ISR Institute for Software Research
University of California, Irvine

Cross-Workspace Impact Awareness for Early Detection of API-induced Indirect Conflicts in Configuration Management

Anita Sarma
University of California, Irvine
asarma@ics.uci.edu

Gerald Bortis
University of California, Irvine
gbortis@ics.uci.edu

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

December 2006

ISR Technical Report # UCI-ISR-06-17

Institute for Software Research
ICS2 110
University of California, Irvine
Irvine, CA 92697-3455
www.isr.uci.edu

Cross-Workspace Impact Awareness for Early Detection of API-induced Indirect Conflicts in Configuration Management

Anita Sarma, Gerald Bortis, and André van der Hoek

Institute of Software Research
University of California, Irvine
Irvine, CA 92697-3440 USA
{asarma, gbortis, andre}@ics.uci.edu

ISR Technical Report # UCI-ISR-06-17

December 2006

Abstract

Parallel development has been shown to frequently lead to conflicting changes. These conflicts can be categorized into two main classes: (1) direct conflicts, which arise due to concurrent changes to the same artifact, and (2) indirect conflicts, which arise due to changes in one artifact affecting concurrent changes in another artifact. While the detection of direct conflicts is supported by current workspace awareness tools, detection of indirect conflicts remains unsupported. To fill this void, we have designed a novel approach for cross-workspace impact awareness that helps in the early detection of API-induced indirect conflicts. The approach relies on the transmission across workspaces of API differences of ongoing changes. By using a local cache of dependencies, it is then possible to calculate the impact of remote API changes and, vice versa, to determine if any local changes cause a new indirect conflict. We present our approach, discuss its implementation in our workspace awareness tool Palantír, and show its value with an experimental evaluation.

Cross-Workspace Impact Awareness for Early Detection of API-induced Indirect Conflicts in Configuration Management

Anita Sarma, Gerald Bortis, and André van der Hoek
Institute of Software Research
University of California, Irvine
Irvine, CA 92697-3440 USA
{asarma, gbortis, andre}@ics.uci.edu

ISR Technical Report # UCI-ISR-06-17
December 2006

Abstract

Parallel development has been shown to frequently lead to conflicting changes. These conflicts can be categorized into two main classes: (1) direct conflicts, which arise due to concurrent changes to the same artifact, and (2) indirect conflicts, which arise due to changes in one artifact affecting concurrent changes in another artifact. While the detection of direct conflicts is supported by current workspace awareness tools, detection of indirect conflicts remains unsupported. To fill this void, we have designed a novel approach for cross-workspace impact awareness that helps in the early detection of API-induced indirect conflicts. The approach relies on the transmission across workspaces of API differences of ongoing changes. By using a local cache of dependencies, it is then possible to calculate the impact of remote API changes and, vice versa, to determine if any local changes cause a new indirect conflict. We present our approach, discuss its implementation in our workspace awareness tool Palantír, and show its value with an experimental evaluation.

1. Introduction

With the advent of powerful Configuration Management (CM) systems, parallel development has become a norm rather than an exception. But, with it comes a price: the high cost of conflict resolution. It has been shown that parallel development frequently leads to conflicts, the resolution of which is often not trivial [1, 2].

Conflicts stemming from parallel development can be grouped into 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 alter the same Java file. Indirect conflicts are caused by changes to one artifact that affect concurrent changes to another artifact. This may occur, for example, when one developer changes a Java interface that another developer just started using.

Direct and indirect conflicts are usually discovered at a time later than when they are actually introduced. Configuration management systems have explicit facilities for detecting and even resolving direct conflicts. At check-in time, the developer of a change that would overwrite changes from a previous check-in is warned that they must merge their efforts. Automated merge tools help them do so. But, when changes truly overlap and alter the same lines of code, these merge tools fail. But beyond direct conflicts, CM systems lack facilities for detecting, let alone resolving, indirect conflicts. Problems that are introduced in the code by an indirect conflict usually become visible at a much later time than those introduced by a direct conflict. It may be during the build process, but it could equally be during integration testing or, worse, after the product is already in the field.

The delay between when a conflict is introduced and when it is detected lies at the core of how difficult it is to resolve them. Conflicting changes have usually been completed by the time they are detected. For one person to understand both changes and to put them together can be very time-consuming and difficult [3, 4].

In many ways, this issue was a driving factor in the development of agile methodologies, which advocate, among others, small changes that are checked in and tested frequently. But, such methodologies cannot be used everywhere and do not have to be the only response to the issue. In fact, the field of Configuration Management has been working on its own solution, one based on promoting awareness.

Awareness is characterized as "an understanding of the activities of others, which provides a context for your own activity" [5]. It was originally introduced as a passive and subconscious form of information gathering, but has since then been explored in more proactive and conscious settings with tools attempting to provoke awareness by providing their users with specific kinds of information.

To date, the field of configuration management has explored the use of awareness in workspace awareness tools that inform developers of potential conflicts (e.g., [2, 6, 7]). The goal is to provide developers with relevant information as to which developers change which artifacts *as these changes are in progress and developing*. The anticipated effect is that developers note any potential conflicts early and then can respond proactively. Such responses may be to call the other party and discuss, to hold off on changes until the other person has checked in their changes, or to use the CM tool to look at the other person's workspace and ongoing changes – all in the name of minimizing the conflicts or even avoiding them entirely. To date, these workspace awareness tools address direct conflicts only.

