An Exploration And Discussion 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 And Discussion of Indirect Conflicts

by

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

Supervisory Committee

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

Dr. X. XXXXXX, Departmental Member
(Department of Computer Science)

Dr. Y. YYYYYY, Outside Member
(Department of Something)

Supervisory Committee

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

Dr. X. XXXXXX, Departmental Member
(Department of Computer Science)

Dr. Y. YYYYYY, Outside Member
(Department of Something)

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 inerest 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 in the low moments.

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 How to Start an Introduction

1.2 Is a Review of All Previous Work Necessary Here?

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 thesis as well as give a more full and rich description of the problem being solved.

In this chapter, two studies will be presented that I conducted 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 a developer to any source code changes preformed by another developers when the two are linked in a technical dependency through a developer pair. *Impact* utilized a non-obtrusive RSS style feed for notifications for visualization. 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 [29, 32, 34].

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 [28, 38]. 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 [22]. 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 predicting software failures, the question I investigated in this study was : "*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, we 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 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.

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.

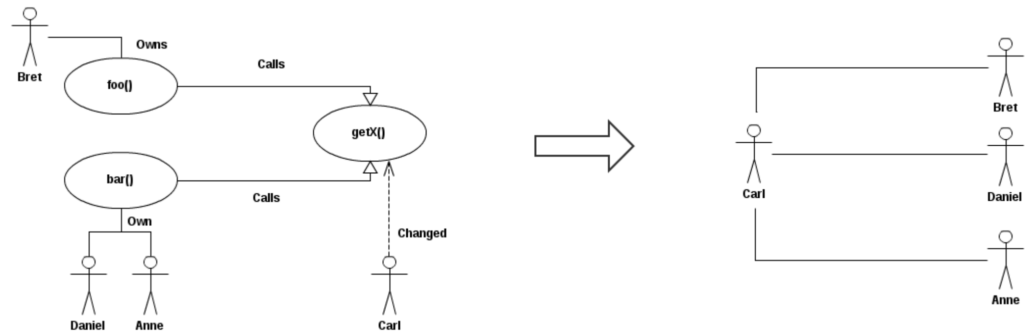

Figure 2.1: A technical network for a code change. Carl has changed method getX() which is being called by Bret's method foo() as well as Daniel and Anne's method bar().

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. [3] 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 give 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. [33] 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.

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

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).

2.1.2 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 [33].

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

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.

2.1.3 Conclusions of Study

Technical dependencies are often used to predict software failures in large software system [22, 28, 38]. 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 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 futher, 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 [2,20,30,37]. 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 [1,29,32,34]. 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 [1, 10, 16, 31].

Indirect conflicts arising in source code are inherently difficult to resolve as most of the time, source code analysis or program slicing [35] 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 [1, 34], most have only created an exploratory environment which is used by developers to solve conflicts which may arise [32]. These tools were not designed to detect indirect conflicts that arise and alert developers to their presence inside the software system. Sarma et al. [29] 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, implementation, and future 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 [24] 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 [25] 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.

2.2.1 Impact

This section will proceed to give an outlined detail 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*.

