University of Nebraska - Lincoln
DigitalCommons@University of Nebraska - Lincoln

CSE Journal Articles | Computer Science and Engineering, Department of

8-1-2012

Palantir: Early Detection of Development Conflicts Arising from Parallel Code Changes

Anita Sarma
University of Nebraska at Lincoln, asarma@cse.unl.edu

D F. Redmiles
University of California - Irvine, redmiles@ics.uci.edu

Andre van der Hoek
University of California - Irvine, andre@ics.uci.edu

Follow this and additional works at: http://digitalcommons.unl.edu/csearticles

Part of the Computer Sciences Commons

Sarma, Anita; Redmiles, D F.; and van der Hoek, Andre, "Palantir: Early Detection of Development Conflicts Arising from Parallel Code Changes" (2012). *CSE Journal Articles.* Paper 104.
http://digitalcommons.unl.edu/csearticles/104

This Article is brought to you for free and open access by the Computer Science and Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in CSE Journal Articles by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

Palantír: Early Detection of Development Conflicts Arising from Parallel Code Changes

Anita Sarma, *Member*, *IEEE*, David F. Redmiles, *Member*, *IEEE*, and André van der Hoek, *Member*, *IEEE*

Abstract—The earlier a conflict is detected, the easier it is to resolve—this is the main precept of workspace awareness. Workspace awareness seeks to provide users with information of relevant ongoing parallel changes occurring in private workspaces, thereby enabling the early detection and resolution of potential conflicts. The key approach is to unobtrusively inform developers of potential conflicts arising because of concurrent changes to the same file and dependency violations in ongoing parallel work. This paper describes our research goals, approach, and implementation of workspace awareness through Palantír and includes a comprehensive evaluation involving two laboratory experiments. We present both quantitative and qualitative results from the experiments, which demonstrate that the use of Palantír, as compared to not using Palantír 1) leads to both earlier detection and earlier resolution of a larger number of conflicts, 2) leaves fewer conflicts unresolved in the code base that was ultimately checked in, and 3) involves reasonable overhead. Furthermore, we report on interesting changes in users' behavior, especially how conflict resolution strategies changed among Palantír users.

Index Terms—Software engineering, computer-supported collaborative work, programmer workbench, configuration management.

1 Introduction

A core function of any Software Configuration Management (SCM) system is to coordinate access to a common set of artifacts when multiple developers all must make modifications to the same code base at the same time [12], [43]. Virtually all SCM systems employ personal workspaces as the key technology to manage parallel changes [10], [12]. The reliance on personal workspaces, however, has historically harbored an inherent weakness in that it is possible for two changes that are made in parallel to conflict. That is, while each change may individually result in a correctly operating version of the code, their combination may not. Combining such changes will necessarily involve resolving which of the two to use and, consequently, reconciling this choice with the needs of the other change—a task that generally is not easy and may involve recoding and adjusting of solution strategies.

Over the years, a new class of coordination technologies called workspace awareness tools has emerged that attempts to identify conflicts arising from parallel work in personal workspaces so as to help developers mitigate the impact of these conflicts. The idea is to enable developers to become aware of conflicts earlier than is possible in current practice. Normally, developers become aware of a conflict only after each of the conflicting changes has been fully coded. Exactly when conflicts become apparent depends on the type of conflict. For instance, conflicts may become apparent when the second change is being checked in and its developer is informed of a merge conflict, when the combined code is being compiled and the compiler issues a build failure, or, for more subtle conflicts, when integration tests fail. In order to reveal conflicts earlier, while changes are still under development in personal workspaces, our approach informs developers of a potential conflict from the moment that two parallel changes begin to conflict. This enables developers to monitor the conflict and proactively resolve it before it becomes too unwieldy. The earlier developers know that their changes are conflicting, the earlier they will be able to address the situation and avoid doing unnecessary work.

We implemented Palantír, a workspace awareness tool that augments existing SCM systems. Palantír monitors all of the artifacts that developers modify in their personal workspaces and, for each of these artifacts, visually shares information regarding the state of its changes. This information helps developers identify when two artifacts are being concurrently modified or when two parallel changes cause a dependency violation. We use the term artifacts to refer to files that are under SCM control, with the reservation that our current implementation is limited to Java files only. We designed Palantír to be unobtrusive, yet catch the attention of developers at certain "natural" points of task or other context switching actions, e.g., when they open a new artifact to begin modifying it. Developers can thus concentrate on their day-to-day task of programming, but have the opportunity to contextualize their work with parallel changes that have been taking place and continue to take place.

Because our solution integrally relies on both the technological advances provided by Palantír and the human responses that developers make to the information that is provided to them, our evaluation focuses on this interplay.

- *A. Sarma is with the Department of Computer Science and Engineering, University of Nebraska-Lincoln, 256 Avery Hall, Lincoln, NE 68588-0115. E-mail: asarma@cse.unl.edu.*
- *D.F. Redmiles and A. van der Hoek are with the Department of Informatics, Donald Bren School of Information and Computer Sciences, University of California Irvine, 5029 Donald Bren Hall, Irvine, CA 92697-3440. E-mail: {redmiles, andre}@ics.uci.edu.*

Manuscript received 29 Dec. 2010; revised 25 Apr. 2011; accepted 7 May 2011; published online 16 June 2011.
Recommended for acceptance by H. Gall.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2010-12-0383.
Digital Object Identifier no. 10.1109/TSE.2011.64.

0098-5589/12/$31.00 © 2012 IEEE Published by the IEEE Computer Society

Through two laboratory experiments, we examine whether and how developers respond to the awareness cues of Palantír. We found that the use of Palantír, as compared to not using Palantír: 1) leads to both earlier detection and earlier resolution of a larger number of conflicts, 2) leaves fewer conflicts unresolved in the code base that was ultimately checked-in, and 3) involves acceptable overhead. We also report qualitatively on the behaviors Palantír users exhibited, particularly regarding how they monitored the awareness cues, how they interpreted them, and what they subsequently did in response.

While our previous conference publications reported on aspects of this work, this paper combines all aspects to provide a concise and complete overview of our work. We also present an extensive, new, qualitative analysis of our experimental data focusing on interviews, chat logs, and observations to provide a detailed account of why Palantír works the way it does, as well as how users reacted to the information provided to them. A discussion of the lessons learned from our research on workspace awareness has also been added, which has implications for the design of future workspace awareness technology.

The rest of this paper is organized as follows: We discuss background material on awareness and conflicts in Sections 2 and 3. Then, we present our approach in Section 4. We briefly present the implementation features of Palantír in Section 5. We discuss the experimental setup and our evaluation in Section 6. In Section 7, we present the threats to validity to our experimental design. We discuss lessons learned in Section 8, followed by related work in Section 9. We conclude with an outlook for our future work in Section 10.

2 Awareness

Working in a distributed team requires an in-depth understanding of the context of one's changes and how they fit with others' tasks. Developers have responded to this need, often by establishing informal conventions to coordinate their work. They may send an e-mail informing other developers of the changes that they have made and the possible effects of these changes [13]. They may also use Instant Messaging (IM) to coordinate their work [29], periodically visit the bug database to informally assess who is working on which tasks and thereby may be changing which artifacts [32], or simply walk over to a colleague's office asking them about their current tasks [23].

These kinds of ad hoc actions aim to improve awareness of one another's activities. The notion of awareness has gathered considerable attention in the Computer-Supported Cooperative Work (CSCW) literature, where it was first characterized as "an understanding of the activities of others, which provides a context for your own activity" [18]. Early work focused on the social behavior of mutual awareness, involving participants who explicitly display their actions and monitor each others' conduct to inform their own behavior [28]. Another form of awareness arises when individuals meet by accident at, for instance, the water cooler, allowing them to exchange relevant information impromptu. Both these forms of awareness rely on collocated or *in situ* settings.

Recognizing that geographically distributed teams cannot rely on *in situ* awareness [27], [30], researchers pursued the development of tools to provide awareness in distributed settings. This line of work is commonly referred to as "workspace awareness" and is defined as the "up-to-the-moment understanding of another person's interaction with a shared workspace" [26], [27]. Workspace awareness involves an understanding about where others are working, what they are doing, and what they are going to do next. Tools for workspace awareness then actively collect information that they share with those individuals for whom (portions of) this information is relevant. Some tools focus on presence information (e.g., porthole systems such as [34]), some share fine-grained information at the level of keystrokes through shared interfaces (e.g., Suite [16], ShrEdit [36], Google Wave [2]), and others present coarser grained information on project level activities (e.g., FASTDash [6] and CollabVS [17]). The overall goal of all these tools is the same: to enable individuals to monitor their colleagues' actions and prod them into assessing if their respective activities could be better adjusted to one another and, if so, proactively engage in self-coordination [40].

The work presented in this paper specifically focuses on improving awareness of potentially conflicting parallel changes with the goal of reducing merge and integration problems arising from parallel work in personal workspaces.

3 The Nature of Conflicts

Many different kinds of conflicts can arise in parallel work when that work is performed in personal workspaces. We first distinguish direct conflicts from indirect conflicts. Direct conflicts involve incompatible, parallel changes to the same artifact. Indirect conflicts involve incompatible, parallel changes across artifacts. This partition simply divides conflicts depending on the locality of the changes, that is, whether a conflict is confined to a single artifact or involves multiple, interacting artifacts.

A second, orthogonal dimension is the degree of difficulty in resolving a conflict. This scale is difficult to quantify as it involves such factors as syntax versus semantics, a developer's personal experience, available tools and test cases, and so on. Intuitively, a conflict involving incompatible syntactical changes is more easily found and addressed than a conflict involving two conflicting changes that introduces an intricate timing problem in a multithreaded system. In the first case, a build failure will likely point a developer in the right direction fairly quickly. In the second case, one has to hope that the available test cases reveal the problem since otherwise it might not be found until well after the product has been deployed in the field.

Our interest is in conflicts whose emerging presence might be detectable using (semi-)automated analyses. Currently, conflicts are generally not detected until after both changes are completed, when the changes are checked-in, merged, built, and tested. To detect these conflicts earlier, relevant information from respective pairs of workspaces must be brought together for analysis. This information will necessarily be partial in that it represents changes in progress that have not been completed as of yet. It also is partial because it is virtually impossible to share entire changes across workspaces in the case of large development efforts, due to issues of scale. Choices must therefore be

made as to which information is shared, for which type of analysis, and for which kind of conflict.

A principal design choice that we use is to notify developers of conflicts at the level of entire artifacts. For direct conflicts, this means that the tools do not differentiate among different degrees of difficulty in resolving a conflict: Any concurrent modifications to the same artifact will be marked. While this is conservative with respect to whether or not an actual conflict is present since the respective changes could well be compatible, it highlights those situations where developers are expected to closely look at the result of merged changes since automated merge algorithms cannot always be trusted [4], [37].

For indirect conflicts, a choice still must be made as to what kind of analysis, or kinds of analyses, to perform. We focus on indirect conflicts arising from mismatches in modifications to, and use of, class signatures (i.e., public class variables and methods). While this choice represents a relatively small percentage of all indirect conflicts and also addresses indirect conflicts for which an analysis is readily available for adaptation, our work represents an initial foray in cross-workspace analysis for an early conflict detection. We realize that addressing increasingly difficult indirect conflicts will bring with it its own challenges, especially when it concerns semantic conflicts.

4 Approach

Our approach builds on two key insights: 1) Conflicts take time to develop—the time during which developers make code changes in their personal workspaces, and 2) developers will want to self-coordinate their activities to avoid or mitigate conflicts.

Any code change involves time spent thinking, typing, compiling, testing, correcting mistakes, and refining the modifications one is making. Consequently, conflicts do not appear suddenly in full, but emerge slowly as the result of parallel coding efforts of two or more developers. At first, a conflict will be a result of a small action, perhaps involving single line modifications to the same file, or a single method signature that has changed while another developer just happens to introduce a call to that method. But, over time a conflict typically grows with additional changes that developers make. Single line modifications may aggregate into significant edits, and a single changed method signature may turn into a significant reorganization of the entire public interface of a class.