In this paper, we present a new cross-workspace impact awareness approach that we specifically designed to begin explore the issue of indirect conflicts. Indirect conflicts can arise in many forms. Here, we concentrate on those indirect conflicts that arise from changes to Application Programmatic Interfaces, which we will call API-induced indirect conflicts. The approach relies on the transmission across workspaces of API differences of ongoing changes. By using a local cache of dependencies, it is then possible to calculate the impact of these remote API changes and, vice versa, to determine if any local changes cause a new indirect conflict. If changes are found to be conflicting, both the originating and the impacted workspace are notified.

We have implemented this approach by extending our own workspace awareness tool, Palantír [8]. Using this implementation, we engaged in an experimental evaluation in which we compared how several test subjects fared with and without the warnings of potential indirect conflicts. Our results demonstrate that our approach is more effective than an approach identifying merely direct conflicts, in fact eliminating API-induced indirect conflicts altogether.

The remainder of this paper is organized as follows. Section 2 introduces background materials in the areas of configuration management and awareness. Section 3 presents a motivating example, Section 4 the high-level approach, and Section 5 our implementation. Section 6 discusses our evaluation, Section 7 is related work, and Section 8 concludes with an outlook at our future work.

2. Background

Configuration management systems come in two varieties: pessimistic and optimistic [9, 10]. Pessimistic CM systems mandate that developers lock any artifact before they make changes, prohibiting others from making changes in parallel. Optimistic CM systems explicitly allow the occurrence of parallel changes and provide merge tools to help resolve direct conflicts.

This paper concerns optimistic CM systems. These systems are based on the assumption that relatively few parallel changes lead to conflicts. They, thus, trade off the need to resolve these conflicts for a speedier development process since developers no longer have to wait for locks to be released.

In practice, this approach indeed works well most of the time and parallel development is now more popular than ever. However, conflicts do occur and have certain detrimental effects. It has been reported that developers rush to be the first to check in their changes, so they do not have to be the one dealing with conflicts [11]. It has also been reported that, the more parallel development takes place, the more bugs are inserted in the code [1]. And, of course, any conflict incurs overhead in terms of time and effort that must be invested to resolve it.

Breakdowns in coordination have been attributed as a principal reason behind conflicting changes. Standard task assignment approaches and careful distribution of code responsibilities over developers are insufficient to guarantee a conflict-free environment.

Developers have responded by establishing their own, informal conventions to create a context for their work. They may send e-mail informing other developers of the changes they have made and the effect these changes could have [12]. They also have been reported to Instant Message to coordinate their work [13], or leveraging the CM repository to try and find out who is making changes where [11]. But these informal practices are entirely developer driven and have little automated support from the CM system for the developers to gather the information that they need.

By adopting solutions that embrace proactive forms of generating awareness, modern CM tools attempt to address this problem. Specifically, workspace awareness tools have been developed that complement traditional CM functionality by hooking into a workspace and distributing relevant information regarding ongoing changes. This relevant information usually pertains to which developer is changing which artifact [7], but also may include information regarding the size of

such changes [14] or regarding which exact lines of code are being changed [15].

Through careful integration and presentation of the resulting "awareness information" in the development environment, the goal is to alter the behavior of the developers such that, when they notice a potential conflict emerging, they will take proactive steps to attempt to avoid it from growing too large. Oft, it may even be possible to avoid a conflict from emerging in the first place; developers generally will not like to change files that they know are being worked on in parallel.

Key to workspace awareness tools, then, is a careful interplay between the information that is provided and the kinds of responses that are anticipated. It is pertinent that humans are considered an integral part of the solution. Their responses determine whether a workspace awareness tool is effective. Compared to automated approaches, two benefits and one drawback exist. A first benefit is that humans are cognitively more capable than computers. They have inherent knowledge of code structures, knowledge that the tools can leverage. The key for a workspace awareness tool is to jar a (correct) response, not to be the be-all end-all information source. A second, related benefit is that this means that workspace awareness tools do not necessarily have to be precise in the information they provide. A simple nugget of approximate information can be sufficient.

The drawback is that workspace awareness tools rely on humans to actually see and respond to the information provided. Humans must exhibit the discipline to do so, and tools must be built in such ways that they do not interfere with the regular job of software development – otherwise, they simply will not be used.

3. Motivating Example

Throughout this paper, we will use a motivating example that illustrates the need for cross-workspace impact awareness. The example involves two developers, Pete and Ellen, working on some hypothetical software to process credit cards. Ellen is responsible for dealing with changes to an existing class *Payment*. Pete has to implement a new class *CreditCard*, which will have to keep track of payments. When Pete arrives in the morning, he checks out for reading the artifact *Payment.java* and begins his work on a new artifact *CreditCard.java*.

Meanwhile, Ellen arrives a little bit later than Pete, and begins her morning by reviewing the implementation of *Payment.java* that she did yesterday. She notices that 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 some future changes she has planned. She is keenly aware of the significance of this change, documenting the code with detailed comments.

