Empirical Evidence of the Benefits of Workspace Awareness in Software Configuration Management

Anita Sarma
Institute for Software Research
Carnegie Mellon University,
Pittsburgh, PA 15213
asarma@cmu.edu

David Redmiles and André van der Hoek
Department of Informatics
University of California, Irvine
Irvine, CA 92697-3440
{redmiles,andre }@ics.uci.edu

ABSTRACT
In this paper, we present results from our empirical evaluations of a workspace awareness tool that we designed and implemented to augment the functionality of software configuration management systems. Particularly, we performed two user experiments directed at understanding the effectiveness of a workspace awareness tool in improving coordination and reducing conflicts. In the first experiment, we evaluated the tool through *text-based assignments* to avoid interference from the well-documented impact of individual differences among participants, as these differences are known to lessen the observable effect of proposed tools or to lead to them having no observable effect at all. This strategy of evaluating an application in a domain that is known to have less individual differences is novel and in our case particularly helpful in providing baseline quantifiable results. Upon this baseline, we performed a second experiment, with *code-based assignments*, to validate that the tool's beneficial effects also occur in the case of programming. Together, our results provide quantitative evidence of the benefits of workspace awareness in software configuration management, as we demonstrate that it improves coordination and conflict resolution without inducing significant overhead in monitoring awareness cues.

Categories and Subject Descriptors
D.2.6 [**Software Engineering**]: Programming Environments – *Programmer workbench.* D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *version control.* D.2.9 [**Software Engineering**]: Management – *software configuration management*

General Terms
Management, Experimentation, Human Factors

Keywords
User experiments, evaluation, conflicts, parallel work, workspace awareness, software configuration management

1. INTRODUCTION
The concept of awareness, characterized as "an understanding of the activities of others to provide a context for one's own activities" [1], has been researched in the field of Computer-Supported Cooperative Work to facilitate coordination in group activities [2, 3]. Specifically, in being aware of the activities of team members, an individual can relate their own activities to those of their colleagues, enabling them to identify and address a variety of coordination problems [1, 4].

Recently, the software configuration management (SCM) community has recognized the potential of awareness, and there is a growing body of research that builds tools centered on awareness concepts to manage coordination in software development. SCM tools in particular are exploring the notion of *workspace awareness* (as it first emerged in groupware systems [5]) to support coordination across multiple developers working in parallel on the same code base [6-8]. The intention is for developers to be continuously informed of ongoing changes in other workspaces, as well as the anticipated effects of those changes, so they can detect potentially conflicting changes and respond proactively. Example responses may include contacting the other party for discussion, holding off on one's changes until another developer has checked in theirs, using the SCM system to look at another developer's workspace to determine the extent of a conflict, and other like-minded actions. Conflicting changes can thus be addressed before they become too severe. They may even be avoided altogether, when developers reconsider whether to edit an artifact that they know someone else is modifying at that time.

Numerous instances of observational case studies exist that articulate the presence and nature of coordination problems in software development and have guided the design and implementation of a host of different coordination tools [8-10]. Few resulting tools, however, have been empirically evaluated (exceptions include, e.g., Hipikat [11], Celine [12], O'Reilly's command console [13]). Of the tools that provide workspace awareness in software configuration management, in fact, just two provide any evidence of their benefits: FASTDash [14] and CollabVS [15]. FASTDash was evaluated through observations of actual use; CollabVS was evaluated in a laboratory experiment. Both evaluations provide initial, relatively coarse-grained evidence (see Table 1, Section 2).

This paper reports the results of an extensive empirical evaluation of Palantír [16], our own workspace awareness tool for SCM. Our results complement the results of FASTDash and CollabVS with a detailed and quantitative analysis that sheds light on *how* developers coordinate their parallel efforts, *when* they detect conflicts, *how* and *when* they resolve them, and *whether* there exists signifi-

cant overhead in using the overall approach of workspace awareness. We are able to achieve these very detailed results through a novel evaluation methodology, which uses a two-stage experiment to address individual differences in programming aptitude. By evaluating the tool with text assignments first and only then confirming the results with programming assignments, we are able to provide clearer and more precise evidence of how workspace awareness supports developers in detecting and resolving conflicts.

The first experiment was designed to evaluate Palantír through *cognitively neutral, text-based* assignments – non-coding assignments involving text that, to avoid bias, was neither too complex nor too interesting. Individual differences arising from variances in technical skills have been reported to drastically impact experiments of the kind we use here (when conducted in programming domain), to the point where either limited or no observable conclusions can be drawn from the data that is collected [17, 18]. To address this problem, this first experiment takes place in a domain where variance due to individual differences is minimal. This experiment evaluates Palantír's basic behavior as well as its user interface and how its design and the information it presents help the person involved in coordinating parallel work. We found that participants showed significant improvement in detecting and resolving conflicts when using Palantír, compared to without it. We further observed minimal overhead in monitoring awareness cues, but noticed clearly extra effort in resolution of indirect conflicts, extra effort that paid of with code checked in to the SCM repository that has fewer remaining inconsistencies.

The second experiment evaluated Palantír in the software domain by using *programming* (Java) assignments. The results were comparable to the text experiment: participants using Palantír showed significantly improved conflict detection and resolution rates over those without Palantír, monitoring awareness cues involved minimal overhead, and resolution of indirect conflicts required extra effort that paid of with code in the SCM repository that was free of indirect conflicts.

The results presented in this paper build upon results presented in a previous, short paper [19]. The previous paper reported some of the findings of the programming-oriented experiment. In this paper, new material includes the text-based experiment, additional findings and detail on the programming experiment, and the quantitative conclusions that we now can draw regarding the value of workspace awareness in SCM.

The rest of the paper is structured as follows. Section 2 presents background work on coordination in software development along with examples of existing SCM workspace awareness tools. Section 3 briefly describes Palantír, the awareness tool that we evaluated. It is followed by a description of our experimental setup and results in Section 4. Section 5 discusses the implications of our findings for coordination tools and their design in software development. We discuss the threats to validity for our experiment in Section 7 and conclude in Section 8.

2. BACKGROUND

A typical software development team consists of multiple developers who work together on closely related sets of common artifacts, a scenario that requires constant and complex coordination efforts. Particularly when change activities that involve multiple developers and interdependent artifacts are not carefully planned, conflicts are bound to occur. Even when the change activities are planned, however, it is well-known that conflicts occur, even with the use of sophisticated SCM systems [9, 20].

Conflicts occur in two cases: (1) when multiple developers concurrently edit the same artifact, and (2) when changes to one artifact affect concurrent changes to another artifact [9, 10]. In the first case, two developers edit the same artifact in separate workspaces, so their respective changes need to be combined to create a consistent version (merge tools help, but cannot always guarantee a semantically consistent and desired outcome [9, 21], as a result of which merging is still a bothersome and often manual process). We term this kind of conflict a *Direct Conflict*. As an example of the second case, it may happen that a developer working in his or her private workspace modifies a library interface that another developer just imported and started referring to as part of a change in his or her private workspace. This kind of conflict is usually more difficult to detect, as it tends to reveal itself at a later stage in the development process (e.g., as a build failure, test case failure, or, worse, bug after deployment). We term this kind of conflict an *Indirect Conflict*.