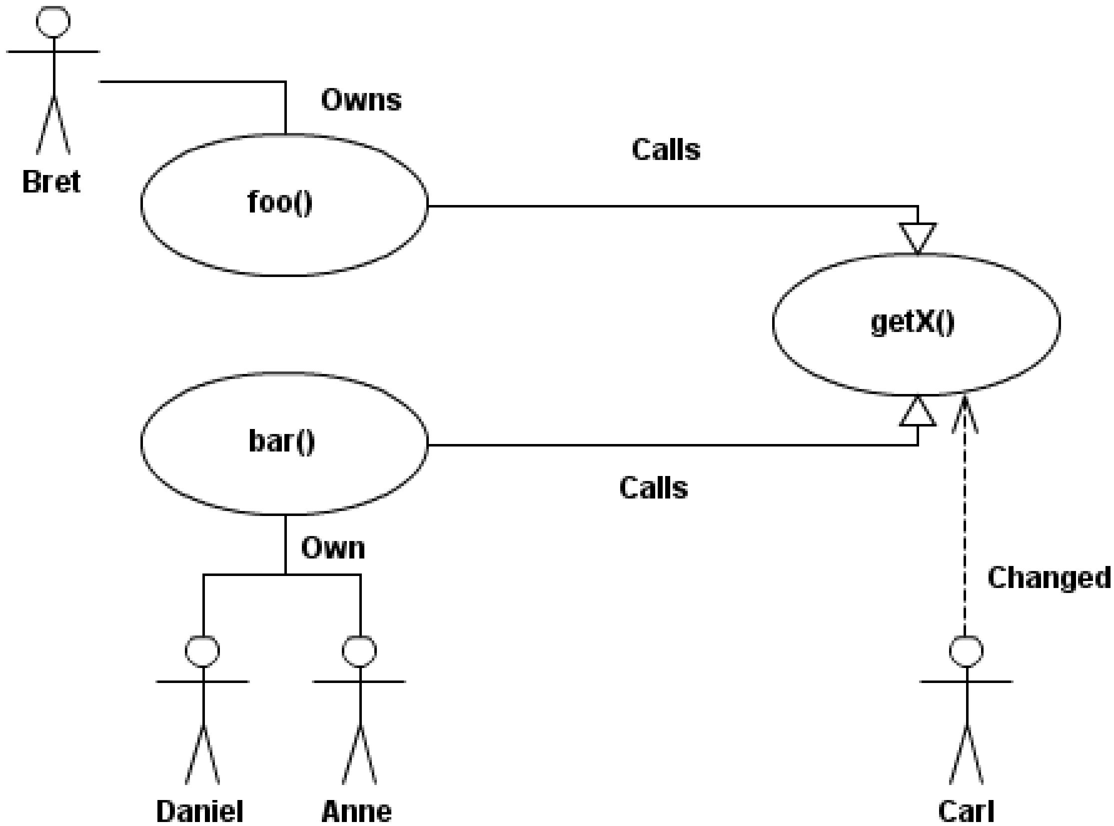

Figure 2.2: Technical object directed graph with ownership

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 [18]. 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, we 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 [13]. 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 apart of 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 [32], 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 [23] 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.

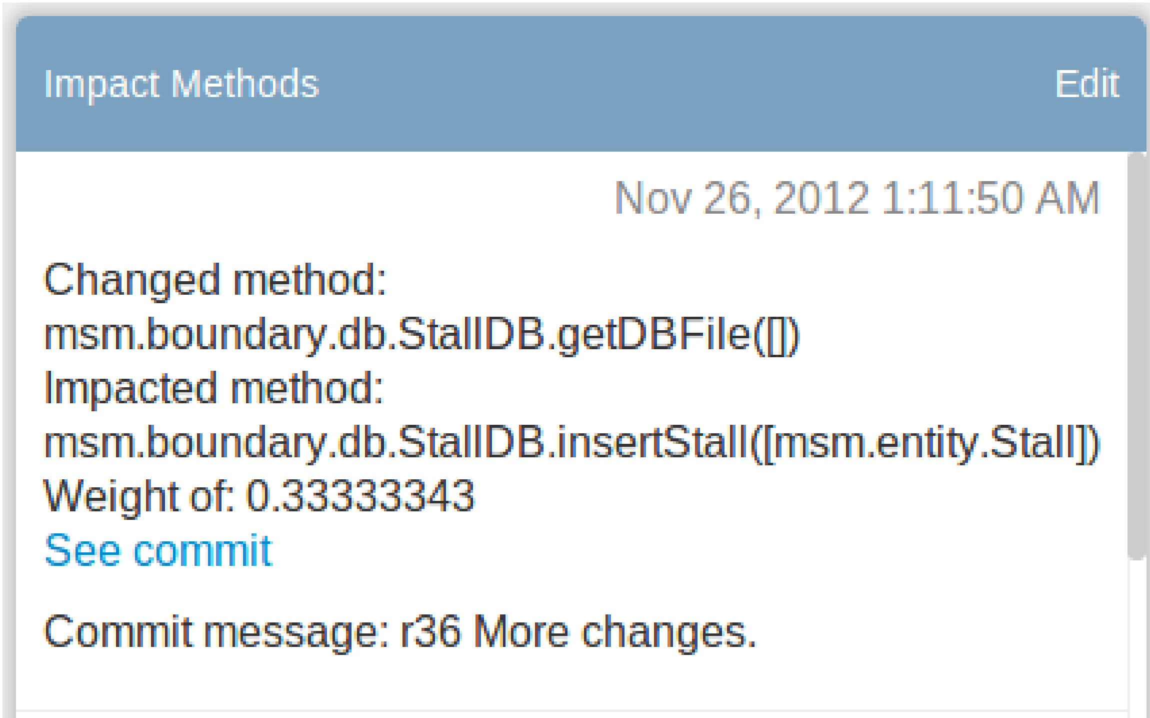

Figure 2.3: *Impact's* RSS type information feed.

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.

2.2.2 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 to large teams or projects as opposed to the small teams which they were apart 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 to indirect conflicts 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.

2.2.3 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 [25] 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. [14] 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

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” [11]. 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 [27].