To take advantage of the "window of time" during which conflicts emerge, our approach informs developers of ongoing parallel changes in other workspaces. The hypothesis is that, on being informed of conflicts, developers will take proactive measures to resolve those conflicts early while they are still small in size. The benefit would be twofold: 1) A reduction in the amount of effort that has already been expended by the time a conflict is detected, lowering the amount of work that must potentially be redone, and 2) a reduction in the amount of effort involved in resolving conflicts since the changes are still small at this time.

The effort expended, naturally, lies in the overhead of monitoring the awareness information provided by Palantír and, if deemed necessary, subsequently taking coordinative actions. Our approach therefore considers developers as an integral part of the solution. Palantír intentionally prods developers into a behavior which, rather than continuing to code and ignoring a conflict, involves them taking proactive steps to assess and, if necessary, resolve an emerging conflict of which they are notified. This might involve telephoning or using IM to contact the other developer, walking over to discuss and redistribute the respective tasks, making a unilateral decision to hold off on one's changes until the other developer has checked in theirs, using the functionality of the SCM system to examine the changes in the other workspace [24], and other courses of action. For such actions to be taken, a developer has to invest some extra effort in monitoring and interpreting the awareness information that Palantír provides. As we will show in Section 6, this extra effort is relatively minor.

Palantír operates by monitoring ongoing changes taking place in personal workspaces and continuously sharing information about those changes with developers to whom it is relevant. By default, this information communicates which other developers are modifying which artifacts, and by how much. As such, Palantír provides an automated and enhanced version of soft locks [38]: A developer can engage in parallel changes to the same artifact, but does so consciously, at all times knowing the amount of change that an artifact has undergone in another workspace. This amount of change is a critical piece of information. If it is small, a developer may engage in a conflict without even contacting the other developer. If it is larger, they may well want to first contact the other developer or wait to make further changes.

Palantír is bidirectional. When a conflict arises, both developers are informed of each other's changes. While one may choose to ignore the warnings, the other developer may consider the conflict too important to ignore and engage with the first developer anyway, change their course of action altogether by temporarily suspending the current task, or use any of the other means described in Section 6.

The above describes how Palantír supports direct conflicts. In the case of indirect conflicts, an additional step is required. Specifically, Palantír performs a cross-workspace analysis of the impact of changes in a class signature on the use of that class: A change to a class signature (i.e., a change to a class' public variables and methods) in one workspace is compared to changes in the use of that class in other workspaces. To perform this analysis, Palantír broadcasts a "diff" detailing the changes to a class signature whenever this signature is modified. The other workspaces interpret this diff and analyze whether its contents leads to an indirect conflict with any of the local changes. If so, Palantír informs all workspaces of this conflict.

It is a conscious choice to inform developers of ongoing changes and not just of changes that are guaranteed to conflict. First, it would be irritating to be under the impression that no one is working on an artifact, open the artifact, begin to make changes, and only then be informed that someone has been working on this artifact for quite some time and has, perhaps, made significant changes. Second, knowing which artifacts are actively undergoing change, and what other artifacts they indirectly impact because of those changes, allows one to plan their tasks around the ongoing changes, rather than engage in a task

blindly and hope no conflicts will occur. Finally, as we shall discuss in Section 6, we do not believe that the extra information deters from the task at hand of programming. The user interface of Palantír is designed so developers notice relevant information at relevant times, particularly when they switch from artifact to artifact or task to task. This enables a developer to work on their code undisturbed, yet provides them with ample opportunity to absorb remote changes when it is pertinent to do so.

Two additional design decisions govern the approach underlying Palantír. The first is that a developer is continuously informed of ongoing changes (it broadcasts relevant information every 5 seconds, or when a file is (auto) saved), but leaves the decision as to whether or not a conflict must be addressed to the developer. Not every conflict needs to be addressed immediately upon detection. Deciding when a conflict merits immediate attention, as well as what specific actions may be needed, depends on how the conflict develops over time and the criticality or urgency of the parallel changes that cause the conflict.

The second decision is that a developer is informed of conflicts at the level of entire artifacts. While information detailing precisely where in the code an individual conflict appears is important (and indeed can be obtained from Palantír), the more practical concern pertains to being able to quickly assess in what ways one's overall work interferes with that of another developer because this determines when a response is warranted. The more conflicts exist, the more interference exists between workspaces, and it is less likely that the respective changes can be easily integrated. Since resolution strategies are also normally expressed in terms of artifacts (i.e., checking out artifacts, reverting changes to an artifact, merging two versions of an artifact), Palantír reports conflicts at the level of artifacts rather than individual methods or lines of code.

Overall, our approach allows parallel work while making it possible to detect and resolve conflicts before all changes are complete and checked in. An interesting outflow of this approach is that, until conflicting changes are actually checked in, it is technically more appropriate to label them as "potential conflicts." It is possible that an emerging conflict grows for some time, but all of a sudden disappears because a developer who was just experimenting decides to roll back their changes (Palantír tracks such rollbacks and modifies the awareness icons accordingly). By the same token, a conflict does not always grow; it can shrink when a developer removes some of the conflicting code. One is therefore not guaranteed that a conflict identified by Palantír actually will result in a real conflict in the future or that it will be at the level it is currently at. Even so, it is a simple step to contact another developer to learn of their intentions, thereby determining whether a potential conflict should be treated as a real conflict to be addressed immediately or as one to be ignored for now. In the remainder of this paper, we will not make a distinction between conflicts and potential conflicts, using the term conflicts for reasons of simplicity.

5 Implementation

Palantír is written entirely in Java and is currently available as a plug-in for Eclipse and supports both Subversion [54] and CVS [20]. It relies on: 1) Subclipse [53], an Eclipse plug-in for the Subversion SCM system, or the built-in CVS plug-in for Eclipse, from which Palantír obtains relevant information regarding SCM actions through event listeners, 2) Dependency Finder [51], an open source code analysis tool that Palantír uses to incrementally extract information regarding differences in the dependency graph of the code being developed, and 3) PostgreSQL [41], a generic database in which Palantír stores all of the events. Palantír, its source, and its documentation are all freely available from its website: http://tps.ics.uci.edu/svn/projects/palantir/trunk/.

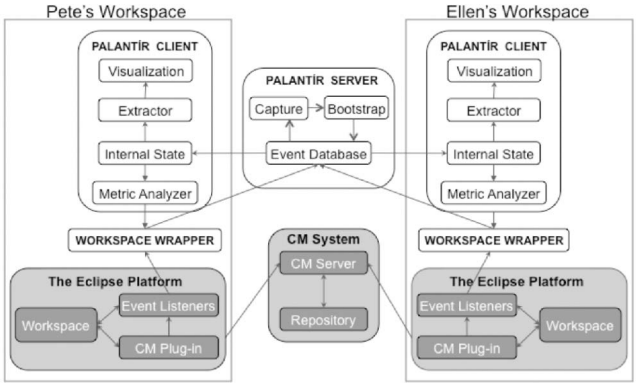


Fig. 1. Palantír architecture.

In this section, we discuss the overall Palantír architecture, briefly describing each of its components. An in-depth description of each component can be found in [44].

5.1 Palantír Architecture

The Palantír architecture (see Fig. 1) consists of the following components: The Workspace Wrapper, specific per SCM system and development editor, monitors SCM and editor related activities; the Internal State stores a local cache of events which are used to calculate potential conflicts and their severity; the Metric Analyzer performs cross-workspace conflict analysis to identify indirect conflicts and also calculates the severity of direct conflicts; the Extractor component is responsible for filtering and formatting the events from the Internal State cache; and the Visualization component presents notifications about conflicts and their information through an integration with the IDE. Finally, a central server, the Palantír Server, keeps a log of all events from events from all workspaces for persistent storage and to support bootstrapping new or returning clients. An in-depth discussion of each of these components and how our choices were shaped as our project evolved can be found here [44].

5.2 Events

At the heart of Palantír are the events that describe the ongoing activities in each workspace. Of importance in the design of these events is the need for Palantír to be portable across different SCM systems and not assume that all of those systems follow the same SCM policy [1]. The key insight is that, instead of capturing operations such as check-in, check-out, or synchronize, events represent particular states in which an artifact may exist in a workspace. Given a specific set of operations for a specific

TABLE 1
List of Palantír Events Describing Workspace Activity and Changes

Event	Interpretation
Populated	Artifact has been copied from the SCM repository into a workspace
UnPopulated	Artifact has been removed from a workspace
Synchronized	Artifact in a workspace has been synchronized with another version in the SCM repository
ChangesInProgress	Artifact has been changed in a workspace
ChangesReverted	Artifact has been returned to its original state when it first was copied into the workspace
ChangesCommitted	Changes to an artifact in a workspace have been stored in the SCM repository
MetricsChanged	Artifact in a workspace has changed, as a percentage of the total LOC
Impacted	Artifact in a workspace has an indirect conflict

SCM system then, the custom Workspace Wrapper for that SCM system is responsible for mapping the results of its SCM system's specific operations onto the states represented by the general events of Palantír.

The most common events are provided in Table 1, along with their interpretations. Additional events exist to deal with more esoteric cases, such as renaming or deleting artifacts. These, as well as a detailed description of how events are implemented and how artifacts are identified across workspaces, are discussed elsewhere [45].

Every artifact will go through one of two basic sequences of events. Artifacts that must be present in a workspace for ancillary purposes, such as compilation or just a quick examination of its code by a developer, trigger the simple sequence of a pair of Populated and Unpopulated events to signify their presence in, and eventual absence from, the workspace a developer uses. Artifacts that are modified trigger the following sequence:

$$\text{Populated (ChangesInProgress (MetricsChanged} \mid \text{MetricsChanged Impacted)}^{*}\text{ChangesCommitted)}^{*}\text{UnPopulated.}$$

The sequence of ChangesInProgress and ChangesCommitted, which may occur multiple times within the overarching Populated, Unpopulated pair, provides the skeleton for the two key events of Palantír: MetricsChanged, which indicates the magnitude of changes to an artifact, and Impacted, which indicates whether the changes lead to an indirect conflict. Because these events are sent out continuously, they enable the monitoring of ongoing changes and conflicts by developers.

The Metric Analyzer generates one or more MetricsChanged events. After an artifact has entered a state of ChangesInProgress, every incremental change made by a developer leads to a new MetricsChanged event being emitted. This event always includes a measure of severity (calculated as the percentage of code that has changed in the artifact as compared to the version that was initially placed in the workspace). This measure of severity is displayed to developers so they can determine the magnitude of the change and decide whether to defer resolution or address the conflict immediately.

A MetricsChanged event also includes a "diff" when a code change involves modifications to a class signature. This "diff" is used internally by Palantír to determine whether an indirect conflict is present. It is analyzed upon receipt by a particular workspace with respect to its local dependency graph that summarizes changes in usage of classes within that workspace. If an indirect conflict is detected, it is reported by emitting an Impacted event to indicate that the remote change interferes with a local change. The Impacted event informs other workspaces of both the artifact that caused the indirect conflict and the artifact that is impacted, along with information about the signature changes causing the conflict.

Receipt of a "diff" is not the only trigger for analysis. Changes in the local dependency graph that involve changes in the use of classes whose signatures have undergone change also trigger an analysis for the potential of an indirect conflict. If such a conflict has been introduced, an Impacted event is once again emitted.

