Università
della
Svizzera
italiana

**Faculty
of Informatics**

USI Technical Report Series in Informatics

A qualitative analysis of preemptive conflict detection

Lile Hattori[1], Michele Lanza[1], Marco D'Ambros[1]

[1] REVEAL @ Faculty of Informatics, Università della Svizzera italiana, Switzerland

Abstract

Preemptive conflict detection is the act of detecting a potential merge conflict at an earlier stage than at check in time, and informing the involved developers about it. Researchers have proposed a number of tools and techniques to detect potential merge conflicts. However, a limited number of studies have been conducted so far to investigate whether the adoption of such tools and techniques brings benefits to developers. We have conducted a qualitative user study to understand how developers behave when dealing with merging and how this behavior changes when they are exposed to preemptive conflict detection. We report on the analysis of the data collected in the user study, as well as the discussion on the findings derived from the analysis.

Report Info

Published

Number
USI-INF-TR-2011/05

Institution
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland

Online Access
www.inf.usi.ch/techreports

1 Introduction

Teamwork plays an important role for a successful delivery of a software system [9]. A team of developers must deal with parallel development and manage the dependencies between their activities. This involves not only the adoption of development processes, norms, methodologies, and tools [4], but also human communication, coordination, and collaboration.

An important aspect of team collaboration is awareness, defined as an understanding of the activity of others that provides a context for one's activity [11]. In the context of software development, awareness is seen as a means by which team members can become aware of the work of others that interdepends with their own [8]. As a concrete example, suppose a developer is implementing a function that is dependent on another one, which is in turn undergoing a semantic change. This developer should be aware that the function his code is calling is under change, because this can impact the behavior of his function.

In a collocated team, awareness is mainly obtained through human interactions, such as meetings, informal conversations, overhearing conversations [12], and helping colleagues. In a distributed team, however, the distance among developers or teams constitutes a barrier for informal interactions, and the awareness of the activity of the team is consequently lower than in collocated teams. Some of the problems that arise with a low awareness level are communication breakdowns [8], lack of willingness to help others, and delays on deliveries [19].

Recently, there has been a significant effort from the software configuration management (SCM) community to increase awareness of distributed teams by supporting coordination across multiple developers working in parallel in the same code base [2, 14, 18, 25]. These approaches to promote *workspace awareness* preempt SCM systems, by detecting in real time concurrent modification to software artifacts, especially those that are potentially conflicting: concurrent changes that are likely to cause merge conflicts at check in time.

A limited number of studies [2, 10, 25] have been conducted to evaluate whether the adoption of tools to promote workspace awareness are beneficial to developers. Their initial findings suggest that, when preemptive conflict detection is introduced, the frequency of communication increases, there is a reduction in overlapping work, and an increase in the detection and resolution of conflicts. However, some fundamental questions concerning these approaches remain open: Were the changes observed in these studies beneficial to developers? Did the developers' strategies to deal with merging change? Is the information being delivered disrupting? Would they prefer to get them in a different way?

We have conducted a qualitative user study to understand how developers behave when dealing with merge, how this behavior changes when they are exposed to preemptive conflict detection, and whether this change is beneficial to them. We also investigated how developers prefer this information to be delivered, by exposing them to two different ways of visualizing emerging conflict and collecting their opinion. The research questions we investigate in this study are:

RQ1 How do developers behave when they have to merge code and resolve conflicts?

RQ2 How does this behavior change when information of emerging conflicts is present?

RQ3 How do developers perceive different approaches to deliver information on emerging conflicts?

We have collected data from observations, interviews, and questionnaires, and analyzed it iteratively by abstracting the findings to obtain a summary of the most important ones. As a highlight of our findings, we have observed a change on the frequency and depth of communication, as well as a change on the strategies to merge code when developers are exposed to preemptive conflict detection. These changes were beneficial to developers, since they successfully dealt with merging, in contrast with the struggle to merge code without the information on emerging conflicts.

Structure of the document. In Section 2 we review the related work on tools and mechanisms to detect conflicts, to then discuss the evaluation performed so far. In Section 3 we present our technique to detect emerging conflicts, and the different ways it can be visualized on the IDE. In Section 4 we describe the design of the qualitative user study we conducted. In Section 5 we present the analysis of the data collected, to discuss our findings in Section 6. We finally summarize our findings and discuss the lessons learned in this study in Section 7.

2 Related Work

Although there have been efforts to help developers to detect and preempt merge conflicts earlier than at check in time, only few of them evaluate their strategies and the developers' behavior when merging and using preemptive conflict tools. We first present the related work on tools and mechanisms to detect conflicts, to then discuss the evaluation performed so far.

2.1 Conflict Detection Tools

To develop software in parallel and coordinate themselves, software developers use coordination tools, such as SCM systems and bug tracking systems, as well as communication tools, such as emails and IRC channels. SCM systems let developers code in parallel by allowing them to check out the project to their private workspace, perform changes, and check them into the repository. The coordination model provided by SCM systems is a tradeoff between working in isolation and being aware of the team activity.

Conflict detection tools aim at complementing SCM systems by increasing the awareness of the activity of the team, yet maintaining the workspace isolation – other developers will only have access to a developer's new code when he checks it in. These tools can be classified into two categories: *after* check in and *before* check in.

2.1.1 Conflict Detection After Check in

Tools belonging to this category aim at detecting conflicts after one has checked in new code to the repository. They have the advantage of reducing the number of false positives (conflicts that are detected but do not exist at check in time) with the price of detecting them at a later stage.

The seminal work of O'Reilly et al. [21] improves conflict detection in Concurrent Versioning System (CVS) by extending the *watches* – a mechanism that permits users to request notification of edit, un-edit and commit actions on files managed by CVS that originates from other users. Their tool, Night Watch, collects the notification of a total of 12 events (including the 3 native ones) and checks for possible conflicts across workspaces.

Brun et al. [3] propose a similar solution with a stand-alone tool, called Crystal, which detects conflicts from Mercurial. Crystal detects seven distinct states a repository can be in: same, ahead, behind, merge, textualX, buildX, and testX. The latter three happen when distinct change sets cannot be automatically merged, when a build fails after a merge, and when tests break after a merge, respectively.

2.1.2 Conflict Detection Before Check in

These tools aim at detecting conflicts as early as possible, before new changes are checked in the repository. They have the advantage of warning developers as soon as conflict arise, allowing them to take preventive actions to avoid dealing with resolving conflicts at check in time. However, they can produce false positives, or short-lived conflicts that might disappear before the check in.

Palantìr [24, 25] is an Eclipse plug-in that listens to source code changes at a developer's workspace, and checks for emerging conflicts. Palantìr is able to detect direct conflicts – concurrent changes to a single file – as well as indirect ones – changes to a file that might affect another one that depends on it. It enriches Eclipse's workspace by adding visual cues on the package explorer to show which files might be conflicting, and providing a view with the detail of each potential conflict, such as authors involved, location, and severity of the conflict. The difference between Palatìr and Night Watch is that the first can preempt conflicts before changes are checked in, while the latter only detects them afterwards.

FASTDash [2] and CollabVS [18] offer different types of awareness information, from which one of them is conflict detection. FASTDash shows who has source files checked out, which files are being viewed, and what methods and classes are currently being changed. When two programmers are editing the same source file, it immediately notifies them of a potential conflicting situation. CollabVS's stronger contribution is to provide different communication channels, such as instant messaging, collaborative code editing, and audio/video sharing. It notifies developers of concurrent dependent changes, which can be used by them to prevent conflicts. The main difference between these two tools is that FASTDash is a stand alone application, which brings the overhead of switching between applications to be aware of what is happening, while CollabVS is an extension for VisualStudio, thus working within the IDE.

Guimarães and Rito-Silva [14] propose a solution that goes beyond awareness of potential conflicts. When the project is checked out, the *real-time integration* creates a merge workspace, which is shared among developers and continuously integrate the changes made by them in their private workspaces. As conflicts emerge, the merge workspace reports them to the affected developers. Conflicts are found in two different ways: direct conflicts are detected when parallel revisions of the same element cannot be merged; and other conflicts are detected every time the merged system is rebuilt (recompiled and retested).

Lighthouse [7, 23] and its extension, CASI [26], are two loosely related work. Their philosophy towards merge conflicts is to prevent them from happening by showing in an emerging design who is changing which parts of the system down to method level. With this information, developers can proactively avoid concurrent modification of the same source code entities. Thus, these tools do not explicitly detects emerging conflicts.

2.2 Evaluation of Coordination Strategies when Merging

Though there has been a significant effort to proactively detect conflicts and notify developers, little work has been done towards understanding how developers behave when facing merge and conflict resolution. What are developers' needs in these situations? Few field studies [13, 9] observed this phenomenon and report on it at a general level. Furthermore, does developers' behavior change when information about emerging conflicts is available? How does the behavior change? Little is known on whether the extra information is beneficial to developers, given the lack of evaluation of the solutions proposed.

Grinter conducted the first field study that investigated developers' coordination strategy [13]. The study is composed of field observations and semi-structured interviews conducted on two development teams of distinct software companies. The aim of the study was specifically to understand how developers use SCM tools to coordinate their work. Grinter observed that sometimes it becomes difficult for a developer to merge without communicating with the other person who worked on a module. When this happens, developers tend to discuss what they did, and work together to solve the conflicts and successfully merge the code. Another behavior observed is that developers are constantly faced with a dilemma: on the one hand they want to rush and finish their work first to avoid merging; on the other hand, they want to produce quality code. One might think that experienced developers have no problems handling merge. However, Grinter observed that even experienced developers face problems merging code.

The second study was conducted through an eight weeks observation and note taking of the coordination practices of a software team at NASA [9]. The team followed a formal software development process that included, among others, communication through emails after files are checked in and code reviews. Similar to the finding of Grinter, de Souza observed that developers tend to speed up to finish their activities earlier to avoid merging. Another behavior observed is related to files that are frequently changed by many developers. They tend to perform partial check ins, which consist of checking in some of the files even when the developers have not finished all their changes. Since it takes long to compile the system under development, it was observed that some developers tend to hold their check ins until the end of a work day. It is also common practice to send an email (containing a brief description of the impact of their changes on other's work) to the team before performing the check in. Thus, in some cases developers tend to think individually trying to avoid merging, while in other instances they think collectively by holding check ins and explaining their changes to the other team members.

The work of Sarma et al. [25] focuses on preemptive conflict detection and the variation of developers' ability to detect and resolve conflicts. They conducted a 90-minutes laboratory study with 40 participants to investigate: (i) whether workspace awareness helps users in their ability to identify a larger number of conflicts; (ii) whether workspace awareness affects the completion time for tasks with conflicts; and (iii) whether workspace awareness promotes coordination. Their results show that participants who used Palantìr detected and resolved a larger number of conflicts than those with no conflict detection tool. When it comes to completion time, the participants using Planatìr took less time to resolve direct conflicts, but more time resolving indirect conflicts. In general, participants using Palantìr coordinated more than those not using it, the main coordination actions being SCM operations and chat. This initial evaluation shows promising results, however we believe that it is important to investigate whether these improvements bring fundamental changes in developers' behavior, and whether the trade-off between the benefits and the cost in added coordination pays off.

Another related study is the one conducted by Biehl et al. [2] that evaluates the impact of change awareness and conflict detection. The tool under evaluation, FASTDash, not only informs of potential conflicting situations, but also which files are being viewed, and what methods and classes are being changed. They conducted an observational study of a development team before and after the introduction of the tool. There were 6 participants involved, who were observed for four afternoons. The most important results of this study were the increase in communication, which the authors attribute to the raise of awareness, and the reduction in overlapping work.

The last related study was conducted by Dewan and Hedge [10] to evaluate the usefulness of CollabVS's mechanism to detect and fix conflicts at editing time. The tool showed to be useful to increase the developer's ability to detect and resolve conflicts with an increase in communication for resolving indirect conflicts.

Our user study reported in this document complements the empirical studies performed so far in several aspects. First, it studies the behavior of developers when performing SCM operations (e.g., check in, check out, merge) in detail, analyzing the different strategies and relating them with the developers' experience. Second, it investigates how developers change their behavior when exposed to preemptive conflict detection, and whether this change is beneficial for them or not. Lastly, it collected developers' opinion of how to present the information of emerging conflicts without disturbing their main focus: to produce quality code.

3 Preemptive Conflict Detection

To study the impact of preemptive conflict detection on developers' behavior when merging code, we have implemented an application to detect conflicts in real time and to notify developers about them. The goal of this application is to support developers to detect emerging conflicts at earlier stage than during check-in, and consequently, coordinate their activities to avoid complex merging.

We present the conflict detection algorithm we devised and *Conflicts*, the plug-in that enriches the Eclipse IDE to show information on emerging conflicts.

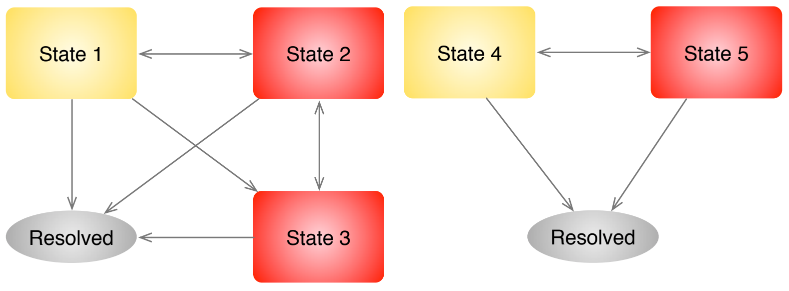
3.1 Conflict Detection Algorithm

Our conflict detection algorithm resides on the server of Syde [17], our tool to help developers to collaborate. Syde is a client-server application that tracks the fine-grained changes that developers perform to a shared software project in the Eclipse IDE and records them on the server for multiple purposes: raising the awareness of the team activity, detecting conflicts, and building recommendation systems that use the recorded data. To do so, a system to be tracked is modeled as an abstract syntax tree (AST), and the changes are modeled as insert, delete and change operations to the AST. On the server, Syde keeps the AST representation of the state of the system in each developer's workspace.

Our algorithm detects structural conflicts [20] related to the change operations Syde tracks. It can detect both direct [22, 24] and indirect conflicts [24, 26]. Direct conflicts refer to changes concurrently made to the same program artifacts, while indirect conflicts refer to changes made on an artifact that can impact an interdependent artifact. Indirect conflicts can be further classified into syntactic or semantic. An example of a syntactic conflict is the change of a method's signature that another method calls, causing a compilation error. A semantic conflict involves changing the behavior of a method, potentially introducing runtime errors or unexpected behaviors on its callers. So far, we have addressed direct and indirect conflicts due to syntactic changes.

The conflict detector is triggered every time a new change operation arrives at the server. It receives as input the operation, the AST of the author of this operation, and it compares them against the ASTs of the other developers currently working on the project; one at a time. If there is a potential conflict, it always refers to two developers and their conflicting entities. In our model, a conflict never involves more than two developers. Although we could have modeled it for multiple developers, we decided to keep it simple to avoid the overhead in complexity of the algorithm, and in potential coordination of developers. Our conflict detection algorithm uses, apart from the change and the AST models, the following information tracked on the Eclipse IDE: the current SCM revision of the file where the entity resides, and whether the entity was modified or not since the last check-in. This extra information helps the algorithm to order the change events.

The created conflicts are classified into two categories: yellow – when there is a structural conflict between two entities, but none was checked in the SCM system; red – when there are structural differences between two entities, and one of them has been checked in the SCM system. Once created, a conflict follows a workflow (See Figure 1) until its resolution. There are three states a direct conflict can be, and two states an indirect conflict can be. Each state is described below. In summary, a conflict can change to another state, stay in the same state or be resolved.



(a) Direct conflicts (b) Indirect conflicts

State 1	Conflicting changes on two versions of same entity, both are up-to-date with SCM.
State 2	Conflicting changes on two versions of same entity, one is outdated with SCM.
State 3	Conflicting changes on two versions of same entity, both are outdated with SCM.
State 4	A developer changed/deleted a method's signature, another developer is changing its caller.
State 5	A developer changed/deleted and committed a method's signature, another developer is changing its caller.

Figure 1: Conflicts workflow

3.2 Conflicts Plug-in

Once potential merge conflicts are detected by the algorithm on the server, this information needs to be shown to the involved developers in their IDE. We have implemented in the Conflicts Plug-in three different ways to show emerge conflicts:

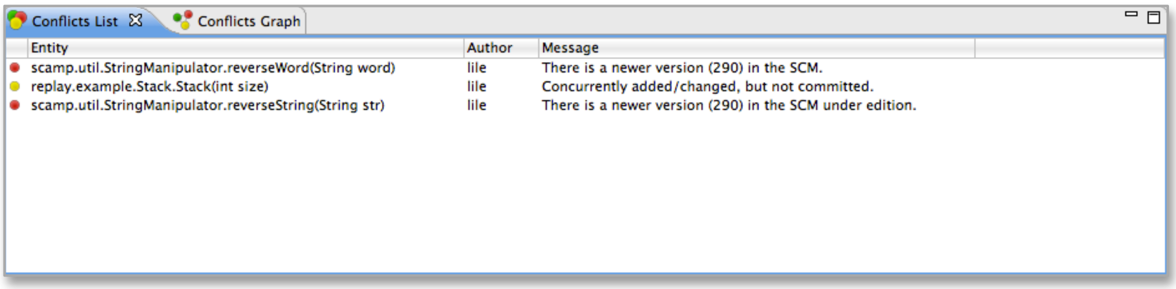
- **List View** shows all potential conflicts as a list.
- **Graph View** shows the classes in a graph and potential conflicts as a color layer on top of each node.

- **Annotations on the Java Editor** show potential conflicts as a marker on the left ruler on the Java editor.

Although in this study we evaluate only the views, in the following, for completeness, we present and explain the three of them.

3.2.1 List View

It shows all the potential conflicts that a developer might be involved in (See Figure 2). It shows in which method or class the conflict is, who the other developer involved is, a description of the conflict, and its status (yellow/red). If a developer clicks on one of the conflicts in the list, the file in which the conflict is located opens with the involved class or method in focus. This view is similar to the ones presented in previous tools [18, 25], and thus can be considered an adaptation of them.



	Entity	Author	Message
🔴	scamp.util.StringManipulator.reverseWord(String word)	lile	There is a newer version (290) in the SCM.
🟡	replay.example.Stack.Stack(int size)	lile	Concurrently added/changed, but not committed.
🔴	scamp.util.StringManipulator.reverseString(String str)	lile	There is a newer version (290) in the SCM under edition.

Figure 2: The Conflicts List View showing three potential conflicts: two red, and one yellow

3.2.2 Graph View

It shows a graph representation of the system, with the packages and classes as nodes and containment or call dependency as edges (See Figure 3). This view notifies developers about emerging conflicts by adding a color layer on the class node. If a developers wants to get more information on the conflict, he can hover over the node to see in which method the conflict is located, who the other involved developer is, and read the description of the conflict.

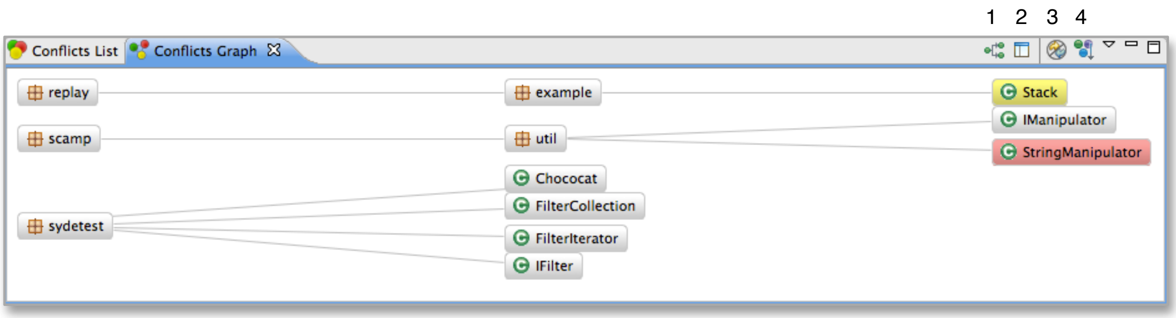


Figure 3: The Conflicts Graph View showing potential conflicts in classes Stack and StringManipulator

The Graph view provides several graph customization options that can be accessed through the controls on the toolbar or by right-clicking on a node. The options found with right-clicking on a node are to collapse/expand it –in case the node is a package– and to hide it. The controls allow developers to:

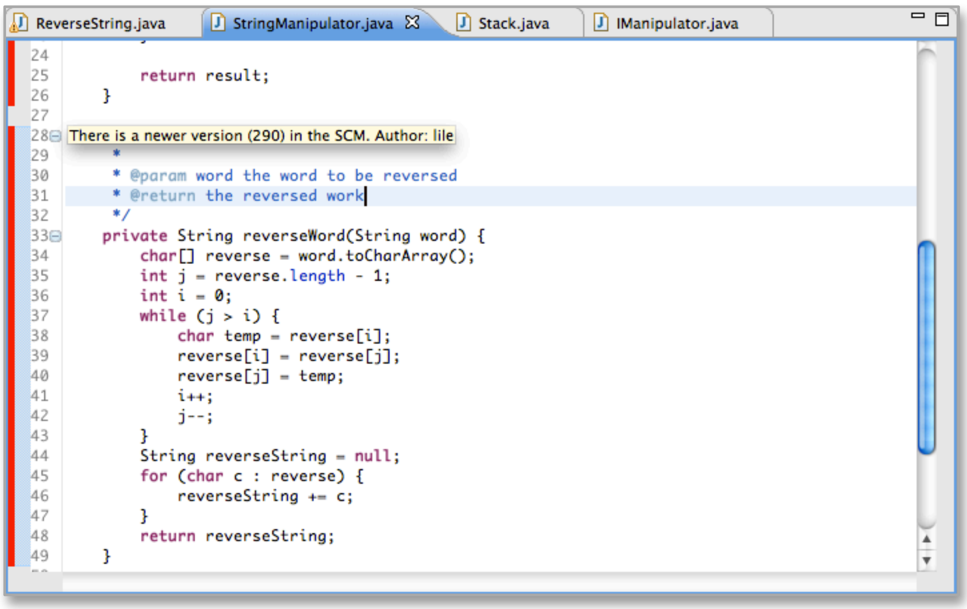
1. change the relationship shown by the edges from containment to call dependency;
2. change the graph layout to one of: horizontal or vertical tree, grid or radial;
3. start/stop updating the graph according to the changes being done (e.g., if a class is added, a node representing it will also be added if this option is selected);

4. enable/disable emerging design, which means the graph only shows the classes that a developer opens while he is working.

