An Exploration of Indirect Conflicts

by

Jordan Ell
B.Sc., University of Victoria, 2013

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

An Exploration of Indirect Conflicts

by

Jordan Ell
B.Sc., University of Victoria, 2013

Supervisory Committee

Dr. D. Damian, Supervisor
(Department of Computer Science)

Dr. M. Tory, Departmental Member
(Department of Computer Science)

Dr. C. Bird, Outside Member
(Microsoft Research)

Supervisory Committee

Dr. D. Damian, Supervisor
(Department of Computer Science)

Dr. M. Tory, Departmental Member
(Department of Computer Science)

Dr. C. Bird, Outside Member
(Microsoft Research)

ABSTRACT

Awareness techniques have been proposed and studied to aid developer understanding, efficiency, and quality of software produced. Some of these techniques have focused on either *direct* or *indirect conflicts* in order to prevent, detect, or resolve these conflicts as they arise from a result of source code changes. While the techniques and tools for direct conflicts have had large success, tools either proposed or studied for indirect conflicts have had common issues of information overload, false positives, scalability, information distribution and many others. To better understand these issues, this dissertation will focus on exploring the world of indirect conflicts through 4 studies. The first two studies presented will focus on motivational circumstances which occur during the software development life cycle and cause indirect conflicts. Developers interactions are studied in order to create a tool which can aid in the workflows around indirect conflicts. The second two studies present a deeper investigation into why most indirect conflict tools fail to attrack developer interest through exploring the root causes of indirect conflicts and how tools should be properly built to support developer workflows.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

David, Leslie, Aaron, and Shelley, for supporting me throughout my research.

Dr. Daniela Damian, for mentoring, support, encouragement, and patience.

Change is the law of life. And those who look only to the past or present are certain to miss the future.

John F. Kennedy

DEDICATION

To Brittany.

Chapter 1

Introduction

1.1 Introduction

As Software Configuration Management (SCM) has grown over the years, the maturity and norm of parallel development has become the standard development process instead of the exception. With this parallel development comes the need for larger awareness among developers to have "an understanding of the activities of others which provides a context for one's own activities" [16]. This added awareness mitigates some downsides of parallel development which include the cost of conflict prevention and resolution. However, empirical evidence shows that these mitigated losses continue to appear quite frequently and can prove to be a significant and time-consuming chore for developers [44].

Large software projects are created using highly modular and reusable code. This creates technical dependencies between methods or functions that can be used in a wide variety of locations throughout the project. This causes changes to any given software object to have a rippling effect across the rest of the project [1]. The larger these effects are, the more likely they are to cause a software failure inside the system during the project's life span [63]. These observations of technical dependencies open the door to types of analysis on the developer networks they infer and preventing software failures by improving coordination amongst dependent developers.

Technical dependencies in a project can be used to predict success or failure in builds or code changes [46, 63]. However, most research in this area is based on identifying central modules inside a large code base which are likely to cause software failures or detecting frequently changed code that can be associated with previous failures [35]. This module-based method also results in predictions at the file or binary level of software development as opposed to a code change level and often lack the ability to provide recommendations for improved coordination other than test focus.

With the power of technical dependencies in tracking unintended consequences from source code changes, several tools have been created to attempt to solve task awareness related issues with some success [4, 33, 48, 59]. However, these tools have been designed to solve task awareness related issues at the direct conflict level.

Two types of conflicts have attracted the attention of researchers, *direct* and *indirect conflicts*. Direct conflicts involve immediate workspace concerns such as developers editing the same artifact. Tools have been created and studied for direct conflicts [4,33,48,59] with relatively good success and positive developer feedback. Indirect conflicts are caused by source code changes that negatively impact another location in the software system such as when libraries are upgraded or when a method is changed and invoking methods are influenced negatively. Indirect conflict tools however, have not shared the same success as direct conflict tools [6, 30, 47, 50, 53]. However, previous interviews and surveys conducted with software developers have shown a pattern that developers of a software project view indirect conflict awareness as a high priority issue in their development [3, 14, 26, 49], meaning that future research is required to address this developer concern.

Indirect conflicts arising in source code are inherently difficult to resolve as most of the time, source code analysis must be performed in order to find relationships between technical objects which are harmed by changes. While some awareness tools have been created with these indirect conflicts primarily in mind [3, 53], most have only created an exploratory environment which is used by developers to solve conflicts which may arise. These tools were not designed to detect indirect conflicts that arise and alert developers to their presence inside the software system. Sarma et al. [47] has started to work directly on solving indirect conflicts, however, these products are not designed to handle internal structures of technical objects.

While indirect conflict tools have shown potential from studies of developers, some of the same problems continue to arise throughout most, if not all tools. The most prevalent issue is that of false positives and information overload, tools eventually being ignored [47, 50]. A second primary issue is that of dependency identification and tracking. Many different dependencies have been proposed and used in indirect conflict tools such as method invocation [53], and class signatures [47] with varying success, but the identification of failure inducing changes, other than those which are already identifiable by other means such as compilers, and unit tests, to these dependencies still remains an issue. Dependency tracking issues are also compounded by the scale of many software development projects leading to further information overload. Lastly, social factors such as Cataldo et al's. [10] notion of socio-technical congruence have been leveraged in indirect conflict tools [3, 6, 36]. However, issues again of information overload, dependencies (in developer organizational structure), and scalability come up.

Clearly, indirect conflicts and its subsequent research areas have a large breadth of limitations, some of which will be explored in this dissertation. The research goal of this dissertation is to explore the limitations which exist with supporting indirect conflicts through awareness techniques as well as to explore possible solutions for industry practice in the area of indirect conflicts. To accomplish this goal, I have researched the following sub topics of indirect conflicts: technical dependencies, how developers are involved in said dependencies, socio-technical congruence as a mitigation strategy to indirect conflicts, what the root causes of indirect conflicts are, what compounding factors exist for indirect conflicts, what current industry mitigation strategies of indirect conflicts are being used, what future steps should be taken by researchers to better industry regarding indirect conflicts, and finally, how software evolution analysis can be used to better tools for indirect conflicts. I have addressed these issues by conducting four studies.

1.2 Research Methodology

In order to address the research goal as laid out in the previous section, I have conducted 4 studies which will now be briefly outlined. Each study builds off the previous one and has research questions informed from the findings of the previous study.

Study 1 focuses on the power of technical dependencies in software projects. The question I investigated were : "*Is it possible to identify pairs of developers whose technical dependencies in code changes statistically relate to bugs?*". This study explains the approach used to locate these pairs of developers in developer networks.The process utilizes code changes and the call hierarchies effected to find patterns of developer relationships in successful and failed code changes. These developer pairs can be seen as indirect conflicts occurring as one developer's code change has negatively affects another developer's work. As it will be seen, I found 27 statistically significant failure inducing developer pairs. These developer relationships can be used to promote the idea of leveraging socio-technical congruence, a measure of coordination compared to technical dependencies amongst stakeholders, to provide coordination recommendations. This notion of socio-technical congruence is my initial proposed solution to indirect conflicts. By identifying these failure inducing pair of developers over indirect conflicts, I hoped that recommended communication would be the correct fix. The results of Study 1 directly influence Study 2.

Study 2 attempts to take the failure inducing pairs of developers from Study 1 and create an awareness tool while answering: "*Can indirect conflicts be supported through an awareness mechanism which leverages pairs of developers whose changing technical dependencies statistically relate to bugs?*". In this study, I report on my research into supporting indirect conflicts and present the design, implementation, and evaluation of the tool *Impact*, a web based tool that aims at detecting indirect conflicts among developers and notifying the appropriate members involved in these conflicts. By leveraging technical relationships inherent of software projects with method call graphs [38] as well as detecting changes to these technical relationships through software configuration management (SCM) systems, *Impact* is able to detect indirect conflicts as well as alert developers involved in such conflicts in task awareness. While this study outlines *Impact's* specific implementation, its design is rather generic and can be implemented in similar indirect conflict awareness tools. *Impact* represents a first step towards the design and implementation of awareness tools which address indirect conflicts in software development. After a brief evaluation of *Impact* with two student software teams, it was found that *Impact* suffers from information overload and a high false positive rate which turn out to be quite large problems found in many other indirect conflict tools [6, 30, 47, 50, 53]. In order to fully understand the causes of these indirect conflict tool issues, a third study was conducted.

In order to fully understand the root causes of information overload, false positives, and scalability issues in regards to indirect conflicts, Study 3 was an empirical study to determine what events occur to cause indirect conflicts, when they occur, and if conditions exist to provoke more of these events. I also set out to understand what mitigation strategies developers currently use as opposed to those created by researchers. Through this exploration, I examined what can be accomplished moving forward with indirect conflicts in both research and industry. This study asked the following 3 research questions: *What are the types, factors, and frequencies of indirect conflicts? What mitigation techniques are used by developers in regards to indirect conflicts? What do developers want from future indirect conflict tools?*

I interviewed 19 developers from across 12 commercial and open source projects, followed by a confirmatory survey of 78 developers, and 5 confirmatory interviews, in order to answer the aforementioned questions. The study findings indicate that: indirect conflicts occur frequently and are likely caused by software contract changes and a lack of understanding of those changes, developers tend to prefer to use detection and resolution processes or tools over those of prevention, developers do not want awareness mechanisms which provide non actionable results, and that there exists a gap in software evolution analytical tools arising from the reliance on static analysis resulting in missed context of indirect conflicts. As a result of the final finding (the gap in software evolution analytical tools), I conducted a fourth and final study.

In order to begin to address the gap in software evolution analytical tools discovered in study 3, I turn my analysis to the notions of software change trends, specifically those trends around major releases. Change trends are trends which indicate a likelihood for a change type to occur around a certain event. Change trends have been used to detect stability in core architecture [56] as well as evolving dependencies [8]. With the power of major release points in open source projects as a starting point for project stability and the understanding that change trends can be leveraged to detect stability and the proneness of indirect conflicts (as will be seen in Study 3), this study investigates the question: "*What trends exist in source code changes surrounding major releases of open source projects as a notion towards a project stability measure?*". I perform a case study of 10 open source projects in order to study their source code change trends surrounding major release points throughout their history. I studied 26 change trends quantitatively and 4 change trends qualitatively, and identified a core group of 9 change trends which occur prominently at major release points of the projects studied.

The remainder of this dissertation is laid out as follows. Chapter two includes Study 1 and Study 2 as the motivational studies which ultimately led to the larger research studies found in Study 3 and Study 4. Chapter 3 includes Study 3 and Study 4 which ultimately press upon indirect conflicts in a more in depth fashion than has been previously seen in research. Chapter 4 contains a lengthy discussion of what has been learned from all four studies of this dissertation as well as implications for further research and tool development in the field of indirect conflicts. Finally, Chapter 5 concludes this dissertation.

Chapter 2

Motivating Studies

While the research problems have been briefly outlined in Chapter 1, this chapter will focus on the underlying studies which motivated the research of this dissertation.

In this chapter, two studies will be presented which motivated, and gave insights into, the final research goals of this thesis. The first study entitled "Failure Inducing Developer Pairs" (Section 2.1), focuses on the prediction of software failures through identifying indirect conflicts of developers linked by their software modules. This study found that certain pairs of developers when linked through indirect code changes are more prone to software failures than others. The ideas of developer pairs linked in indirect conflicts will be useful for the further development of indirect conflict tools as it shows that a human factor is present and may be used to help resolve such issues.

The second study, "Awareness with Impact" ((Section 2.2)), takes the notion of developer pairs in indirect conflicts learned from Study 1, and adds in source code change detection in order to create an awareness notification system for developers called *Impact*. *Impact* was designed to alert a developer to any source code changes preformed by another developer when the two are linked in a technical dependency through a developer pair. *Impact* utilized a non-obtrusive RSS style feed for notifications. While *Impact* showed some promise through its user evaluation, it ultimately suffered the fate of information overload as was seen in other indirect conflict tools [47, 50, 53].

2.1 Study 1: Failure Inducing Developer Pairs

Technical dependencies in a project can be used to predict success or failure in builds or code changes [46, 63]. However, most research in this area is based on identifying central modules inside a large code base which are likely to cause software failures or detecting frequently changed code that can be associated with previous failures [35]. These module-based methods also result in predictions at the file or binary level of software development as opposed to a code change level and often lack the ability to provide recommendations for improved coordination other than test focus.

With the power of technical dependencies in predicting software failures, the question I investigated in this study was:

RQ *Is it possible to identify pairs of developers whose technical dependencies in code changes statistically relate to bugs?*

This study explains the approach used to locate these pairs of developers in developer networks. The process utilizes code changes and the call hierarchies effected to find patterns of developer relationships in successful and failed code changes. As it will be seen, I found 27 statistically significant failure inducing developer pairs. These developer relationships can also be used to promote the idea of leveraging socio-technical congruence, a measure of coordination compared to technical dependencies amongst stakeholders, to provide coordination recommendations.

2.1.1 Related Work

Research has shown multiple reasons for software failures in both technical dependencies as well as human or social dependencies in software development. On the technical side, studies have shown that technical dependencies in software are often powerful predictors or errors in software as well as in builds [27, 46, 63]. These technical dependencies are often accompanied by data mining algorithms in order to set apart failure inducing dependencies from non failing dependencies.

On the other side with the human factor, researchers have examined predicting build outcomes of software using communication patterns from developers. Wolf et al. [57] used patterns of communication from between pre-existing builds in order to predict later build outcomes. Naggappan et al. [43] showed that having a large communication organizational difference between developers who worked on the same software module had a negative influence on the quality of the software.

Studies have also combined both technical and social dependencies into a notion of socio-technical congruence. Cataldo et al. [10] have shown that this notion of socio-technical congruence can be leveraged to predict and improve task completion times in a software projects.

These studies have mostly focused at a very high and abstract level of software development (builds and task completion). Where they have fallen short is in fine grained analysis of software changes. This study is used to take the pre existing knowledge of both technical and social dependencies and apply it to a source code change level of granularity. Instead of large scale failures like long completion times or build failures, this study examines failures at the bug level induced by each code change.

2.1.2 Technical Approach

Extracting Technical Networks

The basis of this approach is to create a technical network of developers based on method ownership and those methods' call hierarchies effected by code changes. These networks will provide dependency edges between contributors caused by code changes which may be identified as possible failure inducing pairings (Figure 2.1). To achieve this goal, developer owners of methods, method call hierarchies (technical dependencies) and code change effects on these hierarchies must be identified. This approach is described in detail by illustrating its application to mining the data in a Git repository although it can be used with any software repository.

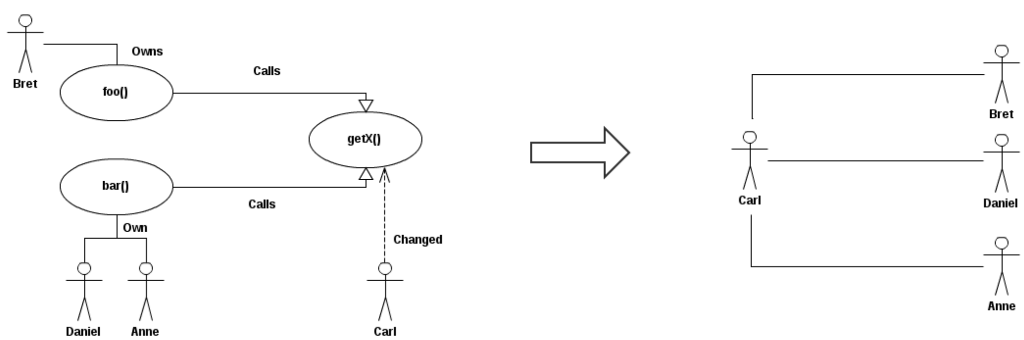

Figure 2.1: A technical network for a code change. Carl has changed method getX() which is being called by Bret's method foo() as well as Daniel and Anne's method bar().

