Improving Social Relations Between Developers By Leveraging The Concept Of Socio-Technical Congruence

by

Adrian Schröter
B.Sc., Saarland University, 2006
M.Sc., Saarland University, 2007

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

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

Improving Social Relations Between Developers By Leveraging The Concept Of Socio-Technical Congruence

by

Adrian Schröter
B.Sc., Saarland University, 2006
M.Sc., Saarland University, 2007

Supervisory Committee

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

Dr. Hausi A. Müller, Departmental Member
(Department of Computer Science)

Dr. Issa Traoré, Outside Member
(Department of Electrical and Computer Engineering)

Supervisory Committee

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

Dr. Hausi A. Müller, Departmental Member
(Department of Computer Science)

Dr. Issa Traoré, Outside Member
(Department of Electrical and Computer Engineering)

Abstract

Efficient coordination among software developers is a key aspect in producing high quality software on time and on budget. Several factors, such as team distribution and the structure of the organization developing the software, can increase the level of coordination difficulty. However, the problem runs deeper as it is often unclear which developers should coordinate their work.

In this dissertation, we propose leveraging the concept of socio-technical congruence (which contrasts coordination needs with actual coordination) to improve the social interactions among developers by devising an approach and its implementation into a recommender system that identifies relevant coordinators. Our unit of analysis is the integration build, whose outcome represents the quality of coordination. After development, this approach was applied in a number of IBM Rational Team Concert development team case studies, as well as a large student project at the University of Victoria, Canada, and Aalto University, Finland.

Since each software product is just the latest integration build, it is of utmost importance for the industry to ensure a failure free build. While developing an approach to improve coordination among software developers we uncovered that unmet coordination needs, as well as the communication structure in a team, have a significant influence on build outcome.

Contents

List of Tables

List of Figures

Acknowledgements

First and foremost, I would like to thank my family for their constant support and for encouraging me to follow my academic path which began in Saarbrücken, Germany and lead me to Victoria, Canada. Not only was I given the opportunity to work with a top notch research group which was headed by my mentor, Daniela Damian, but I also found Jennifer, the love of my life. She not only sustained me throughout my time as a PhD student, but also helped me to make sense of a whole new world.

Furthermore, I owe a great deal of gratitude to my academic advisors and mentors from my previous University, specifically Thomas Zimmermann and Andreas Zeller, for they opened my mind to the field of empirical software engineering. Of course, I especially would like to thank all the friends I made during my time in Victoria. Especially those in the SEGaL group with Sabrina and Irwin who always had much needed advice, and Thanh with his easy going style, showing me that just about everything has an upside. Not to forget, the new generation of SEGaLs: Indira, Germán, Arber, Eirini, Eric, Alessia, and Jorge who kept me going until the very end.

Dedication

To all graduate students by heart and name.

Chapter 1

Introduction

The software industry, often visible through big companies such as Microsoft, Google, IBM, Dell, Apple, Oracle, and SAP, represents several hundred billion dollars of employment and profit a year. For example, according to the US Census, the US software industry produced a total revenue of 103.7 billion USD in 2002.[1] Similar to many engineering companies, those in the software industry strive to optimize their engineering processes in an attempt to produce higher quality software in a shorter period of time.

Throughout the world, software engineering researchers have dedicated countless hours to improving the manner in which software is developed. Several fields that are not directly aimed at increasing productivity, such as developing better programming languages [45], smarter compilers [62], and better educational methods to teach algorithms and data structures [21] contribute indirectly. Other fields are more directly focused on productivity. Among them are research in software processes [91], effort estimation [11, 78], and software failure prediction [76].

The vast body of knowledge collected in an attempt to improve the software engineering process is strongly biased towards analyzing the technical side: supporting coding activities (e.g. [3, 75]) and analyzing source code to improve quality [81, 114]. Since producing source code is the main objective of software developers, optimizing the coding aspect [3, 75] as well as analyzing the produced code for issues [80, 101] is important.

Others have focused on the individuals who produce the code. Specifically, studying their behaviour around coding activities [68], how they communicate [41, 63], and how developer relations relate to productivity [41] and quality [1, 111]. As in the former case, there is much merit in focusing on the developer. In the end, the developer implements the

[1] http://www.census.gov/prod/ec02/ec0251i06.pdf last visited May 10th, 2012

features that a software system consists of, and inevitably the developer introduces errors into the code base.

Both studying the human aspect and studying the technical aspect yielded numerous useful results. For example, on the human side, it appears that the organizational distance between developers is a good predictor of failure on the file level [82], and on the technical side similar changes that are timely close are good failure predictors [60].

To truly be able to optimize the software engineering process a more holistic view is needed to bring together both the technical and social aspects. As stated by Conway [20], one way to merge these mutually influencing aspects is to use the concept of *socio-technical congruence* in software engineering, which was first formalized by Cataldo et al. [19]. They proposed to overlay networks constructed from social (i.e. who communicates with whom) and technical (i.e. whose code depends on whose source code) dependencies to obtain an overview of a project's social and technical interdependencies and derive insight through the miss-match between the two networks.

Socio-technical congruence forms a great basis to leverage several digitally recorded data treasures to generate useful and actionable information. Patterns of developer pairs have shown that when developers share a technical dependency, but are not talking to each other, they are endangering the upcoming software build. Furthermore, in a student project, we found that certain issues experienced during development can be traced back to code dependencies that could have been detected in real time.

To complement the research that studied the relationship between socio-technical congruence and performance, we focus on *build outcome* as a metric for software quality. Build outcome is rarely considered when studying software quality, because it is a course measure that often indicates multiple issues rather than a single specific one. Studying build outcome is important because build success is fundamental in creating a product that can be shipped to a customer. A successful build towards the end of the release cycle often is the only indicator of customer acceptance with respect to requested features and their stability. Hence, build success is of utmost importance to a business, as it forms the very product the business hopes to sell. Therefore, the two guiding research questions we address in this dissertation are:

RQ 1: Does Socio-Technical Congruence influence build success?

RQ 2: Can Socio-Technical Networks be leveraged to generate recommendations to improve build success?

We are using a mixed methods approach to explore these two research questions. For

RQ 1 we employ data mining techniques by studying the artifacts, such as task discussions and source code changes, of a large industrial software project. **RQ 2** requires both quantitative and qualitative analysis methods. To find statistically relevant recommendations we employ data mining techniques, but to explore the usefulness and acceptance of such recommendations we make use of questionnaires, interviews, and observational studies.

1.1 Problem Statement

Socio-technical congruence, as defined by Cataldo et al. [19], describes a measure that outlines the extent by which the technical dependencies in the product are matched by social interactions among developers affected by these technical dependencies. This directly follows Conway's observations [20], that the communication structure of any given organization dictates the underlying technical dependencies. In software engineering, this roughly translates to the idea that the communication flow within software teams needs to match the module dependencies described by the software architecture.

This idea shows great promise when applied to software repositories, such as versioning archives and issue trackers or other recorded communication. Cataldo et al. [17, 19], as well as other researchers [33, 106], found that the higher the satisfaction of the technical dependencies with social interaction is, the higher the productivity and to some extent the software quality [8, 65, 66] becomes. The ability to extract useful socio-technical measures from archives in an automated fashion enables the application to any software project that captures development data electronically.

However, we see three major issues with the concept of socio-technical congruence as it is currently used:

- The socio-technical congruence measure itself does not give much indication with respect to how to improve the overall situation other than suggesting that people should talk to each other in the event that they share a technical dependency.
- The idea of achieving high congruence is based on the notion that it is important to communicate along all technical dependencies, which is not necessarily true.
- The analysis of socio-technical congruence can only be done post-mortem, which although valuable in a retrospective, does not help to improve productivity or quality in an ongoing project.

The issue of imbalance between technical and social relationships between developers is related to the problem of not knowing how to improve the socio-technical congruence other than by pointing out the technical relationships between developers that did not communicate with each other. Given enough resources and time, every technical dependency can be satisfied. However, this might run the risk of decreasing the productivity by introducing too many interruptions.

Over-communication of technical dependencies might arise from the underlying assumption that every technical dependency warrants the dependent developers to communicate with each other. We are not solely referring to the ability of developers to read environment traces [12], but also to the fact that some changes are either not meant to be communicated or that the system architecture was designed to accommodate certain changes (think of optimizations) that should not affect other developers.

To fully leverage the concept of socio-technical congruence it is important to act on it. The current concept is only shown to relate to performance and quality post-mortem. To truly unlock the potential of the socio-technical congruence concept it needs to be extended so that it can make on demand recommendations to improve congruence.

1.2 Dissertation Focus

In this dissertation, we focus on addressing the aforementioned issues in two ways:

What technical dependencies need to be met with communication? Although the recommendation to have every developer talk to every other developer about their work seems to be the easiest solution to gaining perfect socio-technical congruence coverage, as previously stated, it could decrease productivity due to the heavy overhead caused by constant communication. To address this issue, we seek out which technical dependencies exist among developers and go one step further to try to find the technical dependencies that when not accompanied by communication are the most harmful to the project.

Instead of focusing on recommending changes to the source code to remove technical dependencies we focus on improving the communication among developers. Because changes to the technical dependencies would partly imply having to re-architect the product, which would be both time intensive and risky, we focus on optimizing the social interactions among developers. Additionally, as customers rarely derive any tangible benefits from re-architecting a product, there is little willingness

to pay for this type of work, unless the re-architecting is the goal as it is the case when porting a legacy system to a new platform [61].

How to make socio-technical congruence actionable? Although it is possible that socio-technical congruence can be continuously computed and the previously mentioned strategies can be applied in real time, they all take a more project-centered perspective. To support developers to engage in communication when necessary, they need to be informed of potential issues that may arise with respect to socio-technical congruence. Building upon the concept of proximity, proposed by Blincoe et al. [9], we study in depth the development interactions of a large student project at the University of Victoria, Canada, and Aalto University, Finland, and the relation between issues and their fine grained real-time code dependencies.

Furthermore, as Murphy et al. [79] pointed out, users of automated recommendation systems need to trust the system, otherwise they will ignore it. This is especially true when continuously reporting information to developers and trying to steer them in a specific direction. Therefore, we investigate what the daily focus of developers is when it comes to communication to gage if the level of recommendation provided by most methods derived or related to socio-technical congruence might be successful.

1.3 Contributions

This dissertation has two major contributions: (1) an approach to improve social interactions among software developers that leverages the concept of socio-technical congruence and (2) the findings that coordination, both in terms of structure and absence, negatively influences build success.

1.3.1 Approach

The first contribution of this dissertation demonstrates that socio-technical congruence can be used to create recommendations to prevent build failures by improving the social interactions among software developers. We derived the approach presented in Chapter 7 through two case studies that showed that social and socio-technical networks predict build outcome (cf. Chapters 5 and 6). In a follow up study, we demonstrate that we could generate relevant recommendations that exhibit a strong influence on build success (cf. Chapter 5). In Chapters 9 and 10, we demonstrated the usefulness of the information with respect to

whether experts expect the level of recommendations to be of use, as well as if these recommendations could be produced in real time and potentially prevent issues from arising. The approach we present in Chapter 7 consists of five steps:

1. Define scope of interest
2. Define outcome metric
3. Build social networks
4. Build technical networks
5. Generate actionable insights

This approach enables us to provide developers with recommendations that point them to engage in communication with other developer they share technical dependencies with. For example, we found instances where developers, who share a technical dependency but did not communicate, can increase the likelihood of a build failure by more than 80%.

1.3.2 Empirical Findings

The studies we conducted to motivate and appraise the approach yielded separate research findings extending the body of knowledge of coordination in software development teams. Our approach was inspired by the effect of communication structures on build success that we found with our first study (cf. Chapter 5). Furthermore, we investigated the coordination gaps highlighted by technical dependencies among software developers and their effect on build success (cf. Chapter 6).

In the first study, to the best of our knowledge, we were the first to show a definitive relationship between coordination structures of a development team and build outcome (Chapter 5). We further corroborated this evidence by demonstrating that unmet coordination needs have a negative effect on build outcome as well (cf. Chapter 6). Then, we presented evidence that specific unmet coordination needs that reoccur over time have a high chance of inducing a build failure (cf. Chapter 8). While investigating whether developers would accept recommendations produced by our approach, we found that the development process influences how concerned developers are about individual changes (cf. Chapter 9). Finally, in a case study of a large student project at the University of Victoria, Canada, and Aalto University, Finland, we showed that data needed to compute socio-technical network could be collected in real time while a developer edits her source code (cf. Chapter 10).

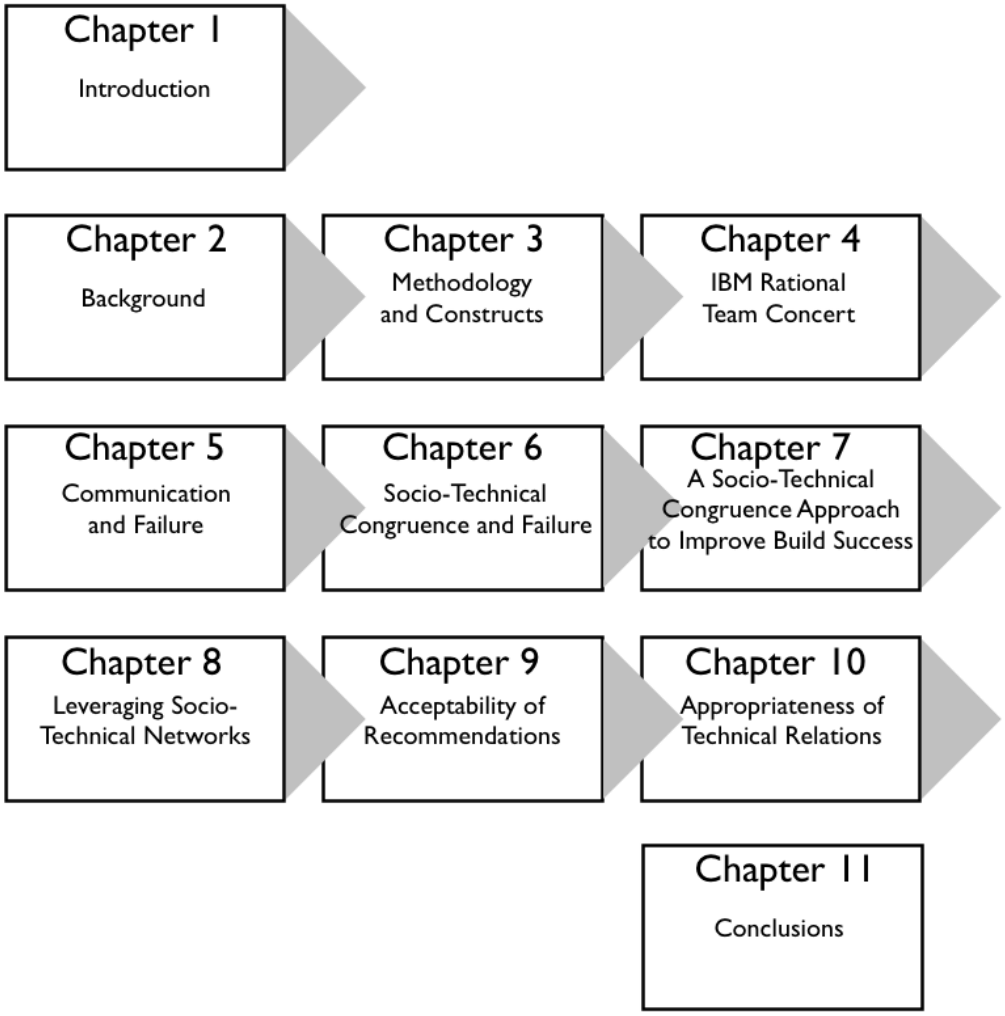


Figure 1.1: Dissertation overview

1.4 Overview

This dissertation is divided into three parts (second to fourth row in Figure 1.1). In part one, we motivate our research by reviewing related work in Chapter 2. We delve into presenting our overarching methodology with explanations of frequently used constructs and analysis methods in Chapter 3, followed by presenting IBM Rational Team Concert (RTC) as well as some key factors of the development team (cf. Chapter 4).

Part two presents two studies (cf. Chapters 5 and 6) that build the foundation for our approach, which we formulate in Chapter 7. In those two studies, we investigated the relationship between social networks, build success, and socio-technical networks, specifically unmet coordination needs, and build success.

Knowing that the social network might lend itself to manipulations with positive effects with respect to build success, we study the development history of the IBM Rational Team Concert development team for recurring patterns of developer pairs that do not coordinate and their statistical relationship to build success (cf. Chapter 8). We continue by presenting a study in Chapter 9, investigating whether the recommendations resulting from those patterns are of use to developers and when the best time to present such recommendations is. Before concluding this dissertation we discuss how our approach to leverage socio-technical congruence (cf. Chapter 11) is supported by the evidence uncovered through our studies. In particular, we present a study in Chapter 10 which provides evidence that our approach can generate recommendations that could have prevented build failures. Chapter 11 outlines the conclusions we derive from our work and points out avenues for future work.

Part I

Foundations

Chapter 2

Background

In this chapter, we provide an overview of the five areas that are relevant to the research conducted for this dissertation: (1) the research on software builds, (2) the research on coordination in software development teams, (3) the research around the concept of socio-technical congruence, (4) failure prediction using social networks, and (5) recommender systems in software engineering.

2.1 Build Outcome

Although software builds are important because the final product is just the latest acceptable build, research in software builds focuses mainly on tools and processes that support the build process. Software products supporting builds are often used to speed up the build process and the execution of all test cases to obtain an assessment of the quality of the build [72]. Similarly, processes that focus on supporting software builds are predominantly dealing with issues of obtaining all required code changes from the different development teams and integrating this code into a final build as fast as possible without introducing additional issues.

The issue that shifts into focus once the actual process of creating the build is thoroughly optimized is to gain an idea of whether a build will fail or succeed before the build process is started. If a project reaches a certain size, meaning the test suite grows considerably in size, the build process can take several hours just to run the whole test suite. To determine whether developers need to stay in order to apply quick fixes such that the product can be shipped or handed over to a team starting their work in a different time zone becomes important.

The following section reviews literature with respect to coordination and integration with builds representing a form of integration. We complement that review with research that studies the relationship between social networks and software development.

2.1.1 Communication, Coordination and Integration

The relationship between communication, coordination, and project outcome, has been studied for a long time in the area of computer-supported cooperative work. More recently, the domain of software and distributed software development showed increased interest as well.

Communication plays an important role in work groups with high coordination needs and the quality of communication has been found to be predictive of project success [23, 64]. The dynamic nature of work dependencies in software development makes collaboration highly volatile [16], consequently affecting a team's ability to effectively communicate and coordinate. Additional difficulties emerge in distributed teams, where team membership and work dependencies become even more invisible [25]. Moreover, team communication patterns are significantly affected by distance [55]. Maintaining awareness [97] becomes even more difficult when developers work in geographically remote environments. Communication structures that include key contact people at each site are effective coordination strategies when maintaining personal cross-site relationships is challenging [55].

With respect to the role of effective coordination in project success, early studies indicate the issues that software development teams face on large projects [23]. A study by Herbsleb et al. [51] showed that Conway's law is also applicable for the coordination within development teams, supporting the influence of coordination on software projects. Kraut et al. [64] showed that software projects are greatly influenced by the quality of coordination of development teams. More recently, a theory of coordination has been proposed that accounts for the influence of coordination on different project metrics, such as rework and defects [54].

The importance of communication in successful coordination is also well documented and makes the study of communication structures important. For example, Fussell et al. [38] found that communication amount as well as tactics were linked to the ability to effectively coordinate in work groups. In software development, others showed that communication problems lead to further problems during the activity of subsystem integration [29, 43]. Coordination conceptualized via communication has also been studied more generally in relation to project success: factors such as "harmony" [104], communication structure [89], and communication frequency [42], was related to project success.

The difficulty in studying failed integration in relation to communication lies in capturing and quantifying information about communication in teams that have a well-defined coordination goal but dynamic patterns of interaction. In our work, we use the Jazz project data, which captures communication of project participants. This enables us to study the structure of the communication networks that emerged around code integrations, both in individual teams and within the entire project.

2.1.2 Can communication predict build failure?

Social network analysis has an extensive body of knowledge concerning analysis and its implications with respect to communication and the knowledge management processes [14, 36]. Griffin and Hauser [42] investigated social networks in manufacturing teams. They found that a higher connectivity between engineering and marketing increases the likelihood of a successful product. Similarly, Reagans and Zuckerman [92] related higher perceived outcomes to denser communication networks in a study of research and development teams.

Communication structure in particular – the topology of a communication network – has been studied in relation to coordination (e.g., [55, 57]), and a number of common measures of communication structure include network density, centrality and structural holes [36, 110].

Density reflects the ability to distribute knowledge [94] by measuring the extent to which all members in a team are connected to one another. Density has been studied, for example, in relation to coordination ease [55], coordination capability [57] and enhanced group identification [92].

Centrality measures indicate importance or prominence of actors in a social network. The most commonly used centrality measures include degree and betweenness centrality having different social implication. Centrality measures have been used to characterize and compare different communication networks constructed from email correspondence of W3C (WWW consortium) collaborating working groups developing new technical standards and architectures for the web [39]. Similarly, Hossain et al. [57] explored the correlation between centrality in email-based communication networks and coordination, and found betweenness to be the best measure for coordination. Betweenness is a measure of the extent to which a team member is positioned on the shortest path in between two other members. People in between are considered to be "actors in the middle" and are send as having more "interpersonal influence" in the network (e.g., [39, 57, 115]).

The structural holes measures are concerned with the degree to which there are missing links in between nodes and with the notion of redundancy in networks [14]. At the node level, structural holes are gaps between nodes in a social network. At the network level, people on either side of the hole have access to different flows of information [46], indicating that there is a diversity of information flow in the network. Structural holes have been used to measure social capital in relation to the performance of academic collaborators (e.g., [40]).

Most prediction models in software engineering to date mainly leverage source code related data and focus on predicting failing software components or failure inducing changes (e.g., [6, 59, 101, 115]). And only few studies, such as Hassan and Zhang [47], stepped away from predicting component failures and used statistical classifiers to predict integration outcome. In this dissertation, we want to extend the body of knowledge surrounding prediction models using communication data or focusing on build outcome by investigating how to improve communication among software developers to an effort to prevent build failures.

2.2 Coordination in Software Engineering Teams

In Section 2.1 we highlighted the connection between coordination and interaction. In this section, we extend this review by discussing work about coordination in software teams, as it is important to understand the coordination in teams in order to manipulate it to influence build outcome.

2.2.1 The Need for Coordination

Software is extremely complex because of the sheer number of dependencies [98]. Large software projects have a large number of components that interoperate with one another. Difficulty arises when changes must be made to the software, because a change in one component of the software often requires changes in dependent components [28]. Because a single person's knowledge of a system is specialized, as well as limited, that person is often unable to make the appropriate modifications to dependent components when a component is changed.

Coordination is defined as "integrating or linking together different parts of an organization to accomplish a collective set of tasks" [108]. In order to manage changes and maintain quality, developers must coordinate, and in software development, coordination is largely achieved by communicating with people who depend on the work that you do [64].

A successful software build can be viewed as the outcome of good coordination since the build requires the correct compilation of multiple dependent files of source code. A failed build, on the other hand, demotivates software developers [25, 56] and destabilizes the product [24]. While a failed build is not necessarily a disaster, it significantly slows down work while developers scramble to repair the issues. A build result thus serves as an indicator of the health of the software project up until that point in time.

Therefore, a developer should coordinate closely with individuals whose technical dependencies affect the work, in order to effectively build software. This brings forth the notion of aligning the technical structure and the social interactions [49], leading us to the foundation of socio-technical congruence.

2.2.2 Coordination in Software Teams

Research in software-engineering coordination has examined interactions among software developers [15, 73], how they acquire knowledge [32, 83], and how they cope with issues, including geographical separation [34, 52]. The ability to coordinate has been shown as an influential factor in customer satisfaction [64], and improves the capability to produce quality work [35].

Software developers spend much of their time communicating [88]. Because developers face problems when integrating different components from heterogeneous environments [93], they engage in direct or indirect communication, either to coordinate their activities, or to acquire knowledge of a particular aspect of the software [83]. Herbsleb, et al. examined the influence of coordination on integrating software modules through interviews [50], and found that processes, as well as the willingness to communicate directly, helped teams integrate software. De Souza et al. [27] found that implicit communication is important in order to avoid collaboration breakdowns and delays. Ko et al. [63] found that developers were identified as the main source of knowledge concerning code issues. Wolf et al. [111] used properties of social networks to predict the outcome of integrating the software parts within teams. This earlier work reiterates the notion that developers communicate heavily about technical matters.

Coordinating software teams becomes more difficult as the distance between people increases [53]. Studies of Microsoft [7, 82] show that distance between people that work together on a program determines the program's failure proneness. Differences in time zones can affect the number of defects in software projects [18].

Although distance has been identified as a challenge, advances in collaborative development environments are enabling people to overcome challenges of distance. One study of early RTC development shows that the task completion time is not as strongly affected by distance as in previous studies [84]. Technology that empowers distributed collaboration includes topic recommendations [15] and instant messaging [86]. Processes are adapting to the fast paced world of software development: the Eclipse way [37] emphasizes placing milestones at fixed intervals and community involvement. This increased focus on software builds warrants more support by research as we conduct it in this dissertation.

2.3 Socio-Technical Congruence

As previously mentioned, this dissertation explores to what extent we can leverage the concept of socio-technical congruence. Before we discuss the work conducted with respect to using the concept of socio-technical congruence to analyze software development teams and their performance, we explain the socio-technical congruence concept.

2.3.1 Socio-Technical Congruence Definitions

The literature exploring and using the concept of socio-technical congruence often relies on two interconnected definitions of socio-technical congruence. Originally defined by Cataldo et al. [19], socio-technical congruence was a single metric describing how much of the work dependencies between developers are covered by the communication between those developers. But the interest in socio-technical congruence took a broader view, and instead of focusing on the metric, the focus shifted to the underlying construct conceptualizing the different connections among developers. We now discuss the two commonly used approaches to infer socio-technical dependencies among developers, starting with the traditional definition initially presented by Cataldo et al. [19], followed by a more network centric definition.