Pete, meanwhile, is working to finish his implementation of *CreditCard.java*. In the process, he makes, as he understands it, appropriate calls to new instances of the class *Payment*. He studied the code carefully, and noticed the initialization code is in the constructor, so he knows he is fine in that regard and does not have to call any special initialization routine(s).

Pete finishes just before lunch, as does Ellen. They both check in their code and the CM system accepts their changes. Clearly, a conflict has been introduced in the code that was not detected by the CM system. The build system also will not find this conflict, as the combined code compiles just fine.

One can argue that Ellen should have synchronized her workspace with the latest changes in the repository and tested her changes with that latest version. In all likelihood, she would have detected the conflict. But, there are two problems with this:

- She has to actually scour the code carefully to find out what is broken. In particular, she has to scour Pete's changes to find the true source of the problem. While in this case it is a trivial find, changes are generally more complex than the toy example introduced here.
- She has to insert a solution that keeps both original changes in tact. In the example, this is relatively straightforward to implement (Ellen needs to add a method call to *init* in Pete's code). When changes are complex and highly interrelated, however, this will generally not be the case.

It would have been preferable had Pete and Ellen been able to detect the conflict as soon as it emerged. Pete could have initiated a conversation with Ellen about her changes, 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 scenario, existing workspace awareness tools that help in detecting direct conflicts may have helped some. They would have indicated that Ellen was working on *Payment.java* and Pete on *CreditCard.java*. But it could only have been Pete, who knew he started referring to the *Payment* class, who could have noticed that he should maybe talk to Ellen. Ellen, however, has no way of knowing that Pete is establishing a dependency on her code. In more complex situations, unless developers are intimately familiar with the structure of

the code and the dependencies among its parts, and unless they use this knowledge to constantly check direct conflicts for potential impact on their work, existing workspace awareness tools will fail to assist in them detecting indirect conflicts.

While the example is necessarily trivial, it is representative of the class of indirect conflicts that concern this paper, namely API-induced indirect conflicts. Previous work has recognized how changes to APIs are a major source of conflicts, both direct and indirect [3, 4]. APIs' critical role as boundary objects, combined with inappropriate task assignments that continue to result in work that overlaps and/or interferes, lies at the heart of this problem [16, 17].

4. Approach

Our overall approach follows that of existing workspace awareness tools [18] in collecting, distributing, and visualizing relevant information regarding parallel development. In our case, this means we must collect, distribute, and visualize information regarding changes to APIs. Whereas workspace awareness tools addressing direct conflicts can obtain all necessary information solely from the workspace in which changes originate, indirect conflicts involve an approach in which relevant information must be collected from pairs of workspaces to gauge whether changes in one affect changes in the other. Specifically, detecting API-induced indirect conflicts requires an impact analysis over the combination of API changes in one workspace with changes in API use in other workspaces; an impact analysis that must be performed for all pairs of workspaces.

To achieve such cross-workspace impact analysis of API changes, our approach relies on six steps:

1. *Identification of relevant changes in a workspace.* As with other workspace awareness tools, we rely on continuous monitoring of changes that are being made in the workspace. Whenever any artifact undergoes any change, we track the change. From the resulting set of all changes, we filter two types of information: changes to APIs and changes in the use of APIs.
2. *Distribution of API changes*. Each API change has to be distributed to those workspaces where it may have an impact. Specifically, we distribute them to any workspace in which the API is currently used. Note that we only distribute API changes and not any changes in the use of APIs. Steps 3 and 4 address this issue. Note also that the choice of only distributing changes to workspaces in which the API is already used could lead to missed indirect conflicts (i.e., if a workspace starts using the API after the API change has been made but before the change is committed to the central CM repository). To avoid this problem, we bootstrap a local workspace when a developer imports a new API.
3. *Maintenance of a local dependency cache.* To prepare for the analysis step, our approach keeps an up to date cache representing the current set of dependencies among all code elements in a workspace. This dependency cache is updated continuously, with each local save in a workspace.
4. *Change impact analysis.* Whenever a workspace is notified that an API has changed in another workspace, an analysis is performed over the local dependency cache. This updates two sets of artifacts: (1) the set of artifacts whose API changes impact other artifacts, and (2) the set of artifacts that are impacted by API changes. This way, a developer can trace an API to all of the artifacts with which it has an indirect conflict and, vice versa, trace an artifact to the API that causes a potential indirect conflict. Of course, we also perform this analysis every time the local dependency cache changes.
5. *Distribution of the impact awareness information.* Once an API-induced indirect conflict is detected, it is communicated back to the workspace where the change to the API originated so its developer is also aware of the indirect conflict.
6. *Display of the impact awareness information.* The developers must be informed of the indirect conflict. This is done by unobtrusively integrating the impact awareness information in their standard development environment so that, when an indirect conflict arises, there is a high chance they actually notice it peripherally.

We make several tradeoffs with this approach. First, we note that the presence of an indirect conflict does not necessarily mean there is a problem to be solved. It is an indication that there *may* be a problem. This is in line with how workspace awareness tools handle direct conflicts: they notify developers that they are modifying the same artifact, but it may be that they are modifying different parts that a merge will simply resolve. Our approach operates analogously: it notifies two developers when they modify two closely-related artifacts, but in so doing it may highlight a broader set of situations than strictly those exhibiting actual problems.