The concept of emerging design was inspired by previous tools [7, 23, 26] that are specialized in showing to developers the emerging design.

3.2.3 Annotation on the Java Editor

If a developer does not want to keep the views open, he can still get hints of potential conflicts, but only on the file that he is currently working on (See Figure 4).



```
        return result;
    }

There is a newer version (290) in the SCM. Author: lile
     *
     * @param word the word to be reversed
     * @return the reversed work
     */
    private String reverseWord(String word) {
        char[] reverse = word.toCharArray();
        int j = reverse.length - 1;
        int i = 0;
        while (j > i) {
            char temp = reverse[i];
            reverse[i] = reverse[j];
            reverse[j] = temp;
            i++;
            j--;
        }
        String reverseString = null;
        for (char c : reverse) {
            reverseString += c;
        }
        return reverseString;
    }
```

Figure 4: The Annotation on the Java Editor showing a potential conflict on the method reverseWord of class StringManipulator

The left ruler of the Java editor showing the conflicting entity receives a yellow or red color and the developer can hover over the annotation to read the conflict's description and the name of the other developer involved.

4 User Study Design

We want to qualitatively investigate how developers behave when dealing with merging code, and how their behavior changes when they are exposed to preemptive conflict detection. As a secondary goal, we want to investigate how developers prefer the information on emerging conflicts to be delivered to them.

We formulate the following research questions:

RQ1 How do developers behave when they have to merge code and resolve conflicts?

RQ2 How does this behavior change when information of emerging conflicts is present?

RQ3 How do developers perceive different approaches to deliver information on emerging conflicts?

To observe developers' behavior, ideally we would conduct a field study with a team of developers that adopt the preemptive conflict detection tool to observe how their behavior change while they develop software in practice. However, it is difficult to convince practitioners to change their programming environment without having previous evidence of the usefulness of the tool that is being presented to them.

Therefore, we have designed a user study in a laboratory setting to simulate developers working in parallel to closely observe their behavior. The study is composed of a programming assignment with three tasks that are performed by a pair of developers simulating a situation of parallel development. Each assignment is designed to cause a merge conflict. To simulate a distributed environment, developers are not allowed to talk nor to see the screen of each other. All the communication has to be done over instant messaging (IM). In addition, developers are given the Eclipse IDE preconfigured with the necessary tools (Subversive, connectors to SVN, and Syde's plug-ins) to perform the assignment, and an IM client for communication.

4.1 Data Collection

We follow the guidelines of Creswell [6] and Barbour [1] to use mixed methods of data collection for a better interpretation of our findings. The data sources we have collected are:

- **questionnaires** regarding the participants' experience level, and regarding the assignment;
- **observation** through video recording;
- **interviews** to gather participants' feedback on their experience with using preemptive conflict detection;
- **documentation** in the form of the IM logs, and the list of changes and conflicts generated during the implementation assignment.

In the following we explain how each data was collected and what procedure we used to analyze it.

4.1.1 Questionnaires

Participants were asked to answer two questionnaires for different purposes:

1. **Screening questionnaire.** Before the assignment session, participants were asked to answer a screening questionnaire (See Appendix A) that collected personal (name, email address) and technical experience (level and number of years of experience with the tools and concepts used in the assignment) information. Personal information was used for contact purposes, while the information on the participants' experience level was used to characterize the participants and take their knowledge into consideration on the data analysis.
2. **Debriefing questionnaire.** After the assignment, participants were asked to answer a short questionnaire (See Appendix B) to give immediate feedback about the assignment. First, they were asked about their experience in performing the experiment, then they rated some statements related to the usability of the views that showed emerging conflicts. In the second half of the questionnaire, participants were given a few statements regarding each task to be rated according to their opinions.

4.1.2 Observation

The observation was not performed directly during the assignment, but we recorded each participant's interaction with the tool and analyzed the screencast (a digital recording of computer screen output) at a later moment. To properly analyze the screencast, we developed a codebook that we used to annotate the videos. The codes created (see Table 1) can be classified into three categories: communication, interaction with the SCM system, and interaction with the Syde's views of preemptive conflict detection.

To code the screencasts, we used the tool called VCode [16], which provides a timeline, two different types of annotations (range and mark), and keyboard shortcuts for placing the annotations, playing and pausing the video, and adding descriptions to the annotations. After the screencast is annotated, VCode exports all the annotations to a .csv file, which we used for qualitative and quantitative analyses of the generated data.

4.1.3 Interview

At the end of each session, we conducted a semi-structured interview with the two participants at the same time to collect more feedback from them. Table 2 shows the questions that served as a guide to the interviews.

However, since the goal of the interview was to collect the participants' opinion regarding preemptive conflict detection, the different ways to visualize this information, and the user study itself, each interview

Table 1: Codebook used to annotate the screencasts of the participants' coding sessions

Category	Code	Description	Type
Communication	Communicating	When the chat window is in focus	range
	Message notification	When notification of a chat message appears, but the chat window is not in focus	mark
Interaction with SCM system through the IDE	Synchronizing	Synchronizing the project or a part of it with the code-base in the repository	range
	Updating	Updating the project or a part of it with the latest version from SVN	range
	Checking in	Checkin in the changes made to the project	range
	Merging	Merging the code with the latest version from SVN	range
	Resolving conflict	Resolving conflict during merging	range
	Viewing changes	Viewing changes in the Compare editor	range
Interaction with Syde	Conflict graph	When the conflict graph is visible	range
	Conflict list	When the conflict list is visible	range
	Interaction with graph	When the user interacts with the Conflict graph	mark
	Interaction with list	When the user interacts with the list	mark
	Yellow conflict	When a yellow conflict regarding the task appears	mark
	Red conflict	When a red conflict regarding the task appears	mark
Task delimiter	Task starts	User starts a task	mark
	Task ends	User finishes a task	mark

Table 2: The list of questions that guided the semi-structured interview at the end of each session

	Questions
1	Did you have to resolve conflicts during the experiment? In which situations?
2	Do you think being aware of emerging conflicts at implementation time helped you to prevent conflicts or to reduce their complexity at check in?
3	Did the information about emerging conflicts help you to be aware of what your colleague was doing?
4	Were emerging conflicts and incentive for you to talk with your colleague and to coordinate your tasks?
5	Did you get to know what were the tasks of the other participant?
6	Was the Conflict list useful? Was it intuitive? What are the advantages and disadvantages of this view?
7	Was the Conflict graph useful? Was it intuitive? What are the advantages and disadvantages of this view?
8	Which one of the two visualizations do you prefer? Why?
9	Can you think about other ways that this information could be visualized?
10	Can you think about situations in your everyday coding in which these visualizations could be helpful?
11	Do you have any other suggestions or comments about the tool?
12	Do you have comments or suggestions about the experiment?

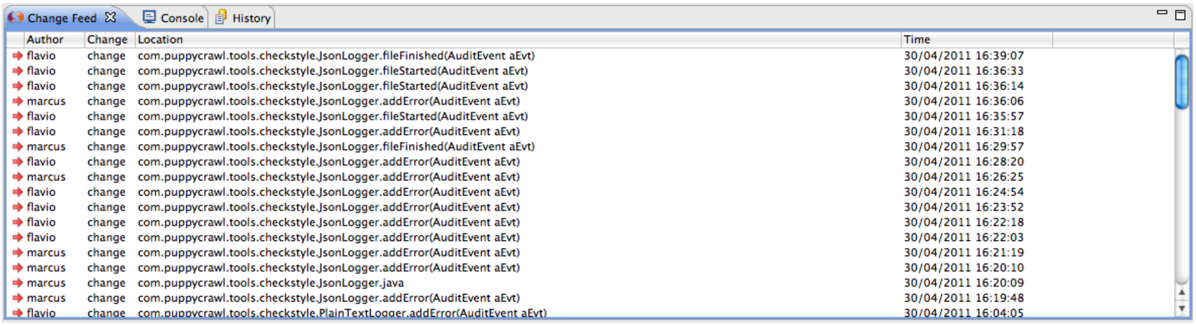
had different set of questions that were asked according to the answers participants gave to questions previously asked.

In most cases, there was a short break between the assignments and the interviews. Hence, the participants had some time to think about their experience with the tool, what the advantages and disadvantages of having information in preemptive conflict detection were, and how this information could be better presented. In a few occasions, when participants could not meet with the experimenter at a later time, the interview was conducted right after the end of the assignment.

The interviews were recorded and later on transcribed to be used on the data analysis. We decided to not develop a codebook for the interviews, because they were usually short (around 10 minutes). Hence, we mainly used statements of the participants to illustrate their opinion, and support the findings that were emerging from the data analysis.

4.1.4 Documents

A few documents were also collected at each session, however they were only used in the data analysis when some of the other data collected was missing. We collected: the chat logs of each participants, and the list of changes performed by them (collected through Syde). The list of changes indicates who changed where in the system and in which order (See Figure 5).



Author	Change	Location	Time
flavio	change	com.puppycrawl.tools.checkstyle.JsonLogger.fileFinished(AuditEvent aEvt)	30/04/2011 16:39:07
flavio	change	com.puppycrawl.tools.checkstyle.JsonLogger.fileStarted(AuditEvent aEvt)	30/04/2011 16:36:33
flavio	change	com.puppycrawl.tools.checkstyle.JsonLogger.fileStarted(AuditEvent aEvt)	30/04/2011 16:36:14
marcus	change	com.puppycrawl.tools.checkstyle.JsonLogger.addError(AuditEvent aEvt)	30/04/2011 16:36:06
flavio	change	com.puppycrawl.tools.checkstyle.JsonLogger.fileStarted(AuditEvent aEvt)	30/04/2011 16:35:57
flavio	change	com.puppycrawl.tools.checkstyle.JsonLogger.addError(AuditEvent aEvt)	30/04/2011 16:31:18
marcus	change	com.puppycrawl.tools.checkstyle.JsonLogger.fileFinished(AuditEvent aEvt)	30/04/2011 16:29:57
flavio	change	com.puppycrawl.tools.checkstyle.JsonLogger.addError(AuditEvent aEvt)	30/04/2011 16:28:20
marcus	change	com.puppycrawl.tools.checkstyle.JsonLogger.addError(AuditEvent aEvt)	30/04/2011 16:26:25
flavio	change	com.puppycrawl.tools.checkstyle.JsonLogger.addError(AuditEvent aEvt)	30/04/2011 16:24:54
flavio	change	com.puppycrawl.tools.checkstyle.JsonLogger.addError(AuditEvent aEvt)	30/04/2011 16:23:52
flavio	change	com.puppycrawl.tools.checkstyle.JsonLogger.addError(AuditEvent aEvt)	30/04/2011 16:22:18
flavio	change	com.puppycrawl.tools.checkstyle.JsonLogger.addError(AuditEvent aEvt)	30/04/2011 16:22:03
marcus	change	com.puppycrawl.tools.checkstyle.JsonLogger.addError(AuditEvent aEvt)	30/04/2011 16:21:19
marcus	change	com.puppycrawl.tools.checkstyle.JsonLogger.addError(AuditEvent aEvt)	30/04/2011 16:20:10
marcus	change	com.puppycrawl.tools.checkstyle.JsonLogger.java	30/04/2011 16:20:09
marcus	change	com.puppycrawl.tools.checkstyle.JsonLogger.addError(AuditEvent aEvt)	30/04/2011 16:19:48
flavio	change	com.puppycrawl.tools.checkstyle.PlainTextLogger.addError(AuditEvent aEvt)	30/04/2011 16:04:05

Figure 5: Example of a list of changes

It was more frequently used by the experimenter to monitor the assignment and take action whenever unexpected problems happened. The chat logs were only used during the data analysis whenever there was a problem with the screencast of a participant, which happened three times, or the chat window was located outside the recorded area, which happened once.

4.2 Object System

The system we chose as object of our experiment is *Checkstyle*.[1] We used version 5.3, which consists of 341 classes distributed across 22 packages, for a total of 46 KLOCs.[2] Our choice was motivated by the following factors: Checkstyle's size allows for performing a user study session, yet being representative of real life programs. It is written in Java, with which many potential participants are sufficiently familiar. It has been used in previous experiments [5, 15, 27, 28], from which one was conducted by the authors of this work.

4.3 Tasks

The assignment was composed of three coding tasks that had to be done by two participants collaborating with each other. Each participant had a different set of tasks, but they were complementary. For a task to be considered finished, each participant had to implement what his task was asking him to do, coordinate with his pair, check in his changes, update the code with the changes done by his pair, and make sure all the tests related to the task pass.

In the following we describe each task, showing what each participant had to code (refer to Appendix B to see the complete instructions for the set of tasks of one participant).

Task 1 – Improve class `MethodCountCheck`. In this task, participants are asked to make different modifications in method `checkCounters`, which basically counts the number of (private, public, package, and default) methods in a class. One of the participants is asked to refactor the method `checkCounters` by creating a utility method in order to remove the repetition on the code of `checkCounters`. The other participant is asked to implement a few checks that are missing in `checkCounters`.

Task 2 – Finishing class `PlainTextLogger`. In this task, participants were asked to finish the implementation of class `PlainTextLogger`. For one participant, it meant complementing the implementation of method `addError`, and implementing method `fileStarted`. For the second participant, it meant complementing the implementation of method `addError` with different functionality, and implementing method `fileFinished`.

[1]See `http://checkstyle.sourceforge.net/`
[2]Measured using `http://eclipse-metrics.sourceforge.net/`

Task 3 – Finishing class `JsonLogger`. In this task, participants were asked to finish the implementation of class `JsonLogger`. For one participant, it meant complementing the implementation of method `addError`, and implementing method `fileFinished`. For the second participant, it meant complementing the implementation of method `addError` with different functionality, and implementing method `fileStarted`. Though the description is similar to the one of Task 2, what was asked them to change in method `addError` was different in all four cases.

Each task was accompanied by instructions to prepare for it, which contained the tools and views that participants were or were not allowed to use in each task (See Appendix B).

4.4 Pilot Studies

To plan a user study in a laboratory setting in which two participants have to collaborate through IM and merge conflicts have to appear is a complex task. Thus, in our first pilot study, we initially tried to simulate the second person of a pair of participants. This showed to be unfeasible, because there were several limitations with the simulation of the second participant. For instance, it was difficult to simulate the decisions an average developer would take, and to make sure the participant does not notice the second person is actually a simulation.

We then, redesigned the assignment by creating a set of complementary tasks for two participants, and ran a second pilot study. Though the complexity of the user study discouraged us to try it out at first, the second pilot study showed that it was possible to run it. The merge conflicts appeared when we planned to, and the participants, who in some cases were located in different continents, were able to communicate over chat and to solve the tasks collaboratively.

4.5 Operation

The user study is composed of runs, where each run involves two participants changing the object system in parallel to solve the tasks. The set of tasks each of the participants of a run receives is different but closely related to each other.

Each run includes a training session of approximately 15 minutes and one experimental session. The training session consists in a tutorial on the views, a hands-on session with a toy system, in which the participants can intentionally cause conflicts and experiment with the views, and a warm-up task to allow the participants to get used to the object system. The experimental session is composed of three programming tasks with unlimited time to solve them. The first task is done without preemptive conflict detection, while the second and third ones are done using either the list or the graph view.

There were 6 experimental runs, with slightly different settings among them.

In runs 1-3 and 6, the participants used laptops with 2 GB and 4 GB of RAM, running Mac OS X, with Eclipse and Skype installed. In runs 4 and 5, the participants used their personal computers to run a VirtualBox image with Ubuntu 10.04, and 1.5 GB of RAM, with Eclipse installed, and Gmail chat for communication. The VirtualBox image might have constituted a disadvantage to the participants of runs 4 and 5, since it is slower than the laptops running Mac OS X.

In runs 1-3, the participants had distinct set of tests to fix. However, we observed that this caused an overhead in communication while they tried to understand they had different tasks and tests. For runs 4-6, the two set of tests were put together, so a participant could see there were failures that the other participant was responsible to fix, inferring immediately that they have different tasks.

In runs 1-3 the participants used the list view for Task 2 and the graph view for Task 3. In runs 4-6 the participants used the graph view for Task 2 and the list view for Task 3. The swap of the views avoids serial position effects.

Runs 1-3 and 6 took place in a reserved room at the University of Lugano, while runs 4 and 5 took place in a laboratory (shared with other people) at the University of British Columbia.

5 Data Analysis

We analyze each run individually to, in Section 6, address the research questions. For each run (R), we first describe the participants' background based on the screening questionnaire. Then, we describe the unexpected events and problems that happened in the run. Then, we summarize the observations for each participant

(P) to, finally, correlate it to the data collected from the debriefing questionnaire and the interview. The complete answers to the questionnaires can be found in Appendix C. For neutrality reasons, all participants will be treated with the masculine pronoun, which does not imply that all participants are males.

5.1 Run 1

The participants (P1, P2) of this run are Master students in Computer Science with at least 6 years of experience in Java development and use of Eclipse, and consider themselves advanced users in the subject. P1 has 4 years of experience with SCM systems, and 3 years of experience working in teams, while P2 has 2 years of experience with SCM, and considers himself knowledgeable in working with teams.

P1 indicated that he usually works in teams of 3-5 people, and uses Git and SVN. He checks in the code more than once per day, but does not have to resolve conflicts frequently, because the team he works with has different roles and tasks. P2 does not work in teams, and seldom uses systems SCM.

Problems. A couple of problems happened in R1 that directly influence the behavior of the participants throughout the user study and their opinion about emerging conflict, and the study itself.

First, the participants did not follow the instructions given both by the experimenter and the handout regarding the use of the views of emerging conflicts. Both participants used the list and the graph to visualize emerging conflicts for Task 1, while P2 used the list for part of Task 3.

In addition, P1 did not understand how to use the graph, thus was unable to see anything with it. This is evidence that the participants need more time to get comfortable with using the views, even though the preparation instructions for Task 3 describe the steps to open and load the graph view.

5.1.1 Observations from Videos

Table 3 shows the frequencies and the descriptive statistics for the observations of participant P1, taken from a 60-minutes screencast of P1's computer. There is a total of 93 events, from which the majority are communication events (22). The annotations of the events of P1's programming session are incomplete for two main reasons: (i) the video recording application crashed when P1 was at the end of Task 1, hence most of the events for this task were lost; (ii) we restarted the video recording before the beginning of Task 2; (iii) to avoid further crashes, we reduced the captured window and made it follow the cursor, which cut some of the notifications (especially message notifications) that were located outside the window .

Table 3: Frequencies and descriptive statistics for observations of participant P1. The total duration of the events is given in seconds. The descriptive statistics refer to the duration (in seconds) of the events that have a range

Events	T1	Dur.	T2	Dur.	T3	Dur.	Total	Dur.	Median	Mean	Stdev.
Communicating	1	19	14	285	7	278	22	582	16	26	35.6
Conflict graph	0	0	0	0	4	567	4	567	114	142	88.8
Conflict list	1	592	5	361	5	934	11	1887	49	172	247.9
Checking in	0	0	2	31	2	33	4	64	16	16	1.3
Merging	0	0	2	197	0	0	2	197	98	98	49.2
Resolving conflict	0	0	0	0	0	0	0	0	0	0	-
Synchronizing	0	0	6	25	3	14	9	39	4	4	1.8
Updating	0	0	3	13	0	0	3	13	4	4	1.3
Viewing changes	0	0	4	371	0	0	4	371	94	93	23.5
Running tests	0	0	6	152	3	78	9	230	22	26	7.7
Message notification	1		4		1		6				
Interaction with graph	0		0		2		2				
Interaction with list	0		4		3		7				
Yellow conflict	0		1		0		1				
Red conflict	0		2		0		2				
Task starts	1		1		1		3				
Task ends	1		1		1		3				
Total	5		51		31		93				

Table 4 shows the frequencies and the descriptive statistics for the observations of participant P2, taken from a 140-minutes screencast of P2's computer. The list of events is complete and covers the entire programming session. There is a total of 178 events, from which the 4 most frequent ones are communication (29), message notification (22), interaction with the list (22), and interaction with the graph (21).