To determine which developers own which methods at a given code change, the Git repository is queried. Git stores developers of a file per line, which was used to extrapolate a percentage of ownership given a method inside a file. If developer A has written 6/10 lines of method foo, then developer A owns 60% of said method.

A method call graph is then constructed to extract method call hierarchies in a project at a given code change. Unlike other approaches such as Bodden's et al. [5] of using byte code and whole projects, call graphs are built directly from source code files inside of a code change, which does not have the assumptions of being able to compile or have access to all project files. It is important to not require project compilation at each code change because it is an expensive operation as well as code change effects may cause the project to be unable to compile. Using source files also allowed an update to the call graph with changed files as opposed to completely rebuilding at every code change. This creates a rolling call graph which is used to show method hierarchy at each code change inside a project opposed to a static project view. As some method invocations may only be determined at run time, all possible method invocations are considered for these types of method calls while constructing the call graph.

The code change effect, if any, to the call hierarchy is now found. The Git software repository is used to determine what changes were made to any given file inside a code change. Specifically, methods modified by a code change are searched for. The call graph is then used to determine which methods call those that have been changed, which gives the code change technical dependencies.

These procedures result in a technical network based on contributor method ownership inside a call hierarchy effected by a code change (Figure 2.1 left hand side). The network is then simplified by only using edges between developers, since I am only interested in discovering the failure inducing edges between developers and not the methods themselves (Figure 2.1 right hand side). This is the final technical network.

Identifying Failure Inducing Developer Pairs

To identify failure inducing developer pairs (edges) inside technical networks, edges in relation to discovered code change failures are now analysed. To determine whether a code change was a success or failure (introduce a software failure), the approach of Sliwerski et al. [51] is used. The following steps are then taken:

1. Identify all possible edges from the technical networks.

2. For each edge, count occurrences in technical networks of failed code changes.

3. For each edge, count occurrences in technical networks of successful code changes.

4. Determine if the edge is related to success or failure.

To determine an edge's relation to success or failure, the value FI (failure index) which represents the normalized chance of a code change failure in the presence of the edge, is created.

$$\text{FI} = \frac{\text{edge}_{failed}/\text{total}_{failed}}{\text{edge}_{failed}/\text{total}_{failed} + \text{edge}_{success}/\text{total}_{success}} \tag{2.1}$$

Developer pairs with the highest FI value are said to be failure inducing structures inside a project. These edges are stored in Table 2.1. A Fisher Exact Value test is also preformed on edge appearance in successful and failed code changes, and non-appearance in successful and failed code changes to only consider statistically significant edges (Table 2.1's p-value).

2.1.3 Results

To illustrate the use of the approach, I conducted a case study of the Hibernate-ORM project, an open source Java application hosted on GitHub[1] with issue tracking performed by Jira[2].

This project was chosen because the tool created only handles Java code and it is written in Java for all internal structures and control flow and uses Git for version control. Hibernate-ORM also uses issue tracking software which is needed for determining code change success or failure [51].

In Hibernate-ORM, 27 statistically significant failure inducing developer pairs (FI value of 0.5 or higher) were found out of a total of 46 statistically significant pairs that existed over the project's lifetime. The pairings are ranked by their respective FI values (Table 2.1).

Pair	Successful	Failed	FI	P-Value
(Daniel, Anne)	0	14	1.0000	0.0001249
(Carl, Bret)	1	12	0.9190	0.003468
(Emily, Frank)	1	9	0.8948	0.02165

Table 2.1: Top 3 failure inducing developer pairs found.

2.1.4 Conclusions of Study

Technical dependencies are often used to predict software failures in large software system [35, 46, 63]. This study has presented a method for detecting failure inducing pairs of developers inside of technical networks based on code changes. These developer pairs can

[1] https://github.com/

[2] http://www.atlassian.com/software/jira/overview

be used in the prediction of future bugs as well as provide coordination recommendations for developers within a project.

This study however, did not consider the technical dependencies themselves to be the root cause of the software failures. This study focused purely on developer ownership of software methods and the dependencies between developers as the possible root cause of the failures. To study this root cause further, a study of indirect conflicts and their relationship to developer code ownership will be conducted.

2.2 Study 2: Awareness with Impact

In response to Study 1, a second investigation was conducted. Study 1 revealed that pairs of developers can be used around technical dependencies in order to predict bugs. The natural follow up to these findings was to conduct a study of indirect conflicts surrounding these developer pairs that are involved in source code changes. These indirect conflicts were primarily studies through the notion of task awareness.

Tools have been created to attempt to solve task awareness related issues with some success [4,33,48,59]. These tools have been designed to solve task awareness related issues at the direct conflict level. Examples of direct conflict awareness include knowing when two or more developers are editing same artifact, finding expert knowledge of a particular file, and knowing which developers are working on which files. On the other hand, task awareness related issues at the indirect conflict level have also been studied, with many tools being produced [3,47,50,53]. Examples of indirect conflict awareness include having one's own code effected by another developer's source code change or finding out who might be indirectly effected by one's own code change. Previous interviews and surveys conducted with software developers have shown a pattern that developers of a software project view indirect conflict awareness as a high priority issue in their development [3,14, 26,49].

Indirect conflicts arising in source code are inherently difficult to resolve as most of the time, source code analysis or program slicing [55] must be performed in order to find relationships between technical objects which are harmed by changes. While some awareness tools have been created with these indirect conflicts primarily in mind [3, 53], most have only created an exploratory environment which is used by developers to solve conflicts which may arise [50]. These tools were not designed to detect indirect conflicts that arise and alert developers to their presence inside the software system. Sarma et al. [47] has started to work directly on solving indirect conflicts, however, these products are not designed to handle internal structures of technical objects.

In this study, I report on research into supporting developer pairs in indirect conflicts and present the design, and implementation of the tool *Impact*, a web based tool that aims at detecting indirect conflicts among developers and notifying the appropriate members involved in these conflicts. Through Impact and its evaluation I ask:

RQ *Can indirect conflicts be supported through an awareness mechanism which leverages pairs of developers whose changing technical dependencies statistically relate to bugs?*

By leveraging technical relationships inherent of software projects with method call graphs [38] as well as detecting changes to these technical relationship through software configuration management (SCM) systems, *Impact* is able to detect indirect conflicts as well as alert developers involved in such conflicts in task awareness while limiting information overload by using design by contract [40] solutions to method design. While this study outlines *Impact's* specific implementation, its design is rather generic and can be implemented in similar indirect conflict awareness tools.

After a brief evaluation of *Impact* with two student software teams, it was found that *Impact* suffers from information overload and a high false positive rate which turn out to be quite large problems found in many other indirect conflict tools [6, 30, 47, 50, 53].

2.2.1 Related Work

Although there is an abundance of awareness tools developed in research today, only a handful have made an attempt to examine indirect conflicts. Here, I will outline four of the forefront projects in indirect conflicts and how these projects have influenced the decision making process in the design and implementation of *Impact*.

I first start with both Codebook [3] and Ariadne [53]. These projects produce an exploratory environment for developers to handle indirect conflicts. Exploratory pertains to the ability to solve self determined conflicts, meaning that once a developer discovers a conflict, they can use the tool as a type of lookup table to solve their issue. Codebook is a type of social developer network that relates developers to source code, issue repositories and other social media while Ariadne only examines their source code for developer to source code association. Through Codebook, developers become owners of source code artifacts. Both projects also use program dependency graphs [31] in order to relate technical artifacts to each other. These projects make use of method call graphs in order to determine which methods invoke others which forms the basis for linking source code artifacts creating a directed graph. While these projects can be great tools for solving indirect conflicts which may arise, by querying such directed graphs to view impacts of conflict creating code, they lack the ability to detect potential conflicts on their own.

A serious attempt at both detecting and informing developers of indirect conflicts is the tool Palantir [47]. Palantir monitors developer's activities in files with regards to class signatures. Once a developer changes the signature of a class, such as by modifying changes in name, parameters, or return values of public methods, any workspace of other developers which are using that class will be notified. Palantir utilizes a push-based event model [23] which seems to be a favored collection system among awareness tools. Sarma et al. [47] also developed a generic design for future indirect conflict awareness tools. However, Palantir falls short in its collection and distribution mechanisms. First, Palantir only considers "outside" appearance of technical objects, being their return types, parameters, etc. Secondly, Palantir only delivers detected conflicts to developers who are presently viewing or editing the indirect object while other developers who have used the modified class previously are not notified.

I will lastly examine the tool CASI [50] which uses a sphere of influence for each developer to determine how source code changes are indirectly related to other components of the software. CASI uses dependency slicing [2] instead of the call graphs in Ariadne [53] which gives dependencies among all source code entities. This provides a verbose output of dependencies when source code is changed. CASI also implements a visualization where a developer can see what parts of a software projects he or she may be effecting with the source code change. This allows the developers themselves to go and fix potential issues elsewhere in the project before the code change is committed to the software repository. While CASI covers great ground in its approach, it still leaves the issue of information overload, although attempts were made to solve this by having severity levels of indirect conflicts presented to the user.

2.2.2 Impact

This section will proceed by giving a detailed outline of *Impact* in both its design and implementation. The design of *Impact* was created to be a generic construct which can be applied to other indirect conflict awareness tools while the implementation is specific to the technical goals of *Impact*.

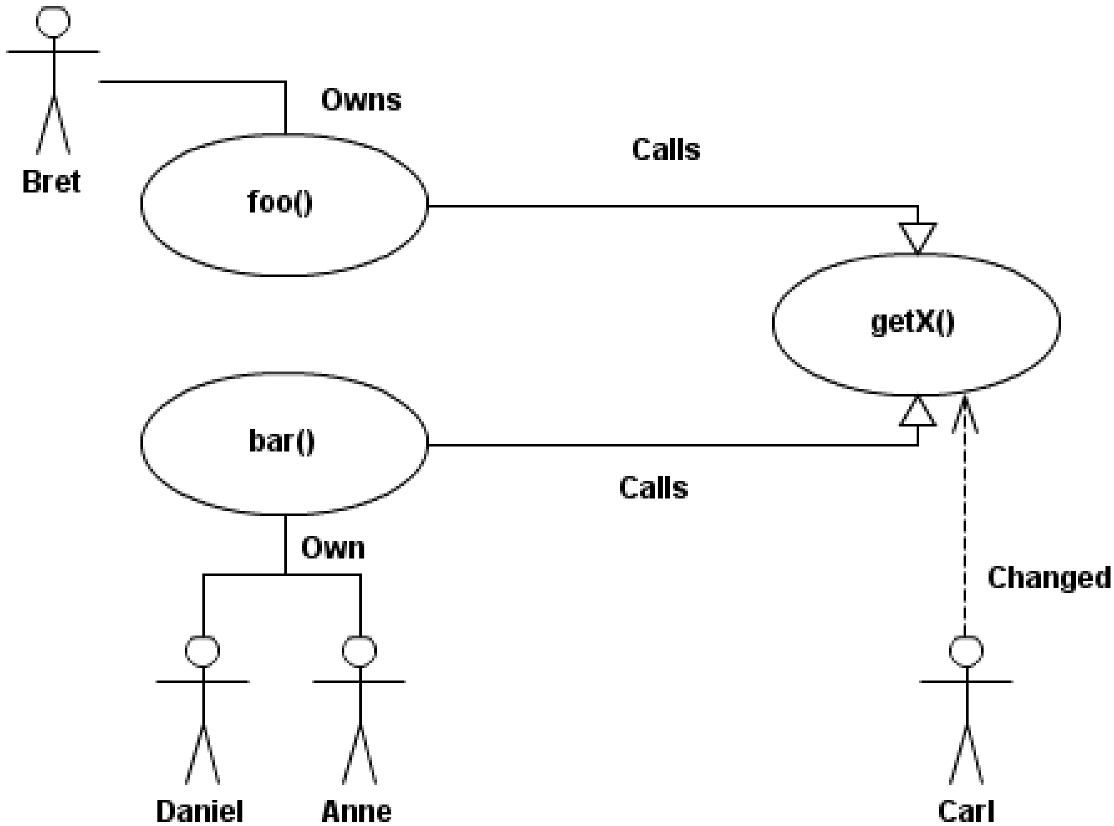

Figure 2.2: Technical object directed graph with ownership

Design

Compared to tool design for direct conflicts, the major concern of indirect conflict tools is to relate technical objects to one another with a "uses" relationship. To say that object 1 uses object 2 is to infer a technical relationship between the two objects which can be used in part to detect indirect conflict that arise from modifying object 2. This kind of relationship is modeled based on directed graphs [31]. Each technical object is represented by node while each "uses" relationship is represented by a directed edge. This representation is used to indicate all indirect relationships within a software project.

While technical object relationships form the basis of indirect conflicts, communication between developers is my ultimate goal of resolving such conflicts (as was seen in Study 1). This being the case, developer ownership must be placed on the identified technical objects. With this ownership, I now infer relationships among developers based on their technical objects "uses" relationship. Developer A, who owns object 1, which uses object 2 owned by developer B, may be notified by a change to object 2's internal workings. Most, if not all, ownership information of technical objects can be extracted from a project's source code repository (CVS, Git, SVN, etc.).

Finally, the indirect conflict tool must be able to detect changes to the technical objects defined above and notify the appropriate owners to the conflict. Two approaches have been proposed for change gathering techniques: real time and commit time [23]. I propose the use of commit time information gathering as it avoids the issue of developers overwriting previous work or deleting modifications which would produce information for changes that no longer exist. However, the trade off is that indirect conflicts must be committed before detected, which results in conflicts being integrated into the system before being able to be dealt with as opposed to catching conflicts before they happen. At commit time, the tool must parse changed source code in relation to technical artifacts in the created directed graph detailed above. Where *Impact's* design differs from that of Palantir's is that the object's entire body (method definition and internal body) is parsed, similar to that of CASI [50], at commit time, as opposed to real time, to detect changes anywhere in the technical object. This is a first design step towards avoiding information overload. Once technical objects are found to be changed, appropriate owners of objects which use the changed object should be notified. In Figure 2.2, Carl changes method (technical object) 1, which effects methods 2 and 3 resulting in the alerting of developers Bret, Daniel, and Anne. I have opted to alert the invoking developers rather than the developer making the change to potential solutions as my conflicts are detected at commit time and this supports the idea of a socio-technical congruence [36] from software structure to communication patterns in awareness systems.

With this three step design: (i) creating directed graphs of technical objects, (ii) assigning ownership to those technical objects, and (iii) detecting changes at commit time and the dissemination of conflict information to appropriate owners, I believe a wide variety of indirect conflict awareness tools can be created or extended.

Implementation

For *Impact's* implementation, I decided to focus on methods as my selected technical objects to infer a directed graph from. The "uses" relationship described above for methods is method invocation. Thus, in my constructed dependency graph, methods represent nodes and method invocations represent the directed edges. In order to construct this directed graph, abstract syntax trees (ASTs) are constructed from source files in the project.

