Towards Supporting Awareness of Indirect Conflicts across Software Configuration Management Workspaces

Anita Sarma, Gerald Bortis, and André van der Hoek
University of California, Irvine
Department of Informatics
Irvine, CA 92697-3440 USA
+1 (949) 824 6326

{asarma,gbortis,andre}@ics.uci.edu

ABSTRACT

Workspace awareness techniques have been proposed to enhance the effectiveness of software configuration management systems in coordinating parallel work. These techniques share information regarding ongoing changes, so potential conflicts can be detected during development, instead of when changes are completed and committed to a repository. To date, however, workspace awareness techniques only address *direct conflicts*, which arise due to concurrent changes to the same artifact, but are unable to support *indirect conflicts*, which arise due to ongoing changes in one artifact affecting concurrent changes in another artifact. In this paper, we present a new, cross-workspace awareness technique that supports one particular kind of indirect conflict, namely those indirect conflicts caused by changes to class signatures. We introduce our approach, discuss its implementation in our workspace awareness tool Palantír, illustrate its potential through two pilot studies, and lay out how to generalize the technique to a broader set of indirect conflicts.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments – *Programmer workbench*. D.2.7 [**Software Engineering**]: Distribution, Maintenance, Enhancement – *Version control*. D.2.9 [**Software Engineering**]: Management – *Software configuration management*.

General Terms

Design, Management, Experimentation, Human Factors.

Keywords

Configuration management, software configuration management, awareness, direct conflicts, indirect conflicts.

1. INTRODUCTION

With the growing maturity and increasingly powerful functionality of Software Configuration Management (SCM) systems, parallel development has become a norm rather than an exception. It is rare to find a project in which strict locking is practiced. Optimistically checking out artifacts and merging any conflicts that arise is a far more popular approach. But, this optimistic approach does have a price: the cost of conflict resolution. It has been shown that parallel development frequently leads to conflicts, which anecdotally and empirically are known to sometimes be trivial to resolve, but at other times involve a significant and time-consuming exercise [11, 28].

It is useful to group the conflicts that may occur in parallel development in two classes: *direct conflicts* and *indirect conflicts*. Direct conflicts are caused by concurrent changes to the same artifact, for instance, when two or more developers unknowingly alter the same Java file at the same time in their respective workspaces. Indirect conflicts are caused by changes in one artifact that affect concurrent changes in another artifact. This may occur when one developer, working in their private workspace, alters a Java interface file that another developer just imported and started referring to from a Java class they are editing in their respective workspace. But beyond this straightforward, syntactic example, numerous kinds of indirect conflicts exist that may be of a more intricate, often semantic nature.

Both direct and indirect conflicts are generally discovered at a time later than when they are actually introduced. That is, between the time a conflict begins to emerge (e.g., a developer starts to change a file that another developer is already in the process of changing, a developer inserts references to Java methods that another developer already removed) and the time it is actually detected, significant time passes during which the conflict may grow from small and innocuous to large and complex to resolve. Direct conflicts are typically detected at the time of check-in or when the developer synchronizes their workspace right before checking in. Indirect conflicts, however, may slip by undetected and reveal themselves only during the build process, testing phase, or, worse, after the product is already in the field. This delay lies at the core of why it can be so difficult to resolve conflicts, both direct and indirect: a developer must go back in time, understand both of the conflicting changes in full, and find a way to meaningfully combine them [9, 16].

In an attempt to detect conflicts earlier, and thereby minimize the impact they have, recent research in the SCM discipline has focused on adopting workspace awareness techniques to notify developers of conflicts as they emerge (e.g., [7, 11, 32]). These techniques operate by sharing relevant information regarding who is changing which artifacts – *while the changes are in progress*. In effect, SCM systems that include workspace awareness "spy" on ongoing efforts and inform pertinent developers when their efforts seem to be conflicting. The intended effect is for developers to respond proactively and early. Responses may be

ASE'07, November 5–9, 2007, Atlanta, Georgia, USA.

to call the other party and discuss, to hold off on one's changes until the other developer has checked in theirs, to use the SCM system to look at the other person's workspace to determine the extent of the conflict, and other such human-inspired approaches to coordination. In so doing, a conflict can be caught before it grows too large, and often may be avoided altogether, since the developer may reconsider before editing an artifact they know someone else is modifying at that time.

This paper presents a new cross-workspace awareness technique that we designed specifically to begin addressing *indirect conflicts*. To date, SCM workspace awareness techniques address *direct conflicts* only, in effect aiming to avoid merge conflicts. But indirect conflicts are as important, if not more so. They can arise in many different forms, ranging from straightforward syntactic conflicts to intricate semantic conflicts. In this paper we provide a foundation for research into indirect conflicts by demonstrating how one particular kind of indirect conflict can be addressed, but providing an architectural approach that we believe is generic and can be adapted to address other kinds of indirect conflicts. This represents a modest first step, but makes a key conceptual leap in moving from just informing developers about ongoing changes (as is the case for how direct conflicts are addressed) to incorporating cross-workspace analysis (as is necessary to determine whether an indirect conflict exists).

We specifically address indirect conflicts that stem from changes in class signatures. When one developer modifies the signature of a class while another developer simultaneously develops code that uses this class, the potential for indirect conflict arises. Our technique informs developers when this happens. It does so by sharing details of relevant changes across workspaces, summarizing these changes in a cache, and analyzing the changes present in the cache for the potential existence of indirect conflicts. If such conflicts are present, notifications are distributed to inform developers through peripheral presentation of the awareness information in the software development environment.

We evaluated our work with two pilot studies, comparing our approach to one that just notifies developers of direct conflicts and to one that does not employ awareness at all. Our results provide preliminary evidence that our approach is more effective than identifying direct conflicts only (or no conflicts at all, as in the case of no awareness) and supports early detection of indirect conflicts as well as effective resolution of the conflicts well before the code is committed.

The remainder of this paper is organized as follows. In Section 2, we introduce background material. Section 3 presents a motivating example, Section 4 our high-level approach, and Section 5 our implementation. Section 6 discusses our evaluation with two pilot studies. Section 7 presents related work and Section 8 concludes with an outlook at our future work.

2. BACKGROUND

Although numerous software configuration management systems with features of many different kinds are available, they all employ essentially one of two strategies to coordinate the work of developers. *Pessimistic* SCM systems [33] mandate that developers lock an artifact before they make changes, prohibiting others from making changes to that artifact in parallel. *Optimistic* SCM systems [5] explicitly encourage developers to modify the same artifact(s) in parallel, choosing to focus on providing developers with merge tools to help resolve conflicts that may occur.