Table 4: Frequencies and descriptive statistics for observations of participant P2. The total duration of the events is given in seconds. The descriptive statistics refer to the duration (in seconds) of the events that have a range

Events	T1	Dur.	T2	Dur.	T3	Dur.	Total	Dur.	Median	Mean	Stdev.
Communicating	12	790	12	320	5	70	29	1180	15	41	70.8
Conflict graph	5	685	0	0	1	964	6	1649	156	275	367.6
Conflict list	6	699	9	1277	4	308	19	2284	41	120	140.5
Checking in	1	42	2	101	1	11	4	154	43	38	19.4
Merging	1	2	1	6	3	20	5	28	6	6	3.7
Resolving conflict	0	0	0	0	1	70	1	70	70	70	-
Synchronizing	4	30	9	30	4	12	17	72	3	4	3.5
Updating	1	2	1	2	1	1	3	5	2	2	0.5
Viewing changes	1	41	1	74	3	289	5	404	74	81	56.9
Running tests	2	41	8	149	3	62	13	252	20	19	4.4
Message notification	13		7		2		22				
Interaction with graph	14		0		7		21				
Interaction with list	18		4		0		22				
Yellow conflict	1		1		0		2				
Red conflict	0		3		0		3				
Task starts	1		1		1		3				
Task ends	1		1		1		3				
Total	79		52		34		178				

Task 1. Since the screencast of P1 for Task 1 was almost fully lost, the summarization of the events for this task is mostly based on the events from P2. Even though the participants were not supposed to use the emerging conflicts views, P2 keeps both views open, with the list in focus. At the beginning of the task, P1 tells P2 that he modified `MethodCountCheck`, and asks whether he has got any issues, to which P2 responds negatively. After P2 also modifies `MethodCountCheck`, a yellow warning appears notifying that they are both modifying method `checkCounters`. P1 asks P2 whether he sees the notification, to which he answers positively, but the only communication that happens regarding this notification is P1 telling P2 that he is "waiting for something to happen".

P2 finishes his task and checks in the changes. When P1 tries to update the code, he gets conflicts and has to resolve them. At this moment, they discuss and find out what their tasks are. After P1 merges, checks in the code, and P2 updates it, the yellow warning disappears. P2 tells P1 everything is working, and they finish Task 1.

An interesting observation is that P2 interacts quite a lot with the two views, and it becomes clear that he is trying to get used to them while performing the programing assignment. This is evidence that they did not spend enough time to feel comfortable with the views before starting the tasks. From the events of P1 (See Table 3), it can be observed that he uses the list view while solving Task 1, which was not allowed.

Table 3 and Table 4 show that events related to the use of the conflict list appear in all three tasks for P1 and P2. This is further evidence that the participants did not follow the instructions related to the use of the views for the tasks.

Task 2. In this task, both the participants start working on method `addError` of class `PlainTextLogger`. When P2 is almost done, a yellow warning appears on the conflict list, to which P1 reacts by asking P2 whether he was changing `addError`. P2 has already checked in his changes, and lets P1 know. P1 then merges the changes from P2 and checks in his changes to `addError`. A red warning appears for method `fileFinished`, which leads to a discussion of whether P1 is changing it or not. In the middle of the task, P1 and the experimenter realize he has the wrong set of tests, which causes some disruption. Another red warning appears for method `fileStarted`, but for both red warnings there is no further need of merging at check in time.

Table 3 and Table 4 show that both participants interact a fair amount of time with the list view, communicating and viewing changes. The screencasts show that the information of emerging conflicts generate a good amount of communication. However, the participants seldom try to understand what was the other's task and how their tasks relate. This type of communication is only triggered after a first round of check in and check out.

Task 3. Unexpected events happened in this task that prevented an effective use of the graph view. First, P2 did not understand how to load the graph, and did not ask for help. As consequence, the graph is empty during the entire task. Second, from the middle of the task on, P1 switches to the list and uses that instead of the graph. Because of these issues, P1 and P2 do not communicate in the first half of the task and do duplicated work on method `addError` of class `JsonLogger`. When P2 has to merge the changes previously checked in by P1, he struggles to resolve the conflicts and ends up by doing it manually on the Java editor.

The number of communication events in Task 3 is the lowest for the 3 tasks, ignoring the incomplete data of Task 1 from P1. We attribute the lower communication to the absence of notification of emerging conflicts, which prevents the participants from being aware that they are implementing the same method.

5.1.2 Discussion

The answers of participants P1 and P2 to the questionnaire indicate that P1 had to resolve conflicts for Task 1 and Task 2, while for Task 3 it was P2's time to resolve conflicts. They indicate that communicating with the other participant was somewhat helpful (agree/neither disagree nor agree); however different opinions appear when it comes to the usefulness of knowing about emerging conflicts, ranging from disagree to agree.

On the interview, the participants commented that they were mostly communicating during check in time:

"I think it was when we had to commit." (P1)
"So mostly we were communicating if there was trouble in merging." (P2)

In addition, P2 came into a conclusion that he was rushing to check in his changes to avoid dealing with merging:

"... after the first commit I think that, for the first two tasks I was always focusing on committing first so I don't have to resolve conflicts. I reflected on it..." (P2)

Another issue that became evident during the interview is that the participants struggled to understand the concept of emerging conflicts and were partially unable to benefit from it. They strongly believed they misused the tool by not being collaborative enough:

"You know what? When I saw that he was editing I was like 'we're gonna have problems'. So I think I just completely misused the thing because after seeing that done in a collaborative way you'd say 'what are you changing?'. Instead I was like 'we're gonna have problems'." (P1)

The participants mainly communicated when they had to merge code and resolve conflicts, even when information of emerging conflicts was present to help them. While the extra information might have not influenced the moment that they started talking, it seems to have influenced the amount of communication that underwent between them.

Regarding the different views, P1 affirms that he prefers the graph view with the color metaphor to inform about emerging conflicts, although he thinks there is a defect on the view that prevented him to see the information at times. P2 did not have a chance to use the graph, and expressed that the list was useful to see where in the code the other user was editing.

Finally, regarding other ways to visualize the emerging conflicts, P1 suggested to change the background color of the Java editor, or add markers on its ruler. P2 thinks that it is beneficial to show the information on the editor, because they do not need to keep a dedicated view open.

5.2 Run 2

The participants (P3, P4) of the run are PhD students in Computer Science, with 5 and 2 years of experience with Java development. They have 1-2 years of experience in developing industrial size systems and team development. P3 has 3 years of experience with using IDEs and SCM, and 2 years of experience with JUnit. P4 has 2 yeas of experience with using IDEs, and 1 year of experience with using SCM and JUnit. P3 is somewhat familiar with Checkstyle, while P4 has no familiarity with it.

The participants usually do not work in teams, and currently use language-dependent SCM systems for Smalltalk. They indicate that they check in code frequently (after new tests are created and run, and hourly), but they rarely have to resolve conflicts because of the granularity of the SCM (method level).

Problems. A couple of problems happened on R2 that might have influenced the behavior of the participants throughout the programming session, and their opinion about the emerging conflicts, and the study itself. First, they had problems with incompatible tests, which provoked an extra loop of updating, running the tests, and checking in the changes for Task 2. Then, for Task 3, they did not see conflicts in the graph, because participant P4 kept the code with compilation errors for the entire duration of the task, which prevented the tool to detect changes and, consequently, potential code conflicts.

5.2.1 Observations from Videos

Due to technical problems with the video recording application that were not detected during the experiment, there is no screencast for P3's session to be analyzed. Hence, the analysis of this run is based on the events of P4 and on the communication log.

Table 5 shows the frequencies and the descriptive statistics for events observed from P4, taken from a 81-minutes screencast of P4's computer. There is a total of 205 events, from which message notification events (42) and communication events (39) are the most frequent. The annotations of the events of Task 3 of P4 are incomplete, because this task was interrupted as soon as P3 checked in his changes. The goal of Task 3 was to provoke conflicts, so the participants would see them with the graph view. However, since this did not happen, the experimenter decided to interrupt the task before P4 finished it.

Table 5: Frequencies and descriptive statistics for observations of participant P4. The total duration of the events is given in seconds. The descriptive statistics refer to the duration (in seconds) of the events that have a range

Events	T1	Dur.	T2	Dur.	T3	Dur.	Total	Dur.	Median	Mean	Stdev.
Communicating	9	568	27	525	3	60	39	1153	17	30	38.4
Conflict graph	0	0	0	0	7	1106	7	1106	32	158	238.5
Conflict list	0	0	11	1342	1	1	12	1343	44	112	145.8
Checking in	1	8	3	61	0	0	4	69	8	17	20.9
Merging	0	0	0	0	0	0	0	0	0	0	-
Resolving conflict	0	0	0	0	0	0	0	0	0	0	-
Synchronizing	4	11	11	40	0	0	15	51	3	3	2.3
Updating	1	3	2	7	0	0	3	10	3	3	0.6
Viewing changes	2	466	5	234	0	0	7	700	13	100	164.8
Running tests	2	49	8	241	0	0	10	290	29	29	9.6
Message notification	4		28		10		42				
Interaction with graph	0		0		33		33				
Interaction with list	0		18		2		20				
Yellow conflict	0		3		0		3				
Red conflict	0		4		0		4				
Task starts	1		1		1		3				
Task ends	1		1		1		3				
Total	25		122		58		205				

Task 1. The participants start to implement what is asked in the handout, and P3 finishes and checks in his code first. When P4 finishes and synchronizes his code, he notices that P4 did not add method `checkMax`,

and starts to communicate with him. After sharing code over chat, they realize that they had to make different modifications, but think their modifications reflect the same code behavior. Thus, they discuss which one to keep, and the chosen one is P4's. P4, then, replaces the code from P3 with his, checks in the changes and tells P3 to update. When P3 updates the code, his tests break, and he realizes the functionalities of the code they implemented were different. They exchange messages to clarify that they have different tasks, P3 redoes the modifications, checks-in the code and tells P4 to update them. P4 updates it and tells P3 that the code works without rerunning the tests.

The communication between the participants is totally concentrated after the first check in of P3 and the first attempt to checkin of P4. Although they communicate to try to understand what the other is doing before P4 checks in, the fact that they do not realize their implementations had different functionalities provokes an extra cycle of check out, modification and check in, which is performed by P3 after P4's check in. Before the assignments started, the experimenter informed the participants that they had related tasks, to which they implied were the same. Spending the extra time understanding that the tasks are not the same is not the goal of this study. Hence, the experimenter must be clear about this in the next runs.

An interesting behavior observed is that the participants shared code snippets with their modifications over chat twice to coordinate between themselves.

Task 2. The participants start communicating at the beginning of the task, with P3 informing P4 he has to fix 2 tests, to which P4 says he also has to fix 2 tests. However, they do not verify whether they have to fix the same tests. They start implementing, until P3 says he sees some conflicts. P4 replies by saying he sees P3 is also working on `addError` method. They continue the conversation by sharing the code snippets of their modifications and realizing they are different. P3 asks P4 to check in first, so P3 can handle the merge. When P3 updates and runs the tests, he realizes the changes introduced by P4 break his tests. P3, then fix the code again and checks in the changes. When P4 updates the code, his tests break. At this moment, the experimenter is called and realize the participants have incompatible tests, asking them to move to the next task.

A couple of interesting behaviors are observed in this task. The first one being the attempt of the participants to coordinate their activities before starting to implement, probably already expecting conflicts to appear, but trying to avoid them. Second, as soon as they see a conflict emerging, they communicated and the more experienced developer offers to handle the merge. Third, in the middle of the task, P4 inadvertently overrides the changes introduced by P3 instead of merging them, consequently breaking a test that was passing before.

Task 3. The first part of this task is to implement a rather long method. While working on it, P4 keeps the code with compilation errors, which prevents Syde from capturing his changes and checking for conflicts. In addition, P3 is familiar with what is asked in the task, and quickly finishes it. Thus, no emerging conflicts were generated, preventing the participants from properly using the graph view.

5.2.2 Discussion

The answers of P3 and P4 to the questionnaire indicate that both had to merge code and resolve conflicts in Task 1, while P3 had to merge and resolve conflicts in Task 2. They indicate the communicating with the other participant was helpful (strongly agree) to perform the tasks, and on Task 2 as soon as they saw a conflict, they communicated with one another (strongly agree/agree). When asked whether knowing about emerging conflicts helped them to better coordinate, P3 was neutral (neither agree nor disagree), and P4 as positive (strongly agree). The last one is due to the fact that the participants agreed for P4 to check in first, and P3 be responsible for merging on Task 2.

When asked in the interviews whether emerging conflicts helped them to coordinate, they answered positively:

"... as soon as I noticed that P4 was changing something, then I asked him 'man, what are you changing?'." (P3)

" We coordinated before committing." (P4)

When asked whether knowing of existing conflicts before checking in was helpful to reduce the complexity of the merge, they were unsure:

"I'm not sure, because at the end... So probably the merge tool would also be helpful. So, it's more like a psychological thing. I would expect some conflicts to be there, so I will merge them. But I'm not sure it reduces the complexity." (P3)

"Maybe it reduces the overhead of the final merging, because you merge by kind of merging in place, right? Via Skype or via the tool. Maybe that complexity, but regarding the code complexity I think that no." (P4)

Indeed knowing about the conflict in advance on Task 3 made them deliver the changes into 2 smaller check ins instead of a single one comprising the changes in 2 methods.

Participant P3 seemed to be convinced of the usefulness of the tool:

"While I think that the usefulness of this tool is that you are working on something you have to do and then you see that someone else is also working on that. So in this case it was useful to coordinate maybe before doing the commit, but the real strength of this tool is that maybe P4 is in another city, we are working, and then I see P4 working in the same method, and then we can start interacting." (P3)

In addition, when asked whether they could think about situations in their everyday coding in which emerging conflicts would be useful, P3 expressed that he would feel more comfortable when working with other people if he could know in which parts of the system they are working:

"Well, in my case for example, I can think about the work we're doing with P4. I would have been more relaxed to know that P4 wasn't working on the same place I was working on. So, that would have been useful... and in the case we were working on the same thing, we would have chatted. No, I think it's useful." (P3)

The results of this run show that emerging conflicts were useful for the participants to communicate, exchange code, become familiar with one another's tasks, and to coordinate code check ins and merging. In addition, the participants (especially P3) felt that the information on emerging conflicts was helpful in the assignment, and can be helpful in their development environment.

Regarding the different views, P3 affirms that he prefers the graph view, because he can use it instead of Eclipse's package explorer. However replacing the package explorer is not the goal of this view. P4, on the other hand, expresses his aversion for graphs in general, and affirms that he prefers the list.

The suggestions on how improve the view of emerging conflicts were to add a quick view with the comparison of the conflicting code, or to have a shortcut from the conflict notification to a compare view of the code differences.

5.3 Run 3

The participants (P5, P6) of this run are Master students in Computer Science, with 4 and 5 years of experience with Java development. They have 4 years of experience with team development, but no experience with developing industrial size systems. They have 3-4 years of experience with using SCM, IDEs, specifically Eclipse for Java development, and JUnit, considering themselves advanced users. Both have indicated that they have no experience with Checkstyle.

Both participants have indicated to work often in teams of 2-4 people. P5 uses SVN and Git, checks in the code daily, and rarely has to resolve conflicts. P6 uses SVN, checks in the code once or twice per day, and has to resolve conflicts 1-3 times per week.

Problems. A couple of problems happened on R3 that might have influenced the behavior of the participants throughout the programming session, and their opinion about the emerging conflicts, and the study itself. First, a defect on Syde prevented the participants from seeing the emerging conflicts at the beginning of Task 2. When the problem was detected by the experimenter, the participants were towards the end of Task 2. This might have influenced the fact that the participants only communicated right before checking in the changes for this task. Second, P6 did not initialize the graph view correctly until the middle of Task 3, when he started to see the emerging conflicts that were already happening before that moment.

5.3.1 Observations from Videos

Due to technical problems with the video recording application that were not detected during the experiment, there is no screencast for P5's session to be analyzed. Hence, the analysis of this run is based on the events of P6 and on the communication log.

Table 6 shows the frequencies and descriptive statistics for events observed from P6, taken from a 104-minutes screencast of P6's computer. There is a total of 188 events, from which message notification events (36) are the most frequent, followed by interaction with list (28), communication (26), and interaction with graph (21). Distinct from the participants of the first 2 runs, communication events do not appear as the most frequent. However, notification events are the first in the list, which means P5 was trying to communicate with P6, who at many instances completely ignored the messages.

Table 6: Frequencies and descriptive statistics for observations of participant P6. The total duration of the events is given in seconds. The descriptive statistics refer to the duration (in seconds) of the events that have a range

Events	T1	Dur.	T2	Dur.	T3	Dur.	Total	Dur.	Median	Mean	Stdev.
Communicating	20	1060	3	43	3	71	26	1174	17	58	9.4
Conflict graph	0	0	0	0	10	1283	10	1283	17	45	58.2
Conflict list	1	179	7	993	1	1	9	1173	104	130	154.3
Checking in	2	15	2	51	3	33	7	99	11	14	9.4
Merging	2	27	1	24	2	79	5	130	15	26	26.8
Resolving conflict	0	0	0	0	1	45	1	45	45	45	-
Synchronizing	2	7	0	0	3	6	5	13	2	3	0.8
Updating	4	25	1	2	1	5	6	32	3	5	5.6
Viewing changes	4	214	3	151	1	87	8	452	56	57	19.2
Running tests	3	46	4	58	6	121	13	225	17	17	3.5
Message notification	25		5		6		36				
Interaction with graph	0		0		21		21				
Interaction with list	0		28		0		28				
Yellow conflict	0		3		2		5				
Red conflict	0		0		2		2				
Task starts	1		1		1		3				
Task ends	1		1		1		3				
Total	65		59		64		188				

Task 1. Before the participants start the assignment, the experimenter explains that they would have three related, but not identical implementation task. However, a few minutes before the participants start implementing Task 1, P5 asks P6 a technical question related to his task in a manner that assumes P6 has the same task. P6 simply ignores the question and replies saying he is writing Javadoc. Then, P6 asks P5 whether method `checkMax` also has to log, to which P5 argues that there is no such method in his code, and that he only needs to change method `checkCounter`. They have a long discussion about what the task is asking them to do, until they realize they have different tasks. Meanwhile, P6 finished the task and checks in the changes. When it is time for P5 to check in, he tells P6 he never did it through Eclipse. This is an unexpected information for the experimenter, since in the screening questionnaire, P5 informed that he had 3-4 years of experience with using SCM, IDEs, and specifically Eclipse for Java development.

What follows is that P5 tries merging the code, but ends up copying the code snippet he has implemented into method `checkMax` that P6 introduced, which breaks the code. Ignoring this fact, P5 proceeds with the check in. When P6 updates to the version P5 checked in and sees the errors, he simply reverts the code to the previous version (the one P6 had checked in before), checks it in again, and informs P5 he did not see any new checks, seemingly unaware that he just erased P5's code snippet. P5 says he is going to add it again and asks P6 whether he added new code, to which P6 answers "the resolution to the conflict you introduced with the faulty class". With this answer, it is clear that P6 thought he actually resolved the conflicts correctly, which is not true. After re-implementing his changes, now fixing errors and making the tests pass, P5 has difficulties checking in the changes into the repository, and asks the experimenter for help. After the experimenter explains to him how to proceed, he checks in the code, P6 updates it, and successfully runs the tests.

A careful observation of P6's screencast points out a few difficulties he has when dealing with conflicting versions in Task 1. He mistakenly erases the changes introduced by P5 without even trying to understand the meaning of these changes. He clearly thought he had resolved the conflicts, unaware that he only erased the changes introduced by P5. This behavior indicates that P6 is also fairly unexperienced with merging code, although he also indicated having 3-4 years of experience with using SCM, IDEs, and specifically Eclipse for Java development.

Task 2. Different from what happened in Task 1, in this task the participants do not communicate while implementing the changes. Due to a defect on Syde, the list of emerging conflicts remains empty until the participants finish implementing the changes related to this task and restart Eclipse. Meanwhile, P5 finishes his implementation and says he is going to check in the changes. P6 sees there are three emerging conflicts and waits for P5 to check in. While waiting for P5, P6 tries to investigate the emerging conflicts by clicking on them (probably trying to see the code differences in each of the cases).

When P5 informs P6 he can update, P6 opens the Compare editor with the latest version from the repository and copies the new code to his local copy, instead of using SVN's synchronize and update options. When he tries to commit the merged code, SVN complains that he does not have the newest version. After that, he updates the code, verifies whether the tests are still passing, and checks in the changes.

In this task, the communication between the participants was restricted to coordinating at check in time, different to what happened in the first task. They finished the task much faster, but still had trouble to merge changes.

Task 3. At the beginning of this task P6 does not follow the instructions to initialize the graph view correctly. He only realizes about his mistake after he finishes the implementation of the changes, when he starts to see the emerging conflicts that were already happening before that moment. Right after, P5 asks whether P6 has already checked in, to which P6 says he is waiting to be the one to resolve the conflicts.

After P5 checks in, P6 opens the Compare editor with the newest revision and copies the new code into his local version. During the process of copying, he introduces a code redundancy that breaks two tests. Meanwhile, P5 tells P6 to ask if he does not understand P5's code, and continues explaining what he did. After the explanation, P6 realizes what is the problem in the code that is breaking the tests, fixes them and tries to check in. Because again he did not update, SVN complains he has an outdated version. When he updates, SVN performs an automatic textual merge, which breaks the code. The fix is straight forward: Erase the code copied from the newest version into his code (since he had already done it through the Compare editor). Once more, P6 runs the tests and checks in the changes, finishing the assignment.

5.3.2 Discussion

The answers of P5 and P6 to the questionnaire indicate that P5 had to merge code and resolve conflicts in Task 1, while P6 chose to merge and resolve conflicts in Task 2 and Task 3. Overall, they think that communication was helpful to coordinate themselves to perform the tasks (agree), except P6 for Task 3, who disagreed it was helpful. When asked whether knowing about emerging conflicts was an incentive for them to communicate, they had slightly different opinion: P5 is neutral for Task 2, and agrees for Task 3; while P6 strongly disagrees for Task 2, and disagrees for Task 3. Their answers are a reflection of the behavior observed from the screencast: P5 was communicating much more and being more proactive than P6. When asked more specifically whether knowing about emerging conflicts helped to avoid them at check in time, P5 was neutral, while P6 agreed.