Once the directed graph is constructed, I must now assign ownership to the technical objects (methods) as per the design. To do this, I simply query the source code repository. In this case I used Git as the source code repository, so the command *git blame* is used for querying ownership information. (Most source code repositories have similar commands and functionality.) This command returns the source code authors per line which can be used to assign ownership to methods.

To detect changes to technical objects (methods), I simply use a commit's *diff* which is a representation of all changes made inside a commit. I can use the lines changed in the *diff* to find methods that have been changed. This gives cause of potential indirect conflicts. I now find all methods in the directed graphs which invoke these changed methods. These are the final indirect conflicts.

Once the indirect conflicts have been found, I use the ownership information of technical objects to send notifications to those developers involved in the indirect conflict. All owners of methods which invoke those that have been changed are alerted to the newly changed method. Impact can been seen in Figure 2.3, the user interface of *Impact*. Here, in an RSS type feed, the changing developer, time of change, changed method, invoking methods, and commit message are all displayed. The weight provided is the percent changed of changed method multiplied by ownership of the invoking method. This allows developers to filter through high and low changes affecting their own source code.

2.2.3 Evaluation

To fully evaluate both the generic design of detecting and resolving indirect conflicts as well as *Impact*, extensive testing and evaluation must be performed. However, I felt that a simple evaluation is first needed to assess the foundation of *Impact*'s design and claims about indirect conflicts at the method level.

I performed a user case study where I gave *Impact* to two small development teams composed of three developers. Each team was free to use *Impact* at their leisure during their development process, after which interviews were conducted with lead developers from each development team. The interviews were conducted after each team had used *Impact* for three weeks.

I asked lead developers to address two main concerns: do indirect conflicts pose a threat at the method level (e.g. method 1 has a bug because it invokes method 2 which has had its implementation changed recently), and did *Impact* help raise awareness and promote quicker conflict resolution for indirect conflicts. The two interviews largely supported the expectation of indirect conflicts posing a serious threat to developers, especially in medium

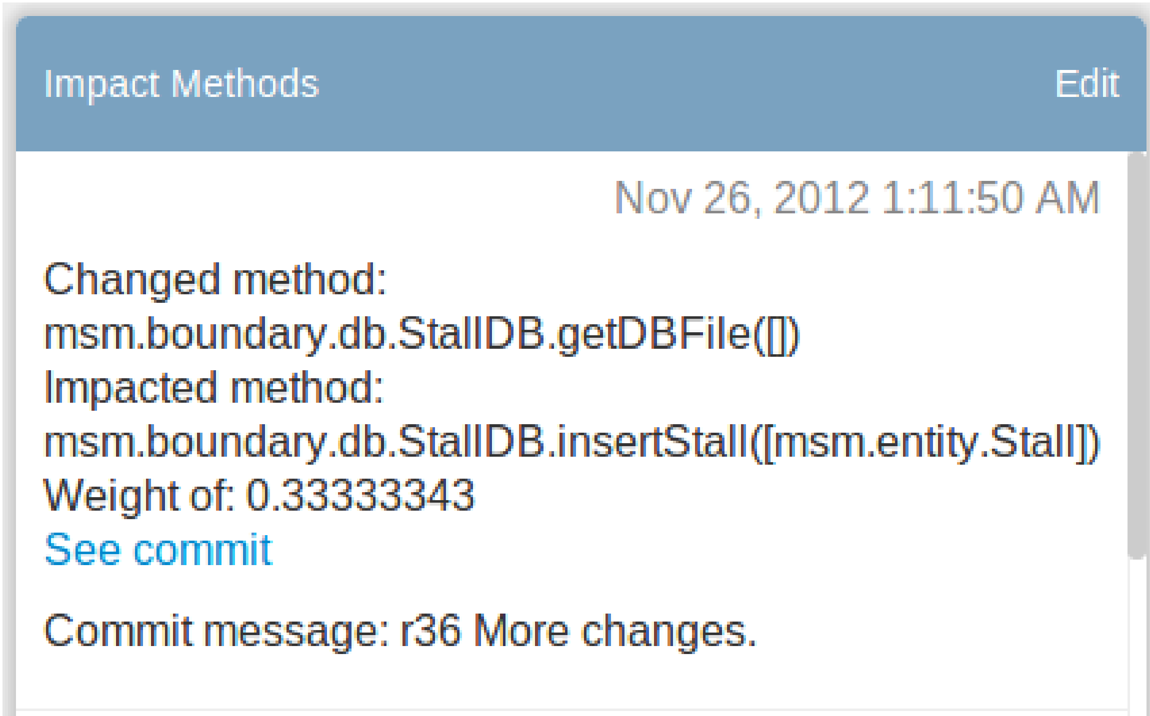

Figure 2.3: *Impact's* RSS type information feed.

to large teams or projects as opposed to the small teams which they were a part of. It was also pointed out that method use can be a particularly large area for indirect conflicts to arise. However, it was noted that any technical object which is used as an interface to some data construct or methodology, database access for instance, can be a large potential issue for indirect conflicts. Interview responses to *Impact* were optimistically positive, as interviewees stated that *Impact* had potential to raise awareness among their teams with what other developers are doing as well as the influence it has on their own work. However, *Impact* was shown to have a major problem of information overload. It was suggested that while all method changes were being detected, not all are notification worthy. One developer suggested to only notify developers if the internal structure of a method changes due to modification to input parameters or output parameters. In other words, the boundaries of the technical objects (changing how a parameter is used inside the method, modifying the return result inside the method) seem to be more of interest than other internal workings. More complex inner workings of methods were also noted to be of interest to developers such as cyclomatic complexity, or time and space requirements.

These two studies have shown that my design and approach to detecting and alerting developers to indirect conflicts appear to be on the correct path. However, *Impact* has clearly shown the Achilles heel of indirect conflict tools, which is information overload because of an inability to detect "notification worthy" changes.

2.2.4 Threats to Validity

Because of the tool validation nature of this work, I chose participant interviews as my research validation method, which has some implications regarding the limitations and threats to validity of this study. While I did have some positive results regarding the potential of Impact in this study, populations studied from outside of this study's participants may add new insights into the pool of findings. As a result of this, findings from this study may not relate to everyone or generalize to outside of the group of participants involved in this study.

I conducted this case study with undergraduate students at the University of Victoria. This being said, participants may not have had enough real world experience to validate this study at an industry level. The students were also consumed with their regular course work which could have limited the time spent using Impact and the enthusiasm put forward while conducting this study.

To counter this, my study was conducted purely on a volunteer basis to eliminate those participants which may have been to busy to focus on the study or provide appropriate feedback where needed.

2.2.5 Conclusions of Study

In this study, I have proposed a generic design for the development of awareness tools in regards to handling indirect conflicts. I have presented a prototype awareness tool, *Impact*, which was designed around the generic technical object approach. However, *Impact* suffered from information overload, in that it had too many notifications sent to developers.

A potential solution to information overload comes from the ideas of Meyer [40] on "design by contract". In this methodology, changes to method preconditions and postconditions are considered to be the most harmful. This includes adding conditions that must be met by both input and output parameters such as limitations to input and expected output. To achieve this level of source code analysis, the ideas of Fluri et al. [24] can be used on the previously generated ASTs for high granularity source code change extraction when determining if preconditions or postconditions have changed.

Aside from better static analysis tools for examining source code changes, the results of this study potentially imply a lack of understanding into the root causes of indirect conflicts.

A theme of information overload to developers continues to crop up in indirect conflicts, of which the root cause should be examined in future studies.

Chapter 3

Exploring Indirect Conflicts

Through the two previous studies, I have shown that developers linked indirectly in source code changes can be statistically related to software failures. In the attempts of mitigating these loses through added awareness, I implemented an indirect conflict tool called *Impact*. However, *Impact* ultimately suffered from information overload as seen in its evaluation which was caused by false positives and scalability of the tool.

While other indirect conflict tools have shown potential from developer studies, some of the same problems continue to arise throughout most, if not all tools. The most prevalent issue is that of information overload and false positives. Through case studies, developers have noted that current indirect conflict tools provide too many false positive results leading to information overload and the tool eventually being ignored [47, 50]. A second primary issue is that of dependency identification and tracking. Many different dependencies have been proposed and used in indirect conflict tools such as method invocation [53], and class signatures [47] with varying success, but the identification of failure inducing changes, other than those which are already identifiable by other means such as compilers, and unit tests, to these dependencies still remains an issue. Dependency tracking issues are also compounded by the scale of many software development projects leading to further information overload.

Social factors such as Cataldo et al's. [10] notion of socio-technical congruence, have also been leveraged in indirect conflict tools [3, 6, 36]. However, issues again of information overload, false positives, dependencies (in developer organizational structure), and scalability come up.

While these issues of information overload, false positives, dependencies, and scalability continue to come up in most indirect conflict tools, only a handful of attempts have been made at rectifying these issues or finding the root causes [30, 34]. In order to fully understand the root causes of information overload, false positives, and scalability issues in regards to indirect conflicts in this study, I examine and determine what events occur to cause indirect conflicts, when they occur, and if conditions exist to provoke more of these events. By determining the root causes of source code changes in indirect conflicts, we may be able to create indirect conflict tools which have filtered monitoring in order to only detect those changes with a high likelihood of causing indirect conflicts. I then determine what mitigation strategies developers currently use as opposed to those created by researchers. Since developers have identified indirect conflicts as a major concern for themselves, but at the same time are not using the tools put forth by researchers, I wish to find what they use to mitigate indirect conflicts. Through these findings, we can create tools which are similar to those already in use by developers in the hopes of a higher adoption rate of tools produced by researchers. Finally, I examine what can be accomplished moving forward with indirect conflicts in both research and industry.

To explore and answer the research goals listed above, I performed a study (Section 3.1) in which I interviewed 19 developers from across 12 commercial and open source projects, followed by a confirmatory questionnaire of 78 developers, and 5 confirmatory interviews.

Based on some of the findings (to be seen in details in Section 3.1) I performed a follow up study which did not relate directly to the themes of this dissertation but helped strengthen the results found in Study 3. Some results of Study 3 showed that indirect conflict tools should take into account a contextual setting of development progression in software projects to better inform developers of potential indirect conflicts. In other words, an indirect conflict tool should be able to tell what phase of development inside the development life cycle a project is currently active in. In order to better explore and support this finding, I performed a complimentary study of software evolutionary trends (Section 3.2). In Study 4, I perform a case study of 10 open source projects in order to study their source code change trends surrounding major release points throughout their history. I studied 26 change trends quantitatively and 4 change trends qualitatively, and identified a core group of 9 change trends which occur prominently at major release points of the projects studied. These change trends can provide context as to when indirect conflicts are more likely to occur based on the findings from Study 3 as I found that indirect conflicts tend to be become less frequent near major release and more frequent after a release or at the start of a new development cycle. The findings of this study can be applied over the lifetime of a project to determine the probability of indirect conflicts occurring and thus aiding developers in dealing with indirect conflicts in their projects.

3.1 Study 3: An Exploration of Indirect Conflicts

I want to understand why it is so hard to tackle indirect conflicts, specifically through tool based solutions. To do so, I take a step back and intend to obtain a broader view of indirect conflicts with a large field study of practitioners. I investigate the root causes of information overload, false positives, and scalability issues in regards to indirect conflicts to better understand why indirect conflict tools fail to achieve the success of other domain tools. I determine: general events which cause indirect conflicts, when they occur, if compounding conditions exist, mitigation strategies developers use, and what practitioners want from new tools. My research questions for this particular study are as follows:

RQ1 *What are the types, factors, and frequencies of indirect conflicts?*

RQ2 *What mitigation techniques are used by developers in regards to indirect conflicts?*

RQ3 *What do developers want from future indirect conflict tools?*

I interviewed 19 developers from across 12 commercial and open source projects, followed by a confirmatory questionnaire of 78 developers, and 5 confirmatory interviews, in order to answer the aforementioned questions. My findings indicate that: indirect conflicts occur frequently and are likely caused by software contract changes and a lack of understanding, developers tend to prefer to use detection and resolution processes or tools over those of prevention, developers do not want awareness mechanisms which provide non actionable results, and there exists a gap in software evolution analytical tools from the reliance on static analysis resulting in missed context of indirect conflicts.

3.1.1 Related Work

Many indirect conflict tools have been created, tested, and published. Sarma et al. [47] created Palantir, which can both detect potential indirect conflicts, at the class signature level, and alert developers to these conflicts. Palantir represented one of the first serious attempts at aiding developers towards indirect conflicts. Holmes et al. [30] take it one step further with their tool YooHoo, by detecting fine grained source code changes, such as method return type changes, and create a taxonomy for different types of changes and their proneness to cause indirect conflicts. This tool and its taxonomy had a severely reduced false positive rate, however, the true positives detected may already be detectable by current tools such as compilers and unit testing. The tool Ariadne [53] creates an environment where developers can see how source code changes will affect other areas of a project at the method level, using method call graphs, showing where indirect conflicts may occur. This type of exploratory design has been used often in the visualization of indirect conflict tools, allowing developers a type of search area for their development needs. Another indirect conflict tool, CASI [50], utilizes dependency slicing [2] instead of method call graphs to provide an environment to see what areas of a project are being affected by a source code change. Most of these tools have all shown to have the same common difficulties of scalability, false positives, and information overload, which were explored in this study. My own tool Impact! [18] also suffered these same fates.

Since Cataldo et al. [10] have shown that socio-technical congruence can be leveraged to improve task completion times, many indirect conflict tools support the idea of a socio-technical congruence [36] in order to help developers solve their indirect conflict issues through social means [3] [6]. Begel et al. [3] created Codebook, a type of social developer network related directly to source code artifacts, which can be used to identify developers and expert knowledge of the code base. Borici et al. [6] created ProxiScientia which used technical dependencies between developers to create a network of coordination needs. Socio-technical congruence however, is largely unproven in regards to its correlation to software quality [37] and again the problems of scalability and information overload become a factor.

For developer interest in regards to software modifications, Kim [34] found that developers wanted to know whose recent code changes semantically interfere with their own code changes, and whether their code is impacted by a code change. Kim found that developers are concerned with interfaces of objects and when those interfaces change. Kim also identified the same issues towards information overload through false positives with developers noting “I get a big laundry list... I see the email and I delete it”. Kim’s field study does however fall short in actually creating a resolution for indirect conflicts, or finding new concerns of developers which are not already detected by compilation or other static analysis tools. For impact awareness, DeSouza et. al. [15] found that developers use their personal knowledge of the code base to determine the impact of their code changes on fellow developers, teams, and projects. However, DeSouza does not study which types of changes (types, frequencies, compounding factors) developers should concern themselves with more in terms of using their personal knowledge to stop indirect conflicts. DeSouza also does not study formal mitigation strategies, or resolutions of indirect conflicts directly.

This study is intended to fill the gap which has been left by aforementioned tools papers as well as the field study paper. I will not only study why information overload, false positives, and scalability are such difficult problems to tackle in indirect conflict tools, but I will also study how developers currently deal with indirect conflicts in practice through their mitigation strategies, their largest concerns, and their ideas for future suggestions in regards to indirect conflicts.