A number of factors contribute to why these kinds of conflicts occur and why they are difficult to deal with:

- *Software development is inherently multi-synchronous*. Developers check out artifacts from SCM repositories into their workspaces and thereafter essentially work in isolation, making changes to the artifacts in their own, private workspaces. Only after changes are complete do developers interact with the SCM repository to check in the artifacts that they modified. Between the time a developer checks out an artifact and the time they check it back in, they have no knowledge of the ongoing changes in other workspaces and how these changes relate to their own work (and vice versa) [8, 22].
- *Software involves intricate code dependencies, which evolve continually [3, 10]*. This means that any mental picture a developer has of the code's modularization and that may assist him or her in relating their own code changes to those of others, can become out of date and miss important elements.
- *Changes to artifacts are not instantaneous, but occur at the pace of human coding*. Between the time when a developer checks out an artifact and the time they check it back in, a significant window of time exists in which conflicts may be introduced and grow from small and innocuous at the beginning to large and complex as time passes and code changes continue to be made [10, 20].
- *Conflict resolution after the fact is a complicated activity*. In particular, once a conflict has been identified, a developer must go back in time, understand both conflicting changes in full, and find ways to meaningfully combine them. Evidence shows that this is not an easy task, and often will need to involve other team members to resolve issues that arise [16].

Various ethnographic studies have confirmed these observations and documented how developers have to work outside of the current coordination functionalities offered by SCM systems to address coordination problems that arise. Frequently, indeed, ad hoc coordination conventions emerge [8, 23, 24] For example, Grinter observed that developers in a software firm used the SCM repository to pace their development efforts to avoid having to resolve conflicts, specifically by periodically querying who checked out what artifacts [25]. If they thought a conflict might be imminent,

developers would try to complete their work before others, as the developer who checks in first would generally not be responsible for reconciling any future conflicts. It is the developers who checks in later who must integrate their changes with the current version in the repository [26]. As a second, related example, de Souza et al. found that developers frequently checked in incomplete changes to reduce the probability of having to resolve conflicts themselves [23]. As a final example, Perry et al. found that developers used Web posts to warn colleagues about changes that they were about to commit as well as their anticipated effects on other artifacts, so those developers who were editing or otherwise using those other artifacts were at least forewarned [9]. In all of these cases, we note that state-of-the-art SCM systems were in use and that the support provided by the SCM system was found critical and was used all the time. At the same time, however, these and other studies are highlight that modern SCM systems provide insufficient capability in enabling coordination styles that rely on a more direct and informal basis of communication.

Workspace awareness is a relatively new approach in the field of SCM, aiming to improve the coordination functionalities provided by SCM systems, primarily by overcoming the workspace isolation "enforced" by SCM systems [13, 16, 27]. Workspace awareness tools are based on the third observation above, namely that human coding takes time and that therefore conflicts emerge slowly. They particularly operate in the resulting window between check out of the original artifact and check in of the final modified artifact by transmitting information about ongoing changes across workspaces. The intended goal is to enable developers to build an understanding of which changes in which other workspaces might interfere or otherwise relate to their own. With this understanding, they can proactively coordinate their work with that of other developers, particularly if they note that a (direct or indirect) conflict is emerging. They may contact the other developer, use the SCM system to inspect an ongoing change in another workspace, abort their current change until the other person's work is done, or employ other such responses. The cost of these kinds of responses, since the conflict emerges slowly and is generally small in size when it is first detected, is anticipated to be much cheaper than when the conflict is fully developed and must be addressed later.

A number of design guidelines have emerged for the construction of workspace awareness tools for SCM: (1) provision of *relevant* information, (2) *timeliness* of when the information is shared, (3) *unobtrusive* presentation of the information, and (4) *peripherally* embedding awareness cues in existing development environments to avoid context switches [5, 28]. Following these guidelines, a number of different types of workspace awareness tools have been researched and built. Some tools provide basic information about the presence of direct conflicts stemming from concurrent changes to the same artifact (e.g., BSCW [29], Jazz [27], FASTDash [14]). Other tools provide additional information regarding direct conflicts, such as the nature and size of the conflict (e.g., Celine [12], State Treemap [30]). A final set of tools performs code analyses to identify potential indirect conflicts that arise because of dependent artifacts that are modified in parallel (e.g., TUKAN [31], Palantír [16], CollabVS [15]).

To the best of our knowledge only two of these workspace awareness tools have been empirically evaluated. FASTDash [14] is a workspace awareness tool that presents information of ongoing project activities and uses both a large wall display and personal visualizations to highlight concurrent edits to the same artifact. It was evaluated through observing the coordination patterns in an agile team, both before and after deployment of the tool. The authors found the developers to communicate more with the use of FASTDash and found a reduction in the amount of overlapping work. CollabVS [15] is close in functionality to Palantír and was evaluated through a user experiment. Surveys were used to determine how participants valued different features of CollabVS.

Complementing these two evaluations, this paper contributes detailed and quantitative evidence of the benefits of workspace awareness in SCM, both in case of direct and indirect conflicts. Through our novel evaluation methodology, we provide statistical evidence of an increased number of conflicts that are detected, an increased number of conflicts that are resolved, and a low overhead in monitoring awareness cues for both types of conflicts. We also find a correlation between the increased number of indirect conflicts detected and the need to communicate about these conflicts to resolve them. A summary of how our work enhances and details the previous results is provided in Table 1.

Table 1. Comparison of FASTDash, CollabVS, and Palantír.

Feature	FASTDash	CollabVS	Palantír
focus of the study	impact of awareness on a team's work practices (broadly)	impact of awareness on conflict resolution and associated coordination actions (specifically)	impact of awareness on conflict detection, resolution, and associated coordination actions (specifically)
study type	observational in actual development setting	laboratory experiment	comparative laboratory experiment
conflict type	any conflict detected from awareness of actual concurrent edits	one seeded conflict, involving both direct and indirect aspects	three seeded direct conflicts, three seeded indirect conflicts
method	survey, observation	survey, video recording	observation, video recording
team size	6	2	3
experiment sets	2(pre), 2 (post)	8	26 (text), 14 (Java)
result type	quantitative	qualitative	quantitative
granularity of results	coarse-grained	coarse-grained	fine-grained
observed results	increase in communication, reduction in overlap of work	improved ability to detect and resolve conflicts	improved ability to detect and resolve conflicts, minimal overhead in monitoring awareness cues, increased communication for resolving indirect conflicts