At the end of Task 1, when P5 had difficulties merging and checking in, P6 decided to wait on the following tasks to be the one to merge the code. Although the decision was taken without a discussion involved, P5 was happy to accept it in the following tasks. During the interview, the experimenter asked P6 whether he took this decision to complete the tasks faster, to which he answers positively. Thus, already expecting that the following tasks would involve resolving conflicts, the participants coordinated themselves to perform the tasks faster. The positive side of this decision is that the most experienced participant showed willingness to help the least experienced. The negative side is that having a preconceived notion that conflicts would happen might have influenced their behavior during Task 2 and Task 3.

Indeed, the communication between the participants in Task 2 and Task 3 was concentrated at check in time, which was mostly when they shared code snippets and discussed about their implementations. One interesting statement given by P6 is that after seeing the emerging conflicts, he could deduce what P5 was doing:

“They were also small tasks, so you can deduce after seeing the conflicts what the other was doing.” (P6)

Since the experimenter noticed during the run that the participants had little experience with using SVN through Eclipse, she asked them what they think about Eclipse’s support for resolving conflict, to which P6 answered it depends on the situation:

“Short answer is it depends. There are situations in which it’s straight forward ... there is an arrow that copies from the version that he developed. In other kinds of situation it’s tricky because if we are really working on the same piece of code and changing stuff, that is messy because you have to decide whether your version is correct or the other one is correct.” (P6)

When asked which visual metaphors they preferred, both answered the list, explaining why:

“It was clear that, ok, it’s red, then you have some conflicts. You read, and it specifies where and what’s going on. And the dots, the yellow, red, are really useful because you don’t spend time reading all of them, but you see, ok the red ones are really the important ones. You go through them.” (P5)

When asked to comment about the disadvantages of the graph, P6 explanation evidences that he did not understand how to initialize it:

“The disadvantage I found is that in the third task I missed the view because of the layout of Eclipse, because it was centered, but in the view I was seeing it was black. Then I had to realize that I have to scroll and the center was there and the graph evolves on the right. And then I looked at it just at commit time, I think.” (P6)

Even though he thinks in the beginning the graph was not showing up because he had to scroll down to see it, the fact is that he did not follow the steps given on the handout to create the graph when he initialized the view. This could have influenced his preference for the list.

When asked how they would like to see the information about emerging conflicts, they expressed preference for highlighting the code on the Java editor directly. They think the highlighting should be a layer that can be disabled if it becomes disturbing for the developer.

5.4 Run 4

Participant P7 is a PhD student at the University of British Columbia, and participant P8 is a PhD student at Federal University of Campina Grande and assistant professor at University of Feira de Santana. They are experienced developers, with respectively 5 and 7 years of experience in Java development, 5 and 4 years of experience in team development, and 2 and 3 years of experience in developing industrial size systems. Furthermore, they have used IDEs for 5 and 7 years, Eclipse for 3 and 5 years, SCM for 4 and 3 years, and JUnit testing for 2 and 4 years. None have previous experience with Checkstyle.

P7 currently uses SVN and works in a team of 2-3 people. He checks in weekly and deals with conflicts once a month. P8 currently uses CVS, and sometimes works in teams of 3-4 people, but not at the moment. He checks in daily, but rarely has to resolve conflicts in the current project.

Problems. A couple of problems happened on R4 that might have influenced the behavior of the participants throughout the programming session, and their opinion about the emerging conflicts and the study itself. First, there was a unplanned 30 minutes break between Task 1 and Task 2 because P7 had to meet a student who was working on a project with him. Second, at the beginning of Task 2 the connection was lost and the participants were working offline for some time until the experimenter detected the problem. As a consequence, the participants did not see conflicts as they emerged, but only after reestablishing the connection. Third, P7 ignored the instructions at the beginning of Task 3 and continued using the graph instead of the list.

Another problem is that P8 updated the tests to the version in the repository at the beginning of Task 3, which removed the test from the user’s workspace. After some time trying to locate the test class, P8 called the experimenter, who had to copy it back to the workspace. This disturbed the start of P8’s implementation, which gave P7 enough time to finish his implementation and check in the changes before P8 had effectively started, eliminating the concurrent implementation. Lastly, P7 could not stay for the debriefing interview, so it was conducted only with P8.

5.4.1 Observations from Videos

A couple of problems happened that influence the analysis of the data collected from the videos. First, the screencasts have the video accelerated in comparison with the voice. Since we ignore the voice, and it was not possible to compute the acceleration factor, the duration of events are skewed: the more towards the end an event is, the more skewed to a lower value than it originally is. Second, P7 kept the chat window outside the recording area, so communication events are missing for him. Checking the log, there are 40 communication events with a couple of exchanged messages in each of them for the entire session.

Table 7 shows the frequencies and descriptive statistics for events observed from P7, taken from a 98-minutes screencast of P7's computer. There is a total of 137 events, from which the most frequent are interaction with graph (41), running the tests (24), use of the graph view (20), and synchronization (13) events. Communication events were lost, because P7 kept the chat window outside the recording area. Moreover, message notification events do not exist on Gmail chat, which limits the observations drawn from P7's communication strategies to the chat log.

Table 7: Frequencies and descriptive statistics for observations of participant P7. The total duration of the events is given in seconds. The descriptive statistics refer to the duration (in seconds) of the events that have a range

Events	T1	Dur.	T2	Dur.	T3	Dur.	Total	Dur.	Median	Mean	Stdev.
Communicating	0	0	0	0	0	0	0	0	0	0	0
Conflict graph	0	0	13	1177	7	457	20	1634	26	82	110.9
Conflict list	0	0	4	24	0	0	4	24	6	6	1.9
Checking in	4	52	1	16	1	30	6	98	12	16	12.5
Merging	0	0	0	0	0	0	0	0	0	0	-
Resolving conflict	0	0	0	0	0	0	0	0	0	0	-
Synchronizing	4	22	2	20	7	61	13	103	11	8	8.3
Updating	2	20	2	7	3	14	7	41	4	6	5.1
Viewing changes	1	29	1	24	0	0	2	53	27	27	3.7
Running tests	7	81	11	108	6	19	24	208	4	9	8.3
Message notification	0		0		0		0				
Interaction with graph	0		28		13		41				
Interaction with list	0		9		0		9				
Yellow conflict	0		1		2		3				
Red conflict	0		0		2		2				
Task starts	1		1		1		3				
Task ends	1		1		1		3				
Total	20		74		43		137				

Table 8 shows the frequencies and descriptive statistics for events observed from P8, taken from a 92-minutes screencast of P8's computer. There is a total of 126 events, from which the most frequent are communication (30), interaction with graph (16), and synchronization (16) events.

Task 1. The participants start to implement their tasks, until P7 finishes the implementation and tries to communicate with P8, who does not respond. Given the situation, P7 proceeds with trying to check in his code, but P8 has already checked in his changes. Meanwhile, P8 sees that P7 was trying to talk to him, and answers to the chat saying he is going to update the code. What happens is that P7 does not manage to check in because there is already a new version of `MethodCountCheck` in the repository. In the process of updating to the newest version, P7 and P8 discuss about their tasks, P7 examines the differences between the versions and decides to overwrite his local copy, which means he completely erases the changes he implemented. It becomes clear that P7 did not realize the two implementations were semantically different, because after re-running the tests and getting two failures (the same ones he had in the beginning of the task), he says that P8's

Table 8: Frequencies and descriptive statistics for observations of participant P8. The total duration of the events is given in seconds. The descriptive statistics refer to the duration (in seconds) of the events that have a range

Events	T1	Dur.	T2	Dur.	T3	Dur.	Total	Dur.	Median	Mean	Stdev.
Communicating	15	314	6	160	9	149	30	623	11	21	25.2
Conflict graph	0	0	11	658	1	1	12	659	16	55	81.5
Conflict list	0	0	3	8	7	780	10	788	8	79	152.8
Checking in	1	23	1	20	2	23	4	66	16	16	5.8
Merging	0	0	4	32	2	26	6	58	7	10	7.0
Resolving conflict	0	0	0	0	0	0	0	0	0	0	-
Synchronizing	8	76	3	25	5	42	16	143	9	9	3.9
Updating	2	9	0	0	0	0	2	9	4.6	4.6	0.3
Viewing changes	1	26	1	134	2	117	4	277	65	69	58.5
Running tests	5	88	4	68	4	70	13	226	18	17	1.3
Message notification	0		0		0		0				
Interaction with graph	0		15		1		16				
Interaction with list	0		0		2		2				
Yellow conflict	0		1		2		3				
Red conflict	0		0		2		2				
Task starts	1		1		1		3				
Task ends	1		1		1		3				
Total	34		51		41		126				

changes broke his tests, which is not true. He then re-implement the changes, checks them in and informs P8, who updates the code and successfully re-runs the tests.

Task 2. The first half of this task is disturbed by the loss of connection with the server, which prevents the participants from seeing the first emerging conflicts as they occur. When P7 is about to finish and P8 is half way through the implementation, the experimenter detects the problem and asks them to reconnect. Straight away P8 observes the emerging conflict in `PlainTextLogger` and contacts P7 to tell him what methods he is modifying. P8 also says what he is changing, and they identify that they have conflicts only in method `addError`. They decide that P7 should check in his changes and P8 can handle the merging. After P7 checks in, P8 merges the code, which is straight forward, runs the tests, and checks in the code. P7 then updates and successfully re-runs the tests.

Task 3. In this task, P7 keeps on using the graph view instead of using the list. In addition, at the beginning of the task, P8 updates the tests to the version in the repository, which removes the test he has to fix from the workspace. After some time trying to locate the test class, P8 calls the experimenter, who has to copy it back to the workspace. Meanwhile, P7 keeps looking at the graph while implementing the changes, but no emerging conflict appears until he finished and checks in. At the same time P7 checks in, an emerging conflict appears, and P8 asks P7 whether he is changing `JsonLogger`. P7 says he has already checked in and asks P8 whether he was saving his file, to which P8 answers he was. The task ends with P8 finishing his implementation, merging P7's version into his code, and checking in. Unfortunately, no real parallel coding happened in this task.

5.4.2 Discussion

The answers of P7 and P8 to the questionnaire indicate that P7 had to merge code for Task 1 and Task 2, and P8 had to merge code for Task 2 and Task 3. They think that communication was helpful to coordinate themselves to perform the tasks (strongly agree - P7, agree - P8), except in the case of Task 3 for P8, who was neutral about it. For Task 2 and Task 3, both participants saw emerging conflicts and communicated as soon as they saw them (agree). However, they have different opinion on whether knowing about conflicts in advance helped them to avoid them at check in time (agree - P7, disagree - P8). Indeed, P8 had to merge code in the last 2 tasks, and the communication they had after seeing emerging conflicts revolved around coordinating who would check in first, and who would handle the merging.

During the interview, the experimenter asked how the communication took place after they saw emerging conflicts, to with P8 answered:

"I remember that I was asking him some questions about which methods he was changing... It seemed to be not really hard tasks, so it was easy to solve the conflict, but we still had to talk about it at least to decide who was going to fix first, who was going to commit the code first." (P8)

The coordination was kept shallow, and P8 did not manage to understand what the task of P7 was:

"No, not really. I just knew that he was changing some code in parallel with me. That was it." (P8)

At the beginning of Task 2 there was a disruption caused by a connection loss. When asked about it, P8 expressed that it disturbed the task:

"After we got the connection back, we saw it (conflict). That kind of disturbed the whole task. Made it feel a little bit artificial." (P8)

However, even with the disruption, P8 expressed that he preferred the graph view over the list:

"The graph. I think it looks better. Especially the graph that you filter out all the design, and only have the emerging design. That's more interesting to me. I think lists... they can be ok, but they don't allow you to focus on what's going on. When you have the graph, you immediately see the issues. There's not too much of information. There's just some colors that just show you like, 'oh, there's something going on here', and if you want to take a look, then you just point to the node and see what's going on. When you have a list, you kind of get lost with so much of information." (P8)

P8 expressed that on a project with very few people involved, he does not think there is a need for preemptive conflict detection, but in a project with many people and lots of concurrent modification, he might change his mind. In addition, when asked whether he could think of other ways to visualize emerging conflicts, he proposed to add cues on the package explorer. This way, he could use it in combination with Mylyn, and see the emerging conflicts only for the classes he is focused on without having to use another view for that.

5.5 Run 5

Participant P9 is a PhD student at the Federal University of Campina Grande, and P1 is a PhD student at the Federal University of Minas Gerais. They have, respectively, 5 and 7 years of experience in Java programming, 3 years of experience in team development, and 1 and 2 years of experience in developing industrial size systems. They have used SCM, IDEs, and specifically Eclipse for 5 and 6 years, and have 5 and 7 years of experience with JUnit. P9 has no previous knowledge of Checkstyle, while P10 has 1 years of experience with it as a user and considers himself comfortable with it.

P9 currently uses SVN and works in a team of 2 people. He checks in the code once a week, and has to resolve conflicts once a month. P10 uses Git, SVN, and Rietveld, and works in teams that vary from 2 to 5 people depending on the project. Currently he is working on a project with another person, thus he checks in the code every couple of days, but does not have to resolve conflicts frequently.

5.5.1 Observations from Videos

A couple of problems happened that influence the analysis of the data collected from the videos. The screencasts have the video accelerated in comparison with the voice. Since we ignore the voice, and it was not possible to compute the acceleration factor, the duration of events are skewed: the more towards the end an event is, the more skewed to a lower value than it originally is. In addition, the experimenter forgot to start recording P9's session, and only realized that at the end of Task 1. Thus, there are no events recorder for P9 for Task 1.

Table 9 shows the frequencies and descriptive statistics for events observed from P9, taken from a 58-minutes screencast of P9's computer. There is a total of 225 events, from which the most frequent are interaction with graph (63), communication (46), viewing the conflict graph view (31), and viewing the conflict list view (29) events. Message notification events do not exist on Gmail chat, hence were not captured.

Table 10 shows the frequencies and descriptive statistics for events observed from P10, taken from a 96-minutes screencast of P10's computer. There is a total of 131 events, from which the most frequent are communication (37), interaction with graph (20), and running test (16) events. Message notification events do not exist on Gmail chat, hence were not captured.

Table 9: Frequencies and descriptive statistics for observations of participant P9. The total duration of the events is given in seconds. The descriptive statistics refer to the duration (in seconds) of the events that have a range

Events	T1	Dur.	T2	Dur.	T3	Dur.	Total	Dur.	Median	Mean	Stdev.
Communicating			22	125	24	210	46	335	4	6	4.5
Conflict graph			29	679	2	45	31	724	6	23	52.2
Conflict list			0	0	29	472	29	472	10	16	18.3
Checking in			1	4	2	14	3	18	4	6	4.5
Merging			3	26	2	11	5	37	4	7	6.8
Resolving conflict			0	0	0	0	0	0	0	0	-
Synchronizing			5	20	5	14	10	34	3	3	6.2
Updating			1	3	1	2	2	5	2	2	0.3
Viewing changes			2	55	2	32	4	87	17	22	13.7
Running tests			6	33	9	67	15	100	4	7	6.2
Message notification			0		0		0				
Interaction with graph			58		5		63				
Interaction with list			0		6		6				
Yellow conflict			2		2		4				
Red conflict			1		2		3				
Task starts			1		1		2				
Task ends			1		1		2				
Total			132		93		225				

Although P9's screencast is missing the events of Task 1, he has generated almost double the number of event P10 did. In particular, P9 has a high number of events of communication and interaction with the views. This happens because P9 constantly switched between Eclipse and and the chat window with both taking the full screen, while P10 had a small chat window on top of Eclipse. Indeed, by inspecting the total duration of these events, it is possible to see that the total duration of P10's events is bigger than P9's, which is expected since the evens from P9's first task are missing.

Task 1. The participants start to implement their tasks, and P9 manages to finish it first. He checks in the code and lets P10 know there is new code in the repository. When P10 finishes his implementation, he checks the changes implemented by P9. When viewing the changes, he is not able to merge the code directly, because the compare view detects a conflict. He then manually resolve the conflict by copying the code from P10's version into his, making the necessary changes to conform with the refactoring he performed, marking the code as merged and checking it in. Then, P9 tells P10 to update and re-run the tests, which P10 does successfully. This task finishes smoothly, with P10 handling the conflict without breaking any tests.

Task 2. This task starts with both participants initializing the graph view and taking some time to interact with it. P9 spends more time interacting with the graph, while P10 goes to the implementation task. As soon as they see a warning of emerging conflict they start communicating, and P10 asks P9 which methods he is modifying. They tell each other which methods they are modifying, and discuss for a while about whether they can see this information from the graph until P9 finds out it is possible to see it. They decide P10 should commit the changes to `addError` (the method both changed) first, so P9 can merge it before they continue implementing the other methods. As soon as P10 checks in, the conflict warning on the graph becomes red and P9 notices it. He goes to the chat and sees that P10 has checked in the code. P9 then merges the code, finishes implementing, checks in the code and tells P10 to update. P10 updates, re-run the tests and checks in the changes to method `fileFinished`, and tells P9, who successfully updates the code and re-run the tests.

Table 10: Frequencies and descriptive statistics for observations of participant P10. The total duration of the events is given in seconds. The descriptive statistics refer to the duration (in seconds) of the events that have a range

Events	T1	Dur.	T2	Dur.	T3	Dur.	Total	Dur.	Median	Mean	Stdev.
Communicating	4	111	13	356	20	241	37	708	14	19	21.8
Conflict graph	0	0	11	1012	1	1	12	1013	51	84	109.1
Conflict list	0	0	0	0	6	558	6	558	66	93	77
Checking in	1	16	2	37	2	35	5	88	16	18	7.8
Merging	1	3	3	17	4	52	8	72	3	9	14.1
Resolving conflict	1	31	0	0	1	39	2	70	35	35	5.4
Synchronizing	1	10	3	19	3	10	7	39	4	6	3.4
Updating	0	0	0	0	0	0	0	0	0	0	-
Viewing changes	1	17	2	46	2	132	5	195	36	39	26.9
Running tests	3	62	5	91	8	87	16	240	17	15	5.6
Message notification	0		0		0		0				
Interaction with graph	0		20		0		20				
Interaction with list	0		0		2		2				
Yellow conflict	0		1		1		2				
Red conflict	0		2		1		3				
Task starts	1		1		1		3				
Task ends	1		1		1		3				
Total	14		64		53		131				

This task also finished smoothly, with the participants taking the strategy to break the check in into smaller ones when they notice a potential conflict emerging.

Task 3. In this task, instead of jumping into the code immediately, the participants start by asking each other what methods they have to modify. They find out there is a common method and decide to adopt the same strategy as in Task 2: Check in and merge the method they had in common as early as possible, and then continue implementing the other. P9 finishes the implementation of `addError` first and checks in for P10 to merge the code. P10 does it with no need to resolve conflicts, and checks it in with all tests referring to `addError` passing. Meanwhile, P9 finishes the implementation of `fileFinished`, and as soon as P10 checks in, P9 merges the new code and checks in the changes of `fileFinished`. P10 also finished the implementation of `fileStarted`, does the last merge and checks in for P9 to do the last update and re-run the tests successfully.

5.5.2 Discussion

The answers of the participants to the questionnaire indicate that P9 had to merge code in Task 2 and Task 3, while P10 merged code in all three tasks. Both participants think that communication was helpful to coordinate themselves to perform the task (agree - P9, strongly agree - P10). When asked about preemptive conflict detection, they had different opinion. P9 saw emerging conflicts and communicated right after seeing them (strongly agree), what he thought was very helpful to avoid resolving them at check in time (strongly agree). Conversely, P10 did not see emerging conflicts and did not think they triggered communication (strongly disagree). P10 was neutral about whether knowing about conflicts helped to avoid them at check in (neither agree nor disagree). P10's negative answer about seeing emerging conflicts, however, is not confirmed on the interview, when he talks about details of the two views and how they showed conflicts emerging while they were implementing the tasks.

In general, both participants showed similar behavior when updating the code changed by the other. They first synchronize the code, then view the differences between the local and the remote versions. After understanding the differences, they merge the code by using the built-in automatic merge function of Subversive. After that, they mark the file as merged and check it in. This process did not change with the introduction of preemptive conflict detection. What changed, and was also recognized by the participants, is the frequency and granularity of the check ins. Knowing in advance that they had conflicts, and where these conflicts were made them decide to resolve them before going on with coding. By keeping the check ins more frequent and the granularity of the changes smaller, they believe the conflict resolution becomes less complex:

"I think it did help, because we started communicating before actually committing all the code, and before doing other changes in other methods we started communicating when we detected conflicts. So it helped to see and synchronize what things we were adding and committing the conflicts first to first resolve all the conflicts and then continue." (P9)
"I think like it was easier to detect conflicts and do the merge only with small changes instead of looking through all the code." (P10)

When asked about which view they preferred, both said to prefer the list for this assignment in particular, but that the graph could be useful in other situations (e.g., when more people are involved, or when there are conflicts in depending classes):

"Well, in my opinion, I think the second view (list) is a bit easier to use, because it shows only the classes that are actually in conflict. So it filters for you, and you don't need to see the whole thing, you don't need to, even in the graph view you can go to the square class (node) that you changed and you can see the same thing, but you have to do it manually. You have to go there and see what is changing. Maybe the good thing is that you can see if dependencies are being checked at the same time... but I think the second view was much straight forward. You can see exactly what's wrong and resolve conflicts." (P9)
"I would agree that the second view is better, but the first view, like P9 said, I didn't think about this before like, you could see dependencies being changed, but the test didn't have any. We were always changing one file only. So the second view is better for that, because you have little changes and you can see. Maybe if we were changing larger files the first would be good." (P10)

P10 also reported that the graph was not so intuitive, and that they had difficulties to realize that by hovering over a node, it showed information of which methods of a class were in conflict:

"I think I didn't know this for a long while. I just saw the square changing colors and I think at the end of the task while I was waiting for P9 to commit something, I started playing around and then I saw that there was like 'oh, if I put the mouse here then I can see the changes', but it wasn't clear that, and it's not intuitive that you were gonna see the changes like that." (P10)

P9 gave a suggestion to improve the graph view by adding a search option to find a node quickly. When asked about other ways to warn developers of potential conflicts, P9 suggested to add annotation directly at the Java Editor to avoid having to use a separate view for that.

