurs because Crystal will try to update the local files to the newest commit on the branch specified in the settings file and somehow Crystal managed to have the remotes get merge conflicts with themselves. Clearing these local files and letting Crystal do a fresh pull fixed that problem.

Another issue that came up during the project was people who switched to another working directory without the other team members changing their Crystal configuration to match. This meant that Crystal couldn't get the correct status.

A third issue was probably related to permissions. Crystal tried to get data from other team members repositories but failed for unknown reasons.

4 Qualitative Study - Crystal

As we have mentioned before, we have used Crystal in the PVG project to help ascertain a good framework for evaluating workspace awareness tools. We hope that Crystal will provide a good baseline for what makes this type of tool good. A smiliar project [6] was conducted last year, again using Crystal, where one of the main questions was: "Does Crystal reduce the amount or lessen the severity of conflicts?". It is very interesting to compare the results of what the developers thought last year and compare that to what our new team think today. Because of the limited amounts of data that could be gathered during such a short project and with the difficulties of getting good data from other teams we felt it was better to more of a qualitative study.

4.1 Method

We started the project by helping the students set up Crystal so they could see each other's workspaces. During the active development portion of the PVG project we offered some support regarding Crystal and we tried to encourage the students to take full advantage of Crystal. As the project was nearing its culmination we conducted interviews with the team members to find out what they thought about Crystal in particular and what they would want from workspace awareness tools in general.

4.2 Results

- How did things go with Crystal?

  Most people were displeased with Crystal. It stopped working for many people when they switched to a different branch. Some people liked it in the beginning but then gradually gave up due to problems. A few persons remarked that the program updates its status too seldom. A common complaint was that the icon in the taskbar did not show the correct relationship in relation to the master branch on github[1]

- How often do you look at Crystal?

  The answers people gave here can be divided into a couple groups. There were those that gave up very early and then very seldom (or never) checked, there were those that had the program always on top, and those who mainly looked at the taskbar icon and only rarely looked at the mainscreen.

- How has Crystal affected how often you pull?

  The general consensus here was that Crystal was a good reminder to pull in the beginning. After a while they got into the habbit of pulling often and then they felt like it didn't affect them that much. In Simko and Gullstrand's paper [6] they got similar results and suggested that the same function could be served by a timer.

- How has Crystal affected how often you commit?

  Everybody agreed that Crystal did not have an effect in this regard. Most people simply did commit and push at the same time when they were done with a task.

[1] What this icon actually represents is the "dominant" relationship. The dominant relationship is basically the most serious one which might not be the relationship to github

- How has Crystal affected the number of merge conflicts?

  The consensus here was that if Crystal had an effect, it was not very noticable.

- What is your opinion regarding the icons in Crystal?

  Almost everybody thought the colors were pretty logical. Almost nobody understood what the shading meant. There were some confusion about what the direction of the arrows meant.

- If Crystal existed as a plugin to Eclipse, how would that affect your workflow?

  Almost everyboody thought it would be more convenient if Crystal were a plugin. They remarked that the information would be more visible. One person was afraid it would make Eclipse to cluttered.

- If Crystal were a plugin, where in Eclipse would be a good place to put that kind of information?

  If we try to distill the answers down to a concept it would be something like: put it somewhere to the side where it wouldn't be in the way. Maybe have the information as a tab like the console is. Or have it only be visible in a particular perspective.

- Do you think there is a need for this type of tool?

  Some people thought it was a good general idea. Some thought it was unecessary because there exists good branching patterns that solve the problem well enough.

- If you wanted to leave a suggestions for the authors of Crystal, what would you recommend they improve?

  One suggestion was to have a common config-file between all members of the team. This would allow you to for instance only have to add a new workspace once and it would update for all the team members. Doing the initial configuration was a bit tricky and a couple of persons thought that could be made simpler. Crystal could definitely be made more robust. When we switched branches it required manual intervention to make Crystal work again. A setting where you could switch the orientation of the relationships would let Crystal take up far less useful space on widescreen monitors. Easily accessible help straight in the program regarding the meaning of the icons was also requested. Many thought that making Crystal into an Eclipse plugin would help a lot.

4.3 Discussion

The general impression we got during the interviews were that those who did not help set up Crystal did not understand it that well and they quickly gave up on using Crystal. Having a short introductory lesson about how Crystal works, what the icons mean, and explain better why this type of tool is useful would probably help a lot. That Crystal only works on changes that have been commited is something we should have brought up earlier and put more emphasis on.

Unfortunately it seems that a lot of the students did not read the user manual like we asked them to. If they understood the tool better they would probably thought Crystal was more useful. If there were good explanations for the icons included with Crystal that might have alleviated the problems a bit.

That good branching patterns would be a good solution to the same problems that workspace awareness tools are supposed to solve is something we had not considered earlier. This is definitely something one could look into more. That Crystal works so poorly with branching is very unfortunate since you then cannot combine the solutions easily.

5 Evaluation framework

The evaluation framework is split into two sections: workspace status accounting and visual presentation of information. The workspace status accounting can be seen as the functionality of the tool, mainly based on the background section in this paper. For the second section, we focus on our results from the qualitative study to list the most important characteristics of visual presentation. In order to reduce redundancy, we do not include software quality metrics such as robustness or update frequency.