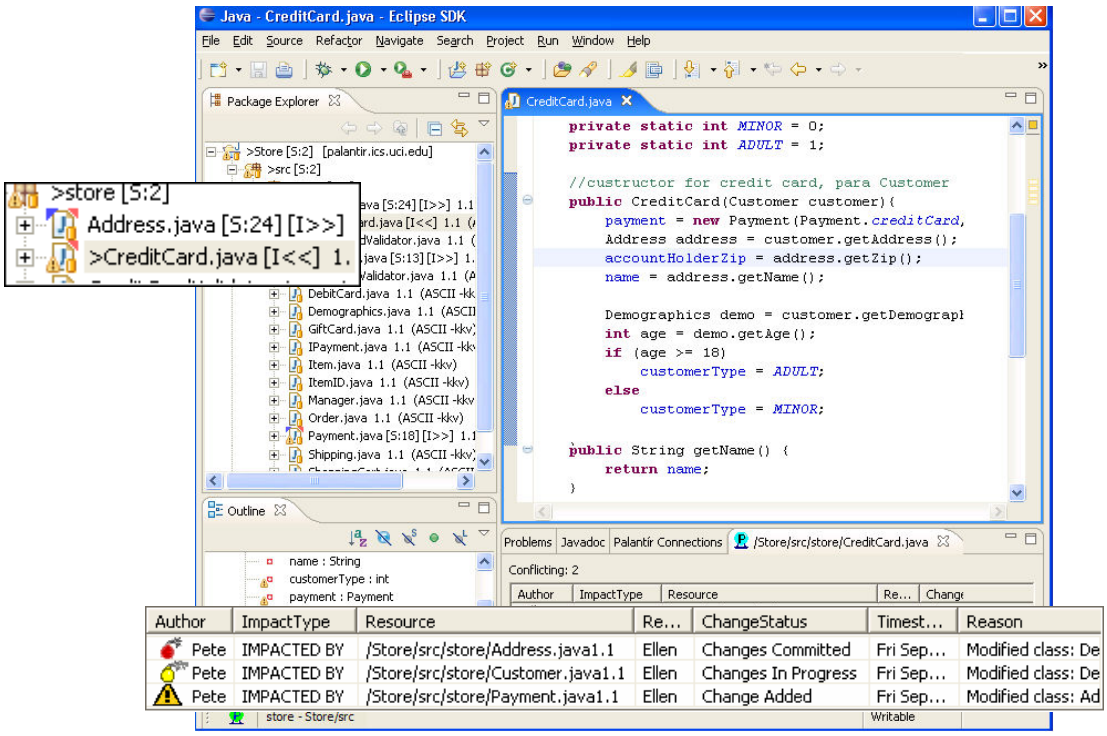

Figure 1. Palantír user interface.

3. PALANTÍR

We performed our experiments with Palantír, a workspace awareness tool for SCM that we have described elsewhere in detail [16, 22]. Here, to contextualize the following discussion, we briefly highlight the relevant functionality of Palantír.

Palantír informs developers of the two types of potential conflicts mentioned previously: (1) direct conflicts, which arise when the same artifact is concurrently modified in multiple workspaces, and (2) indirect conflicts, which arise when changes to one artifact in one workspace are incompatible with parallel changes to another artifact in another workspace. Unsurprisingly, a broad set of indirect conflicts exists, both syntactic and semantic in nature and of various degrees of difficulty to detect and handle [15, 16]. Out of these, Palantír currently addresses those indirect conflicts arising from changes to public methods and variables (see [16] for details on the kinds of conflicts supported by Palantír).

To provide workspace awareness, Palantír intercepts all edits that a developer performs in the local workspace as well as all of the configuration management operations that the developer issues. It translates these intercepted actions into a series of standard events that it subsequently shares with other workspaces for which these events are deemed relevant, that is, those workspaces in which the artifact(s) to which the events pertain is (are) checked out. Events that are received are communicated to a developer via awareness cues that are peripherally and visually embedded in Eclipse. These cues are designed to summarize what is happening in other workspaces and draw a developer's attention when it is appropriate to do so. This avoids presenting developers with too much information, which would result in unproductive distractions or an ignoring of the information altogether.

Figure 1 presents Palantír and its user interface as we evaluated it in the experiments reported in this paper. We integrated Palantír in the Eclipse development environment by making enhancements in two distinct places. *Annotations in the package explorer view* inform developers of activities in other workspaces (see top inset in Figure 1) and a new Eclipse view, *the impact view*, is available for developers to use to obtain further detail of changes that cause indirect conflicts (see bottom inset in Figure 1). Both extensions are briefly discussed in the following.

Palantír annotates resources in the package explorer view graphically and textually. Graphically, it uses small triangles to indicate parallel changes to artifacts. A blue triangle may appear in the top left corner of a resource (see *Address.java*). This triangle indicates the presence of ongoing parallel changes to that artifact, signifying that a direct conflict exists. A red triangle may appear in the top right corner of a resource (see both *Address.java* and *CreditCard.java*). It highlights the presence of an indirect conflict. For both the blue and red triangles, the larger the triangle appears, the larger the conflict that may be present. The typical pattern, then, is that a small triangle appears first, signifying the emergence of a conflict. Over time, this triangle may grow and shrink to reflect the current state of the changes. This pattern is what is important: by building an understanding of which patterns indicate conflicts that are to be considered seriously, developers are able to monitor at a glance how their work relates to and possibly interferes with that of others.

Textual annotations, to the right of a resource's filename, provide additional detail. For direct conflicts, it shows the size of a change in a remote workspace, as based on the relative lines of code that have changed. In the example, *Address.java* has been changed by 24%. Should multiple direct conflicts occur on the same resource, the percentages are added to indicate a more severe situation. The symbols *[I>>]* and *[I<<]* are used to indicate whether an artifact causes an indirect conflict or is affected by one, respectively (or both, if both *[I>>]* and *[I<<]* are present).

These extensions to the package explorer view are designed to be unobtrusive and not notably distract from the day-to-day work of a developer. They only provide the information necessary to draw the user's attention when needed. More information can then be obtained in the Palantír impact view, where various kinds of icons provide additional information about the state of an indirect conflict. For instance, the red "bomb" icon on *Address.java* indicates an indirect conflict with changes that are already committed to the SCM repository, whereas the yellow "bomb" on *Customer.java* indicates an indirect conflict with changes that are still ongoing in another workspace. *Payment.java* is marked with an exclamation mark, representing that it has undergone changes in another workspace that may be indicative of an indirect conflict, but cannot be proven to be so based on dependency analysis alone (for instance, the addition of multiple methods to a class may be reason for concern, but in and of itself is not a conflict until the addition of those methods starts resulting in changes in the rest of the class).

4. USER EXPERIMENTS

User evaluations of software tools have primarily been qualitative in nature. A key reason for this is that, in qualitative experiments involving software tools, individual differences among study participants often dominate the effect the tool is intended to have. As a consequence, statistically significant results cannot be achieved, because they would require inordinate numbers of participants to compensate for the dominating effect of individual differences [18, 32]. In our case, the ideal experiment of comparing participants with Palantír versus participants without Palantír on a team-based collaborative programming task would require an estimated 60 or more participants, and then still run a significant risk of not yielding statistically significant results in all of the variables [17].

The particular individual differences that concern our study are a programmer's technical skills [17] and the anticipated variance in how individuals in a team respond to their team mates' activities (the latter being both something we want to study and something we want to control for, as explained shortly). We explicitly designed our experiments to address these two individual differences.

With respect to individual differences in a programmer's technical skills, we benchmark the evaluation with non-programming tasks first, where variances stemming from individual differences are minimal [17], and then use these benchmarks to validate evaluation results from an analogous experiment with programming tasks.

