**Timely and Efficient Facilitation of Coordination of Software Developers' Activities**

A Thesis

Submitted to the Faculty

of

Drexel University

by

Kelly Coyle Blincoe

in partial fulfillment of the

requirements for the degree

of

Doctor of Philosophy

January 2014

## Dedications

For my family and friends who supported and encouraged me.

**Acknowledgements**

I would like to express my deepest gratitude and appreciation to my advisors, Dr. Giuseppe “Peppo” Valetto and Dr. Daniela Damian. Thank you for helping me to grow as a researcher and for your encouragement. I would not have been able to accomplish all that I did without your motivation, ideas, and suggestions. I would also like to thank the other members of my thesis committee, Dr. Sean Goggins, Dr. Spiros Mancoridis, and Dr. Dario Salvucci. Thank you for the time you dedicated to my thesis and for your feedback.

This thesis would not have been possible without the support and encouragement of my family, for which I cannot begin to express my gratitude. A special thanks to my wonderful husband, Adrian, who encouraged me to start this journey and supported me in many ways throughout the journey. Also, thanks to my parents, Frank Coyle and Dr. Joanne Coyle, who always believed in me and instilled in me the belief that I can achieve anything I set my mind to. Finally, thanks to my daughter, Ella, who inspires me everyday.

## Table of Contents

## List of Tables

## List of Figures

**Abstract**

Timely and Efficient Facilitation of Coordination of
Software Developers' Activities
Kelly Coyle Blincoe
Advisors: Giuseppe Valetto, Ph.D. and Daniela Damian, Ph.D.

When software developers fail to coordinate, build failures, duplication of work, schedule slips and software defects can result. However, developers are often unaware when they need to coordinate, and existing methods and tools that help make developers aware of their coordination needs do not provide timely awareness or efficient recommendations. Without timely awareness, developers cannot act on their coordination needs while development is underway. Further, existing tools recommend only which developers should coordinate. This introduces inefficiencies since developers are often working on multiple tasks in parallel. This dissertation describes a set of techniques that aim at improving the timeliness and efficiency of coordination recommendations. It introduces a method that provides timely coordination recommendations by analyzing developer actions as they occur using IDE monitoring facilities. It presents an approach that identifies coordination needs between pairs of tasks and leverages additional task properties and machine learning to identify a subset of the coordination needs that are more critical for the developers' work. This dissertation describes a series of investigations of coordination needs on eight releases of the Mylyn project. Our techniques were validated through a mixed methods approach including statistical analysis, in-depth examination of task records, and developer interviews. Our research shows that coordination recommendations can be made both timely and efficient by applying the techniques described in this thesis.

# CHAPTER 1: INTRODUCTION

This chapter motivates the research described in this dissertation. It introduces the importance of coordination in software development projects and describes how a developer's lack of awareness of coordination needs can introduce problems and inefficiencies into the development process. It provides a high level summary of the focus of this dissertation and highlights its contributions. It concludes by describing the organization of the remaining chapters.

## 1.1 Research Problem

Today's software development projects are becoming increasingly large and complex. Large software projects have a large number of work dependencies [115], which are the technical dependencies that exist in software projects. Examples of technical dependencies are syntactic or semantic dependencies between software artifacts. Dependencies exist between tasks when those tasks involve dependent artifacts. Work dependencies between development tasks can lead to coordination needs between the assignees of those tasks [27], [28], [30], [44], [73]. It is well recognized that work dependencies must be managed in software development projects to avoid integration problems and software failures [29], [73], [120]. Initially, beginning with Parnas' recognition of the workflow implications of modularization [105], research focused on ways to streamline the technical dependencies between modules as a way to maximize task parallelism [7], [139]. However, it is not possible to eliminate all inter-module dependencies in large software projects. Therefore, research began to focus on ways to satisfy, as opposed to reduce, work dependencies

through coordination [73]. Communication is the main form of coordination in software teams [87], and software developers spend a large amount of time communicating [106]. It has been found that a decrease in communication can cause team members to be unaware of work dependencies resulting in coordination problems [43], [63], [72]. Herbsleb et al. found that when developers are willing to communicate directly, integration problems are reduced [71]. Kwan et al. found that aligning the coordination in software teams based on the tasks they must complete can bring about productivity benefits [89]. This is in line with the intuition by Conway [33], who was the first to describe the possibility of such an alignment in software engineering projects.

However, even if developers are willing and able to coordinate, they may often be unaware of their coordination needs. This can be complicated by the fact that developers' coordination needs are often fluid and change throughout the course of development [42]. This fluidity contributes to a lack of awareness of coordination needs among developers. Dourish [47] defines awareness as "*an understanding of the activities of others, which provides a context for your own activity.*"

A lack of awareness of coordination needs can lead to missed coordination, which can result in build failures, duplication of work, schedule slips and software defects [27], [28], [30], [39], [41], [44]. Therefore, providing awareness of coordination needs can help improve software productivity and quality. To be effective, awareness must be timely, and it must provide enough information to allow developers to fully understand their coordination needs and act on them efficiently.

Achieving timely awareness of coordination needs in large software engineering projects remains an open problem. Configuration management conflict

detection tools, like Palantír [111], [112] and CollabVS [45], were among the first attempts to provide such awareness. They help alert developers of possible conflicts by letting them know which other developers are making changes to the files they are currently modifying (direct conflicts). They also provide only limited support for indirect conflicts where one developer makes a change in one artifact that affects another developer's work in a separate artifact. For example, Palantír includes only one very specific type of indirect conflicts that occur when class signatures are conflicting. However, these conflicts are only a subset of all possible coordination needs, so tools like these do not provide a comprehensive view of coordination needs to developers. Cataldo et al. [27], [28], [30] were the first to introduce a framework for establishing a more comprehensive view of coordination requirements between developers. Many awareness tools [10], [44], [97], [110] have been created based on their method. However, their method relies on commit data. This data is typically available only towards the end of the development work for a task, so the awareness this approach provides may not be timely. Without timely awareness of coordination needs[1], developers are not able to focus their coordination to reap the proven performance and quality benefits.

Existing approaches also provide inefficient recommendations since they require developers to take time away from their development efforts to better understand their coordination needs. The configuration management conflict detection tools provide a stream of notifications regarding each potential conflict at the source code level. This approach is likely to cause information overload for developers,

---

[1] The terms "coordination needs" and "coordination requirements" are used interchangeably throughout this dissertation.

especially since any concurrent modification to the same artifact will generate a notification regardless of complexity. This brings about inefficiency since the developers are potentially required to sift through a large number of notifications to determine which conflicts really matter. The coordination requirement detection tools also risk information overload, especially when the team is large [26], [40]. Moreover, they provide awareness of only which pairs of developers should coordinate. Since developers may work on multiple tasks in parallel, coordination requirements at the developer level may encompass the work dependencies of many tasks. This puts the burden on the developers to identify which tasks require coordination and introduce inefficiency.

This work focuses on providing timely and efficient awareness of coordination needs by identifying coordination needs at the task level and focusing on the more critical coordination needs.

### 1.2 Research Questions and Methodology

This dissertation focuses on solving these issues of current coordination requirement detection methods and providing timely and efficient coordination recommendations. We first sought to explore techniques for providing timely awareness of coordination needs to software developers. Current methods rely on commit data, thus are not timely. Without timely awareness, developers are not able to act on their coordination needs. Our first research question addresses this problem:

*RQ1: Is timely coordination requirement detection possible?*

We developed a new method and metric, called Proximity, which detects coordination needs between pairs of developers in a timely way. This timely detection of coordination needs provides awareness to developers while their work is still

underway. Developers can act upon and resolve their coordination needs as they surface. In the words of one senior developer that we interviewed, "*If you find out next week that you should have talked to this guy last week, that's not helpful. Real-time collaboration is a better choice.*" Chapter 4 describes the Proximity metric in detail, shows how it is timelier than existing methods, and shows how it can be even more accurate in identifying actual coordination requirements.

Developers are often working on many tasks in parallel, so being aware of only which other developers they need to coordinate with does not provide enough context to allow for focused and efficient coordination. We address this with our second research question:

> *RQ2: Can coordination requirements be identified efficiently at the task level of granularity?*

Ko et al. [85] found that developers are especially interested in awareness about what information was relevant to their tasks. Additionally, through developer interviews, we found that developers would prefer awareness of coordination needs that exist between pairs of tasks, since tasks are their unit of work. We adjusted Proximity to identify coordination requirements between pairs of tasks - instead of developers - to provide better-scoped awareness and allow for more efficient coordination. However, current coordination requirement detection algorithms (including Proximity) cast too wide a net when computed at the task level, since they consider all work dependencies between pairs of tasks as potential coordination needs. Therefore, to provide efficient recommendations at the task level, we introduced ProximityML, an approach to reduce the set of coordination recommendations by identifying the more critical

coordination needs. We measure criticality of coordination requirements by task complexity (change size) and task performance (task duration).

To devise ProximityML, we examined several task properties that could enhance measures like Proximity to identify the more critical coordination needs between task pairs. ProximityML uses machine learning on Proximity and those other identified task properties to reduce the set of coordination needs, while focusing on the more critical ones.

To evaluate the accuracy of the ProximityML results, we compared its results to the actual coordination needs experienced by the software development team. Since current software project repositories only partially capture this information [6], we established a method and a set of guidelines to extract the ground truth of coordination requirements experienced by the team from the task records obtained from the software repositories. We used this ground truth to evaluate our results. Chapter 5 describes our ProximityML approach and shows how it is able to detect a set of the more critical coordination needs.

Finally, we analyzed whether the ProximityML approach allows for timely detection of coordination needs as they emerge. This analysis aims at answering our third and final research question:

*RQ3: Are the more critical coordination needs actionable?*

To answer this research question, we streamed each event (developer actions and task updates) in a time-ordered sequence and re-ran the ProximityML approach after each event. This allowed us to evaluate the exact moment when ProximityML first recognizes a coordination need. Chapter 6 describes this evaluation exercise, which assesses the consistency of the results over the duration of one major project release

as well as the timeliness of the coordination needs detected by ProximityML. It also addresses the usability and actionability of the coordination recommendations made by our approach through developer interviews.

### 1.3 Contributions

This dissertation describes a set of techniques that aim to provide timely and efficient coordination recommendations. Furthermore, it evaluates those techniques through a mixed-methods approach and describes a number of studies that show that timely and efficient coordination recommendations are possible. The key contributions of this dissertation are:

1. It provides a method for timely and accurate detection of coordination needs between software developers.
2. It provides an approach for timely and accurate detection of coordination needs at the level of tasks to provide more granular and efficient recommendations.
3. It provides an approach to avoid information overload by identifying a set of the more critical coordination needs at the task level.
4. It describes a method for identifying the ground truth of coordination needs experienced during development work by examining task reports.
5. It discusses developer needs gathered through interviews and discusses the implications of our work for research in coordination within software teams and the design of support tools for collaborative software development.

### 1.3 Dissertation Roadmap

The rest of this dissertation is organized as follows: Chapter 2 outlines related research on 1) coordination in software engineering, 2) modularity of software design, 3) awareness in software engineering, and 4) providing awareness of coordination needs. Chapter 3 describes our research setting, research questions and methodology. Chapter 4 describes our approach for providing timely coordination recommendations. Chapter 5 describes our approach for efficient coordination recommendations. Chapter 6 evaluates the feasibility and usability of a tool that could be implemented using our methods. Chapter 7 discusses the contributions and how our approach can be applied to larger contexts. Finally, Chapter 8 concludes with a discussion on the implications for research in coordination and tool design and points out opportunities for future work.

# CHAPTER 2: BACKGROUND

This chapter provides an overview of the main areas of research that are relevant to the research in this dissertation: 1) coordination in software engineering, 2) modularity of software design, 3) awareness in software engineering, and 4) providing awareness of coordination needs.

## 2.1 Coordination in Software Engineering

### *2.1.1 Need for Coordination*

In their seminal paper, Kraut and Streeter [87] argued that tight coordination is required among development team members in order to deliver a successful software system. Unfortunately, they found that there are several problems inherent in software development projects that make such coordination difficult. They note several software characteristics – scale, interdependence, and uncertainty – that cause unavoidable coordination problems.

**Scale:** Software systems are becoming increasingly large, thus making scale a particularly significant characteristic. Often, projects involve millions of lines of code and the development cycle spans multiple years. The size of these projects makes it impossible for any one individual or even a small group of individuals to fully understand all details of the system being developed. When projects become large, it is necessary to divide the development work among several teams of developers. This can create efficiency by allowing teams to work in parallel. However, parallel streams of work must eventually be integrated, which introduces additional coordination

needs. Moreover, developers are often separated by geographic, organizational or social boundaries, and these boundaries can create coordination barriers [87].

**Interdependence:** Software that has been broken into small components to be developed independently by many teams or developers must eventually be integrated into one deliverable software system. There are often many dependencies between the various components. In order for the end system to function correctly, the components must work together properly. Integration of software must be very precise. Lack of coordination among developers working on dependent components can lead to integration problems [71], [87].

**Uncertainty:** Software development work is subject to continuous change, which causes many difficulties and produces ongoing coordination needs. Requirements can change over time due to changes in user needs, hardware changes or changing business needs. Requirements also tend to be incomplete, usually due to the requirement engineers' lack of domain knowledge. The developer responsible for implementing an incomplete requirement frequently interprets the requirement incorrectly. Also, requirement engineers often introduce errors into requirements when translating the many needs and points of view of all the different stakeholders into requirements [87].

These characteristics are inherent in modern software projects and introduce coordination overhead. While steps can be taken to reduce this coordination overhead, the need to coordinate cannot be completely eliminated in any project [87].

### *2.1.2 Types of Coordination*

#### *2.1.2.1 Explicit Coordination*

Explicit coordination is direct coordination such as email, chat, face-to-face meetings, or phone calls. Kraut and Streeter [87] found that communication - both formal and informal - is the main form of coordination that occurs during software development. Perry found that software developers spend a large part of their work time communicating [106]. Both formal and informal communications are useful coordination mechanisms. Formal communication, including written documents like requirement specifications and structured meetings, is useful during routine software development. Informal communication is most valuable whenever high levels of uncertainty are present in the project. Informal communication is often more personal and interactive than formal communication, including telephone calls, emails or impromptu face-to-face discussions between a pair of individuals or within a small group. Informal communication is more likely to occur between individuals who are in close physical proximity. While this form of communication is valuable, it can often be inefficient or imprecise [87].

#### *2.1.2.2 Implicit Coordination*

Implicit coordination consists of consequential communication – obtaining information about a task by watching another developer as they complete that task – and feedthrough – obtaining information about a task by examining changes to artifacts [66]. The latter is an example of implicit coordination via stigmergy. "*A process is stigmergic if the work done by one agent provides a stimulus that entices other agents to continue the job*" [74]. In software development, stigmergic

coordination occurs when enough information is contained within a software artifact to enable a new developer to pick up that software artifact and complete a task that is already underway or start a new dependent task without resorting to explicit coordination. This type of coordination happens very frequently in open source development [74]. Elliot [52] argued that stigmergy is most important on large open source development teams (> 25 people).

Stigmergy can be direct or indirect. Direct stigmergy occurs when the content of the software artifact promotes or facilitates later contributions. The simplest example of direct stigmergy in software development is code comments. Indirect stigmergy occurs when the side effects of work being performed cause additional work to be completed [74]. Examples of indirect stigmergy are posting comments on a development task and the modification of its state within a bug tracking tool. Bolici et al. [17] investigated how trading zones and boundary objects are being used as stigmergic forms of coordination on software development teams. A trading zone facilitates cross-boundary coordination by agreeing on common terms and processes [81]. A boundary object is an artifact that allows coordination of different perspectives across multiple stakeholders [122], [123].

Recently, many software development tools have shifted from supporting the work of an individual developer to encouraging team-based and social software development. Features like tags, feeds, and microblogging made popular through social media sites such as Facebook and Twitter have been incorporated into software development tools. Many of these features support implicit coordination.

***Tags***

A tag is a user-defined keyword that is attached to an item to help describe it. Yew et al. [144] described social tagging as *"the collaborative activity of marking shared content with keywords, or tags, as a way to organize content for future navigation, filtering or search."* Tags have been introduced into numerous software development tools. One of the first ways to tag source code was through the use of Java code annotations, such as TODO, FIXME, or HACK. Storey et al. [126] found that developers use these annotations to help manage their tasks.

In the IBM® Rational Team Concert team-oriented software development environment, also known as Jazz, developers can create source code annotations using snippets of chat conversations to help document important decisions made through informal communication channels [77]. Jazz also allows developers to tag work items. A work item is equivalent to a bug report, modification request or change request in other development environments. Work items are assigned to developers as tasks in the Jazz environment. Developers use tags mainly to organize and categorize [129], [130]. While developers indicated that tags are not used to directly communicate with other members of the team, developers tag not only their own work items, but also the work items of other developers [129], [130]. This could indicate that tags are being used for various forms of stigmergic collaboration.

TagSea, an Eclipse plug-in, is another tool that provides a tagging feature through a feature called Waypoints [125]. Waypoints are implemented as a series of tags that can be used to save locations of interest and create a "map" or an itinerary through the code base. A series of Waypoints, called a route, can help guide developers through the code. Developers can share Waypoints as a form of

collaboration. A preliminary evaluation of TagSea found that developers used tags and Waypoints to: 1) temporarily mark areas of code they are changing; 2) relay information about code changes to other team members; and 3) implement similar tasks by following Waypoint routes created by other developers.

One downfall of using user-defined keywords as tags is that different tags can refer to the same concept. This can make searching and filtering on tags more difficult, especially in software engineering where domain specific verbiage is often used. Wang et al. [135] studied the tags associated to 45,470 projects on Freecode[2], a project hosting site, and created a similarity metric for the words used in those tags to infer semantically related software terms. They used this similarity metric to group related tags. Through user studies, they found users agreed with their similarity metric and the resulting taxonomy. Other tools, like TagRec [3] and TagCombine [140], have proposed techniques that recommend tags based on similar tags that have previously been used on related objects. These tools could make tagging objects in software development projects easier.

***Feeds***

Feeds broadcast project-related or user events such as build results, modifications to tasks or incoming tasks. Treude and Storey [128] found that, in Jazz, developers use feeds to track work, get information, and understand common practices. On GitHub, users can subscribe to activity feeds by "watching" a project or "following" a user. These feeds include information on code related changes, issues and comments.

---

[2] Freecode: http://freecode.com

Calefato et al. [23], [24] introduced a tool called SocialCDE whose goal is to increase trust in global software development teams by enhancing the feeds provided by current tools with additional social information. SocialCDE displays developer's personal social content obtained from sites like Facebook, Twitter and LinkedIn directly within a developer's development environment.

The amount of information available within feeds can quickly lead to information overload [128]. Yet, Dullemond et al. [48] found distributed software engineers are interested in a diverse set of information and the relevant information changes frequently. Individual developers often have different opinions on the most relevant information [50]. Fritz and Murphy [57] found that developers use four factors in determining the relevancy of information within a feed. These factors are: 1) content, 2) target of content, 3) relation with the creator, and 4) previous interaction. Feeds can be improved by considering all of these factors to include only relevant information.

***Dashboards***

Jazz also provides web-based dashboards that help keep developers aware of what is happening on a software project or within a software team. The Jazz dashboard provides high-level project health information and makes it easy to navigate and drill down to more complete information. Treude and Storey found that developers use dashboards to achieve a high-level overview of project-status and an understanding of what other developers/teams are working on [128].

***Wikis, Blogs and Microblogs***

Software developers have adopted other social media tools like wikis, blogs and microblogs for collaboration. Al-asmari and Ya [1] found that wikis are easy to

use, reliable and inexpensive when compared to other coordination methods. Several wiki tools, like WikiDev 2.0 [9], Galaxy Wiki [142], and Wikigramming [69], have been created to specifically support software development activities. Huh et al. [76] found that blogging allows easy access to knowledge and can serve as a coordination mechanism.

Microblogging became popular through Twitter. Many companies have adopted Yammer, a commercial product that provides features similar to Twitter for individual companies. Guzzi et al. [68] created James, an Eclipse plugin, which combines microblogging with developer activities collected through IDE monitoring. In their study, participants used James to communicate future intentions, indicate the status of ongoing and concluded activities, comment, and mark future tasks. They found that developers are willing to microblog and that microblogging is helpful in maintenance tasks. Dullemond et al. [49] found that distributed software developers used microblogging to coordinate. They also found that microblogging helped team members become more connected, and it provided easier access to information. Wang et al. [136] found that microblogging also plays an important role in open source software communities through their analysis of Twitter accounts.

***User profiles***

On GitHub, user profiles are publicly accessible and include identifying information, a list of projects the user is currently working on, and the user's recent activity on those projects. SCI [8], a collaborative development environment, also incorporates user profiles including information like project activity, technical interests, currently active sessions, and availability. CARES [67] populates source code files with photos of developers who have recently modified the software artifact.

Each photo is augmented with a tooltip containing user profile type information like previous code commits, position in the organization, physical location, and availability.

Dabbish et al [37] found that GitHub users make use of the social information available in user profiles, in addition to the information collected via feeds, to make *"a rich set of inferences around commitment, work quality, community significance and personal relevance. These inferences support collaboration, learning, and reputation management in the community."* Marlow et al [96] found that GitHub users use the information from user profiles to form impressions of each other and make judgments about potential contributors, which then influence whether or not their code contributions are accepted.

All of the features discussed above have potential as means for implicit coordination. The information reported in both feeds and dashboards is generated directly from the work done by developers, and no additional work is required on the part of the developer to populate these information sources. Up-to-date information is then constantly displayed to the rest of the team. Developers are able to gather information from these sources without explicitly communicating with any other team members. Tags, wikis, blogs, microblogs and user profiles need to be created by developers, but this can be done when it is most convenient for the developer. The rest of the team can then use the information contained in these sources without the need for explicit coordination. These coordination methods can help archive important design decisions and task details while development is underway for easy referencing on future tasks. More research is still needed on how implicit coordination

can be encouraged and facilitated to increase awareness in software development teams and reduce the need for more expensive forms of coordination.

### *2.1.3 Coordination Problems*

Brooks [20] observed the problems associated with coordination in software projects. He famously explained that adding more people to project that is already behind schedule further delays the project due to the added project coordination and communication overhead. Coordination can be even more difficult when the involved developers span team boundaries. Sosa et al. [120] found that when cross-boundary dependencies exist, developers often do not coordinate due to a lack of awareness of the importance of the coordination as well as a lack of social relationships across teams. They found that the lack of coordination resulted in integration problems. In an empirical study, Curtis et al. [36] found that coordination is one of the biggest problems in large software projects.

de Souza et al. [44] found that developers are not always aware of their coordination needs and that when developers are unaware of the coordination that is required to manage their work dependencies, problems can occur. Studies have found that unfulfilled coordination needs can result in an increase in task resolution time, an increase in software faults, build failures, redundant work, and schedule slips [27], [28], [30], [39], [41], [44].

The literature described here shows that awareness of coordination needs is a critical concern in large software projects.

## 2.2 Modularity

Today's software systems are increasingly large and complex. It is, therefore, necessary to divide the development work among numerous developers. Breaking the implementation of a large software program into independent modules is a well-established software programming technique. Modules are defined as elements of a larger system that are independent yet work together [7]. Simon [121] and Alexander [5] argued that decomposing a system was a logical, and perhaps the only, response to the complexity that was inherent in solving such large problems. Simon [121] said nearly decomposable systems, where interactions between the components are weak but not negligible, were ideal. Parnas [105] recognized that it is possible to reduce coordination needs by minimizing the technical dependencies between software modules. Modules can then be developed independently in parallel and later integrated to form a complete product.