This paper concerns optimistic SCM systems. A key assumption governing the functioning of these kinds of systems is that relatively few parallel changes lead to conflicts. They, thus, trade off the possibility of conflicts that must be resolved for a development process that can progress more rapidly, since developers no longer have to wait for locks to be released.

In practice, this approach indeed works well most of the time [12, 13] and parallel development is widely practiced. Yet, conflicts do occur and can have detrimental effects. Situations have been reported where developers rush to be the first person to check in their changes, so they do not have to be the one dealing with the merges necessary to resolve conflicts that will occur [15]. It has also been reported that the more parallel development takes place, the more bugs appear in the code [28]. And, of course, every conflict involves overhead in terms of the time and effort that must be invested to resolve it. While careful task assignments and distribution of responsibilities over multiple developers are important steps in partitioning the overall work, they are in and of themselves insufficient to guarantee that changes will not overlap or otherwise conflict with other ongoing changes.

Groups of developers have responded by establishing their own, informal conventions to coordinate their mutual efforts. They may send e-mail informing each other of the changes they have made and the effect these changes are expected to have on other's work [8]. They also have been reported to leverage Instant Messaging to continuously keep each other up to date and coordinate ongoing changes [21], or querying the SCM repository to find out who is making changes where in the code [15]. To date, however, these practices are entirely developer driven and have little automated support from the SCM system for developers to gather the information they need. As a result, these informal practices are not as effective as they could be and not employed as often as they should be.

It is no surprise, then, that an overarching goal in the SCM literature has been to try and better support these informal practices so fewer conflicts arise and those that do arise can be recognized and addressed before they grow too large and become difficult to handle. A particularly promising approach revolves around the notion of *awareness*. Awareness is characterized as "an understanding of the activities of others, which provides a context for your own activity" [10]. It was originally introduced as a passive and subconscious form of information gathering, but has since been explored in more proactive and conscious settings with tools attempting to provoke awareness by presenting users with specific kinds of information [20]. In the case of SCM systems and the goal to reduce conflicts, this means capabilities that highlight ongoing parallel work efforts and indicate the potential presence of conflicts among these efforts.

The way in which awareness has been brought into SCM systems has been through workspace awareness techniques. These techniques, typically embedded in plug-ins to the SCM system, hook into the development environment to collect the necessary information, share this information across workspaces, and provide some kind of visualization to inform developers of the potential conflicts that exist. To date, the information that is shared pertains to which developer is changing which artifacts

[7], but also may include information regarding the "size" of changes [25] or the lines of code that are being changed [23].

Through careful and usually peripheral integration of the presentation of the resulting "awareness information" into the day-to-day development environment, the goal is to alter the behavior of the developers such that when they notice a potential conflict emerging, they take proactive steps and take those steps early. This may involve calling or Instant Messaging the other party, walking over to discuss and reassign the 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 look at the changes in the other workspace [7, 11]and deciding that it is ok to continue, and many other courses of action. This, indeed, means that a small amount of effort must be expended now, though it is anticipated that this extra effort is "gained back" by avoiding a much larger issue later on.

To date, existing workspace awareness techniques in SCM identify direct conflicts only, as their goal has always been to reduce the number and size of merge conflicts. While simply knowing which developer changes which artifacts may be good enough for an individual developer to deduce the potential presence of an indirect conflict, in general the structure and the relationships embedded in the source code are not transparent, making it difficult for developers to draw solid conclusions.

Many different kinds of indirect conflicts exist, ranging from purely syntactic to entirely semantic in nature. For example, conflicts arising from changes in class signatures are syntactic. A semantic indirect conflict could arise when two developers modify the execution times of different components in a multi-threaded system, with the changes fine in isolation, but the combination leading to synchronization problems. Not all indirect conflicts can be detected automatically or accurately diagnosed at all times. However, as long as automated analyses techniques help in identifying where changes exert what kind of influence, awareness techniques may be able to assist developers in detecting potential conflicts. The approach taken in this paper specifically focuses on addressing indirect conflicts from changes in class signatures, but we believe that the strategy we present generalizes to other kinds of indirect conflicts – as long as suitable analysis are available, adjustments are made in the internal data structures, and conflict visualizations are updated as needed.

Finally, we note that not every conflict that is identified as arising between changes in a pair of workspaces will eventually be a "real" conflict. A developer may simply be experimenting with the code, rolling back their changes shortly. It could also be that they made an honest mistake that they recognize later. Therefore, any conflicts that are identified by a workspace awareness technique should be considered *potential* conflicts and further human examination, interpretation, and communication will still be needed. The reward for this investment now, however, is the avoidance of a potentially much larger, real problem later on.

3. MOTIVATING EXAMPLE

Throughout this paper, we use a motivating example to illustrate the need for a cross-workspace awareness technique that supports indirect conflicts caused by changes in class signatures. The example involves two developers, Ellen and Pete, working on some hypothetical software to process credit cards. Ellen is responsible for dealing with a few requested feature enhancements to an existing class *Payment.java*. Pete has to implement a new class *CreditCard.java*, which will have to keep track of the payments made on a credit card.

Upon Pete's arrival in the morning, he checks out for reading only the class *Payment.java* and, after studying its code for a while, he begins his work on the new class *CreditCard.java*. Ellen arrives a bit later than Pete and begins her morning by reviewing the implementation of *Payment.java* that she did yesterday. She does not like how she has put a lot of logic in its constructor and decides to move parts of the initialization code to a new method called *init*; this is to prepare the code for future changes she has planned. She is keenly aware of the significance of this change, documenting the code with detailed comments.

Meanwhile, Pete implements *CreditCard-.java*. In the process, he makes, as he understands it, appropriate calls to new instances of the class *Payment.java*. Since he studied its code, and noticed the initialization code is in the constructor, he knows he does not have to call any initialization routine(s).

Pete finishes just before lunch, as does Ellen. They both check in their changes to the SCM system, which does not flag a problem since the changes are to different artifacts. Clearly, a conflict has been introduced in the code that was not detected by the SCM system. Since Ellen did not change the signature of the constructor, the build system also does not find the conflict, since the combined code compiles just fine.