5.3 Workspace Wrapper

Workspaces and their access mechanisms differ per SCM system. While most of the architecture of Palantír can be insulated from that fact because it is driven by the generic events defined in the previous section, the Workspace Wrapper is necessarily specific to the SCM system for which it must translate SCM actions and their outcomes into Palantír events. By the same token, the Workspace Wrapper is specific to the editor from which it must receive events regarding editing actions that have not yet been committed to the SCM system. We chose to use Subversion [54] and CVS [20] as the SCM system, and Eclipse [19] as the editor. To access information from Subversion, we use Subclipse [53], an Eclipse plug-in that allows us to interface with Subversion in a convenient and "Eclipse-like" manner through listeners that intercept Subversion commands invoked directly from the Eclipse interface. Similarly, we use the CVS plug-in that is included in the standard Eclipse distribution to track CVS commands. We note that other editors and other SCM systems have features similar to the ones upon which we relied in implementing the functionality of Palantír. In particular, we note that most SCM systems provide Eclipse plug-ins, which new workspace wrappers for these SCM systems could leverage.

5.4 Internal State

The Internal State maintains a summary of: 1) the changes being made in remote workspaces, and 2) the changes being made in the local workspace. It uses two sources of

information. First, it queries the Event Database to always maintain an up-to-date view of the activities in other workspaces. This query only requests relevant information, that is, events that pertain to the artifacts that are present in the local workspace. Furthermore, the query is incremental, only requesting new events since the last query. The second source of information is the Workspace Wrapper, which transmits any events emerging from the local workspace. All of the events are combined in a single cache, which is optimized for use by the Extractor and Metric Analyzer components.

5.5 Metric Analyzer

Palantír calculates two measures, severity and impact, for two kinds of artifacts—elementary artifacts (i.e., files) and compound artifacts (i.e., directories). For elementary artifacts, the Workspace Wrapper performs the calculation of severity as it already has all of the necessary information (the "diff") at hand in performing its other tasks. All other calculations are performed by the Metric Analyzer based upon the information in the cache maintained by the Internal State.

The severity measure is calculated based on the percentage of change present in an artifact. For individual files, the severity is calculated as the total number of lines of noncomment code changed divided by the total number of lines of noncomment code. To provide an overall picture of the state of the workspace, the individual severities of the artifacts are communicated up the directory tree. The directory severity is a compilation of the severities of the child artifacts contained within a directory, calculated as a weighted average over the severities of the artifacts that have been changed. A detailed discussion about this choice of how to calculate compound severity can be found elsewhere [42].

Palantír implements a binary measure for impact: An artifact is either impacted by one or more artifacts or not. Conversely, an artifact either causes impact on one or more other artifacts or it does not. Palantír performs this determination incrementally with information of each change transmitted to the Metric Analyzer. Particularly, it analyzes each remote change in a class signature and each local change in the use of classes to determine whether a potential conflict arises (e.g., insertion of a call to a method that in another workspace has a changed signature or insertion of a call to a method that no longer exists in another workspace). Impact for compound artifacts is calculated similarly to compound severity: as a relative measure that divides the number of impacted files by the total number of files in the directory. Details on the implementation of impact analysis are presented elsewhere [44].

While certainly more complicated and detailed analyses could be implemented to calculate the impact of indirect conflicts, we purposely chose a straightforward binary measure. Higher levels of precision incur increased cost, particularly when the algorithm used is not incremental, or require significant amounts of information to be exchanged among workspaces. This cost may be prohibitive, and it is unclear whether the benefits of increased precision are necessary. Finally, Palantír currently works only on Java projects because we use Dependency Finder for the underlying analysis. Implementing Palantír to work on other programming languages would require replacing Dependency Finder with other third party tools that analyze other programming languages (e.g., Understand[1] for C or C++). Note that once the internal cache of dependencies and diffs are generated via a given analysis tool, the rest of the implementation for Palantír remains the same.

5.6 Extractor

The Extractor component is responsible for filtering and formatting events stored in the Internal State so as to match user specified preferences or special formatting requirements of different visualization components. For instance, a user may decide to further filter the information presented to them by specifying that they be notified of only those changes that have a severity measure of 50 percent or higher. Alternatively, they may wish to only monitor a few select workspaces, selecting only those developers who are working on closely related tasks that they know have a high chance of interfering. The Extractor component is responsible for selecting the subset of events that matches a developer's preferences.

Additionally, different visualizations require different amounts and kinds of information (see [45] for all of the visualizations with which we have experimented). For example, it is possible in one of our visualizations to pairwise compare workspaces, whereas in another visualization all changes from all workspaces are aggregated. The Extractor component is therefore tailored to each visualization and, in addition to filtering user preferences, is responsible for maintaining a visualization's particular state.

5.7 Visualization

Throughout the development of Palantír, we experimented with a number of different visualizations to present developers with information on the direct and indirect conflicts that arise during the course of their work [45], [47]. Based on informal interviews with developers, managers, and students, we found the Eclipse package explorer view to be the most popular.

We designed the Eclipse package explorer view to integrate awareness fluidly in the day-to-day practices and work environment of software developers. Particularly, it aims to balance between effectively informing developers of conflicts, yet not continuously distracting them from their day-to-day coding activities. Rather than invasively notifying developers of conflicts, the Eclipse package explorer view inserts small awareness cues in selected parts of the standard Eclipse user interface. The idea is that the cues are unobtrusive, but clearly noticeable at relevant times when, for instance, developers switch from artifact to artifact or overall task to overall task. Developers can concentrate on coding, but are provided with the opportunity to gauge the severity and impact of remote changes when they need to.

Fig. 2 presents the Eclipse package explorer view with an illustrating example. Imagine a scenario where two developers, Ellen and Mike, are working on a hypothetical software system for a bank, written in Java. Ellen is responsible for updating an existing class, Payment.java, which offers facilities for keeping track of the payments that a person makes using his or her bank account. Mike has the

1. www.scitools.com/.

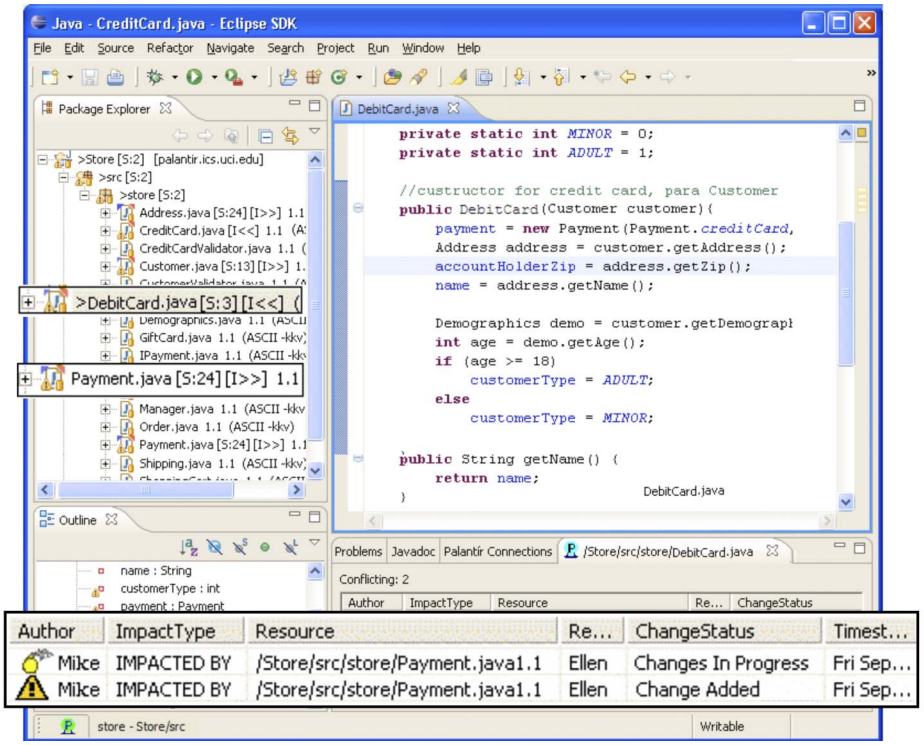


Fig. 2. Palantír package explorer user interface.

task of updating the class DebitCard.java, which uses Payment.java to keep track of expenses made with a person's debit card. Ellen, on reviewing the implementation of Payment.java that she completed previously, decides to move the initialization code to a new method called initialize(). Ellen also decides on another change, one in which the method addPayment (Amount amt) is modified to take as additional input a date and time stamp. At present, these are automatically set by Payment.java, but she realizes that it is more accurate to let the date and time stamp be set by the calling class. Her development environment warns her that this method is being called from DebitCard.java, so she checks it out and updates the calling code to reflect the new method signature.

In the meantime (while Ellen was contemplating and making her changes), Mike significantly refactors DebitCard.java, making some drastic changes to the set of methods, particularly splitting some and merging others.

In this scenario, one direct conflict and one indirect conflict have been introduced. The direct conflict involve Ellen and Mike both making changes to DebitCard.java. The indirect conflict is caused by Ellen's changes to the signature of the method addPayment() in Payment.java, which is incompatible with how it is used in DebitCard.java by Mike in his workspace. Fig. 2 shows the interface that would be presented to Mike as he is completing his changes to DebitCard.java.

Artifacts that exhibit a direct conflict are marked in the top left with a blue triangle, the size of which grows and shrinks with the evolving severity of the conflict. The color of the icon darkens or lightens as well. Artifacts involved in an indirect conflict, whether as the artifact that causes it or as an artifact that is affected by it, are marked with a red triangle on the top right. In textual annotations next to the name of an artifact, Palantír further explains the status of a change. In this case, the annotation of [S:24] on Payment.java indicates that 24 percent of the file has been modified, [I ≫] on the same file that its changes are causing at least one indirect conflict, and [I ≪] on DebitCard.java indicates that it is affected by at least one indirect conflict.

Additional information detailing the number of indirect conflicts affecting an artifact can be found in an extra tab that Palantír provides at the bottom of the Eclipse environment. In our example, Mike has selected (implicitly, by opening and editing it) DebitCard.java, for which the indirect conflict with Payment.java is annotated with a yellow bomb icon. The color yellow indicates that the indirect conflict still is in progress, residing in the workspaces only. Once Ellen checks in Payment.java, the color will change to red to indicate that the indirect conflict has become concrete. The exclamation annotation (!) indicates that a new method (initialize()) has been added to a file (Payment.java) being currently used by DebitCard.java.

6 Evaluation

We conducted a two-tiered experiment that evaluated the effectiveness and usability of Palantír through a programming and a nonprogramming task in order to minimize the impact of individual differences arising because of variations in participant expertise.

We present and analyze our experiment results by addressing three research questions:

1. *Does workspace awareness help users in their ability to identify and resolve a larger number of conflicts?*
2. *Does workspace awareness affect the time-to-completion for tasks with conflicts?*

3. *How does workspace awareness affect development behavior?*

The first two questions are answered through quantitative analyses, the results of which demonstrate the effectiveness of Palantír through statistical measures. The third question is necessarily qualitative, helping us understand how participants subtly changed their behavior in response to the notifications provided by Palantír.

In the rest of the section, we first present how our experimental design minimizes individual differences. We then present our experiment setup, followed by the experiment results and analysis.

6.1 Eliminating Individual Differences

In the case of user evaluations of workspace awareness tools such as Palantír, individual differences among study participants may dominate the results of the experiment by eclipsing the effects that the tool is intended to provide. The particular individual differences that concern our study are a programmer's technical skills [35] and the anticipated variance in when conflicts emerge because of differences in the order and the pace at which a team of developers perform its tasks. We explicitly designed our experiments to minimize these two individual differences.

With respect to differences in a programmer's technical skills, we used a two-stage approach. In the first stage, we evaluated the user interface of Palantír with nonprogramming tasks, where variances stemming from individual differences are minimal [35] and we could examine the effects of Palantír in a purer form. In the second stage, we repeated the experiment with programming tasks to determine the extent to which the same effects also arose in the programming domain. In so doing, we established a baseline with the first experiment that helped put the result of the second experiment in context.