Parnas [105] defined modules as "*a work assignment unit rather than a subprogram.*" His definition of modules is based on an earlier definition made by Gauthier and Ponto [61], which states "*each task forms a separate, distinct program module.*" Parnas argued that modularization is a mechanism for improving flexibility and comprehensibility of a system. He provided criteria for decomposing a system into modules based on his information hiding principle.

Parnas' information hiding principle [105] was very influential, and it forms the foundation for many modern approaches to reduce dependencies between modules. He argued that modules should expose only stable interfaces and should hide anything that is likely to change. By exposing only the stable interfaces that are

not likely to change, changing one module should not impact other parts of the software. Once the interfaces have been defined, modules can then be developed independently. This allows developers to work in parallel without the need for coordination.

Baldwin and Clark [7] defined design rules as the high-level design decisions that secondary decisions depend on. These design rules are stable decisions that are not likely to change throughout the project lifecycle. A design rule can describe the decisions on 1) a product's architecture, 2) the interfaces between modules, and 3) the integration tests that will ensure modules are working together. Baldwin and Clark [7] argued that modularity of a system increases the value of that system by providing design options. Modules can be easily swapped for one another when unanticipated changes occur to the design. This supports software evolution.

However, it has been found that designers will rarely choose the optimal module decomposition [54]. There are two well-known measures, coupling [32] and cohesion [124], that help assess the modularity of a design. Coupling considers the number of dependencies between modules. Highly coupled modules have many dependencies between them and will result in more coordination needs between the developers who implement those modules. Cohesion considers the number of dependencies within a module. Higher cohesion means the elements within a module are functionally related. The best designs will have low coupling and high cohesion.

Wong et al. [139] created a Design Rule Hierarchy (DRH) metric to identify modules that can be independently assigned to developers for parallel work. DRHs are computed from Design Structure Matrices (DSMs) [7]. A DSM is a square matrix that identifies technical dependencies between software modules. Consistent with

Parnas' definition of modularization [105], these independent modules can be worked on in parallel without incurring coordination overhead. A DRH clusters modules into "layers" where each layer depends only on the layers above. The layers can be used to differentiate artifacts that represent influential design decisions (design rules) from low-level artifacts that depend on those decisions. Wong et al. [139] established three categories of work that can be used to differentiate between tasks that can be completed independently and those that will require coordination:

1. Same Layer Same Module (SLSM) pairs: Two tasks include edits to artifacts that have a dependency and are in the same module. Tasks that have a SLSM relationship may require coordination.

2. Across Layer (AL) pairs: Two tasks include edits to artifacts that have a dependency and are in different modules and different layers. Tasks that have an AL relationship may require coordination.

3. Same Layer Different Module (SLDM) pairs: Two tasks include edits to artifacts that are in different modules of the same layer. By definition, there are no dependencies between these artifacts, so tasks with only SLDM relationships should be able to be completed independently.

For illustration purposes, Figure 1 shows an example of a hypothetical two-layer DRH. The large thick-bordered boxes represent the two different layers while the boxes within the layers represent modules. The X's show the dependencies between the modules. Tasks 1 and 2 are an example of an SLSM pair since they are operating on the same module. Tasks 2 and 3 are an example of a SLDM pair since they are operating on the same layer but on different modules. Tasks 1 and 3 are an

example of an AL pair since they are operating on modules in different layers with a dependency.

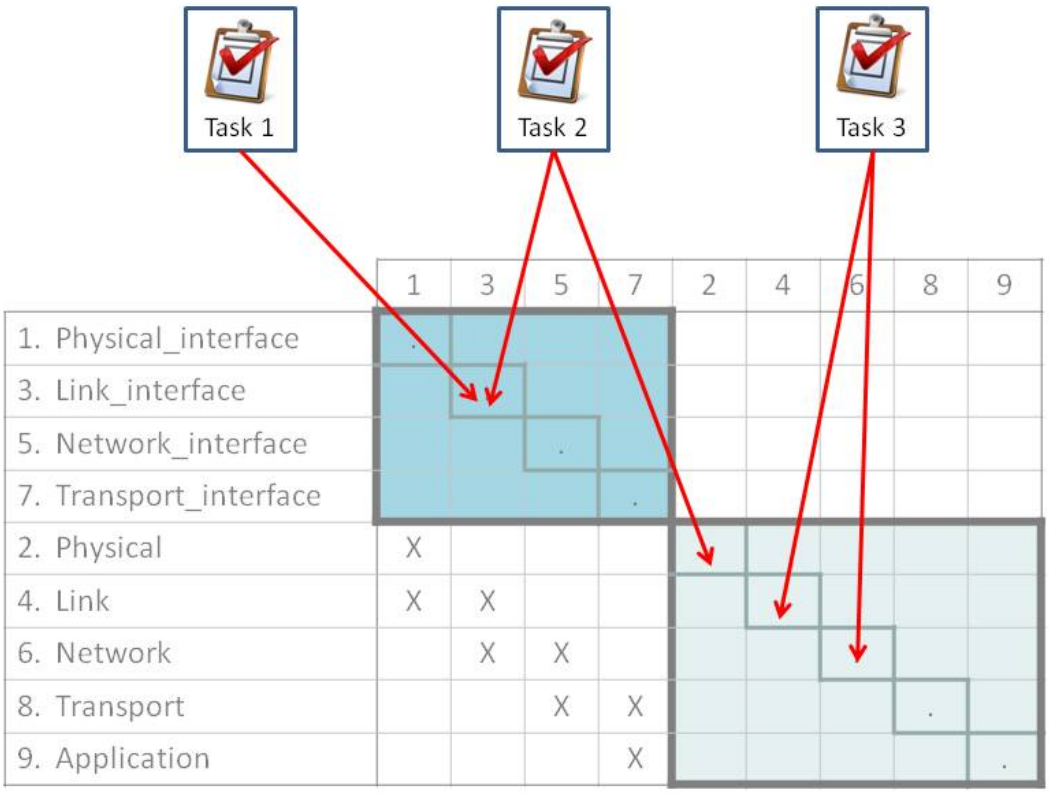


**Figure 1: Design Rule Hierarchy Example [139].**

The SLSM and AL categories can be seen as potential coordination needs since dependencies exist between these task pairs. Wong et al. [139] observed that developers working on tasks that involve either the same software modules (SLSM) or software modules in different layers of a DRH (AL) tend to communicate (a dominant form of coordination in software development [87]) significantly more than developers working only on modules in the same layer (SLDM). Therefore, Wong et

al.'s DRH approach, given an existing software product or design, provides an automatic way to identify modules that can be developed independently and in parallel, without requiring coordination.

However, not all technical dependencies can be eliminated, and not all modules will be able to be developed in parallel without coordination. Schedule optimization algorithms introduced another approach to dealing with dependencies between software modules. Rather than satisfying the technical dependencies through coordination, modules are developed according to a schedule that will reduce the occurrence of conflicts. di Penta et al. [46] found that optimization of project scheduling can reduce coordination overhead through evaluation of their search-based optimization techniques. More recently, Kasi and Sarma [80] introduced a tool, Cassandra, which identifies potential conflicts between tasks based on the files in their workspaces and suggests optimal scheduling to avoid those conflicts. While these schedule optimization approaches can certainly reduce the coordination needs of a development team, they will not be able to fully eliminate the need for coordination. This is particularly true when the schedule is tight and large amounts of work need to be done in parallel, despite the conflicts that may arise.

When more work can be done in parallel, the team's productivity can increase. When technical dependencies exist, work is being done in parallel, and the required coordination does not occur, problems can occur when integrating the modules [43]. Herbsleb et al. [73] argued for computing and fulfilling coordination requirements to satisfy these technical dependencies rather than trying to minimize the dependencies themselves. However, developers are not always aware of their coordination needs.

## 2.3 Awareness

Gutwin et al. [65] argued that team members must maintain awareness of each other to achieve successful collaboration. Awareness is defined as "*an understanding of the activities of others, which provides a context for your own activity* [47]." Awareness is especially important in Software Engineering since Software Engineering is a collaborative effort that is often performed in large, distributed settings. In large and distributed development projects, it is particularly difficult for developers to stay aware [25]. Kiani [84] found that teams are more aware when teams are smaller in size, have more experienced members, use agile processes, and have frequent interactions between teammates both within and across teams. Awareness is especially difficult when teams span organizational boundaries [102].

In face-to-face settings, team members can naturally become aware of each other and each other's work. However, in distributed settings, teams need tools to help support awareness [47]. There have been a slew of awareness tools built to help support Software Engineering teams. Portillo-Rodriquez [107] found awareness features exist in nearly all types of Software Engineering tools including requirement, design, development, configuration management, project planning, process, quality, knowledge management and social tools. Some of the awareness features provided by these tools help developers understand the team structure [2], [113], review project history [59], become aware of ongoing file changes and potential conflicts [12], [21], [22], [38], [45], [55], [56], [64], [70], [104], [111], [112], [118], [119], [137], and understand technical dependencies and the coordination they may require [10], [44], [97], [110], [141].

Social- and team-oriented features in software development tools seem to be effective in supporting teams to overcome challenges of distance and promoting awareness. A study of Jazz, one of the first collaborative software development environments, found that task duration is not as strongly impacted by geographic distance as found in previous studies [103]. Another study [13] found that geographic distance between team members did not significantly affect the number of software faults.

van Gameren et al. [133], [134] have a vision that all awareness information can be automatically analyzed, filtered and combined to provide an overall awareness picture to developers similar to the awareness obtained in face-to-face settings. This is in line with the concept of continuous coordination tools described by Sarma et al. [114]. They envision a future where developers will no longer need to use explicit or separate coordination tools since their virtual work environment will seamlessly combine coordination and work. Redmiles et al. [109] described this as "*flexible work practices supported by tools that continuously adapt their behavior and functionality so coordination problems are minimized in number and impact.*" However, this future vision of seamless awareness will not be achieved quickly and must evolve through incremental improvements in existing tools.

This work focuses on ways to improve awareness of coordination needs in software engineering teams. Herbsleb et al. found that "*development work is faster when those performing mutually constraining work 1) are on the same team, 2) are located at the same site, 3) communicate using an asynchronous text tool, or 4) communicate in a chat room.*" [73]. This insight suggests that by correctly targeting coordination, a software team can see performance benefits. This is in line with the

original intuition by Conway [33] who was the first to recognize that the social structure of the team plays a role in the design of the software product that team is developing. He said, "*any organization that designs a system … will produce a design whose structure is a copy of the organization's communication structure.*" This has since been dubbed Conway's Law. The message behind Conway's Law is that when a dependency exists between two modules, the developers responsible for those modules must coordinate to ensure the modules interface correctly with each other. Recently, Kwan et al. revisited Conway's Law, and they found that aligning software development organizations based on the tasks that the developers are working can provide benefits in the team's ability to work together [89]. Betz et al. [11] describe what they call the "rubber band effect" of Conway's Law, which states that changes in a team's organization structure will eventually trigger changes in the design of their software products.

It has been found that an increase in communication can bring greater awareness of work dependencies and reduce coordination problems [43], [63], [72]. Herbsleb et al. found that when developers are willing to communicate directly, integration problems are reduced [71]. However, even if developers are willing and able to coordinate, they may often be unaware of their coordination needs [44]. This can be complicated by the fact that developers' awareness networks are often fluid and change throughout the course of development [42].

de Souza et al. [44] observed two major types of coordination problems prevalent in software development – a lack of awareness of other team members' work and a difficulty in identifying other software developers with whom it would be important or interesting to communicate. Specifically, they found that managers lack

an awareness of evolving social dependencies within their teams and developers lack an awareness of evolving technical dependencies. Developers have difficulties finding other developers with the required expertise to answer questions or help guide them in their development tasks. They also have difficultly finding developers whose work has similar dependencies, for example those who depend on the same interface or component. Finding such developers can help, for example, in minimizing duplicate work.

According to Fritz and Murphy, software developers are most interested in awareness tools that will help them understand who is working on what and what changes are made to the code base [58]. A recent survey of developers at Microsoft found that, for a software engineer, the most important form of awareness is locating and keeping up to date with other developers whose work is relevant to their own [10]. Another study conducted by Ko et al. [85] found that developers are most interested in awareness about what information is relevant to their tasks, how artifacts changed, and what their co-workers have been doing.

## 2.4 Providing Awareness of Coordination Needs

Two main approaches have been proposed to provide awareness of coordination needs in software development teams: 1) configuration management conflict detection and 2) coordination requirement detection.

### *2.4.1 Conflict Detection*

Configuration management tools have been used for a long time in software development to help coordinate concurrent work between developers. Sarma et al. defined two classes of configuration management tools, pessimistic and optimistic

[111]. Pessimistic tools force developers to lock the files that they are editing to ensure that no other developers make edits to that artifact until the lock is released. This approach does not allow for direct conflicts (concurrent edits to the same artifact), but it does limit the amount of concurrent work that can occur. It also still allows indirect conflicts where one developer makes a change in one artifact that affects another developer's work in a separate artifact. Optimistic tools allow developers to work concurrently on the same artifacts. A merge must be performed when developers are ready to check in their changes into the code repository. Developers remain unaware of who else is working on the same artifacts until they have completed their work. This approach can result in both direct and indirect conflicts. Direct conflicts will be resolved when the merge occurs, but indirect conflicts are not detected.

Gutwin et al. [66] found that distributed software developers obtained awareness through monitoring configuration management system check-in logs. Many configuration management systems send automatic emails with recent changes to subscribed developers. However, the amount of information from check-in logs can be overwhelming and requires a significant time commitment for developers to obtain awareness. In addition to the email notifications, most configuration management tools have commands that allow developers to pull change information for a particular file. For example, Git Blame[3] allows a developer to identify who has made changes to a particular file in the Git[4] version control system. However, this requires the developer to manually run a command whenever they want to obtain awareness. The

[3] Git Blame: http://git-scm.com/docs/git-blame
[4] Git: http://git-scm.com

awareness obtained from the configuration management system is not timely since development work is often complete, or nearly complete, when code is checked-in; thus, it may be too late for developers to coordinate effectively once that information is obtained. Configuration management conflict detection tools were introduced to provide this type of information in a more continuous, timely way.

Palantír is one of the earliest awareness tools in software engineering and an example of a configuration management conflict detection tool. Palantír was developed to help alert developers of possible conflicts by letting them know which other developers are making changes to the files which they are currently modifying. It changed the flow of information from "pull", where developers only receive information when they perform certain interactions with the configuration management system, to "push" where information is made continuously available to developers regardless of their actions [111].

Palantír uses notifications to keep a developer abreast with what happens in her colleagues' workspaces [111], [112]. Palantír detects conflicts by making use of information from the configuration management system. It looks at the artifacts in each developer's workspace and their state, and it compares them to the state of the "master copy" for the same artifacts maintained in the configuration management repository. It notifies developers of changes occurring to the artifacts they have in their own workspace. While these notifications are timely, they only regard direct conflicts on the same artifact and a very specific type of indirect conflicts that occur when class signatures are conflicting. These are a narrow subset of the many technical reasons that can induce a coordination need.

CollabVS is another awareness system that uses notifications. Compared to Palantír, it has an expanded model of interest. The CollabVS model captures additional conflicts by considering a subset of syntactical dependencies between artifacts [45]. It issues instantaneous warnings to developers as an individual instance of conflict emerges, but it does not offer a model for quantifying the strength of a coordination need. Even with its expanded model of interest, it does not capture all coordination needs. So while they are timely, CollabVS and Palantír provide an incomplete view of coordination needs, which are not prioritized, quantified or filtered in any way. Tukan [118] provides similar conflict support, but it presents information only at certain intervals rather than instantaneously, reducing the timeliness of the awareness it provides.

CollabVS and Palantír provide a stream of notifications regarding each potential conflict at the source code level. This approach is likely to cause information overload [53], [75], [101], [127] for developers, especially since any concurrent modification to the same artifact will generate a notification regardless of complexity. This brings about inefficiency since the developers are required to sift through a large number of notifications to determine which conflicts really matter. Augur [59] is another tool built on top of configuration management systems that visualizes potential conflicts for each line of code in a software artifact. The level of detail that Augur provides does not scale to large projects where it is also likely to cause information overload.

Syde [70] introduced a way to reduce this information overload by reducing false positives through abstract syntax tree (AST) modification analysis. This results in only flagging potential conflicts when the two developers are working on the same

area of code within a large file. However, Syde is limited because it only considers direct conflicts. FASTDash [12], Lighthouse [38], Celine [55], Elvin [56], War Room Command Console [104], and CASI [119] are other configuration management conflict detection tools that provide support for only direct conflicts.

Crystal [21], [22] identifies conflicts in decentralized version control systems like Git. It also finds far fewer false positives than tools like CollabVS and Palantír since it waits for code to be committed to a local repository, but it is not timely. It examines commits made in a developer's local repository and integrates those local changes into a shadow repository of the main branch. It runs the build and test scripts to identify any problems that would occur when merging a developer's existing local changes to the main branch. Therefore, Crystal does not report conflicts until changes have already been committed to some repository reducing the timeliness of detection.

WeCode [64] and Safe-Commit [137] employ similar methods in a more timely way. WeCode continuously merges all developers' uncommitted code into a shadow copy to identify any merge conflicts. Safe-Commit runs test scripts to detect conflicts without waiting for local commits. It looks at the changes in a developer's local workspace and identifies changes that pass the project's existing tests. The goal of Safe-Commit is to identify changes that can be checked in early with the belief that frequent check-ins can decrease duplicate work and decrease merge conflicts. Crystal, WeCode and Safe-Commit are still unable to provide a complete view of coordination needs since they rely on test cases to detect conflicts. Conflicts that are not covered by any test case will not be detected. Therefore, their solutions rely on the quality of the test suite of the development team.

### *2.4.2 Conceptualization of Coordination Requirements and Socio-Technical Congruence*

Cataldo et al. [27], [28], [30] introduced a framework to detect and quantify a more complete view of Coordination Requirements between pairs of software developers. They do this by identifying the technical dependencies between software artifacts modified during their assigned tasks. They compute those technical dependencies through logical coupling [60], which tracks files that have been historically checked in together and aims at identifying semantic relationships that may not manifest in the syntax of the programmatic implementation of the software product. While syntactic dependencies can be identified prior to implementation, logical couplings reflect accumulated empirical evidence about how the development work unfolds in the project. Cataldo et al. offer empirical evidence that logical coupling provides a more reliable representation of the technical dependencies relevant for coordination requirement detection than syntactic coupling does [29]. Once technical dependencies between artifacts have been established, they compute coordination requirements using the following formula:

$$CR = TA \times TD \times TA^t$$

In this formula [27], [28], [30], TA is a people-by-task matrix representing task assignments, and $TA^t$ is its transpose. TD is a task-by-task matrix capturing the work dependencies between tasks. Those are established by considering the technical dependencies occurring between artifacts involved in those tasks. CR is the resulting matrix of coordination requirements. According to this formula, a coordination requirement between two developers, Alice and Bob, can be represented graphically as in Figure 2. Arc $TD_{ab}$ represents a technical dependency between software artifacts

$S_a$ and $S_b$. These artifacts are involved in tasks to which Alice and Bob, respectively, are assigned (denoted by arcs $TA_a$, $TA_b$). Empirical evidence suggests that when coordination activities focus on the identified coordination requirements, productivity is likely to improve [27], [28], [30].

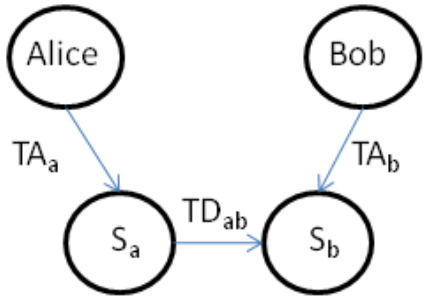


**Figure 2: Representation of a Coordination Requirement.**

The conceptualization of coordination requirements led to the concept of Socio-Technical Congruence (STC) [27], [28], [30], which states that when coordination is focused between the team members with identified coordination requirements the project can see performance benefits. STC measures the extent to which coordination needs and coordination behavior are aligned in practice. STC is expressed as a simple ratio between coordination requirements that are satisfied by actual acts of coordination (fulfilled coordination requirements) and the set of remaining coordination requirements between developer pairs that are unfulfilled (coordination gaps). Recent research has found that having low congruence and many coordination gaps can significantly increase the number of software failures in mature development settings and in new and dynamic settings alike [27].

The method proposed by Cataldo et al. identifies coordination gaps. If developers could be made aware of those gaps in a timely way, they could take action to fulfill those gaps therefore increasing software productivity and quality. Since developers are often limited in the amount of time they can spend coordinating their work, a way to prioritize the list of coordination gaps could be useful. Valetto et al. [131] introduced an alternative graph-based algorithm for detecting coordination requirements. The graph contains dependencies between software artifacts, the connections between the software developers and those artifacts, and the interactions that have occurred between the developers. They created an algorithm for analyzing the graph to rank the coordination requirements. Since the graph contains information on which developer pairs have already engaged in coordination, the gaps can easily be highlighted and ranked, helping to focus coordination efforts where it is most needed. Tesseract [110] is an awareness tool that similarly highlights coordination gaps by considering the developers with evidence of prior coordination.

However, there are several problems with the way coordination gaps are identified. Coordination requirements are counted as fulfilled by any single act of coordination. Developers may only have coordinated on a subset of the technical dependencies that contribute towards a coordination requirement. In those cases, coordination requirements that are considered fulfilled may actually still indicate a coordination gap regarding other technical dependencies. Kwan et al. [91] proposed an enhanced weighted communication model which counts the number of communications that occur and the content of those communications to better understand which technical dependencies have been fulfilled. Wolf et al. [138] introduced an approach for mining large software repositories to identify task-based

communication between developers. However, communications that are not directly linked to a task cannot always be easily associated to a particular task. Another problem with the conceptualization of coordination gaps is that the absence of direct communication between a pair of developers does not always indicate a gap. Ehrlich et al. [51] first introduced the idea of coordination brokers in a software development team. Brokers are people who act as an intermediary between two developers or groups of developers to facilitate coordination. They suggested that brokers may be able to mitigate the effect of gaps. Kwan and Damian [90] later introduced a method to extend the conceptualization of coordination requirements to account for brokers.

These coordination requirement detection methods and the STC measure highlight the importance of coordination in software teams. However, these methods are retrospective and, therefore, not timely. Coordination requirements are identified by examining the artifact commits made by developers in the project's source control repository. Commit data is typically available only after the majority of development work for a task has been completed. In addition, logical coupling is used to determine technical dependencies between artifacts. Computing dependencies in this way is based on past project history which is only visible after much work is completed. Even when syntactic dependencies are chosen, as done by Ehrlich et al. [51], they only become fully known following a commit since dependencies between artifacts can change throughout the development process. This lack of timeliness limits the potential of coordination requirements as a means to support coordination as the development work unfolds.

In addition, a recent study has shown that in large projects, even when coordination requirements are computed simply between pairs of developers, the

number of potential coordination requirements that are listed for a given developer may be very large [40]. Cataldo and Ehrlich [26] also found that current collaborative tools may be sufficient for small teams, but they risk introducing information overload when used in larger teams.

Identifying coordination needs between pairs of developers may also introduce inefficiencies. Since developers often work on multiple tasks in parallel, coordination requirements at the developer level may encompass the work dependencies of many tasks. This puts the burden on the developers to identify which tasks require coordination. This can increase coordination overhead and reduce efficiency.

### *2.4.3 Applications of Coordination Requirements*

There are many tools that try to achieve awareness by employing abstractions similar to, or derived from, the concept of coordination requirements:

Ariadne [44] pulls data from the configuration management repository and uses Cataldo et al.'s algorithm to detect coordination requirements between developers. It uses those coordination requirements to create visualizations of socio-technical networks including a visualization to alert developers of their coordination needs, a visualization to allow management of a team's overall coordination needs, and a visualization to show developers that have experience on a given code module.