Task Assignment and Dependency

Cataldo et al. [19] defined technical dependencies among developers as the multiplication of the matrix task assignment matrix (defining the assignment of a developer to a task) with the task dependency matrix (defining the dependencies among tasks) multiplied with the transpose of the task assignment matrix. The creation of separate matrices was motivated by the need to extract information from different data repositories. In the original

$$\underbrace{\begin{pmatrix}1 & 0 & 1 & 1\\ 0 & 0 & 0 & 1\\ 1 & 0 & 0 & 0\\ 0 & 1 & 0 & 1\end{pmatrix}}_{A} \times \underbrace{\begin{pmatrix}0 & 1 & 0 & 0\\ 1 & 0 & 1 & 0\\ 0 & 1 & 0 & 1\\ 0 & 0 & 1 & 0\end{pmatrix}}_{D} \times \underbrace{\begin{pmatrix}1 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\\ 1 & 0 & 0 & 0\\ 1 & 1 & 0 & 1\end{pmatrix}}_{A^T} = \begin{pmatrix}2 & 1 & 0 & 3\\ 1 & 0 & 0 & 0\\ 0 & 0 & 0 & 1\\ 3 & 0 & 1 & 0\end{pmatrix}$$

Figure 2.1: Calculating technical dependencies among developer using the task assignment and task dependency matrix.

study, conducted by Cataldo et al., task dependencies and task assignments were defined in different repositories requiring different approaches to extract the information. The matrix multiplication allows us to derive the developer interdependencies without requiring direct access to the data. Thus, two matrices need to be inferred from a data set: (1) task assignment matrix describing which developer is assigned to which task and (2) the task dependency matrix describing which tasks share dependencies.

Task Assignment Matrix The task assignment matrix dimension is the number of developers multiplied by the number of tasks. Each entry in the matrix denotes whether a given developer is assigned to a given task, this notation allows for more than one developer to be assigned to a task as well as one developer being assigned to multiple tasks. This information is inferred from task management systems such as BugZilla[1] or Jira[2] that show who is assigned to work on a given task.

Task Dependency Matrix The task dependency matrix dimension is the number of tasks multiplied by the number of tasks with each row and column representing all tasks. Each entry in the task dependency matrix indicates whether two tasks are dependent; note that nonzero entries refer to the existence of a dependency but not its strength. The task dependency matrix is populated by identifying the code written to finish a task, and infers dependencies among the various code changes implementing different tasks. For example, Cataldo et al. [19] defined two tasks to be dependent if the associated changes modify the same file.

Once the matrices for task assignment and task dependency are derived, we can compute the technical dependency among developers. Through a matrix multiplication of the

[1] `http://www.bugzilla.org`
[2] `http://www.atlassian.com/software/jira`

task assignment with the task dependency matrix we obtain a matrix describing on which task a developer depends. Further multiplying this matrix with the transposed task assignment matrix yields a developer-by-developer matrix that indicates which developers are dependent on each other's work through at least one task. Thus, the calculation of the technical dependency among developers follows the formula presented below:

$$A \times D \times A^{\mathrm{T}} = \text{Coordination Needs} \tag{2.1}$$

Figure 2.1 shows an example of how to derive the technical dependencies among developers given a task assignment and task dependency matrix. Following the formula presented in Equation 2.1, we multiply the task assignment matrix, the task dependency matrix, and the transposed task assignment matrix with the transposed task assignment matrix to obtain a matrix of dimension of number of developers by number of developer with each entry in the matrix greater than zero denoting a technical dependency between two developers. The resulting matrix is also referred to as the coordination needs matrix.

The technical dependency matrix obtained through the matrix multiplication described above needs to be contrasted with the actual coordination that happened during the project. For this purpose, Cataldo et al. [19] proposed the creation of a matrix recording whether two developers coordinate their work. Note that communication is often [19,31,33,66,106, 111] used as a proxy for coordination, relying on recorded communications found in email archives or task discussions in issue management systems. The congruence metric itself is the ratio between developers that have both a technical dependency and coordinated over the number of developers that have a technical dependency.

The actual coordination matrix depicts a social network with developers represented by nodes and coordination instances as edges. Similarly, the coordination needs matrix depicts a social network connecting developers when they share a technical dependency. Thus, another method is to approach socio-technical congruence through taking a more social networks analysis point of view and construct the two types of social networks directly as is discussed in the following section.

Social and Technical Networks

As seen in the previous section, the task dependency matrix depends on the changes made to the software. Therefore, it is often more simple to directly construct the coordination needs matrix, or the social network connecting developers, through technical dependencies drawn from the changes made to the system. This is possible because changes to a software

system are usually recorded in a source code repository, and each change belongs to a developer. Thus research [19, 31, 33, 66, 106] working with the socio-technical congruence concept with a social network view contrasts social and technical networks.

Technical Networks In Cataldo et al.'s [19] formulation of technical dependencies, task assignment and task dependency matrix are multiplied together. Since the task dependency matrix is inferred from the overlap in code modifications, such that tasks are accomplished by modifying the same source code files, the technical dependencies among developers can be directly inferred from a software repository. This more direct approach enables the construction of technical networks, connecting developers through the dependencies of the changes they made to a software project, without it ebbing necessary to access a task management system.

Social Networks The social network representation of the ongoing communication is exactly the same as the actual coordination matrix as described by Cataldo et al.'s [19] as the matrix is in fact a way of representing a network (also known as adjacency matrix).

The technical difficulties associated with this approach are that matching the social and technical networks as the usernames used for code repositories and task management can be different. This is especially an issue with open source development as it is less likely that processes demanding naming conventions of account names are going to be enforced [101].

2.3.2 Socio-Technical Congruence and Performance

Social-technical congruence, as originally observed by Conway [20], states that any product developed by an organization would inevitably mirror the organization's communication structure. From this starting point, Cataldo et al. [19] along with other researchers [31, 33, 106], investigated whether the lack of this reflection relates to changes in productivity by studying the overlap of communication among developers and their technical dependencies. The communication among developers represents the organizational communication structure whereas the technical dependencies between the work done by each developer represents the products organization. If the communication structure completely covers the work dependencies among developers, then developers accomplish their tasks faster mainly due to knowledge seeking and sharing [30]. For example, a developer can better accomplish their task if they are talking directly to co-workers that need to modify related code in order to avoid failures or because someone can help them to more clearly understand the

impact of the code they are about to modify.

The main performance criteria researchers investigated to measure the effect of socio-technical congruence is task completion time. For this purpose, Cataldo et al. [19] measures the congruence on a task basis and tests for the correlation between congruence and the time it took to resolve the task. Overall, Cataldo et al. [19] found that there was a statistically significant relation between the amount of congruence and a tasks resolution time, which was confirmed by other studies [33, 106].

2.4 Networks and Failure

Because we are investigating how to improve communication among software developers following their technical dependencies with each other, we offer an overview of the work that involves changes to source code that directly or indirectly indicates technical dependencies.

2.4.1 Artifact Networks

Using dependencies within a product one can construct a network of software artifacts that is connected via the dependencies. Artifacts that have direct dependencies in the case of source code are referred to as code peers. One interesting property of code peers is that in the case that a code peer exhibits a defect, the likelihood that the code artifact (whose peer contains a defect) will also have a defect itself increases [85].

From the notion of a code peer, and its influence on other peers, the idea of analyzing these network with respect to an artifact and its surrounding artifacts can be derived. In a first study, Zimmermann et al. [115] analyzed call dependencies of a single artifact and found measures characterizing those dependencies to be a good predictor for software defects.

In a follow up study, Zimmermann et al. [116] extended the influence of an artifacts peer by taking in to account the dependencies among an artifacts peers instead of focusing solely on an artifacts dependencies. This enables the application of network measures and social-network measures to characterize the ego network constructed around a software artifact. As it turns out, the predictive power of such a network is stronger than when one only considers dependencies between an artifact and its peers [116].

2.4.2 Technical Networks

To go from artifact networks to technical networks developers can be included in the already existing artifact network and thus be represented as a kind of artifact [90]. These two mode networks can be used for the same analysis that Zimmermann et al. [115, 116] performed by focusing on the software artifacts in order to predict the failure likelihood of each. Meneely et al. [74] use networks that consist only of developers, that within a given release, modified the same file. Social network measures extracted from these networks are able to predict whether a file contains a failure.

2.5 Recommendations in Software Engineering

In the software engineering community knowledge extracted from software repositories is usually brought to developers in the form of recommender systems. Because the goal of this dissertation is to create an approach forming the basis for a recommender system, we present recommender systems using the socio-technical congruence concept. Several recommender systems derived from the implication of socio-technical congruence described by Conway's Law [20] provide additional awareness to improve coordination among software development, especially in a distributed setting where coordination is most difficult [87]. In the following, we will describe five such awareness systems. We are aware that this list is not exhaustive. Nonetheless, we think that it presents a reasonable overview of awareness systems proposed by software engineering researchers.

Ariadne [105] provides awareness to developers by showing call dependencies between the code a developer is working on and the code that they are potentially affecting. This allows a developer to see which other developers they might need to coordinate with in order to avoid negatively impacting the developer's code.

Palantir [96] complements the dependencies among developers by providing the reverse awareness showing a developer what source code she is currently accessing in their workspace is affected by code changes submitted by co-workers. For example, Palantir indicates which source code files have been changed by other developers that are present in the current workspace and thus might hint at possible merge conflicts.

Tesseract [95] extends the concept of showing code dependencies among developers by fostering awareness through visualizing task and developer centric socio-technical networks, thus extending the networks underlying Ariadne and Palantir by a social component. A task centric socio-technical network is built from all developers and the source

code changes that are related through code dependencies or task discussions. Developer centric networks that show a specific developer what social, technical, or socio-technical relationships they have with their colleagues complement these task centric socio-technical networks.

Ariadne, Palantir, and Tesseract suffer from the fact that they cannot provide real time feedback on changes in technical networks, as they solely rely on changes that take place in the source code repository. *Proxiscentia* [13] addresses this issue by implementing an approach proposed by Blincoe et al. [9] to instrument IDEs used by software developers and gather code edit events as recorded by tools such as Mylyn [58]. This forewarns a developer of changes that are made to related code, for example that Palantir relies on.

Ensemble [113] provides a constant stream of events consisting of modifications to artifacts that are related to the stream owner. For example, if developer Adam posts a comment on a task owned by developer Eve, then Eve's stream would contain an event showing that Adam commented on her task. Similarly, the stream of a developer also contains information about relevant code modifications that overlap, or potentially interact with code that has been previously modified.

Overall, these recommender systems provide awareness of who might be worth interacting with. None of these systems are aimed to accomplish a concrete goal other than achieving awareness. We think that a focus is needed, such as awareness, with respect to dependencies that are relevant for build success. Without such a focus the information that a developer needs to survey can quickly take up too much precious development time and may lead a developer to abandon the systems as they are taking up more time than they save.

2.6 Research Questions

The concept of socio-technical congruence shows potential to help make software development more efficient. Cataldo et al. [19] demonstrated its relation to productivity, and in this dissertation we show the ability to use socio-technical congruence to predict build outcome. The concept of socio-technical congruence lends itself to improve software development as it is based on social networks connecting developers on coordination and technical level. Because the concept is based on networks it is possible to manipulate them.

Any socio-technical network can be manipulated in two ways: (1) change the technical dependencies among developers by refactoring or architectural changes to make them unnecessary and (2) by engaging developers in discussions concerning their recent work and

therefore creating a coordination edge in the socio-technical network. Since many products are not developed from scratch, and because architectural changes once development has been going on for a number of months are costly and time consuming [107], we aim at generating recommendations to change the actual coordination in order to improve the socio-technical network where it matters. Therefore, since a first step, we need to assess if the actual communication structure among software developers has an influence on build success to lay the basis for manipulating the actual coordination to increase build success. Following that, we need to explore the relationship between socio-technical networks and build success. We are especially interested in investigating whether missing actual coordination while coordination needs exists is related to build failure.

In the second part of this dissertation, we begin with investigating the influence of communication among team members in the form of social networks on build success. Next, we investigate if gaps (unfilled coordination needs) between developers, as highlighted by socio-technical networks and the socio-technical networks themselves, can be brought into relation with build success. Chapter 5 and 6 investigate the following two research questions respectively:

RQ 1.1: Do Social Networks influence build success? (cf. Chapter 5)

RQ 1.2: Does Socio-Technical Networks influence build success? (cf. Chapter 6)

Having found a relationship between socio-technical networks, specifically gaps between coordination and coordination needs with build success, while knowing that communication alone has an effect on build success, we formulate an approach to leverage socio-technical networks (cf. Chapter 7). The third and final part of this dissertation focuses on evaluating this approach in three ways: (1) gathering general statistical evidence demonstrating that parts of the network can be manipulated to increase build success, (2) exploring the acceptance of such recommendations based on the manipulations by developers, and (3) a proof of concept that the recommendation could prevent failures. Hence, the first three chapters of the third part of this dissertation are guided by the following three research questions:

RQ 2.1: Can Socio-Technical Networks be manipulated to increase build success? (cf. Chapter 8)

RQ 2.2: Do developers accept recommendations based on software changes to increase build success? (cf. Chapter 9)

RQ 2.3: Can recommendations actually prevent build failures? (cf. Chapter 10)

In the discussion in Chapter 11 we highlight how our findings from these three research questions support the approach we detailed in Chapter 7.

2.7 Summary

In this chapter, we discussed relevant related work that both motivated and enabled us to conduct the research presented in this dissertation. We began with presenting work that is related to software build with a particular focus on how the influence software teams, and the way they communicate and coordinate their work, affect integrating their work into a product. The current body of knowledge reinforces the notion that lapses in coordination (insufficient processes or communication tools) are a major cause for integration issues.

Further exploring existing literature on coordination within development teams pointed us to that software developers have a need to coordinate their work and that those needs are often expressed by the interdependence in their work. This leads to the study of socio-technical congruence as a measure of productivity. Motivated by Conway's Law, several studies demonstrated that a better overlap in the social and technical dimension of a software development effort results in higher productivity.

These social and technical dimensions can be expressed as networks of developers that only differ in their connections that can either be social or technical. Leveraging a combination of technical and social relationships among developer proved to be a good predictor for failures at various granularities ranging from files to binaries. The knowledge that can be gained from this network information is not limited to building failure predictors but has been used to create recommendation systems that enhance the awareness of developers of the work of their fellow colleagues. From the reviewed body of knowledge, we formulated five research questions that guide this dissertation.

Chapter 3

Methodology and Constructs

Before we dive into our actual studies of the effect of socio-technical congruence and its use to form recommendations, we present the overall roadmap for this dissertation (cf. Section 3.1) and some of the common definitions (cf. Section 3.2) and constructs (Section 3.3) that are used throughout the dissertation. Furthermore, we will discuss the general approach to the data collection methods that are employed (cf. Section 3.4).

3.1 Methodology Roadmap

In this section, we discuss the methods that were applied in order to answer the research questions presented in Chapter 2. Figure 3.1 depicts the relationship between the research questions and the contribution of this dissertation, an approach used in attempt to improve social interactions among developers by characterizing the quality of interactions by the build outcome of the related build. Research questions 1.1 and 1.2 discussed in Chapters 5 and 6 motivate the approach. Research questions 2.1 and 2.3 (cf. Chapters 8 and 10) explore whether socio-technical networks can be used to form recommendations that can then prevent build failures, whereas research question 2.2 inquires in Chapter 9 whether such recommendations are acceptable by developers.

3.1.1 RQ 1.1: Do Social Networks influence build success? (cf. Chapter 5)

This dissertation's goal is to design an approach that is able to improve the social interactions in the form of communication among software developers. As a first step, we need

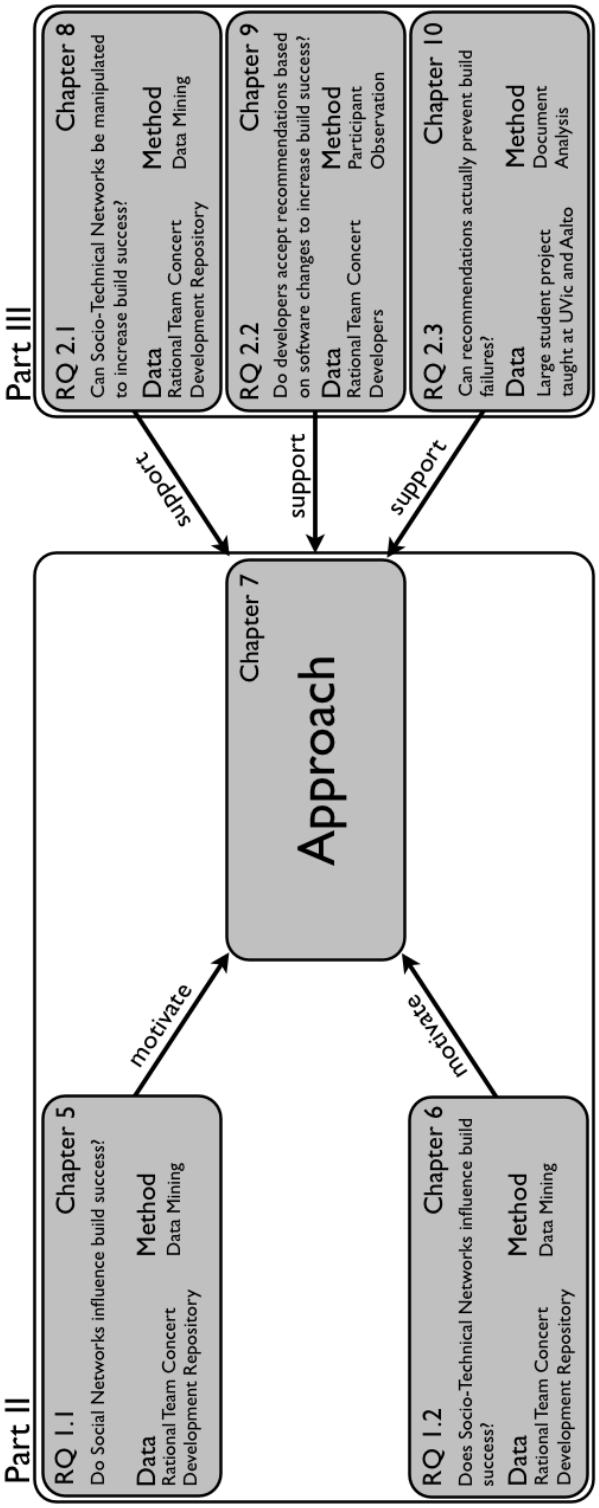


Figure 3.1: What chapter addresses which research questions in relation to our approach to improve social interactions among software developers.

to establish if the communication among software developers has an influence on the build success.

We were allowed access to the development repositories used by the IBM Rational Team Concert development team such as their source code management system, communication repositories in the form of work item discussions, and their build results. All these artifacts are linked together in a way that allows us to trace from the build result which changes went into the build and which work items a change is meant to implement.

Using this information, we can construct social networks from all of the work items that are related to builds. These networks are then described using social network metric and form the input for machine learning algorithms to predict whether a build based on these metrics is more likely to fail or succeed.

Via this machine learning approach we want to establish a connection between a build's social network and its outcome. If we are able to predict the build outcome more accurately than by simply guessing, using the likelihood for a build failure, we demonstrate that there is a statistical relationship between build outcome and social networks. This result forms the first evidence that manipulating the social network might yield a positive effect on build success.

3.1.2 RQ 1.2: Do Socio-Technical Networks influence build success? (cf. Chapter 6)

Knowing that a social network can influence the success of the corresponding build leads us to question how networks should be manipulated in order to improve the likelihood for a build to succeed. Therefore, we explore the relationship between socio-technical networks generally and gaps within that networks, with a gap being formed by two developers that share technical dependencies but failed to communicate about work related to the build of interest and build success. Similarly to the previous research question, we based the analysis on the same data set, allowing us to directly infer technical relationships with developers related to a software build from the changes submitted to the source code management tool. We used these changes previously to infer the work items developers used to communicate among each other about the build.

Since the socio-technical networks have two semantically different edges connecting two developers within a network (technical dependencies and communication among developers) we refrain from using social network metrics as they assume only one mode of connection among nodes within a network. Instead we investigate the relationship of the

socio-technical congruence index and build success as well as focusing on the influence of gaps in the network on build success.

Via statistical analysis methods such as regression analysis we want to establish a relationship between the existence of gaps within the socio-technical network and build success. By addressing this research question we obtain another piece of evidence that allowed us formulate an approach to recommend actions to increase build success that are specifically alleviating gaps within the socio-technical network by recommending developers to communicate.

3.1.3 RQ 2.1: Can Socio-Technical Networks be manipulated to increase build success? (cf. Chapter 8)

The previous two research questions enable us to formulate an approach to generate recommendations that are meant to foster communication among developers in order to increase build success. This leads to the next step, in which we explore whether this approach can generate recommendations that show a statistical relationship to build success.

Using the same data source as we did earlier, we try to relate individual reoccurring gaps in socio-technical networks to build failure. Knowing those gaps, or developers that frequently share a technical dependency without communicating with respect to a build that failed, we check if adding a social dependency would change the likelihood of a build to fail. We expect to find a number of gaps that, when mitigated, increase the likelihood of build success.

3.1.4 RQ 2.2: Do developers accept recommendations based on software changes to increase build success? (Chapter 9)

Before exploring whether the recommendation holds actual value concerning the prevention of build failures, we explore whether developers would welcome recommendations with respect to changes. To do this, we joined the development effort of one of the Rational Team Concert development teams as participant observer to get an insight into how the actual developer communicate during their day to day work.

We complement these observations using followup interviews in order to gain a better understanding of the team dynamics and their discussion topics, since as a project newcomer our work is limited to more basic tasks in contrast to higher level decision making. To extend our reach beyond the local team we deployed a questionnaire to the product

team at large to gain a better understanding whether the recommendations we would supply could be easily integrated in their typical discussions with fellow developers.

This study should give us a better understanding of whether developers are discussing individual changes, thus justifying the appropriateness of the level of recommendations. Furthermore, we expect to uncover general suggestions on when and how to supply such recommendations as developers might not always be interested in individual changes even when they may pose a threat to build success.

3.1.5 RQ 2.3: Can recommendations actually prevent build failures? (cf. Chapter 10)

We conclude our evaluation of our proposed approach with a proof of concept in a more controlled environment to ascertain whether the recommendation we can generate actually does help in preventing builds from failing. As root cause analysis of failures can be very tedious, and due to the various reasons a build can fail that are not necessarily due to software changes, we depart from studying the development of Rational Team Concert and observe students extending an open source project during a course taught at the University of Victoria, Canada, and Aalto University, Finland.

This change in setting allows us to ask the study participants to use tools we developed to collect more fine grained data on their development efforts, such as when they edited which parts of the source code. Additionally, we have access to the actual source code repositories such as source code management, issue tracking, email, and text chat sessions. Furthermore, we ask the students to answer regular questionnaires and keep a development diary that are meant to collect information on issues that the students encounter during the course and are specifically related to their development effort.

We then analyze the collected data from the questionnaires and development diaries manually, by reading and annotating the different responses and entries to uncover build issues that were important to the students. Knowing the build issues that the students encountered, we select the most severe in terms of the number of reports, and trace them to the actually root cause, by identifying the code changes that cause them (if applicable) and continue to analyze the offending changes as well as identifying corresponding communication.

We expect to find one representative example of a failed build that the students encounter during the course that can be resolved using recommendations based on information that is available before the students actually incorporate their modifications in to the source code repository of the project.

3.2 Definitions

We start by giving our definitions of the three constructs that we are heavily relying upon: (1) Work Item being a complete unit of work, (2) Change-set being the technical work submitted by a developer, and (3) builds referring to a testable product.

3.2.1 Work Item

A *Work Item* is a unit of work that can be assigned to a single developer. A unit of work can be anything from a bug-report, reported by either the end user or by a developer, to a feature implementation. *Work Item*s can be hierarchically organized to show the work breakdown from high-level requests to manageable pieces. One project team member is responsible for a work item to be completed, the sub work items that it is broken-down to do not necessarily need to be assigned to the owner of the parent work item.

For instance, in the case of the IBM Rational Team Concert the development team creates story items to describe larger functionality from the user point of view and assigns them depending on the complexity and implication of the story. The owner of the story then either breaks down the story into multiple stories or tasks that are again assigned to team members according to their complexity and implications. Once the work item level is sufficiently low, the developer assigned to it can make the necessary modifications to the project to accomplish the work detailed in the work item.

Definition. *A **Work Item** is a defined and assignable unit of work.*

3.2.2 Change-Set

A *Change-Set* is a set of source code changes applied to a number of source code files, with a file being the artifact that a developer would change to add to, modify, or delete from the current product. The developer that applied those changes to the product bundles them into one or multiple change-sets. For example, in the Eclipse project[1] the developers use CVS[2] as their version control system to manage changes to the Eclipse IDE. A developer will check out their current version of the repository and started editing, creating and deleting files in order to fulfill a work item she is currently working on. Once the developer decides that she has accomplished the work to finish her current work item, she commits her

[1] `http://www.eclipse.org`
[2] `http://www.nongnu.org/cvs`

changes. Those changes that consist of file creations, deletions, and modifications taken together are referred to as a *Change-Set*.

Definition. *A **Change-Set** is a set of modifications, additions, and deletions of software artifacts such as source code files, classes or methods.*

3.2.3 Build

The goal of each software development team is to deliver a finished or improved product at some point in time. This finished product is often referred to as the final *Build*. A *Build* can generally be referred to as any instance of the product that can be run to some extent. To create a build, a team gathers all of the changes implementing work items, that are required for the new build and compiles and packages the product. The amount of work items and their respective changes included in a build, will gradually increase over time since as the project progresses more work will be completed.

In the case of the IBM Rational Team Concert development team, builds are created on a frequent basis to test the product as a whole in order to look for integration issues. The team also subscribes to the philosophy to use their own products and thus tries to bring each build to a level at which it can then be used for development. This intense use enables the team to spot issues that are still within the product and enables them to assess the severity of these issues.