Under normal circumstances, Ellen would synchronize her work with the latest state of the repository and test the combined system before checking in her changes. In all likelihood, she would have detected the issue. However, three problems persist:

1. Depending on the nature and complexity of the changes at hand, and what exactly the test cases cover, Ellen may or may not find the issue. If she does not find it, it could enter production code and actually be delivered to customers.
2. If she does find the conflict, Ellen still has to scour the changes carefully to find out where the exact problem is. In particular, she has to read and interpret Pete's changes and relate them to hers. While this is trivial in this example, in general changes are of a much more complex nature.
3. Finally, after she finds the source of the conflict, Ellen has to devise a solution that keeps the intent of both changes in tact. Again, it is straightforward to do so for the example presented (Ellen needs to add a method call to *init* in Pete's code). When changes are complex and highly interrelated, as often the case, resolution will be more complex.

It is preferable that Pete and Ellen are able to detect the conflict as soon as it emerges. Then, Pete could have initiated a conversation with Ellen, as a result of which he could insert the appropriate call to *init* right away. Alternatively, he could ask Ellen to do so for him once she finished her changes.

In this example, existing workspace awareness tools that address direct conflicts may have helped some. They would have indicated that Ellen was changing *Payment.java* and Pete *CreditCard.java*. But it could have only been Pete, who knew he started using the class *Payment.java*, who could have noticed

that he maybe should talk to Ellen; Ellen could not have known. And even then, all Pete knows is that Ellen is changing *Payment.java*, which could be for many different reasons and involving many different parts of the code. Nothing prompts him that she is changing a part of the code that is relevant to him.

The example is necessarily simple, but it is representative of the indirect conflicts that concern this paper, namely those resulting from changes to class signatures. While such changes are a well-understood part of programming, seem simple to address when a problem emerges, and have many conventions and best practices that aim to avoid these kinds of conflicts altogether, prior work nonetheless identifies them as a major source of conflicts, direct and indirect [9, 16]. The critical role that class signatures have in being boundary objects lies at the heart of this problem [17, 26]. The ability to detect potential conflicts early, therefore, can lead to significant improvements in development practices.

4. APPROACH

Compared to approaches for direct conflicts, the critical hurdle to overcome in addressing indirect conflicts is that some form of cross-workspace analysis is necessary to relate concurrent, ongoing changes in different workspaces. With direct conflicts, it is sufficient for each workspace to broadcast which artifacts are changing. The visualizations can display this information, and any overlap in workspaces changing the same artifact is immediately visible. This approach, however, does not work for indirect conflicts, because they fundamentally concern the *relationship* between non-overlapping changes.

Any approach wishing to address indirect conflicts (of whichever kind) must bring information regarding changes in different workspaces together, so their combinations can be examined. It is important that this is done in accordance with several general objectives to be met by any workspace awareness technique: (1) unobtrusiveness, so developers are not detracted from their day-to-day coding activities, (2) scalability, so the solution supports a large number of developers modifying a large number of artifacts, (3) flexibility, so different analyses and visualizations are easily integrated, and (4) configurability, so users are provided control over the behavior of the awareness technique [18, 19].

Our approach to addressing indirect conflicts builds upon several strategies that have been successfully employed in existing awareness techniques. Specifically, key strategies that we adopt are a push-based event model [14], peripheral visualization via careful integration of the awareness information in the user interface [19], and display of relevant conflicts only [20]. For direct conflicts, these strategies combine into a four-step process of collecting, distributing, filtering, and visualizing awareness information. Adjusting this process to indirect conflicts requires two new steps: (1) cross-workspace analysis of ongoing changes and (2) informing other workspaces when indirect conflicts are found. In Figure 1, the resulting six-step process is compared to the original four-step process. A key difference is that in four-step process events remain "independent", flowing up through the steps separately. In the six-step process, however, events are related during the analysis at one of the workspaces, with one or more new events redistributed if one or more indirect conflicts are found.

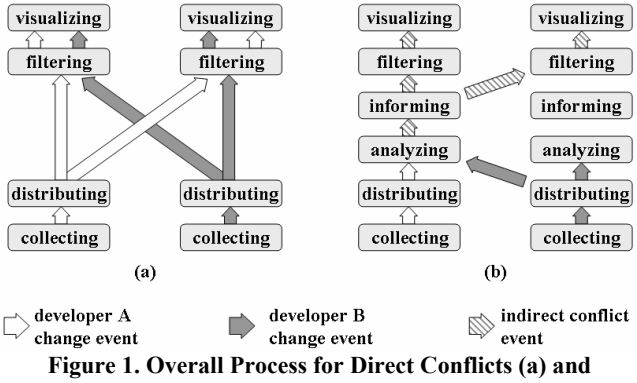

Figure 1. Overall Process for Direct Conflicts (a) and Indirect Conflicts (b).

Below, we detail our approach, as applied to the problem of indirect conflicts emerging from changes in class signatures.

Collecting. The first step is to collect the information necessary for the cross-workspace analysis. Two issues must be addressed: what information is collected and how often is it collected? With respect to the first issue, we capture: (1) changes in the name, parameters, return value, and scope of public methods, whether specified by an interface or class, (2) addition and deletion of classes, interfaces, and public methods, (3) changes in the extends and implements relationship among classes and interfaces, and (4) changes in the uses relationships of classes, interfaces, and methods. This provides us with all of the information pertaining to changes in what a class has to offer to other classes and changes in how these other classes "use" the class.[1]

With respect to the second issue (how often is the information collected), we adopt an approach that continuously monitors the editing process. We track any of the aforementioned changes to any artifact immediately, resulting in an up-to-date picture of ongoing changes at all times, which is necessary to support early detection of conflicts. Waiting until a change is complete and checked in would yield information that is "after the fact". This choice of continuous monitoring is in line with other awareness techniques currently in existence.

Distributing. Once the information is collected, some of it has to be shared with other workspaces. The choice here is whether to distribute the information concerning changes in what a class has to offer or information concerning changes in how a class is used; it is not necessary to distribute both. We chose the former, largely because of an intuition that this leads to fewer events. Especially in the later stages of a project, when the structure of the code has been largely established, we believe that changes in uses relationships will be more frequent than changes in class signatures.[2]

We package the information regarding changes in what a class has to offer as diff's that are sent to other workspaces through a push-based event service. Since each diff only has to be sent to a select set of workspaces, namely those in which the class (or interface) is used, we can leverage the subscription facility of

[1] This is not completely accurate, since public member variables are also part of a class signature. Our implementation does not handle them at this time, but can be easily extended.