In the first experiment (hereafter referred to as *text experiment*), we used text-based assignments that relied on a cognitively neutral text. Specifically, we tested a set of sample texts on a sample population, and chose from those a geology text after concluding that it was neither too complicated nor too interesting. That is, the text was of sufficient complexity to take time to work with, but not so complicated that it significantly differentiated the ability of sample participants to complete their given tasks. Similarly, it was sufficiently interesting for participants to stay engaged, but at the same time not too interesting (or familiar) to some subset of the participants, thus avoiding a bias resulting from overly eager performance by those who truly liked the subject of the text.

The text itself mimics some key properties of software, most specifically "modularity", as the text consists of several separate artifacts, and "dependencies", as the text contains references that link text across modules and must be kept consistent. This experiment, then, has participants perform a series of change tasks to the geology text to emulate software changes and in the process evaluates Palantír's basic behavior as well as its user interface. It particularly sheds light on how Palantír's design and the information it presents help the person involved in coordinating parallel work.

The second experiment (hereafter referred to as *Java experiment*) evaluated Palantír in the programming domain with an analogous study, but now involving participants making parallel changes to a shared code base. This experiment sought to confirm results from the first experiment, but takes into account the limitation of the programmer's individual differences becoming visible, especially in the time it takes for them to complete change tasks.

With respect to the second type of individual differences influencing the experiment results (the anticipated variance in how individuals in a team respond to team mates' activities), we note that the issue here is that we want to understand and draw conclusions about individual behaviors, but must do so in a team setting. Undesirable or wildly varying actions by one team member, however, may influence the conclusions we can draw regarding the behavior of other team members. To mitigate this risk, we designed both the text experiment and the Java experiment to use confederates, research personnel acting as virtual team members. This enabled us to precisely control and keep constant the change behavior of the "other team members", particularly in terms of when conflicts were introduced. Participants in the study were unaware of the set of tasks assigned to the confederates, the order in which the confederates would attempt their tasks, or even that the other participants were confederates (facts verified in a post experiment questionnaire). Participants, thus, believed they were in a genuine collaborative development setting.

The experiments were conducted at the University of California, Irvine. All experiment participants were students, at the graduate and undergraduate level, in the Donald Bren School of Information and Computer Sciences. Twenty-six participants participated in the text experiment and fourteen in the Java experiment. Participants volunteering for the experiment completed an online background survey documenting their experience in programming (including industry experience), using SCM tools, and using the Eclipse development environment, as well as providing additional demographic information. This information was used to carry out a stratified random assignment of participants [33]. Based on the spread of experience of the subject pool, participants with four or more years of experience in using SCM systems and Eclipse (for the Java experiment) or more than 1 year of such experience (for the text experiment) were assigned to stratum 1, while the remaining participants were assigned to stratum 2. Participants from each group were then randomly selected for treatment groups, that is, in the remainder of the paper all results are cumulative across strata but rely on comparisons within strata.

4.1 Experiment Setup

The goal of the experiments was to mimic team software development settings in which conflicts arise, and to observe how individuals note conflicts and take action to resolve them, both with and without the Palantír workspace awareness tool. The distributed nature of the activity allowed the experiment design to test one participant at a time, that is, because collaborating individuals each operate in their own workspace, we could simulate a team by

observing one participant as they interact with the other, virtual team members who are under our control. All such interaction took place through IM. Specifically, our experimental setup consisted of one participant collaboratively solving a given set of tasks in a three-person team in which the other two team members were confederates. These confederates were responsible for introducing a given number of conflicts with the participant's tasks at given times into the participant's tasks, so the timing and nature of the various conflicts remained constant across the participants.

Each experiment took about 90 minutes. Participants first completed a set of tutorial tasks to ensure that they could use the tool functionalities required in the experiment. The CONTROL group was given tutorials on Eclipse and CVS. The EXPERIMENT group was given tutorials on Palantír, Eclipse, and CVS. The tutorials were designed to ensure that participants were not biased to expect conflicts in the experiment, and merely focused on explaining the functionality of the various tools. Participants were then given the set of tasks to be completed. At the end of the session, participants were compensated $30 and they were briefly interviewed by the experimenter, who was present throughout the experiment as an observer. Screen capturing software was used to record all of the keyboard and mouse interactions as well as the screen content throughout each entire session for analysis.

We introduced two kinds of conflicts for each experiment: direct conflicts and indirect conflicts. These two are typical in software development, with direct conflicts representing conflicts that lead to merge problems and indirect conflicts representing conflicts that lead to build, integration, and test problems. We closely controlled when each type of conflict was introduced (generally ten to fifteen seconds after a participant began or completed a particular task). Tasks were presented to participants in the same order and the times when conflicts were introduced in the tasks were consistent across experiments. Our goal was to observe the effects of the tool on the way in which participants handled both different kinds of conflicts. Therefore, we treated the data for direct and indirect conflicts separately. We did not investigate any interaction effects with respect to the order in which conflicts were introduced. For instance, whether conflicts that are introduced later in the experiment are resolved faster is an interesting question, but a topic for future study.

Experiment Tasks: For the text-based experiment, the participant was given the role of the editor for a textbook on geology, as collaboratively written. Each chapter of the book was treated as a separate text file in the project, and the overall project consisted of thirty artifacts. Participants were given a set of *nine* tasks, six of which had conflicts: three direct and three indirect. Direct conflicts were introduced when a confederate changed the same file that the participant was editing. Such conflicts were introduced in *Tasks 2, 4,* and *8*. Indirect conflicts were introduced when a confederate changed an artifact that affected an artifact the participant was using and for which they were responsible. For example, the confederate deleted a chapter or changed a chapter heading without changing the *Table of Contents*. Such conflicts were introduced in *Tasks 3, 5, and 7*. The final task in the experiment (*Task 9*) required the participant to ensure the consistency of all chapters, particularly the *Table of Contents* and the *List of Figures*. The remaining tasks (*Tasks 1* and *6*) were benign (did not contain any conflicts).

For the Java experiment, participants were given a list of functionality to implement in an existing Java project. The project contained nineteen Java classes and approximately 500 lines of code. Participants were given a set of *six* tasks, four of which had conflicts: two direct and two indirect. Direct conflicts were introduced when a confederate modified the same Java file. Such conflicts were introduced in *Tasks 1* and *2*. Indirect conflicts were introduced when a method on which the participant's task depended was deleted or modified. These conflicts were introduced in *Tasks 4* and *6*. *Tasks 3* and *5* were benign tasks. Participants were provided with a Unified Modeling Language design diagram of the project to help them understand code dependencies. Unlike the text experiment, the Java experiment did not require participants in either group to integrate all the code at the end of the experiment. To be realistic, such would have additionally required an extensive set of build and test scripts, as well as the seeding of several indirect conflicts not caused by those scripts. This would have seriously complicated the experiment, and introduced several other potential design variables that we did not want to introduce.

Dependent variables: The primary variables of interest were the number of seeded conflicts that participants: (1) identified and (2) resolved. Different participant responses were grouped into four categories, namely conflicts that were (1) Detected and correctly Resolved [D:R]; (2) Not Detected by the participant until notified by the SCM system of a check-in (merge) problem, after which they were forced to Resolve it [ND:R]; (3) Detected by the participant but Not Resolved [D:NR]; and (4) Not Detected and Not Resolved by the participant [ND:NR]. Conflicts that were incorrectly resolved are treated here as Not Resolved.