Definition. *A **Build** is to some extent a executable version of the product that includes a number of changes implementing work items.*

3.3 Constructs

From the definitions introduced previously we can derive the three central constructs that we work with in this dissertation: (1) the social network connecting communicating and coordinating developers, (2) the technical network connecting developers that are dependent through code artifacts, and (3) the socio-technical network that combines the social and technical network in a meaningful way. These constructs are important for the three chapters that are mining the repository provided by the Rational Team Concert development team (cf. Chapters 5, 6, and 8).

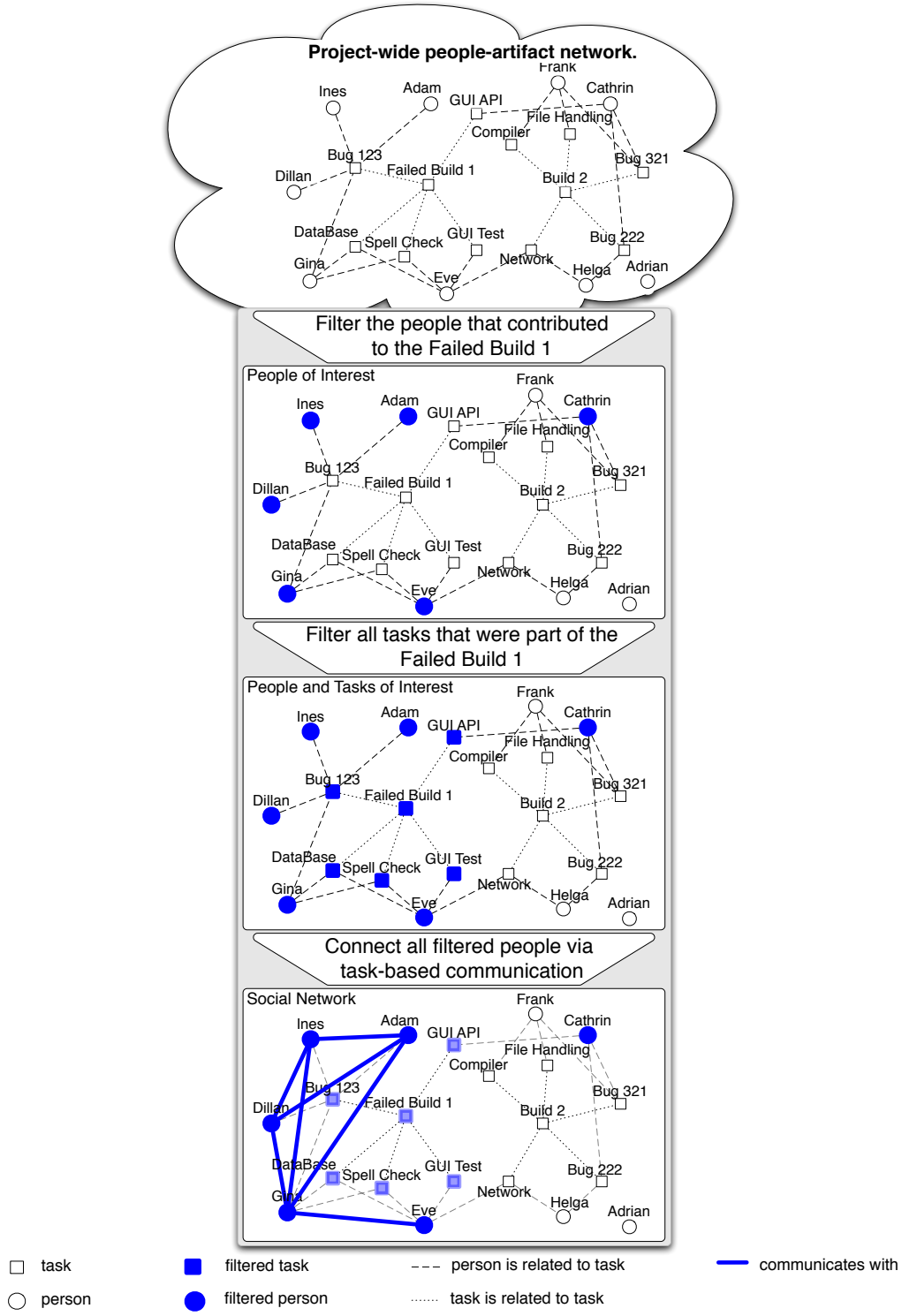


Figure 3.2: Social network construction examples in our approach

3.3.1 Social Network

To illustrate our approach to construct social networks we go through the example of a failed build illustrated in Figure 3.2. A social network is represented as a graph that consists of nodes connected by edges. In our approach, the nodes represent people and edges represent task-related communication between these people.

The approach is repository and tool independent and can be applied to any repositories that provide information about people, tasks, technical artifacts, and communication, this includes work, issue, or change management repositories, such as BugZilla or IBM Rational Team Concert; or source code management systems, such as CVS or IBM Rational Team Concert; or even communication repositories such as email archives.

We construct and analyze social networks within a collaboration scope of interest, using a collaboration scope defining the people and interactions of interest. In this example, around Failed Build 1, the collaboration scope is the communication of the contributors to the failed build. Other examples include the collaboration of people working on a critical task, in a particular geographical location, or in a functional team such as testing.

There are three critical elements that are necessary to construct task-based social networks for a collaboration scope and that need to be mined from software development repositories:

Project Members are people who work on the software project. These project members can be developers, testers, project managers, requirements analysts, or clients. Project members, such as Cathrin and Eve, become nodes in the social network.

Work Items are units of work (as defined earlier) within the project that may create a need to collaborate and communicate. Examples for a work items include resolving Bug 123 or implementing the GUI API. More generally, implementing feature requests and requirements can also be considered collaborative tasks.

Work Item Communication is the information exchanged while completing a work item and is the unique information that allows us to build task-based social networks. In our example, dashed black lines represent task-related communication such as a comment on Bug 123, or an email or chat message about GUI API. Task-related communication is used to create the edges between developers in the social networks.

The data underlying the social network used throughout this dissertation is based on work items and their associated discussions. In IBM Rational Team Concert each work

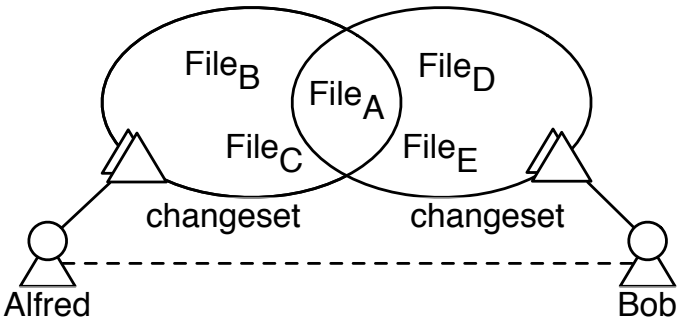


Figure 3.3: Creating a technical network by connecting developers that changed the same file.

item has an attached discussion thread where developers can discuss the work item or simply note down their thoughts while working on the work item. This means, we would create a link between two developers if they comment together on the same work item to indicate that they are part of the same discussion. Note that this section draws heavily from our work done in collaboration with Timo Wolf, Daniela Damian, Lucas Panjer, and Thanh Nguyen [112].

3.3.2 Technical Network

Building technical networks follows a very similar approach as we described for building social networks. In fact, the technical network is a social network whose main distinction from the social network lies in the way edges between nodes are created. We derive the name of technical networks from the way we link developers together, namely if they are modifying related source code artifacts. As in the previous network construction, the construction of the technical network is based on three components:

Project Members are people who work on the software project. These project members can be developers, testers, project managers, requirements analysts, or clients or in general anyone that modifies software artifacts through change-sets. In Figure 3.3 project members, Alfred and Bob, become nodes in the technical network because they modified the same file.

Change-Sets are changes made to software artifacts by individual users. A set consists of a number of artifacts that have been modified as well as the modifications themselves. For example, Alfred as shown in Figure 3.3 modified File$_A$ and File$_B$.

Software Artifact Relation describes the relation between developers in the technical networks. This relationship can be defined in several different ways. For example, in Figure 3.3 Alfred and Bob are related through a technical relationship because they modified the same file. Note that we are mostly interested in relationships between artifacts that are affected by changes.

Therefore, constructing technical networks follows three steps: (1) gather all change-sets of interest, (2) identify the relations between artifacts, (3) infer the relation between the artifact owners using the change-sets and the relations between the source code artifacts. For example, after having selected the set of change-sets of interest we define the change-sets themselves as the source code artifact and identify the owners of those artifacts. We then infer the relationship between those source code artifacts by relating all change-sets that affect the same source code file. In the case of Alfred and Bob, this means that they are connected because both own a change-set that modifies the same file (cf. Figure 3.3).

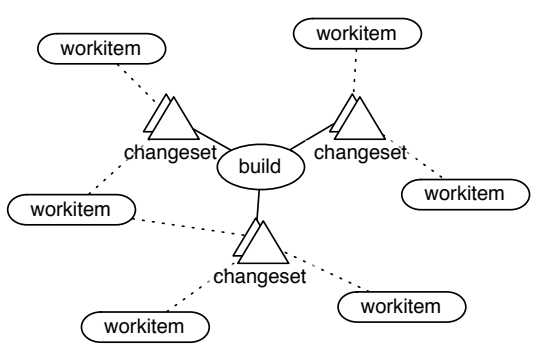
3.3.3 Socio-Technical Network

Socio-Technical networks are a meaningful combination of both social and technical networks. Selecting this meaningful combination reflects itself in the selection of the work-items in the case of building the social network and selecting the change-sets and their relations in the case of the technical network. Hence, constructing a socio-technical network requires the following four steps:

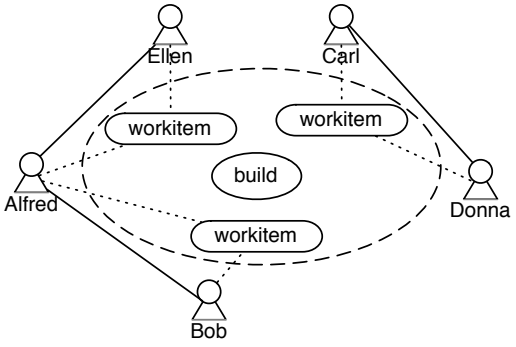
1. **Selecting the Focus** used for the socio-technical network represents the glue that binds the social and technical network into a socio-technical network. This focus, also referred to as filter in our earlier publication [112], determines the content of the networks. In this dissertation, we select as the focus software builds to construct networks that describe the coordination among developers for a given software build.

2. **Constructing the Social Network** follows the description above with the focus determining the work items that are selected in order to generate the nodes from the work item participants and the edges from the communication among the participants through a work item.

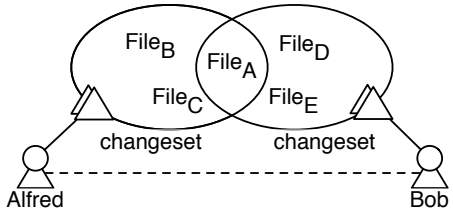
3. **Constructing the Technical Network** follows the description of constructing technical networks above, with the focus determining the change-sets being used to determine and connect developers in the network.



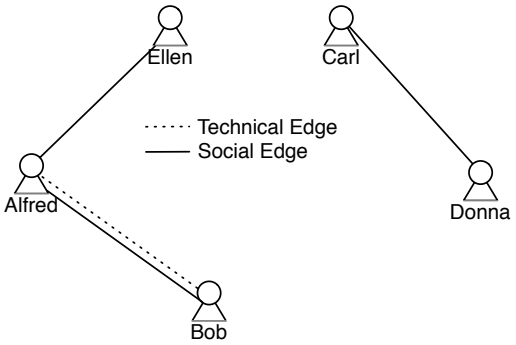
(a) Inferring to the build focus relevant change-sets and work items.



(b) Constructing an social networks from work item communication.



(c) Linking developers in a technical networks via change-set overlaps.



(d) Combine social and technical networks into a socio-technical network.

Figure 3.4: Constructing socio-technical networks from the repository provided by the IBM Rational Team Concert development team.

4. **Combining Networks** is the final step that overlays the networks by unifying the set of developers. Thus, a pair of developers can be directly connected to one another through two edges, one representing the edge from the technical network and the other the edge from the social network.

Figure 3.4 shows an example of how we, in our studies of the IBM Rational Team Concert development team, create socio-technical networks. In the first step (cf. Figure 3.4a) we set the focus on a software build which allows us (via the change-sets that made it into the build) to infer what work items are also represented in said build. Given the focus, the social network can be constructed using the work items that can be linked to the software build (cf. Figure 3.4b). Similarly, the construction of the technical network relies on the change-sets that went into a build. To actually infer edges between developers, we relying on co-changed files within a build as an indicator of work dependency (cf. Figure 3.4c). Finally, the two networks are combined and yield the socio-technical network shown in Figure 3.4d.

3.4 Data Collection Methods

To conduct the research for this dissertation we drew upon multiple data sources. We employ repository-mining techniques to identify larger trends in measurable activities. This allows us to gain a more in-depth understanding in on how developers actually work and deal with interdependencies, especially how they would react to certain recommendations and whether they are can be made useful we employ qualitative methods.

3.4.1 Repository Mining

Software development usually uses a number of tools to manage information electronically, such as version archives and issue trackers. Additional to storing source code and tasks/issues, these software repositories are also able to contain digital communication, such as forum and email discussions, logs of IRC[3], and instant messenger chats.

Repositories can grow to considerable sizes depending on the projects life span and intensity. Therefore, it is often not possible to manually review the history of a project and it is then necessary to employ data-mining techniques in order to analyze trends. Although this approach is limited in terms of gaining a deeper understanding of the intricacies of a development project, it nevertheless is invaluable to place in-depth results in the bigger picture of the project. Furthermore, data mining approaches are one way to easily give back value to the development team without burdening any individual developer by diverting time to other non-automatic data collection instruments. This is an important point, as one goal of this dissertation is to explore the ability of the concept of socio-technical congruence to generate actionable recommendations. In case a developer needs to personally provide a large amount of information manually, the overhead generated by a system might outweigh the benefit of recommendations and therefore render the system useless.

We extract information from three different types of repositories: (1) version control, (2) task management, and (3) build engine systems. The version control supplies us with the knowledge of how developers are connected through their technical work. The task management supplies us with information on which individuals were communicating with respect to a specific work item, and lastly the build engine supplies us with the focus to construct socio-technical networks.

In order to derive socio-technical networks we need to link the different artifact types. Within IBM Rational Team Concert, as illustrated in Figure 3.4a, work items are linked

[3] `http://en.wikipedia.org/wiki/Internet_Relay_Chat` last visited June 8th, 2012

to change-sets and change-sets are linked to builds, therefore, establishing the connections needed to construct socio-technical networks with a build as focus. Similarly, these links can be inferred, as proposed by Cubranic et al. [22], from repositories that are missing the capabilities of creating formalized links. We used repository mining techniques in Chapter 5, 6, and 8 to explore the rich repositories provided by the IBM Rational Team Concert team.

3.4.2 Surveys

To complement the insights that were obtained from mining repositories we use surveys. Surveys are designed iteratively and piloted before deployment. They are intended that will collect input to enrich and clarify information obtained from the software repositories. With each survey we try to minimize the time each developer needs to spend completing them, which usually limits ourselves to focus on closed questions offering prepared answers. We constrain ourselves in this way to minimize the distraction to each individual developer and thus increase the response rate.

Our surveys are deployed through web services in an attempt to make the collections more convenient to each developer as they are spending most of their times working at a computer. Keeping track of a paper version is more cumbersome as they might not easily be returned, especially considering that the development teams we are collaborating with are distributed across different continents.

We both deployed surveys in an attempt to make with the Rational Team Concert development team (Chapter 9) and when working with students on a large course project at the University of Victoria, Canada, and Aalto University, Finland (Chapter 10).

3.4.3 Observations

The next richer, and also to the developer more distracting method, of data gathering are observations. The act of observing can distract developers and also change their behaviour, although we do not actively interrupt or distract developers. In order to minimize this type of distraction and to mitigate the observer bias, we employed a special form of observation study known as participant observation. In short, we became both an observer and a participant. This has a multitude of advantages:

- **Reciprocity.** By participating in the actual development we can provide value to the development team from the very beginning. This, in turn, motivates the developers

to give us the time we need to conduct other parts of the study, like surveys and interviews.

- **Learning the Vocabulary.** Each development project has its own project vocabulary [34] in order to effectively and clearly communicate. Understanding this vocabulary as an outsider can be difficult, but is definitely very important when it comes to making sense of comments.

- **Understanding the Context.** For example, in one study, our observation period coincided with the months prior to a major release during which the team focused on extensive testing rather than new feature development. Due to the proximity of the observation period to a major release we observed mainly activities around integration testing with little coding activity aside from fixing major bugs.

  Although it is easy to ascertain when the next major release of the product is, the affect this has on the developer, besides a change in the process, is harder to gauge. Being part of the development effort allowed us to better understand how developers react to the change in process and better understand their struggles.

- **Asking more Meaningful Questions.** A better understanding of the project and how it affects the individual developer as well as gaining a better understanding of the vocabulary helps with phrasing better questions in the sense of both more meaningful to the developer.

Besides gaining a better understanding of some easily missed (or miss-understood) intricacies, working together with the developer establishes a trust relationship [69]. This trust helps to mitigate observation biases that are introduced solely by observing as well as prompts developers to be more forthcoming during interviews and surveys [69]. It is difficult to separate the different data collection methods that involve human interaction; therefore, we think combining these data collection methods in the right order can greatly enhance the quality of the collected data. We were able to join the Rational Team Concert development team as a participant observer mainly due to the convincing argument that we take the place of an intern, thus, contributing to their development effort (Chapter 9)

3.4.4 Interviews

To further enhance our understanding of how developers view the situation and further make sense of survey responses as well as results from mining repositories and our obser-

vations, we employed interviews. Instead of following a structured interview approach, we opted for a semi-structured interview with a focus on war stories. War stories [70] ask the interviewee to share memorable stories from work life. The interviewer can explore these war stories and help shape the focus of the discussion of the events. This type of interview comes with two major benefits over structured interviews that follow a set of questions:

- **Focus onto for the interviewee important events.** Our goal with this dissertation is to better support software development. Knowing the pain points, as the projects participants perceive them, allows us to focus on important issues. With prepared questions, the focus of the interview might not uncover what is important to the interviewee and thus we might miss areas.

- **Better recall of events by interviewee.** Recall of important events is better than arbitrary ones [70]. This allows us to place more confidence on the reports and answers given by the interviewees. Dissimilarly, in structured interviews, the interview itself is guided by a set of prepared questions and goals that do no necessarily address events that the interviewee can easily recall.

The main drawback of using war stories over prepared interview questions in a structured interview framework lies with the loss of focus of the interviews. By asking the interviewee to tell war stories of memorable events it can be more difficult to gain insight into a particular area of interest if the war stories veer too far off the topic. It is therefore necessary that the interviewer have a good understanding of the project and the project language in order to explore the stories for relevance in terms of the topics of interest, thus making the process more demanding for the interviewer.

We tried to minimize the interruption to project members as much as possible. To do this, we limited the time we require for each interview as much as possible. We aimed to make each interview approximately 30 minutes with a 30-minute overflow in case a participant desires to continue the interview. Furthermore, we gave the work of the interviewee priority over the interview, and assured the interviewees that they could stop the interview if at any point they felt that their work needed attention. This was especially valuable with a professional development team such as the IBM Rational Team Concert development team as we joined their development effort when they were nearing a major milestone (cf. Chapter 9).

3.5 Summary

Throughout this dissertation, we present our methodology and constructs. We started with a roadmap describing the methodology used to answer our five research questions. Subsequently, we gave our definitions of what we refer to as a work item, change-set, and a build. These three definitions build the foundation that enables us to conceptualize social interaction and technical dependencies into networks of developers.

We further described the methods we used to elicit our data while we conducted our studies. Trying to minimize the interruption with actual development, we focus on exploiting mining repository techniques. When interacting with developers we use a mixed method approach which consisting of observations, surveys, and interviews.

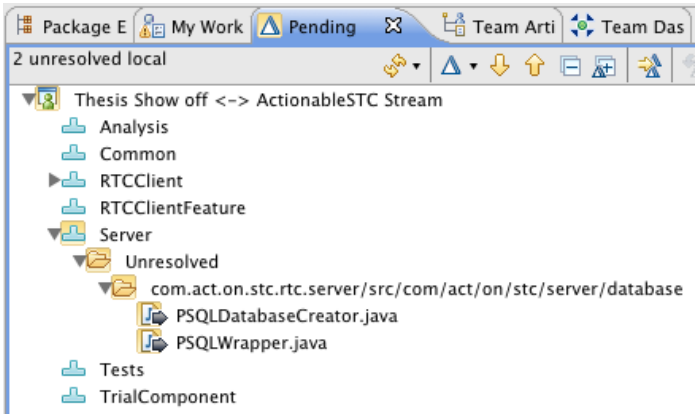
Chapter 4

IBM Rational Team Concert

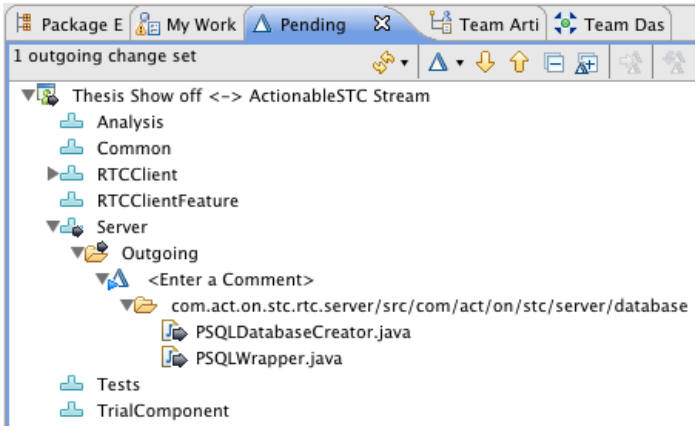
In this chapter, we describe the IBM Rational Team Concert[1] products as well as its development team, since most of the research carried out throughout this dissertation was in collaboration with this product development team which as part of their development process is also using their own product. We begin by introducing the product and its functionalities before moving on to the development team's composition and the development process.

4.1 The IBM Rational Team Concert Product

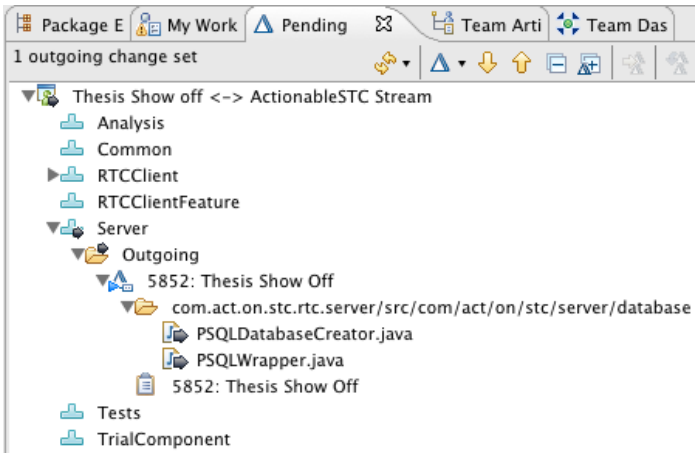
IBM Rational Team Concert (RTC) is a server client application meant to support software development. It is comprised of four major functionalities that bear relevance to this dissertation: (1) source control management that is based on similar principles as GIT[2] and mercurial[3], (2) work item management to keep track of issues, tasks and features that need to be resolved and implemented, (3) planning capabilities to organize work items into iterations working towards milestones, and (4) a build engine that allows for continuous builds and regression tests.



(a) A set of changes that is only on the developer's local machine.



(b) A change-set that is also in the developer's remote workspace.



(c) A change-set that is attached to a work item.

Figure 4.1: From having created local changes over adding them to the remote workspace to attaching it to a work item.

4.1.1 Source Control

The source control system provided by Rational Team Concert is server-based, this is similar to traditional version control systems such as CVS[4] and SVN[5]. Yet, developers have their own workspaces similar to a local GIT or mercurial repository. These branches, or in RTC terminology streams, can contribute their changes in the form of change-sets to other streams shared by teams or that can be exchanged between workspaces.

With this, the main difference between the RTC source code management system and a distributed source code management systems such as GIT lies with all the repositories or streams being present on one single machine as opposed to many different machines. Even branching and sharing of changes to the source code work similarly to distributed version control systems as they can be moved freely in-between workspaces, in-between streams, and between streams and work spaces.