[2] An actual study should be performed to validate this intuition. If it turns out to be the inverse, the locus of analysis can simply be changed by inverting the flow of events.

the event infrastructure to route events to only those workspaces.

Finally, we employ a special-purpose XML diff format instead of a generic, line-based textual diff format. This creates a precise context and minimizes the amount of processing needed on the receiving side. The DTD defining this diff format is tedious but straightforward, enumerating all possible types of changes.

Analyzing. The analysis step is at the heart of our approach, as it is here that information regarding changes made in the local workspace and in the remote workspaces is brought together to determine the presence or absence of potential indirect conflicts. Several key considerations must be made. First, because changes are incremental and analysis can be expensive, it is important to maintain a cache that abstracts the state of the local workspace, as well as the state of remote workspaces through summarizing the diff's received from those workspaces. With each local or remote change, we update this cache. The local workspace part of the cache captures the dependencies among all of the code elements in the workspace, so that determining whether a remote change conflicts with this state becomes a matter of looking up the remotely changed artifact in the local cache and examining its dependencies, both forwards and backwards.

A second consideration is that diff's can accumulate. In a typical setting, multiple diff's capture the sequence of changes a developer makes to an artifact. Some of those diff's may negate parts of previous diff's, for instance, when a developer undoes their earlier addition of some set of methods. Upon receipt of a diff, we therefore analyze it and cull any extraneous (parts of) other diff's that are already in the cache to provide a minimal yet accurate summary of the remote changes. (See also "Informing".)

A third consideration regards the analysis algorithm to be used. Many dependency analysis algorithms are available [3, 4] and it is possible to obtain results at various levels of precision. This, in turn, makes it possible to provide more detailed information than just a statement that changes in one artifact indirectly conflict with changes in another artifact. But, there is a cost, namely the cost of analysis. Particularly, detailed algorithms tend to not be incremental and require re-analysis of the entire system – an intrusive and possibly prohibitive feature. Since we first want to understand whether a straightforward annotation indicating the presence of an indirect conflict is effective before embarking on in-depth studies of all sorts of indirect conflicts, we opted not to include a more sophisticated analysis at this point.

When our technique should perform the analysis is the final consideration that we address for this step. As with the question of "when to collect the information", we answer "immediately", because it is important for the awareness information provided to the developers to be as up-to-date as possible. As long as this can be done without extreme use of resources, we believe this is the right choice. Of course, if performance does become an issue because too many events arrive shortly after one another, we can change our technique to perform the analysis every few minutes. This would not detrimentally influence the effectiveness of the technique as coding is still a relatively slow activity.

Informing. The fourth step, informing, is straightforward. Once the analysis step has completed, and indirect conflicts have been found, information regarding these conflicts is distributed to the originating workspace, as well as to any other workspaces where one or more of the involved artifacts is present. The event service is once again leveraged to appropriately route these events.

A few observations must be made with respect to this step. First, during the analysis step, if (parts of) existing diff's are removed from the cache due to a later diff that negates previous changes, this means that events that were sent earlier are no longer valid. Appropriate events that undo those events are created and sent.

Second, the choice of also sending the indirect conflict events to other workspaces is deliberate. While these workspaces are not involved in the indirect conflict, it is still useful for the developers to have access to this information because of future changes in which they may be involved. They, for instance, will be able to anticipate that a particularly complex modification they have planned that includes changes to one of the involved artifacts would create a situation in which three or more people might be engaged in an indirect conflict – a situation that is not desirable. Notifications are distributed to all workspaces in which a particular artifact is present to enable up-front planning of change activities.

Finally, we reiterate that the information that is sent identifies both the artifact that causes an indirect conflict and the artifacts that are affected by this indirect conflict. It is pertinent to inform developers about both.

Filtering. In this step, developers are presented with the option to specify filters according to which the set of events they will receive can be further reduced. In the case of direct conflicts, it is common to allow developers to select a subset of artifacts to monitor or for them to set a threshold for the size of the conflicts for which they are notified. We adapt these filters to also address indirect conflicts, allowing developers to select some minimum number of affected artifacts that must be reached before they are informed.

Visualizing. Finally, as with any approach relying on awareness, it is critical to unobtrusively, yet effectively integrate the awareness information in the development environment. Generally, this integration is performed peripherally, with insertion of subtle clues embedded in the user interface where there is a high probability that developers will notice the warnings for conflicts as they arise. We adopt this strategy as well, but leave the details of how we designed the user interface extensions to Section 5, where we discuss the implementation of our technique.

We do, however, need to make two observations here. First, the extensions to the user interface must clearly communicate which artifacts cause indirect conflicts and which artifacts are affected by indirect conflicts. Ideally, developers can readily move back and forth to examine a particular conflict and make a judgment as to whether or not it is a conflict to worry about.

Second, the issue of scalability arises. The extensions to the user interface of the development environment should be designed in such a way that, even when numerous indirect conflicts arise, they do not overburden the developer and make it impossible to find those indirect conflicts that really pertain to their ongoing work. We also return to this subject in Section 5.

Summary. We have described a six-step process that is explicitly designed to detect indirect conflicts arising from changes in class signatures. A key choice is to distribute the analysis that is

necessary. Rather than making the workspace that is responsible for a change in a class signature the locus of all computation regarding that change (which would involve a comparison to the latest state of all other workspaces), our technique broadcasts a diff and involves all of the relevant other workspaces to verify if any indirect conflicts arise as a result. The computational load is equalized, allowing our technique to scale appropriately. A second benefit is that any changes in the usage of class signatures can be dealt with locally simply by updating the local cache and re-verifying with all summarized diff's if an indirect conflict has emerged.

Our technique is purposely instantaneous, such that whenever a change is made, it is tested as to whether it represents an indirect conflict. This is critical to support the role of the human in our solution. While one small indirect conflict may not be a problem (as in the case of our motivating example in Section 3), a growing number of indirect conflicts emerging between two workspaces most certainly is. By instantaneously sharing information, it becomes possible for developers to watch trends, understand a broader context, and appropriately respond when they believe it is time to do so. In some cases, it may even be possible to avoid indirect conflicts altogether. If developers know which artifacts other developers are changing, and the extent of impact of those changes on other artifacts, then they are at least given the option to choose to work on (aspects of) tasks that do not overlap.