We also measured the time that participants took to complete a task. Task completion times include the *time to implement* a task and, when applicable, the *time to coordinate* with team members and the *time to resolve* a conflict. When participants in the CONTROL group did not identify or resolve a conflict, we did not penalize them with extra minutes or "infinity" time. We chose not to do so since we wanted to investigate the overhead that is involved in using an awareness tool. Specifically, in the case of conflicts, a participant in the EXPERIMENT group who detected and resolved a conflict expended extra effort. Not including a penalty allowed us to precisely measure this extra effort as compared to participants in the CONTROL group.

Finally, we recorded the coordination actions that the participants performed to resolve a conflict, including SCM operations, chat conversation with confederates, and other miscellaneous actions.

5. EXPERIMENT RESULTS

We present and analyze our experiment results by addressing three questions. For each question, we first summarize the results, then motivate the question, and conclude with a more detailed discussion of the results for the text experiment and the Java experiment. This section concentrates on presenting raw results; implications are discussed in the next section.

1. Does workspace awareness help users in their ability to identify and resolve a larger number of conflicts?

Results: Table 2 shows the conflict detection and resolution observations for both the text and the Java experiment. Participants in the EXPERIMENT group (using Palantír) detected and resolved a larger number of conflicts for both conflict types (direct and indirect) and did so in both experiments. Further, in both experiments,

Table 2. Conflict detection and resolution data; text experiment concerns a total of 39 direct and 39 indirect conflicts (13 participants in each group, 3 seeded conflicts of each type per participant); Java experiment concerns a total of 14 direct and 14 indirect conflicts (7 participants in each group, 2 seeded conflicts of each type per participant).

	detect	EXP	CNTRL	Pearson χ^2	df	p*
Text DC	D:R	37	0	70.39	1	.001
	ND:R	2	39			
Text IC	D:R	31	2	58.20	2	.001
	D:NR	7	3			
	ND:NR	1	34			
Java DC	D	12	7	4.09	1	.04
	ND	2	7			
Java IC	D:R	14	0	28.00	1	.001
	ND:NR	0	14			

the results are found to be statistically significant ($p<.05$ for the χ^2 test; Fisher's exact test confirms the p values). Of note is that the results for direct conflicts for the Java experiment are categorized somewhat differently into Detected (D) versus Not Detected (ND) to address low expected cell counts in the χ^2 test (see below). Of the twelve conflicts detected by the EXPERIMENT group, nine were detected early and resolved and three were detected later, but not resolved (these were for the conflict that was introduced after the participant had already finished their task). Of the seven conflicts detected by the CONTROL group, all seven were detected during a check in (which resulted in a merge conflict) and resolved.

Discussion: Conflicts in software development that occur due to coordination problems frequently lead to a delay in project completion and/or an increase in defects in the code [9, 10]. We investigate the hypothesis that workspace awareness helps developers to identify and resolve potential conflicts while their changes are still in progress, which should lead to fewer delays and a reduction in defects. As a first step towards this goal, we compare differences between the treatment groups (CONTROL vs. EXPERIMENT) in their ability to identify and resolve seeded conflicts in the experiment.

Text Experiment: Participants in the EXPERIMENT group detected and resolved a much larger number of direct conflicts (DC) while they were still working on their tasks (row 1, Table 2). These conflicts were resolved either immediately upon noticing them or after the participant had finished his or her edits. We do note that, in two cases, participants ignored the notifications provided by the tool about a potential conflict. They continued working until their changes were complete and they attempted to check in their artifacts, subsequently facing a merge conflict[1].

The results of participants in the CONTROL group are significantly different. *None* of the participants detected a single conflict beforehand. This is not surprising as the SCM system shields them entirely from parallel work and they would have to continuously poll the SCM repository for updates and potential conflicts. Such a manual process is too cumbersome, as evidenced by some participants who indeed had an early practice of updating their workspaces before each next task, but discontinued this practice over time. Participants therefore discovered direct conflicts only when they attempted to check in the changes and the SCM system generated a merge conflict.

In the case of indirect conflicts (IC), we again find that a majority of participants in the EXPERIMENT group identified and resolved a much larger number of conflicts (row 2, Table 2). The difference here is more important than for direct conflicts, since in the case of direct conflicts, the conflicts were at least detected due to the merge conflict warnings from the available SCM system. In the case of indirect conflicts, however, participants in the CONTROL group identified only five indirect conflicts; the other thirty-four remained undetected and entered the SCM repository, even though participants were explicitly encouraged in the last step of the experiment to look for inconsistencies in the text. By comparison, participants in the EXPERIMENT group detected and resolved thirty-one conflicts early.

In both the EXPERIMENT and CONTROL group, several participants identified conflicts early, but could not resolve them. We attribute these situations to conditions in which the participants updated their workspaces, but could not correctly understand the dependencies among the artifacts. For instance, some participants could not detect when the confederate slightly modified the caption of a particular figure in a file, and that it affected the *List of Figures* file that they were supposed to update accordingly (recall participants had the role of "editor" of the document and were responsible for the table of contents and list of figures).

Java Experiment: For direct conflicts, the outcomes regarding detection and resolution rates resulted in low expected cell counts in the χ^2 test. These low counts can be attributed to two factors: (1) the experiment had a relatively small sample size (14) for a χ^2 test, and (2) one of the conflicts was seeded after the participant had already completed the task. With respect to this second point, we observed that even when participants in the EXPERIMENT group did notice the conflict, they did not go back to the task to resolve it (explaining the three conflicts detected later but not resolved, as reported in *Results* at the beginning of this section). Instead, they either made a note to themselves or informed their team members of a potential conflict and then continued on with their current task. This is an expected behavior in the way SCM systems implement conflict resolution. A developer who checks in first generally is not responsible for conflict resolution. It is the responsibility of any developer who next checks in their changes to ensure that those changes do not conflict with the version in the repository. For purposes of the experiment, this meant that, instead of the standard four-category breakdown we used otherwise, we had to group the results into Detected (D) versus Not Detected (ND).

We found that the EXPERIMENT group detected a larger number of direct conflicts (DC) early, differing significantly from the CONTROL group (row 3, Table 2). In the case of indirect conflicts (IC), we notice that *all* participants in the EXPERIMENT group identified and resolved conflicts, whereas *none* in the CONTROL group even detected a single conflict (row 4, Table 2). Particularly in the case of indirect conflicts, this is again critically important. Our results confirm the findings of the text experiment, demonstrating that incompatible changes entered the SCM repository unnoticed. One

[1] Participants were required to successfully commit their changes before they could move on to their next task

Table 3. Time-to-completion of tasks.

	group	minutes	sd	z	M-W U	p
Text DC	EXP CNTRL	9:12 12:30	2:14 1:43	-3.1	24	.001
Text IC	EXP CNTRL	7:57 6:30	1:55 1:14	-2.1	42.5	.03
Java DC	EXP CNTRL	8:57 7:09	2:44 0:48	-1.2	15	.26
Java IC	EXP CNTRL	9:09 5:33	3:59 1:14	-2.1	8	.04

can only hope that build or test failures quickly find these incompatible changes, though the literature suggests that they do so only to some degree [9, 34].