In the first experiment (which we hereafter refer to as the Text Experiment), we used text-based assignments that relied on a cognitively neutral text. We specifically used a geology text, which we selected from a set of sample texts that we tested on a sample population (see [46] for details on how we performed this selection). The geology text exhibited minimal bias with respect to its complexity (it was neither too straightforward nor too complex to understand) and with respect to the level of expressed interest by the participants (they were neither too excited nor too agnostic about the subject).

Additionally, the text chosen for the experiment mimics some key properties of software, such as: 1) modularity, as the text consists of several separate artifacts, and 2) dependencies, as the text contains references that link text across artifacts, which must be kept consistent.[2] As such, the tasks provided to participants sufficiently resembled tasks regarding software changes that may directly or indirectly conflict, and therefore can be used to evaluate Palantír's basic behavior in terms of how the information that it presents supports individuals in coordinating parallel work.

The second experiment (which we hereafter refer to as the Java Experiment) evaluated Palantír in the programming domain with an analogous study, but involved participants making changes to a shared code base. This experiment sought to confirm results from the Text Experiment while taking into account the limitation that programmers' individual differences become visible, especially in the time it takes for them to complete tasks.

The second type of individual differences that we address in the design of our study concerns the anticipated variances in the occurrence of conflicts when a team of developers works on some set of tasks. With each team member making changes at their own pace, conflicts may or may not be introduced in the experimental setting, and those that are introduced will arise at different times for different teams. Drawing any statistical conclusions from such data is difficult, if not impossible. To mitigate this risk, we designed our experiments to use confederates, research personnel acting as virtual team members [7], [50]. This enabled us to keep the nature and timing of conflicts that were introduced constant. Participants were unaware of the tasks assigned to the confederates, the order in which the confederates would work on their tasks, or even that their teammates were confederates. We verified these facts in a postexperiment questionnaire. Participants thus believed they were in a genuine collaborative development setting.

6.2 Experiment Setup

The goal of the experiments was to mimic team development settings in which conflicts arise, allowing us to observe how individuals note conflicts and take action to resolve them, both with Palantír (Experimental group) and without Palantír (Control group). The Control group used Eclipse with a CVS plug-in, while the Experimental group had the additional use of the Palantír plugin. The individual nature of coding allowed us 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 interacted with the other, virtual team members under our control. All interaction among the team members took place via Instant Messaging. Each study participant was given the task of making a given set of changes, and was told that two other teammates (the confederates) were making some changes to the same artifacts as well. The confederates had the responsibility of introducing a given number of conflicts at set times, so the timing and nature of the various conflicts remained constant across the participants. The confederates were blind to the treatment condition of participants. Further, they were given a strict script of responses to chat conversations to mitigate any bias that a confederate may develop.

Each experiment took 90 minutes. Participants first completed a set of tutorial tasks to ensure that they could use the tools upon which the experiment relied. The Control group was given tutorials on Eclipse and CVS, which covered hands-on instructions for finding and resolving merge conflicts along with other relevant SCM functionalities available via Eclipse. The Experimental group was given the same tutorials on Eclipse and CVS, but included additional instructions on using Palantír. 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

2. The text, tasks, and all other materials used in our experiment are available on the Palantír website at http://www.cse.unl.edu/~asarma/palantir_experiment.

given the set of tasks to be completed. At the end of a session, participants were compensated $30 and were interviewed by the experimenter, who also was present throughout the experiment as an observer. Screen Capture software was used to record all keyboard and mouse actions as well as the screen content throughout the session.

The experiments were conducted at the University of California, Irvine. All experiment participants were graduate and undergraduate students in the Donald Bren School of Information and Computer Sciences. We had 40 participants total, 26 for the Text Experiment and 14 for the Java Experiment. Participants volunteering for the experiment first completed an online background survey asking about their experience in programming (including industry experience), SCM tools, and Eclipse, as well as demographic information. This information was used to make a stratified random assignment of participants [50]. Based on the spread of experience of the participant pool, participants in the Text Experiment with more than one year of experience in both SCM systems and Eclipse were assigned to Stratum 1 and the others to Stratum 2. In the Java Experiment, individuals with four or more years of experience in using SCM systems and Eclipse were assigned to Stratum 1 and the rest to Stratum 2. Following standard procedure in stratified random assignment, participants from each stratum were randomly selected for treatment groups such that each treatment group had an equal number of participants from each stratum to avoid biasing the experiments through participants' past experiences.

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 30 artifacts. Participants were given a set of nine tasks, six of which had conflicts: three direct and three indirect. Direct conflicts were introduced in Tasks 2, 4, and 8 when, a short while after each participant engaged in one of these tasks, a confederate would begin to change one or more of the same artifacts. Indirect conflicts were introduced in Tasks 3, 5, and 7, for example, by a confederate deleting one chapter or changing a chapter heading without changing the Table of Contents. 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 and did not contain any conflicts.

For the Java Experiment, participants were provided with a list of functionality to implement in an existing Java project. The project contained 19 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 involving parallel changes to the same Java artifact were introduced in Tasks 1 and 2. Indirect conflicts involving deleted or modified methods were introduced in Tasks 4 and 6. Tasks 3 and 5 were benign. Participants were provided with a UML diagram to help them understand the dependencies among the Java artifacts. 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, this would have required an extensive set of build and test scripts, as well as the seeding of several indirect conflicts not caught by those scripts. This would have seriously complicated the experiment and introduced several other potential design variables that we did not want to introduce or could realistically control for.

We closely controlled when each conflict was introduced (generally 10 to 15 seconds after a participant began or completed a particular task). Participants were presented with the set of tasks in the same order. Our goal was to observe the effects of the tool on the way in which participants handled both kinds of conflicts. Therefore, we treated the data for direct and indirect conflicts separately. We did not investigate interaction effects with respect to the order in which conflicts were introduced. For instance, whether conflicts that were introduced later in the experiment were resolved faster is an interesting question, but a topic for future study.

Dependent variables. The two primary variables of interest in our experiment were the number of seeded conflicts that participants: 1) identified and 2) resolved. Results were grouped into four categories, namely, conflicts that were:

1. Detected and correctly Resolved [D:R],
2. Not Detected until the participant was notified by the SCM system of a merge problem, forcing them 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] (refer to Table 2, which will be explored in the next section).

Conflicts that were incorrectly resolved are treated here as Detected, but Not Resolved [D:NR].

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.

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

6.3 Experiment Results—Question I

The primary goal of Palantír is enabling developers to detect emerging potential conflicts, thereby providing them the opportunity to avoid the conflict altogether or mitigate its effects. To test this hypothesis, we observed how effective Palantír was in enabling participants to identify and resolve a larger number of conflicts than participants who did not use Palantír.

Results. Table 2 presents the conflict detection and resolution observations for both the text and the Java Experiment. Participants in the Experimental 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, the results are found to be statistically significant ($p < .05$ for the χ^2 test; Fisher's exact test confirms the p values).

TABLE 2
Conflict Detection and Resolution Data

	detect	EXP	CONTROL	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 text experiment concerns a total of 39 direct and 39 indirect conflicts (13 participants in each group, three seeded conflicts of each type per participant). The Java experiment concerns a total of 14 direct and 14 indirect conflicts (seven participants in each group, two seeded conflicts of each type per participant).

Note that the results for direct conflicts for the Java Experiment are categorized differently into Detected versus Not Detected to address low expected cell counts in the χ^2 test. Of the 12 conflicts detected by the Experimental group in this experiment, nine were detected early and resolved and three were detected later, but not resolved (these three were for a direct conflict that was introduced after the participant had already completed the task). Of the seven conflicts detected by the Control group, all seven were detected during check in (which resulted in a merge conflict) and resolved.

Text experiment. Participants in the Experimental group detected and resolved a larger number of direct conflicts (DC) while they were working on their tasks (row 1, Table 2). These conflicts were resolved either immediately upon noticing them or after the participant had finished editing. We 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 then attempted to check in their artifacts, subsequently facing a merge conflict that they resolved.

The results for participants in the Control group are significantly different. None of the participants detected a single conflict beforehand. This is not surprising since the SCM system shields them entirely from parallel work and their only option to perhaps avoid conflicts would involve them continuously polling the SCM repository or asking their teammates repeatedly. Such a 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 upon attempting to check in the changes and being notified by the SCM system of a merge conflict.

In the case of indirect conflicts (IC), we again observe that a majority of participants in the Experimental group identified and resolved a 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 SCM system. In the case of indirect conflicts, however, participants in the Control group identified only five indirect conflicts; the other 34 remained undetected and remained in the final version stored in the SCM repository. This failure occurred even after participants were explicitly encouraged to carefully examine the text for inconsistencies as the last step in the experiment. By comparison, participants in the Experimental group detected and resolved 31 conflicts.

In both the Experimental and Control groups, 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 deduce the dependencies among the artifacts. For instance, some participants did not detect that the confederate had slightly modified the caption of a particular figure in one of the chapters and that it affected the List of Figures that they were supposed to update accordingly.

Java experiment. We found that the Experimental group detected a larger number of direct conflicts early, 12 out of 14, differing significantly from the Control group, which detected 7 out of 14 (row 3, Table 2). In the case of indirect conflicts, we notice that all participants in the Experimental group identified and resolved conflicts, whereas none in the Control group even detected a single conflict (row 4, Table 2). These results confirm the findings of the Text Experiment and show how incompatible changes once again entered the SCM repository unnoticed without Palantír.

Note that for direct conflicts, the observed outcomes for 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) there was one conflict that was seeded by the confederate after the participant had already completed the task. With respect to this second point, in a typical SCM environment, a developer who checks in first generally is not the person who is responsible for conflict resolution; instead, it is the responsibility of the next

TABLE 3
Time-to-Completion of Tasks

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

developer who checks in their changes. Therefore, it was not surprising that our participants did not always resolve this conflict. For analysis purposes, the low expected cell counts meant that, instead of using the standard four-category breakdown, we had to group the results into Detected versus Not Detected.

Summary. While the p-value for direct conflicts in the Java Experiment is somewhat higher than in the other three cases, which had $p < 0.01$, it is statistically significant (p value < 0.05). Overall then, we can confirm that the use of Palantír leads to statistically better improvements in the detection and resolution rates of conflicts in our experimental settings.

6.4 Experiment Results—Question II

Monitoring the awareness cues provided by Palantír and taking the coordination step is an additional activity that is bound to have some time overheads. Here, we wanted to quantitatively determine how using a workspace awareness solution affects 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). We use this table to analyze the overhead that is imposed by the use of Palantír. As such, the time-to-completion for each individual task includes the time to detect, investigate, coordinate, and resolve a conflict, as applicable. We do not penalize participants who missed detecting or resolving a conflict, choosing to simply report the time they took to complete the task.

In the Text Experiment, participants in the Experimental group took less time for direct conflicts, but longer for indirect conflicts (rows 1 and 2, Table 3). But, in the Java Experiment, the Experimental 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 $p = 0.26$.

Text experiment. We observe that participants in the Experimental group took less time (on average, 3 minutes fewer) to complete tasks with direct conflicts. At the same time, we see a reverse trend for indirect conflicts (the Experimental group took on average one-and-a-half minutes longer). This difference can be explained because, in the case of direct conflicts, Control group participants were warned by the SCM system of a merge conflict and forced to resolve each such conflict, while no forcing factor existed for indirect conflicts. This forcing factor resulted in participants in both the Control and Experimental group resolving exactly 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 Experimental Group, on the other hand, coordinated with the confederate upon noticing an emerging conflict 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 those in the Control group faced (see also Section 6.5).

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 Experimental group detected and resolved a majority of the conflicts, causing them to incur an extra coordination effort (primarily additional communication through instant messaging) in investigating conflicts and resolving them with their teammates (confederates). As a result, the average time per task was higher. The tradeoff, of course, is that the code delivered by the Experimental group had most of the conflicts resolved, which means that it would incur no further future effort to resolve these conflicts. Our experimental setup was designed to quantify this future effort by asking participants in the Control group to examine and correct the text after all change tasks were completed. Participants, however, could rarely find any of the remaining inconsistencies. Therefore, no usable data were obtained regarding the amounts of time and effort that might have been saved.