3.1.2 Methodology

I performed a mixed method study in three parts. First, a round of semi-structured interviews were conducted which addressed my 3 main research questions. Second, a questionnaire was conducted which was used to confirm and test what was theorized from the interviews on a larger sample size as well as obtain larger developer opinion of the subject. Third, 5 confirmatory interviews were conducted by re-interviewing original interview participants to once again confirm my insights. I used grounded theory techniques to analyze the information provided from all three data gathering stages.

Interview Participants

My interview participants came from a large breadth of both open and closed source software development companies and projects, using both agile and waterfall based methodology, and from a wide spectrum of organizations, as shown in Table 3.1. My participants were invited based on their direct involvement in the actual writing of software for their respective companies or projects. These participants' software development experience ranged from 3-25 years of experience with an average of 8 years of experience. In addition to software development, some participants were also invited based on their experience with project management. See Table 3.1 for more demographic details.

Interview Procedure

Participants were invited to be interviewed by email and were sent a single reminder email one week after the initial invitation if no response was made. I directly emailed 22 participants and conducted 19 interviews. Interviews were conducted in person when possible and recorded for audio content only. When in person interviews were not possible, one of Skype or Google Hangout was used with audio and video being recorded but only audio being stored for future use.

Interview participants first answered a number of demographic questions. I then asked them to describe various software development experiences regarding my three research

Company	# of Participants	Software Development Experience (years)	Development Process	Software Access	Current Language Focuses
Amazon	2	5, 7	Agile	Closed source	C++
Exporq Oy	1	8	Agile	Closed source	Ruby, JavaScript
Fireworks Design	1	6	Agile	Closed source	JavaScript
Frost Tree Games	1	4	Agile	Closed source	C#
GNOME	1	13	Agile	Open source	C
James Evans and Associates	5	3, 3, 3, 4, 13	Waterfall	Closed source	Oracle Forms
Kano Apps	1	10	Agile	Closed source	JavaScript, PHP
IBM	2	5, 18	Agile	Open and closed source	Java, JavaScript
Microsoft	2	6, 10	Agile	Closed source	C#
Mozilla	1	25	Agile	Open source	C++, JavaScript
Ruboss	1	5	Agile	Closed source	JavaScript
Subnet Solutions	1	5	Agile	Closed source	C++

Table 3.1: Demographic information of interview participants.

questions. Specifically, ten semi-structured topics directly related to my research questions guided my interview:

- What tools are used for dependency tracking?
- What processes are used for preventing indirect conflicts?
- What artifact dependency levels are analyzed and where can conflicts arise?

- How are software dependencies found?
- Give examples of indirect conflicts from real world experiences.
- How are indirect conflicts detected or found?
- How are indirect conflict issues solved or dealt with?
- Opinion of preemptive measures to prevent indirect conflicts.
- What types of changes are worth a preemptive action?
- Who is responsible for fixing or preventing indirect conflicts?

While each of the 10 topics had a number of starter questions, interviews largely became discussions of developer experience and opinion as opposed to direct answers to any specific question. However, not all participants had strong opinions or any experience on every category mentioned. For these participants, answers to the specific categories were not required or pressed upon. I attribute any non answer by a participant to either lack of knowledge in their current project pertaining to the category or lack of experience in terms of being a part of any one software project for extended periods of time.

Questionnaire Participants

My questionnaire respondents were different from my interviewees. I invited my questionnaire participants from a similar breadth of open and closed source software development companies and projects as the interviews participants with two main exceptions. The software organizations that remained the same between interview and questionnaire were: Mozilla, The GNOME Project, Microsoft Corporation, Subnet Solutions, and Amazon. Participants who took part in the round of interviews were not invited to the questionnaire but were asked to act as a point of contact for other developers in their team, project, or organization who may be interested in completing the questionnaire. Further, two other groups of developers were asked to participate as well, these being GitHub users as well as Apache Software Foundation (Apache) developers. The GitHub users were selected based on large amounts of development activity on GitHub and the Apache developers were selected based on their software development contributions on specific projects known to be used heavily utilized by other organizations and projects.

Questionnaire Procedure

Questionnaire participants were invited to participate in the questionnaire by email. No reminder email was sent as the questionnaire responses were not connected with the invitation email addresses and thus participants who did respond could not be identified. I directly emailed 1300 participants and ended with 78 responses giving a response rate of 6%. I attribute the low response rate with: the questionnaires were conducted during the months of July and August while many participants may be away from their regular positions. Also, my GitHub and Apache participants could not be verified as to whether or not they actively support the email addresses used in the invitations. In addition, the questionnaire was considered by some to be long and required more development experience than may have been typical of some of those invited to participate.

The questionnaire I designed [1] was based on the insights I obtained from the round of interviews, and was intended to confirm some of these insights but also to broaden them to a larger sample size of developers who may have similar or different opinions from those already acquired from the interviews. The questionnaire went through two rounds of piloting. Each pilot round consisted of five participants, who were previously interviewed, completing the questionnaire with feedback given at the end. Not only did this allow me to create a more polished questionnaire, but it also allowed the previously interviewed developers to examine the insights I developed.

The question topics asked in the questionnaire were:

- What frequency do ICs occur at around different project milestones?
- How does team size affect IC frequency?
- What software change types do developers care about?
- What processes are used for preventing ICs?
- What tools are used for detecting ICs?
- What tools are used for debugging ICs?

Data Analysis

To analyze both the interview and questionnaire data, I used grounded theory techniques as described by Corbin and Strauss [13]. Grounded theory is a qualitative research method-

[1] http://thesegalgroup.org/people/jordan-ell/iced_survey/

ology that utilizes *theoretical sampling* and *open coding* to create a theory "grounded" in the empirical data. For an exploratory study such as mine, grounded theory is well suited because it involves starting from very broad and abstract type questions, and making refinements along the way as the study progresses and hypotheses begin to take shape. Grounded theory involves realigning the sampling criteria throughout the course of the study to ensure that participants are able to answer new questions that have been formulated in regards to forming hypotheses. In my study being presented, data collected from both interviews and questionnaires (when open ended questions were involved) was analyzed using open and axial coding. Open coding involves assigning codes to what participants said at a low sentence level or abstractly at a paragraph or full answer level. These codes were defined as the study progressed and different hypotheses began to grow. I finally use axial coding it order to link my defined codes through categories of grounded theory such as context and consequences. In Section 3.1.4, I give a brief evaluation of my studying using 3 criteria that are commonly used in evaluating grounded theory studies.

Validation

Following my data collection and analysis, I re-interviewed 5 of my initial interview participants in order to validate my findings. I confirmed my findings as to whether or not they resonate with industry participants' opinions and experiences regarding indirect conflicts and as to their industrial applicability. Due to limited time constraints of the interviewed participants, I could only re-interview five participants. Those that were re-interviewed came from the range of 5-10 years of software development experience. Re-interviewed participants were given my 3 research questions along with results and main discussion points, and asked open ended questions regarding their opinions and experiences to validate my findings. I also evaluated my grounded theory approach as per Corbin and Strauss' [13] list of criteria to evaluate quality and credibility. This evaluation can be seen in Section 3.1.4

3.1.3 Results

I now present my results of both the interviews and questionnaires conducted in regards to my 3 research questions outlined in this chapter and Chapter 1. I restate each research question, followed by my quantitative and qualitative results from which I draw my discussion to be seen in Chapter 4. In each subsection, quantitative data given refers to interviews conducted unless explicitly said otherwise.

What are the types, factors, and frequencies of indirect conflicts?

The most common occurrence of indirect conflict is when a software object's contract changes. The frequency of indirect conflicts, while usually high, decreases as a stable point is reached in the development cycle. The frequency of indirect conflicts is compounded by the number of developers working on a project.

12 developers believe that a large contributing factor to the cause of indirect conflicts comes from the changing of a software object's contract. Object contracts are, in a sense, what a software object guarantees, meaning how the input, output, or how any aspect of the object is guaranteed to work; made famous by Eiffel Software's [2] "Design by Contract"™. In light of object contracts, 14 developers gave examples of indirect conflicts they had experienced which stemmed from not understanding the far reaching ramifications of a change being made to an object contract towards the rest of the project. Of those 14, 3 dealt with the changing of legacy code, with one developer saying "legacy code does not change because developers are afraid of the long range issues that may arise from that change". Another developer, in regards to changing object contacts stated "there are no changes in the input or changes in the output, but the behavior is different". Developers also noted that the conflicts that do occur tend to be quite unique from each other and do not necessarily follow common patterns.

In regards to object contract changes, 9 developers currently working with large scale database applications listed database schemas as a large source of indirect conflicts while 5 developers that work on either software library projects or are in test said that methods or functions were the root of their indirect conflict issues. 7 developers mentioned that indirect conflicts occur when a major update to an external project, library, or service occurs with one developer noting "their build never breaks, but it breaks ours". Some other notable indirect conflict artifacts were user interfaces in web development and full components in component base game architecture.

11 developers explained that indirect conflicts occur "all the time" in their development life cycle with a minimum occurrence of once a week, with more serious issues tending to occur once a month. To confirm this, 64% of questionnaire participants answered that indirect conflicts occur bi-weekly or more frequent, with the 25% saying that weekly occurrences are most common (seen in Table 3.2). 5 questionnaire participants also stated that the stage of the development cycle can greatly cause the frequency of indirect conflicts to differ.

[2] http://www.eiffel.com/

Occurrences	Daily	Weekly	Bi-Weekly	Monthly	Bi-Monthly	Yearly	Unknown
In general	18%	25%	21%	16%	3%	5%	11%
Early stages of a development	32%	18%	4%	5%	0%	5%	36%
Before the first release	13%	29%	6%	8%	1%	3%	40%
After the first release	6%	18%	8%	18%	1%	5%	44%
Late stages of development	6%	5%	5%	18%	8%	12%	46%

Table 3.2: Results of questionnaire as to how often indirect conflicts occur, in terms of percentage of questionnaire participants.

12 developers said that when a project is in the early stages of development, indirect conflicts tend to occur far more frequently than once a stable point is reached. Developers said “At a stable point we decided we are not going to change [this feature] anymore. We will only add new code instead of changing it.” and “the beginning of a project changes a lot, especially with agile”. Questionnaire participants also added “indirect conflicts after a release depend on how well the project was built at first”, “[indirect conflicts] tend to slow down a bit after a major release, unless the next release is a major rework.”, and “[indirect conflicts have] spikes during large revamps or the implementation of cross-cutting new features.” in order to confirm mu interview results. Questionnaire participants also answered that indirect conflicts are more likely to occur before the first major release rather than after at the daily and weekly occurrence intervals as seen in Table 3.2.

In terms of organizational structure, questionnaire participants answered that as a project becomes larger and more developers are added, even to the point that multiple teams are formed, indirect conflicts become more likely to occur. However, indirect conflicts still occur at a lower number of developers as well with even 43% of developers saying they are like to occur in single developer projects. This can be seen in Table 3.3.

What mitigation techniques are used by developers in regards to indirect conflicts?

Three preventative techniques are used to mitigate indirect conflicts: design by con-

Environment of conflicts	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Developing alone (conflicts in own code)	18%	20%	19%	24%	19%
Developing in a team between 2 - 5 developers (inter-developers conflicts)	3%	8%	22%	49%	18%
Developing in a multi team environment (inter-team conflicts)	1%	11%	14%	39%	35%

Table 3.3: Questionnaire results about development environments in which indirect conflicts are likely to occur, in terms of percentage of questionnaire participants.

tract, add and deprecate, and personal experience. For catching indirect conflicts, developers use: testing (unit and integration) with proper use case coverage, scheduled build processes, and static analysis tools built into IDEs. For resolving indirect conflicts, developers use: historical error and information logs, compiler and debugger tools, as well as static analysis tools built into IDEs.

In terms of preventative processes used for indirect conflicts, 3 major components were found. First, design by contract is heavily used by interviewed developers as a means to avoid indirect conflicts or understand when they are likely to occur. The use of design by contract here means that developers tend not to change an object's contract when possible, and that an object's contract is used as a type of documentation towards awareness of the software object. One developer stated that "design by contract was invented to solve this problem and it does it quite well", while another noted that software object contracts do solve the problem in theory, but that doesn't mean that problems don't still occur in practice. Second, 21% of developers mentioned that the add and deprecate model is used to prevent indirect conflicts once the project, feature, or component has reached a stable or mature point. Add and deprecate meaning instead of editing code, the developer simply clones old code (if needed), and edits the clone while slowly phasing out the out the old code in subsequent releases or ad needed. This allows other software to use the older versions of software objects which remain unchanged, thus avoiding indirect conflicts. Lastly, pure developer experience was mentioned with 7 developers mentioning that when planning code changes, either a very experienced member of the project was involved in the planning and has duties to foresee any indirect conflicts that may arise, or that developers must use their personal knowledge to predict where indirect conflicts will occur while implementing.

Of the the 37 questionnaire participants who could give positive identification of preventative processes for indirect conflicts, 27% said that individual knowledge of the code base and their impact of code change was used, 59% mentioned some form of design by contract or the testing of a methods contract, and 14% said that add and deprecate was used in their projects to avoid indirect conflicts. This is a confirmation of what was found in the interviews.

In regards to catching indirect conflicts, 13 developers mentioned forms of testing (unit, and integration) as the major component of catching indirect conflict issues, subscribing to the idea of "run the regression and integration tests and just see if anything breaks". The words "use case coverage" were constantly being used by developers when expressing how proper unit and integration tests should be written. Developers expressed that with proper use case coverage, most if not all indirect conflicts should be caught. In corroboration, 31% of questionnaire participants said build processes (either nightly builds or building the project themselves), and others mentioned code reviews while those dealing with a user interface mentioned user complaints from run time testing. The questionnaire participants confirmed these results with 49% mentioning forms of testing as the major tool used to catch indirect conflicts, 33% said build processes, while 31% used work their IDE or IDE plug-ins to catch indirect conflicts. Questionnaire participants also mentioned review process and personal expertise as factors of catching indirect conflicts.

Once an indirect conflict has occurred and developers need to resolve it, 14 developers said they checked historical logs to help narrow down where the problem could originate from. Most developers had the mindset of "Look at the change log and see what could possibly be breaking the feature.". The log most commonly referred to was the source code change log to see what code has been changed, followed by build failure or test failure logs to examine errors messages and get time frames of when events occurred. Of the questionnaire participants, 23% said they used native IDE tools and 21% said they use features of the language's compiler and debugger in order to solve indirect conflicts. Interestingly, only 13% of developers mentioned a form of communication with other developers in aid to solving these conflicts and only 4% mentioned the reading of formal documentation.

Through the processes and tools of prevention, detection, and resolution of indirect conflicts, it is important to note that most developers ascribe to the idea of "I work until something breaks", or taking a curative rather than preventative approach. This means that while developers do have processes and tool for prevention, they would rather spend their time at the detection and resolutions stages. One developer noted that preventative processes are "too much of a burden" while a project manager said "[with preventative process] you will spend all you time reviewing instead of implementing".

What do developers want from future indirect conflict tools?

Being that developers believe "good enough" solutions exist for preventing and detecting indirect conflicts developers would rather see improvement to debugging processes. Developers are looking for automated debugging tools, and better source code analysis tool for impact management and supporting cross team and cross sub-project projects.

When asked about preventative tools, developers had major concerns that the amount of false positives provided by the tool which may render the tool useless. Developers said "this would be a real challenge with the number of dependencies", "it depends on how good the results are in regards to false positives", and "I only want to know if it will break me", meaning that developers seem to care mostly about negative impacts of code changes as opposed to all impacts in order to reduce false positives and to keep preventative measures focused on real resulting issues as opposed to preventing potential issues. Overall, developers had little interest in preventative tools or processes.