EEL [97] also pulls data from the configuration management repository and uses Cataldo et al.'s algorithm to detect coordination requirements between developers. EEL uses these coordination requirements to display a suggested buddy list for each developer, that is, a ranked list of other developers with dependencies – and expertise - on the users' current change set. Ensemble [141] provides a similar

suggested buddy list, but filters its recommendations based on the coordination gaps that exist by considering project communication records.

Tesseract [110] uses Cataldo et al.'s coordination requirement detection algorithm to graphically display coordination requirements in a dashboard. It pulls information from various sources, such as the project's configuration management system, mailing lists and bug tracking system, and it shows the relationships between developers and the various software artifacts. Like, Ensemble, Tesseract also uses information obtained from project communication records to highlight fulfilled coordination requirements as well as gaps. This allows developers and/or managers to better focus their coordination efforts, but the accuracy of this feature is limited by the amount of communication that can be automatically captured. Tesseract and Ensemble do not process the content of communications; they simply mark a coordination requirement as fulfilled by any single act of communication between a pair of developers. This can be especially troublesome when a coordination requirement is comprised of multiple technical dependencies.

Codebook [10] is a graph-based framework for determining coordination requirements between developers. Codebook mines data from many different repositories including the configuration management system, email system, bug tracking system and employee directory using crawlers designed for each repository. A directed graph is then created to capture the relationships between people, code, tasks, requirement specifications and other work artifacts. Since Codebook contains far more data than is available in just the configuration management system, it can provide a much richer set of information than that included in Cataldo et al.'s coordination requirement algorithm.

Since all of these tools identify coordination requirements between pairs of developers by mining commit data from the configuration management system, they all suffer from a lack of timeliness and efficiency.

# CHAPTER 3: RESEARCH QUESTIONS, SETTING, AND METHODOLOGY

This Chapter introduces our research questions, describes the setting of our case studies, and provides details of the research methods used while addressing each research question.

## 3.1 Research Questions

This dissertation presents a set of methods and techniques that address the two main limitations of existing coordination requirement detection methods: lack of timeliness and inefficient recommendations. The development of these techniques has been guided by three main research questions:

*RQ1: Is timely coordination requirement detection possible?*

*RQ2: Can coordination requirements be identified efficiently at the task level of granularity?*

*RQ3: Are the more critical coordination needs actionable?*

## 3.2 Research Setting

To answer these research questions, a series of investigations was performed using data from the Mylyn[5] open source project. Mylyn is an Eclipse[6] plug-in that is now bundled in the main Eclipse distributions. It transforms an individual software developer's Integrated Development Environment (IDE) into a task-centric view to make context switching between tasks easier. To fulfill its own purposes, Mylyn

---

[5] Mylyn: http://www.eclipse.org/mylyn/
[6] Eclipse: http://www.eclipse.org

records all developer interactions within the IDE as they occur. These events are stored as context data for the task in focus. We used this context data to solve the lack of timeliness of existing approaches since it provides a record of developers' activities as they occur. While there are other tools, such as Cubeon[7], which provide IDE logging, Mylyn is the most well-known and widely used tool. Tasktop Technologies[8] created and leads the Mylyn open source project. Tasktop also has an enterprise version of the Mylyn open source project, called Tasktop Dev, which is available as a plugin for Eclipse and Visual Studio and as a standalone application.

The developers involved in the Mylyn open source project make routine use of the Mylyn plugin in their IDE and attach their Mylyn context data, which details developer activities, to each change request. There are several types of actions captured in the Mylyn context data. For this study, we consider only artifact selection (consultation) and edit actions. Other actions used within Mylyn, such as prediction, propagation and manipulation, were purposely discarded. Manipulation actions represent information that developers can explicitly provide to Mylyn to emphasize the importance (or lack thereof) of a given artifact for the task at hand. Prediction and propagation events occur when Mylyn itself "suggests" other artifacts, which are not included in a developer's working set, but appear to be structurally relevant. Since these event types are specific to Mylyn, including them would make the replication of our experiments and findings outside of the Mylyn framework difficult (e.g. in projects and environment that employ different IDE logging facilities such as

---

[7] Cubeon: http://code.google.com/p/cubeon/
[8] Tasktop: http://tasktop.com/dev

Cubeon). In the remainder, Mylyn context data refers, therefore, only to the artifact consultation and edit activities captured within that data.

We mined the project repositories including the change request repository, Bugzilla[9], and the configuration management system, CVS[10]. We collected all Bugzilla change requests and developer activities (Mylyn context data) from eight releases of the Mylyn project, releases 2.0 to 3.3, which spanned nearly three years of development. As shown in Table 1, each release involved two to nine months of development. We included all Bugzilla change requests for which development work occurred during the release's development period. We determined the time of development for a change request by the artifact selection and edit activity obtained through the Mylyn context data attached to the Bugzilla record.

**Table 1 Mylyn Releases.**

| ***Release*** | ***Start*** | ***End*** |
|---|---|---|
| 2.0 | December 2006 | June 2007 |
| 2.1 | June 2007 | September 2007 |
| 2.2 | September 2007 | December 2007 |
| 2.3 | December 2007 | February 2008 |
| 3.0 | February 2008 | June 2008 |
| 3.1 | June 2008 | March 2009 |
| 3.2 | March 2009 | June 2009 |
| 3.3 | June 2009 | October 2009 |

On the Mylyn project, developers are assigned change requests as their unit of work and encouraged to deliver their work as code patches that correspond to (and

[9] Bugzilla: http://www.bugzilla.org
[10] CVS: http://www.nongnu.org/cvs/

resolve) a single change request. The bug-tracking database is the way the Mylyn team defines and assigns developer tasks. Therefore, we refer to Bugzilla change requests as tasks.

To better understand the coordination problems of the Mylyn team, we interviewed six developers from the Mylyn open source project and Tasktop, the enterprise version of Mylyn. The developers told us that they coordinate by ensuring all task details are documented on the task report in Bugzilla. They use the "cc" feature of Bugzilla to alert another developer of a task. They use the chat feature of Skype and also hold Skype video or audio calls when working with distributed team members. They believe that their code is sufficiently modular, and they strive for small tasks that affect only one or two modules to reduce coordination needs. The Mylyn team has 4 core developers and up to 10 other contributors at any time. The Tasktop team is comprised of approximately 30 developers.

While they are a relatively small, well-established team, Mylyn developers still experience occasional problems stemming from a lack of coordination. We asked the interviewees: "*Do you recall any problems due to lack of coordination as you completed your development tasks?*" The developers all stated that they had experienced coordination problems during development. The most common issue discussed was duplication of work caused by a lack of awareness of what others are working on. Other coordination problems discussed include:

- Developers are unaware of how their task affects other tasks.
- Developers are unaware of how other tasks affect their own tasks.
- Developers incorrectly assume someone else is handling a task, and the task is left unmanaged.

These are in line with the ‘questions developers commonly ask’ that were identified by Fritz and Murphy [58]. The developers’ remarks are all related to a lack of awareness of what others are working on, in relation to their own work, and are consistent with the hypothesis that inspired our research.

One developer noted that, although they do not experience a large number of coordination problems, “*when they do happen, they can be expensive.*” The interviewees noted that they currently use tests and source code management tools as their primary way to deal with coordination problems when they do occur. Both of these methods handle problems only after the problem has already been introduced in the code base. Efficient upfront coordination or awareness of coordination needs could reduce the time spent fixing problems and resolving conflicts. In this dissertation, we present ProximityML, an approach that allows for timely detection of the more critical coordination needs between pairs of tasks.

### 3.3 Research Methods

*RQ1: Is timely coordination requirement detection possible?* We developed a new method and metric, called Proximity, which detects coordination needs between pairs of developers in a timely way. It is timely because it computes coordination requirements using data obtained through IDE monitoring tools, like Mylyn, which capture developer actions as they occur. We evaluate the accuracy and timeliness of the Proximity method by comparing against the Cataldo et al. coordination requirement detection method, the most well-known existing method. Chapter 4 describes the Proximity metric in detail and shows how it is timelier than existing methods and can be more accurate in identifying actual coordination requirements.

*RQ2: Can coordination requirements be identified efficiently at the task level of granularity?* We adjusted Proximity to identify coordination requirements between pairs of tasks. However, computing coordination needs at this level produces a large number of potential coordination needs. We identified an approach, ProximityML, which identifies the more critical coordination needs to allow for more efficient coordination. The accuracy of the ProximityML results was evaluated relative to the coordination needs experienced by the team, obtained from a thorough examination of task records. The criticality of the ProximityML results was evaluated by considering the measures of task complexity (change size) and task performance (task duration). Chapter 5 describes our ProximityML approach and shows how it is able to detect a set of the more critical coordination needs.

*RQ3: Are the more critical coordination needs actionable?* We analyzed whether the ProximityML approach allows for timely detection of coordination needs as they emerge. We streamed each event over the life of each task (developer actions and Bugzilla task updates) in a time-ordered sequence and re-ran the ProximityML approach after each event. This allowed us to evaluate exactly when ProximityML first recognizes a coordination need. Chapter 6 describes this exercise and evaluates the consistency of the results over the duration of the release as well as the timeliness of the detected coordination needs. It also addresses the usability and actionability of the coordination recommendations made by our approach through developer interviews.

# CHAPTER 4: TIMELY COORDINATION REQUIREMENT DETECTION

This chapter addresses our first research question: *RQ1: Is timely coordination requirement detection possible?* We address this research question by introducing a new method for computing coordination needs between software developers, Proximity. Proximity is a quantitative measure. It outputs a score for each pair of developers indicating the strength of their coordination need. Higher Proximity scores denote stronger coordination needs. We evaluate the accuracy and timeliness of Proximity scores in relation to the coordination requirements established using the best-known existing approach, Cataldo et al.'s method [27], [28], [30]. We evaluate the coordination needs over eight releases of the Mylyn project. This investigation has been published in the Proceedings of the Conference on Computer Supported Cooperative Work and presented at that conference [16].

## 4.1 Approach

### *4.1.1 Proximity Method*

Proximity is a method and metric that detects coordination needs between pairs of developers. It outputs a score for each pair of developers indicating the strength of their coordination need. Higher Proximity scores denote stronger coordination needs.

Proximity computes coordination requirements by monitoring the actions developers take in their IDE as they occur, using the data obtained from the Mylyn framework [82], [83]. The captured actions can be very granular and, most

importantly, are collected while the developers work. That can make Proximity timelier than other methods and turn coordination requirements into an actionable concept for managing coordination while development is underway. The Proximity measure looks at artifact consultation and modification activities and weighs the overlap that exists between the working sets associated to pairs of developers. It considers all actions recorded for each artifact in each working set in order to apply a numeric weight to that artifact's Proximity contribution. Weights are applied based on the type of overlap and are based on the weights Mylyn uses for its degree-of-interest (DOI) model [82], [83]. Mylyn's DOI model prioritizes the presentation of artifacts in its task-based interface, and its weighting system has been empirically validated. The most weight is given when an artifact is edited in both working sets (weight = 1) and the least amount of weight is given when an artifact is simply consulted in both working sets (weight = 0.59). When an artifact is edited in one working set and consulted in the other working set, we consider this a mixed overlap (weight = 0.79).

Figure 3 illustrates an example Proximity computation. The algorithm computes the ratio of actual to potential overlap. Actual overlap is calculated as the intersection of the two working sets. Potential overlap represents the maximum possible Proximity score had there been perfect overlap between the two sets of actions and is calculated as the union of the two working sets. Proximity scores can be scaled based on the number of overlapping events to place greater weight on complex tasks that are likely to require coordination. Proximity scores range from zero to infinity where a score > 0 indicates a coordination need. Higher Proximity scores denote stronger coordination needs.

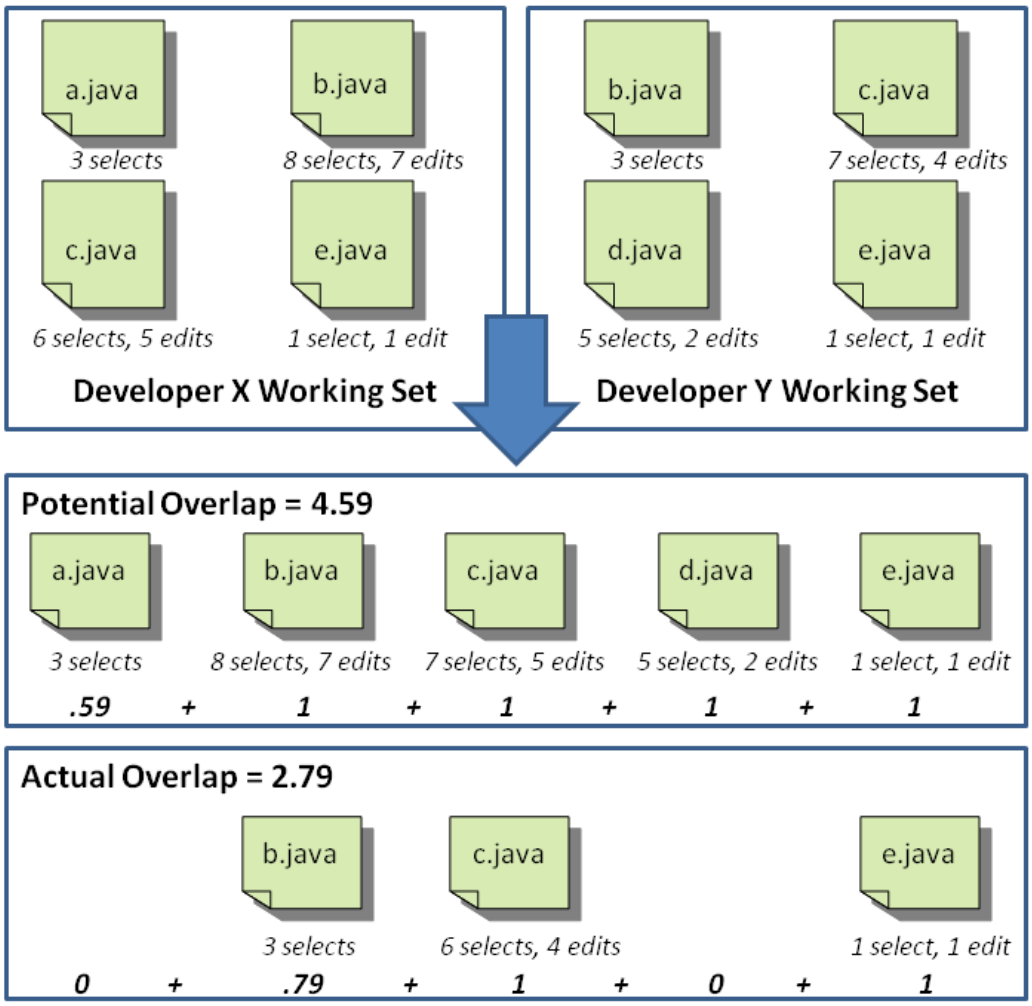


**Figure 3: Proximity Algorithm Example.**

### *4.1.2 ProxiScientia Tool*

Proximity is supported by a tool, ProxiScientia [19], which provides a visualization of coordination requirements in software teams. It was developed in collaboration between Drexel University and University of Victoria. The tool was developed as a plugin for IBM's Jazz development environment, and it has a client/server architecture. The server component has a shared central database for each development team. The client component is built on top of Mylyn and is hosted

within the developer's IDE. It automatically stores Mylyn context information for each developer and pushes the events to the server as they occur. The Proximity calculation is then performed on the server. When new Proximity relationships are detected, they are pushed back to the client for display in each developer's IDE. This allows Proximity relationships to be computed and continuously updated as development is underway with no effort on the part of the developers. The tool demonstrates the feasibility of such a method in detecting coordination requirements using data similar to that collected from the Mylyn context events.

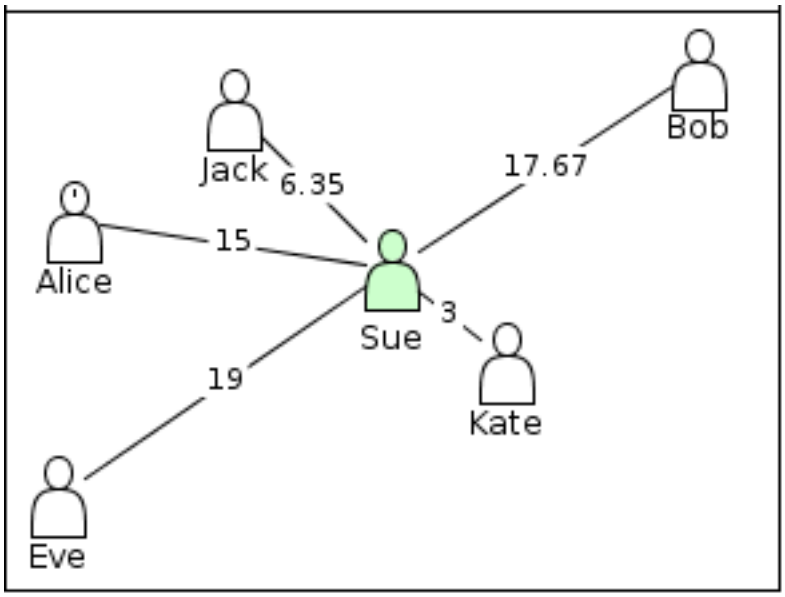


**Figure 4: ProxiScientia Visualization Example.**

ProxiScientia provides a developer-centric visualization of coordination requirements. Figure 4 shows a sample visualization generated for the developer Sue (highlighted in green). The edges denote the reciprocal values of the Proximity scores

for a more intuitive visualization. This allows the highest Proximity scores (strongest coordination needs) to appear closest to the developer in focus. The tool visualizes only the strongest coordination needs. The default configuration, which is customizable, displays Proximity scores that are greater than two standard deviations from the mean. Developers for which there is no coordination need or coordination needs that do not meet the threshold are left out of the visualization to minimize the amount of information the developer must process.

## 4.2 Evaluation Methodology

To answer our first research question – *RQ1: Is timely coordination requirement detection possible?* – we evaluated the coordination needs on eight releases of the Mylyn project. We evaluated the accuracy and timeliness of our Proximity method. Proximity scores were evaluated relative to the coordination requirements established using Cataldo et al.'s method [27], [28], [30]. The Cataldo et al. method was selected for comparison since it is the most well-known method for detecting coordination requirements, and many of the awareness tools created to detect coordination requirements are based on this method.

## 4.3 Analysis and Results

### *4.3.1 Description of Data Set*

For our evaluation, data was collected from the development of the eight releases of the Mylyn project. Data was gathered for all tasks in those releases that had Mylyn context data (attached to the task). There were 1,970 tasks in this data set. There were 51 distinct developers who attached Mylyn context data to these tasks (context attachers). The data we collected was separated into three data sets.

**Data Set 1 (DS1):** Commit data is required for computing the Cataldo et al. coordination requirements, so all commit data was collected. Over all eight releases, there were 8 distinct developers who committed code (committers). In our data set, 92.8% of all commits are associated with a particular task through an explicit link included by committers in their commit comment. There are 1,127 tasks which have both context data attached and associated commit records. This set includes 10,647 artifact commits and 450,757 context events related to Java source code artifacts.

We found that commits are not always matched by any proof of editing of the involved file in the associated Mylyn context data by the developer who committed the change. There are two possible reasons for this misalignment of activity: (1) the developer who committed the change did not attach their Mylyn context data to the task, or (2) the developer who committed the change was not the developer who contributed it. Since commit rights are often limited to a small set of developers on open source projects, there are typically many developers who contribute code without commit privileges. These developers submit their code contributions for another developer with commit rights to commit to the code base. The developer who contributes the code also attaches Mylyn context data to the task.

We split DS1 in two: **DS1-a** includes the 4,140 commits for which we have matching events within the Mylyn task context data; **DS1-b** includes the other 6,507 commits. DS1-a, therefore, is the set of commits that were both contributed by and committed by the same developer. DS1-b provides a less homogenous data set since the Mylyn context events do not align with the artifacts that were committed for the associated developers.

**Data Set 2 (DS2):** In each release in our data set, there were 4 to 6 committers, but 10 to 32 context attachers. While Proximity allows coordination needs to be calculated between the actual code contributors, the Cataldo et al. approach can only detect coordination needs between those developers who have committed code. Therefore, using commit data, we are only able to compare the coordination requirements for a small set of developers. To expand our evaluation, we compiled an additional data set by considering patch descriptions (attached to the task). Patch descriptions are semantically equivalent to commits: they report the diff information for all artifacts that were modified as part of a patch. They are attached to tasks by the developers who contribute code but do not have commit privileges. We use these patch descriptions to compute Cataldo et al. coordination requirements as a proxy for commit data for those developers without commit access. There are 936 tasks with attached patch description files. Those tasks have 345,521 associated consultation and edit context events related to java source code artifacts. There are 1,387 file changes detailed in those patch description files that are matched by proof of editing of the same file in the Mylyn context data for these tasks. Thirty-four developers contributed these patch description files.

DS1 and DS2 are disjoint since there is only a single developer common to both sets. This ensures there are no overlapping pairs of developers between the two sets. Therefore, DS1 and DS2 represent complementary analyses over the full picture of the project activity. The 1,387 patch file changes in DS2 represent substantially different development work from what is captured in DS1. The one common developer is responsible for 219 changes in DS2, and a manual inspection revealed that only 11 of the 219 changes overlap with commits made by that developer in DS1.

**Data Set 3 (DS3):** Finally, we combined DS1-a and DS2 into a third data set DS3, which incorporates all records of file changes (either via commit traces or patch diff files) and all context events. Our three data sets are summarized in Table 2.

**Table 2 Summary of RQ1 Data Sets.**

| *Data Set* | *Actors* | *Artifact Info* | *Context Events* | *Commit/Patch Matches Context Data* |
|---|---|---|---|---|
| DS1 a | 8 Committers | 4,140 commits | 450,757 | YES |
| DS1 b | | 6,507 commits | 450,757 | NO |
| DS2 | 34 Contributors | 1,387 edits | 345,521 | YES |
| DS3 | DS1-a and DS2 combined | | | YES |

### *4.3.2 Accuracy of Proximity Scores*

Proximity scores and coordination requirements detected by the algorithm proposed by Cataldo et al. [30] were calculated for each pair of developers in each release. Work in each release was analyzed separately, since releases are a logical unit of concurrency for tasks in an open source project. To evaluate the accuracy of our Proximity scores, we (1) computed correlations between the Proximity scores and the Cataldo et al. coordination requirements, (2) ran a regression model, (3) computed precision and recall against the Cataldo et al. coordination requirements, and (4) manually evaluated the cases where Proximity scores do not align with Cataldo et al. coordination requirements.

#### *4.3.2.1 Correlations*

For each data set, two correlation tests were performed: (1) a point-biserial correlation with Proximity scores and a binary vector denoting the presence of a Cataldo et al. coordination requirement and (2) a Spearman correlation between the count of Cataldo et al. coordination requirements for each developer pair and the Proximity scores. We used a Spearman correlation because both the Cataldo et al. coordination requirement counts and Proximity scores are not normally distributed and strongly skewed.

The Mylyn context events used for the Proximity calculation provide more granular information than is available from commit data. The Mylyn context data identifies the file name, class name and even the name of the class element (method or attribute). This allows Proximity to determine coordination needs more granularly, for example, to see whether two developers were working on the same area of code within a large file. This is not possible when looking only at commit information unless diff information for each commit is processed and analyzed, which is not done in existing coordination requirement detection techniques. We, therefore, ran the two correlation tests at two different units of work: (1) File and (2) Granular. At the file level, we computed Proximity scores considering only the file associated with each Mylyn context event since Cataldo et al.'s method calculates coordination requirements at the file level. At the granular level, we computed Proximity at the lowest granularity level for artifacts reported in Mylyn context events. The Cataldo et al. coordination requirements were still calculated at the file level since that method does not consider a more granular calculation.