Nonetheless, a critical observation arises: At the expense of extra effort, the number of conflicts remaining in the text that was delivered was significantly lower.

Java experiment. The data for the Java Experiment showed a larger variance in average time-to-completion. In the 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 Experimental group did take longer than the Control group on average. In closely examining our data, we did not find any factors

other than a probable cause of individual differences in programming skills explaining this difference. As both treatment groups detected and resolved about the same number of conflicts, this seems to be the likely explanation.

In the case of indirect conflicts, we note a pattern similar to the Text Experiment, with statistical significance ($p < 0.05$). In particular, the Experimental 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, with the number of (indirect) conflicts remaining in the code taken as a proxy for quality.

For reasons explained previously, we did not attempt to force the Control group to reexamine the code base at the end of the Java 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 might take significant amounts of time (sometimes on the order of days) [11], [31]. Any indirect conflict saved from entering the SCM repository thus constitutes one fewer and possibly major future concern.

Summary. There is an overhead to using Palantír, but our results suggest that this overhead is strongly justified. For direct conflicts, the Text Experiment indicates less time spent per task, despite the overhead of monitoring and interpreting Palantír notifications. This is the clearest evidence that resolving conflicts early is beneficial as compared to resolution later. For indirect conflicts, more time is spent, though the overhead is not unreasonable, especially when one realizes that the result of the extra effort is a less error-prone body of code in the SCM repository.

6.5 Experiment Results—Question III

A critical functionality of Palantír is its ability to prompt users to take steps to self-coordinate. Evaluating whether Palantír is successful in eliciting such coordination behavior has to be qualitative in nature as users can take a wide spectrum of possible actions. Here, we present the different behaviors that we observed during the experiments regarding development work.

Using concepts of grounded theory, we analyzed data gathered from exit interviews, observation logs, video transcripts, and chat transcripts between the subjects and confederate(s) to identify behavior patterns emerging from the data. Grounded theory is the systematic generation of theory from data acquired by a rigorous, iterative qualitative research method [52]. From our analysis, four broad themes emerged:

1. conflict resolution,
2. conflict mitigation,
3. manifestation of awareness, and
4. awareness monitoring behavior.

Together, these help explain why some of the effects we saw in the previous two sections occurred, as well as how the users actually coordinated their efforts and how their strategies were positively influenced by Palantír. In the following sections, we discuss each of these behavior patterns.

6.5.1 Conflict Resolution

Participants in the treatment groups varied in their conflict resolution strategies. We found that participants in the Experimental group developed a team spirit and interacted with their team members to resolve a conflict when they were made aware of their team members' activities. In contrast, Control group members followed an individualistic model, favoring their changes over those of others. Further, the time at which conflict resolution was performed (e.g., at the onset of a conflict, after full completion of a task) was largely dependent on individual preferences and varied across participants.

Human-in-the-loop resolution. We found that participants typically liked having a "human in the loop" when resolving a conflict. Even though, in both cases (Control and Experimental groups), participants felt that their own changes were of higher quality as compared to those of the confederate, exposing a personal bias, Palantír users felt a higher camaraderie with their team members and typically had chat conversations when resolving a conflict. The appreciation of having a human to discuss to resolve a conflict is best summarized by one of the participants using Palantír: "*It was extremely important to be able to chat with the other workers because changes need a human touch.*" Two examples illustrate how users resolved conflicts with the help of other users.

First, from the Java Experiment: "*...so [file name] has changed, there's no more...[details of deleted methods]. I'm assuming you want me to fix the codes with errors?*"

Second, from the Text Experiment:

Subject: you added some timeline pictures to [timescale.txt].

Subject: the middle one should be [timeLine2.png], right?

Subject: also, will you be uploading these files to the repository?

Confederate: I have actually replaced it with 1 picture.

Confederate: graphic [timeline.tif].

Subject: ok, please update [timescale.txt] to reflect that change.

Subject: also, make sure your caption is written with the proper syntax.

Confederate: Sure I will do so.

Interviews confirmed how Palantír helped spark human conversation. One Palantír user (Text Experiment) mentioned: "*In order to mitigate conflicts when performing changes I looked at the blue arrows before editing the file, then proceeded to IM the person modifying the file to check to see if I could obtain access to the file.*" Another, when asked specifically about how Palantír affected conflict resolution behavior, responded: "*Yep; [I] told him what I did, I asked him if I can do some stuff and commit it and he said yes, and I think there was code that was slightly same, I verified with him if I can overwrite and he said yes.*"

We also note that, when participants who were using Palantír found a problem, they were able to start a conversation with their team with the exact details of the problem immediately at hand. This is possible because Palantír affords both parties with information about the ongoing conflict. Here, is an example of a typical chat exchange from the Text Experiment:

Subject: why did you change the [name-year] pair to [Jean-Andre Gucci, 1558] and [Horace de Saussure, 11779]?

Confederate: which file is this?
Subject: the introduction text.
Confederate: it was part of my task.
Subject: My task said to change the pairs to [Jean-Andre Deluc,1778] and [Horace-Benedict de Saussure, 1779].
Confederate: oh Ok!
Confederate: I have completed my changes.
Confederate: You can overwrite them if you want.
Subject: ok, thanks.

Another example shows similarly rich context to immediately anchor the conversation:

Subject: I noticed you deleted the getter/setter for [Demographics] in [Customer]. I am implementing a change to [CreditCardValidator] that needs those methods in order to work.
Confederate: Yes, I am making some changes to the [Customer] class.
Confederate: I will let you know as soon as I complete.
Subject: will the getter/setter still be there when you are done making your changes.
Confederate: For the [Demographics] field? No it will not be there.
Confederate: I am replacing it with the "Age" field.
Confederate: along with a getter and a setter for that.
Subject: alright that works.
Subject: thanks.

In contrast, a majority of participants in the control group kept their own changes without thinking to contact the team member responsible for the conflict. For instance, a participant comments aloud when resolving a conflict: "*So...hmmm, someone already had done this! Oh!! I see someone had done it, but hmmm I think they made a mistake...*" [the participant then goes ahead with the assumption that his task is the right one and overwrites the file in the repository]. Another participant, on finding a conflict, would repeatedly ponder aloud "*Is there any way I can just overwrite the CM repository with my changes. I know my changes are better.*"

When questioned about conflict resolution strategies at the exit interview, a non-Palantír participant said: "*so, I just used the commit thing, so when it said a problem, I synchronized to see the differences and then I updated...I compared the differences and if there was additional information then I included it and if it was information that was, I guess worse than what I was adding, I ignored it.*" It was interesting to note that during the experiment the participant had largely kept all his changes.

We found that the majority of participants in the control group did not feel the need to communicate with their team members. When asked about this in the exit interview, a participant responds: "*Yea...that [confederate's parallel actions] would make conflict, if I don't know what they are doing or what they are supposed to be doing. If I knew what they were doing then I could see if their changes were applicable or not.*" Another subject responds: "*umm...they [changes] were pretty obvious things—[I] didn't need any help.*"

Timing. We found that the resolution behavior of participants regarding when they resolved a conflict can be categorized into two groups: 1) participants who took immediate action and 2) participants who noticed the conflict when it appeared, sometimes even investigated the conflict further, but only resolved the conflict after they had completed their own task.[3]

For example, one subject was highly cognizant of the conflict icons appearing and made sure that she resolved the conflict then and there. As soon as she started working on a file, she would check to see whether the file had a conflict. When she found that it did, she updated the file with the contents from the repository and then proceeded to check the "impact view" to find out more information about the conflict.

Compare this behavior with that of another subject, who skipped all tasks that involved files with conflicts. In fact, he skipped four tasks in succession (Tasks 2, 3, 4, and 5, each of which included a direct or indirect conflict). He even stopped working and skipped checking in the files for Task 3 when a conflict arose while he was working on the file. After finishing all nonconflicting tasks, he finally went back to the tasks involving conflicts. When asked about this behavior, he commented: "*I skipped the task, when I saw that there were changes on the file that I was working on. Later on I realized that it was always going to be the case and that I should complete my task so I went back.*"

We also noted that participants were interested in ensuring resolution of all problems. In fact, even when they noticed a conflict with a task they had already finished and checked in, they would inform their team member (who apparently had forgotten to check out the latest version before making their changes) about the problem. Here, is a chat excerpt where the participant warns their team member about the conflict that they will have to resolve: "*...ok, be warned that you are going to break [artifact name] when you check it in...*"

6.5.2 Conflict Avoidance

One of the key user characteristics to emerge was that, while Palantír helped users in dealing with conflicts, users still attempted to avoid them when possible. The previous section has an example (the participant who skipped all tasks with conflicts), but this behavior is worth discussing in more detail. Two representative comments are:

"*The hardest part of the tool was merging two documents together. Just adding text and bizarre syntax symbolizing changes made it extremely complicated to make it [understand] what the two documents should look like together. When I was making changes to a document that had already been changed and committed, the hardest part was merging my changes with the new version.*"

"*...well I don't know if people like that [conflict resolution], but I didn't. It made me happy when the tasks that I did didn't include anyone else's; anyone else has not trampled on it yet so didn't have to worry about that.*"

This behavior mirrors current anecdotal complaints on merge conflicts and observations on how developers create intricate informal practices (e.g., partial commits [15], rush to complete changes first [24], informal locks on artifacts [30]) to avoid performing conflict resolution. We found that

3. Note that this behavior was observed only for experimental group participants. Participants in the control group found conflicts only during a check-in or at the end of the experiment.

being armed with awareness of ongoing activities actually helped developers fine-tune these informal practices.

Conflict mitigation strategies. The different coordination actions that participants took to avoid or at least mitigate the impact of conflicts on detecting a conflict can be grouped into five classes:

1. partial commits (checking-in their artifacts even when their task was still unfinished),
2. rush to complete their changes so they can check-in first,
3. use placeholders,
4. skip their current task, and
5. chat to informally coordinate their tasks (e.g., by informally locking an artifact or determining the future tasks of their team members).

Partial commits: We found a few instances of this strategy being employed when a task required edits to a set of files or a set of edits to a particular file and a conflict would arise in that task. In such a situation, the files or changes that were already completed were checked-in before the rest of the task was completed, For instance, one participant was extremely cautious and averse to conflict resolution. The first time a conflict arose the subject ignored it and faced a merge conflict. After that she made sure to check for any conflict icons before starting a task. She also committed every small unit of change that she made. For example, when her task (Task 3) involved two files and sets of instructions for each of these files, she made a total of five commits during the task (in a time period of around 14 minutes). Several other participants exhibited similar behavior, breaking up their tasks to preventively check-in partial work.

Rush to commit first: We found scattered instances of participants rushing to commit their changes first to avoid facing a merge conflict. For instance, a participant responds on his conflict resolution strategy: "*ha! There was this one time, when I purposely committed right away, so wanna make sure that I wasn't the one to resolve the conflict*." However, this strategy did not work in our experiment setting because a majority of the conflicts introduced by the confederate were scripted and instantaneous, so there was not enough time within which a subject could notice the conflict, understand its significance, and rush to quickly commit their changes. Subjects would try it the first time and then quickly learn that this strategy was infeasible in the given setting. For example, upon spotting a conflict, a participant exclaims: "*Yo, oh no!!! What are you doing this [confederate name]. I am going to commit first. I don't want to resolve.... I know what it means, but just don't like dealing with it, hmm lets see [he tries to commit, but gets a merge conflict].*"

Use placeholders: Subjects who identified an indirect conflict sometimes would adjust their own code to already accommodate the change in the other workspace. While their current code therefore would not compile, the final merged version would. Below are observations and chat excerpts of two participants (one from each experiment setup) following this strategy. In the first case, a participant in the Text Experiment notices a conflict icon on one of her files. She then tries to synchronize her work with the repository, but finds that the changes are still work-in-progress. She starts a chat conversation with the confederate to further investigate the conflict.