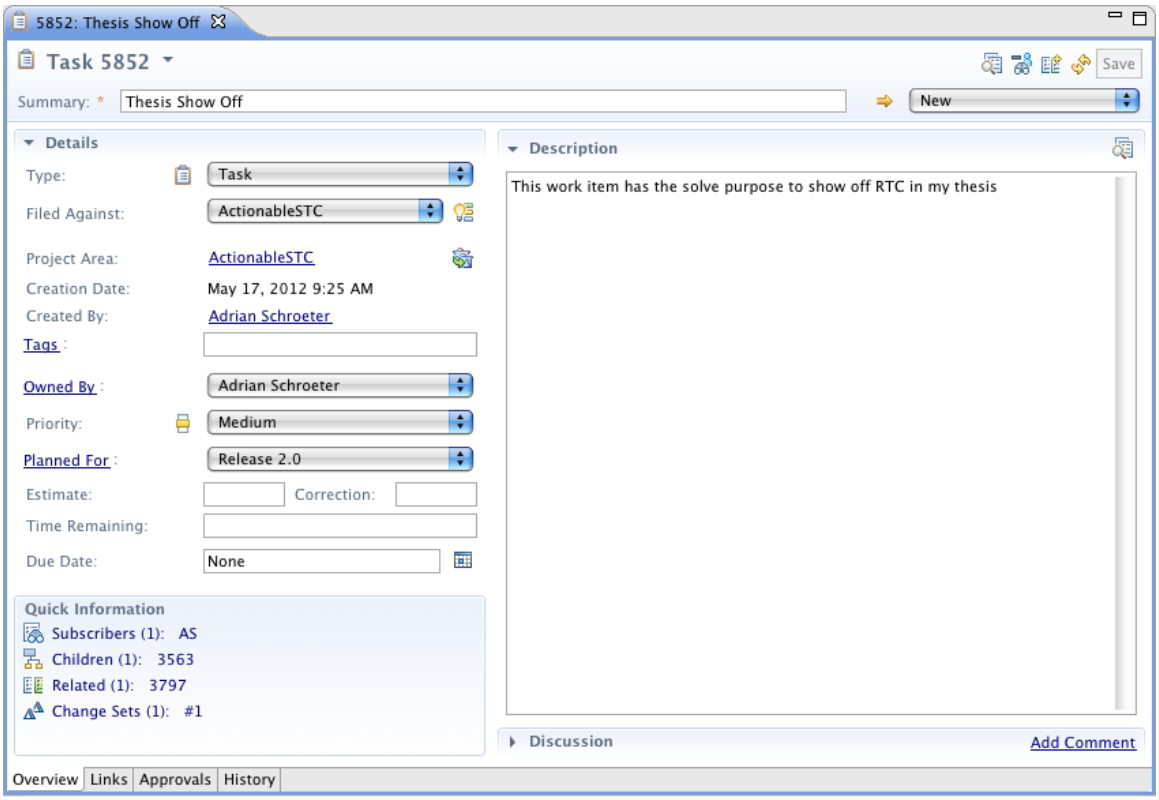
For example, Figure 4.1 presents a typical work flow after a developer finished a unit of work that needed to be finalized. A developer would gather all the changes that are only available on her local machine (cf. Figure 4.1a) and combine them into a change-set (cf. Figure 4.1b) that is at that point made available on her remote workspace. After ideally linking that change-set to the task it is meant to resolve (cf. Figure 4.1c) the developer can deliver the change-set to the team stream to make it available to the whole team.

4.1.2 Work Items

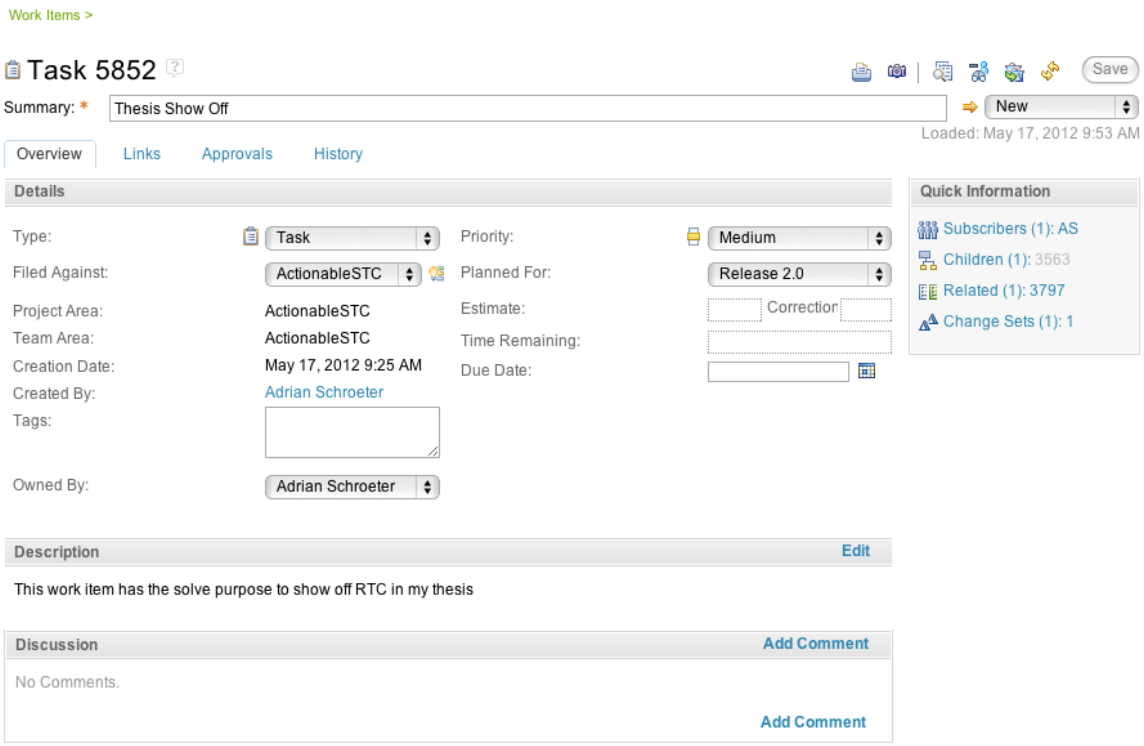
The work item component serves two functions: (1) it allows for customer interaction as it allows a customer or user to file feature requests and file bug reports and (2) it allows developers to manage their work by creating tasks as well as discussing and documenting thoughts about a defined unit of work. These work items are basically an extension of bug reports as filed to a BugZilla[6], since they serve more purposes than capturing bug reports and feature-requests.

Developers have the ability to link work items into logical units and build hierarchies of work items to structure their work. Furthermore, work items allow tracking meta-information, such as time estimates, severity, and priority, and on top of that lend them-

[1] https://www.jazz.net
[2] http://GIT-scm.com/
[3] http://mercurial.selenic.com/
[4] http://www.nongnu.org/cvs
[5] http://subversion.tigris.org/
[6] http://www.bugzilla.org/

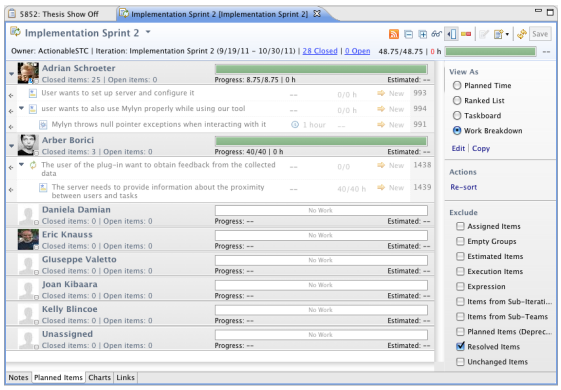


(a) A work item as most developer look at it from within the Eclipse client.

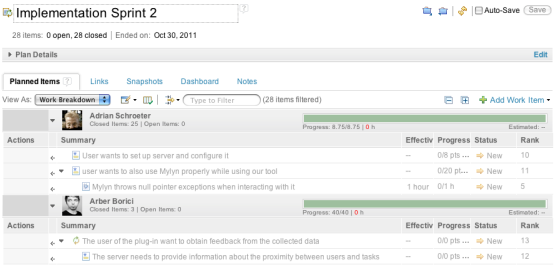


(b) A work item as most manager look at it from the web ui.

Figure 4.2: Workitems as shown by the different RTC UI's.

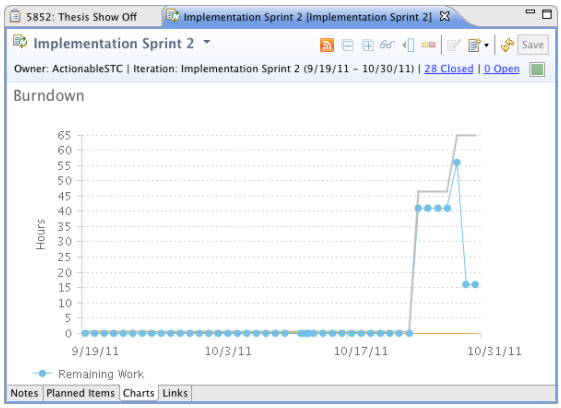


(a) Planning from the Eclipse UI.

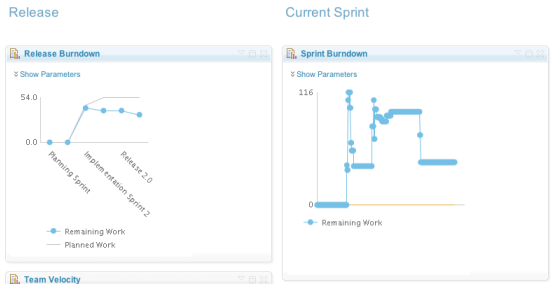


(b) Planning from the Web UI.

Figure 4.3: Plans as shown by the different RTC UI's.



(a) Chars in the Eclipse UI.



(b) Chars in the Web UI.

Figure 4.4: Charts as shown by the different RTC UI's.

selves to be customized with self-defined fields on a project configuration level. For example, in Figure 4.2 we see the work item view from within Eclipse (cf. Figure 4.2a) and through the web (cf. Figure 4.2b). Both representations show links to other work items in the Quick Information pane as well as links to other artifacts such as change-sets.

4.1.3 Planning

In contrast to issue trackers such as BugZilla, IBM Rational Team Concert offers planning capabilities. Within RTC, plans can be defined that consist of a number of work items that can be ranked against each other and assigned to different team members (cf. Figure 4.3 for an example). RTC also supports multiple plans for different teams to support a better overview for the teams instead of cluttering plans with less relevant tasks.

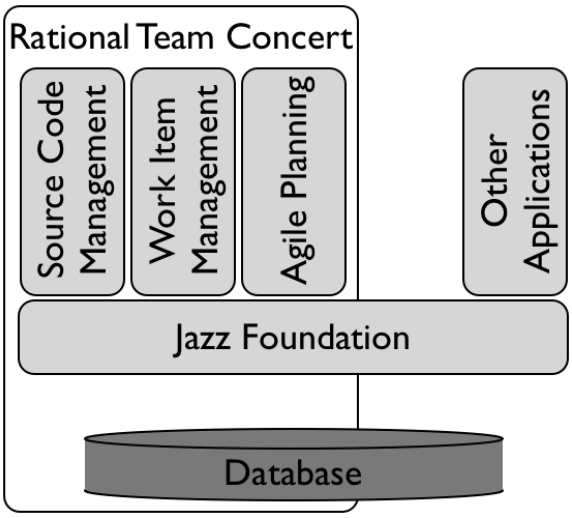


Figure 4.5: RTC topology

The planning component supports several development styles, reaching from agile-development styles such as XP and SCRUM [67], to more traditional development methods, such as Waterfall [10]. Each unique development style comes with a different default of control measures and charts such as burn down charts. These charts, as well as the planning, can be both reviewed in the Eclipse and web client. Both can be viewed in a customizable dashboard for quick information retrieval to stay up to date. Figure 4.4 shows some examples of burn down charts as they show up on the Eclipse client (cf. Figure 4.4a) and the Web UI (cf. Figure 4.4b).

4.1.4 Build Engine

The RTC build engine is comparable to build engines, such as IBM Rational Buildforge[7] and Jenkins[8]. As build engine it allows for both automated builds in pre-defined intervals, such as nightly builds as well as on demand builds from users. Build Results RTC can either be `ERROR`, `WARNING` or `OK`.

4.1.5 Foundation/Integration

The IBM Jazz Foundation builds the layer on which the previously described components run. By itself, it does not offer any capabilities to the user, but it enables each component

[7] http://www-01.ibm.com/software/awdtools/buildforge/enterprise/
[8] http://jenkins-ci.org/

to be integrated with the others. This allows the developer to both manually link artifacts with each other and define queries that cross-reference multiple repositories. Figure 4.5 shows the three layers of the RTC product. At the top is the user that interacts with the application servers, ranging from source control over planning to work item management. These applications run on a common platform on the bottom, also referred to as the Jazz Foundation server, that runs onto a database instance.

For example, a developer is working on implementing a work item and creates changes to the code base that they bundle into a change-set. Before they submit the change-set resolving the work item they link that change-set to the work item. This enables their other colleagues to easily accept the change-set into their workspaces to both review and test them. Once the work item is resolved, the plan and the burn-down charts update to reflect these changes and become visible to anyone with access to the plan and charts.

Developers often need to modify existing code and therefore need to understand the thought process that has gone into the code they are looking at [109]. For this purpose, developers can create queries that look for work items associated with changes to the code parts and instead of reading source code they have access to documentation related to the code they are investigating.

4.2 The IBM Rational Team Concert Product Development Team

The IBM Rational Team Concert product development team is distributed across multiple sites following an agile development methodology. In this section, we characterize the development team at large as well as describe their development process.

4.2.1 The People

The Rational Team Concert team is a large distributed team and uses the Rational Team Concert for development. The Rational Team Concert development involves distributed collaboration over 16 different sites located throughout the United States, Canada, and Europe. Seven sites are active in RTC development and testing. There are 151 active contributors working in 47 teams at these locations where contributors belong to multiple teams. Each team is responsible for developing a subsystem or component of RTC. The team size ranges from 1 to 20 and has an average of 5.7 members. The number of developers per

	total	minimum	maximum	average
Number of Locations	>16			
Number of Teams	47			
Number of Developers	151			
Range of Team size		1	20	5.7
Range of Developer per Location		7	24	14.8

Table 4.1: Descriptive statistics of Rational Team Concert development team.

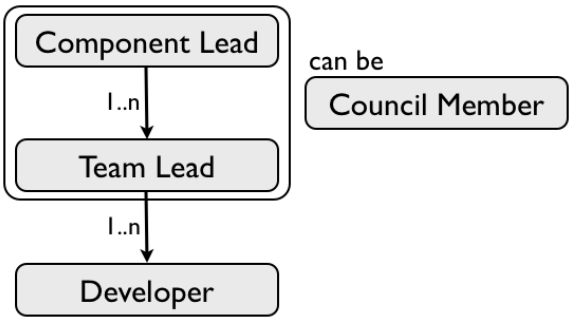


Figure 4.6: Organizational structure of the technical personal in the RTC development team

geographical site ranges from 7 to 24 and is 14.8 on average. Table 4.1 summarizes the mentioned team statistics.

The teams are organized into larger component based teams as well as subcomponent and cross-component teams. A developer can be part of multiple teams but usually is part of only one component team. Most teams are located in one specific spot with some satellite members in different locations.

Teams on a component level represent a wealth of experiences that can range from standing veterans that did software development for several decades, fresh university graduates, and even the occasional intern. This variance in experience is also reflected in the organizational structure. The more experienced project members take leading roles ranging from team leads to component leads and program management council roles (see Figure 4.6):

Team Leads are still involved with daily development activities and usually have more development experience especially with the sub-component their team is developing and maintaining. Additionally, team leads are both the first contact person when cross component issues arises that need to be discussed with other teams. Note, that this is for the initiation of the cross team collaboration and to communicate the severity

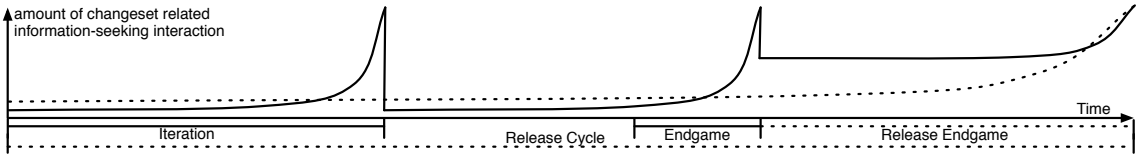


Figure 4.7: The pattern of information-seeking interactions throughout several iterations of a release cycle. Every release cycle consists of a number of iterations; each iteration includes an endgame phase. Change-set-based interactions are more frequent during endgame phases and during the last iteration of the release cycle.

and the need for the other team to re-prioritize their work items.

Component Leads work together with multiple team leads and are responsible for the implementation of larger components, such as the source code management component, and do less development work than team leads.

Council Members deal with component crossing issues, such as architectural or process related issues. They decide how components interact, as well as what changes need to be implemented, to fulfill certain feature requests on a higher level. Furthermore, council members also discuss how feasible the implementation of features with respect to the existing architecture is.

There are still "regular" software developers that are less involved in cross-component issues unless they are part of a cross-component team to implement a feature that is similar across several components. In contrast to team leads, developers focus on their daily implementation work and interact with other developers mainly to resolve issues.

4.2.2 The Process

The IBM Rational Team Concert development team follows an agile development methodology and high community involvement. In short, they are following the Eclipse Way of Development process described by Frost [37]. This development methodology places a large emphasis on community involvement. Although IBM Rational Team Concert is not an open source product, and non-employees are not given access to the actual source code repository, the development team encourages the community to ask question on how to enrich both the server side components and the RTC client with additional plug-ins.

The development team progresses in six-week iterations towards larger releases of IBM Rational Team Concert. These iterations as well as the release cycle are further divided into end-game and non-end-game phases. A typical release cycle begins with setting high-level

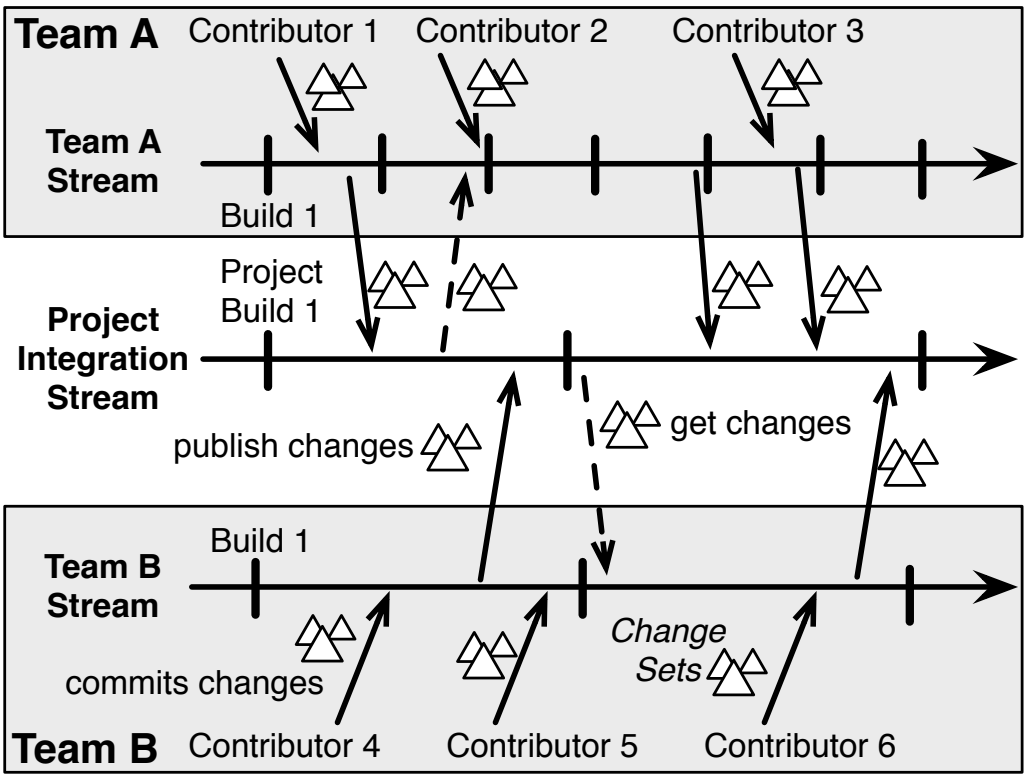


Figure 4.8: Teams contribute to their own source streams, which are then merged into one project stream.

goals for the iteration to implement certain features that are determined by a committee (see Figure 4.7). These features represent a mix of customer, community, and developer suggestions, which are then formulated into user stories and epics for the release cycle. These stories are then estimated and distributed across the six week iterations within a release cycle with a major milestone around the IBM Rational's Innovate conference[9] to show case the latest version of IBM Rational Team Concert. At the end of each iteration the Rational Team Concert development team tries to switch to the latest RTC version for self-hosting in accordance to eating their own dog food for quality insurance.

On a day-to-day basis, developers work on work items that are assigned to them by the team leads to work towards their iteration goals. While working on a task, developers document their thoughts on the respective work item and engage in discussions of work items related to their work. Once the coding work for a work item is finished, developers group all changes into a change-set and attach it to the respective work item and then deliver the change-set to the team stream.

[9]http://www.ibm.com/software/rational/innovate

Change-sets in the component team's stream are used to create nightly builds and run all the test cases on. Once a build is stable for a component team, the change-sets provided by the team are bundled and distributed to the other component teams (see Figure 4.8). During the non-end-game phase this process is very fluent and stream lined. Once entering an endgame-phase the process of including change-sets into a build becomes more regulated. Each change-set needs approval before it is included in a build. Moreover, before a developer can start working on a work item its risk to the project and value needs to be assessed.

4.3 Summary

In this chapter we introduced the Rational Team Concert product and development team. The product Rational Team concert consists of an advanced source code management system combined with issue tracking and agile planning tools that are delivered to developers through both a web UI as well as the Eclipse IDE. Rational Team Concert also contains a build engine, thus it contains a wealth of information about the development process and build success.

The Rational Team Concert development team is distributed over several locations spanning more than eight time zones. The team follows an agile development methodology resulting in a release cycle that is subdivided into six-week iterations. The development team uses their own tool for development allowing them to structure their development to the best abilities of the planning and source code management.

Part II

The Approach

Chapter 5

Communication and Failure

We open our investigation into how to modify the social relationships among software developers, represented by the communication among them, by searching for a relationship between communication and build success [111]. This forms the basis and justification for an approach that we describe in Chapter 7 to allow us to manipulate the social interactions among developers. Thus this chapter explores our first research question:

RQ 1.1 Do Social Networks influence build success?

A connection between communication among developers and any sort of software quality, including software builds, makes sense intuitively. For example, any non-trivial software project consisting of several interdependent modules, and with the growing size and number of modules, more than one software developer is required to finish the project within a certain time frame. Now due to the interdependence of the software modules, developers assigned either to the same or two interdependent modules, need to coordinate their work. This coordination is in most part accomplished through communication, which can take any form from a face-to-face discussion to electronically asynchronous messages such as email. Coupled with the fact that communication is inherently ambiguous and can often lead to misunderstandings, errors based on such misunderstandings may be introduced into the source code. Thus, we are confident that there exists a connection between developer communication and build success.

In this chapter, we start by describing the methodology that is relevant to exploring our research question (cf. Section 5.1). Then, in Section 5.2 we present our analysis and results followed by a discussion of the results in Section 5.3. We conclude this chapter by offering an answer to our research question.

5.1 Methodology

To address our research question, we analyze data from the large software development project IBM Rational Team Concert described in great detail in Chapter 4.

5.1.1 Coordination outcome measure

In our study, we conceptualize the coordination outcome by the Build Result, which is regarded as a coordination success indicator in Jazz and can be `ERROR`, `WARNING` or `OK`. We analyze build results to examine the integration outcomes in relation to the communication necessary for the coordination of the build.

Conceptually, the `WARNING` and `OK` build results are treated similar by the Jazz team, as they require no further attention or reaction from the developers. In contrast, `ERROR` build results indicate serious problems, such as compile errors or test failures, and require further coordination, communication and development effort. We thus treated all `WARNING`s as `OK`s to clearly separate between failed and successful builds in our conceptualization of coordination outcome.

5.1.2 Communication network measures

To characterize the communication structure represented by the constructed social networks for each build (cf. Chapter 3), we compute a number of social network measures. The measures that we include in our analysis are: Density, Centrality and Structural holes. Some of these measures characterize single nodes and their neighbours (ego networks), while others relate to complete networks. Because we are interested in analyzing the characteristics of complete communication networks associated to integration builds, we normalize and use appropriate formulas to measure the complete communication networks instead of measuring the individual nodes.

Density

Density is calculated as the percentage of the existing connections to all possible connections in the network. A fully connected network has a density of 1, while a network without any connections has a density of 0. For example, the density in the directed network in Figure 5.1 is the number of directed edges over the number of all possible directed edges $12/42 = 0.28$.

Centrality measures

We use the centrality measures *group degree centralization* and *group betweenness centralization* for complete networks, which are based on the ego network measures degree centrality and betweenness. The degree centrality measures for the ego networks are:

- The *Out-Degree* of a node c is the number of its outgoing connections $C_{oD}(c)$. For example, $C_{oD}(c_1) = 2$ in Figure 5.1.
- The *In-Degree* of a node c is the number of its incoming connections $C_{iD}(c)$. For example, $C_{iD}(c_1) = 1$ in Figure 5.1.
- The *InOut-Degree* of a node c is the sum of its In-Degree and Out-Degree $C_{ioD}(c)$. For example, $C_{ioD}(c_1) = 3$ in Figure 5.1.