In all data sets, higher values of proximity correlate with the likelihood of a Cataldo et al. coordination requirement (Point-biserial test) and with the count of Cataldo et al. coordination requirements (Spearman test) at both units of work, as shown in Table 3. In most cases, the granular tests have slightly lower levels of correlation. This is in line with expectations since we are comparing coordination requirements calculated using slightly different data between the two approaches for the granular tests.

**Table 3 Proximity vs. Cataldo et al. Correlations**

| *Data Set* | *Test* | *Unit of Work* | *Rho* |
|---|---|---|---|
| DS1-a | Spearman | File | 0.69** |
| | Point-biserial | File | 0.55** |
| | Spearman | Granular | 0.62** |
| | Point-biserial | Granular | 0.49** |
| DS1-b | Spearman | File | 0.60** |
| | Point-biserial | File | 0.59** |
| | Spearman | Granular | 0.54** |
| | Point-biserial | Granular | 0.55** |
| DS2 | Spearman | File | 0.55** |
| | Point-biserial | File | 0.54** |
| | Spearman | Granular | 0.57** |
| | Point-biserial | Granular | 0.55** |
| DS3 | Spearman | File | 0.68** |
| | Point-biserial | File | 0.66** |
| | Spearman | Granular | 0.68** |
| | Point-biserial | Granular | 0.66** |

(* $p < 0.01$, ** $p < 0.001$)

#### *4.3.2.2 Regression Model*

Using our largest data set, DS3, we further investigated the relationship between Cataldo et al. coordination requirements and Proximity scores by means of a

regression model. We employed a zero-inflated negative binomial regression (zinb) since the Cataldo et al. coordination requirement count is highly skewed and presents many zeroes (264 out of 347 developer pairs have no coordination requirements). The zinb model is statistically significant ($\chi^2$=161.69, df=2, $p < 0.001$). Results from the regression are shown in Table 4 for both the count and the excess zeroes portions of the model (white and grey rows, respectively). In particular, a one-unit increase in Proximity (a large increase in the Proximity scale for this data set) causes a 2.20-times increase in the log of expected Cataldo et al. coordination requirement count. That is an expected ~9-times increase in Cataldo et al. coordination requirements for each one-unit increase in proximity.

**Table 4 ZINB Regression: Proximity vs. Cataldo et al. Correlations**

| | ***Estimate*** | ***Std. Error*** | ***Z*** |
|---|---|---|---|
| (Intercept) | 5.22 | 0.37 | 14.17** |
| Proximity | 2.20 | 0.51 | 4.33** |
| Log (theta) | -2.01 | 0.14 | -14.53** |
| (Intercept) | 2.32 | 0.30 | 7.61** |
| Proximity | -106.49 | 33.18 | -3.21* |

(* $p < 0.01$, ** $p < 0.001$)

The edit events contained within the Mylyn context data can be viewed as a super-set to the data available from commits. They are a super-set since not all artifact edits result in a commit. The consultation events are not as closely related to the commit data. Developers are likely to consult, but not edit, many files as part of their development work. Since this activity cannot be detected from commit data, the inclusion of consultation events is a difference in the Proximity method compared to

other existing methods. To investigate the influence of the consultation events in defining our Proximity metric, we recomputed Proximity including only edit event overlaps. We then ran the same zinb regression and obtained a new model. This new model captures only direct edit conflicts and is, therefore, similar to what could be observed with conflict detection tools, such as Palantír. That model is still statistically significant ($\chi^2$=157.17, df=2, $p < 0.001$). We then compared the AIC scores of the new model and our original model. We found that our original model has considerably better support since it has a lower AIC. The difference in the AIC scores is 4.51. That means that our original Proximity model, which includes consultation events, is almost 10 times as likely as the edit event only model to minimize information loss. This indicates that the consultation event information included in Mylyn context data provides valuable insight for the detection of coordination needs.

#### *4.3.2.3 Precision/Recall*

We computed precision and recall, comparing Proximity scores against the Cataldo et al. coordination requirements. We observed high levels of precision and recall for each data set (Table 5).

**Table 5 Proximity vs. Cataldo et al. Precision/Recall (Granular Unit of Work)**

| *Data Set* | *Number of Pairs* | *Precision* | *Recall* |
|---|---|---|---|
| DS1-a | 70 | 42/58 = 0.72 | 42/46 = 0.91 |
| DS1-b | 75 | 33/61 = 0.54 | 33/33 = 1 |
| DS2 | 277 | 24/40 = 0.6 | 24/37 = 0.65 |
| DS3 | 347 | 70/100 = 0.7 | 70/97 = 0.72 |

#### *4.3.2.2 Examination of False Positives/Negatives*

We thoroughly examined the cases for DS1-a where Proximity scores and the Cataldo et al. coordination requirements do not align and found that Proximity can be even more accurate than the Cataldo et al. baseline. In the case of the 16 potential false negatives (Proximity score > 0 but no Cataldo et al. coordination requirement), 15 are missed by the Cataldo et al. approach simply because work by one or both of the developers was never committed to the code base. However, context events prove that those developers were, for some time, engaged in development on the very same artifacts - the epitome of a coordination requirement. The other potential false negative is missed by the Cataldo et al. method since that method does not know which artifacts are consulted by a developer while completing a task. In that case, involving developers 3 and 7 during release 3.3, proximity contributions came exclusively by selection and mixed overlaps. The pair had seven mixed overlaps and six selection overlaps. Meaning that developers 3 and 7 viewed 13 of the same artifacts, of which seven were edited at some point by either developer 3 or developer 7, but no single artifact was edited by both developer 3 and developer 7. Since there were no overlapping commits, the Cataldo et al. method does not allow for a coordination requirement to be detected. However, our algorithm picks up what is likely to be an actual work dependency. Developer 3 and developer 7 repeatedly examined the same area of the software code base and consulted each other's code during their work for release 3.3.

In the case of the 4 potential false positives (Cataldo et al. coordination requirement but Proximity score = 0), the Cataldo et al. method identified a coordination requirement due to a technical dependency between two semantically

unrelated tasks because they involved files that had been historically changed together by other developers often enough to cause a logical dependency to be established. For example, a coordination requirement is established between developers 6 and 7 in release 3.2 using the Cataldo et al. method. Developer 6 committed BugzillaClient.java, while developer 7 committed BugzillaTaskEditorPage.java. The changes by developer 6 involve a character encoding method that is private to the BugzillaClient class. Developer 7 added a new section to the Mylyn task editor. Although we could ascertain those changes were semantically unrelated, the two involved files had been historically changed together by other developers often enough to cause a logical dependency to be established by the Cataldo et al. detection algorithm. We noticed analogous incidents in the other three cases in DS1-a. Those coordination requirements are therefore false positives of the traditional method that our Proximity algorithm correctly eliminates.

All cases examined in DS1-a turned out to be false positives or negatives of the traditional coordination requirement detection method. More importantly, they highlight drawbacks of that method's reliance on post-mortem information and dependency conceptualizations. We conclude that Proximity can be more accurate than existing methods.

### *4.3.3 Timeliness of Proximity Scores*

To evaluate the timeliness of our Proximity method, we compared the time Proximity scores >0 appear against the time when Cataldo et al.'s method identifies a coordination requirement. For this analysis, we used the two data sets for which we have task context data associated with file changes (DS1-a and DS2). We considered

all pairs of developers who have a Cataldo et al. coordination requirement and have a granular proximity score > 0. There are 36 such pairs in DS1-a and 18 in DS2.

Proximity scores are calculated using events garnered instantaneously; while, Cataldo et al.'s method waits for changes to be committed. We obtained the date when the first contribution to the Proximity score occurred, by considering the timestamp for the first overlapping event for a developer pair recorded in the Mylyn context data. Similarly, we considered the time the Cataldo et al.'s method first identifies a coordination requirement, by considering the timestamp when the first technical dependency appears in the commits for a developer pair. For DS1-a, the first evidence of Proximity is detected on average 14.2 days after parallel work begins. The first Cataldo et al. coordination requirement detection happens 60.7 days on average after the beginning of concurrent work by a pair (a delay of 46.5 days). To put this in perspective, parallel work intervals last 102 days on average. The average "advance notice" provided by Proximity is, therefore, 87.8 days, compared to 55.5 days for the Cataldo et al. approach showing that Proximity significantly improves the timeliness of detection. For DS2, the first evidence of Proximity is detected on average 6.2 days after parallel work begins. The first Cataldo et al. coordination requirement is detected 17.9 days after the concurrent work begins (a delay of 11.7 days). Parallel work intervals last 31.4 days on average in this data set providing 25.2 days "advanced notice".

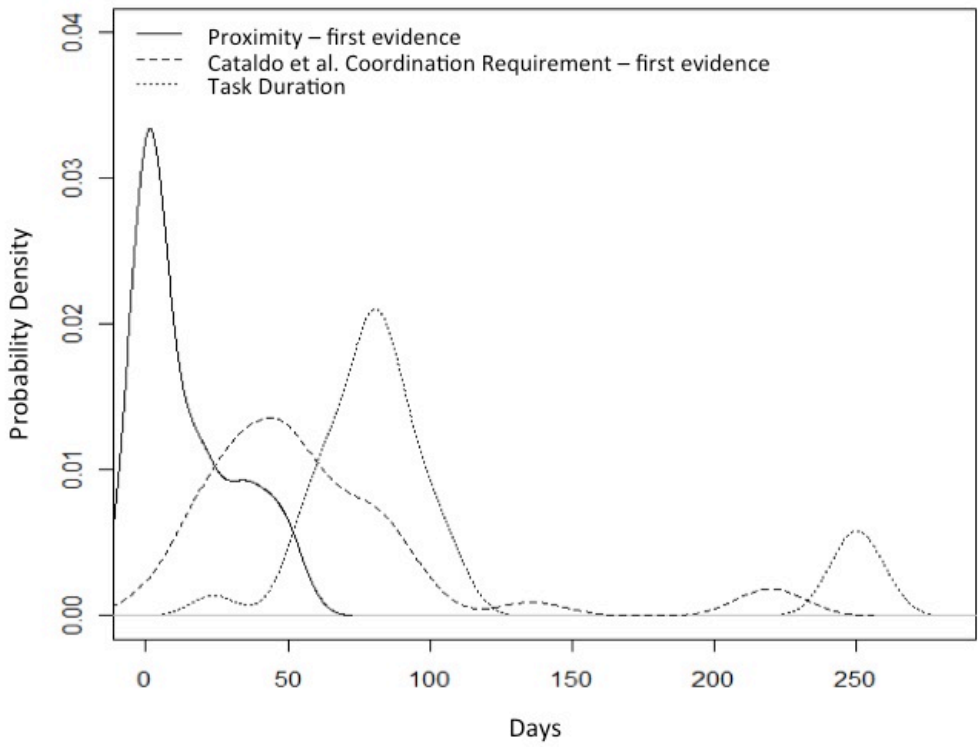


**Figure 5: Proximity Algorithm Timeliness.**

Figure 5 shows the probability density functions of Proximity detection, Cataldo et al. coordination requirement detection and task duration for DS1-a. It illustrates that Proximity can be detected much earlier than Cataldo et al. coordination requirements. Similarly distributed probability densities were seen in DS2.

### *4.3.4 Proximity Applied to Groups*

The original proximity algorithm computes only dyadic relationships (between pairs). Often, as large software projects progress, groups begin to emerge, and coordination becomes a group activity. We extended our work on the proximity algorithm to detect groups by looking at the intersections of multiple working sets

[62], [132]. We constructed a weighted bi-partite network revolving around the development tasks. The nodes in the bi-partite network represent (1) the developers involved in the tasks and (2) each set of overlapping artifacts between a pair of tasks. The edges link together the developers with those artifact intersections. The edges are weighted according to the number of artifacts in that intersection for which the developer consulted or edited for each of her tasks.

We used the arcs in the bi-partite network with weights above the median to construct bi-cliques [17]. These bi-cliques capture the groups of developers who tend to consult and manipulate the same artifact sets. Based on this set of bi-cliques, we computed a structural correlation matrix between developers. This matrix is a developer-by-developer network in which the weights between nodes represent the Pearson correlation between any two developers and signify how similar those two developers are in terms of the bi-cliques they are part of. To identify cohesive subgroups, we filtered out weak correlations using a cutoff point of 0.4 correlation between developers. In each of those simplified networks, the groups can be identified simply by visual inspection.

To evaluate this approach, we constructed these networks for eight releases of Mylyn development. We validated the groups established in these networks with qualitative information that was easily collected from Mylyn repositories on the web, such as conversations and developer profiles. We confirmed with this information the organizational structures and groups that were identified by our analysis. For additional validation and analysis, we used the communication traces of the team to construct alternate social networks. We used these "talk" social networks for comparison to the "work" networks constructed with our approach. These alternate

networks constructed based only on communications that have occurred offer a confirmatory view of the Mylyn team obtained through our analysis. The results of these two validation methods show that the groups we identified do represent emergent groups within the Mylyn development team [62], [132].

### 4.4 Discussion

Our results suggest that coordination needs between developers can be determined accurately based exclusively on the similarity of developer's consultation and edit activities on software artifacts. Unlike methods that rely on data that is available after work has been completed (commits), these developer activities are accessible while development is underway using IDE monitoring facilities like Mylyn. Our method, Proximity, adequately models the presence and intensity of coordination requirements independent of any conceptualization of technical dependencies. Proximity is not only timelier, but it can also be richer and more accurate. The timeliness and comprehensiveness that our Proximity measure provides is not currently available in other awareness tools for software engineering.

Our method uses both artifact selection and edit events to calculate Proximity scores. We speculate that developers could use a tool, like ProxiScientia, that is built using our Proximity method to identify coordination needs prior to beginning development work. The developer would simply need to open the source files that are likely involved in some task within their IDE. The use of selection events would allow Proximity to be calculated prior to any file edits. The developer, therefore, could be provided with a list of developers with high Proximity simply by identifying the set of artifacts that must be modified.

However, Proximity is limited in that it recommends only which developers need to coordinate. As developers often work on many tasks in parallel, this leaves the developers to identify which tasks require coordination and introduces inefficiencies. In the next chapters, we extend the Proximity method to provide more efficient recommendations.

### *4.4.1 Threats to Validity*

One limitation is that we considered any Proximity score >0 as an indicator of a possible coordination requirement. However, conceptually, a threshold of 0 seems sensible for comparison against the Cataldo et al. method as done in our study. The lowest possible Cataldo et al. coordination requirement score of 1 indicates that a developer pair worked on only one pair of dependent files. The lowest proximity score of 0.01 also indicates (at least) one artifact overlapping in the developers' working sets, which is conceptually similar.

Another possible limitation is that our analysis considered concurrent work at the release level. When considering finer grained temporal units, the outlook on coordination requirements and/or Proximity may differ. However, the Mylyn project does track tasks at the release level, so releases are likely a good unit for consideration of concurrent work. In addition, the major release cycle of the Mylyn project during the period of study is relatively short with an average release length of four months.

Finally, there may be issues of repeatability. Although Mylyn is widely adopted in open source as well as industrial settings, its consistent use by all developers during all of the project activities is not guaranteed. Currently, Mylyn context data must be manually attached to tasks by developers since a tool, like

ProxiScientia, that automatically sends Mylyn context data to a central database has not yet been adopted. The Mylyn team makes a consistent effort to attach their context data, but finding another project for an additional case study that also makes consistent use of the Mylyn plug-in may prove difficult. However, there are 108 projects in the Eclipse community alone that freely report Mylyn context data. Additionally, data analogous to what we obtained from the Mylyn context data and used in our study can easily be obtained from the other available IDE monitoring facilities, like Cubeon or Tasktop Dev.

### 4.5 Conclusion

We conclude this chapter by answering our first research question: *RQ1: Is timely coordination requirement detection possible?* Timely coordination requirement detection is possible with our Proximity method, which obtains developer actions as they occur through existing IDE monitoring facilities and analyzes the overlap of those actions to detect coordination needs.

# CHAPTER 5: EFFICIENT COORDINATION REQUIREMENT DETECTION

This chapter addresses our second research question: *RQ2: Can coordination requirements be identified efficiently at the task level of granularity?* When developers are working on multiple tasks concurrently, methods that recommend only which pairs of developers should coordinate may not provide enough information to allow for efficient coordination. Developers are left to decide which of their tasks require coordination. Identifying pairs of tasks that require coordination can provide more useful context for the involved developers and facilitate their coordination more efficiently. We explored the application of our Proximity method between pairs of tasks with the goal of avoiding information overload. We evaluated our results against the ground truth of coordination needs experienced by the team that we garnered by examining task records obtained from Bugzilla. A preliminary version of this investigation was published in the Proceedings of the 9th Joint Meeting on Foundations of Software Engineering and presented at that conference [14].

## 5.1 Approach

### *5.1.1 Applying Proximity to Identify Coordination Needs Between Tasks*

Many of the senior Mylyn developers we interviewed mentioned that awareness of coordination requirements would be more beneficial at the task level. When asked if a tool that recommended who to coordinate with would be useful, one developer stated "*if there was a lot more, than just talk to Joe. If it said like a new*

*defect was filed or look at this related bug, and Joe is the assignee. Then I would consider it and decide if it makes sense for me.*"

To compute coordination needs between pairs of tasks, we applied Proximity as described in Chapter 4 to Mylyn release 3.2 at the individual task level rather than at the developer level by aggregating the captured developer actions at the individual task level. The Mylyn release 3.2 had 245 tasks (29,890 task pairs). Since the events were aggregated at the task level, a Proximity score >0 indicates a coordination need between the tasks in the pair and the score itself denotes the strength of this coordination need.

We found 2,209 task pairs with Proximity scores > 0, and 226 of the 245 tasks were found to require coordination with at least one other task. This large number of coordination needs signals likely information overload when applying Proximity at the task level. It is unrealistic to expect that more than 92% of all tasks require coordination, and our interviews with senior Mylyn developers confirmed this. When looking at a potential coordination need, one developer stated, "[the two tasks] *are both working on the same area of code, but I don't see a direct need for coordination.*" Another developer focused on the simplicity of some tasks regardless of their technical dependencies saying on simple tasks, "*I wouldn't consider coordinating anything with anyone. I would just go in fix it, close the bug and be done with it.*"

This led us to believe that Proximity, when applied at the task level as opposed to the developer level, signaled coordination needs between too many task pairs. We, therefore, considered coordination needs identified by Proximity as the set of

potential coordination needs and investigated ways to identify a smaller set of the more critical coordination requirements.

#### *5.1.1.1 Defining Critical Coordination Needs*

While previous research has proposed ways to rank the most important coordination needs at the developer level by considering the number of task dependencies involved in those coordination requirements [51], [91], no prior research has examined the criticality of coordination needs at the task level. We consider two measures to evaluate the criticality of coordination needs at the task level: task duration and change size.

First, fulfilling coordination needs has been shown to reduce task resolution time [27], [28], [30], therefore we examined the durations of the tasks involved in the coordination needs. We compute task duration using the Mylyn context events. Since these events detail exactly when developers begin and complete their consultation and modification of artifacts for each task, using these context events allows us to compute the duration of the actual period of time developers spent working on a task. Long-duration tasks with coordination needs are likely those that can benefit the most from the productivity benefits provided by increased awareness and focused coordination.

Second, since the Mylyn team noted that they do not coordinate on simple or trivial tasks, we examined the complexity of the tasks involved in coordination needs. Cataldo et al. found that change size, measured as the number of code files modified during the course of development on a task, is an accurate measure of task complexity [30]. We, therefore, adopted change size as our metric of task complexity.

Task complexity is one of many factors that may influence a task's duration. Change size and task duration are strongly correlated in our data set (Spearman rho = 0.58, $p < 0.001$). Considering complexity as well as task duration helps us to avoid a bias towards tasks whose long duration may be due to some other factors that would not benefit as much from awareness and coordination, like low priority or inexperienced developers.

We, therefore, define critical coordination needs as those that involve complex tasks and can cause the most disruption to task duration.

#### *5.1.1.2 Criticality of Proximity Coordination Needs*

We evaluated the criticality of the coordination needs identified by Proximity using change size and task duration. While Proximity identifies potential coordination needs with longer task durations, it is not able to discriminate task complexity (Table 6). Therefore, we investigated how to provide more efficient recommendations by focusing our method on the identification of a smaller set of the more critical coordination requirements.

**Table 6 Criticality of Potential Coordination Requirements Identified by Proximity**

| | ***Potential Coordination Requirements*** | ***No Coordination Requirements*** | ***Mann-Whitney Test*** |
|---|---|---|---|
| Number of Tasks | 226 | 19 | -- |
| Change Size | 4.3 files | 4.0 files | W=28731.5 |
| Task Duration | 9.1 days | 0.7 days | W= 2799.0* |

(* $p < 0.01$, ** $p < 0.001$)

### *5.1.2 Detecting the More Critical Coordination Needs*

To better understand the characteristics of critical coordination needs, we manually examined a subset of task records to identify the actual critical coordination needs experienced by the team between that subset of tasks. We describe how we identified these critical coordination needs, which we use as our ground truth for evaluation, in Section 5.1.2.2. After a thorough analysis of that ground truth, we identified additional task properties that characterize the critical coordination needs experienced by the team (Section 5.1.2.3). We describe our approach, ProximityML, which enhances Proximity with these additional properties and leverages machine learning to automatically identify a reduced set of the more critical coordination needs (Section 5.1.2.4). We evaluated the accuracy of the results of ProximityML against the ground truth set of critical coordination needs experienced by the team, and we evaluated the criticality using change size and task duration.

#### *5.1.2.1 Description of Data Set*

We collected data from the development of two releases of Mylyn, Releases 3.1 and 3.2. For each release, we gathered data for all tasks for which we were able to obtain Mylyn context data attached to the task. There were 485 such tasks (117,370 task pairs) in Release 3.1 and 245 tasks (29,890 task pairs) in Release 3.2. We used the tasks from these releases to develop and analyze our techniques.

#### *5.1.2.2 Establishing Ground Truth Critical Coordination Needs*

A reliable way of capturing coordination needs is not currently available in any existing software repositories. Bugzilla and other bug tracking repositories allow developers to indicate dependencies between tasks, but this relationship may not

capture all coordination needs. In addition, a recent study by Aranda and Venolia [6] found repositories like Bugzilla often provide incomplete information because of omission, oversight, or simply because of project conventions. Therefore, we performed a thorough analysis of a set of task records to identify the ground truth of the coordination needs on the Mylyn team. We turned to content analysis and manual coding techniques that are well established in other research fields [88] and have recently been used in Software Engineering [95]. We used manual coding to develop a better understanding of critical coordination needs and provide us with a more accurate and exhaustive approximation of ground truth for a subset of the task pairs in our data set, which we could use when evaluating the results of our algorithms.

To perform the manual coding, we developed a coding scheme that provides detailed task pair scoring criteria. We used a data driven method and reviewed several task pairs in which the need for coordination is explicitly discussed within the task reports. Through analysis of these task pairs, we established a set of four characteristics that appeared within the task reports indicating the coordination need. These were: (1) task summary similarity, (2) task discussion similarity, (3) evidence of task conflict, and (4) artifact overlap. A preliminary version of our coding scheme, which includes each of these four characteristics, and a description of the manual coding method was published in the proceedings of the International Workshop on Social Software Engineering and presented at that workshop [15].