We implemented our technique as an extension to our existing workspace awareness tool, Palantír, which is an Eclipse plug-in that we previously implemented to address direct conflicts [31]. The original Palantír followed the four-step process presented in Figure 1a. To update Palantír to the six-step process illustrated in Figure 1b so that it also supports indirect conflicts, we had to make several changes to its architecture. Presented in Figure 2, the revised architecture needed one additional component (*Analyzer*) and updates to several existing components (highlighted through dotted lines). All other components could stay the same, particularly the *Palantír Server* and *Extractor*, both of which we had implemented using reflection, so they could support future, entirely new kinds of events.

The architecture of Palantír consists of three distinct parts. First, there are the Eclipse platform and general SCM system, shown in Figure 2 using dark gray boxes. Palantír uses these as is, obtaining the information it needs using a custom-built *Workspace Wrapper*, which collects and emits events regarding the relevant ongoing changes to all relevant artifacts. To incorporate support for indirect conflicts stemming from changes in class signatures, we had to modify this wrapper significantly. One of the main modifications concerned the events that notify other workspaces of ongoing changes to an artifact. This event was modified to not just capture who changed which artifact by how much, but to also include a diff capturing the details of the changes to a class' signature when that signature has been changed. For example, Figure 3 shows the diff that is generated when Ellen, in the scenario described in Section 2, adds the new *init* method to the class *Payment.java*.

The other main modification to the *Workspace Wrapper* was the integration of Dependency Finder [22], an open source analysis tool that we use in creating the internal cache of dependencies among artifacts. An important property of Dependency Finder is

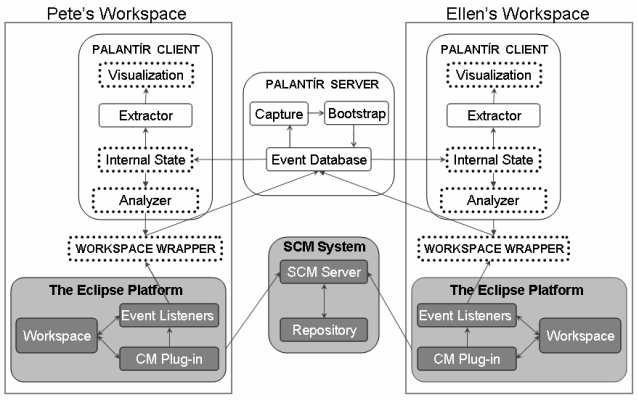

Figure 2. Revised Palantír Architecture to Support Indirect Conflicts.

that its analysis is incremental; that is, with each change, it reanalyzes only the minimal set of artifacts affected by the change. This helps to address the issues of scalability and unobtrusiveness, as frequent analyses can be performed that take up a minimal amount of resources. To further aid with these issues, the set of dependencies output by Dependency Finder is transformed by Palantír to an internal cache format that uses various hash tables to make look up of individual elements efficient and help relate changes in a local workspace to diff's from remote workspaces. Specifically, we leverage the artifact naming scheme of Palantír [31] to relate artifacts and diff's across workspaces.

The Palantír server is the second major component in the architecture of Palantír. It did not need to change, but we do mention two important roles of this component here. First, the server is responsible for routing events such that only relevant events are delivered to each of the workspaces, with relevancy defined by whether an artifact is present in the "target" workspace. Second, the server is responsible for bootstrapping. It provides historical data to workspaces that are created at a later time, so that these new workspaces have an up-to-date picture of the status of the other workspaces that have been existence for a long time. This is critical, since workspaces are opened and closed continuously.

The final major component is the Palantír client, which consists of several subcomponents. We discuss the two most important components here: the *Analyzer* and the *Visualization*. With the arrival of each diff, after the internal hash tables have been updated, the *Analyzer* component performs two analyses. First, it

```
<?xml version="1.0" encoding="UTF-8"?>
 <differences>
 <name>Payment.java</name>
  <modified-interfaces>
   <classXML>
     <new-methods>
     <declaration signature="init()"
       full-signature="store.Payment.init()"
       visibility="public" throws="" return-type="double"
       name="init()">init()</declaration>
    </new-methods>
   </classXML>
  </modified-interfaces>
 </differences>
```

Figure 3. XML Diff for Ellen's Changes to the *Payment.java* class.

examines whether the new diff negates any previous diff's, either partially or completely. In such cases, it reconciles the contents of those diff's, removes the extra diff's from the cache, and sends out a set of events that represent the reconciled state (typically via events that "undo" previously-announced indirect conflicts). Second, it uses the cache of local dependencies to check for a variety of conditions that indicate an indirect conflict, such as when a diff refers to a method that no longer exists, to a class that has a revised "extends" relationship, to a method that has a changed signature, and so on. If one or more potential indirect conflicts is found, the algorithm broadcasts an event that identifies both the artifact that causes the conflict(s) and the artifacts that are affected by it.

Note that these two analyses are also performed when the local dependency cache changes as a result of modifications in a local workspace (i.e., when Dependency Finder notices that changes have affected the dependency structure representing the code in the local workspace). In such cases, the changes are also verified against the cached set of diff's from remote workspaces.

Figure 4 shows how Palantír visualizes the results of the analyses to developers. Specifically, it shows the view of Pete who is in the process of making his changes to *CreditCard.java*. The code already makes use of a few other classes, including *Payment.java* and *Address.java*. Palantír leverages the package explorer view of Eclipse to highlight the existence of direct and indirect conflicts. Specifically, artifacts that exhibit a direct conflict are marked in the top left, with a blue triangle that grows and shrinks in size in concert with the evolving size of the conflict (this represents behavior of the original version of Palantír, which we did not change). 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 conflict. In this case, the annotation of *[S:24]* on *Address.java* indicates that 24% of the file has been modified, *[I>>]* on the same file that it is the source of an indirect conflict, and *[I<<]* on *CreditCard.java* that it is affected by an indirect conflict.

Additional information detailing the conflicts can be found in the Impact View at the bottom of the screen. Pete has selected (implicitly, by opening and editing it) *CreditCard.java*, which actually has three indirect conflicts. The first conflict, with *Address.java*, is the most serious of the three, because the changes are already in the repository and Pete can expect a build failure. The second conflict is caused by Ellen deleting a method from *Customer.java*, and is almost as serious. However, the changes are still in Ellen's workspace. Therefore, the icon in front of this conflict is the same (a bomb), but Palantír uses a different color (yellow instead of red). Finally, the third conflict represents the original issue discussed in Section 3. Ellen has added a new *init* method to *Payment.java*. Our analysis algorithm cannot identify whether this actually conflicts with Pete's changes, but the addition of a method to a dependent file might represent a potential problem. Hence, Palantír uses an exclamation mark icon to draw attention to the addition of methods. It is, of course, up to Pete to interpret the information that is presented to him. Because he studied the original code of *Payment.java* carefully, he remembers that all of the initialization code was in its constructor. He notes Ellen's addition of the *init* method, prompting him to contact her to obtain clarification of his understanding of the class.