Lastly, when asked on whether preemptive conflict detection would be helpful on their everyday coding, both said it would be useful only in the context of a team greater than two:

"... if you code with someone else, there is two people and maybe that's easier to synchronize, but in the past definitely, when we worked on Ourgrid and stuff, there were always conflicts. Maybe having a large project and, maybe not even on the class scale there aren't two people editing the same class, but there's a dependency that both people are editing and there's gonna be a problem. And also, some people just don't want to commit right away, and you're doing the same thing he did or whatever. So that would be helpful." (P10)
"Nowadays the team is only two people working, so we don't really need something like that, because when we plan to do something on the code, we always try to share the most and try not to have conflicts, but on the planning phase, not afterwards when implementing. But actually, sometimes even with two people it happens. It happens sometimes with me, so it would be useful and even more useful on the past when I worked with 5-6 people in the same project, I think it would be even more useful. " (P9)

5.6 Run 6

The participants (P11, P12) of this run are Master students with, respectively, 5 and 4 years of experience in Java development, 6 years and 1 year of experience with team development, and no experience with development industrial size teams. They have 5 and 4 years of experience with using IDEs, 3 years of experience with Eclipse, 4 and 2 years of experience with using SCM, and 3 years and 1 year of experience with JUnit. They have no previous experience with Checkstyle.

The participants currently use SVN and often work in teams of 3-5 people (P11) and 5 people (P12). P11 reports that his check in frequency depends on how much he is involved in the project. For projects in which he has a relevant role, he checks in many times per day, because he prefers to check in frequently to avoid conflicts, though he has to deal with them once a week. P12 reports that he checks in often, and rarely has to resolve conflicts. He says sometimes he forgets to check out before start working, and that is when he mostly deals with conflicts.

5.6.1 Observations from Videos

Table 11: Frequencies and descriptive statistics for observations of participant P11. The total duration of the events is given in seconds. The descriptive statistics refer to the duration (in seconds) of the events that have a range

Events	T1	Dur.	T2	Dur.	T3	Dur.	Total	Dur.	Median	Mean	Stdev.
Communicating	4	1644	5	330	12	1016	21	2990	56	142	235.2
Conflict graph	0	0	4	1249	0	0	4	1249	185	312	309.6
Conflict list	0	0	0	0	1	2028	1	2028	2028	2028	-
Checking in	2	34	2	63	1	22	5	119	19	24	11.7
Merging	0	0	0	0	0	0	0	0	0	0	-
Resolving conflict	0	0	0	0	0	0	0	0	0	0	-
Synchronizing	0	0	0	0	0	0	0	0	0	0	-
Updating	0	0	1	5	3	11	4	16	4	4	1.2
Viewing changes	0	0	0	0	0	0	0	0	0	0	-
Running tests	1	15	3	33	4	47	8	95	11	12	2.3
Message notification	0		6		5		11				
Interaction with graph	0		3		0		3				
Interaction with list	0		0		4		4				
Yellow conflict	0		1		1		2				
Red conflict	0		1		2		3				
Task starts	1		1		1		3				
Task ends	1		1		1		3				
Total	9		28		35		72				

Table 11 shows the frequencies and descriptive statistics for events observed from P11, taken from a 92-minutes screencast of P11's computer. There is a total of 72 events, from which the most frequent are communication (21), message notification (11), and running tests (8) events. The number of events from P11's video is the lowest compared with all other participants, and there is a strong reason for that. P11 managed to finish the implementation of all the tasks before P12 (sometimes even before P12 had started), which means he did not have to merge or resolve conflicts throughout the assignment. Another observation is that after P12 had checked in, P11 was supposed to update his code and re-run the tests. However, he did not re-run the tests in any of the tasks.

Table 12 shows the frequencies and descriptive statistics for events observed from P12, taken from a 93-minutes screencast of P12's computer. There is a total of 132 events, from which the most frequent are message notification (41), communication (32), and running tests (25) events.

Task 1. The participants start to implement the task, and when P11 finishes, he asks P12 whether he is done, but P12 says he is still working on the task. P11 then waits for a while, but then decides to check in his changes to the repository. After a while, P12 also finishes and tries successive times to check in, but they failed for different reasons. In the first attempt, P12 is trying to check in the entire project, and keeps receiving a message that some folders are outdated in his copy. P12 then updates the code, which introduces conflicts between his new code and P11's one. Faced with a broken class, P12 tries to understand the changes introduced by P11. Meanwhile, P11 suggests P12 to copy the new code P12 introduced, revert the code, and paste the code back in. P12 reverts the code, but forgets to copy what he implemented before, thus he has to re-implement everything from scratch. After he finishes re-implementing his changes, he tries to check in the code twice with no success. After, telling to P11 he could not check in, P11 alerts him that he should only

Table 12: Frequencies and descriptive statistics for observations of participant P12. The total duration of the events is given in seconds. The descriptive statistics refer to the duration (in seconds) of the events that have a range

Events	T1	Dur.	T2	Dur.	T3	Dur.	Total	Dur.	Median	Mean	Stdev.
Communicating	13	196	9	161	10	106	32	463	11	14	9.8
Conflict graph	0	0	2	1057	0	0	2	1067	528	528	375.9
Conflict list	0	0	0	0	1	2031	1	2031	2031	3021	-
Checking in	5	150	1	30	1	43	7	223	30	32	11.7
Merging	1	10	1	8	0	0	2	18	9	9	1.1
Resolving conflict	1	174	1	41	0	0	2	215	107	107	94.2
Synchronizing	0	0	0	0	0	0	0	0	0	0	-
Updating	2	23	1	7	2	9	5	39	7	8	4.4
Viewing changes	0	0	0	0	0	0	0	0	0	0	-
Running tests	3	50	5	84	17	293	25	427	16	17	5.5
Message notification	14		12		15		41				
Interaction with graph	0		3		0		3				
Interaction with list	0		0		2		2				
Yellow conflict	0		1		1		2				
Red conflict	0		0		2		2				
Task starts	1		1		1		3				
Task ends	1		1		1		3				
Total	41		38		53		132				

check in the classes from the src folder. After four failed attempts, P12 checks in the code and informs P11. P11 does not re-run his tests and they finish Task 1.

Task 2. In this task, P12 has the initiative of asking P11 what his task is before they start implementing. However, none take any extra action after they share what they have to do. In addition, P12 starts by fixing a test that is of P11's responsibility to fix. P11 is fast implementing, and when he is about to finish, P12 starts the implementation of method `addError`. The warning of emerging conflict only appears when P11 is about to check in, and he immediately calls P12's attention to the existence of the conflict. P11 asks P12 whether he should check in, to which P12 answers positively. P12 directly updates the code, without inspecting the changes introduced by P11 first, which causes a conflict that breaks the code. This is when P12 realizes he and P11 implemented the same check (referring to the severity level). P12 continues the implementation, and when all the tests for `PlainTextLogger` are passing, he checks in the code and lets P11 know. Once more, P11 does not update the code, nor re-run the tests.

Task 3. Again in the beginning of the task, P12 asks P11 what he has to implement, but P11 takes a long time until he looks at the chat window. P11 finally answers P12 at the same time that a warning of emerging conflict in method `addError` appears. He asks P12 what he is doing, and calls P12's attention about the warning. P11 says he fixed the test related to the message, and P12 says he is working on the test related to the security level. They decide that P11 should check in first, and P12 merges the code, hence P11 checks in. To merge, P12 copies the code he implemented, reverts the local copy of `JsonLogger` to the latest version in the repository (the one P11 checked in), and pastes his code in the updated version. This work around prevents the appearance of conflicts, and can be considered an unconventional way of merging. P12 implements the second method of this task, checks in the code, and they finish the task once more with P11 not performing the last code update and test re-run.

5.6.2 Discussion

The answers of P11 and P12 to the questionnaire confirm that P12 was the one merging code in Task 1 and Task 2. For Task 1 one, they indicate that communication was helpful for them to coordinate and to perform the task (agree), even though P12 had problems merging the code in this task. For Task 2 they think that communication was helpful (agree), and that they communicated with each other as soon as they saw the warning of conflicts emerging. They think knowing about conflicts in advance helped them to avoid them at check in time (agree - P11, strongly agree - P12), even though it did not prevent P12 from having to merge. For Task 3, none indicates they had to merge, even though the screencast shows that P12 merged code. They indicate that communication was somewhat helpful to perform the task (agree - P11, neither agree nor disagree - P12). Lastly, they communicated with each other as soon as they saw conflicts emerging, and they think it was useful to help them to avoid merging code at check in time (agree - P11, strongly agree - P12).

When observing the videos of P11 and P12, it became clear that P12 was unfamiliar with using SVN through Eclipse, although on the screening questionnaire he indicated that he currently uses SVN. During the interview, the experimenter asked P12 whether he experienced difficulties when using SVN through Eclipse, to which he answered that this was the first time he was accessing SVN within Eclipse, because he usually code in Objective-C and uses SVN through command line. This is probably the reason why P12 exposed a different behavior from the majority of other participants when merging code: P12 did not synchronize the code, nor investigated the differences before updating it; in the first task even losing his implementation and having to redo it.

Apart from P12's lack of experience with using SVN through Eclipse, another phenomenon observed is that P11 was extremely fast to resolve the tasks, which almost prevented the warnings of emerging conflicts to be useful. They are especially useful when both developers are starting to implement a conflicting part of the code, or at least one of them is at the beginning. In this run, what happened in Task 2 and Task 3 was that P11 was finishing the implementation when P12 was starting his implementation. Thus it was still useful for P11 to wait for P12 to finish and check in while P12 was still in the beginning.

During the interview P11 and P12 expressed that knowing about conflicts when they emerge helped them to coordinate, and they did not need to communicate excessively to know what the other was modifying:

"I think it helped us to coordinate... We asked just 'Which methods are you modifying?' just to prevent some modification." (P11)

"We didn't go in the specific like 'well I'm implementing...', only the method. It was sufficient to coordinate because we knew when we were editing the same method." (P12)

When asked when they talked to each other, they confirmed that they communicated when they saw conflicts emerging, but also tried to coordinate in advance:

"I contacted him when I saw a conflict and we started discussing why there was this conflict." (P11)

"And there were times we contacted each other just at the beginning of the task before starting to do anything. Maybe I asked him 'What part of the code are you going to modify?'. Well, for the conflicts we used just the view on Eclipse" (P12)

Both P11 and P12 preferred to view the warnings of emerging conflicts on the list view:

"In this example I don't really have one that I prefer, but I think that if we had to work for hours in a project, then the second one (list) for me is better because in a table you have all conflicts organized. In the other I didn't see, because I didn't have many files open and so on, but maybe it becomes a really huge graph and it might be difficult to scroll." (P11)

"The last one (list) maybe could be better because we are just working in pairs. If we were working in a group of one hundred of developers, maybe the graph is not so easy to understand when you work." (P12)

The participants suggested to add visual cues directly in the Java Editor, that could be a mark on the ruler or a background color on the code.

6 Discussion

In this section we discuss the research questions based on the results reported in Section 5. We first report on specific behavior that happened with a single individual or in a single run, to then concentrate on behavior observed across different runs that could indicate a general behavior.

While analyzing the results, we noticed that the participants are naturally split into two distinct groups:

- Beginners: The participants of three runs (1, 3, 6) are master students with no experience with developing industrial size systems. Though they reported to have experience with SCM systems, some of them clearly have little experience, and struggled to perform the basic SVN operations from Eclipse.

- Advanced: The participants of the other three runs (2, 4, 5) are PhD students with at least one year of experience developing industrial size systems. Most of them reported to have previously worked in the industry, thus having practical experience in the field.

In the following, we take into account the two different groups to discuss each research question individually.

6.1 RQ1: How do developers behave when they have to merge code and resolve conflicts?

The answer of this question is derived from the data collected for Task 1, which involved the improvement of a method for one developer, and the refactoring of the code of the same method into a second one for the other developer.

Beginners. The participants of R1 seldom tried to coordinate, and concentrated the communication after the first check in. When they communicated, the intention was to let the other know there were new changes in the repository, instead of explaining what these changes were. The behavior of rushing to avoid dealing with merging was observed in one of the participants (P2).

The participants of R3 actively communicated in the beginning of the first task to understand what the task of the other was. They did not wait for check in time to start communicating, which was a different behavior from the one observed in the other runs. They were inexperienced on using SVN through Eclipse and struggled to merge code and resolve conflicts. In the first task, the participant who had no previous experience (P5) was the one to merge the code. During the merge, he introduced compilation errors in the code and checked in without solving them. The most experienced participant (P6) also showed lack of experience when he tried to solve the errors introduced by the other one, and inadvertently reverted the code to an unfinished version. P6 showed an unconventional behavior to deal with merging. He did not synchronize the files before inspecting the changes, and manually copied the new code into his local (and outdated) version. When he tried to check in, SVN complained that he did not have the newest version, which is when he updates his code. This behavior costed him time and increased the complexity of a simple merge.

In R6 only the behavior of one participant (P12) could be observed, because he was the only one merging in all tasks. Although he has experience with using SVN outside Eclipse, he has no previous experience with using it within Eclipse. Because of his lack of experience, he used a naïve strategy to merge code. First, he directly updated the code, which introduced compilation errors. After unsuccessfully trying to solve the errors, he then reverted to the version with the other participant's changes and lost his own, having to re-implement them. In terms of communication, the participants started to communicate at check in time, and kept it shallow, only sharing which methods they have to modify.

With the exception of R3, the participants concentrated their communication after the first check in and kept it shallow: they communicated either to inform the other there was new code in the repository, or to briefly explain which methods they were modifying.

The participants of R3 and R6 clearly struggled to deal with merging and showed unexpected strategies to resolve the conflicts that arose. Their unusual behavior when merging is the main indicator of their lack of experience with using SVN.

Advanced. In R2 the participants started to communicate after the first failed check in. They tried to coordinate through the chat and shared snippets of the code they implemented. However, the communication was insufficient for them to understand the changes they introduced had different behavior. This caused an extra loop of check out, implementation, testing, and check in. After they started to communicate, they kept the message exchange rate constant, actively coordinating with each other. Regarding the behavior during the actual merge, they followed the expected behavior: They synchronized the code, inspected the conflicting classes, merged the code by resolving the conflicts either manually or with the help of the Subversive assistant, and checked the code in.

The participants of R4 started to communicate at check in time, when one was about to check in and let the other know, who had already checked in first. They discussed to understand each other's change, however they did not realize their changes were semantically different and the removal of the changes of one of them broke the related tasks. Identical to what was observed in R2, this generated an extra check out, implementation, testing and check in. Regarding the behavior during the actual merge, they followed the expected behavior described above.

The participants of R5 also started to communicate at check in time. In terms of how they merged, they have similar strategy described below. The behavior of rushing to check in first to avoid dealing with merging was observed on one of the participants (P10). The other one had no difficulties to understand the semantic differences of the two implementations and merged the code successfully.

The behavior observed for the advanced participants is more constant than the one from the beginners. They all started to communicate after the first check in with the intention to let the other know there was new code, and then to discuss and understand each other's differences in order to handle the merge. However, in two instances the communication and the information provided by the Compare view were not sufficient for the participants to understand the changes they implemented were semantically different.

The behavior when merging was essentially the same for the three runs: to synchronize the code, inspect the conflicting classes understanding their differences, merge the code by resolving the conflicts either manually or with the help of the Subversive assistant, and check the merged code in.

Discussion. A common behavior observed was that developers communicated for coordination purposed, however they started to effectively communicate after the first check in, except for the participants of one run who started to coordinate earlier. The communication started with one developers telling the other that there was new code in the repository, and sometimes explaining the changes he did. In some cases, it evolved to a discussion in order to understand each other's implementation; in one instance with developers even sharing code snippets to help the comprehension. Thus, the coordination strategies were concentrated on alerting about new code and on understanding the necessary changes to merge successfully.

Regarding the merge itself, there was a clear distinction between beginners and advanced developers. While the advanced showed a common behavior, in two cases the beginners showed unexpected strategies to merge and resolve conflicts. Instead of synchronizing the code and inspecting it before merging, they directly updated the code, which performed the textual merge and introduced conflicts in the form of compilation errors. In one of the cases, to solve the compilation errors, the participant reverted the code to the latests version his colleague checked in, which completely erased his implementation.

The struggle to merge is expected when developers are inexperienced. However, an observation that might be unexpected for some is that in most cases the advanced developers had problems when merging. In two cases while analyzing the code differences and discussing with their counterparts, developers failed to understand their implementations were semantically different, consequently failing to merge successfully. Note that when they performed the merge, they solved the compilation errors, meaning that the direct conflict between the two different versions were solved. However, the tests related to the merged code failed, meaning that the participants failed to solve the semantic conflict that existed between the implementations. This observation confirms what Grinter observed through a field study [13]: Even experienced developers face problems merging code. Another behavior observed by Grinter and de Souza [9, 13] that we observed in at least two instances (when it was explicitly said by a participant) is that developers rushed to check in first to avoid dealing with merge.

Summary. The observations of how developers behave when they have to merge code can be summarized as follows:

- **Beginners adopt different naïve strategies.** Developers who are inexperienced with SVN tend to use different naïve strategies to merge code. These strategies seem to derive from strategies taken when using SVN through command line. They struggle to understand how to properly merge the code, sometimes erasing their own changes, or erasing the changes of others.

- **Advanced developers behave similarly, but struggle with merge.** They usually follow the strategy of synchronizing the code, inspecting the conflicting classes to understand their differences, merging the code by resolving the conflicts, and checking the merged code in. However, in most cases they had difficulties to understand how to properly merge the code, introducing errors that broke the tests. This confirms Grinter's observation that even experienced developers face problems merging code [13].

- **Developers communicate after the first check in.** With one exception, both groups of developers started to communicate after the first check in or after the first failed attempt to check in.

- **Communication is kept shallow.** In most cases, communication was kept shallow, with developers only informing the other about new code to the repository.
- **A few developers have deeper communication.** A few developers communicated more, trying to explain the changes they introduced or to understand the changes introduced by the others. Some even shared code snippets over the IM.
- **Some developers rush to check in first to avoid dealing with merge.** This behavior we observed confirms previous findings that developers try to avoid dealing with merge by check in changes before their colleagues [9, 13].

6.2 RQ2: How does this behavior change when information of emerging conflicts is present?

The answer to this question is derived from the data collected for Task 2 and Task 3. In these tasks developers had the aid of preemptive conflict detection to alert them of emerging conflicts in real time.

Beginners. The behavior of participants from R1 related to merge strategies did not change when emerging conflicts were introduced. Their communication triggered by emerging conflicts did not facilitate the resolution of conflicts. In addition, during the interview the participants shared that they thought they did not understand the concept of preemptive conflict detection, which might be the explanation for the stagnation of behavior. The communication shifted to start when conflicts emerged, thus a bit before check in time.

The behavior of the participants of R3 changed in an unexpected way when the information of emerging conflicts was present. Instead of intensifying the communication and starting to communicate earlier, the participants did the opposite: They communicated less, and only started to communicate at check in time. During the interview, the most experienced participant attributed this change of behavior to the fact that the tasks were small, so they could understand what the other was doing just by reading the information available on the warnings of emerging conflicts. However, the fact that they struggled to merge code and resolve the conflicts on the tasks in which preemptive conflict detection was present is evidence that they did not understand the task of the other enough to perform a smooth merge. The participants adopted a strategy to concentrate the resolution of conflicts with the most experienced one, with the aim of speeding up the resolution.

For the participants of R6, communication started earlier. On the second task, it started as soon as a conflict emerged, and the participants exchanged messages to understand which methods they were changing after one of them check in first. On the third task, one of the participants attempted to start communicating at the beginning of the task, but the other only answered when the first conflict emerged. In terms of dealing with merge, one participant updated the code without previous inspection, which again introduced compilation errors, but he was able to fix them and finish his implementation before checking in. In the last task, he used a work around: He copied his code, overrode and updated the changed class, and pasted the code back in the class. This avoided the appearance of compilation errors and he managed to finish the task and check in the code.

In terms of communication, there were changes in the behavior of the participants. In R3 and R6 the participants started to communicate earlier, when conflicts started to emerge. In the last task of R6, one of the participants even attempted to start communicating as soon as the task started. Also at R6 we observed a slight change in the way participants coordinated: They exchanged messages that actually explained what changes they performed instead of only informing the other that there were new changes. However, in R1 we did not observed a change of behavior. Participants did not start to communicate before check in time, probably because they did not understand the concept of preemptive conflict detection.

In terms of dealing with merge, the change of behavior was also different in each run. In R1 there were no significant changes, while in R3 and R6 a few changes were observed. Participants of R3 decided to leave the resolution of conflicts to the most experienced developer with the goal of speeding up the task completion, although he also struggled to perform the merges. In R6 we observed a gradual change in the behavior of the participant who resolved the conflicts. First, he was aware of existing conflicts, but did not know how to benefit from it, thus taking the same steps to merge code as in the first task, which introduced compilation errors. Then, he adopted the following strategy to successfully avoid break the code when merging: He copied his code, overrode and updated the changed class, and pasted the code back in the class. The strategy he used to merge is unusual for those who use SVN through Eclipse, but very similar to what would be expected from someone using SVN through command line.