We obtained practical validation of these four characteristics through interviews with the Mylyn developers. Without indicating our identified characteristics, we asked three senior Mylyn developers what they would look for within the task reports to identify tasks with a coordination need. All three developers

stated they would review the discussion threads on the task reports looking for references to similar features or problems, similar areas of the code, or conflicts occurring between the tasks. Two of the developers did not think the task summary would provide enough information since the summary is often incomplete or inaccurate. None of the developers suggested looking at the overlapping artifacts between the two tasks. Artifact overlap suffers from the same problem that we are trying to solve, that is, it considers too many task pairs as having coordination needs.

We, therefore, removed two task characteristics and established the two characteristics – task discussion similarity and evidence of task conflict – that allow for the identification of coordination requirements between tasks. We put together a coding scheme that provided guidance on how each task pair should be rated for the two characteristics. The guidelines, which rate each characteristic on a three-point scale, are shown in Table 7.

To perform the content analysis, we used the relevant task information collected from the Bugzilla tasks. Each task was summarized in an easily digestible format, which allowed for two tasks to easily be viewed side-by-side for comparison.

**Table 7 Manual Coding Guidelines**

*NO COORDINATION NEED* *CRITICAL COORDINATION NEED*

| ***Characteristic*** | ***No*** | ***Somewhat*** | ***Very*** |
|---|---|---|---|
| **Task Discussion Similarity:** Task discussions often include details of the task and any problems that have been encountered. We asked the coders to rate the similarity of the discussions occurring on each task. | The discussions of the two tasks do not share any of the same concepts. | The two task discussions refer to common aspects of the system from the perspective of EITHER the user (system features) or the system architecture (specific reference to code, modules, etc.)<br>OR<br>The two task discussions indicate that the problems may be occurring in the same area of the code. | The two task discussions refer to common aspects of the system from the perspective of BOTH the user (system features) and the system architecture (specific reference to code, modules, etc.)<br>OR<br>The two task discussions refer to the same or similar problems. |
| **Evidence of Task Conflict**: Task conflict is the epitome of a coordination need and often indications of conflicts exist in the task discussions (explicitly or implicitly). We asked the coders to look for such evidence. | The discussion in the two tasks does not seem to indicate that the two tasks were conflicting in any way. | The discussion in one of the tasks does not explicitly mention a conflict between the two tasks. However, based on reviewing the timing of the tasks and their discussions, it seems there may have been a conflict between the two tasks that the team may not have been not aware of at the time. | It is apparent based on the timing of the tasks and the discussion thread that there was a conflict between the pair of tasks. The conflict is clearly discussed and may or may not explicitly link the two tasks by ID. |

To prepare the set of task pairs, we identified each task pair as either a potential critical coordination need or not. We considered a pair of tasks as a potential critical coordination need if the pair met one or more of the following criteria: the tasks had a high Proximity score where high is greater than mean + (2 x stddev) of

Proximity scores over all pairs; the tasks were explicitly marked as dependent or duplicate within their Bugzilla records; the tasks were cross-referenced in their discussions; the tasks were dependent on the same task (the Mylyn team often uses this relationship to track subtasks of a large task); or the tasks were marked with the same tag. Once each task pair was designated as either a potential coordination need or not, we used a random number generator to select pairs from each set. We selected 155 potential critical coordination needs and 195 that were likely not coordination needs for a total set of 350 pairs. The number of pairs included in the manual coding was based on the time availability of the coders. We choose to include a large number of potential critical coordination needs because we suspected that many of those potential needs would not be confirmed as critical coordination needs through manual coding.

We used two external people familiar with software development practices to perform the manual coding. To ensure higher confidence, the two coders performed the content analysis and coding independently. After each of the coders completed 12 task pairs, the two coders compared their findings and discussed differences as a way to calibrate between each other. Another comparison and calibration round was carried out after 100 task pairs. We checked intercoder reliability with Krippendorff's alpha measure [88]. We obtained a Krippendorff score of .91 for task discussion similarity and .87 for evidence of task conflict after one coding run. Those sores are indicative of high intercoder reliability. We did not perform a second reconciliation pass considering this high intercoder reliability. Instead, we removed the task pairs where there was disagreement between the coders.

We considered any task pair that was rated positively (a score of either "somewhat" or "very" on our three-point scale in Table 7) for either characteristic as a coordination need experienced by the team. We removed the task pairs for which the coders had a conflicting outcome leaving us with 313 task pairs. These task pairs serve as our approximated ground truth, which we use for evaluation purposes in the rest of the analysis described in this dissertation. Among these 313 task pairs, 32 (10.2%) were identified as coordination needs by the coders.

To examine whether these 32 coordination needs identified by the coders are critical, we analyzed the task duration and change size (our measures of criticality) of the 52 individual tasks involved in those 32 pairs. Those tasks have significant differences in both measures (results in Table 8). This suggests that the coders were successful – using the coding scheme we devised - in identifying the more critical coordination needs experienced by the team.

**Table 8 Criticality: Manual Coding Results**

| | ***Coordination Requirements*** | ***No Coordination Requirements*** | ***Mann-Whitney Test*** |
|---|---|---|---|
| Number of Tasks | 208 | 52 | -- |
| Change Size | 8.2 files | 5.3 files | W=9398** |
| Task Duration | 26.8 days | 19.9 days | W=8603** |

(* $p < 0.01$, ** $p < 0.001$)

#### *5.1.2.3 Analysis of Task Pair Properties*

We examined task pair properties of the ground truth critical coordination needs identified through manual coding and compared them to all other task pairs to identify properties that can be used to distinguish the more critical coordination needs. The task properties we examined include (1) architecture-related properties available from the project's change request database such as: the affected product, component, platform and operating system (OS) of the task and (2) modularity characteristics of the software artifacts involved in each task.

We examined the architecture-related properties by checking, for each task pair, if the tasks involved in that pair shared any of those properties (i.e. if they affect the same product, component, platform, or OS). A Chi-squared test of difference in proportion for each property shows that there is a significant difference between the ground truth critical coordination needs and all other task pairs for all but one of the tested properties: there is not a statistically significant difference for the number of task pairs marked for the same product (results in Table 9).

**Table 9 Task Property Comparison**

| ***Property*** | ***Coordination Requirements*** | ***Other Task Pairs*** | ***Chi-Squared Test*** |
|---|---|---|---|
| Task Pair Count | 18 | 29,890 | -- |
| # with Proximity | 15 | 2,209 | $x^2 = 150.7$ ** |
| # with same Product | 13 | 21,082 | $x^2 = 0.03$ |
| # with same Component | 12 | 6,450 | $x^2 = 21.6$ ** |
| # with same Platform | 16 | 13,858 | $x^2 = 13.1$ ** |
| # with Same OS | 13 | 9,713 | $x^2 = 12.9$ ** |
| | | | ***Mann-Whitney Test*** |
| Mean SLSM | 3.67 | 0.28 | W=399713.0** |
| Mean SLDM | 6.94 | 3.27 | W=301919.5 |
| Mean AL | 3.89 | 0.51 | W=436253.5** |

(* $p < 0.01$, ** $p < 0.001$)

To consider the modularity characteristics of the software artifacts involved in each task, we derived a Design Rule Hierarchy (DRH) [139] of the Mylyn code base for the two releases of interest. We choose the DRH among the various metrics that describe a project's code base and its modularity since it was conceived specifically to identify modules that can be independently assigned to developers for parallel work. A DRH (described in detail in Chapter 2) assigns software artifacts to modules and layers based on technical dependencies within the code. We use the DRH modules and layers to identify potential coordination needs by considering three categories of work: (1) Same Layer Same Module (SLSM) pairs, (2) Across Layer (AL) pairs, and (3) Same Layer Different Module (SLDM) pairs. The SLSM and AL categories are potential coordination needs since dependencies exist between these task pairs.

The Mylyn Project DRH of release 3.1 consists of 11 layers and 671 modules. The DRH of release 3.2 consists of 11 layers and 786 modules. We identified the associated DRH layer and module for each java artifact edit action captured in the

Mylyn context data. Using this information, we obtained the number of SLSMs, SLDMs, and ALs for each task pair.

We analyzed each of these properties to identify any that appear significantly different between the ground truth critical coordination needs and all other task pairs. A Mann-Whitney test of difference in distribution shows that the difference is statistically significant for both the number of SLSMs and ALs, but there is not a significant difference for the number of SLDMs (results shown in Table 9). This is consistent with the semantics of DRH and the empirical results by Wong et al. [139] that found developers engaged in SLSM and AL pairs communicate (a dominant form of coordination in software development [87]) significantly more than those engaged in SLDM pairs.

We, therefore, determined the following set of task pair properties that differentiate task pairs with ground truth critical coordination needs from all other task pairs:

- Within same component
- Within same platform
- Within same operating system
- Number of SLSMs
- Number of ALs

#### *5.1.2.4 ProximityML*

We developed ProximityML, which enhances the Proximity method with these identified properties to find a reduced set of coordination needs that includes the more critical coordination needs. ProximityML uses the Support Vector Machine

(SVM) classification technique [34]. An SVM is a supervised machine learning classification algorithm. Given a training set, it produces a model that can be used to predict the classification of unknown instances given a set of known parameters of those unknown instances [34]. SVM was selected because of its accuracy in general and its tolerance to noise and irrelevant, redundant and interdependent attributes [86].

In earlier attempts to leverage machine learning [14], we used the k-nearest neighbor algorithm [35] due to the simplicity of implementing the algorithm and the exploratory nature of that study. We also previously looked at the DRH properties differently, simply considering the number of overlapping layers and modules between task pairs. We found that looking at the number of SLSMs and ALs is a much better predictor of coordination needs since these types of overlaps are directly related to dependencies in the code base. The results we achieved using the SLSM and AL characteristics and adopting SVM far surpass those achieved using the k-nearest neighbor algorithm, where we achieved high recall but a precision score of only 0.09.

We used LIBSVM [31] as our implementation of the SVM algorithm. LIBSVM is a java software package that provides support vector classification. It performs data scaling, parameter selection and model creation automatically. It ensures the data scaling is consistent across all data sets based on the range of each parameter in each set. For example, if a parameter in the training set had a range of [-10, +10] and the same parameter had a range of [-9, +12] in the test set, that parameter would be scaled to a range of [-1, +1] in the training set and to a range of [-0.9, +1.2] in the test set. The LIBSVM library uses the RBF (radial basis function) kernel. To perform parameter selection, it estimates the accuracy of each combination

of parameters through cross validation (CV). The parameter combination with the highest CV score is selected.

We used the properties determined to have statistical significance in Section 5.1.2.3 as the known parameters. We used the ground truth dataset that had been manually coded through content analysis (313 total task pairs, with 32 coded as critical coordination needs) to train and evaluate the machine learning algorithm. The subset of task pairs from the ground truth data set from release 3.1 (200 task pairs with 18 critical coordination needs) was used as a training set. We classified each of the 29,890 task pairs from Release 3.2 using ProximityML. The subset of the pairs from that release that are part of the ground truth data set (113 task pairs with 14 critical coordination needs) was used to evaluate our results. Each parameter in our training set was linearly scaled to the range [-1, +1]. The parameters in the unknown and evaluation sets were scaled accordingly based on their range compared to the training set values.

## 5.2 Evaluation Methodology

To answer our second research question – *RQ2: Can coordination requirements be identified efficiently at the task level of granularity?* – we evaluated the coordination needs at the task level on Release 3.2 of the Mylyn project. We evaluated the accuracy and criticality of the ProximityML coordination needs. Accuracy was evaluated relative to the ground truth critical coordination needs established through content analysis and manual coding. Criticality was evaluated using change size and task duration as described in Section 5.1.1.1.

## 5.3 Analysis and Results

### *5.3.1 Accuracy of ProximityML*

ProximityML significantly reduced the number of predicted coordination needs compared to Proximity alone. Proximity identified 2,209 coordination needs, while ProximityML predicted only 394 coordination requirements, a reduction of 82%.

**Table 10 Accuracy: Ground Truth Critical Coordination Needs vs. Proximity and ProximityML Coordination Needs**

| | ***Precision*** | ***Recall*** | ***F1-score*** |
|---|---|---|---|
| Proximity | 0.33 | 1 | 0.5 |
| ProximityML | 0.77 | 0.71 | 0.74 |

We compared the task level coordination requirements identified by both Proximity and ProximityML with the ground truth critical coordination needs established through content analysis and manual coding. The differences in precision, recall, and f1-score of Proximity and ProximityML are shown in Table 10 for the 113 task pairs in our evaluation set, which included 14 ground truth critical coordination needs. In this small evaluation set, ProximityML had both high precision (low false positives) and recall (low false negatives) resulting in high overall accuracy, as shown by the f1-score. While a small number of coordination needs may be missed when employing ProximityML, it does not risk introducing a large number of false

positives. On the other hand, Proximity has no false negatives, but a high number of false positives. Overall, the accuracy of ProximityML is much higher than Proximity.

To select the model parameters, we performed a grid search using exponentially growing sequences of C and γ using five-fold cross-validation. Five-fold cross validation distributes the training set into five subsets of equal size. Each subset is tested using a model that is trained with the training instances from the other four subsets. The cross-validation (CV) rate is the percentage of the training set instances that are correctly classified. The grid search selects various pairs of the C and γ parameters and selects the pair with the best CV rate [78].

The selection of the C parameter introduces a trade off between error and over fitting [34], [79]. Low values of C may obtain a high error rate. High values of C may obtain better results for the given data set, but the model may not generalize well to other data. The γ parameter defines how much a single instance in the training set influences the produced model [4]. Training set instances have greater influence with lower γ values. We obtained an average CV rate of 93.0 with the best $c=2^{11}$ and $\gamma=2^{-7}$ (Figure 6). This high CV rate indicates we have a stable model that is able to accurately predict different samples; thus, we have avoided over fitting our model. To create our model, we used the best C and γ parameters selected using the grid search.

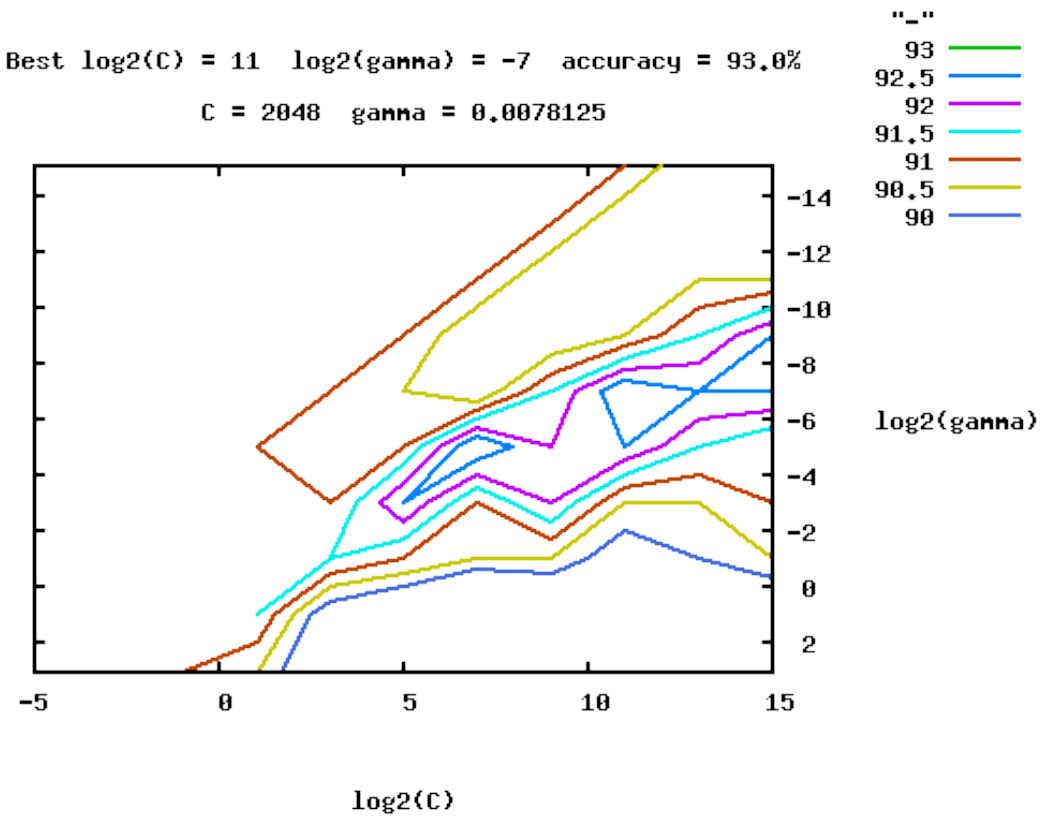


**Figure 6: Grid Search Parameter Selection Results.**

A Receiver Operating Characteristic (ROC) curve plots the true positive rate against the false positive rate for a binary classifier. The ROC curve shown in Figure 7 illustrates the high performance of our classifier with the Area Under the Curve (AUC) equal to 0.8452.

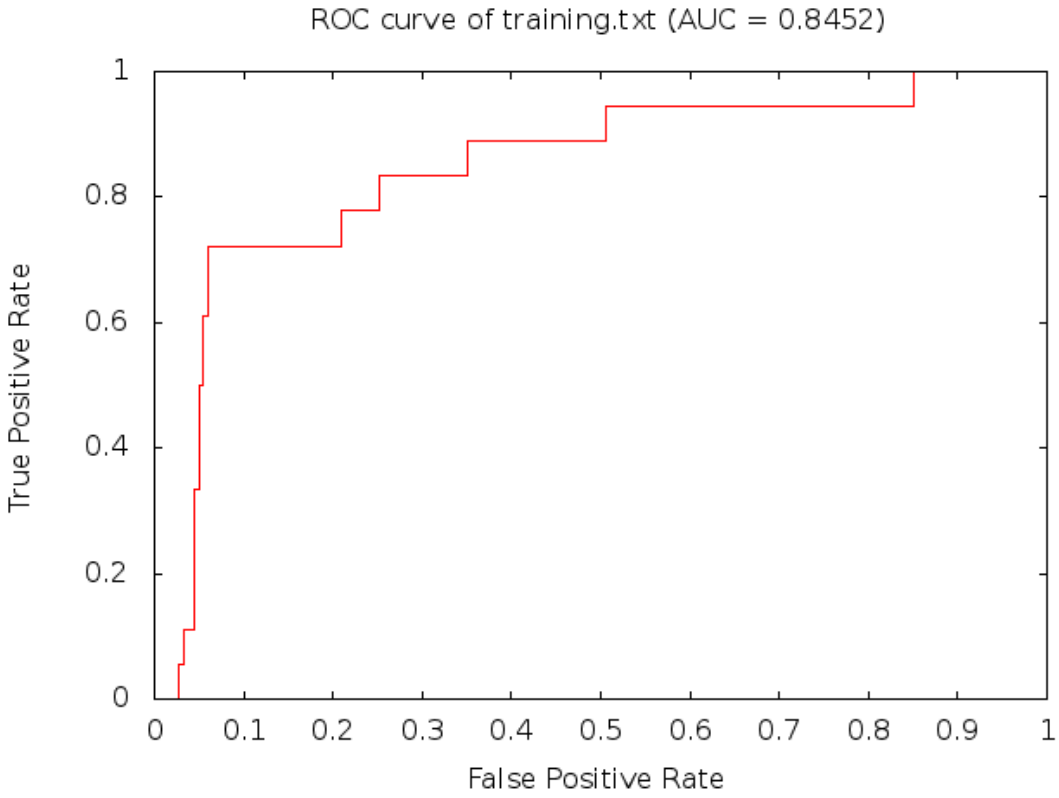


**Figure 7: ROC Curve.**

### *5.3.2 Evaluating Criticality of Coordination Needs with ProximityML*

We examined the ProximityML coordination requirements using our two measures of criticality described in Section 5.1.1.1: change size and task duration. We see a strong, significant difference in both change size and task duration between the ProximityML coordination requirements and those tasks pairs without ProximityML coordination requirements (Table 11).

**Table 11 Criticality: ProximityML Coordination Requirements vs. ProximityML Non-Coordination Requirements**

| | ***Coordination Requirements*** | ***No Coordination Requirements*** | ***Mann-Whitney Test*** |
|---|---|---|---|
| Number of Tasks | 152 | 93 | -- |
| Change Size | 5.6 files | 4.0 files | W = 22709** |
| Task Duration | 12.16 days | 2.3 days | W = 9666** |

(* $p < 0.01$, ** $p < 0.001$)

In addition, Mann-Whitney tests show both the change size and task durations of the tasks involved in the coordination requirements detected through the ProximityML approach are significantly different than when considering the Proximity method (Table 12). The mean task durations are significantly longer and the mean change size is significantly bigger than the means of the Proximity method. This suggests that the properties used to enhance the Proximity metric and the use of the machine learning techniques described in this dissertation are identifying the more critical coordination needs when criticality is conceptualized using change size and task duration.

**Table 12 Coordination Requirements Criticality: ProximityML vs. Proximity**

| | ***Proximity*** | ***ProximityML*** | ***Mann-Whitney Test*** |
|---|---|---|---|
| Change Size | 4.3 files | 5.6 files | W = 20294.5** |
| Task Duration | 9.1 days | 12.16 days | W = 7829.0** |

(* $p < 0.01$, ** $p < 0.001$)

### 5.4 Discussion

This is the first research to explore and exploit the differences that exist between potential coordination requirements at the task level. Through developer interviews, we confirmed that developers do not believe that all technical dependencies require coordination. However, all other existing methods and tools (besides ProximityML) are based on that assumption (that all technical dependencies require coordination).

We investigated techniques to identify the more critical coordination needs in a software project. To provide developers with more efficient recommendations on their coordination needs, we computed Proximity scores between tasks rather than between developers. To avoid information overload, we enhanced Proximity with additional task properties and machine learning techniques to identify a set of the more critical coordination needs. The resulting approach, ProximityML, identified a subset of the coordination requirements that appear more critical, as measured by change size and task duration.

In addition, we have shown how code modularization properties that can be derived from the system DRH are also useful indicators of coordination needs. These findings build upon and reinforce previous empirical results that found that DRHs are adept at highlighting the intertwined relationships between issues of coordination and issues of modularity [139].

An approach that is able to narrow the set of coordination needs to those that are more critical has implications for both research and practice. In the next chapter, we will evaluate the reliability, timeliness and usefulness of ProximityML.

### *5.4.1 Threats to Validity*

One threat to validity is that we were limited in the number of task and code properties that we could investigate. There may be additional, or even better, properties that could be used to differentiate the overall set of potential coordination requirements and highlight the more critical ones. The properties that are relevant in this case study may not be relevant in others. In addition, all properties may not be portable across different bug tracking systems.

Our measure of task duration, which we used to evaluate the criticality of coordination needs between tasks, could be affected by other factors, such as the priority of the tasks, workload of the team, physical location of the developers, and experience level of the developers. However, we considered task complexity in addition to task duration to evaluate the criticality of coordination needs between tasks, which helps us to avoid a bias towards tasks whose long duration may be due to these other factors. In addition, previous empirical studies, such as the study by Cataldo et al. [27], found that while these factors impact development time, the impact of unmanaged coordination needs is also highly significant. In our Mylyn study, this risk is further mitigated by the characteristics of the Mylyn project itself and the general nature of open source projects. The Mylyn team is comprised of well-established, experienced developers. Open source projects are accustomed to working in distributed environments [98], [116], and developer overload is not a large concern, since contributors choose which tasks to work on [98], [143].

Finally, although we cannot exclude that our results could be caused by some other hidden factors that underlie the properties we selected, this threat is mitigated by the large size and diversity of our data set (29,890 task pairs).