The second issue is related: our impact analysis is performed as a straightforward dependency analysis at the method level; we track the *extends*, *implements*, and *uses* relationships. This is relatively coarse-grained and can be refined through more precise dependency analysis techniques (e.g., [19-21]). Such techniques, however, are more expensive in terms of computational requirements. At the cost of being more inclusive, we choose a method that is cheap to execute, which is important in our case since analyses take place relatively frequently on each developer's machine.

While, at first sight, being conservative in flagging potential problems is a weakness of the approach, it in fact is a strength when we return to the notion of awareness and what it is trying to achieve. Awareness builds a context for one's work so one can plan their actions accordingly. If an awareness approach for indirect conflicts only presents information whenever a problem is actually introduced, it would not allow the developers an opportunity to avoid the problems in the first place. If two developers were not aware of each others efforts, got each 75% of the way, and then finally introduced an actual conflict that the tool tells them is a real problem, the tool did not help them very much. They are both far into their changes, have ingrained assumptions, and face a difficult time to put the changes together.

This strongly echoes the sentiment discussed in Section 2: solutions that involve awareness rely on technical and social responses. Therefore, it is critical in our approach that we provide information regarding all of the parallel changes that interlink the efforts of multiple developers. Only when providing them this context can they appropriately plan their future changes.

5. Implementation

We have implemented our approach as an extension to our existing workspace awareness tool, Palantír [8]. Palantír operates as an Eclipse plug-in. Its architecture is shown in Figure 1, with gray boxes representing the Eclipse platform, white boxes Palantír, and arrows the information flow. The original architecture of Palantír was described elsewhere [22]. To handle indirect conflicts, we had to add one component and modify a number of its existing components. These changes as compared to the original Palantír architecture are identified through dotted lines around components.

Briefly, the *Workspace Wrapper* collects and subsequently emits events regarding relevant workspace activities. These events are stored and distributed by the *Palantír Server*, which also supports bootstrapping any new workspaces that developers may open to perform their work. The *Palantír Client* pulls, stores, and organizes the events, which are finally displayed to the user.

We needed to change just one of the existing Palantír events, *MetricsChanged*, to support indirect conflicts. In the original, it contained a metric value only, to capture the percent of change that an artifact had undergone from original check out to its current state. In our new version, it contains this same metric value, but also a list of API differences. Specifically, the event captures any methods that have been added, removed, or changed in signature. This data is captured as a straightforward XML data structure.

The *Workspace Wrapper* also did not change much. The only change that we needed to make was to augment data collection at the time of a *MetricsChanged* event. Specifically, we integrated a third-party tool, the Dependency Finder [23], to compute all API differences between the checked-out version and the current version of the artifact being changed. This determination takes place each time an artifact is saved.

The *Internal State* was altered to maintain a cache of dependencies for the local workspace. We used the Dependency Finder tool to do so. Because the Dependency Finder continuously monitors changes in the project, it incrementally updates the dependencies rather than recalculating them wholesale each time. The resulting table, then, instantaneously reflects the current dependencies among the artifacts in the workspace.

The *Metric Analyzer* is at the heart of our work in addressing indirect conflicts. It is responsible for analyzing incoming changes, whether these changes stem from remote workspaces (as reflected in API diffs that are received via the *MetricsChanged* event) or from the local workspace (as reflected by changes in the cache of local dependencies). In case of remote changes, the *Metric Analyzer* walks through the set of API method removals and modifications, for each such change verifying if a possible indirect conflict exists with the local dependencies (e.g., if there is a local change adding a method call to a removed or modified method, if there is a class that is changed to implement a new interface so it will not be called directly anymore). All of the API additions are labeled as possible indirect conflicts, since they represent ongoing work and maybe indicative of future changes to other places in the API. In case of changes to the local dependency cache, the analysis is performed in reverse, reexamining previous API changes to verify whether they indirectly conflict with the new local changes. If an indirect conflict is found, the *Internal State* is updated with this information and a new event, an *Impact* event, is

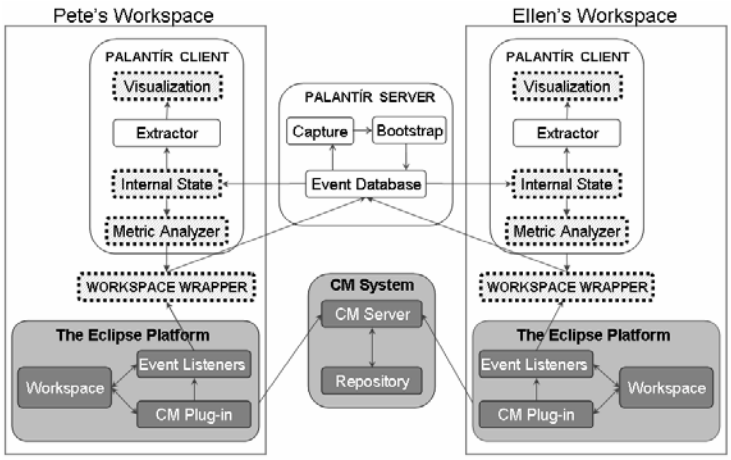

Figure 1. Revised Palantír Architecture for Handling Indirect Conflicts.

sent to inform the originating workspace of the indirect conflict.