To compute the *Group Degree Centralization* index for the complete network we use Equation 5.1 from Freeman [36], in which g is the number of nodes in a network, and $C_{*D}(c_i)$ are the degree centrality measures of node c_i as described above. $C_{*D}(c^*)$ is the largest Degree for the set of contributors in the network [39, 55].

$$C_{*D} = \frac{\sum_{i=1}^{g}[C_{*D}(c^*) - C_{*D}(c_i)]}{(g-1)^2} \tag{5.1}$$

Note that this formula can be used for each centrality. In the following, we compute the group degree centrality indices for each degree centrality measure for the social network shown in Figure 5.1:

- Out-Degree:
$$C_{oD} = \frac{\sum_{i=1}^{7}[C_{oD}(c_7) - C_{oD}(c_i)]}{(7-1)^2} = 0.056 \tag{5.2}$$
- In-Degree:
$$C_{oD} = \frac{\sum_{i=1}^{7}[C_{oD}(c_1) - C_{oD}(c_i)]}{(7-1)^2} = 0.25 \tag{5.3}$$
- InOut-Degree:
$$C_{oD} = \frac{\sum_{i=1}^{7}[C_{oD}(c_7) - C_{oD}(c_i)]}{(7-1)^2} = 0.306 \tag{5.4}$$

To calculate the *Group Betweenness Centralization* index for a whole network, we need to compute the betweenness centrality probability index for each actor of the network. The probability index assumes that a "communication" takes the shortest path from contributor

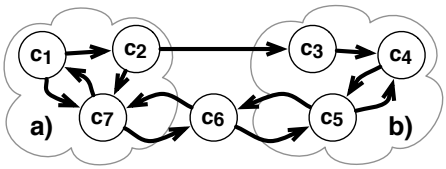


Figure 5.1: Example of a directed network to illustrate our social analysis measures.

c_j to contributor c_k and if the network has more shortest paths, all of them have the same probability to be chosen. If g_{jk} is the number of shortest paths linking two contributors, $1/g_{jk}$ is the probability of using one of the shortest paths for communication. Let $g_{jk}(c_i)$ be the number of shortest paths linking two contributors j and k that contain the contributors c_i. Freeman [36] estimates the probability that contributor c_i is between c_j and c_k by $g_{jk}(c_1)/g_{jk}$. The betweenness index for c_i is the sum of all probabilities over all pairs of actors excluding the ith contributor. Equation 5.5 shows the normalized betweenness index for directed networks.

$$C_B(c_i) = \frac{\sum_{j<k} g_{jk}(c_i)/g_{jk}}{(g-1)(g-2)} \tag{5.5}$$

The betweenness centrality for node c_1 in Figure 5.1 is computed as follows, note that the values for $g_{jk}(c_1)$ are shown in Table 5.1:

$$C_B(c_1) = \frac{\sum_{j<k} g_{jk}(c_1)/1}{(7-1)(7-2)} = 0.2 \tag{5.6}$$

To compute a betweenness index for the complete network instead of a single node, we used Freeman's equation for *Group Betweenness Centralization*, shown in Equation 5.7, in which $C_B(c^*)$ is the largest betweenness index of all actors in the network.

$$C_B = \frac{\sum_{i=1}^{g}[C_B(c^*) - C_B(c_i)]}{(g-1)} \tag{5.7}$$

Structural-holes

We use the following structural-hole measures:

- The *Effective Size* of a node c_i is the number of its neighbours minus the average degree of those in c_i's ego network, not counting their connections to c_i. The effective size of node c_1 in Figure 5.1a is $2-1=1$. Note, that only direct neighbours of c_1 are

Table 5.1: Listing the number of occurrences of c_1 on the shortest path between c_j and c_k with $j < k$ shown in Figure 5.1 with g_jk being one for each combination.

c_j	c_k	$g_{jk}(c_1)$
c_1	c_2	1
c_1	c_3	1
c_1	c_4	1
c_1	c_5	1
c_1	c_6	1
c_1	c_7	1
c_2	c_3	0
c_2	c_4	0
c_2	c_5	0
c_2	c_6	0
c_2	c_7	0
c_3	c_4	0
c_3	c_5	0
c_3	c_6	0
c_3	c_7	0
c_4	c_5	0
c_4	c_6	0
c_4	c_7	0
c_5	c_6	0
c_5	c_7	0
c_6	c_7	0

considered while the directed connections are replaced with undirected. The effective size of node c_4 in Figure 5.1b is $2 - 0 = 2$.

- The *Efficiency* normalizes the effective size of a node c_i by dividing its effective size with the number of its neighbours. The efficiency of node c_1 in Figure 5.1a is $(2 - 1)/2 = 0.5$. The efficiency of node c_4 in Figure 5.1b is $(2 - 0)/2 = 1$.

- *Constraint* is a summary measure that relates the connections of a node c_i to the connections of c_i's neighbours. If c_i's neighbours and potential communication partners all have one another as potential communication partners, c_i is highly constrained. If c_i's neighbours do not have other alternatives in the neighborhood, they cannot constrain c_i's behavior.

To calculate the network measures of structural-holes, we compute the sum of the measures for each node of a network. Since the measures are based on network connections,

we normalize the sum by computing the fraction of the sum and the number of possible network connections.

5.1.3 Data collection

We mined the Jazz development repository for build and communication information. A query plug-in was implemented to extract all development and communication artifacts involved in each build from the Jazz server. These build-related artifacts included build results, teams, change sets, work items, contributors, and comments. We imported the resulting data into a relational database management system in order to handle the data more efficiently.

We extracted a total of 1,288 build results, 13,020 change sets, 25,713 work items and 71,019 comments. Out of a total of 47 Jazz teams, 24 had integration builds. The build results we extracted were created during the time spanning from November 5, 2007 to February 26, 2008.

Our selection criterion was that we analyze a number of build results that are large enough for statistical tests and include both `OK` and `ERROR` builds. Some teams used the building process for testing purposes only and therefore only created a view build result, while others had either only `OK` or only `ERROR` build results. Predicting build results for a team that only produced `ERROR` builds in the past will most likely yield an `ERROR`, since no communication information representing successful builds is available. Thus, we considered teams that had more than 30 build results and at least 10 failed and 10 successful builds. Five teams satisfied these constraints and were considered in our analysis. In addition, we included the nightly, weekly, and one beta integration build, although they did not satisfy our constraints, because they integrate all subsystems of the entire project.

5.2 Analysis and Results

Table 5.2 shows descriptive statistics of the considered builds and related communication networks of the five teams (B, C, F, P and W in the first 5 columns) and the nightly, weekly, and beta project-level integrations. For example, team B created 60 builds from which 20 turned out to be `ERROR`s and 40 `OK`. The communication networks of this team had between 3 and 58 contributors (51.58 directed connections on average) and spanned 0 to 131 work items. On average, the builds involved on average 10.83 change sets.

Table 5.2: Descriptive build statistics

	Team Level Builds					Project Level Builds		
	B	C	F	P	W	nightly	weekly	beta
# Builds	60	48	55	59	55	15	15	16
# ERRORs	20	16	24	29	31	9	11	13
# OKs	40	32	31	30	24	6	4	3
# Contributors:								
Min	3	9	6	5	13	43	37	55
Median	6	16.5	18	15	20	55	57	69.5
Mean	12.68	18.02	20.15	17.98	22.87	57.93	52.27	67.81
Max	58	31	64	61	52	75	75	79
# Directed Connections:								
Min	0	1	2	0	11	81	56	144
Median	13	39.5	95	36	74	236	149	280
Mean	51.58	53.4	87.78	63	88.35	253.1	171.9	285.8
Max	361	139	355	401	300	434	496	446
# Change Sets:								
Min	1	15	8	32	83	80	62	82
Median	10	38	35	46	111	117	115	178.5
Mean	10.83	44.38	42.65	47.25	115.3	129	114.2	166.8
Max	33	101	91	75	156	199	173	196
# Work Items:								
Min	0	2	1	1	10	11	5	31
Median	6.5	12	20	12	18	67	51	98
Mean	16.43	15.56	23.07	19.34	29.49	72.13	56.87	96.81
Max	131	50	100	107	119	132	202	170

5.2.1 Individual communication measures and build results

To examine whether any individual measure of communication structure can predict integration failure or success, we analyzed the builds from each team and project-level integration in part in relation to the communication structure measures as follows: For each team we categorize the builds into two groups. One group contains the ERROR builds and the other the OK builds. For each build and associated communication network we compute the network measures described in Section 5.1 and compare them across the two groups of builds (i.e., ERROR and OK).

The communication measures used in the analysis were: Density, Centrality (in-degree, out-degree, inOut-degree, and betweenness), Structural-Holes (efficiency, effective size, and constraint), and number of directed connections. We used the Mann-Whitney test [103] to test if any of the measures differentiate between the groups of ERROR and OK related

Table 5.3: Classification results for team F

		prediction	
		OK	ERROR
actual	OK	26	5
	ERROR	9	15

communication networks. We used an α-level of $.05$ and applied the Bonferroni correction to mitigate the threat of multiple hypothesis testing. None of the tests yielded statistical significance, indicating that, non of the individual communication structure measures significantly differentiate between ERROR and OK builds.

Furthermore, we tested for the possible effect of the technical measures shown in Table 5.2: #Contributors, #Change Sets and the #Work Items on the build result. Our results showed that none of the tests yielded statistical significance to differentiate between ERROR and OK builds.

5.2.2 Predictive power of measures of communication structures

We combined communication structure measures into a predictive model that classifies a team's communication structure as leading to an ERROR or OK build. We explicitly excluded the technical descriptive measures such as #Contributors, #Change Sets and the #Work Items from the model in order to focus on the effect of communication on build failure prediction. We validated the model for each set of team-level and project-level networks separately by training a Bayesian classifier [48] and using the leave one out cross validation method [48].

In the case of team F's 55 build results, we train a Bayesian classifier with all but one of the 55 build results and their communication related network measures. Then we predict the build result for the build we left out and repeat this procedure such that we predict the build result for each build once. We tabulate the results of all predicting as in the case of team F in Table 5.3.

Table 5.3 shows the classification result for team F. The upper left cell represents the number of correctly classified communication networks as related to OK builds (26 vs. 31 actual), and the lower right cell shows the number of correctly classified networks as leading to ERROR builds (15 vs. 24 actual). The other two cells show the number of wrongly classified communication networks.

Table 5.4: Recall and precision for failed (ERROR) and successful (OK) build results using the Bayesian classifier

	Team Level Builds					Project Level Builds		
	B	C	F	P	W	nightly	weekly	beta
ERROR Recall	.55	.75	.62	.66	.74	.89	1	.92
ERROR Precision	.52	.50	.75	.76	.66	.73	.92	.92
OK Recall	.75	.62	.84	.80	.50	.50	.75	.67
OK Precision	.77	.83	.74	.71	.60	.75	1	.67

The classification quality is assessed via recall and precision coefficients, which can be calculated for ERROR and OK build predictions. We explain the coefficients for prediction of ERROR builds.

Recall is the percentage of correctly classified networks as leading to ERROR divided by the number of ERROR related networks. In Table 5.3 the lower right cell shows the number of correct classified networks that are leading to ERRORs, which is divided by the sum of the values in the lower row, representing the total number of actual ERRORs. This yields for Table 5.3 a recall of $15/(9 + 15) = .62$. In other words, 62% of the actual ERROR leading networks are correctly classified.

Precision is the percentage of as to ERROR leading classified networks that turned out to be actually ERRORs. In Table 5.3, it is the number of correctly classified ERRORs divided by the sum of the right column, which represents the number of as ERROR classified builds. In Table 5.3 the precision is $15/(5 + 15) = .75$. In practical terms, 75% of the ERROR predictions are actual ERRORs.

We repeated the classification described above for each team and project-level integration. Note that the model prediction results only show how the models perform within a team and not across teams. Table 5.4 shows the recall and precision values for as to OK and ERROR leading classified communication networks for each of the five team-level and three project-level integrations. Since we are interested in the power of build failure prediction, the error related values from our model are of greater importance to us. The ERROR recall-values (how many ERRORs were classified correctly) of team-level builds are between 55% and 75% and the recall values of the project-level builds are even higher with at least 89%. The ERROR precision values are equally high.

5.3 Discussion

In our analysis we examined the relationship between integration builds and measures of the related communication structure. We found that none of the single communication structure measures (i.e., density, centrality or structure hole measures) significantly differentiated between failed and successful builds at the team-level and project-level. Therefore, none of the individual communication structure measures could be used to predict build outcome.

In addition to the communication related measures, we also examined whether the technical measures we computed when constructing the communication networks – the number of change sets, contributors, and work items – have an impact on the integration build result, as they are an indication for the size and complexity of the development tasks to be coordinated. According to Nagappan and Ball [80], one might expect that increased size and complexity of code changes relate to more build failures. But in our study these single measures did not significantly differentiate between successful and failed build results. However, additional technical measures (i.e., Object Oriented metrics) that were used by Nagappan may prove to be good predictors in Jazz as well.

The second contribution of this work is the predictive model that utilizes measures of communication structures to predict build results. The combination of communication structure measures was a good predictor of failure even when the single measurements were not. Our model's precision in predicting failed builds, which relates to the confidence one can have in the predicted result, ranges from 50% to 76% for any of the five team-level integration builds, and is above 73% for the project-level integration builds.

We found that, for all prediction models, the recall and precision values are better than guessing. A guess is deciding on the probability of an `ERROR` or an `OK` build if the build fails or succeeds. The probability is the number of `ERROR`s or `OK`s divided by the number of all builds. For example, if we know that the `ERROR` probability is 50% and we guess the result of the next build we would achieve a recall and precision of 50%. In our case, our model reached an `ERROR` recall of 62% for team F, where as a guess would have yield only $24/55 = .44 = 44\%$ (cf. Table 5.2).

5.4 Summary

We conclude this chapter returning to the initial research question that we set out to answer:

RQ 1.1 Do Social Networks from repositories influence build success?

The results we presented in Section 5.2 show that our predictions, though not highly accurate, outperform random guesses. Therefore, we conclude that with recall of 55% to 75% and precision of 50% to 76%, depending on the development team, that communication indeed influences build success.

This finding opens the research avenue of investigating whether the manipulation of communication among software developers can yield positive results with respect to build success. This leads us to our next research focus, to search for places within the social networks that we should manipulate to stimulate build success. For this purpose, in the next chapter we shift our focus to the concept of socio-technical congruence which may help us to pinpoint the developers that should have communicated.

Chapter 6

Socio-Technical Congruence and Failure

Understanding that social networks effect build success prompts us to question how, or more precisely which parts of the social network, should be changed to increase the likelihood of success for a build. For this reason, we turn to the concept of socio-technical congruence, as it postulates that developers should communicate upon intersection of their work [66]. Thus, in this section, we explore the effect that socio-technical networks have on build success by answering the following research question:

RQ 1.2: Do Socio-Technical Networks influence build success?

Although socio-technical congruence has only been studied in connection with productivity, intuitively there should be a connection to software quality such as build success. For example, imagine two developers modifying classes that share call and data dependencies and one developer making changes that violate certain assumptions the other developer relies on when using the modified code. This might introduce an error that could have been prevented if both developers would have discussed their work. We hypothesize that the concept of socio-technical congruence relates to software quality as well as productivity, and might be useful in suggesting improvments in the social network by pointing out developers that should communicate. We analyze data from the Rational Team Concert (RTC) development team to evaluate our research question.

In this chapter, we begin by discussing how we calculate socio-technical congruence (cf. Section 6.1). Subsequently, we briefly discuss the methodology that is relevant to exploring our research question (cf. Section 6.2). Furthermore, we present the analysis and results we obtained in Section 6.3 followed by a discussion of the results in Section 6.4. We conclude this chapter by offering an answer to our research question and introducing the subsequent chapter (cf. Section 6.5).

- Communication—A communicates with B [17, 19, 26, 33].
- Location—A is in the same location as B [19, 33].
- Team structure—A is in the same team as B [19].

Figure 6.1: Examples of actual coordination

6.1 Calculating Congruence

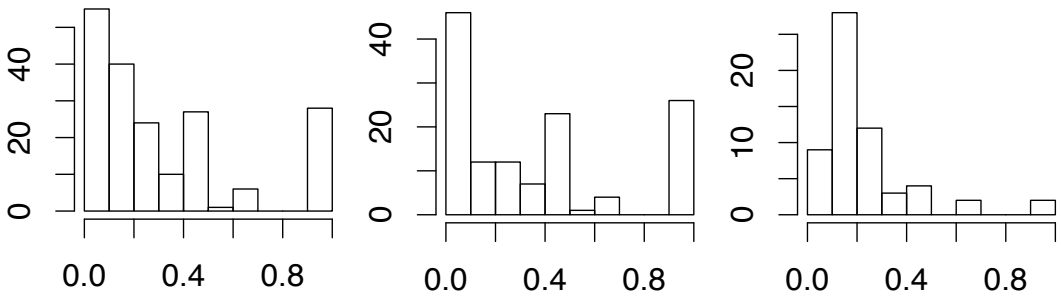
In Chapter 3 we described socio-technical networks and how we conceptualize them in this thesis. If we reformulate this network into the terms originally used by Cataldo et al [19], the matrix representation of the technical dependencies among software developers turns into the coordination needs matrix C and the social network in matrix representation is the actual coordination matrix A. Thus, we compute the socio-technical congruence index as follows:

$$\text{congruence} = \frac{\text{Diff(C, A)}}{|\text{C}|}$$

The main difference to the calculation proposed by Cataldo et al [19] lies solely in our more direct approach of deriving the coordination needs matrix instead of deriving them from task relationships, that are themselves derived from source code dependencies as we used to directly relate software developers with each other.

6.2 Analysis Methods

Logistic regression is ideal [102] to test the relationship between multiple variables and a binary outcome, which in our study is a build result being either “OK” or “Error”. The presence of many data entities in this project means that we must consider confounding variables in addition to the socio-technical congruence when determining its effects on the probability of build success. Informally, logistic regression identifies the amount of “influence” that a variable has in the probability that a build will be successful. The two main variables we are interested in are the socio-technical congruence index as well as the ratio between gaps and coordination needs, that is technical dependencies among developers that are not accompanies by a corresponding social dependency.



(a) All builds (b) OK builds (c) Error builds

Figure 6.2: Distribution of Congruence Values

We show the relationship between a variable and the build success probability by plotting the y-axis as the probability. We use probability because we feel that it is more intuitive than odds ratios or logistic functions. If there is a relationship between a variable and the probability of build success, then we should see that as the variable's value increases, the probability also increases. In the probability figures, the solid line is the expected value, and the dashed lines indicate the 95% confidence intervals.

We run a logistic regression model for congruence. We include the following variables: number of files per build, number of authors contributing to the build, number of files in the build, number of work items per build, the congruence, the build type, and the date of the build. We center and scale each numeric variable. Furthermore, because we were concerned about possible interactions affecting our results, we included first-order interaction effects and used backward stepwise elimination to remove variables to keep AIC (Akaike's Information Criterion) low.

6.3 Results

In the RTC repository, we analyzed 191 builds; of these builds, 60 were error builds, and 131 were OK builds. Table 6.1 displays summary statistics per build. Figure 6.2 displays histograms for congruence. The histograms compare the frequencies for each type of congruence for all of the builds, the OK builds, and the error builds only.

On average, the congruence values are low, with a mean value of 0.331. This demonstrates that about one-third of the coordination needs are satisfied by actual coordination.

Table 6.1: Summary statistics

	Min	Median	Max	Mean
Authors	2	17	44	18.62
Files	5	131	3101	342.3
change-sets	4	34	226	54.2
Work items	4	34	182	48.3
Build date range (days)	0	345	361	319.2
Congruence	0	0.21	1	0.331

Table 6.2: Pairwise Correlation of Variables per Build

	2.	3.	4.	5.	6.
1. Congruence	-0.27	-0.22	-0.33	0.08	0.19
2. Authors	–	0.41	0.76	0.10	-0.30
3. Files		–	0.37	0.08	-0.20
4. change-sets			–	0.02	-0.38
5. Work items				–	0.04
6. Build date					–

We calculated pairwise correlations between the variables congruence, number of authors, number of files, number of change-sets, number of work items, and build date, using a Spearman Rank Correlation Coefficient [102] (cf. Table 6.1). To avoid multicollinearity problems in our data, we choose to remove change-sets from our logistic regression analysis because, due to the enforced processes in RTC, we know that there is exactly one author per change-set, and thus there is at least as many change-sets as authors per build.

To assess the fit of the logistic regression models, we use the Nagelkerke pseudo-R^2 and AIC. R^2 shows the proportion of variability explained by the model, and AIC is a measure of how well the model fits the data. Ideally, R^2 is high and AIC is low. Our current model contains 19 variables and has an R^2 of 0.581. We present our model in Table 6.3 to a model containing every first-order interaction effect with 27 variables and a model that contains the 7 main effects only (in Table 6.5). We found that 19 variables are optimal and that removing further variables lowered the R^2 value while raising the AIC.

Table 6.3: Model comparison

Model	Variables	AIC	R^2
Every interaction	27	188.6	0.595
Main effects only	7	213.2	0.269
Our model	**19**	**175.8**	**0.581**

6.3.1 Effects of Congruence on Build Result

The result of logistic regression indicates that the following effects are significant: The congruence $\times$ build type effect, the congruence $\times$ build date interaction effect, the number of work items $\times$ build date interaction effect, and the build date $\times$ build type effect. In addition, the number of authors and the number of files are significant main effects, although their coefficients are lower than the interaction effects involving congruence.

In the next section, we discuss the main effects and interactions, effects that involve congruence affecting build probability. We discuss the effects of the non-congruence effects, including the authors, files, work items$\times$date interaction effect and the date$\times$nightly build effect in Section 6.3.3.

Effects of interactions involving congruence

The type $\times$ congruence interaction effect, the date $\times$ congruence interaction, and the type $\times$ date effect are each significant in our model (cf. Table 6.4). In Figure 6.3, we plot the effects of congruence vs. probability of build success at the 10% date quantile (2008-01-25), at the 25% date quantile (2008-05-14), the 50% date quantile (2008-06-07), and the latest build (2008-06-26).

The congruence model (cf. Table 6.4) the effect of congruence on continuous builds is significant, and that increasing congruence also increases the probability that a continuous build will succeed. For integration builds (cf. Figures 6.3), an increase in congruence decreases build success, with the exception of the 2008-01-25 build (Figure 6.3a). In our 2008-01-25 build, we see that low congruence leads to low build probability, but high congruence has high build probability. As the project ages, this trend reverses and congruence is clearly inversely related to build success probability (cf. Figure 6.3d). The effect of congruence is totally opposite for continuous builds and integration builds. Based on Figure 6.3d, increasing congruence significantly improves the continuous build success rate. However, increasing congruence significantly decreases the integration build success rate.

Table 6.4: Logistic Regression models predicting build success probability with main and interaction effects

Variable	Coef.	S.E.	p
Intercept	-0.5459	0.4663	0.2417
Congruence	**6.3410**	**1.6262**	****0.0001**
Authors	**-1.9759**	**0.5310**	****0.0002**
Files	**-1.0734**	**0.4561**	***0.0186**
Work items	-0.1456	0.2355	0.5363
Build type=I	**2.1533**	**1.0526**	***0.0408**
Build type=N	4.6833	200.7587	0.9814
Build date	-0.6560	0.6709	0.3282
Congruence * Build type=I	**-9.2151**	**2.5572**	****0.0003**
Congruence * Build type=N	-7.7308	91.8053	0.9329
Congruence * Build date	**-5.1266**	**1.9290**	****0.0079**
Authors · Build type=I	1.2688	0.7028	0.0710
Authors * Build type=N	105.4123	535.8792	0.8441
Authors * Build date	-0.6061	0.3616	0.0937
Authors * Files	0.7663	0.4289	0.0740
Files * Build type=I	1.0920	1.1838	0.3563
Files * Build type=N	-37.9274	199.2314	0.8490
Work items * Build date	**0.8040**	**0.3003**	****0.0074**
Build type=I * Build date	**2.6442**	**0.7678**	***0.0006**
Build type=N * Build date	84.7252	344.8129	0.8059
Model likelihood ratio	101.92		$R^2 = 0.581$
	191 observations		
	Build type is set to continuous		
$^*p < 0.05$; $^{**}p < 0.01$	Nagelkerke is used as the pseudo-R^2 measure		

6.3.2 Effect of Gap Ratio on Build Result

We build a logistic regression model based on the model in Table 6.4 using the gap ratio measurement (percentage of unmet coordination needs). In the interest of saving space, we report only the odds ratio. We retain every significant interaction from our previous congruence logistic regression in Table 6.4.

The effect of gap ratio on build result is significant (cf. Table 6.6). This indicates that increasing the gaps ratio significantly increases the odds that an OK build will occur, which is the opposite of what we hypothesized (cf. Figure 6.5). This means that if the gap is large, the build success probability increases.

Table 6.5: Logistic Regression models predicting build success probability with main effects only

Variable	Coef.	S.E.	p
Intercept	0.5265	0.3040	0.0833
Congruence	0.9371	0.6807	0.1686
Authors	**-0.5702**	**0.2003**	****0.0044**
Files	**-0.6398**	**0.2477**	****0.0098**
Work items	-0.1755	0.1713	0.3055
Build type=I	0.1693	0.4269	0.6917
Build type=N	0.2133	0.7791	0.7842
Build date	-0.1331	0.1821	0.4649
Model likelihood ratio	40.59		$R^2 = 0.269$
		191 observations	
	Build type is set to continuous		
$^*p < 0.05$; $^{**}p < 0.01$	Nagelkerke is used as the pseudo-R^2 measure		

6.3.3 Social and Technical Factors in RTC Affecting Build Success and Congruence

In light of our results, we not only examined the number of work items $\times$ date significant interaction found in Section 6.3.1, but different social and technical factors that may affect congruence and build success probability to find explanations for the interactions between socio-technical congruence and build success probability in RTC. Specifically, we examined the effect of build date on work items, coordination around fully-congruent builds and incongruent builds, and the effect of commenting behaviour on builds.

Other Effects on Build Success

Authors As the number of authors involved in a build increases, the probability that the build succeeds decreases. The build probability is significantly lowered after more than 15 authors are involved in the build (cf. Figure 6.6a). When over 30 authors are involved in the build, the estimated build success probability falls below 10%.

Files As the number of files involved in a build increases, the probability that the build will succeed decreases (cf. Figure 6.6b).