## 5.5 Conclusion

We conclude this chapter by answering our second research question: *RQ2: Can coordination requirements be identified efficiently at the task level of granularity?* By using additional task properties, our approach, ProximityML, was able to efficiently identify coordination needs by identifying coordination needs between pairs of tasks and focusing on the more critical coordination needs. Of course, we cannot conclude that we have identified all coordination needs or all of the most critical ones. However, we have shown that ProximityML can identify a subset of the more critical coordination needs, and that ProximityML is an approach with a low number of both false positives and false negatives. Thus, ProximityML is an efficient method of identifying coordination requirements.

# CHAPTER 6: TIMELY DETECTION OF CRITICAL COORDINATION REQUIREMENTS

This chapter addresses our last research question: *RQ3: Are the more critical coordination needs actionable?* We evaluated the ProximityML approach described in Chapter 5, which identifies the more critical coordination needs between tasks. In this chapter, we examined the (1) reliability of the ProximityML recommendations over time and (2) timeliness of the recommendations. We also evaluated our approach with a number of Mylyn developers. We asked the developers to comment on how actionable the recommendations made by a tool that implements our approach would be and how they would use such a tool in their project.

## 6.1 Evaluation Methodology

To evaluate the reliability and timeliness of ProximityML, we ran our machine learning techniques on our time-ordered data, which included each Mylyn context event and Bugzilla update event (task creation and task modifications). We streamed the data, one event at a time, to replicate the actual progression of development work and live collection of the data. We ran the machine learning algorithms to calculate coordination needs after every event. We performed this exercise on the data collected for Mylyn release 3.2. The machine learning algorithm was pre-trained with the training set from release 3.1. Since data was streamed one event at a time, the machine learner initially had no knowledge of any data beyond the training set. This also allowed us to evaluate the start-up behavior of ProximityML.

It should be noted that we did not take task start or end times into consideration when computing ProximityML coordination requirements within a

release. This means that coordination needs could be found between any pair of tasks within the release regardless of when in the release those tasks were completed. We did not decay ProximityML coordination requirements in any way as the involved tasks aged meaning that if a coordination need appears in the beginning of the release, it will not go away when those tasks have been completed. While these could be potential features of a tool that would implement ProximityML, they were not needed for this analysis since we are not presenting these coordination needs to developers as recommendations.

## 6.2 Analysis and Results

### *6.2.1 Reliability of ProximityML*

To evaluate the reliability of our approach, we recomputed the ProximityML coordination needs after each event. Reliability is important because a tool that continuously changes its recommendations would not be trusted. Murphy and Murphy-Hill [99] found that users' trust immediately drops when a recommender tool produces an irrelevant recommendation. A recommendation must only be made once it is firmly established as a critical coordination need.

After each event, we examined the number of ProximityML coordination requirements that had been identified. At each point in time, we identified which predicted coordination requirements were not included in the final set of ProximityML coordination requirements (those detected after all events have been streamed). We consider any predicted coordination need that does not appear in the final set of ProximityML coordination requirements a false positive for the purposes of this exercise. Figure 8 shows the number of coordination requirements as well as

false positives over the duration of the entire dataset. We observe that there is a short period of unreliability during the early stage of the data streaming, but after that very brief initialization period, the results are reliable and contain a minimal number of false positives. Also, once a ProximityML coordination requirement is recommended, it tends to remain a recommendation.

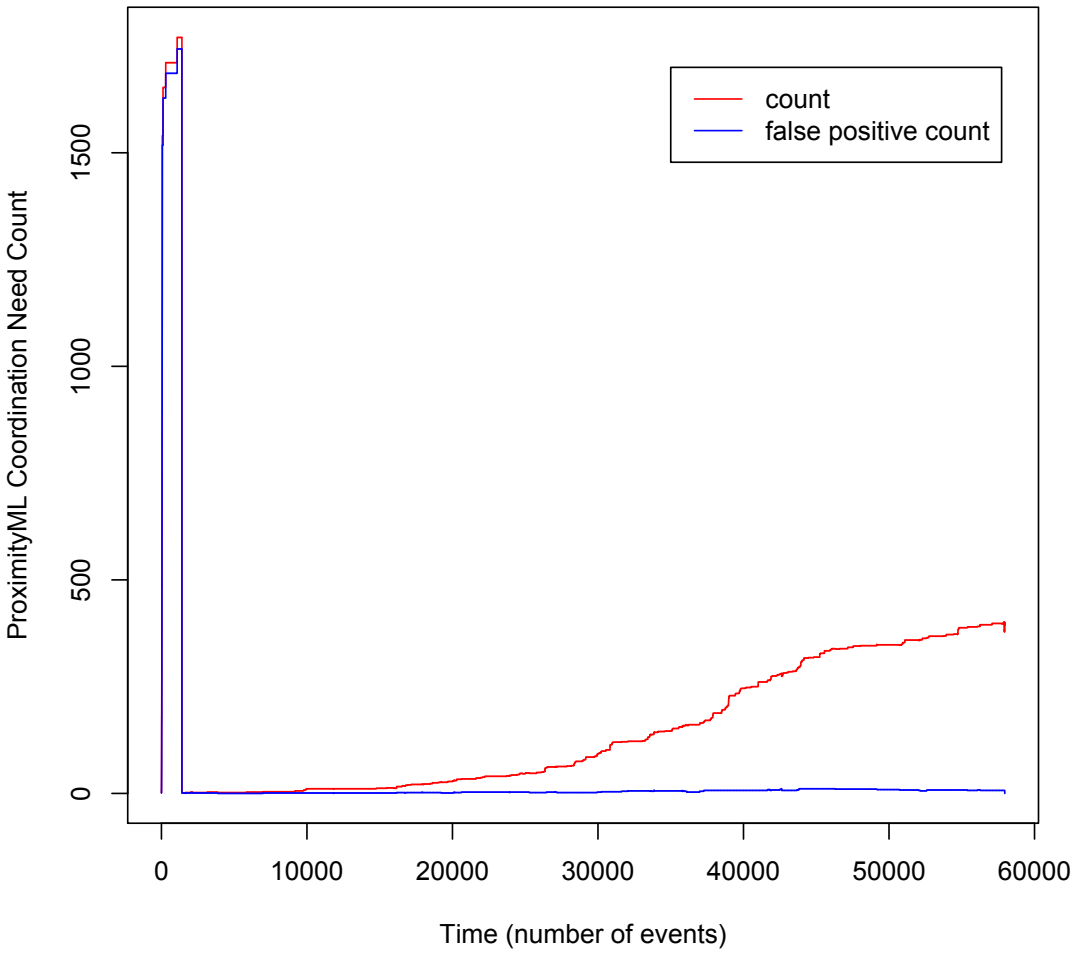


**Figure 8: Evolution of ProximityML Coordination Requirements Over Time.**

### *6.2.2 Timeliness of ProximityML*

To examine the timeliness of the detection of ProximityML coordination requirements, we identified the timestamp when each of the 394 ProximityML coordination requirements was first identified by ProximityML. We examined the 19 coordination requirements in this set that were recognized by the team in the Bugzilla task reports. To identify task pairs that had been considered dependent by the development team, we consulted several areas of the Bugzilla task reports and identified three types of dependencies that can be mined: (1) the explicitly marked "depends on/blocks" relationship, (2) the "duplicate" relationship between tasks, and (3) the task cross-referencing relationship. There were 57 task pairs that were identified as dependencies within the task records when considering these three recorded dependency types.

Sixteen of these 19 recognized coordination requirements were known dependencies immediately at the time of the second task creation. These tasks represent either task/subtask relationships or offshoot tasks where some new task is created based on something that was discovered during the development of the first task. In these cases, we cannot expect ProximityML to perform better than the development team. Still, in all but one case, ProximityML automatically identifies these recognized coordination requirements promptly after the creation of the second task: as shown in Figure 9, most are identified on the same day or the day after the second task is created. The remaining three recognized coordination requirements were not identified by the team until sometime later during the development of the second task. ProximityML identifies two of these recognized coordination requirements on the same day as the team. The remaining one recognized

coordination requirement is identified by ProximityML more than one month before the team.

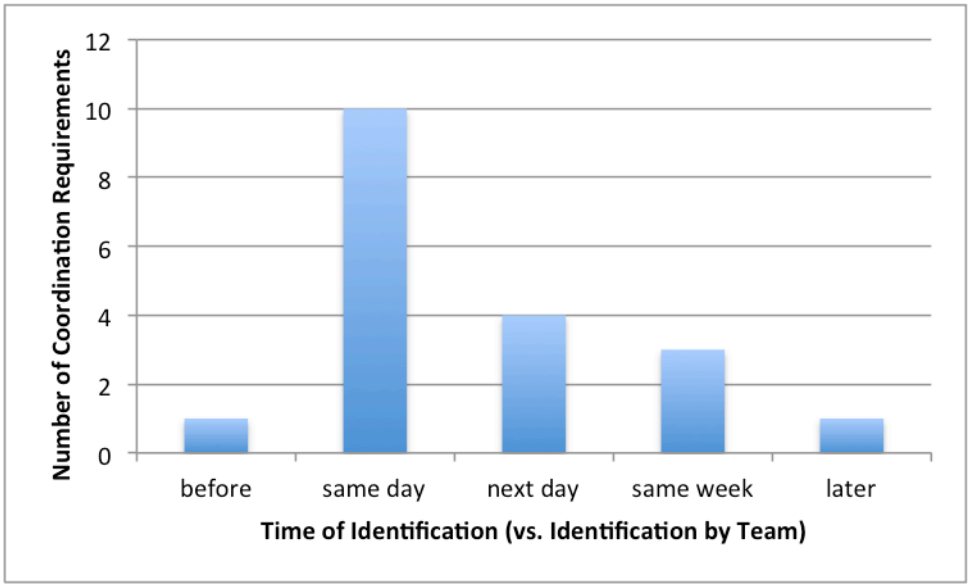


**Figure 9: Coordination Need Detection Timeliness for Recognized Dependencies.**

While this represents only a very small set of recognized coordination requirements, it shows the promise of ProximityML to automatically provide timely awareness to the development team. Since it provides both accurate detection and early recognition, ProximityML delivers recommendations that are actionable. This is especially important when those coordination needs are not immediately evident to the team members.

The remaining coordination requirements in our case study do not seem to have been recognized by the team based on the data within the Bugzilla records. It is certainly possible (though we do not have any recorded evidence) that some of these coordination needs were documented or managed in some other way. However, it is

likely that many of the ProximityML coordination requirements were unknown by the development team and, therefore, unmanaged.

Since we have no direct way to compare the time of detection for the remaining 375 unrecognized coordination requirements, we instead analyzed the timeliness of the detection of coordination requirements relative to the start of overlapping work. The start of overlapping work was calculated by considering the timestamp of overlapping Mylyn context events for each coordination requirement. ProximityML coordination requirements are identified on average 3.6 days after the start of overlapping work with the median detection occurring on the same day as the start of overlapping work. This provides actionable recommendations considering the average development duration for tasks in this data set is nearly 25 days. Figure 10 illustrates the timeliness with probability density functions showing that ProximityML typically detects coordination needs when overlapping work starts or shortly after. This early detection makes the ProximityML coordination requirements actionable.

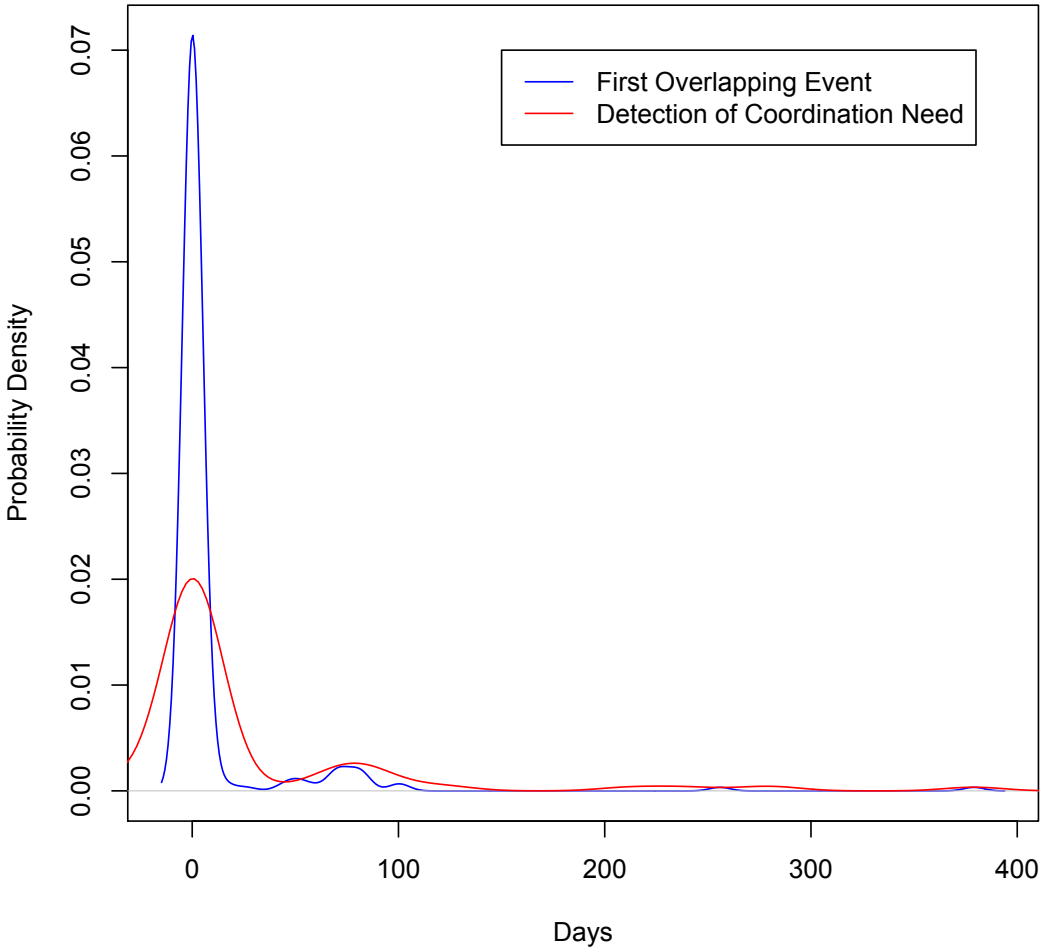


**Figure 10: ProximityML Timeliness Probability Density.**

### *6.2.3 Usefulness of ProximityML: Developer Interviews*

We further evaluated the usefulness of ProximityML by conducting semi-structured interviews with six developers of the Mylyn project (one junior and five senior developers). The goal was to understand their perspective on the actionability of timely coordination recommendations. Three interviews were conducted in person, one was conducted through Skype and two were completed asynchronously through email. The interviews lasted 45 minutes on average. We asked these main questions:

- *If you had a tool to recommend coordination needs as they emerge, how would it be useful? What features would it have?*
- *How would you decide if you were going to act on the coordination needs it suggested?*
- *What type of time window for recommending coordination needs do you think would be useful and why?*
- *How would it help with your coordination?*

All developers believed that such a tool would be useful in their work and that coordination needs are most useful at the task level. They also stated that the number of recommendations would need to be small. A large number of recommendations would overwhelm the developer. This could affect the efficiency of the developers and cause them to ignore all recommendations. This is in line with previous research on the risk of information overload in awareness tools [53], [75], [101], [127].

While keeping the number of recommendations small is important for efficiency, extremely relevant recommendations should not be disregarded simply because of the amount of time that has elapsed since the completion of the other task. One developer noted that "*of course, the more recent tasks are generally more interesting because they are people you can actually do real coordination with, but from the point of view of understanding similarities, having a history of things that happened, there is something really interesting about.*" For example, he said "*you may be doing something that someone tried 5 years ago, and they have information about why it failed and that's something that could be completely lost because of turnover or forgetfulness. And that is huge information even though it is 5 years ago, so having the whole time window is useful.*" Having awareness of that older, very

relevant task can prevent developers from repeating mistakes and increase efficiency on the current task.

Interestingly, nearly all of the developers indicated that simply being aware of the tasks that are identified as coordination requirements would suffice as a form of coordination. The developers would avoid interrupting the assignee of that task through explicit coordination methods like email or chat. One developer stated "*just by looking at the bug report too, you can rule out your potential need to go interrupt that person or figure out, alright I'll just hold off my development until they are done or whatever the case may be, instead of actually going and interrupting that person. So you can glean a lot of information from that report just by being aware of the similar reports you should be looking at.*"

Of course, while avoiding direct communication can be efficient, it is not always possible. One developer cautioned that a tool that provides both coordination recommendations and built-in coordination mechanisms could cause projects experts (who may be involved in a large number of coordination recommendations) to be overwhelmed with coordination requests. He suggested a way for experienced team members to flag themselves as busy during certain time periods when they cannot afford to be interrupted.

The developers thought timely coordination need awareness would be most useful for large teams. They stated that recommendations would be particularly useful for task pairs that span across teams within a project since they are less aware of what other teams are working on. They also believed more experienced developers would benefit the most from awareness of coordination needs since they have the knowledge to understand the related tasks. This is in line with the feedback we received from the

junior developer who thought the most useful recommendations would be those that addressed essentially the same problem, so the other task could be used as a model for the current task. The junior developer did not see the value in being made aware of potentially conflicting tasks since she assumed that senior project personnel, such as her supervisor, would make her aware of any conflicts that required coordination.

## 6.3 Discussion

We showed that ProximityML provides timely coordination recommendations that are reliable and consistent over time. Through developer interviews, we found that our approach is useful for developers. The developers stressed the need to keep the number of recommendations small, confirming that it is necessary to narrow the set of recommendations to those that are more critical. However, they also stated that even recommendations for tasks that were completed significantly before the current task are useful when they are extremely relevant to help understand project history. This reinforces the fact that the machine learning component of our approach must be able to focus on the most relevant coordination needs. Identifying an appropriate training set is extremely important. In Chapter 8, we discuss how the machine learning algorithm may even be trained individually for each developer to ensure developers obtain recommendations they are most likely to utilize. Also in Chapter 8, we describe several avenues for future research that can improve the accuracy of our approach.

### *6.3.1 Usefulness*

Existing awareness tools that provide recommendations of coordination needs to developers have not yet been largely adopted in practice. Awareness provided by

such a tool has the potential to help avoid coordination breakdowns and decrease task resolution time, software faults, build failures, redundant work, and schedule slips [27], [28], [30], [39], [41], [44]. However, the state of the art for these tools suffers from two main limitations that we believe are hindering their adoption: lack of timely support and inefficient recommendations.

Thus, identifying a more timely and efficient approach to coordination need detection is useful. We have shown that our approach is *timely* and accurate. It focuses on the more critical coordination needs at the task level. Such recommendations are *efficient* and the preferred level of detail for developers. We have also shown that our approach is reliable and consistent over time.

Our approach could be implemented into an awareness tool that provides coordination recommendations for developers. Through developer interviews, we found that a tool implementing our approach would be useful for developers. We discuss recommendations and guidelines for such a tool in Chapter 8.

### *6.3.1 Threats to Validity*

This Chapter presents additional analysis on the ProximityML approach described in Chapter 5. Therefore, the threats to validity described in Section 5.4.1 also apply here. In addition, our interviews were limited to a small set of developers who volunteered to discuss coordination. The views of those developers may not reflect the majority. However, our interviews did span a wide range of individuals including a junior developer, senior developers including a Tasktop scrum master, and one developer who has recently taken a management position. Furthermore, some of our interviewees were physically located at the Tasktop office in Vancouver, Canada and some were distributed.

## 6.4 Conclusion

We conclude this chapter by answering our last research question: *RQ3: Are the more critical coordination needs actionable?* ProximityML's more critical coordination recommendations are actionable since they are reliable, timely and useful for developers.

# CHAPTER 7: DISCUSSION OF RESEARCH CONTRIBUTIONS

This chapter revisits our research questions, techniques and results. It describes how our approach can be used in other projects and discusses our contributions. It concludes with a discussion on the limitations of our studies.

## 7.1 Summary

In this dissertation, we described a number of techniques for providing timely and efficient coordination recommendations in software teams. To summarize our studies and results, we revisit each of our research questions:

*RQ1: Is timely coordination requirement detection possible?*

To identify timely coordination requirements, we developed Proximity, which uses IDE monitoring to consider the overlap between developer artifact consultation and edit activities in a working set and infers coordination needs between developers. Since IDE monitoring captures developer actions as they occur, Proximity enables the timely identification of coordination needs. Through an empirical study of eight releases of the Mylyn open source project, we found that Proximity scores adequately model the presence and intensity of coordination requirements when compared against the Cataldo et al. method [27], [28], [30] by examining correlations, precision, recall and a regression model. We showed that Proximity provides for more timely identification of coordination needs when using the Cataldo et al. method as a baseline.

*RQ2: Can coordination requirements be identified efficiently at the task level of granularity?* To provide more efficient recommendations, we adjusted Proximity to

detect coordination needs between pairs of tasks, that is one level of analysis more granular than the developer level. To avoid information overload and a high number of false positives, we used a set of task properties that distinguished the ground truth coordination needs to enhance the Proximity metric in our ProximityML approach. ProximityML uses machine learning on those task properties and Proximity scores to filter recommendations to the more critical coordination needs, providing a smaller number of more critical coordination recommendations (394 ProximityML coordination needs compared to 2,209 Proximity coordination needs in our study). We validated our techniques on the tasks from one release of the Mylyn project.

We compared the ProximityML coordination requirements with the ground truth critical coordination needs established through content analysis and manual coding. We showed that ProximityML is accurate in terms of precision, recall and f1-scores. We plotted the true positive rate against the false positive rate in an ROC curve and verified that our machine learning classifier is accurately predicting the more critical coordination needs. We found that the coordination needs identified by ProximityML are more critical, as measured by change size and task duration.

*RQ3: Are the more critical coordination needs actionable?*

Finally, we evaluated the reliability and timeliness of ProximityML by streaming each event (developer action and Bugzilla task update) in a time-ordered sequence and re-running the ProximityML approach after each event. We found that ProximityML is able to identify coordination needs shortly after overlapping work begins and that when a coordination need is established, it tends to stay a coordination need. Through developer interviews, we confirmed that our approach is useful for developers.

## 7.2 Contributions

This dissertation described a set of techniques for providing timely and efficient coordination recommendations in software teams. The key contributions of this dissertation are:

1. A method for timely and accurate identification of coordination needs between software developers, Proximity.
2. An approach for identifying coordination needs at the level of tasks.
3. A method for identifying the ground truth of coordination needs experienced during development work by examining task reports.
4. An analysis of task properties that are indicative of critical coordination needs.

The final outcome, the ProximityML approach, represents a systematic approach to use task properties and machine learning techniques in a software project for timely and efficient coordination needs recommendation. The key contributions of ProximityML are:

5. It provides timely and accurate detection of coordination needs.
6. It considers coordination needs at the task level instead of developer level to provide more granular recommendations.
7. It avoids information overload by identifying a set of the more critical coordination needs.
8. It can be applied to many software projects as described in Section 7.3.

Finally, we also contribute to the growing body of research on coordination needs through our detailed discussions:

9. A discussion on developer needs gathered through interviews.

10. A discussion on the implications of our work for both research in coordination and tool design in Sections 8.1 and 8.2.
11. A discussion on avenues for future research in this area in Section 8.3.

### 7.3 Using our Approach in Other Projects

Our approach is feasible to any software project in which IDE monitoring and logging is possible. Coordination recommendations can be identified timely and efficiently in other projects by considering each component of our approach: (1) a ground truth of critical coordination needs, (2) task properties that distinguish critical coordination needs, and (3) an SVM machine learning algorithm.

**Develop Ground Truth**: First, a set of task pairs with known critical coordination needs and task pairs that do not require coordination must be identified. If this information is not reliably or completely available within the repositories of the project, it can be established using the manual coding guidelines we developed in Section 5.1.2.2 or through consultation with the development team. The ground truth will be used to analyze task properties and to train the machine learning algorithm.