Through the two previous studies, I have shown that developers linked indirectly in source code changes can 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 [29, 32]. 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 [34], and class signatures [29] 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. [7] notion of socio-technical congruence, have also been leveraged in indirect conflict tools [1, 4, 23]. 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 [17, 21]. In order to fully understand the root causes of information overload, false positives, and scalability issues in regards to indirect conflicts, I will proceed by taking a step back 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 set out to understand what mitigation strategies developers currently use as opposed to those created by academia. 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 academia, 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 academia. Finally, I look to find what can be accomplished moving forward with indirect conflicts in both research and industry.

I restate the research goals of this thesis for ease of the reader:

What events or conditions lead to indirect conflicts?

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

What do developers want from future indirect conflict tools?

To answer these research questions, 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 survey of 78 developers, and 5 confirmatory interviews. Based on the findings (to be seen) to the aforementioned research questions, I also performed a secondary complimentary study of software evolutionary trends (Section 3.2) to solidify developer opinion and provide a starting point for the future development of indirect conflict tools.

3.1 Study 3: An Exploration of Indirect Conflicts

In order to fully understand the root causes of information overload, false positives, and scalability issues in regards to indirect conflicts, I conducted 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 then set out to understand what mitigation strategies developers currently use as opposed to those created by academia. Through this exploration, I look to find what can be accomplished moving forward with indirect conflicts in both research and industry.

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. 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 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 survey 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: IBM, Mozilla, The GNOME Project, Microsoft Corporation, Subnet Solutions, Ruboss Technology Corporation, Amazon, Exporq Oy, Kano Apps, Fireworks Design, James Evans and Associates, and Frost Tree Games. My participants were invited based on their direct involvement in the

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.

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 at some capacity. 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 our three research questions. Specifically, ten semi-structured topics directly related to our research questions guided our interview:

- Software development tools for dependency tracking and awareness.
- Software development process for preventing indirect conflicts.
- Software artifact dependency levels and where conflicts can arise.
- How developers find internal or external software dependencies.
- Examples of indirect conflicts from real world experiences.
- How indirect conflicts are detected and found.
- How indirect conflict issues are solved or dealt with.
- Developer opinion of preemptive measures to prevent indirect conflicts.
- Developer opinion on what types of changes are worth a preemptive action.
- Developer opinion on 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 apart of any one software project for extended periods of time.

Survey Participants

My survey respondents were different from my interviewees. I invited our survey 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 survey 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 survey 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 survey. 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.

Survey Procedure

Survey participants were invited to participate in the survey by email. No reminder email was sent as the survey 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 surveys were conducted during the months of July and August while many participants may be away from their regular positions. and 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 survey was considered by some to be long and require more development experience than may have been typical of some of those invited to participate.

The survey 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 survey went through two rounds of piloting. Each pilot round consisted of five participants, who were previously interviewed, completing the survey with feedback given at the end. Not only did this allow me to create a more polished survey, but it also allowed the previously interviewed developers to examine the insights I

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

developed.

Data Analysis

To analyze both the interview and survey data, I used grounded theory techniques as described by Corbin and Strauss [9]. Grounded theory is a qualitative research methodology 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 surveys (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 3.1.3, 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' [9] list of criteria to evaluate quality and credibility. This evaluation can be seen in Section 3.1.3

3.1.2 Results

I now present my results of both the interviews and surveys 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.

What events or conditions lead to indirect conflicts?

From the interviewed participants, 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 interviewed 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 interviewed 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.

From the interviewed participants, 11 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. 65% of surveyed developers answered that indirect conflicts occur on minimum bi-weekly, with the majority of developers saying that weekly occurrences are most common.

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

12 developers interviewed 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". Surveyed developers 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.". Surveyed 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.

Occurrences	Daily	Weekly	Bi-Weekly	Monthly	Bi-Monthly	Yearly	Unknown
Early stages of 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 survey questions to how often indirect conflicts occur, in terms of percentage of developers surveyed.

In terms of organizational structure, surveyed 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.

As per organizational structure, Table 3.3 shows which development team environment developers believe to be the most prone to indirect conflicts.

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: Results of survey questions to development environments in which indirect conflicts are likely to occur, in terms of percentage of developers surveyed.

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

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 interview 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 developers surveyed 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.

In regards to catching indirect conflicts, 13 interviewed 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. 31% of surveyed developers 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 surveyed developers 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. Surveyed developers 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 interviewed 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 developers surveyed, 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?

When asked about preventative tools, interviewed 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, interviewed 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. 68% of surveyed developers said that they would always want to be notified about method signature changes as they have a high chance to break the code as opposed to only 23% who always wanted to be notified on a pre or post condition change and 27% who want to be notified for a user interface change. Other change types varied from never being notified to most times being notified, showcasing the complexity of change types which may or may not negatively affect a project. A complete breakdown of change types and developer preference can be found in Table 3.4.