In terms of catching indirect conflicts, developers suggested that proper software development processes are already in place to catch potential issues such as testing, code review, individual knowledge, or static language analysis tools. Further, developers said that having strict change type monitoring for indirect conflicts does not work due to the inherent complexities of indirect conflicts. This can be seen confirmed in that questionnaire participants had very few cases in which they always wanted to be alerted to a change type as seen in Table 3.4. Only method signature changes caused 68% of questionnaire participants to want to be always notified as they have a high chance to break the code. This is opposed to other change types listed in Table 3.4 which have 27% or lower of developers always wanting to be notified. This again showcases the complexity of indirect conflicts through change types which may or may not negatively affect a project.

When asked about curative tools, developers could only suggest that resolution times be decreased by different means. Interview and questionnaire participants suggested the following improvements to curative tools:

Code change type	Never	Occasionally	Most Times	Always	I Don't Care
Method signature change	5%	8%	12%	68%	7%
Pre-condition change	5%	27%	37%	23%	7%
Main algorithm change	11%	45%	19%	15%	11%
User interface change	12%	32%	20%	27%	9%
Run time change	13%	29%	25%	20%	12%
Post-condition change	7%	28%	32%	23%	11%

Table 3.4: Questionnaire results about source code changes that developers deem notification worthy, in terms of percentage of questionnaire participants.

- Aid in debugging by finding which recent code changes are breaking a particular area of code or a test.
- Automatically write new tests to compensate for changes.
- IDE plug-ins to show how current changes will affect other components of the current project.
- Analysis of library releases to show how upgrading to a new release will affect your project.
- Built in language features to either the source code architecture (i.e. Eiffel or Java Modeling Language [3]) or the compile time tools to display warning messages towards potential issues.
- A code review time tool which allows deeper analysis of a new patch to the project allowing the reviewer to see potential indirect conflicts before actually merging the code in.
- A tool which is non-obtrusive and integrates into their preexisting development styles without them having to take extra steps.

[3]http://www.eecs.ucf.edu/ leavens/JML//index.shtml

3.1.4 Evaluation

From the re-interviewed participants, I found extremely positive feedback regarding both my results and major discussion points. Participants often had new stories and experiences to share once they had heard the results of this study which confirmed the findings and often were quite shocked to hear the results as they did not usually think about their actions as such but then realized the results held true to their daily actions for better or worse.

As per grounded theory research, Corbin and Strauss list ten criteria to evaluate quality and credibility [13]. I have chosen three of these criteria and explain how I fulfill them.

Fit. "Do the findings fit/resonate with the professional for whom the research was intended and the participants?" This criterion is used to verify the correctness of my findings and to ensure they resonate and fit with participant opinion. It is also required that the results are generalizable to all participants but not so much as to dilute meaning. To ensure fit, during interviews after participants gave their own opinions on a topic, I presented them with previous participant opinions and asked them to comment on and potentially agree with what the majority has been on the topic. Often the developers own opinions already matched those of the majority before them and did not necessarily have to directly verify it themselves.

As added insurance, I conducted 5 post results interviews with developers to once again confirm my results, and discussions. These procedures can be seen in Section 3.1.2.

To ensure the correctness of the results, I also linked all findings in Section 3.1.3 to either a majority of agreeing responses on a topic or to a large amount of direct quotes presented by participants.

Applicability or Usefulness. "Do the findings offer new insights? Can they be used to develop policy or change practice?" Although my results may not be entirely novel or even surprising, the combination of said results allowed me to discover a the disjoint between theoretical awareness and practical awareness regarding indirect conflict tools as well as provide more insight into the debate of prevention versus cure in software development (as to be seen in Chapter 4). Given how many indirect conflict tools are left with the same common issues, I believe that these findings will help researchers focus on what developers want and need moving into the future more than has been possible in the past. These finding set a course of action for where effort should be spent in research to better benefit industry.

10 of the 78 questionnaire participants sent direct responses to me asking for any results of the research to be sent directly to them in order to improve their indirect conflict work flows. 7 of the 19 participants interviewed expressed interest concerning any possible tools

or plans for tools inspired by this research as well. The combination of research relatability and direct industry interest in my results help us fulfill this criterion.

Variation. "Has variation been built into the findings?" Variation shows that an event is complex and that any findings made accurately demonstrate that complexity. Since interviewed participants came from such a diverse set of organizations, software language knowledge, and experience the variation naturally reflected the complexity. Often in interviews and questionnaires, participants expressed unique situations that did not fully meet my generalized findings or on going theories. In these cases, I worked in the specific cases which were presented as boundary cases and can be seen in Section 3.1.3 as some unique findings or highly specialized cases. These cases add to the variation to show how the complexity of the situation also resides in a significant number of unique boundary situations as well as the complexity in the generalized theories and findings.

3.1.5 Threats to Validity

Because of the exploratory nature of this work, I chose Grounded Theory as my research method, which has some implications regarding the limitations and threats to validity of this study. Inter rater reliability was not used due to a time constraint which means that the code from open and axial coding have not been confirmed by a separate source. While I achieved high saturation regarding the topics focused on in this study, other populations studied from outside of this studies participants may add new insights into the pool of findings. As a result of this, findings from this study may only relate to developers or managers from the projects I studied. The saturation indicated here regards the majority results reached in this study. Since only 19 developers were interviewed, saturation was reached when majority agreed upon trends emerged in the results, thus further interviews did not have to be conducted. This is different from saturation of edge or unique cases often found in other qualitative research where the goal is to find all cases of a phenomenon where this study was only looking for majority cases.

I attempted to cover as great a breadth as possible of software developers in their practices, companies, project, languages, etc. However, it would be extremely difficult to conduct a completely thorough study of indirect conflicts in practice as there exist too many demographic options for participants. This same applies for questionnaire participants. This being said, questionnaire or interview results could become heavily biased towards a developer's good or bad experiences with any given project.

To counter this, my confirmatory interviews found high agreement for several of my findings. Therefore, I believe that I have uncovered valuable insight regarding indirect conflicts in practice.

3.1.6 Conclusions of Study

In this study, I have explored indirect conflicts by examining their root causes, their current mitigation strategies, and how developers wish to handle indirect conflicts in the present in future. To achieve these results, I interviewed 19 developers from across 12 commercial and open source projects, followed by a confirmatory questionnaire of 78 developers, and 5 confirmatory interviews.

In addresses the three issues of scalability, false positives, and information overload, my findings indicate: indirect conflicts occur frequently and are likely caused by software contract changes; while design by contract, add and deprecate, and personal experience help prevent indirect conflicts, developers tend to prefer to use detection and resolution processes or tools over those of prevention, and developers want indirect conflict tools to focus on automatic debugging and better source code analysis.

My result analysis has indicated that: developers do not want awareness mechanisms which provide non actionable results; developers would rather focus on curing existing problems rather than preventing potential issues; and there exists a gap in software evolution analytical tools between what is available and what practitioners need.

3.2 Study 4: Investigating Indirect Conflict Contextual Patterns

Release points are a vital milestone of software projects. From major releases of a Waterfall based project to iterations of an Agile development, releases form an interesting single point of a project's development history. Third party users (outside developers) of a system often only see a product at a release point either major or minor, and expect the system to come with a sense of reliability and stability at this point. Developers often expect to be able to upgrade a library to a newer version without having to make major revisions to their own projects to accommodate the upgrade (unless of course patch notes detailing major changes are released). However, major and minor releases of a library or software resource can cause software quality to degrade in a third party application as indirect conflicts may occur. The knowledge as to when a project is ready for public usage as to its reliability, quality, stability, and thus indirect conflict proneness can be a difficult decision to make for most project owners or maintainers.

While measuring software quality has had a major focus in software engineering research for many years [7] [25] [32], the study of software stability and its implications on reliability and indirect conflict proneness remains a difficult subject to understand. The decision of what makes a project stable and ready for a release often comes down to the release manager or maintainer of a project and is often a reflection of the open source community which surrounds the project [12]. Code churn is an often looked to statistically for stability but can be grossly misleading in terms of pre-release and post-release defects [22], with some exceptions [42] as well as proneness to indirect conflicts both internally and externally to third party applications. Creating an approach to determining software stability, release preparedness, and the proneness of indirect conflicts is still a large open area of interest in software engineering research.

In this study, I examine the notions of software change trends, specifically those trends around major releases. Change trends are trends which indicate a likelihood for a change type to occur around a certain event. Change trends have been used to detect stability in core architecture [56] as well as evolving dependencies [8]. This study was conducted in order to address the results of Study 3 which identified that indirect conflicts are likely to occur more before a major release (i.e. stable point in the code). I look to find the context in which trends occur in order to support indirect conflict tools in their context for when indirect conflicts are more likely to occur. With the power of major release points in open source projects as a starting point for project stability and the understanding that change trends can be leveraged to detect stability and the proneness of indirect conflicts, this study investigates the question:

RQ *What trends exist in source code changes surrounding major releases of open source projects as a notion towards a project stability measure?*

In this study, I perform a case study of 10 open source projects in order to study their source code change trends surrounding major release points throughout their history. I studied 26 change trends quantitatively and 4 change trends qualitatively, and identified a core group of 9 change trends which occur prominently at major release points of the projects studied. These change trends can provide context as to when indirect conflicts are more likely to occur based on the findings from Study 3 as I found that indirect conflicts tend to be become less frequent near major release and more frequent after a release or at the start of a new development cycle. The findings of this study can be applied over the lifetime of a project to determine the proneness of indirect conflicts and thus aiding developers in dealing with indirect conflicts in their projects.

3.2.1 Related Work

While very little has been published about release quality studies and stability (especially in regards to indirect conflicts), there have been a few studies which attempt to address the issues directly or indirectly. Wu et al. [58] performed a case study of SoftPM, a widely adopted project management tool, to explore the relationships of pre-release and post-release failures at major releases. Wu et al. found that the ratio of post-release failures to pre-release failures is significantly low and can be used to show reliability and stability. Hindle et al. [29] performed a case study on MySQL which observed a project's behavior around major and minor release by monitoring artifact check-ins and changes. They found that there are temporary stoppages for source revisions around releases, indicating that a temporary freeze is taking place for developers and that last minute fixes and manual testing may be performed. Zaidman et al. [60], in comparison, studied the co-evolution of production and test code with inspections and analysis at major and minor releases, showing how test and production code can evolve at different rates and times. These results show that production and test code should be handled as different cases for a stability measure around major releases.

The study of open source projects revolving around release points has become more accessible by the work of Tsay et al [54]. Tsay et al. created a resource of historical release dates for open source software projects to be used for future studies by other researchers.

In terms of software stability, development techniques have been proposed to increase software stability. Fayad [21] [20] suggests that "business objects" (BOs) do not change in nature and that they are inherently stable. These objects only need to change to accommodate external modules at the interface. Some studies such as Chow et al. [11] have investigated the stability of changes to interfaces which are considered a good indication of stability. Mockus et al. [41] used major and minor release points to compare industry process quality to customer-perceived quality of the software project. Mockus et al. found that defect density is lowest at major releases but at the same time software quality is at its lowest all when compared to minor releases. The low software quality here relates to end-user errors of installations and configurations. Wermelinger et al. [56] showed that stable core architectures can be detected by using source code changes. Finally, Fayad et al. [19] have investigated the Software Stability Model (SSM) for Software Product Lines to show that the SSM's impact on architecture and design of a software product can help improve the life of the product line and make it more adaptable and applicable.

3.2.2 Methodology

In order to answer my research question, I decided to use the tool ChangeDistiller created by Fluri et al. [24]. This tool allows me to detect fine grained source code changes in Java projects. This tool works by building an abstract syntax tree of a file before and after a code change, then it tries to determine the smallest possible edit distance between the trees. This results in the source code change at a fine grained level performed in the commit.

I conducted a case study of 10 open source Java projects. These projects are: eclipse.jdt.core, eclipse.jdt.ui, eclipse.jetty.project, eclipse.mylyn.commons, eclipse.mylyn.context, hibernate-orm, hibernate-ogm, hibernate-search, eclipse.maven.core, and eclipse.maven.surefire. These project were chosen because ChangeDistiller only works for Java source code and because of their high use amongst other Java projects and to study specific ecosystems of projects and their evolution trends.

For each of the projects, I obtained the software configuration management (SCM) system which is used to store all source code changes of a project. When it was necessary, I converted some forms of SCM system to Git in order to reduce implementation burdens of using multiple SCMs. Once the SCM was obtained, I used ChangeDistiller and iterated over every commit found in a project's git master branch. I stored 34 of ChangeDistiller's built in source code change types for each commit. I noted how many of each change type was performed in each commit and stored that information in a PostgreSQL database. In order to filter and protect my results, I manually inspected the 10 Java projects studied in order to identify code built for test purposes. I separated changes to this test code from all other code to ensure my results only focused on real implementation while allowing us to study changes to test based code separately.

Once the ChangeDistiller information was collected, I decided to examine software change trends surrounding releases of the projects I had selected. Since releases have preconceived notions of software stability and a lack of proneness to indirect conflicts, I decided that by studying the change types surrounding these releases, I could get a better understanding of what types of source code changes or trends constitute software stability or maturity. In order to study the release points, I went to each of my 10 project's web pages and looked through their release histories for major, minor, alpha, beta, and release candidate type releases. In total I identified 472 releases across my 10 studied projects.

Once the release dates were collected, I set about analyzing my data by creating average change ratios surrounding the release dates of each project as a way to measure the trend of a particular change type at a release type. This change ratio simply compares the number of change events (of a given change type) before a release to after the release. Both of the before and after event totals are divided by the number of commits on their respective side of the release to account for activity. I implemented this algorithm through Equation 3.1.