**Identify Relevant Task Properties**: A list of properties that helps distinguish critical coordination needs must be identified for the project. While it is likely that the properties described in our analysis of the Mylyn project data will also apply to other projects, they may not be universally applicable due to specific project processes or conventions. A list of project-specific task properties can be identified by comparing the ground truth critical coordination needs with task pairs that do not require coordination as we described in Section 5.1.2.3 for the Mylyn project. A statistical evaluation can identify properties that differ significantly for the known critical coordination needs for the project.

**SVM machine learning algorithm**: With the ground truth (training set) and task properties, the approach described in Section 5.1.2.4 can be applied. The input to the SVM machine learning algorithm is the training set where each instance of the training set is classified (critical coordination requirement or not) and described by each of the selected properties. After training the machine learning algorithm, unknown task pairs can be classified by providing the values of the selected properties. When task pairs are classified as critical coordination needs, coordination recommendations can be made to the developers assigned to those tasks.

### 7.4 Threats to Validity

Our findings derive from a single case study of the Mylyn project with a relatively moderate number of developers and tasks. Our results could be affected by specificities of the case. To mitigate this risk, we performed a detailed analysis of Proximity using eight versions of the Mylyn project. ProximityML was evaluated using a release with a large number of tasks (245 tasks over four months of development). Our analysis consisted of a mixed methods approach including statistical investigations, in-depth examinations of coordination needs and developer interviews to understand the team's coordination practices and problems. Our detailed analysis of this project allowed us to better understand when coordination is necessary and how to identify critical coordination needs.

# CHAPTER 8: CONCLUSIONS

This chapter describes implications of our work for research in coordination and a set of guidelines for tool design. It also highlights avenues worth pursuing in future research in this area.

## 8.1 Implications for Coordination Research

ProximityML provides coordination recommendations at the task level. This is in line with previous research that found that developers are interested in awareness about information relevant to their tasks [85] and that it is most productive to align software teams based on the tasks they must complete [89]. The developers we interviewed stated that the recommendation of coordination needs is most useful between pairs of tasks.

**Implication #1:** *Identifying only the more critical coordination needs is important.* At the more granular level of tasks, there can be many potential coordination needs, and it is especially important to focus on critical recommendations to prevent information overload. More research is needed to identify ways to further refine our ProximityML approach to further reduce the false positive rate. While, we have achieved high levels of precision (0.77) and recall (0.71), future work could improve those false positive and false negative rates. We describe several future research avenues in this direction in Section 8.3.

**Implication #2:** *Awareness of tasks leads to forms of implicit coordination.* An important finding emerging from the interviews was how the developers said they would attend to coordination needs. All of the senior developers indicated that, upon

receiving a recommendation of a coordination need between tasks, they would review the related task to obtain details of how that task impacts their own work as a first step. This review would result in awareness of the related task. They would prefer to avoid interrupting the developer assigned to the other task, even if it meant delaying their own task. In the Mylyn development environment, reviewing the appropriate related tasks could be seen as a form of stigmergic coordination [17], [52] considering that the team does encourage documentation of all task details within the task report. With timely and efficient coordination recommendations, this practice could be extended to other development environments. We discuss how our approach can support implicit and stigmergic coordination in Implications for Tool Design (Section 8.2).

**Implication #3:** *Effects of implicit coordination in software engineering needs further study.* Many existing empirical studies on team coordination examine explicit means of communication such as email, chat or meetings, largely because they are more easily traceable. We believe it is equally important to take into account other means of coordination. For example, studies that use measures for Socio-Technical Congruence (STC) [27], [28], [30], [92] could be improved by also considering metrics for awareness about tasks as sufficient coordination to fulfill a coordination need. Future studies could examine this possibility by considering either tasks that developers are watching or have subscribed to or tasks that have been reviewed by developers, which could also be obtained through IDE monitoring facilities. Additionally, further information on developer awareness of tasks or of other developers can be garnered from "social" features that have recently been introduced in software repositories and development communities like GitHub [37].

### 8.2 Implications for Tool Support

Existing awareness tools that detect coordination needs are not timely enough to enable developers to act on their coordination needs. Additionally, they identify only the developer pairs that should coordinate. This puts the burden on the developers to identify which tasks require coordination among possibly many concurrent development tasks. Our work shows the potential of a support tool for developers, based on the approach described in this dissertation, which automatically recognizes coordination needs between pairs of tasks as they emerge. Such a tool could be used to provide coordination awareness both within and across teams, support coordination among developers, and automate task dependency management.

The envisioned tool could incrementally and unobtrusively learn from evidence of coordination actions taken by the team (discussions, cross-referencing of task pairs, etc.) to continuously improve the machine learning accuracy. It should have a large pool of potential task properties and perform task property selection based on incremental learning to ensure that the properties selected are best suited for the development processes and practices of the team. Our work indicates some main design guidelines for such a tool.

**Guideline #1:** *The tool must be unobtrusive.* The developers we interviewed suggested displaying coordination recommendations either within the task reports themselves, which developers often consult throughout development, or within an IDE plug-in. The recommendations should include links to the other task reports and any other relevant task information to allow the developer to easily gather information about the task on their own without interrupting the other task assignee. It could also include an easy way to display the areas of code that are overlapping or conflicting.

There should be in-tool coordination mechanisms including email, Skype, Yammer or other communication software used by the project. However, developers should have a way to flag themselves as busy to avoid interruption when necessary. A tool might also consider the priority of a task when making recommendations or when displaying the available options on how to fulfill the coordination need. Perhaps – for low-priority tasks – only implicit types of coordination would be suggested.

**Guideline #2:** *The tool must balance the relevance and timeliness of the coordination need to provide the most valuable recommendations.* A tool would likely decay coordination requirements as the involved tasks aged. It would also need to identify a time window of interest for tasks to incur a coordination requirement (i.e. only overlapping tasks or tasks where development work occurs no more than two weeks apart). This time window should be a tunable attribute since different developers may have different preferences. From our interviews, we learned that understanding very relevant tasks that were completed much earlier in the project's timeframe could still be useful in some cases. For example, when the new task is attempting to tackle the same issue as a previous unsuccessful task. This illustrates another way developers may use such a tool for the awareness of tasks rather than for explicit coordination. Especially strong and relevant coordination needs may be displayed regardless of the completion status of the other task. Again, this feature should be tunable to meet the team's or individual developer's needs.

**Guideline #3:** *The tool should consider the experience level of the developer when making recommendations.* The developers we interviewed believed more experienced developers would benefit the most from awareness of coordination needs since they have the knowledge to understand the related tasks. While previous

research [117] found that developers consider the expertise of others before initiating coordination, our findings suggest that the expertise of the developers themselves may impact what coordination they deem necessary. More junior developers may want a smaller set of only extremely relevant recommendations. Recommendations, therefore, may consider not only the properties of tasks, but also the task assignee.

**Guideline #4:** *The tool should support implicit coordination [17], [52].* Our developer interviewees would prefer to gather task information themselves rather than interrupting a task assignee. While the Mylyn/Tasktop team, by convention, makes an active effort to record all information related to each task within the corresponding task report in Bugzilla, not all projects follow the same convention. Tools to help developers easily review all data related to a given task could be devised. There has been some research in this area; Rastkar and Murphy [108] use Murray's summarization technique [100] to summarize email threads related to a specific bug report. However, there are many other forums that could hold information relevant to a task (IRC, Discussion Boards, Yammer, Skype chats, etc.) as well as other information sources like design documentation or requirement specifications. The awareness tool we described above could be improved by providing a summary for each of the tasks involved in coordination needs so the developer can quickly browse the information. A tool could summarize the task report [94], as well as all task information from these various sources, and prioritize and highlight the most relevant information. The developer could view additional details of tasks that require further investigation. Such a tool would enable a developer to more efficiently become aware of other tasks.

Since the tool is encouraging implicit coordination, it should also have ways to allow developers to indicate their awareness of other tasks and identify what strategy they are taking to diminish any coordination needs. For example, developers may indicate that they are waiting on a particular task to be completed to avoid coordination. This would prevent the assignees of both tasks unknowingly waiting on each other.

We envision the tool supporting software developers, project managers and software architects:

**Software Developers:** A tool that helps make developers aware of their coordination requirements as they emerge while avoiding a large number of false positives and focusing on the more critical coordination needs can allow developers to focus their coordination efforts where it is truly needed. Awareness provided by such a tool can help avoid coordination breakdowns resulting in decreased task resolution time, software faults, build failures, redundant work, and schedule slips [27], [28], [30], [39], [41], [44].

**Project Managers:** Such a tool could also provide a project manager view visualizing the more critical coordination needs within and across teams. This can allow for prioritization of project governance actions aimed at the resolution of coordination requirements that improve productivity the most [51]. As an alternative, changes could be made to the design or the team structure to eliminate coordination requirements [131] lowering the coordination overhead of the project.

**Software Architects:** Finally, the tool could provide a software architect view that highlights the areas of code that cause the most coordination needs. Such a feature would enable software architects to continuously monitor the software design

and focus any redesign efforts to reduce coupling and increase cohesion in problematic areas of the code.

## 8.3 Future Work

We see three avenues for future research in this area, based on the work we presented in this dissertation:

- Continue Investigation of Task Properties
- Implement and Deploy Recommender Tool
- Increase Understanding of Implicit Coordination

### *8.3.1 Continue Investigation of Task Properties*

One future research direction could be to continue our investigation of task properties and their role in identifying coordination needs by analyzing additional task properties beyond those we have described in this dissertation (product, component, platform, Operating system, and the DRH SLSM, SLDM and AL counts). Evaluating properties that are available on different bug tracking systems and characterize the architecture of a task like Trac's component, Redmine's category, Jira's components and labels, and GitHub's labels would be a logical next step. A comprehensive set of properties can improve the accuracy of our approach and allow for easier and more general adoption of our approach.

We have identified several other properties that may potentially be useful indicators for critical coordination requirements: (1) DRH layers, (2) Key Modules, and (3) Additional Granularity of Changes.

**DRH Layers:** A DRH clusters modules into "layers" where each layer depends only on the layers above. The layers can be used to differentiate artifacts that

represent influential design decisions (design rules) from low-level artifacts that depend on those decisions. Changes made in the highest layers (most influential design decisions) may represent more critical coordination needs.

**Key Modules:** Similarly, modules identified as important may also be involved in more critical coordination needs. Key modules may be identified in several different ways by considering: (1) the coupling and cohesion of each module, (2) the frequency of change for each module, (3) the frequency of coordination needs resulting from each module, (4) the size of each module, or (5) the DRH properties of each module.

**Additional Granularity of Changes:** The lowest level of granularity available from the Mylyn context data is the class element (method or attribute). Additional granularity could be useful: the event could indicate if an edit was made to the method's API or within the method body. Changes made to a method's API may indicate more critical coordination needs. While this information is not currently available from Mylyn context data, it would be a relatively simple change within the Mylyn project to make this type of data available.

### *8.3.2 Implement and Deploy Recommender Tool*

A tool that implements the ProximityML approach we presented and validated in this dissertation could be implemented. The tool could build upon the ProxiScientia prototype [19]. The implementation should follow the guidelines and recommendations in Section 8.2. With an implemented tool, further studies of coordination needs in the field could be performed to improve the accuracy of the recommendations. The tool could be deployed in a project and run in the background only, so that the developers on the team are not aware of the recommendations made

during the period of the study. The study could examine the recommendations made by the tool while researchers shadow the development team to understand their coordination needs and problems. The insights gained through such a study would allow for the possible refinement of the training set or selected task properties and, hopefully, increase the accuracy in recommendations.

### *8.3.3 Increase Understanding of Implicit Coordination*

Empirical studies on coordination often do not include implicit coordination mechanisms since they are not as traceable and, therefore, not easily understood. However, the developers we interviewed stated a preference for implicit coordination. As researchers, we must identify ways to better understand and identify implicit coordination. Studies on coordination that do not include implicit mechanisms risk incomplete results. With IDE logging facilities and the introduction of "social" features into many software repositories, implicit coordination may become easier to trace.

## 8.4 Conclusion

In this dissertation, we described a number of techniques for providing timely and efficient coordination recommendations in software teams and a number of studies that evaluated those techniques. Our approach is timely since it analyses data obtained from IDE monitoring facilities, which can be obtained as developer activities occur. Our approach is efficient since it makes recommendations at the task level (the developers unit of work) and focuses on the more critical coordination needs reducing information overload. We also discussed the implications of our work for research in

coordination, a set of guidelines for tool design, and possible directions for future research based on this work.

## List of References


1. K. R. Al-Asmari and L. Yu, "Experiences in Distributed Software Development with Wiki," *Proc. Int'l Conf. Software Eng. Research and Practice (SERP 06)*, CSREA Press, 2006, pp. 389–393.

2. B. Al-Ani, E. Trainer, R. Ripley, A. Sarma, A. Van Der Hoek, and D. Redmiles, "Continuous Coordination within the Context of Cooperative and Human Aspects of Software Engineering," *Proc. Int'l Workshop Cooperative and Human Aspects of Software Eng. (CHASE 08)*, ACM, 2008, pp. 1-4.

3. J. Al-Kofahi, A. Tamrawi, T. Nguyen, H. Nguyen, and T. Nguyen, "Fuzzy Set Approach for Automatic Tagging in Evolving Software," *Proc. Int'l Conf. Software Maintenance (ICSM 10)*, IEEE, 2010, pp. 1–10.

4. R. Albatal and S. Little, "Empirical Exploration of Extreme SVM-RBF Parameter Values for Visual Object Classification," unpublished.

5. C. Alexander, *Notes of the Synthesis of Form (Vol. 5)*, Harvard University Press, 1964.

6. J. Aranda and G. Venolia, "The Secret Life of Bugs: Going Past the Errors and Omissions in Software Repositories," *Proc. 31st Int'l Conf. Software Eng. (ICSE 09)*, IEEE CS, 2009, pp. 298-308.

7. C.Y. Baldwin and K.B. Clark, *Design Rules, Vol. 1: The Power of Modularity*, MIT Press, 2000.

8. H. Bani-Salameh, C. Jeffery, and J. Al-Gharaibeh, "SCI: Towards a Social Collaborative Integrated Development Environment," *Proc. Int'l Conf. Computational Science and Engineering (CSE 09)*, IEEE, 2009, pp. 915– 920.

9. K. Bauer, M. Fokaefs, B. Tansey, and E. Stroulia, "WikiDev 2.0: Discovering Clusters of Related Team Artifacts," *Proc. Conf. Center for Advanced Studies on Collaborative Research (CASCON 09)*, IBM Corporation, 2009, pp. 174-187.

10. A. Begel, K.Y. Phang, and T. Zimmerman, "Codebook: Discovering and Exploiting Relationships in Software Repositories," *Proc. 32nd Int'l Conf. on Software Eng. (ICSE 10)*, IEEE, 2010, pp. 125-134.

11. S. Betz, A. Moss, C. Wohlin, D. Šmite, W. Afzal, J. Börstler, S. Fricker, M. Svahnberg, and T. Gorschek, "An Evolutionary Perspective on Socio-Technical Congruence: The Rubber Band Effect," *Proc. 3rd Int'l Workshop Replication in Empirical Software Engineering Research (RESER 13)*, IEEE, 2013, pp. 15-24.

12. J.T. Biehl, M. Czerwinski, G. Smith, and G.G. Robertson, "FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams," *Proc. Conf. Human Factors in Computing Systems (CHI 07)*, ACM, 2007, pp. 1313-1322.

13. C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy, "Does Distributed Development Affect Software Quality?: An Empirical Case Study of Windows Vista," *Communications of the ACM*, vol. 52, no. 8, 2009, pp. 85-93.

14. K. Blincoe, G. Valetto, and D. Damian, "Do all task dependencies require coordination? The role of task properties in identifying critical coordination needs in software projects," *Proc. 9th Joint Meeting on Foundations of Software Engineering* (ESEC/FSE 13), ACM, 2013, pp. 213-223.

15. K. Blincoe, G. Valetto, and D. Damian, "Uncovering critical coordination requirements through content analysis," *Proc. Int'l Workshop Social Software Eng.* (SSE 13), ACM, 2013, pp. 1-4.

16. K. Blincoe, G. Valetto, and S. Goggins, "Proximity: a Measure to Quantify the Need for Developers Coordination," *Proc. Conf. Computer Supported Cooperative Work* (CSCW 12), ACM, 2012, pp.1351-1360.

17. F. Bolici, J. Howison, and K. Crowston, "Coordination Without Discussion? Socio-Technical Congruence and Stigmergy in Free and Open Source Software Projects," *Proc. 2nd Int'l Workshop Socio-Technical Congruence (STC 2009)*, IEEE, 2009, doi=10.1.1.193.7473.

18. S.P. Borgatti and M.G. Everett, "Network Analysis of Two Mode Data," *Social Networks*, vol. 19, no. 3, 1997, pp. 243–269.

19. A. Borici, K. Blincoe, A. Schroeter, G. Valetto, and D. Damian, "ProxiScientia: Toward Real-Time Visualization of Task and Developer Dependencies in Collaborating Software Development Teams," *Proc. Int'l Workshop Cooperative and Human Aspects of Software Eng. (CHASE 12)*, IEEE, 2012, pp. 5-11.

20. F.P. Brooks, *The Mythical Man-Month: Essays on Software Engineering*, Addison Wesley, 1995.

21. Y. Brun, R. Holmes, M.D. Ernst, and D. Notkin, "Proactive Detection of Collaboration Conflicts," *Proc. 8th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 11)*, ACM, 2011, pp. 168–178.

22. Y. Brun, D. Notkin, R. Holmes, and M.D. Ernst, "Early Detection of Collaboration Conflicts and Risks," *IEEE Transactions on Software Engineering*, Oct. 2013.

23. F. Calefato, D. Gendarmi, and F. Lanubile, “Embedding Social Networking Information into Jazz to Foster Group Awareness within Distributed Teams,” *Proc. 2nd Int’l Workshop Social Software Engineering and Applications (SoSEA 09)*, ACM, 2009, pp. 23–28.

24. F. Calefato and F. Lanubile, “SocialCDE: A Social Awareness Tool for Global Software Teams,” *Proc. 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 13)*, ACM, 2013, pp. 587-590.

25. M. Cataldo, M. Bass, J. Herbsleb, and L. Bass, “On Coordination Mechanisms in Global Software Development,” *Proc. 2nd Int’l Conf. Global Software Engineering (ICGSE 07)*, IEEE, 2007, pp. 71-80.

26. M. Cataldo and K. Ehrlich, “The Impact of Communication Structure on New Product Development Outcomes,” *Proc. Int’l Conf. Human Factors in Computer Systems (CHI'12)*, ACM, 2012, pp. 3081-3090.

27. M. Cataldo and J.D. Herbsleb, “Coordination Breakdowns and Their Impact on Development Productivity and Software Failures,” *IEEE Transactions on Software Engineering*, vol.39, no.3, 2013, pp. 343-360.

28. M. Cataldo, J. Herbsleb, and K. Carley, “Socio-Technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity,” *Proc. 2nd Int’l Symposium Empirical Software Engineering and Measurement (ESEM 08)*, ACM, 2008, pp. 2-11.

29. M. Cataldo, A. Mockus, J.A. Roberts, and J.D. Herbsleb, “Software Dependencies, Work Dependencies and Their Impact on Failures,” *IEEE Transactions on Software Engineering*, vol. 35, no. 6, 2009, pp. 864-878

30. M. Cataldo, P.A. Wagstrom, J.D. Herbsleb, and K.M. Carley, “Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools,” *Proc. 20th Conf. Computer Supported Cooperative Work (CSCW 06)*, ACM, 2006, pp. 353-362.

31. C.C. Chang and C.J. Lin, “LIBSVM: A Library For Support Vector Machines,” *ACM Transactions on Intelligent Systems and Technology*, vol.2, no.3, 2001, pp. 27.

32. L.L. Constantine and E. Yourdon, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice Hall, 1979.

33. M.E. Conway, “How do Committees Invent?” *Datamation*, vol. 14, no. 4, 1968, pp. 28-31.

34. C. Cortes and V. Vapnik, "Support Vector Machine," *Machine Learning*, vol. 20, no. 3, 1995, pp. 273-297.

35. T. Cover and P. Hart, "Nearest Neighbor Pattern Classification," *IEEE Transactions of Information Theory*, vol.13, no.1, 1967, pp.21-27.

36. B. Curtis, H. Krasner, and N. Iscoe, "A Field Study of the Software Design Process for Large Systems," *Communications of the ACM*, vol. 31, no. 11, 1988, pp.1268–1287.

37. L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social Coding in GitHub: Transparency And Collaboration in an Open Software Repository," *Proc. Conf. Computer Supported Cooperative Work (CSCW 12)*, ACM, 2012, pp.1277–1286.

38. I.A. Da Silva, P.H. Chen, C. Van der Westhuizen, R.M. Ripley, and A. Van Der Hoek, "Lighthouse: Coordination Through Emerging Design," *Proc. OOPSLA Workshop on Eclipse Technology eXchange (ETX 07)*, ACM, 2006, pp. 11-15.

39. D. Damian, L. Izquierdo, J. Singer, and I. Kwan, "Awareness in the Wild: Why Communication Breakdowns Occur," *Proc. 2nd Int'l Conf. Global Software Eng. (ICGSE 07)*, IEEE, 2007, pp. 81-90.

40. C.R. de Souza, J.M. Costa and M. Cataldo, "Analyzing the Scalability of Coordination Requirements of a Distributed Software Project," *Journal of the Brazilian Computer Society*, vol. 18, no. 3, 2012, pp. 201-211.

41. C.R. de Souza and D.F. Redmiles, "An Empirical Study of Software Developers' Management of Dependencies and Changes," *Proc. 30th Int'l Conf. Software Eng. (ICSE 08)*, ACM, 2008, pp. 241-250.

42. C.R. de Souza, and D.F. Redmiles, "The Awareness Network, to Whom Should I Display My Actions? And, Whose Actions Should I Monitor?," *IEEE Transaction on Software Engineering*, vol.37, no.3, 2011, pp. 325-340.

43. C.R. de Souza, D.F. Redmiles, L.T. Cheng, D. Millen, and J. Patterson, "How a Good Software Practice Thwarts Collaboration: The Multiple Roles of APIs in Software Development," *ACM SIGSOFT Software Engineering Notes*, vol.29, no.6, Oct. 2004, pp. 221-230.

44. C.R. de Souza, S. Quirk, E. Trainer, and D.F. Redmiles, "Supporting Collaborative Software Development through the Visualization of Socio-Technical Dependencies," *Proc. Int'l Conf. Supporting Group Work (GROUP 07)*, ACM, 2007, pp. 147-156.

45. P. Dewan and R. Hegde. "Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development," *Proc. 10th European Conf. Computer Supported Cooperative Work (E-CSCW 07)*, Springer London, 2007, pp. 159-178.

46. M. Di Penta, M. Harman, G. Antoniol, and F. Qureshi, "The Effect of Communication Overhead on Software Maintenance Project Staffing: a Search-Based Approach," *Proc. Int'l Conf. Software Maintainence (ICSM 07)*, IEEE, 2007, pp. 315-324.

47. P. Dourish and V. Bellotti, "Awareness and Coordination in Shared Workspaces," *Proc. Conf. Computer-Supported Cooperative Work (CSCW 92)*, ACM, 1992, pp. 107-114.

48. K. Dullemond and B. van Gameren, "What Distributed Software Teams Need To Know And When: An Empirical Study," *Proc. 8th Int'l Conf. Global Software Eng. (ICGSE 13)*, IEEE, 2013, pp. 61-70.