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: Results of survey questions to source code changes that developers deem notification worthy, in terms of percentage of developers surveyed.

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

- 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.

3.1.3 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 paper 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 [9]. 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

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

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 3.1.1.

To ensure the correctness of the results, I also linked all findings in Section 3.1.2 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 academia to better benefit industry.

10 of the 78 participants who were surveyed 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 academia 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 surveys, 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 3.1.2 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.

3.1.4 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 survey of 78 developers, and 5 confirmatory interviews.

My findings indicate that: indirect conflicts occur frequently and are likely caused by software contract changes and a lack of understanding of change impacts, indirect conflicts occur quite frequently with a tendency to be more prevelant at the beginning of a development cycle, indirect conflicts occur more often as a software project grows in developer numbers, developers use design by contract, add and deprecate, and experience to mitigate indirect conflicts, and finally, developer prefer to have tools which help them debug quicker once a conflict has occured rather than tools which help prevent possible conflicts for the future.

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 minor 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 [5] [15] [19], the sub 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 [8]. Code churn is an often looked to statistically for stability but can be grossly misleading in terms of pre-release and post-release defects [12], with some exceptions [26] 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.

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 [36] as well as evolving dependencies [6]. 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: "*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 quantitative and 16 qualitative change trends 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 and 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 Methodology

In order to answer my research question, I decided to use the tool ChangeDistiller created by Fluri et al. [14]. 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 took ChangeDistiller and applied it across 10 open source Java projects. Java projects were necessary because ChangeDistiller only works for Java source code. 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 project's 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 our 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 our study, we 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 3.2.2

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

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 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.

3.2.2 Results

To answer my research question, 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,

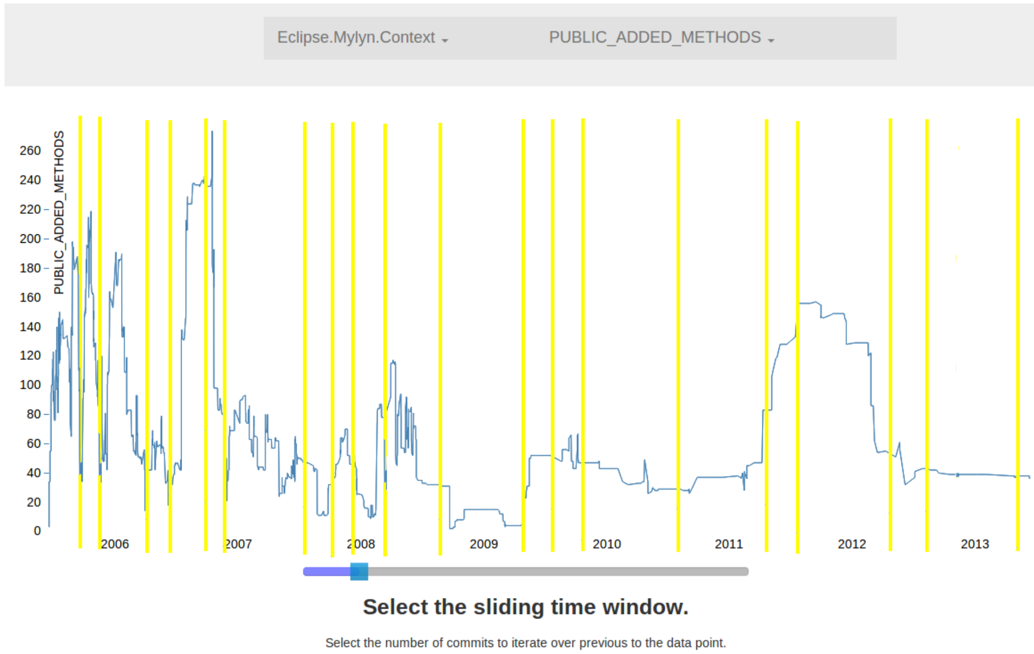

Figure 3.1: A screen shot of the APIE visualizer showing project Eclipse.Mylyn.Context with change type PUBLIC_ADDED_METHODS being analyzed and showing major releases as vertical yellow lines.

and eclipse.maven.surefire. These project were chosen because of their high use amongst other Java projects and to study specific ecosystems of projects and their evolution trends.

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.

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 after. 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 ob-

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.

jects. 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 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 our 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.

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.

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.

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.

3.2.3 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 as 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 paper 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 conflicts 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

Chapter 5

Conclusions

Appendix A

Additional Information

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] A. Borici, K. Blincoe, A. SchrÃ¶ter, 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.

[5] 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.

[6] 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.

[7] 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.

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

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

[36] 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.

[37] 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.

[38] 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.