Build Date and Work items The work items $\times$ date interaction is significant. Early on in the project, as the number of work items increases, the probability of build success

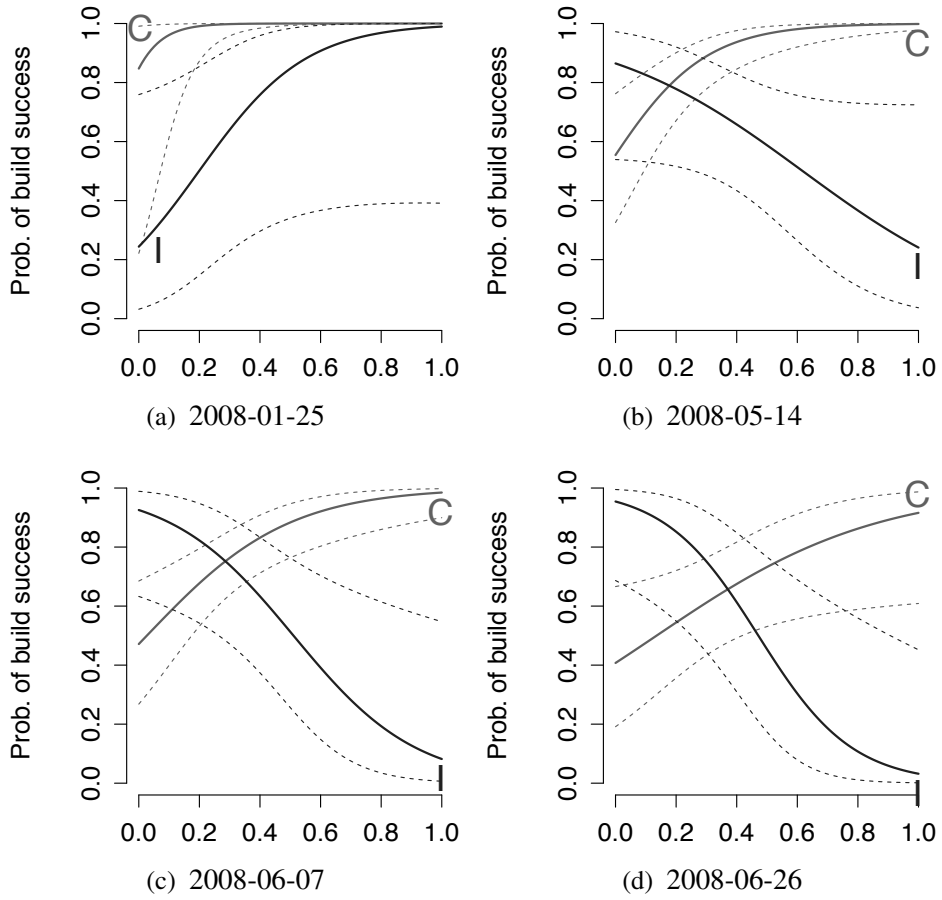


Figure 6.3: Estimated probability of build success for *congruence* and *continuous builds C* or *integration builds I* over time, adjusted to authors ≈ -0.156 (17 authors), files ≈ -0.352 (131 files), work items ≈ -0.399 (34 work items)

decreases (cf. Figure 6.7a). As the project ages, this trend reverses, and as the number of work items increases, the probability of build success increases as well (cf. Figure 6.7b). According to the coefficients in Table 6.4, this effect on build success probability is not as strong as the author's main effect or the files main effect.

Examining Extreme Congruence Values

We are interested in the differences between high-congruence builds and low-congruence builds. We further this investigation by looking at builds that have extreme values of congruence: zero, where absolutely no coordination needs are satisfied with communication, and one, where every coordination need is satisfied with communication. We chose to investigate the extreme cases to see if there were differences in the way people coordinated

Table 6.6: Odds Ratio for Gap Ratio Models

	Model
Intercept	1.32
Authors	0.60
Files	0.63
Work items	0.85
Build type=I	1.31
Gap ratio	8.71
Build date	0.59
Authors * Build date	0.74
Work items * Build date	1.83
Build type=I * Build date	2.52

Table 6.7: Number of Builds with Congruence Values 0 and 1

	Congruence	
	1	0
OK	26	30
ERR	2	2

in fully-congruent builds, and in incongruent builds. Table 6.7 shows the number of OK and error builds that occurred when congruence was equal to one, and equal to zero.

To determine if the presence of commenting affected the builds, we examined the number of comments on work item–change-set pairs in builds with extreme congruence values. Our results are shown in Table 6.8. Build success probabilities improve with respect to builds that have no comments, though work items with no comments are in the minority.

Of note is the high number of comments on work items that have zero congruence. This indicates that individuals who have no technical relationship to the work item are commenting on the work item.

We manually inspected the work items with extreme amounts of congruence, reading the comments for any differences in the content discussed. Unfortunately, there were no obvious qualities between comments made in a build with a congruence of zero, and comments made in a build with a congruence of one. In both builds, individuals discussed technical implementation details, provided updates to colleagues, or requested assistance from colleagues. We are unable to discover root causes of failure without a deeper examination of the technical changes and more knowledge of the RTC context.

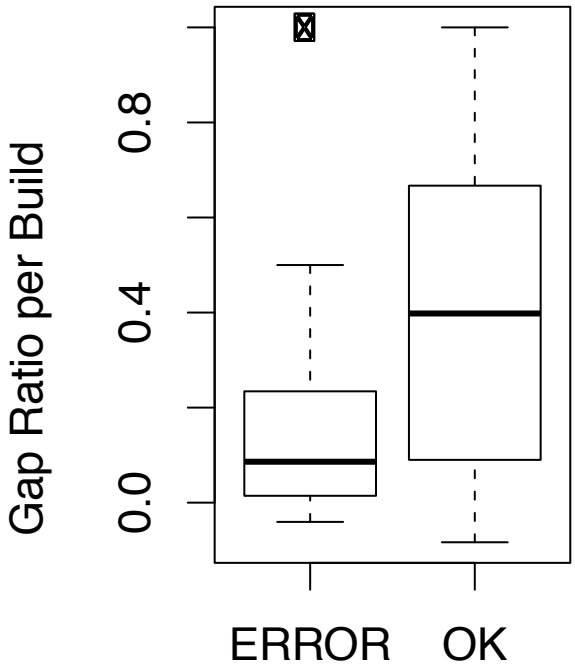


Figure 6.4: Gap Ratio per Build

6.4 Discussion

The concepts illustrated in Conway's Law, as well as previous empirical work on socio-technical congruence, lead us to expect that team members must coordinate according to coordination needs suggested by technical dependencies in order to build software effectively. In this case study, we applied socio-technical congruence to study coordination and its relationship to build success probability in RTC.

Although Conway made his observation before the emergence of programming languages that were design to define interfaces to enable better decoupling of program modules, these advances in programming languages and processes help alleviate issues with module interaction but the underlying issue that Conway observed is still prevalent. The issue Conway observed is that communication structures of organizational structures will lead any software architecture over time to reflect upon that communication structure. Although new programming languages can help smooth interaction across modules, it is not always possible to modularize software such that modules can be assigned to teams that are only part of one organizational unit. Thus, with the evidence found by Cataldo et al. [19], we think that Conway's Law still applies.

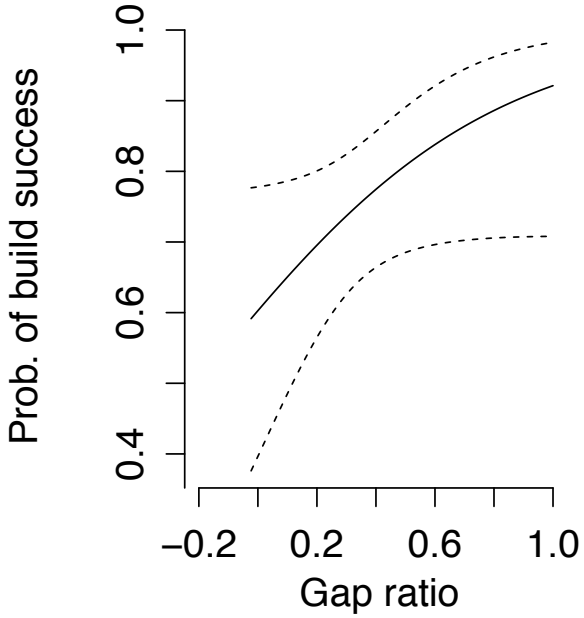


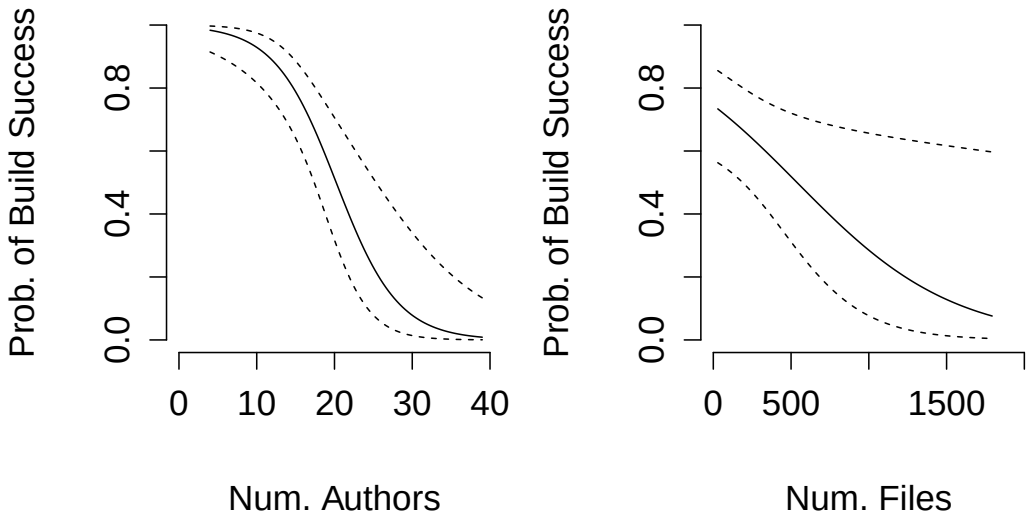
Figure 6.5: Effect of gap ratio on build success probability.

Overall, we found that the average congruence across builds was very low—only 20–30% of the coordination needs in the project were fulfilled with actual coordination. Even in the case of zero congruence, the build result was an OK build in over 90% of the observed cases.

We found that there was an interaction effect involving congruence and build type on build success probabilities (cf. Section 6.3.1). For continuous builds, increasing congruence improves the chance of build success in continuous builds and can actually decrease build success probability in integration builds (cf. Figures 6.3). Congruence significantly reduces integration build success probability.

The gap ratio is a representation of whether enough coordination occurred to fulfill multiple coordination needs. If two developers have multiple dependencies with each other, one would expect them to coordinate more often. We hypothesized that a small gap ratio would increase the probability of successful builds and that a large gap ratio would decrease the probability of a successful build. Instead, we found that as the mean gap increases, the build success probability also increases (cf. Figure 6.5).

Below we discuss the reasons for these observed results based on our knowledge of RTC.



(a) Authors (b) Files

Figure 6.6: Estimated probability of build success for *authors* and *files*, congruence. Adjusted to work items ≈ -0.399 (34), authors ≈ -0.156 (17), files ≈ -0.352 (131), congruence ≈ 0.1446, type = cont, date=2008-06-26

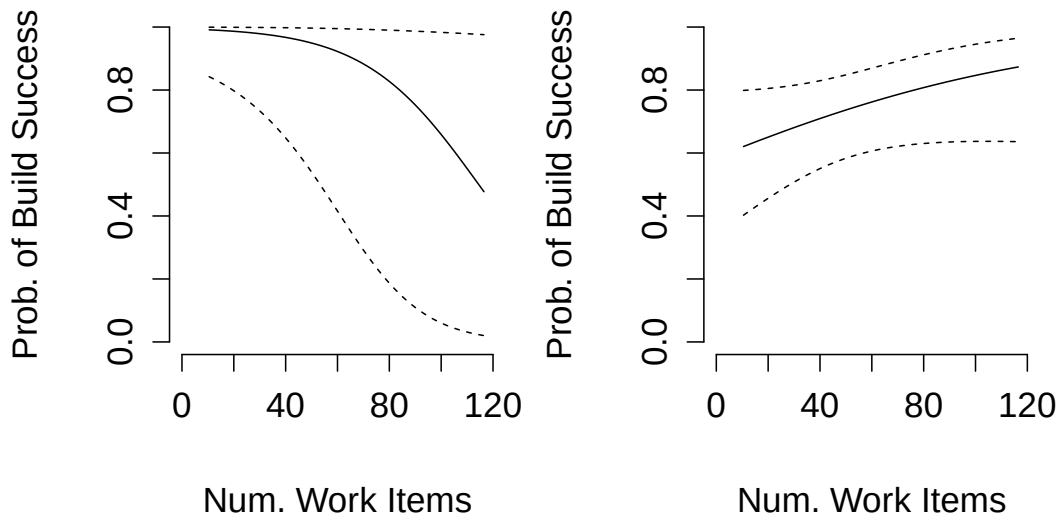
6.4.1 Strong Awareness Helps Coordination

The overall congruence for the majority of builds is low: over 75% of builds have a congruence of less than 0.25 (cf. Section 6.3.1). Despite low congruence, the RTC team is able to successfully build its software in many situations.

If socio-technical congruence is a measure of coordination quality in software, and builds rely on coordination quality to be successful, then there must be reasons why builds can succeed even when the congruence is zero. Because RTC is a highly distributed project, the product under development uses a modular design [71] and is therefore affected less by dependencies. Second, team members in RTC do not conduct all of their coordination through *explicit communication* even though work item inspection and discussion with developers indicate that the RTC corporate culture focuses on the work item as their base for communication. Rather, they use the *shared workspace* that incorporates cues from the environment and from peers in order to address technical issues. Both of these effects may contribute to congruence being lower than expected.

RTC supports Explicit Communication

The RTC team members use the RTC environment extensively to communicate with each other. We were informed that RTC team members rarely use private email, and our inspec-



(a) 2008-01-25 (b) 2008-06-26

Figure 6.7: Estimated probability of build success for *work items* and *date*, congruence. Adjusted to authors ≈ -0.156 (17), files ≈ -0.352 (131), congruence ≈ 0.1446, type = cont

tion of the mailing list reveals that its primary purpose is for announcements such as server outages rather than for discussing technical work.

This leaves the RTC work item comment system and instant messaging as methods for communication, as well as the phone and internal face-to-face meetings. We learned that while face-to-face interaction is efficient for solving local issues, it does not benefit remote teams, and the RTC team as a whole encourages every team member to record face-to-face discussions as comments for the purpose of archiving and sharing information.

However, explicit coordination comes at a cost. There is evidence that involving too many authors in the same build also reduces the build success (cf. Figure 6.6a). The overhead required to coordinate many people may interfere with the ability of the team to build the project successfully. This suggests that there is a limit at which point a developer is overloaded with information.

RTC is a Shared Workspace

The RTC client software helps a developer acquire and maintain *environmental awareness* in terms of what is taking place in the project by providing access to a shared workspace. Much of the work is centered on the RTC technical entities, which include plans, source code, work items, and comments. RTC's awareness mechanisms feature a developer-centered dashboard that reports changes to the workspace, built-in traceability, user notifi-

Table 6.8: Number of work items-change-set pairs with comments and build success probabilities for congruence 0 and 1

		Num. of Pairs		Success rate	
Congruence		1	0	1	0
No comments	OK	42	143	49%	69%
	Error	43	64		
Comments	OK	610	445	68%	69%
	Error	290	199		
Total	OK	652	588	66%	69%
	Error	333	263		

cations, regularly generated reports, and an optional web browser interface. For example, when a change-set is created, it is attached to a work item, thus ensuring that people who are involved with the work item receive notification of this change-set. These automatic notifications cut down the amount of explicit communication and allow people to coordinate implicitly.

Coordinating using the workspace is well known in the computer-supported cooperative work domain [99]. In particular, open-source developers, coordinate around source code [12] and mailing lists [44, 77] because there is little opportunity for face-to-face interaction. RTC shares many characteristics with open-source development, such as a distributed team and a transparent development process.

In light of these results, we believe that, using our conceptualization, the RTC team requires a congruence of only 0.2–0.3 for their tasks to be completed. Much of the need for explicit, point-to-point communication is mitigated by implicit communication and the use of the workspace to coordinate. We expect that the remaining congruence is covered through the RTC workspace, and through face-to-face communication, instant messenger, and phone communication. Though our congruence value appears low for the RTC team, the coordination in reality may be higher. Future studies should keep in mind that congruence may be lower than expected because of conceptualizations that cannot include every type of coordination in a project.

6.4.2 Coordination and Geographic Distribution

As RTC is a distributed team, geographic distribution has an effect on team performance, though the RTC environment helps mitigate some of these effects [84].

From the RTC project, we learned that continuous and nightly builds should involve mainly a co-located team, and that integration builds involve multiple components from RTC teams in different locations. Our results suggest that congruence best benefits builds that occur within co-located teams. However, the design of our study does not allow us to draw a definite conclusion about the influence of both co-location and congruence on build success probability.

It appears that involving too many individuals when coordinating the activities of various teams may harm build success due to information overload [25], especially when the team members are distributed. To negate this effect, development leaders and build managers that have an overall view of the project are suited to coordinate teams in order to ensure build success [55].

6.4.3 Project Maturity and Build Success

We found that early builds exhibited a different type of relationship between congruence and build success probability than later builds (cf. Section 6.3.1). Over the course of the study, we observed 13 internal milestones; the last milestone in our observed builds was a public beta release for end users.

There are three reasons for a build to fail (1) compilation errors, (2) linking errors, and (3) test errors. We did not observe any compilation failures, as developers cannot check in the code that contains errors. Although Java does not produce linking errors, updating API's that are only made available during integration builds can result in similar errors. Due to the maturity of the project the build failures we observed were found to be caused by test errors.

Build success probability decreased significantly over time for continuous builds and stayed roughly the same for integration builds (cf. Figures 6.3). However, the early builds in the project behaved contrary to the later builds (cf. Figure 6.3a and 6.3a). Early in its lifetime, the RTC software is in s state of change. Integration builds are not a priority, and features are being added to the project. This means that dependencies are changing rapidly, as well as the expertise among team members, making it difficult to solidify coordination needs.

In addition to interactions between congruence and type, and congruence and date, we observed a significant interaction effect between build date and work items. We found that early in the project (cf. Figure 6.7a), builds with large numbers of work items have a high probability of failing, but late in the project (cf. Figure 6.7b), these builds succeed. Because

the latest release was focused on a public release, a build linked to numerous work items may indicate that a bug is highly problematic, or a feature is highly desired, and therefore received more attention.

6.5 Summary

We end this chapter by bringing it back to the initial research question we set out to answer:

RQ 1.2: Do Socio-Technical Networks influence build success?

We conducted two investigations: (1) the investigation of the influence of the socio-technical congruence index on build success and (2) the investigation of gaps uncovered by socio-technical networks and their influence on build success. Both avenues of investigations demonstrated that they expose an influence on build success.

These findings, specifically the idea that gaps within socio-technical networks influence build success, opens the door to formulating an approach on how to leverage the concept of socio-technical congruence to improve the communication among software developers. In the next chapter, we detail an approach that we propose to improve developer communication using the concept of socio-technical congruence.

Chapter 7

A Socio-Technical Congruence Approach to Improve Build Success

In this chapter, we propose an approach to leverage the concept of socio-technical congruence to better communication (social interactions) among developers in an attempt to improve build success. We base this approach on the findings presented in the previous two chapters that team communication (cf. Chapter 5) and socio-technical gaps (cf. Chapter 6) influence build success.

7.1 Overview

The main contribution of this thesis lies with the approach to creating actionable knowledge from the socio-technical congruence concept as Cataldo et al. [19] described it in their seminal paper. Thus, we derived the following approach from the first two research questions:

Our approach encompasses five steps:

1. Define scope of interest
2. Define outcome metric
3. Build social networks
4. Build technical networks
5. Generate actionable insights

7.2 Define Scope

Each network, social, technical and thus socio-technical, needs to be built with a specific scope in mind. For example, throughout this thesis we focus on software builds. The focus defines the source and communication artifacts used to construct the social and technical networks. In Chapter 3 Figure 3.2 we demonstrated how we conceptualized social networks and explained how defining software builds let us select communication artifacts for the construction of a social network for a specific build.

7.3 Define Outcome

In order to derive useful insight from the constructed networks, the scope used to construct them needs to be complemented with an outcome metric. For example, in this thesis we are interested in build success, a binary variable that states whether a build is of acceptable quality. With an outcome measure at hand we can contrast networks to gain insights. Note, that the outcome and scope are closely related, as we need to attach the outcome to the social and technical network whose construction depends on the scope.

7.4 Construct Social Networks

After defining the scope and outcome, the next step is to construct the social networks. This involves identifying all communication channels that are programmatically accessible in real time. Some examples of communication channels that can be tapped into in real time are emails, forum style discussions, or text chats. In Chapter 3 Figure 3.2 shows a detailed example of how we construct a social network given a build as the scope of interest using data from the Rational Team Concert development team.

7.5 Construct Technical Networks

To complement the social networks, and thus create socio-technical networks, we need to produce technical networks. The main issue is not to rely on information that is only available after the work has been completed and been submitted to a version repository, but to gather information to construct networks in real time. For instance, Blincoe et al. [9]

proposed to use Mylyn[1] events to accomplish the extraction of interactions with source code in real time. With respect to software quality it suffices to analyze complete changes and give feedback once the implementation work has been submitted to a central code repository.

7.6 Generate Insights

In Chapter 6, we showed that gaps in socio-technical networks influence build success. We use this as a starting point for generating insights by breaking down socio-technical networks into triples that consist of a developer pair together with how they are connected, which can be either through a technical dependency, a social dependency, both, or none.

Since we are focusing on improving the social interactions we focus on identifying the gaps (unmet coordination needs) that are most likely to increase build success. For this purpose we split the triples derived from all socio-technical networks into two groups, those triples that originate from a socio-technical network of a failed build and those that originate from a socio-technical network of a successful build.

For each triple we can now apply statistical tests to see if a given triple is more often observed with failed or successful builds. Suggesting to developer to communicate should break those that are statistically significant for failed builds. To ensure not to worsen the odds of a successful build we contrast the developer pair forming the gap with the same pair being connected both technically and socially.

7.7 Summary

The approach we described builds on our work on software builds and the influence of communication and unmet coordination needs as shown in the previous two chapters. Note that this approach can also be applied to identify those met coordination needs that warrant further examination, by applying the same insight generation to pairs of developers that are connected both through a technical and social dependency.

The following three chapters are aimed at identifying the usefulness of the approach we detailed in this chapter. We start by first investigating whether we can use this approach to generate recommendations that are statistically significant (cf. Chapter 8). Following, we diverge from the path of exploring the validity of the recommendations and focus on

[1] `http://tasktop.com/mylyn/`

whether developers are open to such recommendations (cf. Chapter 9). Finally, in a large student project in a course offered at the University of Victoria, Canada and Aalto University, Finland, we explore whether we could generate recommendations that might have prevented builds from failing (cf. Chapter 10).

Part III

Applying our Approach

Chapter 8

Leveraging Socio-Technical Networks

The first step towards exploring the usefulness of our approach is to test whether we can generate recommendations that break patterns related to build failure. In order to do this, we focus on the following research question:

RQ 2.1: Can Socio-Technical-Networks be manipulated to increase build success?

Intuitively, the idea is that two developers that share a technical dependency should talk, but in some cases when the technical relationship is trivial, instead of preventing failures such a recommendation might only create additional communication issues. Thus, we explore to what extent the history of a project can be used to highlight the cases where developers that shared an unmet coordination need are found more often to failed builds.

In this chapter, we start by briefly presenting the methodology that is relevant to exploring our research question (cf. Section 8.1 and 8.2). Then, we present the analysis and results we obtained in Section 8.3 followed by a discussion of the results Sections 8.4. In conclusion of this chapter, we offer an answer to our research question (cf. Section 8.5).

8.1 Socio-Technical Coordination

We build our socio-technical networks according to the approach outlined in Chapter 7:

1. Define scope of interest
2. Define outcome metric
3. Build social networks
4. Build technical networks

Table 8.1: Contingency table for technical pair (Adam, Bart) in relation to build success or failure

	successful	failed
(Adam, Bart)	3	13
¬ (Adam, Bart)	224	86
total	227	99

As our scope, we select the software build and as outcome metric we use the build outcome. We construct the social and technical networks from the Rational Team Concert development team repository as outlined in Chapter 2.

8.2 Analysis of Socio-Technical Gaps

The lack of communication between two developers that share a technical dependency is referred to in the literature as a socio-technical gap [106]. Because research suggests negative influence of such gaps, we are interested in analyzing pairs of developers that share a technical edge (implying coordination need), but not a social edge (implying unmet coordination need), in socio-technical networks. We refer to these pairs of developers as *technical pairs* (there is a gap), and to those who do share a socio-technical edge (there is no gap) as *socio-technical pairs*.

We analyze the technical pairs in relation to build failure. Our analysis proceeds in four steps:

1. Identify all technical pairs from the socio-technical networks.
2. For each technical pair count occurrences in socio-technical networks of builds that failed.
3. For each technical pair count occurrences in socio-technical networks of successful builds.
4. Determine if the pair is significantly related to success or failure.

For example, in Table 8.1 we illustrate the analysis of the technical pair (Adam, Bart). This pair appears in three successful builds and in 13 failed builds. Thus, it does not appear in 224 successful builds, which is the total number of successful builds minus the number

Table 8.2: Twenty most frequent *technical pairs* that are failure-related

Pair	#successful	#failed	p_x
(Cody, Daisy)	0	12	1
(Adam, Daisy)	1	14	0.9697
(Bart, Eve)	2	11	0.9265
(Adam, Bart)	3	13	0.9085
(Bart, Cody)	3	13	0.9085
(Adam, Eve)	4	16	0.9016
(Daisy, Ina)	3	12	0.9016
(Cody, Fred)	3	10	0.8843
(Bart, Herb)	3	10	0.8843
(Cody, Eve)	5	15	0.8730
(Adam, Jim)	4	11	0.8631
(Herb, Paul)	5	12	0.8462
(Cody, Fred)	5	11	0.8345
(Mike, Rob)	6	13	0.8324
(Adam, Fred)	6	13	0.8324
(Daisy, Fred)	8	13	0.7884
(Gill, Eve)	7	10	0.7661
(Daisy, Ina)	7	10	0.7661
(Fred, Ina)	8	10	0.7413
(Herb, Eve)	8	10	0.7413

of successful builds the pair appeared in, and it is absent in 86 failed builds. A Fischer Exact Value test yields significance at a confidence level of $\alpha = .05$ with a p-value of $4.273 \cdot 10^{-5}$.

Note that we adjust the p-values of the Fischer Exact Value test to account for multiple hypotheses testing using the Bonferroni adjustment. The adjustment is necessary because we deal with 961 technical pairs that need to be tested.