Figure 2 shows our new *Visualization* for Palantír in support of the provision of information regarding indirect conflicts. The screen shot shows the view of Pete, making the changes of Section 3. Pete is working on his *CreditCard.java* class, and makes good progress. We note that he already has incorporated references to the *Payment* class that he inspected previously.

Palantír has altered the Eclipse interface in two distinct places. First, it annotates resources in the package explorer view with both graphical and textual items. In terms of graphics, two small triangles indicate changes to artifacts. The first one, inherited from the original version of Palantír, may appear in the top left corner of each icon representing an artifact. This triangle, visible on *Address.java*, *Customer.java*, and *Payment.java*, is used to indicate that parallel changes to an artifact are in progress. The larger the triangle, the greater are the changes. A small textual annotation, to the right of the filename, details the size of these changes. In our case, Ellen has changed 24% of *Address.java*.

The second triangle may appear in the top right corner of each icon denoting an artifact. This triangle is present on four artifacts: *Address.java*, *Customer.java*, *Payment.java*, and *CreditCard.java*, and indicates the presence of an indirect conflict. A small textual annotation, to the right of the filename, helps the developer distinguish whether the artifact is one causing an indirect conflict (in which case it is labeled with *[I>>]* to denote "outgoing impact"), is one that is affected by an indirect conflict (in which case it is labeled with *[I<<]* to denote "incoming impact"), or is one whose changes both affect changes in other artifacts and are affected by changes in other artifacts (in which case both textual annotations are present).

The package explorer view is designed to be non-obtrusive and provides the minimal information that is necessary to draw the attention of the user. Further details regarding indirect conflicts are presented through a new Eclipse view, the Palantír Impact View (see bottom of Figure 2). On selecting an artifact in the package explorer, the Palantír Impact View displays additional information detailing the indirect conflict(s).

In our example, Pete has noticed in his package explorer view that *CreditCard.java* has a marker in the top right corner and therefore incoming impact. He clicks on the artifact and examines the Palantír Impact View. To his surprise, it turns out Ellen has been making a number of different changes that are impacting his work. She has changed *Address.java* and deleted a method that he began using. Since she has already committed the changes to the CM repository, and since this is a predictable build conflict, the indirect conflict is annotated with a mini red bomb icon. The second line details a similar problem: deletion of a method from *Customer.java* that Pete began to use in his implementation. This change is annotated with a mini yellow bomb icon, because it is still in Ellen's workspace, but would create a build conflict in future should she commit the changes. Finally, Ellen has added the method *init* to her implementation of *Payment.java*. This method is an addition, and therefore is labeled with an exclamation mark to signify that there may be other changes under the hood, or forthcoming, that could lead to a problematic situation. Pete, worried about what else Ellen may change, decides to call her to coordinate their respective efforts. Ellen guides Pete verbally through her recent changes and communicates to Pete that she is planning some more, relatively major modifications that should be done in a couple of hours. Pete decides to hold off. After Ellen is done, he checks out her files into his workspace and creates his implementation of *CreditCard.java* without a problem.

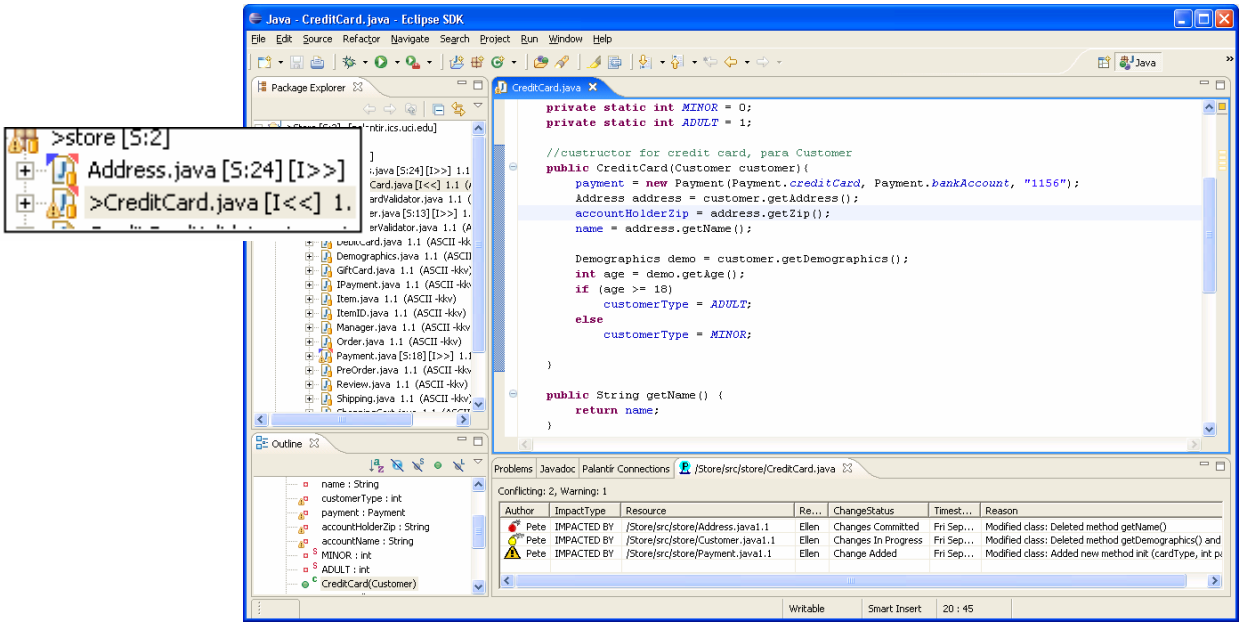

Figure 2. Palantír Visualization of Indirect Conflicts, With a Call-out of the Package Explorer.