Equation 3.1 works to create a change ratio by first creating a numerator by summing across all releases of a given release type a sum of a particular change type in commits (T_c) from the release date (r) to a given number of days after the release (d) divided by the number of commits in this date range ($|c|$). Next the denominator is created by summing across all release of a given release type a sum of that same particular change type in commits (T_c) from a given number of days (d) before the release date (r) to the release date divided by the number of commits in this date range ($|c|$). This numerator and denominator form the final change ratio. This equation gives us a ratio of a particular change type happening before and after a particular release. If the ratio is above 1 then that particular change type occurs more frequently after the release and if it is below 1 then it occurs more frequently before the release. For the purposes of my study, I set the number of days before and after the release (d) to 60 as the projects studies had many months in between their major releases. This quantitative data formed much of the basis for the results to come in Section 3.2.3

$$\text{ChangeRatio} = \frac{\sum_{r_0}^{r_n} \sum_{c=r}^{r+d} T_c/|c|}{\sum_{r_0} \sum_{c=r}^{r-d} T_c/|c|} \tag{3.1}$$

Aside from generating quantitative data, I also created a web application for the visualization of the data called API Evolution (APIE). This visualizer allowed me to inspect a single project and a single change type metric at a time (see Figure 3.1) for qualitative analysis of software evolution trends. I used this tool to manually inspect 4 specific change type trends surround release dates. To do this, I aggregated change types across 50 commits, meaning that each point in the graph represented the date of a commit and the sum of the particular change type's occurrences over the last 50 commits. This was used to smooth out the curves presented by the tool to allow easier manual inspection. This method however does not take activity into account as seen in Equation 3.1, so it represents the true activity and change types occurring. Manual inspections were labeled into 4 categories: upward trending, local maximum, downward trending, and local minimum. Since the graphs were quite turbulent, best estimations were conducted by two judges at each

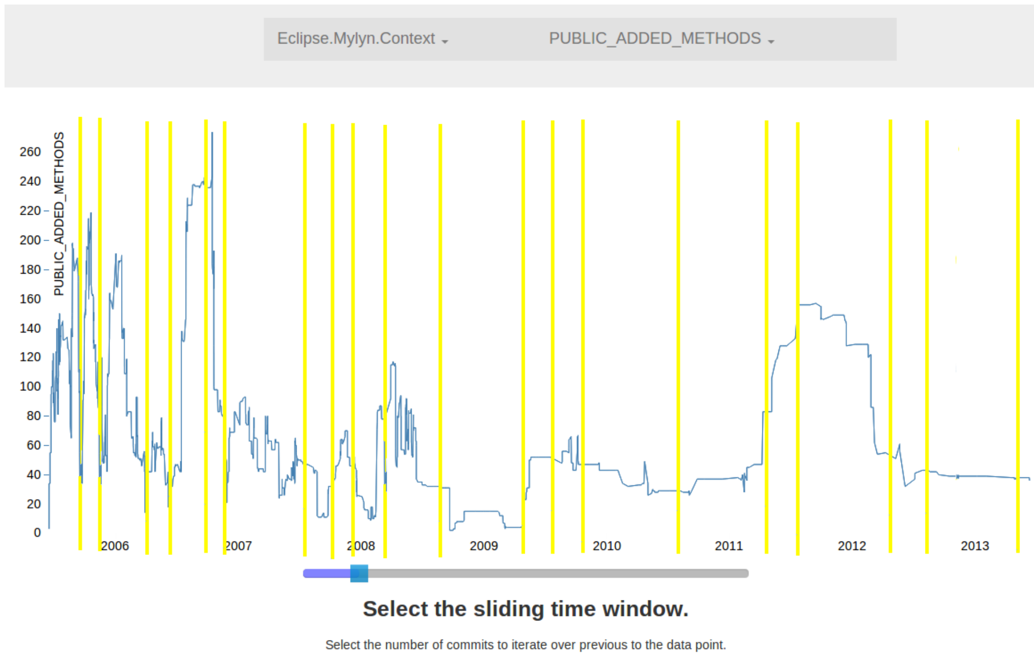

Figure 3.1: A screen shot of the APIE visualizer showing project Eclipse.Mylyn.Context with change type PUBLIC_ADDED_METHODS being analyzed and showing major releases as vertical yellow lines.

release point to fit the graph into the aforementioned 4 slope categories. The two judges used 1.5 months before and after the release date as start and end points for the graph trend line.

I performed 1888 manual inspections across 10 projects, 472 release dates and 4 change types, and used this data to form the basis of my qualitative data. Quantitative data was used to compliment the quantitative ratios found from the previous methodology.

3.2.3 Results

Due to time requirements, I focus my results on major releases of the 10 case study projects and select few of the calculated change ratios. There were 109 major releases across the 10 studied projects. All of the major findings as per values computed from Equation 3.1 for non test metrics can be seen in Table 3.5.

To study the most prevalent change trends, I set a ratio threshold of greater than 1.2, or less than 0.83 (20% greater trend of after the release date or 20% greater trend of before the release date) to indicate the greatest trends.

I found 9 major change trends which surround major releases in the open source projects

Object	Added	Changed	Removed
Public Classes	1.14	0.86	1.16
Public Methods (Signature)	1.07	0.92	1.34
Public Methods (Bodies)	-	1.06	-
Private Classes	0.81	1.18	1.44
Private Methods (Signatures)	1.00	1.10	1.22
Private Methods (Bodies)	-	1.08	-
Files	1.12	0.96	1.14
Documentation	-	0.99	-

Table 3.5: Implementation oriented change types and their normalized average change ratios at 60 days on each side of releases.

studied. 4 change trends found that occur before major releases are: added private classes, changed test method signatures, changed documentation, and removed test classes. 5 change trends found that occur after major release are: added test methods, changed test classes, removed public methods, removed private classes, and removed private methods.

As it can be seen in Table 3.5, there are few change type trends around major releases which pass my threshold. We can see that both public and private methods being removed from a project is more likely to occur after a major release than before. Table 3.5 also shows significance in the changes to private classes. We see that private classes are added more (24%) before a major release and removed more after (44%) the release. All results in Table 3.5 could be used as identified trends of major software releases, while I have just highlighted the larger ratios which meet my threshold criteria.

Another interesting trend that can been seen in Table 3.5 is that of changes to public objects. We can see for public classes and methods that 5 out of 7 ratios indicate changes occur to these objects after major release rather than before. I hypothesize that these changes to the public API after a major release come from newly reported bugs from end users as well as having old features being deprecated while adding new features to the project after the stable build had been released.

My complementary qualitative results from manual graph inspections can be seen in Table 3.6. These results show that adding, changing signatures and bodies of, and removing public methods tend to all be at a local minimum of change type trends at major releases when activity is not taken into account. These results confirm previous results of low code churn as an indication of stability.

Lastly, I found that software changes related to testing can be an indicator of a major

Change Type	Upward Trend	Local Maximum	Downward Trend	Local Minimum
Added Public Methods	21.6%	17.2%	14.7%	33.6%
Changed Public Methods (Signature)	6.0%	19.8%	19.0%	39.7%
Changed Public Methods (Bodies)	9.2%	16.5%	26.6%	37.6%
Removed Public Methods	7.8%	16.4%	12.9%	41.4%

Table 3.6: Qualitative graph analysis results.

release points within the projects studied. The change ratios found can be seen in Table 3.7. As it can be seen, the four ratios which meet my threshold and are indicators of stability with regards to test based changes are: the changing of test classes, the removal of test classes, the adding of methods, and the changing of method signatures, and test classes being changed. Changes to documentation also meets my threshold and occurs more before a major release.

Object	Added	Changed	Removed
Classes	1.07	1.21	0.76
Methods (Signatures)	1.23	0.83	1.01
Methods (Bodies)	-	0.90	-
Documentation	-	0.72	-

Table 3.7: Test oriented change types and their normalized average change ratios at 60 days on each side of releases.

While all change ratios may need to be considered for continued analysis or a taxonomy of change trends, I have offered the strongest change trends in these results as a suggestion for future focus.

3.2.4 Conclusions of Study

In this study, I have conducted a case study of 10 open source Java software projects in order to study their change trends surrounding major releases as previous studies have shown that indirect conflicts occur less at a major release and more so after a major release or at the beginning of a development cycle. I have presented here 9 major change trends which surround major releases in the open source projects studied. The 4 change trends found in this study occurring before major releases are: added private classes, changed test method signatures, changed documentation, and removed test classes. The 5 change trends found in this study occurring after major release are: added test methods, changed test classes, removed public methods, removed private classes, and removed private methods.

These 9 change trends can be used in future indirect conflict tools in order to identify a context for indirect conflicts. For example. Any of the 9 change trends which occur more so after a major release may be used as a sign of high indirect conflict proneness since after a major release a new development cycle is likely to begin. As per change trends which occur more so before a major release, indirect conflict tools may use this context in order to identify a low proneness to indirect conflicts in their results. These two contextual patterns can be used throughout the life of a software project in order to help better inform indirect conflict tools as to the processes of developers and provide better feedback to said developers of indirect conflicts.

Aside from contextual patterns for indirect conflicts, this study has also shown the beginnings of a visualization for source code change trends which may be used as a visual cue towards project stability and potential areas of instability where action may need to be taken.

Chapter 4

Discussion

While each study presented in this dissertation is quite unique, this chapter will focus on the underlying themes and results of all 4 studies. This chapter will address the usefulness of the results presented towards the research community and what those results mean for continued studies in future research as well as discuss how the results can be viewed for larger over arching outcomes than those presented in results sections.

This chapter will proceed in 2 subsections. The first section will address the 2 motivational studies and the lessons learned from each as background information for the richer following studies. The second section will address the 2 large studies found in Chapter 3 in a combined manner. Having reported the trends in industry (what, when, how, mitigation, and resolution) from Study 3, the Study 3 discussion will focus these results in relation to the outstanding issues of tool based solutions regarding indirect conflicts which are information overload, false positives, and scalability. Since Study 4 was directly associated with developer opinion found in Study 3, the two discussions will be intertwined to better support each other.

4.1 Motivating Studies Discussion

The Human Factor of Indirect Conflicts *An important component of indirect conflicts are the developers themselves and how their notions of other's work is perceived across a project.*

As we have seen in Study 1, developers that are tied to source code objects can become a focal point of indirect conflicts through the life of a project. For instance we can see from Table 2.1 that developers Daniel and Anne on Hibernate-ORM are always linked indirect conflicts when dependencies between their source code objects change. This is an important observation to make when moving forward with indirect conflicts.

What is really happening between these two developers is the real issue to consider. Daniel could have an assumption about the way Anne's code works which causes Anne's changes to have negative impacts on Daniel's own work. For instance, Daniel may expect a method of Anne's to have a special return case, which may be correct, however when Anne changes that special case or removes it, Daniel's code can be negatively affected. In this extreme result found in Study 1, further analysis showed that one central method in the software project was being changed frequently and causing Daniel's code to be negatively affected in some way.

The question of how to prevent this type of negative impact is ultimately the goal of indirect conflicts. Study 1 has shown that pairs of developers can often be a large component to that goal as well. (Impact from Study 2 was created to use these pairs of developers in addressing the issue of indirect conflicts.)

Information Overload *My tool Impact, as well as many other indirect conflict tools, suffer from information overload in their delivery to developers which is a key issue in preventing adoption and acceptance of indirect conflict tools.*

As was previously stated, many indirect conflict tools end up suffering from information overload to developers and other end users. Impact was initially created to take the insights from Study 1 and attempt to create a new indirect conflict system based on developer interactions inside the code base which would potentially address information overload. However, we now know from the results of Study 2, that Impact once again suffered defeat to information overload based on the case study evaluation. Ultimately, this sense of information overload ends up causing adoption of indirect conflict tools to fail which in turn causes some research components to have failed as well.

From previous works, as well as from Study 2, we know that information overload is a large issue in indirect conflicts. It has been found that a large number of dependencies in software caused by the nature large software projects is a root issue in information overload [47, 50]. These large sizes in dependencies are ultimately what cause information overload as dependencies cannot be evaluated as to their relevance in a certain source code change with ease. In other words, we cannot determine which of the numerous dependencies will fail (outside of compilation and testing failures) when source code is changed.

A potential solution derived from the evaluation of Impact comes from the ideas of Meyer [40] on "design by contract". In this methodology, changes to method preconditions and postconditions are considered to be the most harmful. This includes adding conditions that must be met by both input and output parameters such as limitations to input and expected output. To achieve this level of source code analysis, the ideas of Fluri et al. [24] can be used on the previously generated ASTs for high granularity source code change extraction when determining if preconditions or postconditions have changed. While this solution does not wholly address the problems of information overload as previously stated through failures in dependencies, it does reduce the number of source code changes to be analyzed which in turn will reduce the amount of information on the whole which is put forth by the indirect conflict system. This solution however does also run the risk of missing more indirect conflicts as many conflicts can occur outside of changes to method contracts. This solution represents my first ideas of fixing information overload in regards to indirect conflicts and is again found in the later discussion of root causes of indirect conflicts found in the next section.

4.2 Indirect Conflict Exploration Discussion

While the previous section discussed the main motivations for taking a step back on indirect conflicts in order to understand better what can be done to improve developer work flow, this section will discuss exactly those steps back. From Studies 3 and 4 we will notice 3 main trends that have been discovered for indirect conflicts in both industry and in research. These 3 major trends are: unwanted awareness, prevention versus cure, and the gap in software evolution analysis. This section will also include a brief discussion regarding the implications, learned from studies 3 and 4, for both research and for industry through tools.

Unwanted Awareness *Developers tend to only care about the awareness of other's activities, if it causes a negative influence on their own work. Developers only want to hear about another developer's actions if it forces them to take some action because of it. This limited awareness is quite different from what literature suggests, which is larger awareness about most, if not all actions, and it also suggests why developers see a high amount of what they believe to be false positives, as the changes being reported are not causing them to take action. These reported non-actionable responses lead to information overload and false positives.*

As we have seen, indirect conflicts are found to be quite a serious problem that occur frequently, sporadically, and differ greatly from case to case. These conditions pose large issues for the creation of generalizable theories or tools in regards to indirect conflicts. These underlying complexities are the probable cause of disinterest of software developers to proposed or implemented tools as discussed in Section 3.1.1. This inability to generalize is what I believe to be the leading cause of information overload and false positives in the awareness system, causing developers to eventually ignore information being presented to them, rendering the system useless. These false positives are caused by a difference in what developers consider to be false positives versus what awareness literature considers they are. This disjoint, as will be seen, is caused by developers only considering events which cause some action to be taken on their part, to be true positives, where as current awareness understanding would state any event which is related to an individual's work [28] [9], to be true positives. "You need a good understanding of what the code change is or else you will have a lot of false positives" said one developer, showing that not all changes around an object should be reported for awareness.

Developers have been found to have a great understanding of what they need to know about and more importantly what they don't in their project awareness. To be able to fully understand a developer's awareness intuition, we can see from the results of this study that developers only want awareness of an event if the event forces the developer to take some action. In a sense, developers don't care about what they don't need to account for. "We would want a high probability of the [reported] change being an issue", meaning that the developer only wants the awareness if the item will require action on their part to resolve the issue. This sense of unwanted or limited awareness is crucial to understand why generalized awareness techniques of difficult to generalize problems, such as indirect conflicts through generalized changes to classes [47], or methods [50, 53], often encounter difficulties of false positives and information overload. Developers simply do not want to know about events which effect them, but require no action on their part.

This unwanted awareness, or limited awareness, seems counter intuitive to current awareness understanding which would state that being aware of all events in ones surrounding leads to higher productivity or other quality aspects. In theory this is correct, but as it was found in practice, this is an incorrect assumption. In regards to this full awareness, one developer said "There is no room for this in [our company], as tools are already in place for analysis of change[s] and code review takes care of the rest". Since software developers have limits on their time, awareness of all events surrounding a developer's work or project is not possible. *Developers prefer to spend their limited time dealing with the awareness of events which cause them to take some action (changing code, communication, etc) rather than simply being aware of events occurring around their work which pose no direct threat to the consumption of their time.*

Of course, whether or not this unwanted awareness is the correct path for developers to take is an open question to consider. When developers encounter a problem which could have been solved by having greater awareness of events which did not directly affect them initially, we must consider the positive and negative influences of adding this, for now, unwanted awareness. A positive influence of total, or near total awareness of events at the developer level, would be the full understanding of a developer's work and environment which comes with higher quality or understanding of the product. A negative influence would be that valuable developer time is spent understanding events which may not directly apply to themselves as opposed to producing more output of their own work. This balance between awareness and productivity is found to be a fine line in practice, however, when given the choice, developers tend to, as previously stated, lean towards less awareness in order to, in their eyes, be more productive.