Advanced. The participants of R2 started to communicate as soon as a warning of emerging conflict appeared, bringing the coordination to an earlier stage. They kept on sharing code snippets to understand each other's code. They tried to coordinate the check ins more efficiently by allowing the least experienced developer to check in first, so the most experienced one could deal with merging. They also split the check ins into smaller ones to reduce the complexity of a merge, when the merge was necessary.

The participants of R4 started to communicate earlier, when conflict warnings emerged. They identified which method had conflicts and decided who would check in first and who would be responsible for merging. Although coordination was kept shallow, and they did not need to get into details of the implementation of the other to deal with merging, they still felt the need to talk about the emerging conflict at least to decide who would merge.

The participants of R5 started to communicate earlier. On the second task, they started to communicate when they saw conflicts emerging and shared which methods they had to implement to identify which ones they had in common. On the last task, the communication started even earlier, when they started the task. They also coordinated by identifying which method would probably be conflicting and decided to check in the changes to it as soon as possible. In terms of merging, they decided to adopt the following strategy: break the check ins into smaller ones comprising of only the conflicting method first, and then the other methods. With this strategy, they merged the code and resolved the conflicts as early as possible, doing all of them successfully.

We observed a change of behavior in all three runs for both communication and merge strategies. In three cases participants started to communicate earlier, when conflicts started to emerge. For coordinating the merge itself, they had different strategies. In R2 participants decided to let the most experienced one deal with merging, and intensified their explanations of the changes they did. They also broke the check ins into smaller ones to reduce the complexity of the merge. The participants of R4 coordinated to decide who would change first and who would merge, however the did not give preference to one person to merge code. In R5 the participants also broke the check ins into smaller ones, though they did not assign a specific person to handle the merge. In general, participants had no difficulties in merging the code that they previously knew was conflicting.

Discussion. First of all, changes in the behavior of participants were observed, with some commonalities among the runs. Except for R1, participants started to communicate when conflicts emerged instead of doing it so only at check in time. Some of them also intensified the explanation of their changes to their counterparts, helping them to understand better what needed to be merged. This finding is more significant than the ones from previous studies [2, 10], because while they found that communication increased, we actually observed that communication was deeper, with developers giving more detailed explanations, and started earlier.

The participants adopted two new strategies to coordinate their activity. The first one was to let the most experienced developer deal with merging with the goal of speeding up the task completion. The second strategy observed was to split the check ins into smaller ones to reduce the complexity of the merges.

In the case of beginners, the ones who adopted the first coordination strategy still had problems with merging, because the most experience participant struggled to resolve conflicts. The ones who only brought the communication and intensified their explanations of new changes had a change on the merge strategy, finishing the last task more efficiently than in the first two tasks.

In the case of advanced, all intensified the coordination in the sense that they discussed who should check in first and who should be responsible for merging. In two runs, they also decided to split the check ins into smaller ones. The increase of awareness of existing conflicts together with increase in coordination helped them to perform the merge more successfully, even though the strategy to merge itself did not change. The participants still used the same strategy as in the first task: to synchronize the code, inspect the conflicting classes to understand their differences, merge the code by resolving the conflicts either manually or with the help of the Subversive assistant, and check the merged code in.

Summary. The observations of how the developers' behavior changed when exposed with notifications of emerging conflicts can be summarized as follows:

- **Developers started to communicate earlier and had deeper communication.** With the exception of one run, all developers started to communicate before check in time. Most of them also put more effort in explaining to their counterpart the changes they introduced.

- **Developers coordinated more effectively.** Two new coordination strategies were adopted by developers in different runs: to discuss and determine who should check in first and who should merge (giving preference to the most experiences developer to merge); and to split the check ins into smaller ones to reduce the complexity of the merge.

- **Beginners showed a slower change of behavior.** They had some difficulties to understand the concept of preemptive conflict detection. In consequence, they had a slower change of behavior. In one case the gradual change of behavior led the participants to successfully deal with merging.

- **Advanced developers succeeded in merging code.** Differently from what happened in the first task, developers managed to merge code successfully. We attribute this success to a higher level of awareness, and deeper communication and coordination.

6.3 RQ3: How do developers perceive different approaches to deliver information on emerging conflicts?

The goal of preemptive conflict detection is to raise the awareness of developers about changes of others that can impact their own work [25]. This information, however has to be delivered in a non-intrusive way to avoid deviating the developers' attention from coding.

This study has the secondary goal of understanding which views developers prefer to use to receive information of emerging conflicts. We have exposed developers to two different approaches: a view that shows conflicts as a list of items, with most of the information delivered as text; and a view that shows a graph of classes that can be easily reduced to the classes that the developer opens, and that delivers information on conflicts visually by changing the color of the graph node. During the interview we asked the participants' opinion on the views and also asked them to give suggestions of other ways to visualize this information.

Participants' opinion. In the following we present a summary of preferences and comments of the participants about the two views.

One participant of R1 prefers the graph, because he thinks the color metaphor in the nodes of the graph (representing the classes) is a good way to visualize the information. The other participant prefers the list, though he did not have a chance to use the graph.

One of the participants of R2 prefers the graph view, but because he can use it instead of the package explorer (he does not like the package explorer metaphor) to keep track of the classes he has recently opened. This emphasis of this participant's choice is not in the information of emerging conflicts itself, but on the possibility of replacing the package explorer with the graph. The other participant prefers the list because he has aversion for graphs in general. Hence, both participants have reasons other than the visual presentation of the conflicts to prefer one view over the other.

Both participants of R3 prefer the list metaphor, because one can identify the severity and the location (class and method) of the conflict just by looking at the items of the list, while with the graph one needs to hover over to see the location. In short, the participants prefer the metaphor that provides for less effort to get the most information in a shorter period.

One of the participants of R4 kept using the graph, however it is not known whether he prefers it over the list, because he could not stay for the interview. The other participant prefers the graph, because it enables him to focus on the classes he is working on, meaning that he will not see warnings of conflicts in classes that he is not working on at the moment if he enables the option of seen only the nodes of classes that he opens. He also prefers the visual layer added on the graph when there is an emerging conflict, which is easy to spot. He thinks that the list has too much information.

Both participants of R5 prefer the list for this assignment, but think the graph could be useful in other situations. They think the advantages of the list are that it is more straight forward and shows only the classes that are actually in conflict. On the other hand, they think the graph can help to show when dependencies are in conflict, because the edges show call dependency (or inheritance). Also, they think the graph might be more useful when changing larger files. One of the participants reported that the graph was not so intuitive and they had difficulties to realize that hovering over the graph would show information of which methods of a class are in conflict. Thus, it took him longer to get used to the graph than to the view.

Both participants from R6 prefer the list, because it shows all conflicts and details in a compact manner. They think the graph can grow to a huge size, if the system is large, and it can become difficult to see the conflict warnings.

Summary. The majority of the participants prefer the list (eight of them), and the strongest reason is that the list presents the important information in a direct and condensed manner. Thus, developers only need to look at the view to get all the information they need about an emerging conflict. Instead, to get the details of emerging conflicts in the graph, one needs to hover over the node.

Only three of the participants prefer the graph, with different reasons for their choice. One prefers it because he can just replace the package explorer with it. Another one prefers the graph because he thinks the color metaphor in the nodes of the graph (representing the classes) is a good way to visualize the information. The last participant prefers it because it enables him to focus on the classes he is working on, meaning that he will not see warnings of conflicts in classes that he is not working on at the moment. He also prefers the visual layer added on the graph when there is an emerging conflict, which is easy to spot. The last participant also argued that the list shows too much information at once.

Suggestions on other ways to visualize emerging conflicts. Participants were asked to suggest other ways to visualize information on emerging conflicts, and surprisingly many of them had the same suggestion: to show this information directly in the Java editor.

The participants of R1 think it is beneficial to show the information on the editor, and suggested to either change the background color or to add markers on the ruler. The participants of R2 and R3 suggested to visualize emerging conflicts by highlighting the code in the Java editor. This highlighting should be a layer that can be disabled if it becomes disturbing for the developer. The suggestion of one participant of R4 is to add cues in the package explorer, because it can be combined with Mylyn to focus on code that is being changed right now and avoid the use of an extra view on Eclipse. The participants of R5 and R6 suggested annotations directly on the Java editor to avoid the use of a separate view. These annotations could be a marker on the ruler or a background color on the editor.

Even though adding information directly in the Java editor is a less intrusive approach, developers showed a strong preference to have it there. Having an enable/disable option should be enough to let a developer disable it if he feels disturbed by the notifications. Showing information on emerging conflicts only at the class level in the Java editor would prevent developers from receiving overall information on the system. Hence, we think that providing the views as an alternative way to look at conflict notifications is also helpful.

7 Concluding Remarks

In the recent years, there has been a significant effort to increase awareness of distributed teams by supporting coordination across multiple developers working in parallel on the same code base [2, 14, 18, 24, 25, 3, 7, 23]. These approaches promote workspace awareness by detecting in real time concurrent modification to software artifacts, especially those that are potentially conflicting: concurrent changes that are likely to cause merge conflicts at check in time.

There has been a limited number of studies [2, 10, 25] to investigate whether the adoption of tools to promote workspace awareness is beneficial to developers. Their initial findings suggest that, when preemptive conflict detection is introduced, the frequency of communication increases, there is a reduction in overlap work, and an increase in the detection and resolution of conflicts. However, some fundamental questions concerning the concept of preemptive conflict detection remained open, e.g., "Were the changes observed in these studies beneficial to developers?", "Did the developers' strategies to deal with merging change?", "Is the information being delivered disturbing?", "Would they prefer to get them in a different way?".

In this work, we have investigated some of these open questions by concentrating in understanding whether the behavior of developers change when they are exposed to preemptive conflict detection, and whether this change is beneficial to them. To perform the investigation, we first devised different ways to deliver information on emerging conflicts on the IDE, with one of them being the adaptation of previous works [18, 25], and another one being inspired by tools that show emerging design [7, 23, 26]. Then, we designed and conducted a qualitative user study in a laboratory setting with pairs of developers who had to resolve a few tasks in collaboration. We collected data from questionnaires, interviews, observations, and documentation, and analyzed it in a iterative process to allow for the emergence of our findings.

We first reported on the behavior of developers when dealing with merging code without the help of preemptive conflict detection. Some of our findings conform with the ones from previous studies: developers, even the experienced ones, struggle with merging [13]; and some developers tend to rush to check in first to avoid dealing with merging [9, 13]. Other important findings are: developers only start to communicate, and consequently to coordinate, after the first (attempted) check in; and in most cases the communication remains shallow, with only a few developers putting effort in explaining to their counterpart the changes they have performed. In terms of how they resolved the conflicts, experienced developers had similar behavior, while beginners used different, and mostly naïve, strategies.

Then, we reported on the behavior of developers after introducing the concept of preemptive conflict detection and allowing them to use it during the programming tasks. We have found significant change in their behavior. First of all, they started to communicate earlier, usually right after the first information on emerging conflicts appeared. The early communication gave them the opportunity to coordinate better by adopting different strategies than what they would normally do. We observed two new strategies: to discuss and decide upfront who would deal with merging the conflicting code; and to break the check ins into smaller ones aiming at reducing the complexity of the merge.

These changes of behavior proved to be beneficial, given that most of them succeeded in merging when preemptive conflict detection was present, in contrast with previous struggle when this information was not available. We believe that having the information at earlier moment than at check in allowed them not only to coordinate ahead, but also to gradually understand the changes their colleagues were doing, and how their implementations complemented each other. Thus, it also helped developers to understand beforehand how to merge the code without introducing errors or breaking tests.

Developers also gave feedback on the different ways information on emerging conflicts was delivered, indicating their preferences and suggesting other ways. Most of them showed preference to see the list of emerging conflicts over the graph with overlaid information, though some showed special reasons to prefer the graph. One of these reasons is that with the graph, one can visualize the dependencies of a class in conflict, which might be impacted by the resolution of the conflict. Most of the suggestions indicated their desire to see emerging conflicts directly on the Java Editor, with the annotations on the left side of the editor we implemented but did not include in the user study as one of the options.

This study has shown that the changes in the behavior of developers when exposed to preemptive conflict detection are beneficial in terms of having more effective communication and coordination, as well as in terms of resolving conflicts with a higher success rate. In addition, we believe that different ways to visualize this information are complementary to one another, because developers have different preferences, and the fact that they can choose among different options might help them in the adoption of preemptive conflict detection. Therefore, there is potential for preemptive conflict detection to be adopted by practitioners, though more research towards effective ways to deliver this information in IDEs should be conducted.

References

[1] R. Barbour. *Introducing Qualitative Research.* Sage, 2008.

[2] J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson. Fastdash: a visual dashboard for fostering awareness in software teams. In *Proceedings of CHI 2007 (25th SIGCHI Conference on Human Factors in Computing Systems)*, pages 1313–1322. ACM, 2007.

[3] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Proactive detection of collaboration conflicts. In *Proceedings of ESEC/FSE 2011 (European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering)*, page to be published. ACM Press, 2011.

[4] E. Carmel. *Global Software Teams - Collaborating Across Borders and Time Zones.* Prentice Hall, 1999.

[5] B. Cornelissen, A. Zaidman, and A. van Deursen. A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering*, 99, 2010.

[6] J. Creswell. *Qualitative Inquiry & Research Design.* Sage, 2nd edition, 2007.

[7] I. da Silva, P. Chen, C. V. der Westhuizen, R. Ripley, and A. van der Hoek. Lighthouse: Coordination through emerging design. In *Proceedings of ETX 2006 (OOPSLA Workshop on Eclipse Technology eXchange)*, pages 11–15. ACM Press, 2006.

[8] D. Damian, L. Izquierdo, J. Singer, and I. Kwan. Awareness in the wild: Why communication breakdowns occur. In *Proceedings of the ICGSE 2007 (International Conference on Global Software Engineering)*, pages 81–90. IEEE Computer Society, 2007.

[9] C. R. B. de Souza, D. Redmiles, and P. Dourish. Breaking the code, moving between private and public work in collaborative software development. In *Proceedings of GROUP 2003 (International ACM SIGGROUP Conference on Supporting Group Work)*, pages 105–114. ACM Press, 2003.

[10] P. Dewan and R. Hegde. Semi-synchronous conflict detection and resolution in asynchronous software development. In *Proceedings of ECSCW 2007 (the 10th European Conference on Computer Supported Cooperative Work)*, pages 24–28. Springer, 2007.

[11] P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. In *Proceedings of CSCW 1992 (ACM Conference on Computer-supported Cooperative Work)*, pages 107–114. ACM Press, 1992.

[12] K. Dullemond, B. van Gameren, and R. van Solingen. Virtual open conversation spaces: Towards improved awareness in a gse setting. In *Proceedings of the ICGSE 2010 (5th IEEE International Conference on Global Software Engineering)*, pages 247–256. IEEE Computer Society, 2010.

[13] R. Grinter. Supporting articulation work using software configuration management systems. *Computer Supported Cooperative Work*, 5(4):447–465, 1996.

[14] M. L. Guimarães and A. Rito-Silva. Towards real-time integration. In *Proceedings of CHASE 2010 (the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering)*, pages 56–63. ACM, 2010.

[15] A. Guzzi, L. Hattori, M. Lanza, M. Pinzger, and A. van Deursen. Collective code bookmarks for program comprehension. In *Proceedings of ICPC 2011 (19th IEEE International Conference on Program Comprehension))*, pages 101–110, 2011.

[16] J. Hagedorn, J. Hailpern, and K. G. Karahalios. Vcode and vdata: illustrating a new framework for supporting the video annotation workflow. In *Proceedings of the AVI 2008 (The working conference on Advanced visual interfaces)*, pages 317–321. ACM, 2008.

[17] L. Hattori and M. Lanza. Syde: A tool for collaborative software development. In *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering)*, pages 235–238, 2010.

[18] R. Hegde and P. Dewan. Connecting programming environments to support ad-hoc collaboration. In *Proceedings of ASE 2008 (23rd IEEE/ACM International Conference on Automated Software Engineering)*, pages 178–187. IEEE CS Press, 2008.

[19] J. Herbsleb, A. Mockus, T. Finholt, and R. Grinter. Distance, dependencies, and delay in a global collaboration. In *Proceedings of CSCW 2000 (ACM Conference on Computer Supported Cooperative Work)*, pages 319–328. ACM Press, 2000.

[20] T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.*, 28(5):449–462, 2002.

[21] C. O'Reilly, P. Morrow, and D. Bustard. Improving conflict detection in optimistic concurrency control models. In *Proceedings of the 2001 ICSE Workshops on SCM 2001, and SCM 2003 conference on Software configuration management*, SCM'01/SCM'03, pages 191–205. Springer-Verlag, 2003.

[22] D. E. Perry, H. P. Siy, and L. G. Votta. Parallel changes in large-scale software development: an observational case study. *ACM Trans. Softw. Eng. Methodol.*, 10(3):308–337, 2001.

[23] T. Proenca, N. Moura, and A. van der Hoek. On the use of emerging design as a basis for knowledge collaboration. *New Frontiers in Artificial Intelligence*, 6284:124–134, 2010.

[24] A. Sarma, G. Bortis, and A. van der Hoek. Towards supporting awareness of indirect conflicts across software configuration management workspaces. In *Proceedings of ASE 2007 (22nd IEEE/ACM International Conference on Automated Software Engineering)*, pages 94–103. IEEE CS Press, 2007.

[25] A. Sarma, D. Redmiles, and A. van der Hoek. Empirical evidence of the benefits of workspace awareness in software configuration management. In *Proceedings of FSE 2008 (16th ACM SIGSOFT International Symposium on Foundations of Software Engineering)*, pages 113–123. ACM Press, 2008.

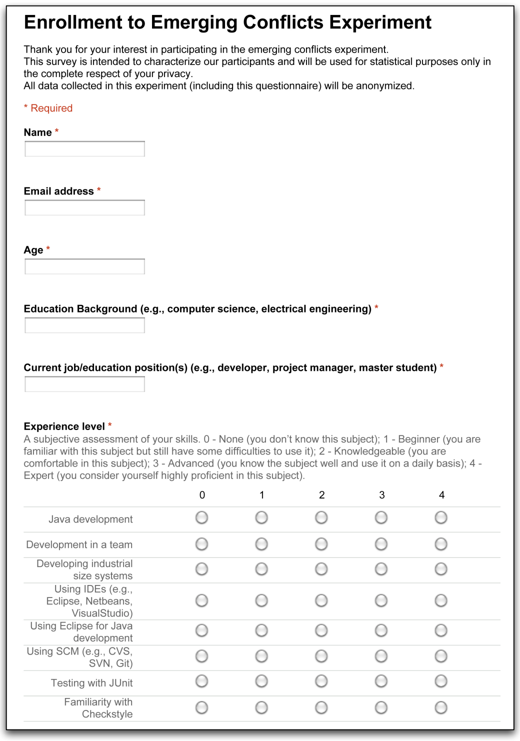
[26] F. Servant, J. A. Jones, and A. van der Hoek. Casi: preventing indirect conflicts through a live visualization. In *Proceedings of CHASE 2010 (the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering)*, pages 39–46. ACM, 2010.

[27] B. van Rompaey and S. Demeyer. Estimation of test code changes using historical release data. In *Proceedings of WCRE 2008 (15th Working Conf. on Reverse Engineering)*, pages 269–278. IEEE Computer Society, 2008.

[28] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16:325–364, 2011.

A Screening Questionnaire

Using Google Docs[3], we designed an online questionnaire that served both to provide an easily accessible platform for the participants to enroll and to allow us to capture their personal and experience information.



Enrollment to Emerging Conflicts Experiment

Thank you for your interest in participating in the emerging conflicts experiment.
This survey is intended to characterize our participants and will be used for statistical purposes only in the complete respect of your privacy.
All data collected in this experiment (including this questionnaire) will be anonymized.

* Required

Name *

Email address *

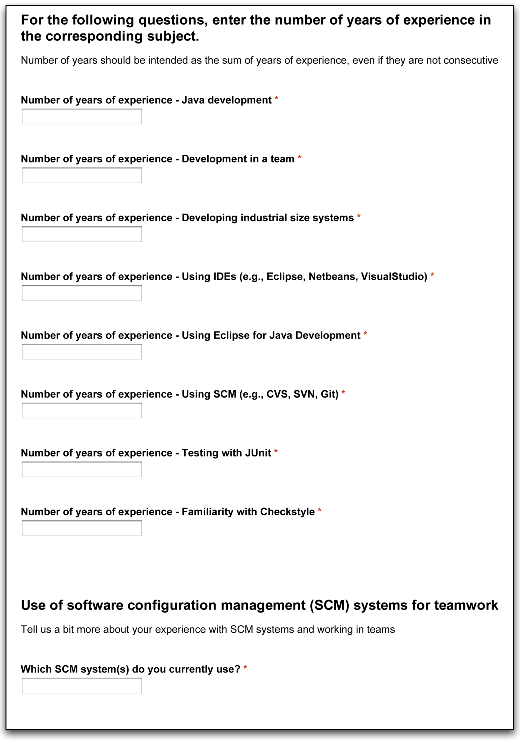
Age *

Education Background (e.g., computer science, electrical engineering) *

Current job/education position(s) (e.g., developer, project manager, master student) *

Experience level *
A subjective assessment of your skills. 0 - None (you don't know this subject); 1 - Beginner (you are familiar with this subject but still have some difficulties to use it); 2 - Knowledgeable (you are comfortable in this subject); 3 - Advanced (you know the subject well and use it on a daily basis); 4 - Expert (you consider yourself highly proficient in this subject).

	0	1	2	3	4
Java development	○	○	○	○	○
Development in a team	○	○	○	○	○
Developing industrial size systems	○	○	○	○	○
Using IDEs (e.g., Eclipse, Netbeans, VisualStudio)	○	○	○	○	○
Using Eclipse for Java development	○	○	○	○	○
Using SCM (e.g., CVS, SVN, Git)	○	○	○	○	○
Testing with JUnit	○	○	○	○	○
Familiarity with Checkstyle	○	○	○	○	○



For the following questions, enter the number of years of experience in the corresponding subject.