To enable us to discuss the findings, and to look at whether closing socio-technical gaps are needed to avoid build failure, or which of these gaps are more important to close, we perform two additional analyses. First, we analyze whether the socio-technical pairs also appear to be build failure-related or not, by following the same steps as above for socio-technical pairs. Secondly, we prioritize the developer pairs using the coefficient p_x, which represents the normalized likelihood of a build to fail in the presence of the specific pair:

$$p_x = \frac{\text{pair}_{\text{failed}}/\text{total}_{\text{failed}}}{\text{pair}_{\text{failed}}/\text{total}_{\text{failed}} + \text{pair}_{\text{success}}/\text{total}_{\text{successs}}} \tag{8.1}$$

Table 8.3: The 20 most frequent statistically failure related technical pairs and the corresponding socio-technical pairs

Pair	#successful	#failed	p_x
(Cody, Daisy)	—	—	—
(Adam, Daisy)	—	—	—
(Bart, Eve)	1	4	0.9016
(Adam, Bart)	—	—	—
(Bart, Cody)	—	—	—
(Adam, Eve)	—	—	—
(Daisy, Ina)	—	—	—
(Cody, Fred)	1	0	0
(Bart, Herb)	1	2	0.8209
(Cody, Eve)	0	3	1
(Adam, Jim)	0	1	1
(Herb, Paul)	1	0	0
(Cody, Fred)	—	—	—
(Mike, Rob)	—	—	—
(Adam, Fred)	—	—	—
(Daisy, Fred)	—	—	—
(Gill, Eve)	—	—	—
(Daisy, Ina)	1	0	0
(Fred, Ina)	0	2	1
(Herb, Eve)	—	—	—

The coefficient is comprised of four counts: (1) $\text{pair}_{\text{failed}}$, the number of failed builds where the pair occurred; (2) $\text{total}_{\text{failed}}$, the number of failed builds; (3) $\text{pair}_{\text{success}}$, the number of successful builds where the pair occurred; (4) $\text{total}_{\text{success}}$, the number of successful builds. A value closer to one means the developer pair is strongly related to build failure. We are using the percentages of failed and successful pairs in this equation to avoid an imbalance between successful and failed builds. In our study the number of successful builds was twice as large as the number of failed builds and thus the occurrences in successful builds might be skew the coefficient towards a lower failure likelihood.

8.3 Results

We found a total of 2,872 developer pairs in all the constructed socio-technical networks, out of which 961 were technical pairs. We choose to present the twenty that are most frequent across failed builds.

Table 8.4: Logistic regression only showing the technical pairs from Table 8.2, the intercept, and the confounding variables, the model reaches an AIC of 706 with all shown features being significant at $\alpha = 0.001$ level (indicated by ***).

Feature	Coefficient	p-value	
(Intercept)	7.897e+74	$<$ 2e-16	***
(Cody, Daisy)	-5.669e+75	$<$ 2e-16	***
(Adam, Daisy)	-9.846e+75	$<$ 2e-16	***
(Bart, Eve)	-1.258e+75	$<$ 2e-16	***
(Adam, Bart)	-1.605e+76	$<$ 2e-16	***
(Bart, Cody)	-3.419e+76	$<$ 2e-16	***
(Adam, Eve)	-2.610e+76	$<$ 2e-16	***
(Daisy, Ina)	-8.105e+74	$<$ 2e-16	***
(Cody, Fred)	-5.348e+76	$<$ 2e-16	***
(Bart, Herb)	-2.977e+76	$<$ 2e-16	***
(Cody, Eve)	-2.315e+76	$<$ 2e-16	***
(Adam, Jim)	-2.724e+76	$<$ 2e-16	***
(Herb, Paul)	-1.636e+76	$<$ 2e-16	***
(Cody, Fred)	-1.645e+74	$<$ 2e-16	***
(Mike, Rob)	-1.327e+75	$<$ 2e-16	***
(Adam, Fred)	-5.250e+76	$<$ 2e-16	***
(Daisy, Fred)	-2.455e+75	$<$ 2e-16	***
(Gill, Eve)	-7.162e+75	$<$ 2e-16	***
(Daisy, Ina)	-5.325e+74	$<$ 2e-16	***
(Fred, Ina)	-2.777e+75	$<$ 2e-16	***
(Herb, Eve)	-1.799e+75	$<$ 2e-16	***
#Change Sets per Build	6.480e+60	$<$ 2e-16	***
#Files changed per Build	-4.530e+60	$<$ 2e-16	***
#Developers contributed per Build	3.386e+61	$<$ 2e-16	***
#Work Items per Build	-3.690e+61	$<$ 2e-16	***

We rank the failure relating *technical* pairs (cf. Table 8.2) by the coefficient p_x. This coefficient indicates the strength of relationship between the developer pair and build failure. For instance, the developer pair (Adam, Bart) appears in 13 failed builds and in 3 successful builds. This means that pair$_{failed}$ = 13 and pair$_{success}$ = 3 with total$_{failed}$= 99 and total$_{success}$= 227 result in p_x= 0.9016. Besides that, we report the number of successful builds the pair was observed with (#successful), as well as the number of failed builds the pairs was observed with (#failed). The p_x values are all above 0.74, which implies that the

likelihood of failure is at least 74% in all builds in which these developer pairs are involved.

We then checked for the 120 pairs whether the corresponding *socio-technical* pairs are related to failure. Only 23 of the 120 technical pairs had an existing corresponding socio-technical pair of which none were statistically related to build failure. In Table 8.3 we show the socio-technical pairs that match the 20 technical pairs shown in Table 8.2 as well as the same information as in Table 8.2. If the corresponding socio-technical pair existed we computed the same statistics as for the technical pairs, but for those that existed we could not find statistical significance. Note that we use fictitious names for confidentiality reasons.

The failure-related technical pairs span 48 out of the total 99 failed builds in the project. Figure 8.1 shows their distribution across the 48 failed builds. The histogram illustrates that there are few builds that have a large number of failure related builds (e.g. 4 with 18 or more pairs) but most builds only show a small number of pairs (15 out of 48 failed builds have 4 or less). This distribution of technical pairs indicates that the developer pairs we found did not concentrate in a small number of builds. In addition, it validates the assumption that it is worthwhile to seek insights about developer coordination in failed builds.

8.4 Discussion

These results show that there is a strong relationship between certain technical developer pairs and increased likelihood of a build failure. Out of the total of 120 technical pairs that increase the likelihood of a build to fail, only 23 had an existing corresponding socio-technical pair. Of these, none were statistically related to build failure. This means that 97 pairs of developers that had a technical dependency did not communicate with each other and consequently increased the likelihood of a build failure. Our results not only corroborate the past findings [17, 19] that socio-technical gaps have a negative effect in software development, but more importantly, they indicate that the analysis presented in this paper is able to identify the specific socio-technical gaps, namely the actual developer pairs where the gaps occur and increase the likelihood of build failure.

A preliminary analysis of developer membership to teams shows that most of the technical pairs related to build failure consist of developers belonging to different teams. Nagappan et al. [82] found that using the organizational distance between people predicts failures. They reasoned that this is due to the lack of awareness that people separated by organizational distance work on. Although the Rational Team Concert team strongly emphasizes communication regardless of team boundaries, it still seems that organizational distance has an influence on its communication behaviour.

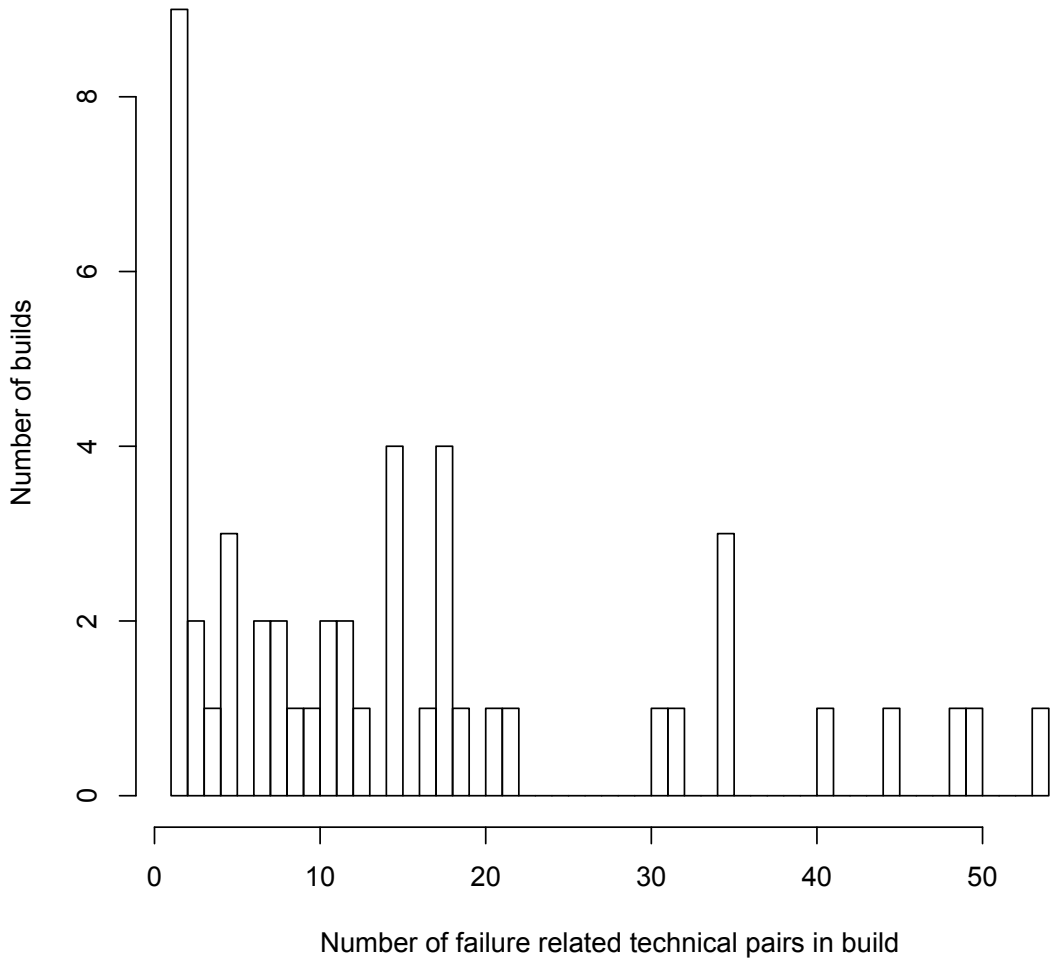


Figure 8.1: Histogram of how many builds have a certain number of failure-related technical pairs.

Although the analysis of technical pairs in relation to build failures showed that they have a negative influence on the build result, it does not take into account possible confounding variables. The question is whether there are developer pairs still affect the build outcome in the presence of confounding variables, such as number of developers involved in a build, number of work items related to a build, number of change sets per build, and number of files changed. We tested this using a logistical regression model including both developer pairs and confounding variables.

Overall, the logistic regression confirms the effect of the developer pairs even in the presence of confounding variables, such as the number of files changed, and numbers of

developers (AIC of 706). Table 8.4 shows an excerpt from the complete logistical regression that demonstrates that the twenty pairs previously identified are still significant under the influence of the four confounding variables. Because we have 2872 developer pairs, space constraints prevent us from reporting the entire regression model. We chose to report on the coefficients for the twenty pairs that we reported in Table 8.4, as well as the coefficients for the four control features.

All features shown in Table 8.4 are significant at the $\alpha = .001$ level (indicated by the p-value and ***). We checked for all the other technical pairs that were reported significant using the approach described in the previous section and found that they are all significant in the regression model as well. Moreover, the four control variables of the number of developers per build, number of work items per build, number of change sets per build, and number of files changed per build are also significant.

Since our model failed builds with 0 and successful builds with 1, a negative coefficient means that the feature increases the chances of a failure. All pairs reported in Table 8.4, and the technical pairs that we identified to be related to build failure, have a negative coefficient.

8.5 Summary

We end this chapter by returning to the initial research question we set out to answer:

RQ 2.1: Can Socio-Technical-Networks be manipulated to increase build success?

The results we presented in Section 8.3 show that there are developer pairs with unmet coordination needs that negatively influence build success. Furthermore, we demonstrated that the corresponding developer pair that fulfills its coordination need is less likely to be associated with build failure and thus theoretically decreasing the likelihood of build failure. To put it in perspective, if any of the twenty most harmful patterns is present in a build, the build has at least a 74% chance to fail.

Besides the technical side of the approach to generate statistically relevant recommendations as Murphy and Murphy-Hill [79] pointed out, developers need to accept such recommendations for them to be ultimately useful.

Chapter 9

Acceptability of Recommendations

Knowing that our approach can generate statistically significant recommendations is only one necessary step towards showing its usefulness. As Murphy and Murphy-Hill [79] pointed out, recommendations are of little use if developers reject them [100]. Thus, in this chapter we pursue the following research question:

RQ 2.2: Do developers accept recommendations based on software changes to increase build success?

We would like developers to communicate as soon as we suspect the likelihood for a build to succeed diminishes. But currently our investigation only takes a very technical stance and ignores many other criteria that might be of relevance to developers. For instance, it is unlikely that every build is of equal importance. Furthermore, we also think that developers' opinions of each other play a role, such as trust and respect. Therefore, it is important to explore those less technical aspects of developer relationships in order to gain more insight on when to deliver a recommendation.

In this chapter, we start by detailing the study design that is relevant to exploring our research question (cf. Section 9.1). Then, we present the findings we obtained in Section 9.2. We conclude this chapter by offering an answer to our research question and introducing the subsequent Chapter 10 (cf. Section 9.3).

9.1 Study Design

We complement our previous analysis of the Rational Team Concert (RTC) development team repositories with rich qualitative data from interviews and observations, considering

that much information on a project's lifecycle is not recorded in its repositories [2]. We focused on studying what factors cause a developer to seek information about a change-set, since change-sets are the smallest units of change to a project that directly impact other developers.

Our research question calls for a mixed methods empirical study. Therefore, we collect data about the team's information-seeking behaviour from three different sources. First, the author of this thesis was embedded in a sub-team of the RTC team as a participant-observer for four months. Second, we deployed a survey with the entire development team to validate some of the findings from the observations. Third, we conducted interviews with the developers from one component team to obtain richer information about their communication behaviour.

9.1.1 Data Collection

We used a mixed methods approach to answer our research question. We obtained data from three sources: participant observation, a survey, and a set of interviews with RTC team members.

Participant-Observation

We joined one of the RTC development sites in Europe for a four-month period in the Fall of 2010. We were involved as an intern with this development team helping with minor bug fixes, feature development, and testing, and thus were in a position to directly participate in the project development and communication activities. Our data collection opportunities were thus much richer than in typical observations conducted over shorter periods of time, or that do not involve active participation in the project. During our period as participant-observers, we kept daily activity logs, recording any information of relevance to the communication behaviour of our team.

The observation period coincided with the months prior to a major release, during which the team focused on extensive testing rather than new feature development. As such, the majority of our observations are concerned with activities around testing the integration of RTC with other IBM Rational products and it offered the experience of collaborating with many developers from different teams and across the remote sites. Finally, in the one month following the release, we also participated in the development of a feature for the upcoming version of the product.

The team in which we were embedded had the responsibility to develop the task management and the planning components of RTC. There were ten developers at the site, including two novices that had recently joined the team and eight experienced developers. The team also includes RTC's three senior project management members, who play a major role in planning the entire product's architecture and development.

Interviews

We conducted interviews with ten developers of the RTC team to inform our observational findings. The setting for the interviews was slightly unusual and beneficial from a research perspective. This is because our participant-observations allowed us to develop working relationships during our stay at the team, and to delve into a discussion of complex communication issues without needing to spend time understanding the basic context of the team.

In the interviews, we requested developers to provide details concerning their communication dynamics through a narration of "war stories" that directly related to communication among developers. "*War stories*" are events that left an impression on the respondent. These War stories emphasize specific events that they witnessed or took in part [70]; their narrative is a particularly powerful too in uncovering the elements of the stories that are relevant to the respondents. The interviews lasted between thirty minutes and one hour, and they were conducted at the end of the observation period.

Survey

To complement the insights we obtained from the observations and interviews, we deployed a survey with the entire RTC team at the end of our observation period. The survey was designed iteratively and piloted with a European team, and intended to collect input about which factors increase the likelihood of a developer requesting information about a change-set from the change-set author. The items we included in the survey were in the categories of code-related, developer-related and process-related items. Each category included 14 items, as shown in Table 9.3. *Code-change related* items describe the change-set delivered or the code that the change-set modifies or affects. *Developer-related* items relate to the developers that delivered a change-set, the developer requesting the information, and their relation to each other. Finally, *process-related* items relate to the development phase in effect when the change-set was delivered, and to items relating to process requirements or practices. For each of the three categories, the survey asked the developers to rank each

Table 9.1: Process-related items and quotes

Survey Items	Interview Quotes (I)/Participant Observation Quotes (O)	Mean Rank
Release endgame	(O) *"Adrian, in the endgame we only do minimal changes. Is your change minimal?"*	3.79
To review the change	(I) *"I often lack sufficient understanding of the part of the code I'm reviewing."*	5.00
To approve the change	—	5.11
Late milestone	—	5.14
During iteration endgame	(O) *"Adrian, in the endgame we only do minimal changes. Is your change minimal?"*	5.42
Obtain a review for the change	(O) *"It is demotivating to get a reject, thus I talk to my reviewer beforehand."*	5.55
Obtain an approval for the change	(O) *"I already fixed it, but I need to convince my team lead to give me approval."*	5.72
Related work item has high severity	—	6.00
Verify a fix	(I) *"Often I need to ask how I can tell that a change-set actually fixed the bug."*	6.37
Topic of work item the change is attached to	—	7.64
Priority set by your team lead	—	8.55
Role of the committer (e.g. developer)	—	9.00
Early in an iteration	(I) *"... there are weeks I sometimes don't talk to my colleagues at all."'*	11.24
Early milestone	—	12.10

Table 9.2: Developer-related items and quotes

Survey Items	Interview Quotes (I)/Participant Observation Quotes (O)	Mean Rank
Author is inexperienced	(O) *"Adrian, please put me always as a reviewer on you change-sets."*	2.60
Author recently delivered sub-standard work	(I) *"She just changed teams and still needs to get used to this component."*	3.04
Author is not up to date with recent events	(I) *"After you return from vacation we ensure you follow new decisions."*	3.15
You do not know the change-set author	(I) *"I'd be very irritated if someone other than [...] would touch my code."*	5.09
Author is currently working with you	(O) *"We sometimes discuss changes we made to brag about a cool hack."*	5.56
Author part of same feature team	—	6.00
Author part of your team	—	7.00
Worked with author before	—	7.77
Busyness of yourself	(I) *"If I see a problematic change-set, I'll ask for clarifications even if I'm busy."*	8.05
Busyness of the author	(I) *"If I need to know why an author made that change I just contact him."*	8.55
Met in person	(I) *"I've worked with him for 5 years now but never even met him."*	9.57
Physical location	(O) *"Here, America or Asia, I don't care, I ping them when they are online."*	10.14
Author is experienced	—	11.00
Author recently delivered high-quality work	—	11.09

Table 9.3: Code-change-related items and quotes

Survey Items	Interview Quotes (I)/Participant Observation Quotes (O)	Mean Rank
Changed API	(O) *"Adrian, why did you change that API?"* [The RTC team avoids changing API.]	2.84
Don't know why code was changed	(O) *"I don't see a reason why you changed that code, you sure you needed to?"*	3.00
Affects frequently used features	(I) *"Before I ok a fix [even to a main feature], I ask if there is a workaround."*	4.10
Complex code	—	4.89
Introduced new functionality	—	6.14
Is used by many other methods	(O) *"Why did you fix this part? The bug is not in this part of the code, it is used and tested a lot."*	6.41
Your code was changed	(I) *"I'd be very irritated if someone other than [...] would touch my code."*	6.42
Stable code was changed	(O) *"Why did you fix this part? The bug is not in this part of the code, it is used and tested a lot."*	6.62
Change unlocks previously unused code	—	7.42
A bug fix	—	8.61
A re-factoring	(O) *"I separate a re-factoring from a fix so that people can ask me questions about the fix."*	9.15
Frequently modified code was changed	(I) *"I like to know where the 'construction sites' of the project are."*	10.35
Code is used by few other methods	—	10.96
Simple code was changed	—	12.57

item in the category according to how strongly it increases their likelihood of requesting information about a change-set.

We initially formulated our list of items from an analysis of the current literature on coordination and communication, as well as from our own previous research. We then refined the list of items through piloting and discussions with the development team. An initial version of the survey included fifty-nine items; the refined list had forty-two. Besides reducing the number of items on our list, the pilot survey led us to add several process-related items, as well as to change the format of the questions (the initial survey had a battery of questions with Likert-scale answers; the final survey asked respondents to rank each item in comparison to those in its own category).

We deployed the survey through a web form, and invited the entire distributed RTC development team to participate. We sent a reminder one month later to increase the response rate. We obtained 36 responses to the survey; approximately a 25% response rate. Of the 36 respondents, 26 are located in North America, 5 are located in Europe, and 5 are located in Asia. Furthermore, 22 of the respondents are developers, 5 team leads, 5 component leads and project management committee, and 4 respondents withheld their information.

9.1.2 Analysis

For our data analysis, we transcribed the interviews, and then coded both the interviews and the participant-observation notes. From the codes we created categories, which we cross-referenced to the survey rankings. We used an open coding approach, during which we assigned codes to summarize incidents in sentences and paragraphs of the transcribed interviews that related to our research question.

We calculated the ranking of the survey items for each category by calculating the mean over all ranks given by developers to a survey item (cf. Table 9.3). Table 9.4 shows the distribution of the rankings for each survey item.

During our analysis, we derived a set of factors affecting communication behaviour based on our list of survey items and qualitative data from our interviews and observations. Whenever possible, we triangulated our findings using all our data sources. We tried to ensure that all of our findings were supported by at least two data sources; none of the findings we reported were challenged by any of our data sources.

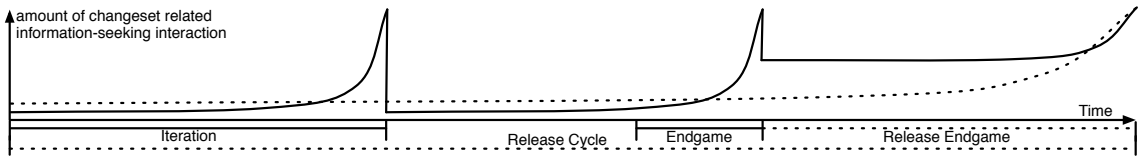


Figure 9.1: The pattern of information-seeking interactions throughout several iterations of a release cycle. Every release cycle consists of a number of iterations; each iteration includes an endgame phase. Change-set-based interactions are more frequent during endgame phases and during the last iteration of the release cycle.

9.2 Findings

The evidence collected from our survey, interviews, and participant observation allowed us to derive seven factors that affect communication behaviour in software organizations, which we outline below. In the remainder of this section, text in italics refers to survey items that can be found in Table 9.3. Text in italics and around quotation marks denotes interview fragments.

9.2.1 Development Mode

One of the strongest findings from our study was the effect of the development mode that the team has in on its communication behaviour. Developers and managers alike give a great deal of importance to the development mode they are in, specifically (1) normal iteration, and (2) endgame.

The normal iteration mode consists mainly of work that can be planned by the developer. This work tends to be new feature development, or modifications to existing features. Most of the planning for it has been laid out in advance; furthermore, each developer individually knows what features have been assigned to them, and can plan ahead to meet their obligations. In contrast, the endgame mode mainly consists of work that is arriving uncontrollably in short intervals from others. As a result of integration and more intense testing, defects crop up more rapidly and need to be addressed more promptly. Beyond allocating time for endgame activities, there is very little in this development mode that can be planned in advance.

The RTC team switches between the two modes in each iteration. Of the six weeks of a regular iteration, four weeks are assigned to normal iteration mode and two weeks to endgame. However, as deadlines approach, there are occasional special iterations that consist mostly of endgame-like work. In this manner, the same detailed pattern of alternation between normal iteration and endgame within an iteration is reproduced at a higher level, as shown in Figure 9.1.

Table 9.4: This table contains the distribution of ranks for each survey item. The leftmost point of each sparkline represents the amount of respondents that ranked the item first; the rightmost point represents the amount that ranked it last (14th).

Process-related items		Release endgame
		To review the change
		To approve the change
		Late milestone
		During iteration endgame
		Obtain a review for the change
		Obtain an approval for the change
		Related work item has high severity
		Verify a fix
		Topic of work item the change is attached to
		Priority set by your team lead
		Role of the committer (e.g. developer)
		Early in an iteration
		Early milestone
Developer-related items		Author is inexperienced
		Author recently delivered sub-standard work
		Author is not up to date with recent events
		You do not know the change-set author
		Author is currently working with you
		Author part of same feature team
		Author part of your team
		Worked with author before
		Busyness of yourself
		Busyness of the author
		Met in person
		Physical location
		Author is experienced
		Author recently delivered high-quality work
Code-change related items		Changed API
		Don't know why code was changed
		Affects frequently used features
		Complex code
		Introduced new functionality
		Is used by many other methods
		Your code was changed
		Stable code was changed
		Change unlocks previously unused code
		A bug fix
		A re-factoring
		Frequently modified code was changed
		Code is used by few other methods
		Simple code was changed

We identified the same pattern with respect to the amount of change-set-related interactions in the team. We found that the mode in which the team is currently in is an important factor, determining whether developers will feel a need to communicate about change-sets. Being in endgame mode increases the need for developers to communicate about a change-set, whereas being in normal mode decreases the need to request information about it.

We experienced both modes during our participant-observer time with the RTC team. We joined the team in a release endgame phase, which was characterized by fixing bugs that were reported by testers, and we helped by fixing minor bugs as well as setting up servers for testing. When the team released the project, it switched gears, and in the decompression after the endgame, we had the opportunity to develop a feature for the product. The differences in the patterns of interaction between the two modes were distinctly clear, for us and for our peers. During the endgame mode, developers were essentially "on demand," available to fix whatever bugs were discovered. Later, during the normal iteration mode, they experienced far more autonomy and control over their time, and began working on activities that were less demanding of an interaction back-and-forth, and especially less demanding of keeping track of the change-sets committed by their peers. In our interviews, developers almost unanimously pointed to the separation of the two types of modes, describing the differences in the type of communication in each, and identifying the autonomous *vs.* uncontrollable, inside *vs.* outside influence.