Discussion. Our implementation to date has focused on providing awareness information. A number of auxiliary functionalities exist that we have not built yet, but that would make the overall experience richer. Users should be able to acknowledge a conflict, use the Palantír interface to browse to the relevant parts of an artifact causing a conflict (and vice versa), and look at remote artifacts side-by-side with local artifacts. None of these is technically challenging nor would they alter our approach, hence we do not expect any hurdles as we implement them in the future.

Scalability and unobtrusiveness are two of the key determinants of the effectiveness of any awareness technique, as discussed in Section 4. To examine how the new version of Palantír manages these two factors, we compare the new version to the original Palantír that supported direct conflicts only. In terms of scalability, we observe that we added one event (a reciprocal event that informs relevant workspaces of the existence of an indirect con-

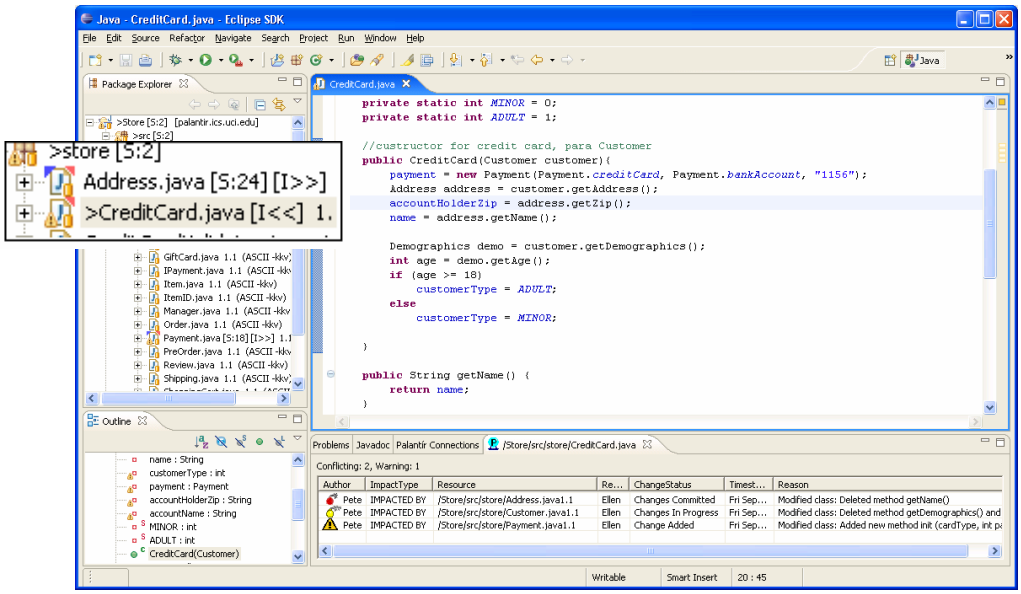

Figure 4. Visualization of Indirect Conflicts, with a Call-out of the Package Explorer.

flict) as well as an extra analysis step. The one extra event does not fundamentally alter the amount of traffic, which is relatively low to begin with (hypothetically, suppose 100 developers each save changes to 10 artifacts every minute and suppose that each of those changes leads to one indirect conflict, that still means only 2000 events a minute, which is very low traffic for today's event services). The extra analysis that is performed also does not incur undue computational cost, because Dependency Finder is incremental and we optimized the local cache of dependencies with look-up tables for fast retrieval of events.

We have kept the same level of unobtrusiveness as the original Palantír, using small icons and annotations in the common area of the environment to alert users. But, in large projects, a potential issue arises: having too many icons appear. Many a system involves an intricate web of program dependencies among its artifacts, which may lead to the generation of numerous indirect conflict events that all must be examined. On the one hand, this can indeed be a hindrance. On the other hand, there are several factors that help in addressing this hindrance. First, as discussed previously, Palantír only shows those events that are relevant to a particular workspace, so not all events are seen by all developers. Second, Palantír offers filters through which the set of notifications that is actually shown can be further reduced based on various criteria. Third, icons in the package explorer summarize for each artifact the full set of indirect conflicts, so that artifacts are decorated only once. Finally, we view the icons in the package explorer as an overall state that one can explore when planning a series of upcoming changes. It is the separate view below the editing pane that we believe is most useful when one is editing. This pane changes its contents based on the artifact that one is currently viewing and/or modifying, so at most a small set of indirect conflicts is present at any time – the set pertinent to the artifact at hand.

We note that our implementation thus far is language specific. In particular, the current version of Palantír operates on Java only. No inherent boundaries exist towards changing the implementation to support other languages, except that one might have to adjust the definition of the diff DTD if the language in question has other kinds of class signature elements. Of course, the actual analysis method used needs to be modified accordingly. Given that Palantír employs a straightforward dependency analysis so far, this should not be an issue.

Finally, we believe that the architecture that we have presented represents an important stepping stone towards addressing other kinds of indirect conflicts. The architecture lays out the variety of challenges to be overcome in cross-workspace analysis, separates fundamental functionality, and makes it clear where it must be changed to address different indirect conflicts. We note that our current implementation, however, can be improved in this regard to provide a pluggable infrastructure with clearly defined extension points. Providing such a generic infrastructure will be part of our future work.

5. EVALUATION

Many different kinds of evaluations must be performed to fully evaluate our work. One can imagine a study that tracks identified potential conflicts and evaluates whether they actually become a conflict or disappear, a study that examines the precision and recall of Palantír as compared to the actual conflicts that arise, and a longitudinal study to find out if developers learn how to properly gauge the information provided and effectively integrate awareness into their day-to-day work. Before we perform these kinds of studies, however, we feel it is necessary to first establish something more basic: does Palantír help developers detect indirect conflicts early improve their ability to coordinate their work, and improve the quality of the code that results from the collaborative effort?