2. Does workspace awareness affect the time–to-completion for tasks with conflicts?

Results: Table 3 presents the average time-to-completion of tasks as organized per kind of conflict (DC and IC) and per experiment type (text and Java). The time-to-completion includes the time to detect, investigate, coordinate, and resolve a conflict, as applicable per task. We do not penalize participants who did not detect or resolve a conflict, choosing to simply report the time they took to complete the task (for reasons we explained in Section 4.1). In the text experiment, participants in the EXPERIMENT group took less time for direct conflicts, but longer for indirect conflicts (rows 1 and 2, Table 3). However, in the Java experiment, the EXPERIMENT group took more time for both conflict types (rows 3 and 4, Table 3). All results are statistically significant (Mann-Whitney test, $p<.05$), with the exception of direct conflicts in the Java experiment, where $p=0.26$.

Discussion: An obvious effect of workspace awareness tools is the fact that they incur some extra overhead "early", that is, a developer must spend time and effort to monitor the information that is provided to them and, if they suspect a conflict, spend time and effort to investigate and resolve it. We examine this overhead by comparing the average time that participants in each of the treatment groups took to complete tasks.

Text experiment: We found that participants in the EXPERIMENT group took less time (on average, three minutes shorter) to complete tasks with direct conflicts. However, we see a reverse trend for indirect conflicts (the EXPERIMENT group took a little longer). This difference can be explained because the SCM system forced participants in the CONTROL group to resolve each direct conflict during a check in, while no such forcing factor existed for indirect conflicts. This forcing factor resulted in participants in both the CONTROL and EXPERIMENT group to resolve the same number of direct conflicts. Because participants in the CONTROL group, however, detected these conflicts later, they incurred extra time and effort in facing a merge conflict and investigating it, leading to an overall longer time-to-completion. Participants in the EXPERIMENT group, on the other hand, coordinated with the confederate upon noticing a conflict was emerging, and rescheduled tasks or already took into account anticipated changes by the confederate in their own changes, thereby saving time as compared to the future problem that is now avoided.

As stated, in the case of indirect conflicts, no forcing factor exists, as a result of which the CONTROL group detected only a few conflicts. In contrast, the EXPERIMENT group detected and resolved the majority of the conflicts, causing them to incur extra coordination effort (primarily communications through instant messaging) in investigating conflicts and resolving them with the team mates (confederates). As a result, the average time per task was higher. The tradeoff, of course, is that the code delivered by the EXPERIMENT group had all of the conflicts resolved, which means that it would incur no further future effort to resolve these conflicts. Our experimental setup attempted to quantify this future effort by asking participants in the CONTROL group to examine the text after all change tasks were completed. Participants, however, could rarely find any inconsistencies. Therefore, no usable data was obtained regarding how much time and effort might have been saved.

Nonetheless, a critical observation arises: at the expense of extra effort, the quality of the text that was delivered was significantly higher because it included far fewer unaddressed conflicts.

Java experiment: The data for the Java experiment showed a larger variance in average time-to-completion. In case of direct conflicts, the groups did not differ significantly ($p=0.26$), even though it is interesting to note that – unlike in the text experiment – the EXPERIMENT group did take longer than the CONTROL group. In closely examining our data, we did not find any factors other than a probable cause of individual differences in programming skills outweighing any differences the use of Palantír made. Particularly given that both treatment groups detected and resolved about the same number of conflicts, seven versus nine (Table 2 and accompanying text), this factor seems to be the likely explanation.

In the case of indirect conflicts, however, we noted a pattern similar to the text experiment, with statistical significance ($p<.05$). In particular, the EXPERIMENT group took notably more time than the CONTROL group (row 4, Table 3), as they became aware of and had to resolve more conflicts. The extra effort in time, however, is again offset by the improved quality of the code that is delivered, as the final code contained zero indirect conflicts.

The Java experiment did not attempt to force the CONTROL group to reexamine the code base at the end of the experiment in order to quantify the time that may have been saved (as we attempted in the text experiment). The research literature, however, shows that conflict resolution at later stages is expensive and can take significant amounts of time (sometimes on the order of days). Any indirect conflict saved from entering the SCM repository, thus, constitutes one fewer and possibly major future concern [35, 36].

3. Does workspace awareness promote coordination?

Results: When participants detected a conflict, they generally took one of the following actions: synchronize, update, chat, skip the particular task, or implement the task by using a placeholder. Table 4 presents results about the specific coordination actions that participants undertook, summed per conflict type for both text and Java experiments. The table groups coordination actions into three categories: (1) *SCM* operations – update or synchronize; (2) *chat*, and (3) *other*s – skip or implement the task with placeholders. In general, we see a comparable number of coordination actions for direct conflicts, but a sharp increase in the number of coordination actions for indirect conflicts. No discernable shift in the types of

Table 4. Coordination actions.

	group	resolved	SCM	chat	others	total
Text DC	EXP	39	71	12	2	85
	CNTRL	39	78	17	0	95
Text IC	EXP	38	30	11	9	50
	CNTRL	5	7	4	0	11
Java DC	EXP	9	13	6	0	19
	CNTRL	7	12	3	3	18
Java IC	EXP	14	17	11	7	35
	CNTRL	0	2	1	3	6

coordination actions was seen. We found a statistically significant correlation (bivariate correlation, $p<0.01$) between the number of conflicts resolved and the number of coordination actions – both in the text experiment and the Java experiment.

Discussion: Coordination is a critical factor in team development, especially when conflicting changes are being made. Instituting an awareness solution is bound to influence how individuals coordinate their efforts, since knowledge of direct conflicts is available at an earlier point in time and additional information is provided regarding indirect conflicts. We examine this influence in terms of the number of coordination actions that participants undertake and in terms of the kinds of coordination actions they employ.

We observe that in both of our experiments, participants did not know the confederates (and so were not colleagues attempting to go to lunch or friends chatting) and they entirely focused on completing the task at hand. Therefore, they did not communicate with their team members unless required to do so by the task, making our data "clean" with respect to the phenomena we studied (i.e., in real-life situations, we can expect communication to also include personal conversation and/or coordination actions for other tasks and purposes).

Text experiment: Results for the text experiment were straightforward. Direct conflicts, whether detected through Palantír or via a warning from the SCM system upon check in, incurred about the same number of coordination actions in either case. The majority of these actions were SCM actions, synchronizing the workspace with the latest version in the repository. Some of the actions involved chat, often requesting what the other developer had done in their workspace and why.

For indirect conflicts, we observe a distinct spike in the number of coordination actions, both in terms of SCM actions and chat. This is no surprise, since participants in the EXPERIMENT group found more conflicts and thus needed to resolve more of them. This lead to both more SCM actions to bring changes from other developers in the workspace and integrate them, as well as more chat actions to briefly converse with the other developers about what they did (though this certainly was not done for every conflict).