Number of years should be intended as the sum of years of experience, even if they are not consecutive

Number of years of experience - Java development *

Number of years of experience - Development in a team *

Number of years of experience - Developing industrial size systems *

Number of years of experience - Using IDEs (e.g., Eclipse, Netbeans, VisualStudio) *

Number of years of experience - Using Eclipse for Java Development *

Number of years of experience - Using SCM (e.g., CVS, SVN, Git) *

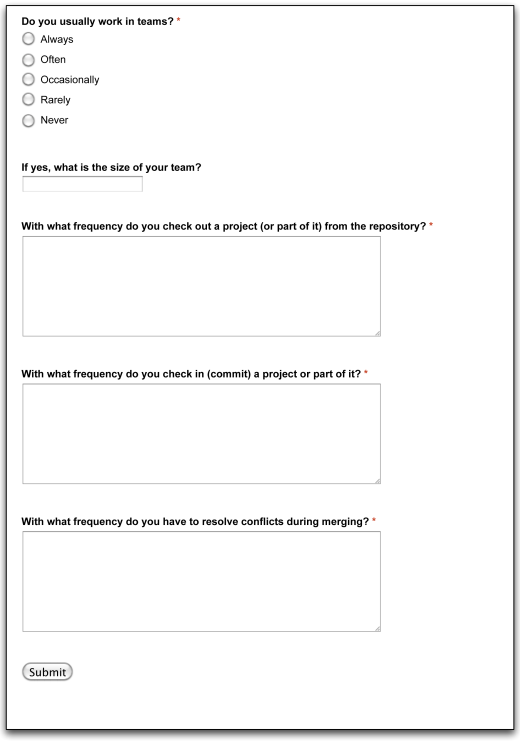
Number of years of experience - Testing with JUnit *

Number of years of experience - Familiarity with Checkstyle *

Use of software configuration management (SCM) systems for teamwork

Tell us a bit more about your experience with SCM systems and working in teams

Which SCM system(s) do you currently use? *



Do you usually work in teams? *

- ○ Always
- ○ Often
- ○ Occasionally
- ○ Rarely
- ○ Never

If yes, what is the size of your team?

With what frequency do you check out a project (or part of it) from the repository? *

With what frequency do you check in (commit) a project or part of it? *

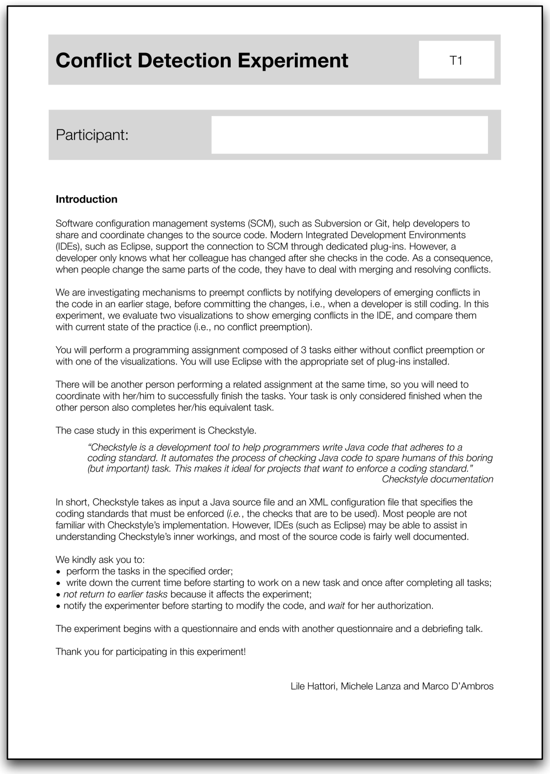
With what frequency do you have to resolve conflicts during merging? *

Submit

[3]http://docs.google.com

B Handout

The participants were given a handout with instructions about the assignment and the tasks. In the following an illustration of the handout is shown.



Conflict Detection Experiment T1

Participant:

Introduction

Software configuration management systems (SCM), such as Subversion or Git, help developers to share and coordinate changes to the source code. Modern Integrated Development Environments (IDEs), such as Eclipse, support the connection to SCM through dedicated plug-ins. However, a developer only knows what her colleague has changed after she checks in the code. As a consequence, when people change the same parts of the code, they have to deal with merging and resolving conflicts.

We are investigating mechanisms to preempt conflicts by notifying developers of emerging conflicts in the code in an earlier stage, before committing the changes, i.e., when a developer is still coding. In this experiment, we evaluate two visualizations to show emerging conflicts in the IDE, and compare them with current state of the practice (i.e., no conflict preemption).

You will perform a programming assignment composed of 3 tasks either without conflict preemption or with one of the visualizations. You will use Eclipse with the appropriate set of plug-ins installed.

There will be another person performing a related assignment at the same time, so you will need to coordinate with her/him to successfully finish the tasks. Your task is only considered finished when the other person also completes her/his equivalent task.

The case study in this experiment is Checkstyle.

> *"Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard. It automates the process of checking Java code to spare humans of this boring (but important) task. This makes it ideal for projects that want to enforce a coding standard."*
> *Checkstyle documentation*

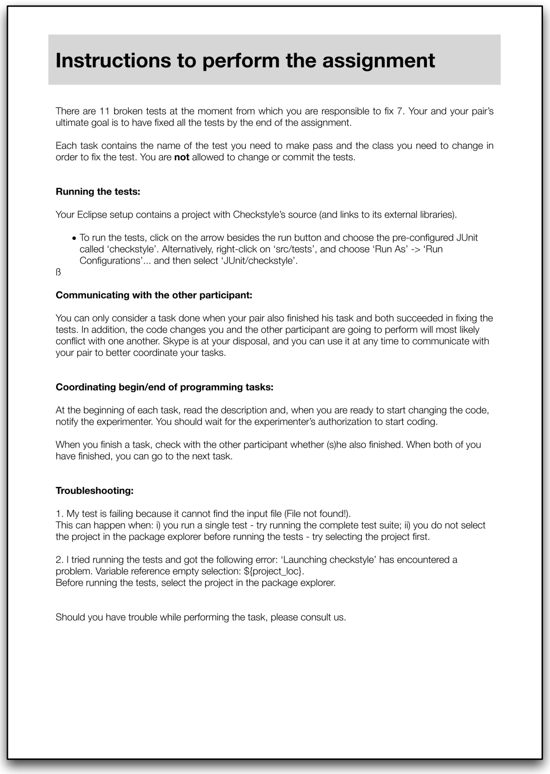
In short, Checkstyle takes as input a Java source file and an XML configuration file that specifies the coding standards that must be enforced (*i.e.*, the checks that are to be used). Most people are not familiar with Checkstyle's implementation. However, IDEs (such as Eclipse) may be able to assist in understanding Checkstyle's inner workings, and most of the source code is fairly well documented.

We kindly ask you to:
- perform the tasks in the specified order;
- write down the current time before starting to work on a new task and once after completing all tasks;
- *not return to earlier tasks* because it affects the experiment;
- notify the experimenter before starting to modify the code, and *wait* for her authorization.

The experiment begins with a questionnaire and ends with another questionnaire and a debriefing talk.

Thank you for participating in this experiment!

Lile Hattori, Michele Lanza and Marco D'Ambros



Instructions to perform the assignment

There are 11 broken tests at the moment from which you are responsible to fix 7. Your and your pair's ultimate goal is to have fixed all the tests by the end of the assignment.

Each task contains the name of the test you need to make pass and the class you need to change in order to fix the test. You are **not** allowed to change or commit the tests.

Running the tests:

Your Eclipse setup contains a project with Checkstyle's source (and links to its external libraries).

- To run the tests, click on the arrow besides the run button and choose the pre-configured JUnit called 'checkstyle'. Alternatively, right-click on 'src/tests', and choose 'Run As' -> 'Run Configurations'... and then select 'JUnit/checkstyle'.

ß

Communicating with the other participant:

You can only consider a task done when your pair also finished his task and both succeeded in fixing the tests. In addition, the code changes you and the other participant are going to perform will most likely conflict with one another. Skype is at your disposal, and you can use it at any time to communicate with your pair to better coordinate your tasks.

Coordinating begin/end of programming tasks:

At the beginning of each task, read the description and, when you are ready to start changing the code, notify the experimenter. You should wait for the experimenter's authorization to start coding.

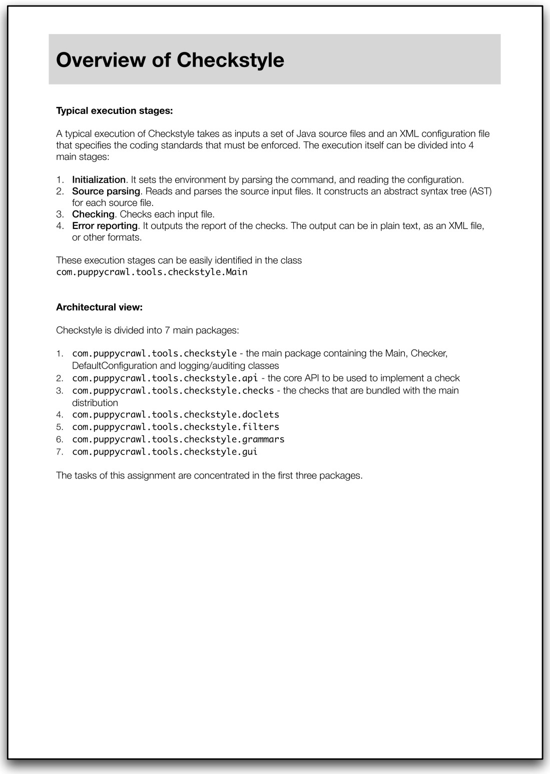
When you finish a task, check with the other participant whether (s)he also finished. When both of you have finished, you can go to the next task.

Troubleshooting:

1. My test is failing because it cannot find the input file (File not found!).
This can happen when: i) you run a single test - try running the complete test suite; ii) you do not select the project in the package explorer before running the tests - try selecting the project first.

2. I tried running the tests and got the following error: 'Launching checkstyle' has encountered a problem. Variable reference empty selection: ${project_loc}.
Before running the tests, select the project in the package explorer.

Should you have trouble while performing the task, please consult us.



Overview of Checkstyle

Typical execution stages:

A typical execution of Checkstyle takes as inputs a set of Java source files and an XML configuration file that specifies the coding standards that must be enforced. The execution itself can be divided into 4 main stages:

1. **Initialization**. It sets the environment by parsing the command, and reading the configuration.
2. **Source parsing**. Reads and parses the source input files. It constructs an abstract syntax tree (AST) for each source file.
3. **Checking**. Checks each input file.
4. **Error reporting**. It outputs the report of the checks. The output can be in plain text, as an XML file, or other formats.

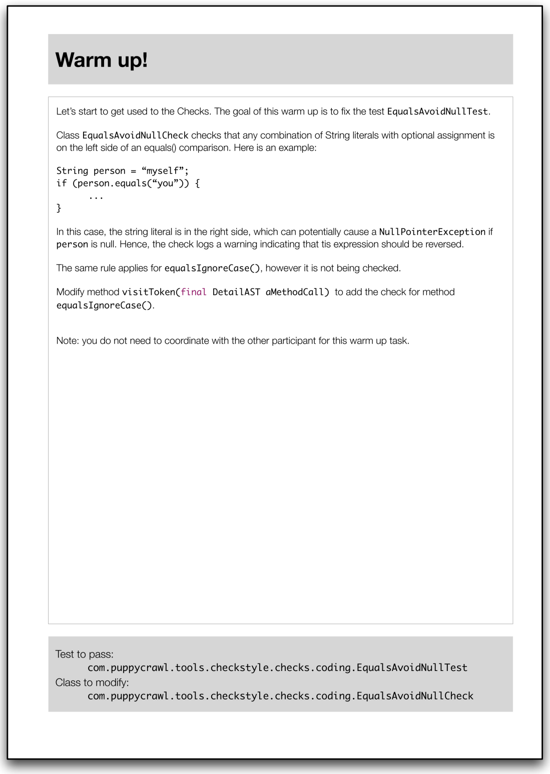
These execution stages can be easily identified in the class
`com.puppycrawl.tools.checkstyle.Main`

Architectural view:

Checkstyle is divided into 7 main packages:

1. `com.puppycrawl.tools.checkstyle` - the main package containing the Main, Checker, DefaultConfiguration and logging/auditing classes
2. `com.puppycrawl.tools.checkstyle.api` - the core API to be used to implement a check
3. `com.puppycrawl.tools.checkstyle.checks` - the checks that are bundled with the main distribution
4. `com.puppycrawl.tools.checkstyle.doclets`
5. `com.puppycrawl.tools.checkstyle.filters`
6. `com.puppycrawl.tools.checkstyle.grammars`
7. `com.puppycrawl.tools.checkstyle.gui`

The tasks of this assignment are concentrated in the first three packages.



Warm up!

Let's start to get used to the Checks. The goal of this warm up is to fix the test `EqualsAvoidNullTest`.

Class `EqualsAvoidNullCheck` checks that any combination of String literals with optional assignment is on the left side of an equals() comparison. Here is an example:

```
String person = "myself";
if (person.equals("you")) {
      ...
}
```

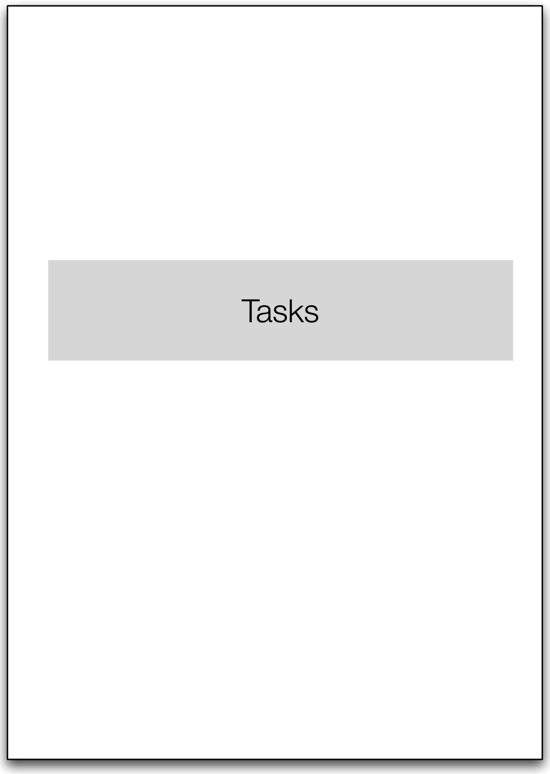
In this case, the string literal is in the right side, which can potentially cause a `NullPointerException` if `person` is null. Hence, the check logs a warning indicating that tis expression should be reversed.

The same rule applies for `equalsIgnoreCase()`, however it is not being checked.

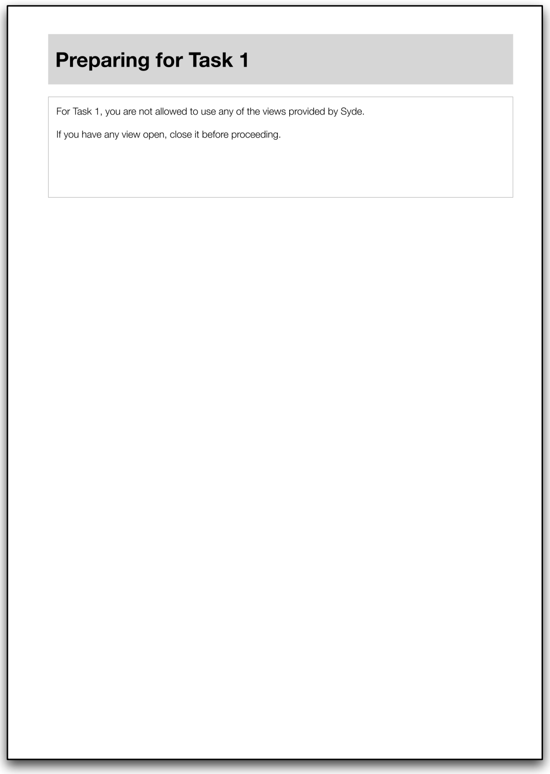
Modify method `visitToken(final DetailAST aMethodCall)` to add the check for method `equalsIgnoreCase()`.

Note: you do not need to coordinate with the other participant for this warm up task.

Test to pass:
    `com.puppycrawl.tools.checkstyle.checks.coding.EqualsAvoidNullTest`
Class to modify:
    `com.puppycrawl.tools.checkstyle.checks.coding.EqualsAvoidNullCheck`



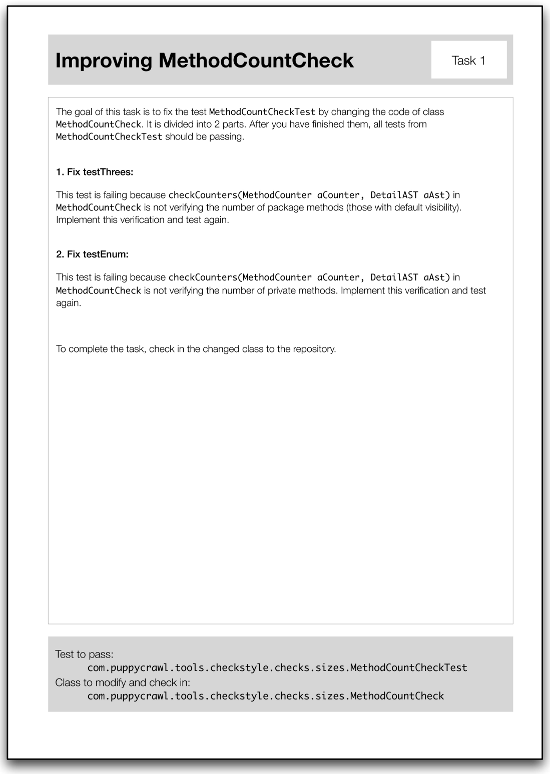
Tasks



Preparing for Task 1

For Task 1, you are not allowed to use any of the views provided by Syde.

If you have any view open, close it before proceeding.



Improving MethodCountCheck

Task 1

The goal of this task is to fix the test `MethodCountCheckTest` by changing the code of class `MethodCountCheck`. It is divided into 2 parts. After you have finished them, all tests from `MethodCountCheckTest` should be passing.

1. Fix testThrees:

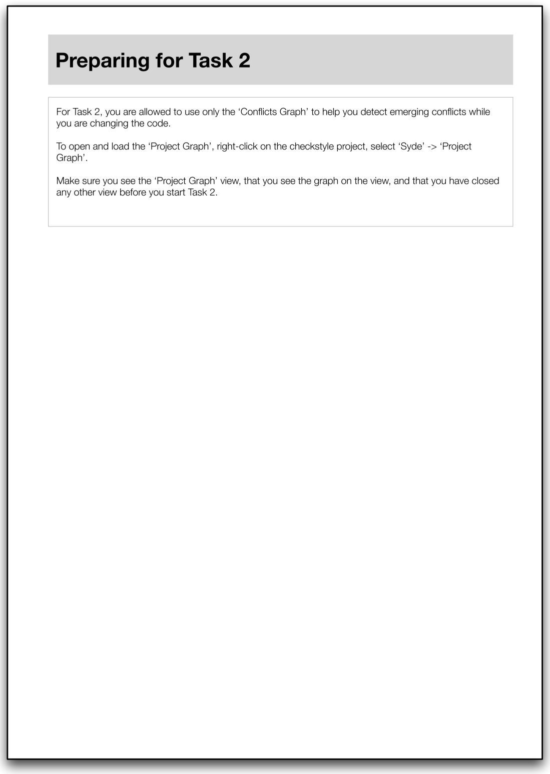
This test is failing because `checkCounters(MethodCounter aCounter, DetailAST aAst)` in `MethodCountCheck` is not verifying the number of package methods (those with default visibility). Implement this verification and test again.

2. Fix testEnum:

This test is failing because `checkCounters(MethodCounter aCounter, DetailAST aAst)` in `MethodCountCheck` is not verifying the number of private methods. Implement this verification and test again.

To complete the task, check in the changed class to the repository.

Test to pass:
      `com.puppycrawl.tools.checkstyle.checks.sizes.MethodCountCheckTest`
Class to modify and check in:
      `com.puppycrawl.tools.checkstyle.checks.sizes.MethodCountCheck`

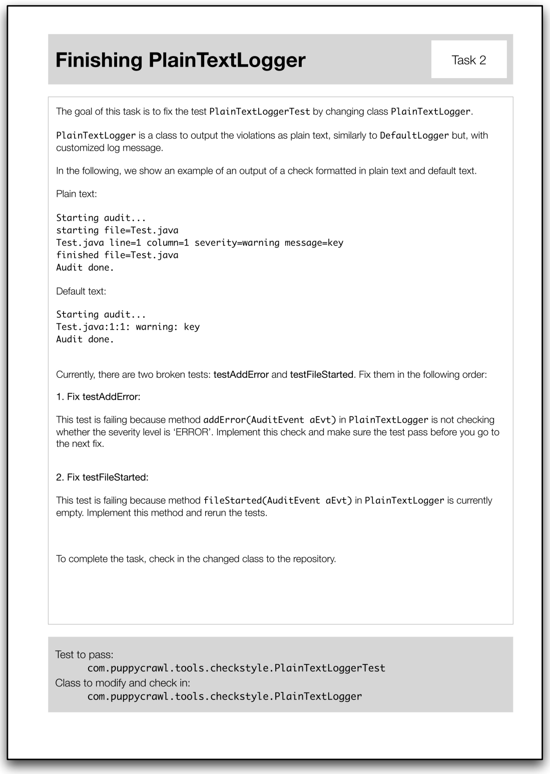


Preparing for Task 2

For Task 2, you are allowed to use only the 'Conflicts Graph' to help you detect emerging conflicts while you are changing the code.