We performed two pilot studies to address this question, one in which we compared results with and without Palantír and a second in which we compared the new Palantír (support for both direct and indirect conflicts) to the old Palantír (support for only direct conflicts). Both pilot studies required subjects to collaborate in a small hypothetical team of three to complete a predefined programming task. The team members with whom subjects interacted were all actually virtual entities: *confederates*, which were controlled by a member of our research team, so conflicts could be inserted in a controlled manner in the otherwise unpredictable activity of programming. Particularly, both direct and indirect conflicts were inserted in the exact same manner at the exact same time related to when a subject started a certain modification task. Subjects were told that they could contact the other team members via IM should that be necessary. The code to be modified comprised approximately 500 lines of code in 19 Java classes. Subjects verbalized their thought process throughout the study and one of the authors was present as an observer.

In the first study, subjects had to complete 12 tasks, 4 of which would lead to direct conflicts and 4 to indirect conflicts. Subjects were given unlimited time, so the experiment could evaluate the up-front cost of monitoring for potential conflicts, communicating, and resolving them early versus the cost of attempting to fix the conflict later (typically at check-in or synchronization time). Three subjects used Palantír, three subjects did not.

In the second study, subjects had to complete 8 tasks, two with a direct conflict and two with an indirect conflict. Subjects were given exactly an hour, in order for us to evaluate how much interference Palantír's indirect conflict notifications had with the development process. Four subjects used Palantír with support for both direct and indirect conflict detection; four subjects used Palantír with support solely for direct conflict detection.

These being pilot studies with limited numbers of subjects, statistical conclusions cannot and should not be drawn. However, we did learn a number of lessons that confirm our intuitions and begin to illustrate the potential of our approach. First, subjects who had the new Palantír did much better than subjects who had no Palantír or the old Palantír. In fact, in the second study, those supported with indirect conflict detection found all indirect conflicts early and resolved them such that no conflicts entered the code base. The other subjects had more trouble, and in a majority of cases, indirect conflicts were left in the changes that were ultimately checked in, deteriorating the quality of the code. Further, subjects with the new Palantír extensively communicated with "team members" and used a variety of methods to actively coordinate work (e.g., some skipped tasks for a bit, others IMed to set up a sequence of tasks, yet others used placeholders in their code until they noticed the new code from the team member being checked in). These results indicate that the notifications provided by Palantír were actively used by the

developers, with the auxiliary result that indirect conflict detection matters, even when developers are already notified of direct conflicts.

With respect to time, in the first study subjects with support for detecting indirect conflicts took more time to complete all of the tasks, but less time when those without such support were asked to manually locate and resolve the remaining conflicts (simulating a build or test error stemming from an indirect conflict). In the second study, subjects with the new Palantír took more time per task on average (but, given that they delivered code that was of a higher quality because it contained no indirect conflicts, did so for a very good reason). This begins to indicate that Palantír indeed could be beneficial in providing developers with an overall reduction in time and effort spent because the small up-front investments made to proactively coordinate early prevent them from having to deal with much larger issues later on.

Verbal feedback from subjects and our personal observations of the subjects in action confirm these results. Even so, we understand that our results have been obtained in a limited experimental setting and need to be corroborated by future studies. Nonetheless, these results are extremely encouraging and should lead to further, statistically significant, study – by us, and by others.

6. RELATED WORK

A number of workspace awareness tools exist that help developers identify direct conflicts. BSCW [2] is a web-based, shared, centralized workspace with integrated versioning facilities that allow it to be used as an SCM system. Awareness is provided statically, via icons that enrich an artifact's web page with information regarding its current state, and dynamically, through a Monitor Applet that continuously informs developers of what activities are taking place. Jazz [7] is an Eclipse-based collaborative development tool that leverages the versioning capabilities of SCM systems to provide information of which artifacts are being edited in remote workspaces and which artifacts in the repository have newer versions than the ones checked out in the local workspace. The War Room Command Console [27] provides a centralized, multi-monitor display where it shows all artifacts in the software repository, color coding and decorating those that are concurrently being edited in private workspaces. None of these systems addresses indirect conflicts to date.

COOP/Orm [23], Celine [11], and State Treemap [24, 25] follow an approach comparable to that of BSCW and Jazz, but integrate additional information on the nature and size of a direct conflict. Using various mechanism to decide upon this information, these tools provide a more detailed development context to the developers. Again, however, they address direct conflicts only.

Chianti [30] identifies which test cases (regression or unit) are affected by a change. To do so, Chianti analyzes the base and current version of an artifact to identify the subset of test cases that are affected and need modification. TUKAN [32] performs program analysis in Smalltalk to determine which artifacts are semantically related and documents these relationships in a semantic network of artifacts that is used to determine if current changes to artifacts affect any other artifacts in the graph. Chianti can be seen as a different analysis technique to be used by Palantír, which would leverage a different source (the test cases) to determine whether indirect conflicts are present. TUKAN is close to the ideas presented in this paper, but two important differences exist. Compared to our approach, TUKAN presents information at certain intervals only, rather than instantaneously, which hinders a user's ability to properly assess the indirect conflicts. Second, TUKAN operates in a centralized manner, whereas our approach supports distributed settings.

Agile methodologies are very related to our work. The issue that we attempt to address with awareness (early detection of emerging direct and indirect conflicts) is also what Agile methodologies attempt to address with an approach that relies on small changes that are checked in and tested frequently [1, 29]. We note that our approach is in reality compatible with Agile approaches: it helps developers more flexibly plan *when* they need to check in their changes by providing insight in what other developer are doing in their workspaces. In fact, one awareness tool, FastDash [6], explicitly targets Agile teams – though it, once again, only addresses direct conflicts.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a novel technique that is explicitly designed to address the problem of indirect conflicts arising from changes to class signatures. Within the broader strategy of using awareness to address emerging conflicts in parallel development, our work represents a conceptual leap from techniques that address direct conflicts by "simply" broadcasting events, to techniques that support detection of potential indirect conflicts by leveraging cross-workspace analysis. Our pilot studies show promise in how our approach helps developers in detecting indirect conflicts early, as well as in responding appropriately.

We recognize that our work only begins to scratch the surface of the problem of how to address indirect conflicts with awareness. The problem we chose to address, indirect conflicts arising from changes in class signatures, is an important one (as discussed in Section 3), but remains squarely syntactic in nature. Within the broad range of indirect conflicts that are possible, this represents an "easier" problem to tackle, especially compared to semantic indirect conflicts. The challenge now is to build upon our work and extend the range of indirect conflicts that can be addressed. Clearly, incorporating additional kinds of analyses into Palantír is an appropriate beginning. To truly push the boundaries, however, it might be interesting to explore bringing build and test techniques into the picture, attempting to continuously build and test across workspaces so combined changed are "pre-built" and "pre-tested" as they are implemented. This brings with it a host of challenges, but can be a promising research direction towards effectively addressing semantic indirect conflicts.