Prevention versus Cure *Developers would rather spend their time on curative measures (fixing problems after they arise) rather than preventative measures (preventing problems from happening). Developers see the task of fixing real problems to be less of a burden than preventing potential problems because of available tools and their perceived notions of time management. Focusing on cure is a direct response to the issue of scalability found in indirect conflict tools. Trying to prevent all conflicts is too large a problem to handle, while curing existing conflicts is manageable by a developer.*

We have seen through the results, that developers posses both tools and practices for the prevention, detection, and resolution of indirect conflicts. We have also seen that through unwanted awareness, and the use of developer time, that developers tend to prefer working on real issues that have already occurred as opposed to preventing issues of the future which may never arise. This is neatly explained by the popular adage "I work until something breaks" taken by most developers. This mindset is a clear example of developers taking a curative approach to software development opposed to a preventative approach. Prevention here refers to taking precautionary steps to stop issues, indirect conflicts, form occurring in the first place while cure refers to fixing issues as they arise which includes not attempting, or putting little attempt, into preventing them in the first place.

Two out of the three identified prevention methods taken by developers are simple blanket risk mitigation strategies accomplished essentially by not changing code (design by contract, and add and deprecate), while the third is simply developer experience and knowledge. Clearly, developers are spending little to no time in prevention. Developers do however spend a large amount of time in detection and cure through the writing of tests and the debugging of issues. In fact, most improvements mentioned by developers in regard to dealing with indirect conflicts occurred at either the detection or cure levels. Obviously, developers either prefer to spend their time in curative measures, or do not posses the proper tools to take better action in the preventative stages. "You're reaction time is much more crucial" said one developer in that resolution tools are believed to be or larger importance as once issues have occurred, it doesn't matter what prevention was taken, the issue must simply be resolves as quick as possible now.

The lack of prevention process and tools being used is believed to be due to 2 factors. The first being the identification of dependencies. Even with an experienced system architect, identifying dependencies and notifying those involved is a daunting task which is ignored more than dealt with. The second being the knowledge of when a dependency will fail, also requires vast knowledge of the product, more than anyone may have. This is compounded by the unique and sporadic nature of indirect conflicts. These 2 factors are what ultimately have led to the amount of false positives and information overload seen in previous tools. The abundance of detection and curative process and tools on the other hand shows once again developer's willingness to debug real issues, that maybe even could have been prevented, rather than prevent the issues in the first place. With this lack of prevention and abundance of curative measures, the question ultimately presented in this area is if curative measures are more productive than preventative measures.

Dromey [17] has raised the debate of prevention versus cure in software development and how difficult a problem it is to measure. The pros and cons of prevention versus cure I have identified, are similar to those of unwanted awareness, in that they result in a trade off of where time is spent and how productive each side of the argument truly is. If prevention can be shown to be more effective in that it reduces the number of indirect conflicts or time spent debugging them compared to the time taken to prevent them, should we then not be moving developers into a more preventative mindset; if curative can be shown to be more effective in that it takes less time to fix real issues compare to preventing potential ones, should we not be putting more time into automatic debugging, such as Zeller's Delta Debugging [61] or program slicing techniques [55], or automatic test case creation.

A last interesting observation about current curative measures being taken in industry, is that developers view testing of software as curative, when one could easily make the argument that it is preventative in that most tests are written to pass originally and are kept in place to ensure future changes do not cause issues. This mindset may come from the notion that writing tests is originally thought of as part of normal code writing, meaning that developers see the extra task of test coverage as part of feature implementation. However, if a test never fails, could it not be said that it is preventing changes from causing it to fail rather than detecting when failures or indirect conflicts do occur? This may suggest that if, given the write tools, developers may no longer view preventative measures as a "burden" and may be more inclined to take a more balanced preventative approach rather than mostly curative, as "prevention is the goal" was commonly said by developers.

This prevention versus cure discussion resides purely on the developer level and should be noted that it may not apply to system designers, architects, or managerial stakeholders. From the project managers interviewed it was shown that they heavily favored planning and prevention (even though their prevention may be on more abstract levels than developers actually need) while leaving the curative approaches to their developers. The prevention versus cure debate may have different outcomes depending on what level of abstraction is being viewed in research.

Gap in Software Evolution Analysis *Due to a lack of productive software analysis tools, caused by project infrastructures and analysis unfriendly languages, contextual indications of indirect conflicts are often missed. This lack of analysis also causes some software project configurations and software languages to be quite prone to indirect conflicts. This is yet another issue related to scalability of existing indirect conflict tools.*

Indirect conflicts are more likely to occur, from developer opinion, before a mature point is reached in the project's evolution. "Once you get the API stable, people are better at communicating changes in regards to dependency concerns."(This mature point may stem from a major release, end of an iteration, a new feature being released or any point considered stable and reliable.) However, this context of a mature point, as well as any other potential contextual attributes, are rarely identified or accounted for in regards to indirect conflicts. Developers have said that different change types may occur at different rates throughout a project's life time and that this may drastically effect the outcomes of indirect conflict tools or processes.

A deeper understanding of what context indirect conflicts occur in seems to be more of a success factor to indirect conflict research than may have been previously thought. Static analysis may be useful in regards to indirect conflict context in that we could identify trends surrounding the mature points of previous projects in order to give a better understanding of what it means for a project to be pre or post mature point, which could affect the outcome of indirect conflict understanding. These change trends are exactly what Study 4 set out to address.

Through Study 4, I showed that 9 critical change trends exist at a major release of a variety of open source projects. This knowledge could help identify, as was previously stated, when indirect conflicts are more likely to occur in a project or not. For example, the trend of adding test methods was found to occur more after a major release than before. This trend along with the knowledge that indirect conflicts occur more at the start of a new development cycle can be used to predict times in development where indirect conflicts are likely to occur automatically. An automated system may see that more test methods are being added than before in the past 4 weeks or so and be able to alert developers to an instability in the code which in turn means an elevated risk of indirect conflicts.

Similarly, the trend of adding private classes is seen to occur more so before a major release than after. This knowledge again can be used in an automated system along with the fact that indirect conflicts are less likely to occur towards the end of a development cycle. The automated system will be able to identify a stability in the project through private classes being added and alert developers that the code is stabilizing and that large changes to the project should be avoided until the start of the next development cycle in order to further reduce the risk of indirect conflicts.

This added level of context, through the notion of a mature point, would add to the prevention versus cure debate as previously discussed as well. Prevention may be a better choice once a project has reached a mature point as the code base becomes more stable and source code changes become more dangerous to the quality of the project. Curative measures may be a better choice before a mature point as code churn can be higher which causes more bugs than can be prevented. These possibilities existing, it may be imperative to discover more of project stability and the mature point in order to fully understand the nature of indirect conflicts and their context.

While this added sense of understanding is important to indirect conflicts, contextual attributes of a project's current progress or measures of performance are often more synonymous with project management rather than software developers themselves. An understanding of a project's evolution, a mature point being reached as an example, may pose as a more useful tool for project management rather than developers. This more abstract tendency of indirect conflict occurrences may add even more power to project management for evaluation of progress, code stability, and code reviews.

In regards to these contextual identifications in software projects, dependency identification and tracking is a key missing component of indirect conflict analysis due to the weaknesses of static analysis. The gap in this identification comes from software and organizational structures of software teams. Between increased modularization (multiple sub projects or repositories), cross language dependencies, and languages which do not lend themselves to static analysis, static analysis tools have become quite limited in identifying and tracking dependencies where they were once strong. As software has become more sophisticated over time, static analysis tools have gone from extremely useful to only occasionally useful. Since many projects involve several languages, sub projects, and database schemas, static analysis has become cripplingly obsolete in the industry of today. In order to move indirect conflicts and many other research areas forward, this gap of cross domain static analysis must be filled.

As an example, relational database schemas are one of the highest sources of indirect conflicts found in their projects, “it breaks stuff all over the place”, yet we know from Maule et al. [39] that relational database schemas have been scarcely researched in terms of indirect conflicts. This falls in with my previous understanding of cross language dependency tracking in that database schemas are independent of languages which may be on the receiving end of their output.

These increases of complexity in software products has left quite a gap in dependency identification and tracking which has lead to some of the deficiencies of indirect conflict research.

4.2.1 Implication for Research

Prevention versus Cure *The largest implication for future research found in this dissertation is the need for continued study of the open question “Which of prevention or cure is more effective for software development and indirect conflicts?”. This simple question will undoubtedly produce extremely complex answers.*

With software developers focusing their current efforts on curative measures, for indirect conflicts, suggests that while it may not be the most effective, it may be the easiest road for developers to take. This may be the path of least resistance, but it may not be optimal. Software engineering as a whole should strive to answer this question or provide more insight into possibilities, as its answer may determine many future actions taken by the research community.

Recommendations towards the study of prevention versus cure involve the examination of formal processes and tools used by industry professionals with measurements of efficiency and effectiveness, similar to the work of Tiwana [52] in coordination tools. With these studies, we may find insight into the correct balance of prevention versus cure, thus being able to increase developer productivity as well as identify more gaps in theory versus practice which may lead to improved tools or the abandoning of existing ones as was shown with UML [45].

Awareness Theory to Model Real World *the need to further characterize the mismatch between awareness approaches and tools as developed in the research community, compared to awareness needs as perceived by industry. We should push our awareness* *theories to model real development environments where interruptions, time management, and full development life cycles are often large factors.*

As was identified in the previous discussion, while current understanding suggests that awareness of all events related to ones work produces a more coherent understanding of a person's environment, developers find this to be overly time consuming to the point where they only want to be aware of events which require action on their part. This difference, of what should be and what is, should be further understood to combat future potential failures in tool development or theories attempted to be used in practice.

4.2.2 Implication for Tools

Give Developers What They Want *Developers have their own notions of what they want, and how "good" that something is for their productivity. We should pay attention to what they want, in order to shape our models and better address their needs.*

The direct implication this research has on tool creation is that of current adoption among developers. As was stated, developers are more keen to invest their time at the detection and cure / resolution stages of indirect conflicts. That being said, focus at these two stages for tool development will lead to larger adoption among developers. It should be noted here that detection must come with an almost zero rate of false positives as current tools (unit and integration and user testing), while they may not have 100% recall, have almost 100% precision.

Stronger Source Code Analysis *With languages like JavaScript becoming popular, having projects with multiple languages, and often having a database schema, we should focus on better techniques for static analysis based on what industry standards are for languages and project structures.*

The more indirect implication of this research on tool creation is that of improving existing tools. While not all existing tools are used for indirect conflicts alone (automatic debugging [62], unit tests, etc), most of these tools have a need for rapid expansion, according to developers, for dealing with indirect conflicts. The ability to have unit tests automatically written for a given software object's contract, the ability to find a change in an external project which has broken a developer's own project, or any automation of the existing detection and resolution stages of indirect conflicts are what developers currently seek. But of course, most of these implications rely on the improvement of static analysis tools.

These tool implications themselves imply the need of further development of static analysis tools. Static analysis lays at the heart of most if not all stages of indirect conflict research. We must be able to track and manage software dependencies across the new landscape of software development that is multiple projects, repositories, and cross language support. These improvements will allow the further development of both current and future indirect conflict tools.

Chapter 5

Conclusions

Awareness techniques have been proposed and studied to aid developer understanding, efficiency, and quality of software produced. Some of these techniques have focused on either *direct* or *indirect conflicts* in order to prevent, detect, or resolve these conflicts as they arise from a result of source code changes.

While the techniques and tools for direct conflicts have had large success, tools either proposed or studied for indirect conflicts have had common issues of information overload, false positives, scalability, information distribution and many others. These issues have ultimately led to poor developer interest in indirect conflict tools as well as a failed adoption rate and partial failures of research.

To better understand these issues, this dissertation has focused on exploring the world of indirect conflicts through 4 studies. The first two studies focused on motivational circumstances which occur during the software development life cycle and cause indirect conflicts. Developer's interactions were studied in order to create a tool which can aid in the workflows around indirect conflicts. The second two studies presented a deeper investigation as to why most indirect conflict tools fail to attract developer interest through exploring the root causes of indirect conflicts and how tools should be properly built to support developer work flows.

5.1 Study 1

Technical dependencies are often used to predict software failures in large software system [35, 46, 63]. However, human interactions as tied to software objects can also be used to predict software failures. Study 1 presented a method for detecting failure inducing pairs of developers inside of technical networks based on code changes. The methodology used was to assigned developers to the source code artifacts which they authored. Next, dependencies were identified between source code artifacts, implying dependencies between developers. Lastly, failure inducing dependencies between developers were identified through using methods to identify bugs in source code throughout the lifetime of a project. These developer pairs were used in the prediction of future bugs and to provide coordination recommendations for developers within a project. These bugs being predicted directly correlate to indirect conflicts. Since the human dependencies being measured in Study 1 are directly tied to software objects, we can say that the indirect conflicts being studied here are between developers themselves, while the dependencies between the developers are found in the source code.

Study 1 however, did not consider the technical dependencies themselves to be the root cause of the software failures. This study focused purely on developer ownership of software methods and the dependencies between developers as the possible root cause of the failures.

Through the analysis of Study 1 as seen in Chapter 4, an important component of indirect conflicts are the developers themselves and how their notions of other's work is perceived across a project. We can see from Study 1 results that a developer Daniel may believe a source code artifact behaves in a particular way (from documentation or other forms of communication for the artifact) so when that artifact changes its behavior, Daniel can become negatively impacted. A human factor is present here as the person who makes the artifact change, say Anne, may not have communicated the change to all other developers affected by the change. This is an important understanding that led directly into the creation of Impact (Study 2) as a way to mitigate the negative affects across human developers as a result of source code changes.

5.2 Study 2

As a direct response to the findings and analysis of Study 1, Study 2 set out to address the human factors presented through an indirect conflict tool. In Study 2, I proposed a generic design for the development of an awareness tool in regards to handling indirect conflicts through human factors. I presented a prototype awareness tool, *Impact*, which was designed around the generic technical object approach. Impact could track which developers were responsible for which artifacts of code as well as their dependencies to other developer's code artifacts. When a dependency to another developer's code was changed, the developer was notified of the change in order to avoid any bugs that may arise from the change. The delivery system used for Impact was that of an RSS type feed where developers could view their notifications in a stream of messages through a web application. However, *Impact* suffered from information overload, in that it had too many notifications sent to developers.

This failure from information overload was ultimately equivalent to the various other indirect conflict tools from previous research (even those not addressing human developer factors). This issue of information overload is the key issue in preventing adopting and acceptance of indirect conflict tools from developers and ultimately leads to some research component failures to understand what developers truly want from indirect conflict tools.As a proposed solution to information overload, the ideas of Meyer [40] on "design by contract" were presented. This methodology examines the post and pre conditions of software objects in order to reduce the number of source code changes that are analyzed by Impact in order to reduce information overload.

While the previous proposed solution was designed to fix information overload in Impact and potentially other indirect conflict tools, it was decided that further investigation into indirect conflicts was needed to truly attempt a solution. The results of Study 2, combined with the knowledge of previous research having similar issues of information overload in indirect conflict tools, sparked an interest in studying indirect conflicts at a deeper level (Study 3) which could be used to better understand causes and developer strategies in solving indirect conflicts.

5.3 Study 3

Indirect conflicts are a significant issue with real world development, however, many proposed techniques and tools to mitigate losses in this realm have been unsuccessful in attracting major support from developers (as was seen in Study 2). In Study 3, I conducted a qualitative study involving 19 interviewed developers from 12 organizations as well as 78 surveyed developers. I provided characterization of indirect conflicts, current developer work flow surrounding indirect conflicts, and what direction any future tools should follow in order to aid developers in their work flow with indirect conflicts.