5.1 Workspace status accounting

These variables focuses on the functionality of the program rather than the presentation of information

- **Remote repository checking**. This is the most basic level of workspace status accounting. It simply checks the current relationship between the local workspace to the remote. When the two are out-of-sync, the developer is prompted to pull the remote changes. This could result in a direct or indirect conflict, but since the developer has not been made aware of this conflict before it has been committed, there is no real difference in regards to the timeliness of conflict resolution.
- **Direct conflict checking between workspaces**. The tool uses some sort of analysis on the workspace-pairings in order to detect direct conflicts before they have been pushed to the remote repository. This could be done both pre-commit and post-commit, depending on the tool and/or the developers individual settings.
- **Indirect conflict checking between workspaces**. The tool analyzes the workspace-pairings for detection of indirect conflicts. This detection could be the result of either

  a) A change that was made to a method within the dependency set of a locally modified method. A simple example of this is if Alice just made a new method m_1 which calls a method m_2. Bob just removed method m_2 since nobody was using it (and Alice's usage was not yet in the remote repository).

  b) The integration tests for the combined workspaces failed.

  c) The compilation for the combined workspaces failed.

5.2 Visual presentation of information

We noticed during our qualitative study how important the visual presentation is. As with any software engineering philosophy or principle, there ought to be a group of naysayers, whom despite positive studies on the subject do not feel the need to change up their development environment. As such, the visual presentation needs to be seamless and unobtrusive.

- **Dynamically allocated notification area**. By dynamically allocating the visual area in which the developer is notified of conflicts, the presented cues are limited to information that is relevant. In Crystal, the developer is constantly made aware of everyone's workspace status. When there is no way for the developer to impact the relationship, this information is simply superfluous.
- **Integration into the development environment**. By integrating the status presentation into the development environment, there is no need to run external programs or switch between windows.
- **Human readable conflict information**. The presented conflict information should not simply be the output of a build failure or CVS status command. Once a conflict has appeared, the developer needs to know two things: *what* went wrong and *who* cause the conflict. It is more likely to recieve valuable information by asking the party involved in the conflict than looking through a stacktrace from Java. A good example of this is the presentation of conflicts in Palantìr [7] which includes a short message of what the problem is e.g. "Bob deleted method getValues()".

6 Conclusion

The use of Crystal in our study did not show the impact we had anticipated. This can be attributed to many factors such as lack of understanding for the tool and its use, a poorly implemented visual presentation or simply the sheer amount of bugs and issues we experienced. Either way, as coaches, there are a lot of things we could've done different.

When performing the interviews, we realised how few actually understood *why* we were doing this study. Without properly educating the project team on the reasons for introducing Crystal, a lot of the students felt that it was simply being forced upon them as more or less bloatware.

Additionally, we assumed that the students would read the manual for Crystal when it was assigned to them as homework. This was not the case however, and since we did not actively enforce the use of the tool, some simply ignored it instead.

Our conclusion is that what we lacked was an introductory presentation to establish some knowledge about the tool and our study. After the presentation, we should have given the students an active choice to use Crystal or not. Ideally, we should have set up the workspaces for the students aswell, but it was not possible within the timeframe of this study.

We are very surprised by the lack of flexible workspace awareness tools in active development. Even though the use of Crystal could've gone much smoother, the majority of our interviews showed that workspace awareness is an issue, and we have yet to find a well-rounded workspace awareness tool.

References

[1] Git - gitk documentation. `http://git-scm.com/docs/gitk`.

[2] Travis ci - free hosted continuous integration platform for the open source community. `https://www.travis-ci.org/`.

[3] Wayne A Babich. *Software configuration management: coordination for team productivity*. Addison-Wesley, 1986.

[4] Yuriy Brun, Reid Holmes, Michael D Ernst, and David Notkin. Crystal user manual. `https://code.google.com/p/crystalvc/wiki/CrystalUserManual`.

[5] Yuriy Brun, Reid Holmes, Michael D Ernst, and David Notkin. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 168–178. ACM, 2011.

[6] F Gullstrand and R Simko. Workspace awareness in an extreme programming context. Technical report, 2014.

[7] Anita Sarma, David F Redmiles, and Andre Van Der Hoek. Palantir: Early detection of development conflicts arising from parallel code changes. *Software Engineering, IEEE Transactions on*, 38(4):889–908, 2012.

A Interview Questions

- How did things go with Crystal?
- How often do you look at Crystal?
  - At the mainscreen?
  - At the taskbar?
- How has Crystal affected how often you pull?
- How has Crystal affected how often you commit?
- How has Crystal affected the number of merge conflicts?
- What is your opinion regarding the icons in Crystal?
- If Crystal existed as a plugin to Eclipse, how would that affect your workflow?
- If Crystal were a plugin, where in Eclipse would be a good place to put that kind of information?
- Do you think there is a need for this type of tool?
  - If yes, under what circumstances are they especially good?
  - If no, why not?
- If you wanted to leave a suggestions for the authors of Crystal, what would you recommend they improve?