49. K. Dullemond, B. Gameren, M.A. Storey, and A.V. Deursen, "Fixing the 'Out of Sight Out of Mind' Problem: One Year of Mood-Based Microblogging in a Distributed Software Team," *Proc. 10th Int'l Workshop Mining Software Repositories (MSR 13)*, IEEE Press, 2013, pp. 267-276.

50. K. Dullemond and R. van Solingen, "Increasing Awareness in Distributed Software Teams: a First Evaluation," *Proc. 9th Int'l Conf. Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 13)*, IEEE, 2013.

51. K. Ehrlich, M. Helander, G. Valetto, S. Davies, and C. Williams, "An Analysis of Congruence Gaps and Their Effect on Distributed Software Development," *Workshop Social Technical Congruence (STC 08)*, 2008.

52. M.A. Elliott, "Stigmergic Collaboration: The Evolution of Group Work," *M/C Journal*, vol.9, no.2, May 2006, http://journal.media-culture.org.au/0605/03-elliott.php.

53. M.J. Eppler and J. Mengis, "The Concept of Information Overload: A Review of Literature from Organization Science, Accounting, Marketing, MIS, and Related Disciplines," *The Information Society*, vol. 20, no. 5, 2004, pp. 325–344.

54. S.K. Ethiraj and D.A. Levinthal, "Modularity and Innovation in Complex Systems," *Management Science*, vol. 50, no. 2, 2044, pp. 159–173.

55. J. Estublier and S. Garcia, "Process Model and Awareness in SCM," *Proc. 12th Int'l Workshop Software Configuration Management (SCM 05)*, ACM, 2005, pp. 69-84.

56. G. Fitzpatrick, S. Kaplan, T. Mansfield, D. Arnold, and B. Segall, "Supporting Public Availability and Accessibility with Elvin: Experiences and Reflections," *Computer Supported Cooperative Work*, vol. 11, no. 3-4, 2002, pp. 447-474.

57. T. Fritz and G.C. Murphy, "Determining Relevancy: How Software Developers Determine Relevant Information in Feeds," *Proc. Conf. Human Factors in Computing Systems (CHI 11)*, ACM, 2011, pp. 1827-1830.

58. T. Fritz and G.C. Murphy, “Using Information Fragments to Answer the Questions Developers Ask,” *Proc. 32nd Int’l Conf. Software Eng. (ICSE 10)*, ACM, 2010, vol. 1. pp. 175-184.

59. J. Froehlich and P. Dourish, “Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams, *Proc. 26th Int’l Conf. Software Eng. (ICSE 04)*, IEEE CS, 2004, pp. 387-396.

60. H. Gall, K. Hajek, and M. Jazayeri, “Detection of Logical Coupling Based on Product Release History,” *Proc. Int’l Conf. Software Maintainence (ICSM 98)*, IEEE, 1998, pp. 190-198.

61. R.L. Gauthier and S.D. Ponto, *Designing Systems Programs*, Prentice-Hall, 1970.

62. S. Goggins, G. Valetto, C. Mascaro, and K. Blincoe, “Creating a Model of the Dynamics of Socio-Technical Groups,” *User Modeling and User-Adapted Interaction*, Springer, 2012, pp. 1-35.

63. R.E. Grinter, J.D. Herbsleb and D.E. Perry, “The Geography of Coordination: Dealing with Distance in R&D Work,” *Proc. Int’l Conf. Supporting Group Work (GROUP 99)*, ACM, 1999, pp. 306-315.

64. M.L. Guimarães and A.R. Silva, “Improving Early Detection of Software Merge Conflicts, *Proc. Int’l Conf. Software Eng. (ICSE 12)*, IEEE Press, 2012, pp. 342-352.

65. C. Gutwin, S. Greenberg, and M. Roseman, “Supporting Awareness of Others in Groupware,” *Proc. Conf. Companion on Human Factors in Computing Systems (CHI 96)*, ACM, 1996, pp. 205.

66. C. Gutwin, R. Penner, and K. Schneider, “Group Awareness in Distributed Software Development,” *Proc. Conf. Computer Supported Cooperative Work (CSCW 04)*, ACM, 2004, pp. 72-81.

67. A. Guzzi and A. Begel, “Facilitating Communication Between Engineers with CARES,” *Proc. Int’l Conf. Software Eng. (ICSE 12),* IEEE Press, 2012, pp. 1367-1370.

68. A. Guzzi, M. Pinzger, and A. van Deursen, “Combining Micro-Blogging and IDE Interactions to Support Developers in their Quests,” *Proc. Int’l Conf. Software Maintenance (ICSM 10),* IEEE CS, 2010, pp. 1–5.

69. T. Hattori, “Wikigramming: a Wiki-Based Training Environment for Programming,” *Proc. 2nd Int’l Workshop Web 2.0 for Software Eng. (Web2SE 11)*, ACM, 2011, pp. 7–12.

70. L. Hattori and M. Lanza, "Syde: A Tool for Collaborative Software Development," *Proc. 32nd Int'l Conf. Software Eng. (ICSE 2010)*, IEEE, 2010, vol. 2, pp. 235-238.

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

72. J.D. Herbsleb and A. Mockus, "An Empirical Study of Speed and Communication in Globally Distributed Software Development," *IEEE Transactions on Software Engineering*, vol.29, no.6, 2003, pp. 481-494.

73. J.D. Herbsleb, A. Mockus, and J.A. Roberts, "Collaboration in Software Engineering Projects: A Theory of Coordination," *Proc. Int'l Conf. Information Systems (ICIS 06)*, Association for Information Systems, 2006, p. 38.

74. F. Heylighen, "Why is Open Access Development so Successful? Stigmergic Organization and the Economics of Information," *Open Source Jahrbuch 2007*, Lehmanns Media, 2007, p. 165-180.

75. S.R. Hiltz and M. Turoff, "Structuring Computer-Mediated Communication Systems to Avoid Information Overload," *Communications of the ACM*, vol. 28, no. 7, 1985, pp. 680–689.

76. J. Huh, L. Jones, T. Erickson, W.A. Kellogg, R.K.E. Bellamy, and J.C. Thomas, "Blogcentral: The Role of Internal Blogs at Work," *Proc. Conf. Human Factors in Computing Systems (CHI 07)*, ACM, 2007, pp. 2447–2452.

77. S. Hupfer, L.T. Cheng, S. Ross, and J. Patterson, "Introducing Collaboration into an Application Development Environment," *Proc. Conf. Computer Supported Cooperative Work (CSCW 04)*, ACM, 2004, pp. 21-24.

78. C.W. Hsu, C.C. Chang, and C.J Lin, "A Practical Guide to Support Vector Classification," *Technical Report, Department of Computer Science, National Taiwan University*, 2003, http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf.

79. T. Joachims, *Learning to Classify Text using Support Vector Machines: Methods, Theory and Algorithms*, Kluwer Academic Publishers, 2002, p. 40.

80. B.K. Kasi and A. Sarma, "Cassandra: Proactive Conflict Minimization through Optimized Task Scheduling," *Proc. 35th Int'l Conf. Software Eng. (ICSE 13)*, IEEE Press, 2013, pp. 732-741.

81. K. Kellogg, W. Orlikowski, and J. Yates. "Life in the Trading Zone: Structuring Coordination across Boundaries in Postbureaucratic Organizations," *Organization Science*, vol. 17, no. 1, 2006, pp. 22–44.

82. M. Kersten and G.C. Murphy, “Mylar: a Degree-of-Interest Model for IDEs,” *Proc. 4th Int’l Conf. Aspect-Oriented Software Development (AOSE 05)*, ACM, 2005, pp. 159-168.

83. M. Kersten and G.C. Murphy, “Using Task Context to Improve Programmer Productivity,” *Proc. 14th Int’l Symposium Foundations of Software Engineering (FSE 06)*, ACM, 2006, pp. 1-11.

84. Z.U.R. Kiani and A. Riaz, “Measuring Awareness in Cross-Team Collaborations--Distance Matters,” *Proc. 8th Int’l Conf. Global Software Engineering (ICGSE 13)*, IEEE, 2013, pp. 71-79.

85. A. Ko, R. DeLine, and G. Venolia, “Information Needs in Collocated Software Development Teams,” *Proc. 29th Int’l Conf. Software Eng. (ICSE ‘07)*, IEEE CS, 2007, pp. 344–353.

86. S.B. Kotsiantis, “Supervised Machine Learning: a Review of Classification Techniques,” *Informatica*, vol. 31, 2007, pp. 249-268.

87. R. Kraut and L. Streeter, “Coordination in Software Development,” *Communications of the ACM*, vol. 38, no. 3, 1995, pp. 69-81.

88. K. Krippendorff, *Content Analysis: An Introduction to Its Methodology*, SAGE Publications, 2003.

89. I. Kwan, M. Cataldo, and D. Damian, “Conway's Law Revisited: The Evidence for a Task-based Perspective,” *IEEE Software*, vol. 29, no. 1, 2012, pp. 90-93.

90. I. Kwan and D. Damian, “Extending Socio-Technical Congruence with Awareness Relationships,” *Proc. 4th Int’l Workshop Social Software Engineering (SSE 11)*, ACM, 2011, pp. 23-30.

91. I. Kwan, A. Schröter, and D. Damian, “A Weighted Congruence Measure,” *Workshop Socio-Technical Congruence (STC 09)*, 2009.

92. I. Kwan, A. Schröter, and D. Damian, “Does Socio-Technical Congruence have an Effect on Software Build Success? A Study of Coordination in a Software Project,” *IEEE Transactions on Software Engineering*, vol. 37, no. 3, 2011, pp. 307-324.

93. F. Lanubile, C. Ebert, R. Prikladnicki, and A. Vizcaíno, “Collaboration Tools for Global Software Engineering,” *IEEE Software*, vol. 27, no. 2, 2010, pp. 52–55.

94. R. Lotufo, Z. Malik, and K. Czarnecki, “Modelling the ‘Hurried’ Bug Report Reading Process to Summarize Bug Reports,” *Proc. 28th Int’l Conf. Software Maintenance (ICSM 12)*, IEEE, 2012, pp. 430-439.

95. W. Maalej and M. Robillard, “Patterns of Knowledge in API Reference Documentation,” *IEEE Transactions on Software Engineering*, vol. 99, no. 1, 2013, p. 1.

96. J. Marlow, L. Dabbish, and J. Herbsleb, “Impression Formation in Online Peer Production: Activity Traces and Personal Profiles in Github,” *Proc. Conf. Computer Supported Cooperative Work (CSCW 13)*, ACM, 2013, pp. 117–128.

97. S. Minto and G.C. Murphy, “Recommending Emergent Teams,” Proc. 4th Int’l Workshop Mining Software Repositories (MSR 07), IEEE, 2007, p. 5.

98. A. Mockus, R.T. Fielding, and J. Herbsleb, “A Case Study of Open Source Software Development: The Apache Server,” *Proc. Int’l Conf. Software Eng. (ICSE 00)*, IEEE, 2000, pp. 263-272.

99. G.C. Murphy and E. Murphy-Hill, “What is Trust in a Recommender for Software Development?” *Proc. 2nd Int’l Workshop Recommendation Systems for Software Eng. (RSSE 10)*, ACM, 2010, pp. 57-58.

100. G. Murray, “Summarizing Spoken and Written Conversations,” *Proc. Conf. Empirical Methods in Natural Language Processing (EMNLP 08)*, Association for Computational Linguistics, 2008, pp. 773-782.

101. M.R. Nelson, “We have the Information You Want, but Getting it Will Cost You!: Held Hostage by Information Overload,” *Crossroads*, vol. 1, no. 1, 1994, pp. 11-15.

102. A. Nguyen-Duc and D.S. Cruzes, “Coordination of Software Development Teams across Organizational Boundary--An Exploratory Study,” *Proc. 8th Int’l Conf. Global Software Eng. (ICGSE 13)*, IEEE, 2013, pp. 216-225.

103. T. Nguyen, T. Wolf, and D. Damian, “Global Software Development and Delay: Does Distance Still Matter?” *Proc. Intl Conf. Global Software Engineering (ICGSE 08)*, IEEE, 2008, pages 45–54.

104. C. O’Reilly, P. Morrow, and D. Bustard, “Improving Conflict Detection in Optimistic Concurrency Control Models,” *Proc. 11th Int’l Workshop Software Configuration Management (SCM 03)*, Springer Berlin Heidelberg, 2003, pp. 191-205.

105. D.L. Parnas, “On the Criteria to be Used in Decomposing Systems into Modules,” *Communications of the ACM*, vol. 15, no. 12, 1972, pp. 1053-1058.

106. D.E. Perry, N.A. Staudenmayer, and L.G. Votta, “People, Organizations, and Process Improvement,” *IEEE Software*, vol. 11, no. 4, July 1994, pp. 36-45.

107. J. Portillo-Rodríguez, A. Vizcaíno, M. Piattini, and S. Beecham, “Tools Used in Global Software Engineering: A Systematic Mapping Review,” *Information and Software Technology*, vol. 54, no. 7, 2012, pp. 663-685.

108. S. Rastkar and G. Murphy, "Summarizing software artifacts: a case study of bug reports," *Proc. 32nd Int'l Conf. Software Eng. (ICSE 10)*, ACM, 2010, pp. 505-514.

109. D. Redmiles, B. Al-Ani, T. Hildenbrand, S. Quirk, A. Sarma, R. Silveira, S. Filho, C. de Souza, and E. Trainer, "Continuous Coordination a New Paradigm to Support Globally Distributed Software Development Projects," *Wirtschaftsinformatik*, vol. 49, 2007, pp. 28–38.

110. A. Sarma, L. Maccherone, P. Wagstrom, and J. Herbsleb, "Tesseract: Interactive Visual Exploration of Sociotechnical Relationships in Software Development," *Proc. 31st Int'l Conf. Software Eng. (ICSE 09)*, IEEE, 2009, pp. 23-33.

111. A. Sarma, Z. Noroozi, and A. van der Hoek, "Palantír: Raising Awareness among Configuration Management Workspaces," *Proc. 25th Int'l Conf. Software Eng. (ICSE 03)*, IEEE, 2003, pp. 444-454.

112. A. Sarma, D. Redmiles, and A. van der Hoek, "Palantir: Early Detection of Development Conflicts Arising from Parallel Code Changes," *IEEE Transactions on Software Engineering*, vol. 38, no. 4, 2012, pp. 889-908.

113. A. Sarma and A. Van Der Hoek, "Towards Awareness in the Large," *Proc. Int'l Conf. Global Software Engineering (ICGSE 06)*, IEEE, 2006, pp. 127-131.

114. A. Sarma, A. Van der Hoek, and D. Redmiles, "The Coordination Pyramid: A Perspective on the State of the Art in Coordination Technology," *Computer*, 2010, doi=10.1109/MC.2010.90.

115. S. Sawyer, "Software Development Teams," *Communications of the ACM*, vol. 47, no. 12, 2004, pp. 95-99.

116. W. Scacchi, "Free/Open Source Software Development: Recent Research Results and Methods," *Advances in Computers*, vol. 69, 2007, pp. 243-295.

117. A. Schröter, J. Aranda, D. Damian, and I. Kwan, "To Talk or Not to Talk: Factors that Influence Communication Around Changesets," *Proc. Conf. Computer Supported Cooperative Work (CSCW 12)*, ACM, 2012, pp. 1317-1326.

118. T. Schümmer and J.M. Haake, "Supporting Distributed Software Development by Modes of Collaboration," *Proc. 7th Conf. European Conference Computer Supported Cooperative Work (ECSCW 01)*, Kluwer Academic Publishers, 2001, pp. 79-98.

119. F. Servant, J.A. Jones, and A. Van Der Hoek, "CASI: Preventing Indirect Conflicts Through a Live Visualization," *Proc. Workshop Cooperative and Human Aspects of Software Engineering (CHASE 10)*, ACM, 2010, pp. 39-46.

120. M.E. Sosa, S.D. Eppinger, and C.M. Rowles, "The Misalignment Of Product Architecture And Organizational Structure In Complex Product Development," *Management Science*, vol. 50, no. 12, 2004, pp. 1674-1689.

121. H.A. Simon, "The Architecture of Complexity," *Proceedings of the American Philosophical Society*, vol. 106, no. 6, 1962, pp. 467-482.

122. S.L. Star, "The Structure of Ill-Structured Solutions: Boundary Objects and Heterogeneous Distributed Problem Solving," *Distributed Artificial Intelligence*, vol. 2, 1989, pp. 37-54.

123. S.L. Star and J.R. Griesemer, "Institutional Ecology, 'Translations' and Boundary Objects: Amateurs and Professionals in Berkeley's Museum of Vertebrate Zoology," *Social Studies of Science*, vol. 19, no. 3, 1989, pp. 397–420.

124. W.P. Stevens, J.G. Meyers, and L.L. Constantine, "Structured Design," *IBM Systems Journal*, vol. 13, no. 2, 1974, pp. 115–139.

125. M.A. Storey, L.T. Cheng, I. Bull, and P. Rigby, "Shared Waypoints and Social Tagging to Support Collaboration in Software Development," *Proc. 20th Conf. Computer Supported Cooperative Work (CSCW 06)*, ACM, 2006, pp. 195-198.

126. M.A. Storey, J. Ryall, R.I. Bull, D. Myers, and J. Singer, "Todo or to Bug: Exploring how Task Annotations Play a Role in the Work Practices of Software Developers," *Proc. 30th Int'l Conf. Software Eng. (ICSE 08)*, AMC, 2008, pp. 251–260.

127. P. Sullivan, "Information Overload: Keeping Current without Being Overwhelmed," *Science & Technology Libraries*, vol. 25, no. 1-2, 2004, pp. 109-125.

128. C. Treude and M.A. Storey. "Awareness 2.0: Staying Aware of Projects, Developers and Tasks Using Dashboards and Feeds," *Proc. 32nd Int'l Conf. Software Eng. (ICSE 10)*, IEEE, 2010, pp. 365-374.

129. C. Treude and M.A. Storey. "How Tagging Helps Bridge the Gap Between Social and Technical Aspects in Software Development," *Proc. 31st Int'l Conf. Software Eng. (ICSE 09)*, IEEE CS, 2009, pp. 12-22.

130. C. Treude and M.A. Storey, "Work item tagging: Communicating concerns in collaborative software development," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, 2012, pp. 19-34.

131. G. Valetto, S. Chulani, and C. Williams, "Balancing the Value and Risk of Socio-Technical Congruence," *Workshop Social Technical Congruence (STC 08)*, 2008.

132. G. Valetto, K. Blincoe, and S. Goggins, "Actionable Identification of Emergent Teams in Software Development Virtual Organizations," *Proc. 3rd Int'l Workshop Recommendation Systems for Software Engineering (RSSE 12)*, IEEE, 2012, pp. 11-15.

133. B. van Gameren, K. Dullemond, and R. van Solingen, "Auto-Erecting Virtual Office Walls," *Proc. 8th Int'l Conf. Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 12)*, IEEE, 2012, pp. 391-397.

134. B. van Gameren, R. van Solingen, and K. Dullemond, "Auto-Erecting Virtual Office Walls A Controlled Experiment," *Proc. 8th Int'l Conf. Global Software Engineering (ICGSE 13)*, IEEE, 2013, pp. 206-215.

135. S. Wang, D. Lo, and L. Jiang, "Inferring Semantically Related Software Terms and Their Taxonomy by Leveraging Collaborative Tagging," Proc. Int'l Conf. Software Maintenance (ICSM 12), IEEE, 2012, pp. 604-607.

136. X. Wang, I. Kuzmickaja, K. Stol, P. Abrahamsson, and B. Fitzgerald, "Microblogging in Open Source Software Development: The Case of Drupal Using Twitter," *IEEE Software*, 2013.

137. J. Wloka, B. Ryder, F. Tip, and X. Ren, "Safe-Commit Analysis to Facilitate Team Software Development," *Proc. 31st Int'l Conf. Software Eng. (ICSE 09)*, IEEE CS, 2009, pp. 507–517.

138. T. Wolf, A. Schröter, D. Damian, L.D. Panjer, and T.H. Nguyen, "Mining Task-Based Social Networks to Explore Collaboration in Software Teams," *IEEE Software*, vol. 26, no. 1, 2009, pp. 58-66.

139. S. Wong, Y. Cai, G. Valetto, G. Simeonov, and K. Sethi, "Design Rule Hierarchies and Parallelism in Software Development Tasks," *Proc. 24th Int'l Conf. Automated Software Eng. (ASE 09)*, IEEE CS, 2009, pp. 197–208.

140. X. Xia, D. Lo, X. Wang, and B. Zhou, "Tag recommendation in software information sites," *Proc. 10th Int'l Workshop Mining Software Repositories (MSR 13)*, IEEE Press, 2013, pp. 287-296.

141. 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," *Proc. Int'l Workshop Recommendation Systems for Software Eng. (RSSE 08)*, ACM, 2008, pp. 21–25.

142. W. Xiao, C. Chi, and M. Yang, "On-line Collaborative Software Development via Wiki," *Proc. Int'l Symposium Wikis and Open Collaboration (WikiSym 07)*, ACM, 2007, pp. 177–183.

143. Y. Ye and K. Kishida, “Toward an Understanding of the Motivation of Open Source Software Developers,” *Proc. 25th Int’l Conf. Software Eng. (ICSE 03)*, IEEE, 2003, pp. 419-429.

144. J. Yew, F. Gibson, and S. Teasley, “Learning by Tagging: Group Knowledge Formation in a Self-Organizing Learning Community,” *Proc. 7th Int’l Conf. Learning Sciences (ICLS 06)*, International Society of the Learning Sciences, 2006, pp. 1010-1011.

145. E. Zangerle, W. Gassler, and G. Specht, “Using Tag Recommendations to Homogenize Folksonomies in Microblogging Environments,” *Proc. 3rd Int’l Conf Social Informatics (SocInfo 11)*, Springer Berlin Heidelberg, 2011, pp. 113–126.

# Vita

Kelly Blincoe received a BE in Computer Engineering from Villanova University in 2004 and an MS in Information Science from Pennsylvania State University in 2008. She received an MS in Computer Science from Drexel University in 2011 where she served as President of the Computer Science Graduate Student Council for two years. She studied as a visiting research student at University of Victoria in Fall 2013. Kelly is a member of the Association for Computing Machinery (ACM), the Institute for Electrical and Electronics Engineer (IEEE), and Upsilon Pi Epsilon (UPE), the international honor society for computing and information disciplines. She won the ACM Student Research Competition (SRC) held at the International Conference of Software Engineering (ICSE) in 2012. Her publications include:

K. Blincoe, G. Valetto, and S. Goggins, "Proximity: a Measure to Quantify the Need for Developers Coordination," *Proc. Conf. Computer Supported Cooperative Work* (CSCW 12), ACM, 2012, pp.1351-1360.

K. Blincoe, G. Valetto, and D. Damian, "Do all task dependencies require coordination? The role of task properties in identifying critical coordination needs in software projects," *Proc. 9th Joint Meeting on Foundations of Software Engineering* (ESEC/FSE 13), ACM, 2013, pp. 213-223.

K. Blincoe, G. Valetto, and D. Damian, "Uncovering critical coordination requirements through content analysis," *Proc. Int'l Workshop Social Software Eng.* (SSE 13), ACM, 2013, pp. 1-4.

A. Borici, K. Blincoe, A. Schroeter, G. Valetto, and D. Damian, "ProxiScientia: Toward Real-Time Visualization of Task and Developer Dependencies in Collaborating Software Development Teams," *Proc. 5th Int'l. Workshop Cooperative and Human Aspects of Software Engineering* (CHASE 12), IEEE, 2012, pp. 5-11.

S. Goggins, G. Valetto, C. Mascaro, and K. Blincoe, "Creating A model of the Dynamics of Socio-Technical Groups," *User Modeling and User-Adapted Interaction*, Springer, 2012, pp. 1-35.