Stepping back from the details of the implementation, our original work on Palantír laid out four important requirements for workspace awareness tools: non-obtrusiveness, scalability, flexibility, and configurability. We have kept the same level of obtrusiveness as the original Palantír, using small icons and annotations in the common area of the environment to alert users, and providing a separate view for further exploring details. We also kept a similar level of flexibility (other views are still usable) and configurability (users still control which events they want to see). In terms of scalability, we note that we have added one extra event (a reciprocal event that informs the workspaces whose API changes originate an indirect conflict) as well as additional analyses at both ends of the cross-workspace approach. The one extra event does not fundamentally alter the amount of traffic, which is relatively low (even when a 1000 developers each save their artifacts once a minute, that is still only a 1000 events per minute, which is mere moderate traffic for event services). The extra analyses also do not incur undue computational cost. The Dependency Finder is incremental, and we have optimized the *Internal State* with look-up tables for fast retrieval of events. Thus far, the overhead we have seen is minimal. A more in-depth look at performance is needed in future to ensure that our approach does indeed scale.

6. Evaluation

This section discusses our experimental evaluation, via which we assess the effectiveness of our approach.

6.1 Experimental setup

We designed the experiment with subjects collaborating in a hypothetical team of four to complete a predefined programming assignment. The assignment was structured to contain explicit dependencies among the artifacts, as well as some overlapping task assignments to introduce both direct and indirect conflicts. The code to be modified involved nineteen Java classes and approximately five hundred lines of code.

Subjects were divided into a control group, which was only provided with information of direct conflicts, and an experimental group, which received information of both direct and indirect conflicts. Ten subjects, all of whom are graduate students in Computer Science, were recruited and assigned to the control or experimental group via stratified random assignment [24].

Subjects participated in the experiment alone, interacting with confederates (virtual entities controlled by the research personnel). This was to ensure consistency in the type, number, and timing of conflicts introduced into the experiment. Subjects did not know the identity of the confederates and were told they can reach their team members through Instant Messaging. In total, we performed ten individual experiments, five each for the control and experimental group.

Each experiment, then, used one subject and three confederates. Subjects had to complete twelve tasks, of which eight were deliberately designed to conflict with the confederates' tasks. Four tasks involved direct conflicts and four indirect conflicts. These conflicts were randomly seeded throughout the group of twelve tasks.

Subjects were asked to verbalize their thought process throughout the experiment and one of the authors was present as an observer during the experiment. Subjects in both groups were provided with a UML class diagram to help them identify the dependencies in the project. The total time to complete the task assignment was restricted to one hour.

6.2. Experimental results

After taking an initial look at our results, we made the decision to only include eight subjects in the final analysis (because two subjects, one in each group, were able to complete just four and five tasks, respectively; the subjects were in the inexperienced stratum). Additionally, we consider only four of the actual eight conflicts that were seeded (two direct, two indirect), as the average number of tasks that subjects completed was only eight. These first eight tasks included the four conflicts. The first and second task had direct conflicts; tasks four and six indirect conflicts.

Our primary analysis focused on whether subjects were able to detect and resolve conflicts. Figure 3 illustrates the results of our analysis, as split into direct and indirect conflicts (DC versus IC) and control group and experimental group (:C versus :E). The y axis indicates the total number of conflicts of a given type that subjects in a given group could have detected and resolved (8, for each combination – 4 subjects and 2 conflicts).

Condition 1 (DC:C): We found that the majority of subjects did detect the presence of parallel changes; six out of eight subjects made note. Only one direct conflict was avoided, however, since most subjects did not realize they could avoid the direct conflict by communicating early with their team members. "I had noticed the blue icons, but I was in the train of thoughts… but after I ran into trouble, it provided me an incentive to talk to my team member and monitor the icons." We noted a definite reluctance to perform conflict resolution and found subjects racing to finish their tasks, corroborating findings from field studies [12, 25]. A subject commented: "I didn't like conflict resolution…it made me happy when the tasks that I did, did not include anyone else's". An interesting pattern was that most subjects only monitored for conflicts before embarking on a task and before committing the artifact.

Condition 2 (DC:E): Given that the conditions for these subjects were the same as the previous subjects (both had only warnings of potential direct conflicts), similar results with some natural fluctuations emerged. The difference was that all potential conflicts were detected and two conflicts were avoided because the

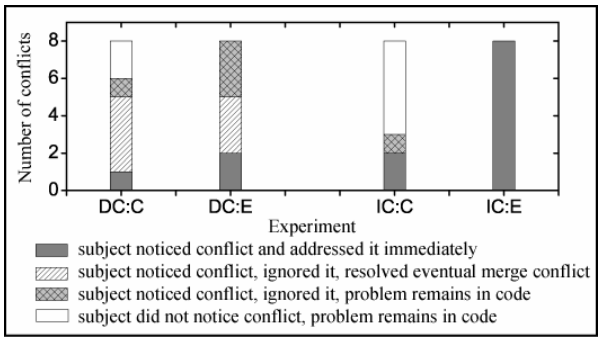

Figure 3. Experimental Results.

subjects noticed the warnings and interacted with peers or updated their workspace. One of them said: "…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". As with subjects in Condition 1, three conflicts remained in the code after the tasks was done.