To open and load the 'Project Graph', right-click on the checkstyle project, select 'Syde' -> 'Project Graph'.

Make sure you see the 'Project Graph' view, that you see the graph on the view, and that you have closed any other view before you start Task 2.



Finishing PlainTextLogger

Task 2

The goal of this task is to fix the test `PlainTextLoggerTest` by changing class `PlainTextLogger`.

`PlainTextLogger` is a class to output the violations as plain text, similarly to `DefaultLogger` but, with customized log message.

In the following, we show an example of an output of a check formatted in plain text and default text.

Plain text:

```
Starting audit...
starting file=Test.java
Test.java line=1 column=1 severity=warning message=key
finished file=Test.java
Audit done.
```

Default text:

```
Starting audit...
Test.java:1:1: warning: key
Audit done.
```

Currently, there are two broken tests: testAddError and testFileStarted. Fix them in the following order:

1. Fix testAddError:

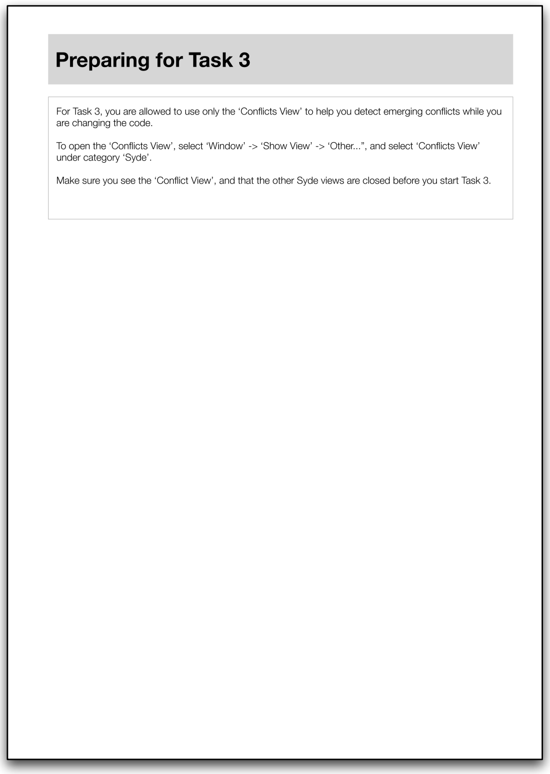
This test is failing because method `addError(AuditEvent aEvt)` in `PlainTextLogger` is not checking whether the severity level is 'ERROR'. Implement this check and make sure the test pass before you go to the next fix.

2. Fix testFileStarted:

This test is failing because method `fileStarted(AuditEvent aEvt)` in `PlainTextLogger` is currently empty. Implement this method and rerun the tests.

To complete the task, check in the changed class to the repository.

Test to pass:
`com.puppycrawl.tools.checkstyle.PlainTextLoggerTest`
Class to modify and check in:
`com.puppycrawl.tools.checkstyle.PlainTextLogger`

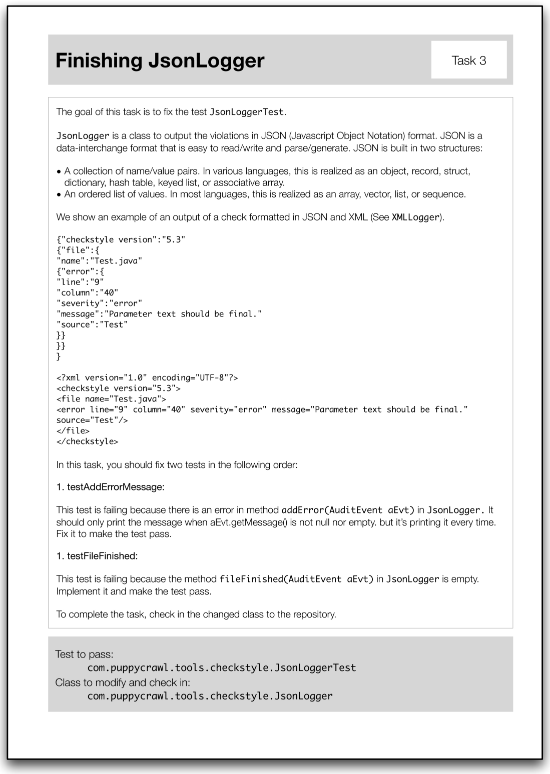


Preparing for Task 3

For Task 3, you are allowed to use only the 'Conflicts View' to help you detect emerging conflicts while you are changing the code.

To open the 'Conflicts View', select 'Window' -> 'Show View' -> 'Other...", and select 'Conflicts View' under category 'Syde'.

Make sure you see the 'Conflict View', and that the other Syde views are closed before you start Task 3.



Finishing JsonLogger

Task 3

The goal of this task is to fix the test `JsonLoggerTest`.

`JsonLogger` is a class to output the violations in JSON (Javascript Object Notation) format. JSON is a data-interchange format that is easy to read/write and parse/generate. JSON is built in two structures:

- A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

We show an example of an output of a check formatted in JSON and XML (See `XMLLogger`).

```
{"checkstyle version":"5.3"
{"file":{
"name":"Test.java"
{"error":{
"line":"9"
"column":"40"
"severity":"error"
"message":"Parameter text should be final."
"source":"Test"
}}
}}
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<checkstyle version="5.3">
<file name="Test.java">
<error line="9" column="40" severity="error" message="Parameter text should be final."
source="Test"/>
</file>
</checkstyle>
```

In this task, you should fix two tests in the following order:

1. testAddErrorMessage:

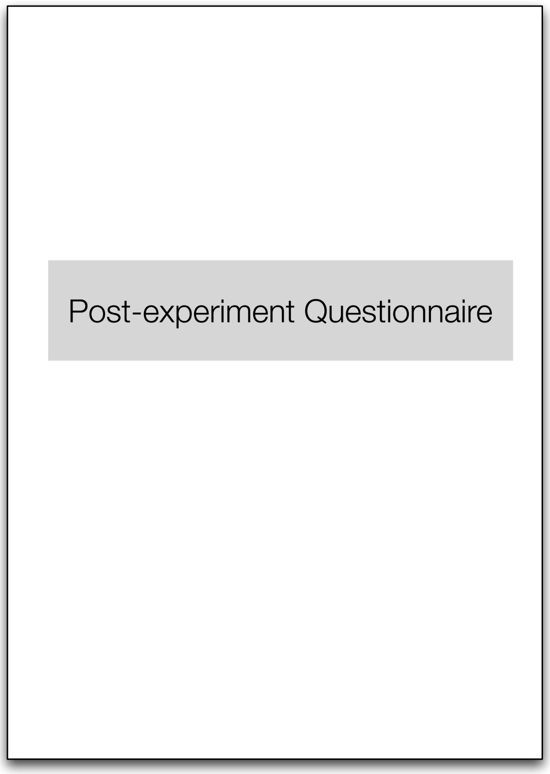
This test is failing because there is an error in method `addError(AuditEvent aEvt)` in `JsonLogger.` It should only print the message when aEvt.getMessage() is not null nor empty. but it's printing it every time. Fix it to make the test pass.

1. testFileFinished:

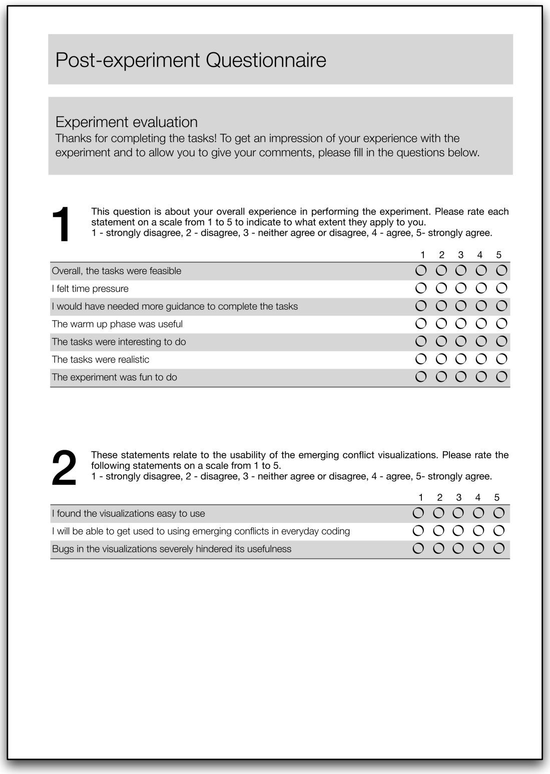
This test is failing because the method `fileFinished(AuditEvent aEvt)` in `JsonLogger` is empty. Implement it and make the test pass.

To complete the task, check in the changed class to the repository.

Test to pass:
`com.puppycrawl.tools.checkstyle.JsonLoggerTest`
Class to modify and check in:
`com.puppycrawl.tools.checkstyle.JsonLogger`



Post-experiment Questionnaire



Post-experiment Questionnaire

Experiment evaluation

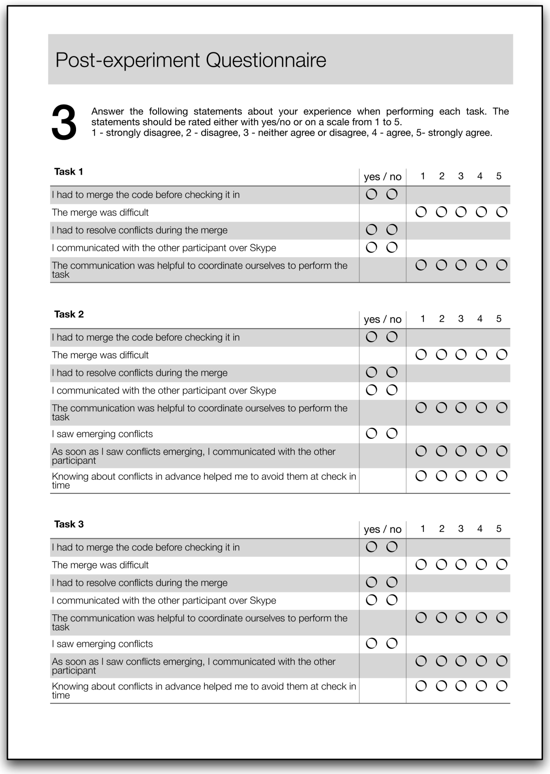
Thanks for completing the tasks! To get an impression of your experience with the experiment and to allow you to give your comments, please fill in the questions below.

1 This question is about your overall experience in performing the experiment. Please rate each statement on a scale from 1 to 5 to indicate to what extent they apply to you.
1 - strongly disagree, 2 - disagree, 3 - neither agree or disagree, 4 - agree, 5- strongly agree.

	1	2	3	4	5
Overall, the tasks were feasible	○	○	○	○	○
I felt time pressure	○	○	○	○	○
I would have needed more guidance to complete the tasks	○	○	○	○	○
The warm up phase was useful	○	○	○	○	○
The tasks were interesting to do	○	○	○	○	○
The tasks were realistic	○	○	○	○	○
The experiment was fun to do	○	○	○	○	○

2 These statements relate to the usability of the emerging conflict visualizations. Please rate the following statements on a scale from 1 to 5.
1 - strongly disagree, 2 - disagree, 3 - neither agree or disagree, 4 - agree, 5- strongly agree.

	1	2	3	4	5
I found the visualizations easy to use	○	○	○	○	○
I will be able to get used to using emerging conflicts in everyday coding	○	○	○	○	○
Bugs in the visualizations severely hindered its usefulness	○	○	○	○	○



Post-experiment Questionnaire

3 Answer the following statements about your experience when performing each task. The statements should be rated either with yes/no or on a scale from 1 to 5.
1 - strongly disagree, 2 - disagree, 3 - neither agree or disagree, 4 - agree, 5- strongly agree.

Task 1	yes / no	1	2	3	4	5
I had to merge the code before checking it in	○ ○					
The merge was difficult		○	○	○	○	○
I had to resolve conflicts during the merge	○ ○					
I communicated with the other participant over Skype	○ ○					
The communication was helpful to coordinate ourselves to perform the task		○	○	○	○	○

Task 2	yes / no	1	2	3	4	5
I had to merge the code before checking it in	○ ○					
The merge was difficult		○	○	○	○	○
I had to resolve conflicts during the merge	○ ○					
I communicated with the other participant over Skype	○ ○					
The communication was helpful to coordinate ourselves to perform the task		○	○	○	○	○
I saw emerging conflicts	○ ○					
As soon as I saw conflicts emerging, I communicated with the other participant		○	○	○	○	○
Knowing about conflicts in advance helped me to avoid them at check in time		○	○	○	○	○

Task 3	yes / no	1	2	3	4	5
I had to merge the code before checking it in	○ ○					
The merge was difficult		○	○	○	○	○
I had to resolve conflicts during the merge	○ ○					
I communicated with the other participant over Skype	○ ○					
The communication was helpful to coordinate ourselves to perform the task		○	○	○	○	○
I saw emerging conflicts	○ ○					
As soon as I saw conflicts emerging, I communicated with the other participant		○	○	○	○	○
Knowing about conflicts in advance helped me to avoid them at check in time		○	○	○	○	○

C Data from Questionnaires

For the sake of transparency and repeatability, we make available the participants' answers to the questionnaires.

Table 13, Table 14, and Table 15 contain the answers to the screening questionnaire, whereas Table 16 contains the answers to the debriefing questionnaire.

Table 13: First part of the answers to the screening questionnaire

					Experience Level							
Run	**Id**	**Age**	**Education background**	**Current position**	**Java**	**Team development**	**Dev. industrial size systems**	**Using IDEs**	**Using Eclipse for Java dev.**	**Using SCM**	**JUnit testing**	**Familiarity w. Checkstyle**
R1	P1	24	Computer science	Master student	advanced	advanced	beginner	advanced	expert	advanced	knowledgeable	none
R1	P2	26	Computer science	Master student	advanced	knowledgeable	beginner	advanced	advanced	knowledgeable	none	none
R2	P3	28	Computer science	PhD student	knowledgeable	knowledgeable	beginner	advanced	beginner	advanced	knowledgeable	beginner
R2	P4	30	Computer science	PhD student	beginner	knowledgeable	knowledgeable	knowledgeable	beginner	beginner	beginner	none
R3	P5	23	Computer science	Master student	expert	expert	beginner	expert	expert	expert	knowledgeable	none
R3	P6	22	Computer science	Master student	advanced	advanced	none	advanced	advanced	advanced	advanced	none
R4	P7	30	Computer science	PhD student	knowledgeable	advanced	knowledgeable	expert	expert	advanced	knowledgeable	none
R4	P8	38	Electrical engineering & Computer science	PhD student & Professor	expert	knowledgeable	knowledgeable	advanced	advanced	expert	knowledgeable	none
R5	P9	24	Computer science	PhD student	expert	knowledgeable	knowledgeable	advanced	advanced	expert	knowledgeable	none
R5	P10	24	Computer science	PhD student	expert	advanced	knowledgeable	advanced	advanced	advanced	advanced	knowledgeable
R6	P11	22	Computer science	Master student	expert	advanced	none	expert	advanced	expert	knowledgeable	none
R6	P12	24	Computer science	Master student	advanced	advanced	none	advanced	advanced	knowledgeable	knowledgeable	none

Table 14: Second part of the answers to the screening questionnaire

		Number of years of Experience							
Run	**Id**	**Java**	**Team development**	**Dev. industrial size systems**	**Using IDEs**	**Using Eclipse for Java dev.**	**Using SCM**	**JUnit testing**	**Familiarity w. Checkstyle**
R1	P1	5	5	0	5	5	4.5	5	0
R1	P2	6	-	0.2	6	6	2	0	0
R2	P3	5	1.5	1	2	1.5	3	2	0
R2	P4	2	2	2	2	0.5	0	0	0
R3	P5	5	3	0	3	4	4	3	0
R3	P6	4	4	0	4	4	4	4	0
R4	P7	5	5	2	5	3	4	2	0
R4	P8	7	4	3	7	5	3	4	0
R5	P9	5	5	1	5	5	5	5	0
R5	P10	7	4	2	6	6	6	7	1
R6	P11	5	6	0	5	5	6	3	0
R6	P12	4	1	0	4	4	1	1	0

Table 15: Third part of the answers to the screening questionnaire

Run	Id	SCM System currently used	Work in teams	Size of the team	Frequency of check out	Frequency of check in	Frequency of appearance conflicts
R1	P1	Git and a bit of SVN	yes	3-5 during UNI, 2-3 outside UNI projects	Various times per day during high activity; once per day during normal activity periods	Various times per day during high activity; once per day during normal activity periods	Not frequently, because the teams I work in have assigned roles/tasks
R1	P2	None	no		Before committing to check for conflicts	As soon as a have a function working, fixed a bug or executed an update	Almost every time
R2	P3	Store (VisualWorks) SVN (mainly for versioning latex files)	Most of the time NO	Maximum of 2 people	Every time (day) I start working on it	After new tests are created and run	Not very often, thanks to store granularity (method level)
R2	P4	Monticello, SVN	No		Once per day	Once per hour	Once per day
R3	P5	SVN	Often	2-4	More than once a day (when working in a team)	1-2 per day	Depends on the team, if well trained once a week, otherwise more than 3
R3	P6	SVN, Git	Often	2	Daily	Daily	Rarely
R4	P7	SVN	Yes	2 to 3	Weekly	Weekly	Once a month
R4	P8	CVS	Sometimes	around 3 or 4	Daily	Daily	It is not so frequent, it usually happens once every 3 months in the present project
R5	P9	SVN	Often	2	Once a week	Once a week	Once a month
R5	P10	Git, SVN, Rietveld	2 to 5 on average	Occasionally	Not often, since currently I am working on small (two people) or personal projects. So there are more check outs. I would say a month	Once every couple of days	Not common, again small sized teams nowadays
R6	P11	SVN	Often	3-5 people	Usually 3-4 times a day, it really depends on the frequency of updates for the project	It really depends on how much I am involved in the project, usually for projects in which I have a relevant role many times a day. I prefer to commit changes frequently to avoid conflicts.	Once a week
R6	P12	SVN	Often	5	Always before I begin working in order to avoid conflicts when I commit something new or modified.	Often, generally when I modify something that works at all.	Rarely. If it happens, it is generally due to my lack of checking out the repository before start working.

Table 16: Answers to the debriefing questionnaire

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
Experiment evaluation (1 - strongly disagree to 5 - strongly agree)												
Overall, the tasks were feasible	5	4	5	5	5	4	5	5	5	5	5	5
I felt time pressure	1	2	4	2	3	2	3	1	3	1	1	2
I would have needed more guidance to complete the tasks	1	1	1	1	2	2	2	1	1	4	1	1
The warm up phase was useful	2	5	3	5	3	4	4	4	5	5	4	4
The tasks were interesting to do	2	5	5	5	4	4	4	3	4	3	3	4
The tasks were realistic	5	4	5	5	5	4	4	3	4	2	4	3
The experiment was fun to do	5	5	5	3	5	5	5	3	5	5	5	5
Visualizations evaluation (1 - strongly disagree to 5 - strongly agree)												
I found the visualizations easy to use	2	3	4	4	5	4	4	4	4	3	5	4
I will be able to get used to using emerging conflicts in every-day coding	2	3	4	4	4	3	4	4	4	4	5	4
Bugs in the visualizations severely hindered its usefulness	1	3	2	1	3	3	4	2	2	2	1	4
Evaluation of each task (1 - strongly disagree to 5 - strongly agree; yes/no)												
Task 1												
I had to merge the code before checking it in	yes	no	yes	yes	no	yes	yes	no	no	yes	no	yes
The merge was difficult	4	-	3	3	-	3	1	-	2	1	-	2
I had to resolve conflicts during the merge	yes	no	yes	yes	no	yes	yes	no	yes	yes	no	yes
I communicated with the other participant over Skype	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
The communication was helpful to coordinate ourselves to perform the task	4	3	5	5	4	4	5	4	4	5	4	4
Task 2												
I had to merge the code before checking it in	yes	no	yes	no	yes	no	yes	yes	yes	yes	no	yes
The merge was difficult	4	-	3	2	3	-	1	2	2	1	-	2
I had to resolve conflicts during the merge	yes	no	yes	no	yes	no	yes	yes	yes	yes	no	yes
I communicated with the other participant over Skype	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
The communication was helpful to coordinate ourselves to perform the task	3	4	5	5	4	4	5	4	4	5	4	4
I saw emerging conflicts	yes	yes	yes	yes	yes	yes	yes	yes	yes	no	yes	yes
As soon as I saw conflicts emerging, I communicated with the other participant	4	3	5	4	1	3	4	4	5	1	5	5
Knowing about conflicts in advance helped me to avoid them at check in time	2	3	3	5	4	3	4	2	5	3	4	5
Task 3												
I had to merge the code before checking it in	no	yes	no	yes	yes	no	no	yes	yes	yes	no	no
The merge was difficult	-	4	-	3	3	-	-	2	2	1	-	-
I had to resolve conflicts during the merge	no	yes	no	yes	yes	no	no	yes	yes	yes	no	no
I communicated with the other participant over Skype	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
The communication was helpful to coordinate ourselves to perform the task	3	2	4	5	2	4	5	3	5	5	4	3
I saw emerging conflicts	no	yes	no	yes	yes	yes	yes	yes	yes	no	yes	yes
As soon as I saw conflicts emerging, I communicated with the other participant	3	3	-	5	2	4	4	4	5	1	5	5
Knowing about conflicts in advance helped me to avoid them at check in time	3	4	-	5	4	3	4	2	5	3	4	5