Subject: What changes are you making to [history.txt].
Confederate: I am making a few to the name [William Smith].
Confederate: I am just about to commit my changes.
Subject: Ok thanks.
Confederate: I have committed the file.
Subject: What are you changing the name to?
Confederate: [William Theodore Smith].
Subject: ok thanks.

She then uses the anticipated change to the name (William Theodore Smith) in the text that she is modifying and continues with her task. Similarly, a participant in the Java Experiment chats with the confederate (chat not shown here) and then decides to already use the variable "age" in her changes, even though it does not exist yet. When finished with her changes, she does not commit the task and moves on to the next task. After completing this next task she checks back with the confederate to find out whether he had committed the changes so she can finish her previous task and commit it.

Skip their current task: We found numerous instances of this strategy—subjects skipping tasks that would involve files already being edited by others or already being marked as being indirectly impacted by an ongoing change. For example, on noticing an indirect conflict warning, one of the participants made the decision to skip their next task. She commented aloud: "*I'm going to be affected by [file name], so I'll skip the task and do something else.*" When questioned at the end of the experiment about her decision the participant replied: "*Right, I skipped that task. And then I sent him [confederate] a message asking whether he is going to add the method, and then he said he will, so I decide to wait for him*."

One interesting outcome of skipping a task was that sometimes it avoided duplicate work since one of the tasks for the participants was a duplicate of a confederate's task. If the participant was cognizant of the conflict and investigated it before they started their task, then it was possible to not only avoid the conflict, but also avoid having to implement the task. A user comments: "*...before I started working on it, Palantír tells me that someone is changing it, so I went and checked, saw that everything is there, so cool, task completed and no conflicts.*"

Chat to informally coordinate: We already have discussed how participants liked having a human in the loop during conflict resolution. They also chatted with their teammates to proactively avoid conflicts altogether, as evidenced by these two excerpts.

Subject: dood? Are you working on [Address.java]?
Subject: I mean, is it pretty stable now?
Confederate: i made some changes.
Confederate: and I've committed them.
Confederate: its stable.
Subject: cool, thanks!

Another participant checks on another artifact of interest:

Subject: hey I am subject 1.
Subject: hey, i will change [creditcard validator java].
Subject: will you change it?
Confederate: Let me check.....
Subject: ok.

Confederate: Hmm.... No I do not plan to touch that file right now

Subject: ok, please do not change it until i will finish, i will let you know when i will finish

Confederate: ok.

We note that participants coordinated their activities based on their task at hand and an awareness of their teammates' changes. Chat messages were used to informally lock artifacts that the subject would change in the near future.

Implicit communication. An undercurrent in the discussion thus far is that of implicit communication. While a fair amount of explicit communication takes place, Palantír users were able to understand ongoing activities through the visual cues that the tool provides and self-coordinate their actions. They did not always need to communicate as Palantír regularly provided them with the information they needed. For example, a participant commented: "*There were many times during the experiment that someone else opened a file that I was editing, and the tool gave me notice of this that I probably wouldn't have noticed otherwise, allowing me to update my version before wasting my time inputting redundant information.*"

Another observed: "*The parallel changes warnings were very useful because it let me know who was working on what at any given time. This allowed me to not only stay away from files others are working on to reduce conflict, but also gave me a heads up to contact other team members when we are working on the same files. The affected files warnings were helpful because it let me know if a change in one file will affect any other files. This is a great help because with the complex relationships between files in program code, a very small change can affect many files down the line. This warning helped to fix those problems early on.*"

Generally, participants would only use chat communications when the information provided by Palantír was insufficient for them to resolve a problem on their own. When asked about this in the exit interview, a participant responds: "*Yeah, I just updated myself when I could. When I saw that there was impact (red icon) I would talk to him [confederate]. Whenever I found that I could go ahead without talking to him I would, but when I could not get the changes I talked to him.*"

Another participant mirrors this sentiment: "*well, early on I was actually asking him [confederate] multiple times when I noticed there was a potential conflict if he had checked in his changed yet. I realized later that that was unnecessary.*"

6.5.3 Mutual Awareness

Maintaining awareness through (explicit) user behavior. We noted that in a majority of cases Palantír users wanted to ensure that they were kept abreast of the activities of their remote team members and likewise wanted to make sure that their team members were aware of their actions. A participant in the Text Experiment comments aloud: "*Usually I would like people to know that I'm changing the file, make sure he knows I'm changing the file. Maybe he knows, but I don't know.*" Another mentions that during the experiment he wanted to ensure that his team members knew about all of the changes he was making: "*I wanted to keep updating, as I wanted all the changes to be seen.*" Many participants at the beginning of the experiment wanted to know about the frequency at which Palantír notified users. One participant noted aloud to himself: "*Does the system show other users only when I make changes and save them? Or [silence follows]...*"

In fact, we noticed that in some cases participants communicated with their team members specifically to keep them abreast of their changes even though they knew that Palantír would send notifications. A participant comments: "*So, I am sending him a message and telling him what I am doing—its kind of duplicated because its in the repository, but never mind.*" A chat excerpt from another participant shows a similar intent: "*Hey, buddy. I'll be hittin' up [item.java]. But, it looks like you're on [shopping cart], so you're fine. Just keep an eye out for my triangle [notification icon].*"

Participants used the information about the artifacts that their teammates' were currently editing along with the knowledge of their own task to warn their teammates about potential future problems that they may face. A user responded: "*Um, Palantír helped me sometimes identify when someone was changing a file that I was supposed to change. It also helped me talk to that person, and try to warn that person that I would like to change the file. Palantír doesn't tell him my intention of changing the file, so I needed to do that through the IM system, but it tells me that someone is doing it, so it helped me prevent that kind of conflicts that I get when I check in a file.*"

Another participant comments about a similar situation: "*I contacted team member. I saw that he was changing a file, which was broken before, and I noticed [task reference], I assumed that he was fixing the bug that I was fixing too. And I then alert him not to do that because I had already fixed the bug. And he was very grateful to me.*"

6.5.4 Monitoring and Internalizing Awareness Cues

One of the interesting challenges to creating workspace awareness is deciding when is the right time and the right way to provide information about ongoing activities to create appropriate context for users' changes. In Palantír we made the decision to broadcast change and impact information every time a user saved their changes. However, we found that users noticed the awareness icons mainly only at specific points. Here, we discuss when participants noticed the awareness cues and how they reacted to the cues.

Monitoring awareness cues. The times when users noticed the Palantír icons can be classified into three categories: 1) when participants started a new task, 2) when participants were taking a short break from their task, which usually occurred when there were multiple files involved in a task, and 3) after they finished their task, but before checking-in their changes.

Regarding the first case (before starting a task), a participant notes: "*It's useful to know what someone is working on before they actually check it in, to try to resolve conflicts before they happen, or at least be aware that they are going to happen. Because I found that even before I started a task, I tended to try to find if someone was currently working in the classes that would affect what I was doing so that if it was possible, to update them and avoid going the wrong direction, I could do it before I actually began the task.*"

Another participant commented: "*When I was working, I was concentrating on the editor and not looking at the side. However, when I was starting a task and I saw the things [icons] on the file, then I tried to understand what that was about. So*

before opening a file, I would like to see what arrows [awareness icons] were there."

Another typical point when participants noticed the icons was when they were taking a break from focusing their attention at the editor. This typically occurred when subjects were switching between files for a task. For example, we observed a participant whose task required edits to two files. The participant made all the changes to the first file and saved the changes. Before she started working on the next file she noticed a conflict warning for that file. She then used Palantír to get a basic understanding of the conflict and thereafter communicated with her teammate to determine its full extent and her next steps.

The final point at which participants typically took notice of the visual cues and became aware of the conflicts was before commenting and committing their changes. One participant comments on his strategy for completing his tasks: "*...my approach of first doing my task and then checking what was happening allowed me to concentrate on putting my energy into something that I was supposed to be doing than trying to figure out what I should do next*." Another participant responded: "*I want to go ahead and finish a task first. And compare after I have finished the task to see how my changes clashes with others changes, so I didn't want to take any premature actions.*" Note that in such situations the participants had to resolve the conflicts after their changes were complete. Still, Palantír provides a benefit compared to standard SCM functionality, particularly because the other changes with which they had a conflict could well still be in progress, so the resolution is likely to still be easier.

Internalizing awareness cues. A common concern of awareness tools is whether the awareness information will distract the user from their main task. We noted that users quickly became adept in filtering pertinent information. Participants would quickly scan the package explorer view for the cues and only pay attention to icons that appeared for artifacts that were relevant to them. Furthermore, a majority of participants did not take any action (even after hovering over the icon and getting an extra information on the conflict) to resolve a conflict unless the conflict involved an artifact that they were currently working on. When asked about this behavior, a participant responded: "*yeah, I noticed there were other icons, but, they were like, on others [artifacts], they did not bother me. I only look at stuff that were on artifacts that I had open in my editor*." Another participant, when asked why he did not take any action on noting a potential conflict in an artifact, responds: "*I noticed the problems, but, you know, they were not affecting me, so I was like fine, it doesn't bother me, so I was working on my own files*."

We explicitly asked users whether the awareness icons provided by Palantír were distracting. Quite a few participants complained that while they were focusing on the tasks the cues were too subtle to grab their attention and recommended "pop up" notifications. A participant when asked said: "*The small blue and red triangles were large and colored enough for me to notice, but not enough so that they were a huge distraction. They also didn't take up noticeable space or require the UI to expand.*" He went on to explain that he would have liked more explicit warnings about ongoing changes to the file that he was currently editing: "*Its just that when I working, I am focusing on a particular file, I don't pay attention to what's going on in the left...If its [icons] blinking, then I will notice, but if it just pops up, then I don't notice it.*"

Another participant provides similar feedback: "*Small icons are really good because they don't interfere too much, but I still had to pay too much attention. A lot of attention to see the changes, I had to keep looking. There weren't anything that caught my attention as soon as it changes. I was thinking about when you are in IM, you see briefly, a good sign that something happened, and then it fades. So I think something like that would help me see things.*"

It would be interesting for a future experiment to determine whether such more explicit actions lead to the same levels of effectiveness and user flexibility in the actual actions they take to address the conflicts.

7 Threats to Validity

7.1 Generalizing Results

Our experiment involved students as experiment participants working on a relatively small project, with a small number of changes to be made in a limited time. Naturally, we agree that the ideal approach to testing a software tool is a thorough longitudinal study set in a real-life software project. At the same time, we 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 detailed conclusions and controlling for individual differences.

It is interesting to observe that, because students were used as participants, it is actually possible that our results are conservative with respect to the real world. Software developers who have experienced the difficulties of integrating their changes with conflicting changes made by others may well be more willing to invest time and effort up-front in order to avoid these difficulties. Also, there is bound to be a learning effect if Palantír were to be installed in the real world. In particular, we would expect developers to learn to ignore certain conflicts and pay particular attention to others. For instance, they may know that some coworker always delivers code that integrates nicely, even though in the beginning any conflicts that arise might seem difficult. They may also know that another coworker tends to engage in changes that are seemingly innocuous at the beginning, but always end up being difficult to resolve in the end. Coordination actions taken by developers in response to activities of either coworker will differ, and these differences possibly will streamline and make the use of Palantír more efficient.

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 did include 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 they were editing or had previously edited. This highlights that the awareness cues used by the tool were unobtrusive, yet effective when they needed to be. We

also note that Palantír is explicitly designed to reduce the number of notifications presented to the user by grouping them per artifact and providing additional filters on artifacts or developers, as discussed in Section 5.6 [44].