Condition 3 (IC:C): Subjects had trouble detecting the indirect conflicts; only three of eight were noticed and five were completely missed. This is no surprise. It is possible to use information on direct conflicts to look for indirect conflicts (as we discussed in Section 3), but this is relatively difficult. A single subject did manage to detect and resolve both indirect conflicts faced; they proceeded cautiously and continuously looked at what efforts were happening in parallel, updating their workspace often "…because I am traumatized, I had past problems with committing things without updating, so I always synchronize my workspace before and after I finish a task". It is important to note that the workspace synchronizations in and of themselves were not sufficient. In one case, doing so broke the local build due to a deletion of a method; the subject needed to carefully examine and update their code in response.

Condition 4 (IC:E): The subjects identified and resolved *all* the indirect conflicts and used both the extension of the package explorer and the Impact View. They used the Impact View to determine which confederate was responsible for the changes, and whether the changes were committed or still a work-in-progress. A subject said: "…the icons, those were very helpful to determine like it was an impact…I found it really useful because I could sort of anticipate that there would be conflict just by looking, and …I could know what I needed to do, so I could have time to prepare, or like I did, I contacted him [confederate] directly to ask him what was happening at that moment." On identifying a conflict, subjects used a variety of methods to avoid or resolve the conflict: they updated their workspace, skipped the task and came back to it later, or coded the task with a place holder. In all cases,

the subject conversed with the confederate and continuously monitored the status of the artifact causing the conflict.

We also analyzed the time it took subjects to complete each conflicting task. For direct conflicts, as expected, there was no significant difference between the subjects in the control and experimental group. Indirect conflicts, however, did lead to differences. Specifically, the experimental group took on *average* three minutes of extra time. This is not a surprise, since they spent extra time resolving the conflict. Three minutes, however, is not a large investment of time in this regard.

Since only two subjects completed all tasks, the experiment was inconclusive as to whether *total* time was reduced. The subjects in the control group missed conflicts that still need to be resolved at a later time. This gives them a time advantage now over the experimental group. We are now planning a follow-on experiment in which we will let subjects complete all tasks, with the condition that all conflicts must also be resolved.

6.3 Discussion

Our study has two concrete results: (1) we know the quality of the code delivered by those in the experimental group is better than the quality of the code delivered by those in the control group, since all indirect conflicts were removed from the code that was checked in, and (2) we, as of yet, cannot conclusively say whether this results in an overall reduction of effort. But given the literature on the nature of conflicts, we fully expect our subsequent experiments to show such a reduction and demonstrate the benefits of our approach to early detection of API-induced indirect conflicts.

There is one major thread to validity with respect to our experiment. Specifically, some consider observation by research personnel and "thinking aloud" methodology a threat to validity, since it has the potential of changing a subject's behavior [26]. When asked afterwards, our subjects indicated they ignored the presence of the observer during their tasks. Clearly, this does not mean altered behavior did not take place, but having the chance to reflect, subjects did not seem to worry.

7. Related work

A number of workspace awareness tools exist that enable the developer to identify direct conflicts. BSCW [27] is a web-based, shared centralized workspace with integrated versioning facilities that allow it to be used as a CM system. Awareness is provided statically, via web-based icons that enrich the web page for each artifact with information concerning its state, and dynamically, via a Monitor Applet that continuously informs authors of what activities are taking place in the central workspace. Jazz [7] is an Eclipse-integrated collaborative development tool that leverages the versioning capabilities of CM systems to represent 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. In a similar fashion, the War Room Command Console approach [28] displays the set of artifacts present in the software repository and color codes the artifacts that are being concurrently edited in private workspaces. In all three cases, only direct conflicts are addressed.

Tools like COOP/Orm [15], Celine [2], and State Treemap [14, 29] follow a comparable approach, but provide additional information on the size of a conflict. In COOP/Orm, active diffs communicate changes to other developers who can see those changes both in the version tree and the actual artifact. Celine analyzes the current changes in an artifact with the state of an artifact in a remote workspace and the reference copy for the team to identify if a merge between the artifacts leads to a conflict. On a similar note, State Treemap shows which artifacts in the workspace are out-of-sync and calculates a divergence metrics that denotes the expected size of the conflict when the artifact will be synchronized with the version in the repository. Again, to date, these tools address direct conflicts only.

As per our knowledge there are only two tools that perform semantic analysis to identify the impact of *ongoing* changes. First, Chianti [30] is a tool geared towards identifying affected test cases (regression or unit). Chianti analyzes the base and current version of an artifact to identify the subset of test cases that are affected and need modification. Second, TUKAN [6] performs program analysis to determine which artifacts are semantically related and creates a semantic network of artifacts. It then uses this network to determine if current changes to artifacts would affect other artifacts in the graph and subsequently creates awareness icons to warn users of potential conflicts. While Chianti can be seen as a potential analysis technique to be plugged into our approach, TUKAN is close to the ideas presented in this paper. There are two important differences, however. Compared to our approach, TUKAN presents information only at certain times, which hinders broader awareness. Second, TUKAN operates in a centralized manner, whereas we support fully distributed settings.