For the root causes of indirect conflicts, I found that indirect conflicts occur due to changing of a software object's contract and the lack of understanding of the far reaching implications (through dependencies) of that change. I also found that indirect conflicts are more likely to occur at the beginning of a development cycle when the code is quite unstable and that these scenarios can become compounded in difficulty when more developers are present on a project.

As per current developer work-flows regarding indirect conflicts, I found that developers have 3 major mitigation strategies to avoid indirect conflicts: “Design by Contract”, add and deprecate, and personal experience. For catching indirect conflicts, I found that use case coverage through proper testing (both unit and integration) are currently thought to be the best developers can achieve. And finally, for the future of indirect conflict tools, developers made it clear that they would prefer an improvement to resolution methods for indirect conflicts such as automatic or aided development techniques.

As per the analysis of Study 3, I have shown the disjoint of current awareness understanding versus the practical awareness needs found in industry. This disjoint, caused by the difference of academic and practical understanding in awareness needs, ultimately lead to tools with information overload and false positive issues. The debate of prevention versus cure was presented along with the industrial tendency towards curative measures. While a curative approach may be favored by developers, further research is needed to fully assess the positive and negative influences of prevention versus cure for productivity and quality concerns. Finally, I have shown the gap of analytical evolution tools needed for dependency identification and indirect conflicts. This gap, unless addressed, may prevent future industrial adoption of tools produced by researchers for lack of fit in industrial needs. This gap directly resulted in the final study of this Dissertation to be completed (Study 4).

5.4 Study 4

As a result of the gap in software evolution analysis shown in Study 3, I conducted a case study of 10 open source Java software projects in order to study their change trends surrounding major releases as Study 3 had shown that indirect conflicts occur less at a major release and more so after a major release or at the beginning of a development cycle.

Through Study 4, I presented 9 major change trends which surround major releases in the open source projects studied. The 4 change trends found in this study occurring before major releases are: added private classes, changed test method signatures, changed documentation, and removed test classes. The 5 change trends found in this study occurring after major release are: added test methods, changed test classes, removed public methods, removed private classes, and removed private methods.

These 9 change trends can be used in indirect conflict tools in order to identify a context for indirect conflicts. For example, the trend of adding test methods was found to occur more after a major release than before. This trend along with the knowledge that indirect conflicts occur more at the start of a new development cycle can be used to predict times in development where indirect conflicts are likely to occur automatically. An automated system may see that more test methods are being added than before in the past 4 weeks or so and be able to alert developers to an instability in the code which in turn means an elevated risk of indirect conflicts.

The results of Study 4 directly supplemented those from Study 3 in that it addressed the gap of software evolution techniques needed for indirect conflicts. However, this study was also quite limited and the issues of cross domain analysis techniques presented in Study 3 are still cause for future research.

5.5 Final Conclusions

This dissertation has covered a wide range of interests all within indirect conflicts. I have shown how human factors can be an important part of indirect conflicts and how pairs of developers can be found to be statistically related to indirect conflicts bugs. I have shown how these human factors can be integrated into indirect conflict tools by using dependencies in authored source code artifacts. Ultimately however, these new human factors added into indirect conflict tools resulted in similar failures of information overload as seen through many previous research tools in indirect conflicts. After these conclusions, I presented a study which found root causes of indirect conflicts to be contract changes and an unawareness of those changes implications, that developers use “Design by Contract”, add and deprecate, and personal experience to avoid indirect conflicts, and some suggestions for future indirect conflict tool development while stating where gaps in source code analysis should be improved to improve the indirect conflict tools. Based on that gap in software evolution analysis in indirect conflicts, I presented a method for finding contextual patterns which can relate to indirect conflicts in order to aid future tool development for indirect conflicts.

Through the analysis in Chapter 4, I have shown the disjoint of current awareness understanding versus the practical awareness needs found in industry. This disjoint, caused by the difference of academic and practical understanding in awareness needs, ultimately led to tools with information overload and false positive issues. This was quite evident from my tool Impact as presented in Study 2. Even when the tool is based around the statistical results of Study 1, the disjoint between what academics determine developers want versus what they themselves want is quite obvious. The debate of prevention versus cure has been presented along with the industrial tendency towards curative measures. However, while a curative approach may be favored by developers, further research is needed to fully assess the positive and negative influences of prevention versus cure for productivity and quality concerns. Finally, I have shown the gap of analytical evolution tools needed for dependency identification and indirect conflicts. This gap, unless addressed, may prevent future industrial adoption of tools produced by researchers for lack of fit in industrial needs. While I have shown a potential solution to address a fraction of the analysis gap through contextual patterns of indirect conflicts, many other problems of the analysis gap remain problems for future research.

Bibliography

[1] Mithun Acharya and Brian Robinson. Practical change impact analysis based on static program slicing for industrial software systems. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 746–755, New York, NY, USA, 2011. ACM.

[2] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Sourcerer: An internet-scale software repository. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, SUITE '09, pages 1–4, Washington, DC, USA, 2009. IEEE Computer Society.

[3] Andrew Begel, Yit Phang Khoo, and Thomas Zimmermann. Codebook: discovering and exploiting relationships in software repositories. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 125–134, New York, NY, USA, 2010. ACM.

[4] Jacob T. Biehl, Mary Czerwinski, Greg Smith, and George G. Robertson. Fastdash: a visual dashboard for fostering awareness in software teams. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 1313–1322, New York, NY, USA, 2007. ACM.

[5] Eric Bodden. A high-level view of java applications. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pages 384–385, New York, NY, USA, 2003. ACM.

[6] A. Borici, K. Blincoe, A. Schröter, G. Valetto, , and D. Damian. Proxiscientia: Toward real-time visualization of task and developer dependencies in collaborating software development teams. In *In Proc*, CHASE 2012.

[7] John B. Bowen. Are current approaches sufficient for measuring software quality? In *Proceedings of the software quality assurance workshop on Functional and performance issues*, pages 148–155, New York, NY, USA, 1978. ACM.

[8] John Businge, Alexander Serebrenik, and Mark van den Brand. An empirical study of the evolution of eclipse third-party plug-ins. In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, IWPSE-EVOL '10, pages 63–72, New York, NY, USA, 2010. ACM.

[9] Marcelo Cataldo, James D. Herbsleb, and Kathleen M. Carley. Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, ESEM '08, pages 2–11, New York, NY, USA, 2008. ACM.

[10] Marcelo Cataldo, Patrick A. Wagstrom, James D. Herbsleb, and Kathleen M. Carley. Identification of coordination requirements: implications for the design of collaboration and awareness tools. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, CSCW '06, pages 353–362, New York, NY, USA, 2006. ACM.

[11] Jonathan Chow and Ewan Tempero. Stability of java interfaces: a preliminary investigation. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Metrics*, WETSoM '11, pages 38–44, New York, NY, USA, 2011. ACM.

[12] M.E. Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968.

[13] J. Corbin and A. C. Strauss. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Sage Publications, 3rd edition edition, 2007.

[14] D. Damian, L. Izquierdo, J. Singer, and I. Kwan. Awareness in the wild: Why communication breakdowns occur. In *Global Software Engineering, 2007. ICGSE 2007. Second IEEE International Conference on*, pages 81 –90, aug. 2007.

[15] Cleidson R. B. de Souza and David F. Redmiles. An empirical study of software developers' management of dependencies and changes. In *Proceedings of the 30th*

international conference on Software engineering, ICSE '08, pages 241–250, New York, NY, USA, 2008. ACM.

[16] Paul Dourish and Victoria Bellotti. Awareness and coordination in shared workspaces. In *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, CSCW '92, pages 107–114, New York, NY, USA, 1992. ACM.

[17] R.Geoff Dromey. Software quality-prevention versus cure? *Software Quality Journal*, 11(3):197–210, 2003.

[18] Jordan Ell and Daniela Damian. Supporting awareness of indirect conflicts with impact. *University of Victoria Computer Science*, 2013.

[19] M. E. Fayad and Shivanshu K. Singh. Software stability model: software product line engineering overhauled. In *Proceedings of the 2010 Workshop on Knowledge-Oriented Product Line Engineering*, KOPLE '10, pages 4:1–4:4, New York, NY, USA, 2010. ACM.

[20] Mohamed Fayad. Accomplishing software stability. *Commun. ACM*, 45(1):111–115, January 2002.

[21] Mohammed E. Fayad and Adam Altman. Thinking objectively: an introduction to software stability. *Commun. ACM*, 44(9):95–, September 2001.

[22] Norman E. Fenton and Niclas Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Trans. Softw. Eng.*, 26(8):797–814, August 2000.

[23] Geraldine Fitzpatrick, Simon Kaplan, Tim Mansfield, Arnold David, and Bill Segall. Supporting public availability and accessibility with elvin: Experiences and reflections. *Comput. Supported Coop. Work*, 11(3):447–474, November 2002.

[24] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, November 2007.

[25] Robert B. Grady. Practical results from measuring software quality. *Commun. ACM*, 36(11):62–68, November 1993.

[26] Christine A. Halverson, Jason B. Ellis, Catalina Danis, and Wendy A. Kellogg. Designing task visualizations to support the coordination of work in software development. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, CSCW '06, pages 39–48, New York, NY, USA, 2006. ACM.

[27] A.E. Hassan and K. Zhang. Using decision trees to predict the certification result of a build. In *Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on*, pages 189–198, Sept 2006.

[28] James D. Herbsleb, Audris Mockus, and Jeffrey A. Roberts. Collaboration in software engineering projects: A theory of coordination. In *In Proceedings of the International Conference on Information Systems (ICIS 2006*, 2006.

[29] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. Release pattern discovery via partitioning: Methodology and case study. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 19–, Washington, DC, USA, 2007. IEEE Computer Society.

[30] Reid Holmes and Robert J. Walker. Customized awareness: recommending relevant external change events. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 465–474, New York, NY, USA, 2010. ACM.

[31] Susan Horwitz and Thomas Reps. The use of program dependence graphs in software engineering. In *Proceedings of the 14th international conference on Software engineering*, ICSE '92, pages 392–411, New York, NY, USA, 1992. ACM.

[32] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

[33] Himanshu Khurana, Jim Basney, Mehedi Bakht, Mike Freemon, Von Welch, and Randy Butler. Palantir: a framework for collaborative incident response and investigation. In *Proceedings of the 8th Symposium on Identity and Trust on the Internet*, IDtrust '09, pages 38–51, New York, NY, USA, 2009. ACM.

[34] Miryung Kim. An exploratory study of awareness interests about software modifications. In *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '11, pages 80–83, New York, NY, USA, 2011. ACM.

[35] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. Automatic identification of bug-introducing changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society.

[36] Irwin Kwan and Daniela Damian. Extending socio-technical congruence with awareness relationships. In *Proceedings of the 4th international workshop on Social software engineering*, SSE '11, pages 23–30, New York, NY, USA, 2011. ACM.

[37] Irwin Kwan, Adrian Schröter, and Daniela Damian. Does socio-technical congruence have an effect on software build success? a study of coordination in a software project. *IEEE Trans. Softw. Eng.*, 37(3):307–324, May 2011.

[38] Arun Lakhotia. Constructing call multigraphs using dependence graphs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 273–284, New York, NY, USA, 1993. ACM.

[39] Andy Maule, Wolfgang Emmerich, and David S. Rosenblum. Impact analysis of database schema changes. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 451–460, New York, NY, USA, 2008. ACM.

[40] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.

[41] Audris Mockus and David Weiss. Interval quality: relating customer-perceived quality to process quality. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 723–732, New York, NY, USA, 2008. ACM.

[42] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 284–292, New York, NY, USA, 2005. ACM.

[43] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. The influence of organizational structure on software quality: an empirical case study. In *International conference on Software engineering*, pages 521–530, Leipzig, Germany, May 2008. IEEE.

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

[45] Marian Petre. Uml in practice. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 722–731, Piscataway, NJ, USA, 2013. IEEE Press.

[46] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. Can developer-module networks predict failures? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 2–12, New York, NY, USA, 2008. ACM.

[47] Anita Sarma, Gerald Bortis, and Andre van der Hoek. Towards supporting awareness of indirect conflicts across software configuration management workspaces. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 94–103, New York, NY, USA, 2007. ACM.

[48] Anita Sarma, Larry Maccherone, Patrick Wagstrom, and James Herbsleb. Tesseract: Interactive visual exploration of socio-technical relationships in software development. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 23–33, Washington, DC, USA, 2009. IEEE Computer Society.

[49] Adrian Schröter, Jorge Aranda, Daniela Damian, and Irwin Kwan. To talk or not to talk: factors that influence communication around changesets. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, CSCW '12, pages 1317–1326, New York, NY, USA, 2012. ACM.

[50] Francisco Servant, James A. Jones, and André van der Hoek. Casi: preventing indirect conflicts through a live visualization. In *Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '10, pages 39–46, New York, NY, USA, 2010. ACM.

[51] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 2005 international workshop on Mining software repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM.

[52] Amrit Tiwana. Impact of classes of development coordination tools on software development performance: A multinational empirical study. *ACM Trans. Softw. Eng. Methodol.*, 17(2):11:1–11:47, May 2008.

[53] Erik Trainer, Stephen Quirk, Cleidson de Souza, and David Redmiles. Bridging the gap between technical and social dependencies with ariadne. In *Proceedings of the*

2005 OOPSLA workshop on Eclipse technology eXchange, eclipse '05, pages 26–30, New York, NY, USA, 2005. ACM.

[54] Jason Tsay, Hyrum K. Wright, and Dewayne E. Perry. Experiences mining open source release histories. In *Proceedings of the 2011 International Conference on Software and Systems Process*, ICSSP '11, pages 208–212, New York, NY, USA, 2011. ACM.

[55] Mark Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, July 1982.

[56] Michel Wermelinger and Yijun Yu. Analyzing the evolution of eclipse plugins. In *Proceedings of the 2008 international working conference on Mining software repositories*, MSR '08, pages 133–136, New York, NY, USA, 2008. ACM.

[57] T. Wolf, A. Schroter, D. Damian, and T. Nguyen. Predicting build failures using social network analysis on developer communication. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 1–11, May 2009.

[58] Shujian Wu, Qing Wang, and Ye Yang. Quantitative analysis of faults and failures with multiple releases of softpm. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, ESEM '08, pages 198–205, New York, NY, USA, 2008. ACM.

[59] P. F. Xiang, A. T. T. Ying, P. Cheng, Y. B. Dang, K. Ehrlich, M. E. Helander, P. M. Matchen, A. Empere, P. L. Tarr, C. Williams, and S. X. Yang. Ensemble: a recommendation tool for promoting communication in software teams. In *Proceedings of the 2008 international workshop on Recommendation systems for software engineering*, RSSE '08, pages 2:1–2:1, New York, NY, USA, 2008. ACM.

[60] Andy Zaidman, Bart Rompaey, Arie Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Softw. Engg.*, 16(3):325–364, June 2011.

[61] Andreas Zeller. Isolating cause-effect chains from computer programs. *SIGSOFT Softw. Eng. Notes*, 27(6):1–10, November 2002.

[62] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[63] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 531–540, New York, NY, USA, 2008. ACM.