7.2 Experimental Design

Our experiment, specifically the Java Experiment, did not force participants to carry out integration testing after all tasks were completed. As participants in the Control group did not detect any indirect conflicts, they thus did not spend any time in coordination attempts. As discussed, this difference penalized the Experimental group in their time-to-completion of tasks and precludes us from comparing the time-to-resolution data across treatment groups. It thus may be possible that the extra time and effort that the developer is asked to invest because they use Palantír is actually more costly than the resolution of full-blown conflicts at a later time. However, given that Palantir allows resolution of conflicts earlier in a change process and knowing from other research how high the cost is for resolving changes later in the process [14], [23], we believe that the cost concerns of using Palantir are minimized.

Another design choice that may represent a threat to validity is the introduction of a conflict 10 to 15 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 would be different if conflicts arise closer to completion of a task since some of the benefit of early resolution would be lost.

A potential issue also exists with respect to conclusion validity in drawing inferences about the influence of awareness on programming tasks from the lessons learned from the Text Experiment. Working with text can be different from working with code, and participants may have an affinity with code that influences their behavior with respect to code 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 and the structure and relationships that the overarching text exhibited. Our results show that the behaviors exhibited by participants in the Java Experiment strongly resemble the behaviors of the participants in the Text Experiment, with the same kinds of statistical significance, building confidence that we should be able to rely on the Text Experiment to draw the conclusions we did regarding how developers react to and use Palantír.

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 sequence of the conflicts in terms of their order and kind. We also observe that learning effects will take place in real life as well, and in fact are a desirable trait of awareness tools—users must calibrate the information that is provided in order to best leverage the tools. The short duration of our experiment potentially undervalues this factor.

8 Discussion

In this section, we summarize the key lessons from the user evaluation and their consequences for the design of future awareness tools, as well as the design of empirical studies on awareness and coordination.

Current interface designs for fostering awareness, including Palantír, are based on the premise that developers should be continuously informed about parallel ongoing changes so they are aware of changes as and when they occur. However, observations from our experiment show that participants typically noticed the awareness cues provided by Palantír only at specific points during a task (before starting a task, when switching across files, taking a short break, or after finishing their tasks). This implies that continuous streaming of information or constantly updating graphical displays may not be necessary. For instance, it may be possible to both query for remote information and present it only when a developer switches which artifact they work on. As another example, it may be possible to provide mini views that show summarized histories. Such solutions might make workspace awareness systems computationally less expensive and reduce the information overload on users.

Nonetheless, no one solution should be assumed to be universal since different users have different preferences in how they would like to be informed. In our experiment, we found participants who desired more explicit notifications about critical events. These participants were usually immersed in their coding activities, focusing solely on the editor. Because of their immersion, they missed notifications that were provided peripherally (e.g., icons in the package explorer view). To cater to these types of users, more explicit notification mechanisms need to be explored—notifications that immediately draw the user's attention to critical events. The tradeoffs between immediately attracting the user's attention versus distraction are critical when designing such explicit notifications, and may perhaps even lead to different kinds of notification mechanisms for different kinds of events.

We found that, despite the availability of Palantír, users still employed informal conflict mitigation strategies, such as partial commits or soft locks—strategies that have been observed in teams without informal coordination support. Our observations show that, by using Palantír, users were able to make these strategies more fine-grained and purposeful. This suggests that formal and informal coordination are both an integral aspect of development and one cannot be totally separated from the other. The critical point is that, with workspace awareness tools, users have the flexibility to adopt the mix of strategies that suits them the best, with the possibility of this mix even varying per conflict and per team member with whom a conflict is generated.

With respect to communication, users of Palantír quickly started rich conversations about a particular task or change. This was possible because the same information about a change was available to all team members and every team member was aware of this fact. We believe that future awareness tools could improve their support to further enrich such conversations. For example, systems that provide

comprehensive background and context information of changes could enable users from different teams to quickly create common ground and thereby help coordination.

A subtle social outcome of using Palantír and the awareness generated by it was that users felt more connected with their team members and not as isolated. Palantír was therefore able to greatly enhance a sense of shared presence with team members as opposed to when teams only used SCM systems. In the latter case participants became aware of others' activities only during specific synchronization (check-in/check-out) points. A direct outcome of this phenomenon was that users were willing to make extra efforts to work together more effectively (e.g., warning their colleague about a potential problem, being more inclined to correctly resolve changes instead of overwriting others' changes in the repository). Further, this willingness to make extra efforts implies that when users are able to see the direct benefits of the effort they make, they are more apt to take the few extra steps (e.g., not leave commit comments empty, explicitly linking a commit with its issue number) to provide the information necessary for awareness technology to be most effective.

Finally, we note that implicit coordination was fostered by the information provided by Palantír. As discussed previously, we found that subjects used awareness cues about ongoing changes to implicitly coordinate their activities (e.g., skipping a task, using a placeholder). Some participants commented that they used chat conversations only when they could not find information that they needed from Palantír. This has implications for a large body of work involving empirical studies that use communication as a proxy for studying coordination [8], [9], [33], [48], [55]. These studies make the assumption that more communication is a sign of good coordination. However, the presence of implicit coordination demonstrates that this is not necessarily a valid assumption, especially in situations where users can acquire information through other sources and take implicit coordination actions. In fact, the presence of more communication may well be a sign of ineffective coordination strategies being used. Researchers performing empirical studies should take this into account.

9 Related Work

CVS [5] provides what might be considered the oldest configuration management awareness solution in existence. Before changing an artifact, developers can announce their intent of doing so by invoking the CVS EDIT command. Any developer who declared their interest in this artifact through the CVS WATCH command will then be notified, usually via e-mail. However, this required manually setting watches and the need to sort through potentially numerous automated e-mails—a tedious process, which developers avoided.

Since then, a number of other systems have implemented improved versions of workspace awareness. BSCW [3] provides a web-based, shared, centralized workspace that integrates versioning facilities to allow it to be used as an SCM system. Awareness of parallel work is provided statically, through icons that decorate an artifact's webpage with information regarding its current state, and dynamically, through a Monitoring Applet that continuously informs developers of what editing activities are presently taking place. While this is one of first tools to successfully implement the concept of awareness, it suffered from two drawbacks: a cumbersome interface and an overload of information, since BSCW presented many fine-grained details.

The War Room Command Console [39] used a centralized, multimonitor display to show all artifacts in the software repository, color coding and decorating those that are concurrently being edited in multiple workspaces. This system took a different approach to awareness as the information was centralized and aimed at supporting group meetings focusing on understanding the system as a whole and project management.

Compared to Palantír, neither CVS, nor BSCW or the War Room Command Console provides information on the severity of direct conflicts. None of them addresses indirect conflicts either.

Elvin [22] and Celine [21] do provide awareness of direct conflicts. Elvin uses a relatively simple ticker tape to inform developers of any SCM activities and allows chat messages to be initiated through its interface. Celine uses the concept of hierarchical workspaces to enable information sharing to scale to very large projects. Both tools provide a more detailed development context to the developers and integrate additional information on the nature and size of (only) direct conflicts.

FASTDash [6] is another awareness system, focusing particularly on agile teams. It uses different traffic metaphors (e.g., traffic signals, stop signs) to indicate direct conflicts and presents this information to individuals in their IDE as well as through a centralized display for teams.

Compared to Palantír, Elvin, Celine, and FASTDash do not address indirect conflicts.

In addition to Palantír, TUKAN [49], CollabVS [17], and WeCode [25] are three other systems that address indirect conflicts. TUKAN uses a program's def-use graph to identify changes that are semantically close and presents this information through icons following a weather metaphor. If two developers modify closely related code elements, a lightning symbol is used to warn them about the potential conflict, with several other icons representing various other situations (a sun is used to represent changes far from each other, for instance). CollabVS works with Visual Studio and uses call graphs to identify conflicting instances, similarly to Palantír. WeCode uses a shadow repository to continuously merge ongoing compiled changes from private workspaces to produce an executable that it tests for integration problems. Of these three, CollabVS is the only tool that has been evaluated, through a small laboratory experiment. The work reported in this paper, then, distinguishes itself from the CollabVS work with an extensive and in-depth evaluation.

10 Conclusions and Future Work

Current team development using SCM systems exhibits a fundamental tension between the need for individual developers to work in isolated workspaces and the need for the overall team to maintain control over the integration of the individual changes into the overall system. To better address this tension, we have developed Palantír, a workspace awareness tool that deliberately breaks workspace

isolation by informing developers of ongoing parallel changes being worked on by other developers. By continuously sharing who modifies which artifacts, the magnitude of the changes, and their impact, Palantír complements current automated SCM policies with the capability of early human intervention regarding any potential conflicts that may emerge.

Our empirical evaluations conclusively show that the use of Palantír: 1) leads to early detection and resolution of a larger number of conflicts as compared to when it is not used, 2) results in fewer conflicts that are unresolved in the code base that is ultimately checked in, and 3) involves a reasonable overhead in terms of the extra effort that must be expended. We also present a detailed qualitative analysis that sheds light on how users reacted to information provided by Palantír, the strategies they followed in coordinating their work, and how awareness affected team dynamics.

Our future work involves a number of different directions. First, we recognize that our work in detecting indirect conflicts merely scratches the surface. The next challenge is to extend the range of indirect conflicts that can be addressed including semantic conflicts.

Second, we wish to perform longitudinal studies of Palantír in use. Our current evaluations are limited in not accounting for the learning effect that takes place when using the tool for extended periods of time on a real system that the developers are intimately familiar with. Factors such as the intuition to differentiate significant conflicts from more innocuous ones, the insight into which developers are more prone to insert problematic changes, and overall team dynamics and emergent coordination strategies can only be understood when Palantír is used in a live project over a longer period of time.

Finally, we are interested in leveraging our experience with Palantír to address collaboration between teams as well as within teams, especially in geographically distributed projects that may or may not involve outsourcing. In such cases, changes generally do not become visible until official release dates of the component(s) for which a team is responsible. Furthermore, the nature of these kinds of projects is such that the kinds of direct interactions encouraged by Palantír likely will need to be adjusted to involve very different notification and resolution strategies.

Acknowledgments

The authors thank Gerald Bortis, Peggy Lin, Zahra Noroozi, Roger Ripley, Ryan Yasui for their contributions to the Palantír project. Effort partially funded by the US National Science Foundation (NSF) under grant numbers CCR-0093489, IIS-0205724, IIS-0534775, IIS-0414698, IIS-1111446, and IIS-0808783. Effort also supported by an IBM Eclipse Innovation grants and an IBM Technology Fellowship.

References

[1] L.G. Alan, "The Evolution of a Source Code Control System," *SIGSOFT Software Eng. Notes,* vol. 3, no. 5, pp. 122-125, 1978.

[2] Apache, *Google Wave,* https://wave.google.com/wave/, 2012.

[3] W. Appelt, "WWW Based Collaboration with the BSCW System," *Proc. Conf. Current Trends in Theory and Informatics,* pp. 66-78, 1999.

[4] U. Asklund and B. Magnusson, "Support for Consistent Merge," *Proc. 10th Int'l Workshop Software Configuration Management: New Practices, New Challenges and New Boundaries,* pp. 27-32, 2001.

[5] B. Berliner, "CVS II: Parallelizing Software Development," *Proc. USENIX Winter 1990 Technical Conf.,* pp. 341-352, 1990.

[6] J. Biehl et al., "FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams," *Proc. SIGCHI Conf. Human Factors in Computing Systems,* pp. 1313-1322, 2007.

[7] E. Bradner and G. Mark, "Why Distance Matters: Effects on Cooperation, Persuasion and Deception," *Proc. ACM Conf. Computer Supported Cooperative Work,* pp. 226-235, 2002.

[8] M. Cataldo and J. Herbsleb, "Communication Networks in Geographically Distributed Software Development," *Proc. Conf. Computer Supported Cooperative Work,* pp. 579-588, 2008.