8. Conclusions

We have presented a novel approach to the problem of API-induced indirect conflicts. Our approach represents an intricate blending of technical propagation and analysis of API changes and human interpretation and action undertaking in response to warnings of potential indirect conflicts. Through an experimental evaluation, we have demonstrated how this blend of technical and human approach actually helps in reducing the number of indirect conflicts, thereby indirectly improving code quality since fewer problems are committed to the CM repository.

The current incarnation of our approach represents only the beginning of our investigation into addressing indirect conflicts. Now that we have demonstrated that API-induced indirect conflicts can indeed be detected and resolved early, we plan to explore how we can address conflicts that are caused by behavioral changes in the code. Such conflicts are even more difficult to detect and to date can be done only in the testing phase; we want to bring such detection earlier. Additionally, we will perform detailed and larger-scale experiments to fully understand strengths and weaknesses of our approach.

9. Acknowledgements

Effort partially funded by the National Science Foundation under grant numbers CCR-0093489, DUE-0536203, and IIS-0205724. Effort also supported by an IBM Eclipse Innovation grant and an IBM Technology Fellowship.

10. References

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

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

[3] R.E. Grinter. *Recomposition: Putting It All Back Together Again*. *ACM conference on Computer Supported Cooperative Work*. 1998. p. 393-402.

[4] C.R.B. De Souza, et al. *How a good software practice thwarts collaboration: the multiple roles of APIs in software development*. *International Symposium on Foundations of Software Engineering*. 2004. p. 22-230.

[5] P. Dourish and V. Bellotti. *Awareness and Coordination in Shared Workspaces*. *ACM Conference on Computer-Supported Cooperative Work*. 1992. p. 107-114.

[6] T. Schümmer and J.M. Haake. *Supporting Distributed Software Development by Modes of Collaboration*. *Seventh European Conference on Computer Supported Cooperative Work*. 2001. p. 79-98.

[7] L.-T. Cheng, et al. *Jazzing up Eclipse with Collaborative Tools*. *18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications / Eclipse Technology Exchange Workshop*. 2003. p. 102-103.

[8] 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.

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

[10] R. Conradi and B. Westfechtel, *Version Models for Software Configuration Management.* ACM Computing Surveys, 1998. **30**(2): p. 232-282.

[11] R.E. Grinter, *Supporting Articulation Work Using Software Configuration Management Systems.* Computer Supported Cooperative Work, 1996. **5**(4): p. 447-465.

[12] 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 (Group 2003).* 2003. p. 105-114.

[13] 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.

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

[15] 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.

[16] R.E. Grinter, J.D. Herbsleb, and D.E. Perry. *The Geography of Coordination: Dealing with Distance in R&D Work*. *ACM Conference on Supporting Group Work (GROUP 99)*. 1999. p. 306-315.

[17] 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: MIT Press. 283-308.

[18] M.-A.D. Storey, et al. *On the Use of Visualization to Support Awareness of Human Activities in Software Development: A Survey and a Framework. ACM Symposium on Software Visualization*. 2005. p. 193-202.

[19] M. Robert Ito and A. Zaafrani. *Data flow analysis for parallel programs*. *Conference on Computer science*. 1993. p. 318-325.

[20] J.R. Ruthruff, et al. *Experimental program analysis: a new program analysis paradigm. International Symposium on Software Testing and Analysis.* 2006. p. 49-60.

[21] L. Lu. *A precise type analysis of logic programs. International Conference on Principles and Practice of Declarative Programming.* 2000. p. 214-225.

[22] R. Ripley, et al. *Workspace Awareness in Application Development*. *OOPSLA workshop on eclipse technology eXchange*. 2004. p. 17-21.

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

[24] W.R. Shadish, T.D. Cook, and D.T. Campbell, *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. 1 ed. 2001: pp. 623.
[25] R.E. Grinter. *Using a Configuration Management Tool to Coordinate Software Development*. *Conference on Organizational Computing Systems*. 1995. p. 168-177.
[26] J.L. Branch. *The trouble with Think Alouds: Generating data using concurrent verbal protocols*. *Annual Conference of the Canadian Association for Information Science*. 2000. p. A. Kublik.
[27] W. Appelt. *WWW Based Collaboration with the BSCW System*. *Conference on Current Trends in Theory and Informatics*. 1999. p. 66-78.
[28] C. O'Reilly,et al. *The War Room Command Console: Shared Visualizations for Inclusive Team Coordination*. *Symposium on Software visualization*. 2005. p. 57-65.
[29] P. Molli, H. Skaf-Molli, and C. Bouthier. *State Treemap: an Awareness Widget for Multi-Synchronous Groupware*. *Seventh International Workshop on Groupware*. 2001. p. 106-114.
[30] X. Ren, et al. *Chianti: A Tool for Change Impact Analysis of Java Programs*. *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2004. p. 432-448.