Our future work also includes restructuring Palantír into a pluggable infrastructure, such that other analyses and visualizations addressing other kinds of indirect conflicts can be experimented with. Moreover, we plan to examine the role that our approach can play in global software development projects, where components and interfaces are typically hidden behind formal APIs, changes to which do not become visible until the official release date. Finally, we plan to investigate the grouping of related notifications (e.g., those related to a refactoring of the code) to further address scalability and unobtrusiveness.

8. ACKNOWLEDGMENTS

Effort partially funded by the National Science Foundation under grant numbers CCR-0093489, IIS-0205724, and IIS-0534775. Effort also supported by an IBM Eclipse Innovation grant and an IBM Technology Fellowship.

9. REFERENCES

[1] P. Abrahamsson, et al., *Agile Software Development Methods: Review and Analysis*. 2002: VTT Publications.pp.478.

[2] W. Appelt, *WWW Based Collaboration with the BSCW System*. Conference on Current Trends in Theory and Informatics, 1999, p. 66-78.

[3] R. Arnold and S. Bohner, *Software Change Impact Analysis (Practitioners)*. 1 ed. 1996: pp. 392.

[4] R. S. Arnold and S. A. Bohner, *Impact Analysis - Towards a Framework for Comparison*. ICSM, 1993, p. 292 - 301.

[5] B. Berliner, *CVS II: Parallelizing Software Development*. USENIX Technical Conference, 1990, p. 341-352.

[6] J. Biehl, et al., *FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams*. SIGCHI conference on Human Factors in computing systems, 2007, p. 1313-1322.

[7] L.-T. Cheng, et al., *Jazzing up Eclipse with Collaborative Tools*. Eclipse Technology Exchange Workshop, 2003, p. 102-103.

[8] C. R. B. de Souza, D. Redmiles and P. Dourish, *"Breaking the Code", Moving between Private and Public Work in Collaborative Software Development*. International Conference on Supporting Group Work, 2003, p. 105-114.

[9] C. R. B. de Souza, et al., *How a good software practice thwarts collaboration: the multiple roles of APIs in software development*. FSE, 2004, p. 22-230.

[10] P. Dourish and V. Bellotti, *Awareness and Coordination in Shared Workspaces*. ACM CSCW, 1992, p. 107-114.

[11] J. Estublier and S. Garcia, *Process Model and Awareness* in SCM. Twelfth International Workshop on Software Configuration Management, 2005, p. 69-84.

[12] J. Estublier, et al., *Impact of Software Engineering Research on the Practice of Software Configuration Management*, ACM TOSEM, vol. 14 (4), 2005, p. 1-48.

[13] P. H. Feiler, *Configuration Management Models in Commercial Environments,* SEI-91-TR-07, Software Engineering Institute, Carnegie Mellon University 1991.

[14] G. Fitzpatrick, et al., *Supporting Public Availability and Accessibility with Elvin: Experiences and Reflections*. ACM CSCW, 2002, p. 447-474.

[15] R. E. Grinter, *Supporting Articulation Work Using Software Configuration Management Systems*. ACM CSCW, 1996, p. 447-465.

[16] R. E. Grinter, *Recomposition: Putting It All Back Together Again*. ACM CSCW, 1998, p. 393-402.

[17] R. E. Grinter, J. D. Herbsleb and D. E. Perry, *The Geography of Coordination: Dealing with Distance in R&D Work*. ACM CSCW, 1999, p. 306-315.

[18] J. Grudin, *Why CSCW applications fail: problems in the design and evaluation of organization of organizational interfaces*. ACM CSCW, 1988, p. 85-93.

[19] C. Gutwin and S. Greenberg, *Workspace Awareness for Groupware*. Conference Companion on Human Factors in Computing Systems, 1996, p. 208-209.

[20] C. Gutwin and S. Greenberg, *The Effects of Workspace Awareness Support on the Usability of Real-Time Distributed Groupware*, TOCHI, vol. 6(3), 1999, p. 243-281.

[21] J. Herbsleb, et al., *Introducing Instant Messaging and Chat in the Workplace*. SIGCHI conference on Human factors in computing systems: Changing our world, changing ourselves, 2002, p. 171-178.

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

[23] B. Magnusson and U. Asklund, *Fine Grained Version Control of Configurations in COOP/Orm*. Sixth International Workshop on Software Configuration Management, 1996, p. 31-48.

[24] P. Molli, H. Skaf-Molli and C. Bouthier, *State Treemap: an Awareness Widget for Multi-Synchronous Groupware*. International Workshop on Groupware, 2001, p. 106-114.

[25] P. Molli, H. Skaf-Molli and G. Oster, *Divergence Awareness for Virtual Team through the Web*. Integrated Design and Process Technology, 2002.

[26] M. Mortensen and P. Hinds, *Fuzzy Teams: Boundary Disagreement in Distributed and Collocated Teams*. Distributed Work: New Research on Working across Distance Using Technology, 2002. p. 283-308.

[27] C. O'Reilly, D. Bustard and P. Morrow, *The War Room Command Console: Shared Visualizations for Inclusive Team Coordination*. ACM symposium on Software visualization, 2005, p. 57-65.

[28] D. E. Perry, H. P. Siy and L. G. Votta, *Parallel Changes in Large-Scale Software Development: An Observational Case Study*, ACM TOSEM, vol. 10 (3), 2001, p. 308-337.

[29] Agile Manifesto principles, http://www.agilemanifesto.org/principles.html.

[30] X. Ren, et al., Chianti: *A Tool for Change Impact Analysis of Java Programs*. Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2004, p. 432-448.

[31] A. Sarma, Z. Noroozi and A. van der Hoek, *Palantír: Raising Awareness among Configuration Management Workspaces*. Twenty-fifth International Conference on Software Engineering, 2003, p. 444-454.

[32] T. Schümmer and J. M. Haake, S*upporting Distributed Software Development by Modes of Collaboration*. Seventh ECSCW, 2001, p. 79-98.

[33] W. F. Tichy, *RCS, A System for Version Control*, Software - Practice and Experience, vol. 15 (7), 1985, p. 637-654.