[9] M. Cataldo et al., "Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools," *Proc. ACM Conf. Computer Supported Cooperative Work,* pp. 353-362, 2006.

[10] R. Conradi and B. Westfechtel, "Version Models for Software Configuration Management," *ACM Computing Surveys,* vol. 30, no. 2, pp. 232-282, 1998.

[11] B. Curtis et al., "A Field Study of the Software Design Process for Large Systems," *Comm. ACM,* vol. 31, no. 11, pp. 1268-1287, 1988.

[12] S. Dart, "Concepts in Configuration Management Systems," *Proc. Third Int'l Workshop Software Configuration Management,* pp. 1-18, 1991.

[13] C.R.B. de Souza and D. Redmiles, "An Empirical Study of Software Developers' Management of Dependencies and Changes," *Proc. 30th Int'l Conf. Software Eng.,* pp. 241-250, 2008.

[14] C.R.B. de Souza et al., "How a Good Software Practice Thwarts Collaboration: The Multiple Roles of APIs in Software Development," *Proc. Int'l Symp. Foundations of Software Eng.,* pp. 22-230, 2004.

[15] C.R.B. de Souza et al., "'Breaking the Code', Moving between Private and Public Work in Collaborative Software Development," *Proc. Int'l Conf. Supporting Group Work,* pp. 105-114, 2003.

[16] P. Dewan and R. Choudhary, "A High-Level and Flexible Framework for Implementing Multi-User Interfaces," *ACM Trans. Information Systems,* vol. 10, no. 4, pp. 345-380, 1992.

[17] P. Dewan and R. Hegde, "Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development," *Proc. Conf. European Computer Supported Cooperative Work,* pp. 159-178, 2007.

[18] P. Dourish and V. Bellotti, "Awareness and Coordination in Shared Workspaces," *Proc. ACM Conf. Computer-Supported Cooperative Work,* pp. 107-114, 1992.

[19] Eclipse.org, Eclipse, http://www.eclipse.org/, 2012.

[20] Eclipse.org, CVS Support, http://www.eclipse.org/eclipse/platform-cvs/, 2012.

[21] J. Estublier and S. Garcia, "Process Model and Awareness in SCM," *Proc. 12th Int'l Workshop Software Configuration Management,* pp. 69-84, 2005.

[22] G. Fitzpatrick et al., "Supporting Public Availability and Accessibility with Elvin: Experiences and Reflections," *Proc. ACM Conf. Computer Supported Cooperative Work,* pp. 447-474, 2002.

[23] R.E. Grinter, "Recomposition: Putting It All Back Together Again," *Proc. ACM Conf. Computer Supported Cooperative Work,* pp. 393-402, 1998.

[24] R.E. Grinter, "Supporting Articulation Work Using Software Configuration Management Systems," *Proc. ACM Conf. Computer Supported Cooperative Work,* pp. 447-465, 1996.

[25] M.L. Guimarães and A. Rito-Silva, "Towards Real-Time Integration," *Proc. ICSE Workshop Cooperative and Human Aspects of Software Eng.,* pp. 56-63, 2010.

[26] C. Gutwin and S. Greenberg, "Workspace Awareness for Groupware," *Proc. Conf. Companion on Human Factors in Computing Systems,* pp. 208-209, 1996.

[27] C. Gutwin et al., "Group Awareness in Distributed Software Development," *Proc. ACM Conf. Computer Supported Cooperative Work,* pp. 72-81, 2004.

[28] C. Heath and P. Luff, "Collaboration and Control: Crisis Management and Multimedia Technology in London Underground Line Control Rooms," *Computer Supported Cooperative Work,* vol. 1, no. 12, pp. 69-94, 1992.

[29] J. Herbsleb et al., "Introducing Instant Messaging and Chat in the Workplace," *Proc. SIGCHI Conf. Human Factors in Computing Systems: Changing Our World, Changing Ourselves,* pp. 171-178, 2002.

[30] J.D. Herbsleb and R.E. Grinter, "Architectures, Coordination, and Distance: Conway's Law and Beyond," *IEEE Software,* vol. 16, no. 5, pp. 63-70, Sept./Oct. 1999.

[31] J.D. Herbsleb et al., "Distance, Dependencies, and Delay in a Global Collaboration," *Proc. ACM Conf. Computer Supported Cooperative Work,* pp. 319-328, 2000.

[32] S. Jonathan et al., "Asking and Answering Questions during a Programming Change Task," *IEEE Trans. Software Eng.,* vol. 34, no. 4, pp. 434-451, July/Aug. 2008.

[33] I. Kwan et al., "Does Socio-Technical Congruence Have an Effect on Software Build Success? A Study of Coordination in a Software Project," *IEEE Trans. Software Eng.,* vol. 37, no. 3, pp. 307-324, May/June 2011.

[34] A. Lee et al., "NYNEX Portholes: Initial User Reactions and Redesign Implications," *Proc. ACM SIGGROUP Conf. Supporting Group Work: The Integration Challenge,* pp. 385-394, 1997.

[35] R.E. Mayer, "From Novice to Expert," *Handbook of Human-Computer Interaction,* M.G. Helander, et al., eds., second ed., pp. 781-795, Elsevier, 1988.

[36] L.J. McGuffin and G. Olson, "ShrEdit: A Shared Electronic Workspace," Technical Report #45, Cognitive Science and Machine Intelligence Laboratory, Univ. of Michigan, 1992.

[37] T. Mens, "A State-of-the-Art Survey on Software Merging," *IEEE Trans. Software Eng.,* vol. 28, no. 5, pp. 449-462, May 2002.

[38] P. Molli, "COO-Transactions: Supporting Cooperative Work.," *Proc. Seventh Int'l Workshop Software Configuration Management,* pp. 128-141, 1997.

[39] C. O'Reilly et al., "Improving Conflict Detection in Optimistic Concurrency Control Models," *Proc. 11th Int'l Workshop Software Configuration Management,* pp. 191-205, 2003.

[40] I. Omoronyia et al., "A Review of Awareness in Distributed Collaborative Software Engineering," *Software Practice and Experience,* vol. 40, no. 12, pp. 1107-1133, 2010.

[41] PostgreSQL, http://www.postgresql.org/, 2012.

[42] R. Ripley, "Improving the Practical Usability of Palantír," master's, Informatics, Univ. of California, Irvine, 2004.

[43] M.J. Rochkind, "The Source Code Control System," *IEEE Trans. Software Eng.,* vol. 1, no. 4, pp. 364-370, Dec. 1975.

[44] A. Sarma et al., "Towards Supporting Awareness of Indirect Conflicts across Software Configuration Management Workspaces," *Proc. Conf. Automated Software Eng.,* pp. 94-103, 2007.

[45] A. Sarma et al., "Palantír: Raising Awareness among Configuration Management Workspaces," *Proc. Int'l Conf. Software Eng.,* pp. 444-454, 2003.

[46] A. Sarma et al., "Empirical Evidence of the Benefits of Workspace Awareness in Software Configuration Management," *Proc. ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 113-123, 2008.

[47] A. Sarma and A. van der Hoek, "A Conflict Detected Earlier Is a Conflict Resolved Easier," *Proc. Workshop Open Source Software Eng.,* pp. 82-86, 2004.

[48] A. Schröter, "Predicting Build Outcome with Developer Interaction in Jazz," *Proc. Int'l Conf. Software Eng.* vol. 2, pp. 511-512, 2010.

[49] T. Schümmer and J.M. Haake, "Supporting Distributed Software Development by Modes of Collaboration," *Proc. Seventh European Conf. Computer Supported Cooperative Work,* pp. 79-98, 2001.

[50] W.R. Shadish et al., *Experimental and Quasi-Experimental Designs for Generalized Causal Inference,* first ed. Houghton Mifflin Company, 2001.

[51] Dependency Finder, http://depfind.sourceforge.net/, 2012.

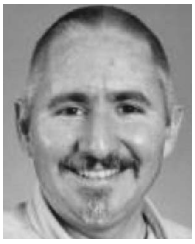
[52] A.C. Strauss and J. Corbin, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory,* second ed. Sage Publications, Inc., 1998.

[53] *Subclipse: Eclipse Team Provider Plug-In Providing Support for Subversion within the Eclipse IDE,* http://subclipse.tigris.org/, 2012.

[54] Tigris.org, Subversion, http://subversion.tigris.org/, 2012.

[55] G. Valetto et al., "Using Software Repositories to Investigate Socio-Technical Congruence in Development Projects," *Proc. Workshop Mining Software Repositories,* p. 27, 2007.

Anita Sarma received the PhD degree from the Department of Informatics in the Donald Bren School of Information and Computer Sciences at the University of California, Irvine (2007). She is an assistant professor in the Department of Computer Science and Engineering at the University of Nebraska-Lincoln. Before this she was a postdoctoral fellow at Carnegie Mellon University, School of Computer Science. Her research focuses on understanding and facilitating coordination in distributed work, which involves both software development as well as other nonroutine intellectual team work. Her work lies at the intersection of Software Engineering and Computer Supported Cooperative Work (CSCW). She directs the Interaction Design and Coordination Laboratory, its research focusing on understanding and advancing the roles of innovative interaction designs, collaboration, and empirical understanding in software development. She is a member of the IEEE.



David F. Redmiles received the BS degree in mathematics and computer science in 1980 and the MS degree in computer science in 1982 from the American University, Washington, DC, and the PhD degree in computer science from the University of Colorado, Boulder, in 1992. From 1979 to 1987, he worked at the US National Institute of Standards and Technology (formerly the National Bureau of Standards), Gaithersburg, Maryland. From 1992 to 1994, he did postdoctoral work at the University of Colorado, Boulder. Since 1994, he has been at the University of California, Irvine, where he is currently working as a professor in the Department of Informatics in the Donald Bren School of Information and Computer Sciences. He formerly chaired that department from 2004 to 2011. During this period there was great expansion of the faculty, facilities, and degree programs. He has a background in software engineering, human-computer interaction, and computer-supported cooperative work and has more than 100 research publications in these areas. For the past decade, he has been researching collaborative software engineering. He is active in the IEEE/ACM Conference on Automated Software Engineering, serving on the steering committee and organizing the 2005 conference as general chair. That research community designated him Fellow of Automated Software Engineering in 2009 and in 2010 awarded him and his coauthors the first Most Influential Paper Award for their 1996 paper on software design environments. He has also organized a number of panels and workshops held in conjunction with the ACM Conference on Human Factors in Computing Systems, the ACM Conference on Computer-Supported Cooperative Work, and the International Conference on Software Engineering. He is a member of the IEEE.

André van der Hoek received the joint BS and MS degrees in business-oriented computer science from the Erasmus University Rotterdam, The Netherlands, and the PhD degree in computer science from the University of Colorado at Boulder. He is a professor in the Department of Informatics of the Donald Bren School of Information and Computer Sciences and a faculty member of the Institute for Software Research, both at the University of California, Irvine. He directs the Software Design and Collaboration Laboratory, its research focusing on understanding and advancing the roles of design, collaboration, and education in software development. He has authored or coauthored more than 80 peer-reviewed journal and conference publications, and in 2006 was a recipient of an ACM SIGSOFT Distinguished Paper Award. He is a coauthor of the 2005 Configuration Management Impact Report as well as the 2007 Futures of Software Engineering Report on Software Design and Architecture. He has served on numerous international program committees, is a member of the editorial board of the *ACM Transactions on Software Engineering and Methodology*, and was program chair of the 2010 ACM SIGSOFT International Symposium on the Foundations of Software Engineering. In 2009, he was a recipient of the Premier Award for Excellence in Engineering Education Courseware. He is the principal designer of the BS in Informatics at UC Irvine and was honored in 2005 as UC Irvine Professor of the Year for his outstanding and innovative educational contributions. He is a member of the IEEE.