Java experiment: Results were very similar to the text experiment. Direct conflicts resulted in about the same number and same type of coordination actions and indirect conflicts lead to a significant increase in coordination actions. In many ways, this is not surprising, and the results are in line with those presented in Question 2 of this section: extra time is spend, and some of that time is spend coordination one's actions with that of others. The tradeoff, however, is once again that a greater number of indirect conflicts are resolved before they enter the code base in the SCM repository.

Table 5. Time-to-resolution of DC.

	group	task_time	res_time	remainder
Text DC	EXP	9:12	2:23	6:49
	CNTRL	12:30	5:04	7:26
Java DC	EXP	8:57	4:00	4:57
	CNTRL	7:09	4:06	3:03
Java IC	EXP	9:09	7:01	2:08
	CNTRL	5:33	-	

6. BROADER IMPLICATIONS

Our findings have several broad implications, which are discussed in this section. First, our work presents an evaluation design that paves the way for future evaluations of software tools. It is well-known that individual differences tend to dominate the effects of a software tool, sometimes by exaggerating and other times by concealing the intended effects, thereby preventing useful empirical evaluations [18, 32], or at least makes it considerably more difficult by requiring large numbers of participants for disambiguation. To overcome this problem, we specifically conducted a user experiment with a cognitively "neutral" text first to benchmark results of a second user experiment involving a programming task. We believe such two-staged experiments can have promise in evaluations of other tools as well, as long as sufficient similarity can be achieved between the properties to be studied in the domain of interest (Java, in our case) and how those properties manifest themselves in the simulated domain (text, in our case).

Second, our results have implications for the design of software development tools and environments, especially those that rely on SCM systems for coordination. Even though our work evaluates a particular workspace awareness tool, Palantír, the results are much in line with other more coarse-grained evaluations to date. Therefore, we believe the lessons learned can be generalized to the class of SCM workspace awareness tools that use visualization to provide awareness of parallel development conflicts. Our experiments provide quantitative evidence that workspace awareness can significantly facilitate users in detecting and resolving both direct and indirect conflicts at a reasonable cost in terms of time and effort. These findings suggest that SCM tools should incorporate awareness features and that software development environments should provide facilities for external tools to easily and peripherally integrate awareness notifications.

Third, our work has implications for the design of workspace awareness tools in the domain of SCM. Our experiments provide quantitative proof of the need for automated detection of *indirect* conflicts. Even though we used simple, syntactic indirect conflicts in the Java experiment and additionally provided participants with a UML design diagram of the code structure and relationships, we found that participants without the tool had difficulty in identifying and resolving indirect conflicts. In fact, none of the participants in the CONTROL group detected or resolved a single indirect conflict, which all remained in the code that was checked in to the SCM repository. While a certain portion of those conflicts will be caught by build and test practices in real life, indirect conflicts can at the same time be of a much more complex nature in real devel-

opment efforts and have been shown to lead to serious problems [20, 37]. The automated identification of emerging indirect conflicts would be an invaluable tool.

7. THREATS TO VALIDITY

Generalizing results: As is the case with any controlled experiment, our experiment was performed in semi-realistic settings, where students served as experiment participants, worked with a relatively small project, and were asked to complete a given set of tasks in a limited time. We agree that the ideal way to test a software tool is a thorough longitudinal study set in a real-life software project, but at the same time observe that it is important to first evaluate the effectiveness of a tool in a controlled environment. Such an environment provides the opportunity, as we have shown, of drawing more detailed conclusions and controlling for individual differences (in our case for technical skills via stratified random assignment and for fluctuating team interactions via confederates). Because students were used as participants, our results are possibly conservative. Software developers may be more willing to invest time and effort up-front in order to avoid the difficulties they themselves have experienced in resolving conflicts.

Issues about scalability of the user interface were not tested in our experiment, since we used relatively small projects. We did not test the Palantír interface in a large project that comprised numerous artifacts, lots of parallel work, and therefore a possible proliferation of awareness cues. Although we used a small project, our experiment included benign tasks that produced extra awareness icons. In our exit interviews, participants responded that once they got used to them, the icons did not bother them. Further, they paid attention to an icon only if it concerned an artifact that the participant was editing or had previously edited. This highlights that the awareness cues used by the tool were unobtrusive, yet effective when they needed to be. Finally, we note that Palantír is explicitly constructed to reduce the number of notifications presented to the user by grouping them per artifact and providing additional filters on artifacts or developers.

Experiment design. Our experiment, specifically the Java experiment, did not force participants to carry out integration testing at the end of the experiment. Therefore, participants in the CONTROL group did not detect any indirect conflicts and thus did not spend any time in coordination attempts. This difference in resolution rates effectively penalized the EXPERIMENT group in their time-to-completion of tasks and precluded us from comparing the time-to-resolution data across the treatment groups (as we discussed). It, thus, may be possible that the extra time and effort that the developer is asked to invest is actually more costly than the resolution of full-blown conflicts at a later time. Given what we know from empirical studies of development projects and the role and impact of conflicts, however, this is highly unlikely [20, 37].

Another design choice that may represent a threat to validity is the introduction of a conflict ten to fifteen seconds after a participant began or completed a task. In real life, conflicts occur at any time. We used specific times to maintain consistency across the participants, and because of the limited time window of the experiment. Results may be different if conflicts arise closer to completion of a task, although we believe developers still would contact the other party, or be contacted by that other party.

A potential issue also exists with respect to the validity of drawing conclusions regarding the influence of awareness on programming tasks from the lessons learned from the text experiment. Working with text can be different than working with code, and participants may have an affinity with code that influences their behavior with respect to conflicts as compared to when dealing with conflicts in a text assignment. We structured the text assignments as much as possible to resemble the activity of coding, especially in terms of the conflicts we seeded into the experiment and the structure and relationships that the overarching text exhibited. Our results seem to indicate that the Java experiment confirms the lessons learned from the text experiment, building confidence that we should be able to rely on the text experiment to draw the conclusions we did.

Finally, all tasks were presented to participants in the same order. Although the lack of counterbalancing leads to learning effects, these effects are the same for all participants. The primary objective of our experiments was to investigate the effect of the two between-participant factors (treatment and strata) and not the effects of the within-subject factor, namely the kinds of conflicts and sequence, which are a subject for future study. Moreover, we observe that learning effects also will take place in real life and in fact are a desirable effect in awareness tools – users must calibrate the information that is provided in order to best leverage the tools. The short duration of our experiment undervalues this factor.

8. CONCLUSIONS AND FUTURE WORK

We have presented an empirical evaluation of the value of workspace awareness as a technique to enhance SCM systems and their ability to assist developers in detecting and resolving both direct and indirect conflicts. We used a novel experimental design consisting of a text-based experiment to obtain baseline data followed by a Java-based experiment that confirmed many of the findings we obtained in the text experiment. This allowed us to address the effects of individual differences in technical aptitudes, which traditionally is a serious hurdle in these kinds of experiments. In addition, our use of confederates allowed us to control for variances in how team members interact in multi-person settings.

Results from our experiments provide quantitative evidence that use of the workspace awareness tool improves the conflict detection and resolution capabilities of users. Further, while for indirect conflicts the time to completion of tasks increased, this time was well spent in removing these indirect conflicts from the code base that is inserted in the SCM repository. Finally, it appears that the overhead in monitoring the awareness cues is minimal – the extra time that participants in the EXPERIMENT group spent in tasks was primarily for conflict resolution, not monitoring. Our results have been obtained with relatively straightforward indirect conflicts. In future, we anticipate supporting more complicated indirect conflicts in Palantír and thereby gain even stronger benefits.

9. ACKNOWLEDGMENTS

Effort partially funded by the National Science Foundation under grant numbers IIS-0205724 and IIS-0534775. Effort also supported by an IBM Eclipse Innovation grant and an IBM Technology Fellowship.

10. REFERENCES

[1] Dourish, P. and V. Bellotti. 1992. Awareness and Coordination in Shared Workspaces. Computer-Supported Cooperative Work. p. 107-114.

[2] Heath, C., et al.,1994. Unpacking Collaboration: The Interactional Organisation of trading in a city dealing room. Computer Supported Cooperative Work 3(2): p. 147-165.
[3] Grinter, R.E., J.D. Herbsleb, and D.E. Perry. 1999. The Geography of Coordination: Dealing with Distance in R&D Work. Conference on Supporting Group Work. p. 306-315.
[4] Gutwin, C., R. Penner, and K. Schneider. 2004. Group Awareness in Distributed Software Development. Computer Supported Cooperative Work.p. 72-81.
[5] Gutwin, C. and S. Greenberg. 1996. Workspace Awareness for Groupware. Companion on Human Factors in Computing Systems. p. 208-209.
[6] Cataldo, M., et al. 2006. Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools. Computer Supported Cooperative Work. p. 353-362.
[7] Herbsleb, J.D. and R.E. Grinter,1999. Architectures, Coordination, and Distance: Conway's Law and Beyond. IEEE Software: p. 63-70.
[8] Grinter, R.E. 1996. Supporting Articulation Work Using Software Configuration Management Systems. Computer Supported Cooperative Work. p. 447-465.
[9] Perry, D.E., H.P. Siy, and L.G. Votta,2001. Parallel Changes in Large-Scale Software Development: An Observational Case Study. ACM Transactions on Software Engineering and Methodology, 10(3): p. 308-337.
[10] de Souza, C.R.B., et al. 2004. Sometimes You Need to See Through Walls - A Field Study of Application Programming Interfaces. Computer-Supported Cooperative Work. p. 63-71.
[11] Cubranic, D., et al.,2005. Hipikat: A Project Memory for Software Development. IEEE Transactions on Software Engineering, 31(6): p. 446-465.
[12] Estublier, J. and S. Garcia. 2005. Process Model and Awareness in SCM. Twelfth International Workshop on Software Configuration Management. p. 69-84.
[13] O'Reilly, C., D. Bustard, and P. Morrow. 2005. The War Room Command Console: Shared Visualizations for Inclusive Team Coordination. ACM symposium on Software visualization. p. 57-65.
[14] Biehl, J., et al. 2007. FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams. SIGCHI conference on Human Factors in computing systems p. 1313-1322.
[15] Dewan, P. and R. Hegde. 2007. Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development. European Computer Supported Cooperative Work. p. 159-178.
[16] Sarma, A., G. Bortis, and A. van der Hoek. 2007. Towards Supporting Awareness of Indirect Conflicts across Software Configuration Management Workspaces. Automated Software Engineering. p. 94-103.
[17] Mayer, R.E., 1988. From Novice to Expert, in *Handbook of Human-Computer Interaction*, M.G. Helander, T.K. Landauer, and P. Prabhu, Editors. p. 781-795.
[18] Redmiles, D.F. 1993. Reducing the Variability of Programmers' Performance through Explained Examples. Human Factors in Computing Systems. Amsterdam, p. 67-73
[19] Sarma, A., D. Redmiles, and A. van der Hoek. 2007. A Comprehensive Evaluation of Workspace Awareness in Software Configuration Management Systems. IEEE Symposium on Visual Languages and Human-Centric Computing. p. 23-26.
[20] Grinter, R.E. 1998. Recomposition: Putting It All Back Together Again. Computer supported cooperative work. p. 393-402.
[21] Mens, T.,2002. A State-of-the-Art Survey on Software Merging. IEEE Transactions on Software Engineering, 28(5): p. 449-462.
[22] Sarma, A., Z. Noroozi, and A. van der Hoek. 2003. Palantír: Raising Awareness among Configuration Management Workspaces. Twenty-fifth International Conference on Software Engineering. p. 444-454.
[23] de Souza, C.R.B., D. Redmiles, and P. Dourish. 2003. "Breaking the Code", Moving between Private and Public Work in Collaborative Software Development. International Conference on Supporting Group Work. p. 105-114.
[24] Olson, G.M. and J.S. Olson,2000. Distance Matters. Human-Computer Interaction, 15(2&3): p. 139-178.
[25] Grinter, R.E. 1995. Using a Configuration Management Tool to Coordinate Software Development. Conference on Organizational Computing Systems. p. 168-177.
[26] Berliner, B. 1990. CVS II: Parallelizing Software Development. USENIX Winter 1990 Tech. Conference. p. 341-352.
[27] Cheng, L.-T., et al. 2003. Jazzing up Eclipse with Collaborative Tools. Conference on Object-Oriented Programming, Systems, Languages, and Applications / Eclipse Technology Exchange Workshop. p. 102-103.
[28] Storey, M.-A., D. Cubranic, and D.M. German. 2005. On the Use of Visualization to Support Awareness of Human Activities in Software Development: A Survey and a Framework. ACM Symposium on Software Visualization. p. 193-202
[29] Appelt, W. 1999. WWW Based Collaboration with the BSCW System. Conference on Current Trends in Theory and Informatics. p. 66-78.
[30] Molli, P., H. Skaf-Molli, and C. Bouthier. 2001. State Treemap: an Awareness Widget for Multi-Synchronous Groupware. Seventh International Workshop on Groupware. p. 106-114.
[31] Schümmer, T. and J.M. Haake. 2001. Supporting Distributed Software Development by Modes of Collaboration. Seventh European Conference on Computer Supported Cooperative Work. p. 79-98.
[32] de Alwis, B., G.C. Murphy, and M. Robillard. 2007. A Comparative Study of Three Program Exploration Tools. International Conference on Program Comprehension. p. 103-112.
[33] Shadish, W.R., T.D. Cook, and D.T. Campbell, 2001. Experimental and Quasi-Experimental Designs for Generalized Causal Inference. 1 ed: Houghton Mifflin Company. pp.623.
[34] Brooks Jr., F.P,1974. The Mythical Man-Month. Datamation, 20(12): p. 44-52.
[35] Herbsleb, J.D., et al. 2000. Distance, dependencies, and delay in a global collaboration. Proceedings of the 2000 ACM conference on Computer supported cooperative work. p. 319-328.
[36] Curtis, B., H. Krasner., and N. Iscoe.1988. A Field Study of the Software Design Process for Large Systems. Communications of the ACM, 31(11): p. 1268-1287.
[37] de Souza, C.R.B., et al. 2004. How a good software practice thwarts collaboration: the multiple roles of APIs in software development. International Symposium on Foundations of Software Engineering. p. 22-230.