This is not to say that developers do not communicate while they are in normal iteration mode, but that the nature of their communication seems to be different. In normal iteration mode, developers communicate less about concrete change-sets, but they communicate more about high level ideas. For instance, they raise questions about how a feature fits into the existing architecture or general tactics to implement a feature, about how much of it actually needs to be implemented or can be reused from other libraries, and so forth. Generally, however, the amount of communication decreases in normal iteration mode. As a developer commented in one of our interviews, *"during feature development iterations there are weeks I sometimes don't talk to my colleagues at all."*

In contrast, the endgame mode is characterized by bug reports and last minute feature requests—that is, by work coming into every developer's desk uncontrollably. Change-sets become first class concerns during this mode because each change threatens the stability of the product: developers need to evaluate whether the change-set actually improves the product instead of making things worse. Therefore, they develop a habit of reviewing each other's change-sets to assess the risk they pose to the project's stability.

Our survey data provides significant support for this finding. In Table 9.1, we can see

that overall, for the RTC development team at large, there is a greater likelihood to request information during product-stabilizing phases, such as *during the release endgame*, and a lesser likelihood to request information when working towards an *early milestone*. The former was ranked as the most important of the process-related items in our survey; the latter was ranked as the least important. All the survey items that refer to a late stage in the iteration, or in the project lifecycle, were ranked highly and all the items that refer to an early stage were ranked lowly. This finding resonates especially with the team we interviewed.

9.2.2 Perceived Knowledge of the change-set Author

One key determinant for a developer to seek information about a change-set consists of what the developer knows about the background of its author. Several aspects of the author's background seem to be important: the quality of her recent work, her level of experience, and her awareness of recent team events or decisions.

First, these factors are important in part because they point to potential problems with the stability of the product. If a change-set author has *recently delivered sub-standard work*, other developers will want to ensure that the new change-set will not deteriorate their product, and therefore they will try to find more information about it.

Similarly, when the author is *inexperienced*, or is not *up to date* with recent team events for any reason, the risk of introducing problems into the product increases. One developer that interacts frequently with a newly founded team told us *"most of the work that I need to review is of very low quality and needs several iterations before it is up to our standards."* Novices, generally speaking, face a ramp-up problem that takes a significant amount of time to overcome [5]. This was the case with our work as well: our first change-sets were subjected to more scrutiny due to our unfamiliarity with the code base.

The survey data corroborates these observations. The three top items in Table 9.2 consist of factors related to an unfavourable perception of the background of the change-set author. In comparison, the two least important items in the table refer to favourable perceptions of said background.

In essence, these items point to the relevance of trust in software development teams. For developers, trust in the skills of their colleagues is important. If this trust is not established (for instance, if a developer does not know the change-set author), developers are more likely to check the change-set and request information to ensure that the modifications were truly necessary and appropriate. But when trust exists (that is, when the author

has a good record of delivering high-quality work, or is known to be an experienced team member), the need to seek information about a change-set decreases.

9.2.3 Common Experience and Location

Shared work experiences affect the likelihood of information-seeking behaviour in the RTC distributed team. This is partly related to the trust and perception issues discussed above, but also to the establishment of interpersonal relations and to the intertwining work responsibilities and expectations.

According to our interviews, this is the case particularly for senior team members. Although they work in a globally distributed team, they have managed to establish personal relationships with developers at many different sites. Through continuous interaction and social events, they have learned more about each other, and thus feel more comfortable initiating contact. In contrast, junior team members, in contrast, know very few of their teammates globally, and since they simply do not have these interpersonal connections with the rest of their team, it is less important to them whether they know the author of a change-set when needing to contact them.

Generally speaking, despite the lack of opportunities to build personal relationships with off-site people, for most developers sharing common experiences makes it easier to contact teammates to request information. One team lead told us: *"I have one or two contact people in each team we usually work with whom I ask for information and then often enough get referred to the right person in their team."*

Our participant-observation data confirm this. At one point, we needed to set up and test a specific component in the RTC product. The development team in charge of that component was located on a different site. However, since we previously had the opportunity to establish relationships with some of the developers of that team through a previous research study, it was easier for us to contact them and ask for help in our setup task.

Our survey data corroborate these observations. It suggests that the developers that form relationships (on a personal or work level) with other developers gave more emphasis on previous experiences with them. For instance, it made a difference whether the developers *are currently working together* (cf. Table 9.2). Additionally, whether they have shared work experiences seems to be a more determinant factor than having *met in person* or sharing the same *physical location*.

This last finding would seem to indicate that the RTC team has managed to overcome obstacles created by geographical distance. Developers state that the physical location of

a change-set author is not an important factor to seek more information about the change-set. Nevertheless, we should note two things. First, this finding merely points out that developers have no greater need to inform themselves of work output of their remote peers than of their local peers; it says nothing about the ease with which such information is acquired. Second, the RTC team has determined to carry out as many of its interactions as possible through textual electronic media. This neglects the natural advantages provided by proximity in favour of uniform accessibility of interactions, no matter the physical location of the interlocutors.

9.2.4 Risk Assessment

A central factor at the heart of the decision to seek information is a concern with the quality of the product, and components developed by the team at large. Both managers and developers share this concern. Because of it, every team member is constantly evaluating whether there are significant risks involved in accepting a change-set and including it into the final product. This concern is greatest close to releases or major milestones, and less important when the team is in normal iteration mode.

Developers request information more frequently about change-sets that touch on code that has a high customer impact. According to our interviews and observations, code parts with a high customer impact are those that are directly related to frequently used features or changes to the API that might be used by customers to customize the RTC product. In the case of the impact of API changes, there is an extensive knowledge exchange in the jazz.net forums between customers and developers about how to use the API to build extensions to RTC, which gives developers plenty of information about the possible impact of API changes to the customers. Consequently, change-sets that modify code that has less impact on the customer are of less concern and less likely to be discussed. For example, when scanning fixes of reported bugs, developers perform a risk assessment to determine whether the bug fix is even needed. In the words of one team member, *"with every tenth bug fix you introduce another bug to the system, so unless the bug does not have a workaround and is something most customers would experience, we give it low priority."*

The ranking of items in Table 9.3 provides additional insights into the types of change-sets that carry a risk for developers. A *changed API* comes at the top of the list, code that *affects frequently used features* comes third, immediately followed by changes to *complex code*, changes that *introduced new functionality*, and changes to code that *is used by many other methods*.

9.2.5 Work Allocation and Peer Reviews

In the later stages of the development cycle, developers have two process-based constraints that prompt them to interact with their peers. Firstly, they need their peers to review their change-sets before they are included in the product. Secondly, they need an approval to start working on a task (a triaging process needs to take place so that developers are allocated to the most important work items).

In the change-set review process, there are two ways the change-set author and the reviewers interact. Either the change-set author submits his change-set for review, or the change-set author discusses the change-set with the reviewer before submitting it for review. The main discriminant between meetings with the reviewer to discuss the change-set before submitting it, or submitting without prior discussion, seems to be whether the reviewer and the author are co-located.

Our mentor during our time with the RTC team explained: *"It is very demotivating to get a change-set rejected, that's why we usually try to discuss things before and then the reviewing just becomes all about quickly testing and accepting the change-set."* Of course, this does not prevent an already discussed change-set from being rejected, but it signals that the process-based constraint is not as strict as to forbid discussions about a change-set between authors and reviewers in the interest of an unbiased code review.

In the case of developers seeking approval to start working on a task, the process is similarly flexible. The approval is supposed to be issued before the developer begins work on the task. However, we observed several times that developers performed quick fixes, and later went to their team leads to acquire approval for the fixes they had already performed.

Several items in our survey point to the importance of information-seeking around the processes of approval and peer review. In Table 9.1, the second- and third-highest items were information needed, because the respondent needed *to review the change* or *to approve the change*. The need to *obtain a review for the change* or to *obtain an approval for the change* was of moderate importance. *Verifying a fix* was not judged as important by our respondents.

9.2.6 Type of Change

The three main reasons for a developer to commit a change-set are to (1) do feature development, (2) fix bugs, and (3) to re-factor. Among them, feature development seems to be the one that prompts developers to seek information the most. In our survey (cf. Table 9.1), code that *introduced new functionality* is ranked fifth among the code-change related

factors, while changes that consist of *a bug fix* and *a refactoring* ranked far lower on the same list.

This may be due to the fact that feature development has less tool support than bug fixing (which is supported by debugging tools and automated test frameworks) and refactoring (which is essentially an automatic process today). The limited tool support and the inherently difficult nature of the task itself makes change-sets that introduce new functionality more difficult to understand or assess. Hence, developers are more likely to request information about them.

9.2.7 Business Goals vs Developer Pride

Senior team members (managers and team leads among them) often ask about the purpose and the real need of a change-set in order to minimize unpleasant surprises that might be introduced with further code changes, and simply to have an economically feasible product. One senior development lead told us: *"often I need to stop my developers from fixing every little bug otherwise we would never be able to ship."*

Although every team member is concerned with the quality of their product, a more realistic, business-oriented perspective seems to be more prominent in the senior team members' information-seeking behaviour: they may request clarifications on the business case of change-sets or work items if they are not immediately clear in the existing documentation.

More junior team members, however, do not have this concern. They express their pride in the actual product and want it to be as bug free as possible. A developer commented to us: *"I want to be proud of what I deliver and I am fairly certain if we don't fix it now it will come back to haunt us because it might upset some customers."* Not only does this lead the developer to try to convince the manager that including change-sets for seemingly unimportant fixes is valuable, it also makes the developer's information-seeking behaviour different with respect to that of the more senior team members. Broadly speaking, managers will be more interested in seeking clarifications as to the business case of a work item if it is not clear in its description, while developers will not care as much about this.

9.3 Summary

We end this chapter by returning to the initial research question that we set out to answer:

RQ 2.2: Do developers accept recommendations based on software changes to increase build success?

The findings that we presented in Section 9.2 demonstrate that there are different factors that influence developers to inquire about change-sets. Although not directly asked for, the developer who reported on those factors answered our research question in a positive way. Our approach would provide developers with recommendations either on a per change-set level or while they are working on the code. We found that developers do talk about change-sets that represent our level of recommendations as well as show interest in build outcome towards the end of a release cycle. Thus we can say that the level of our recommendation appears to be appropriate when keeping it to the change-set level. Examining our findings about risk assessment, which is practiced by every lead and developer, we are confident that developers would welcome notifications on how risky, with respect to build success, the changes they are currently applying are. This gives them a chance to either talk to the recommended person or reconsider the change altogether. Hence, we are confident the developer answers both validated that they would accept recommendations both in the form we present to them, namely per change-set submitted, as well as lending our outcome metric more credibility as they show a keen interest in build success when the date to ship the product approaches.

In the next chapter, we return to our path of a more direct analysis of our recommendations by showing a proof of concept that the recommendations do not only statistically relate to build outcome but that recommendation could actually prevent build failures.

Chapter 10

Appropriateness of Technical Relations

In this chapter, we present the final piece of evidence in the line of exploring the usefulness of our approach presented in Chapter 7. With a case study, we demonstrate that we can generate recommendations to prevent build failures when it actually matters. Therefore, this chapter explores the following research question:

RQ 2.3: Can recommendation actually prevent build failures?

In essence, generating recommendations after the fact, even if the form and content in generally accepted by developers, is of little use. This is because it is too late to act upon it. In this chapter, we explore not only whether the recommendation, if given at the right moment, could have prevented a build from failing, but also if we could have generated that very recommendation at an appropriate time for preventative actions to take place.

In the remainder of this chapter, we describe the course setting in Section 10.1. Following this, we describe our methodology in Section 10.2. Then, we present the analysis and results we obtained in Section 10.2.3 and Section 10.3, respectively, followed by a discussion of the results in Sections 10.4. We conclude this chapter by offering an answer to our research question (cf. Section 10.5).

10.1 A Course on Globally Distributed Software Development

We conduct a case study of the collaboration of students in a project taken in a class in globally distributed software development taught at the University of Victoria, Canada and

Aalto University, Finland, as the basis for exploring our final research question for several reasons:

- Control over a project infrastructure that supports data collection.
- The distribution of the course across countries ensures a minimum of electronically recordable and mineable communication data being generated.
- A course setting allows for more control to ensure data quality.
- A course setting enables better access to study participants.

One key to exploring our research question is to collect data to analyze. In a course project we are able to choose and instrument the infrastructure used by the students to actively develop software, and thus possibly gain more insight, or at least miss less data, than in most industrial settings that rarely focus on collecting data for analysis. This especially allows us to establish traceability links between artifacts that are often missing and need to be inferred using semi-accurate heuristics.

Furthermore, the distributed nature of the project makes synchronous non-text based communication more difficult, and thus increasing the use of text based asynchronous, and therefore recorded, communication. This additional need to rely on flexible and stable communication media, such as email and text based chats, enables us to collect more data that is easier to mine than it would have been realistic in a collocated setting. The distributed nature of the teams, also induced an openness to work in a more distributed fashion even among collocated team members due to a sensibilization of trying to enable their remote team members to follow their discussions.

Due to the nature of the study being conducted in a course setting, we had more control about how participants are using technology, and could give guidance on how to improve their collaboration through means that are easy to record. Because participants need to attend lectures, we can educate them in a way that emphasizes the importance of recording traceability links.

Since the participants are students taking a class on software development, we can more easily, ensure the quality of data that we are collecting. Through a feedback loop we can change the data collection to accommodate the needs of the students more easily and therefore ensure data quality by mitigating the risk of participants neglecting the data collection process.

Table 10.1: Descriptive statistics about the student development effort

	Team A	Team B	Team C	Total
#failed builds	15	13	19	47
#successful builds	41	14	1	56
#builds	56	27	20	103
#Canadians	4	4	4	12
#Fins	2	2	2	6
#Change Sets	440	491	452	1383

10.1.1 Course Details

The globally distributed software development course taught at both the University of Victoria, Canada and Aalto University, Finland, aims at teaching students how to develop software in a distributed setting. In this particular course student teams where forced to deal with the issues that come with a distributed team in an extreme way since both West Coast Canada and Finland are separated by a time difference of eleven hours minimizing the time all team members are awake. This both necessitates and motivates the students to apply lessons learned in the classroom.

The course on globally distributed software development took place starting in October 2011 and ended in April 2012. With the Finnish students starting in October 2011 and the Canadian students joining the development effort in January 2012. This further allowed the students to engage in remote collaboration as the Finish students represent the experts on the system the teams are working on and will need to guide their Canadian teammates in getting used to the projects layout and functions.

Table 10.1 shows some descriptive statistics about the development effort the students undertook. The 18 student developers together committed a total of 1383 change-sets while creating 103 builds. We can see that some teams subscribed to the frequent build mentality more than others. For example, Team B and C created builds substantially less often than Team A over the same time window.

Besides developing a software product, the Canadian students also had regular classes. During a regular class the students participate in a seminar style setting that requires each student to read a set of papers. These papers are discussed on the course blog with respect to implications of the findings detailed in the papers and how these findings may apply to their current development experience. Every two weeks the seminar style class is interrupted for a project wide meeting to bring together all the teams including the Finnish team members, to show case their progress and to plan the next steps for the following two weeks.

The students filled out a survey every other week to give feedback on the class, the development process, and the use of different tools for communication and development. Besides this bi-weekly surveys, the students also had the chance to talk to the course instructors, as well as the teaching assistant, to issue concerns and make suggestions on how to improve the course for the students, as well as bring up issues concerned with the quality of the collected or non-collected data.

10.1.2 Team Composition

As described earlier, each development team consists of both students from Canada and Finland. There are three teams, each of which consist of of four students from Canada and two students from Finland.

In addition to the three teams, we also have a product owner that came up with the idea of the product that we describe in greater detail in Subsection 10.1.3. The product owner is located in Finland, and is responsible for prioritizing the different features that he wants to be implemented. Moreover, he negotiates with the teams which items they should aim for within a given two week span.

To coordinate the work across teams, one Finish student has become the SCRUM master or de-facto project manager. He is both the most experienced developer, and has the most knowledge about the projects, as he is using the product within his own company. Some of his responsibilities entail facilitating coordination across teams, setting up meetings between teams, and structuring the biweekly project meetings. He also takes care of scoping out development tools to use and ensures that everyone is up to speed with issues arising from other teams that may implicate their work.

10.1.3 Development Project

The students in this course are extending the agile-planning tool Agilefant.[1] Agilefant combines task creation, task management, and planning into one tool that allows scheduling different tasks for different sprints, with a sprint being a time period that is meant to implement a fixed set of tasks. Furthermore, tasks can be organized hierarchically and be of different types, such as user stories or bug reports.

As mentioned previously, this product was conceived by the product owner and has since been developed by students participating in cap-stone projects at Aalto University,

[1] www,agilefant.org

Finland. The tool itself is open source and available for anyone to download, install, and extend as they see fit.

Besides being a student driven project it found a number of users in industry. Due to the development being mainly undertaken in the cap-stone projects that are offered yearly to the students, the project progresses in yearly bursts. Nevertheless, the stability of its core feature and lightweight approach made it popular among small and mid-sized companies for planning their development projects.

10.1.4 Development Process

The project follows a SCRUM process[2], and thus development progresses in small sprints working of a common backlog. A sprint, in the case of our project, is defined to be two-weeks long starting with a planning session and ending in a retrospective. The planning sessions and retrospectives take place between two sprints at the same day starting with the retrospective followed by the planning for the next sprint.

10.1.5 IT-Infrastructure

The source code is managed using GIT[3] and is available on github.com[4]. Although Agilefant or github.com issues can be used for task management, both are missing important features. For example, github.com is missing the planning capabilities of a full agile planning tool and Agilefant is missing the ability to discuss work items. Therefore, we use IBM Rational Team Concert for the planning of sprints and tracking of work items. An additional advantage of IBM Rational Team Concert is that it integrates with Eclipse the main development environment, which also integrates with GIT.

Besides using discussions on work items, the students also use email and IRC for communication. To ensure that everyone has the same knowledge students are encouraged to summarize their emails and IRC discussions in work item discussions. For group meetings, especially the planning and retrospective sessions, we provided the student with separate rooms and internet access to use Google hangout for video chatting and screen sharing when discussing their group individual sprint plans. For the project meeting, after each sprint to demo their accomplishment to the development team at large, we provided a conference room with multiple microphones and screens to simultaneously show a number of remote participants as well as screen sharing sessions.

[2] http://en.wikipedia.org/wiki/SCRUM_(development)
[3] http://git-scm.com/
[4] https://github.com

To further support the development process, the students set up a Jenkins instance, which is an open source web-based build system. This build system allows for continuous testing which helps the students to get a better understanding of their progress with respect to implementing their tasks for a given sprint.

10.2 Methodology

10.2.1 Proximity to Infer Real Time Technical Networks

In order to be able to explore the possibility of recommending improvements with respect to the socio-technical networks, we need to be able to intervene before the majority of work has been done. The main issue with creating real time socio-technical networks lies in constructing technical networks, as they are only known once a developer submits their changes to a repository. On the other hand, social-networks can be more readily constructed in real time because in order to communicate, at least electronically, yields a trace for each bit of communication traveling from one person to another allowing the extension of a social network with each new instance of communication.

To gain a similarly effective way of constructing technical networks we build upon the work of Blincoe et al. [9], who proposed a measure called proximity using Mylyn [58] traces to infer real time developer dependencies. Mylyn is an Eclipse plug-in that records edits and selects within source code files. These events show that artifacts were modified and selected, thus allowing the partial construction of a technical network based on ownership information of the modified, selected, and implicated source code artifacts.

In order to record this information in real time in a central repository we developed a plug-in that captures events recorded by Mylyn, connecting them to the current task a developer is working on, and submitting this information to a central server. To encourage students further to install this plug-in, we also provided a visualization of the current proximity of a developer to any other developer given her current task in a visualization plug-in called ProxiScentia [13].

10.2.2 Data Collection

For our study we collect six types of data:

- Communication data such as e-mails, IRC chat logs, and work item discussions.

- The source code repositories on github.com, including each developer's fork of the main repository.
- Log of the build engine indicating failed and successful builds.
- Documentation and diaries created by students during the course.
- Surveys and questionnaires completed by students throughout the course.
- Mylyn events gathered using our custom built Eclipse plug-in.

Although we are not necessarily interested in constructing social networks for this part of the study, but rather we are concerned whether it is possible to prevent build failures given a certain set of collectable data, we need access to as much communication data as possible not only as a means to see what kind of problems arose but also as to whether they were of any importance to the developers. For example, are build failures a problem for the developers and if so are all build failures equal. Knowing which build failures are actually problematic to the developers allows us to focus on those instances that matter.

To further the investigation into the issues reported by the developers, we make use of the version control systems. In our case, these are several GIT repositories hosted on github.com. Each developer owns their own forked repository from which they pushe changes to their respective team repository. Once the product owner has reviewed a feature implementation or bug fix, it is pushed to the project repository to ensure that each development team is working on the latest stable code.

Since implementing quality features was a main concern during the class, the students set up a build-engine to continuously test their implementations. Jenkins,[5] the build engine used by the teams, keeps a log of all the builds as well as their outcomes.

Besides collecting data that is generated by the development activities of the students during the course, we also asked them to keep a development diary as well as fill out frequent surveys that gauge their impression of the progress within the project as well as the adequacy of the tools used.

Finally, as mentioned in the previous Subsection 10.2.1, we also collect the code interactions with respect to which methods in the source code have been edited or selected in a central repository.

[5] `http://jenkins-ci.org/`

10.2.3 Analysis

We analyze data collected during the course in six steps, starting with identifying issues of interest to show how recommendations based on our approach could have avoided these build failures:

1. Identify build issues mentioned by the developers.
2. Identify the actual failed build in the Jenkins build logs.
3. Identify the code changes that contributed to the build failure.
4. Check if the recorded code interactions could have predicted the build failure.
5. Check if the relation between the code changes could have predicted the build failure.
6. Check if using the social-network helps to find the right people to include in order to resolve the issue.

Identifying build issues. To find build issues, that are truly build issues to the developer, and not only reported as failed due to failing test cases that were meant to fail, we code the communication recorded in the IRC channels, work item and email discussions, and the developer diaries with respect to problems experienced in the project. From those coded problems, we select all problems related to building the software. The students mentioned three builds in greater detail in their development diaries and chat sessions.

Identifying the actual failed build. After we use the recorded communication to identify which failed builds are true failures, we identify those builds as they are shown in the Jenkins logs. The Jenkins log showed that the students submitted a total of 103 build requests, of which 47 produced builds with failures. These builds show what went wrong, as well as what changes are new to those builds, that potentially broke it.

Identify code changes. For each actual failed build, we infer which change-sets were newly added to the code base between the previous build and the failed build. From a change-set we can determine the task worked on, as we require every student to annotate each submitted change-set with the work item identifier the change-set was meant to implement.

Check code interactions. After we identify the change-sets that were newly added to a build we can identify the developers that worked on the same source code files.

Check change relations. Knowing both the change-set, as well as the interaction traces that where involved in the failed builds, we can contrast and overlay those change-sets, as well as code interactions, to see if the path of different developers overlap in a way that allows us to predict the issues that were reported on by the students.

Check social network. In case we observe that the code traces that we collected could have predicted the issue, we look for cues in the communication media for the resolution of the issues to see whether the developers knew they shared a technical dependency expressed in those code traces. Finding a relation between the code interactions and the ongoing communication during the implementation contrasted with the communication needed to resolve the issue created by the change-set is the key to building a useful recommendation.

Instead of analyzing each build failure that is important to the developers, we focus on finding one build that let us trace the error back to the code interactions. Moreover, it shows a connection between the code interactions and the communication needed to resolve the build issue.

10.3 Findings

In this section we present the findings from our analysis.

10.3.1 Build Failures That Matter

Throughout the feedback and communication logs received from the students we observe that the most important problems to the students are those build failures that prevent other teams from continuing their work. These issues usually happened towards the end of the sprint, as this is the time in which teams are integrating their code into a common code base.

"Unexpected problems when merging the old code in, had to overload a couple methods [...] and create some new code to mesh it together. Eventually fixed with a decent solution." - Student comment

On the other hand, most failed builds are of less interest to the students for two reasons: (1) tests became outdated as the core functionality of the product changed considerably and (2) the product owner was deciding what features are of acceptable quality regardless of the test case results.

Since most of the work preformed by the students changed, or extended, core functionality, old test cases started to fail. Because of this the product owner is deciding on whether a feature is deployable based on a demo, and manual testing lead test cases to loase value with respect to deciding whether a feature is of acceptable quality.

10.3.2 Preventable Build Failures

Table 10.2 shows two traces that directly link to one of the major build failures caused by merge conflicts. Similarly, the two traces representing the change-sets point to the file responsible for the merge conflict.

We chose this particular failed build for three reasons: (1) this failure occurred early on in the development and was described in most detail by the effected students, (2) it involves members from different student teams further illustrating issues with cross-team communication, and (3) it is representative for both the way we characterize technical dependencies (two developer change the same file) and it is the failure type most students complained about in their development diaries.

Each trace contains a list of methods ordered by when the developer selected or edited a method. The two traces shown in Table 10.2 only show edit events, and removed redundancies, when multiple subsequent edit events in the same method occurred. The two traces overlap by both editing a common function, *ProjectBusinessImpl.java-retrieveLeafStories-QStoryFilters* (emphasis added in the Table), which turned out to be the offending function that causes the build to fail.

Note that the traces in Table 10.2 were recorded during the same sprint but shifted in time. Furthermore, the owners of the two traces, and thus change-sets, belong to different teams whose work is supposed to be integrated towards the end of the sprint.

10.3.3 The Right Recommendations

We saw that most of the build failures that are of high relevance to the developers deal with issues arising from merging code from several teams. Although when such a build failed, communication among team members spikes, it is confined to the team that merges their code last and thus experiences the build failure once they merged their code. Thus,

Table 10.2: Overlapping code interaction traces that lead to a merge conflict

Angelique's Method Trace for Task ...
HourEntryBusiness.java-calculateWeek-Sum-QLocalDate
HourEntryBusiness.java-getDailySpentEffortByInterval-QDateTime-QDateTime
HourEntryBusiness.java-getDailySpentEffortByIteration-QIteration
HourEntryBusiness.java-getDailySpentEffortByWeek-QLocalDate
HourEntryBusiness.java-getEntriesByUserAndDay-QLocalDate
HourEntryBusiness.java-logBacklogEffort-QHourEntry-QSet
HourEntryBusinessImpl.java-calculateWeekSum-QLocalDate
HourEntryBusinessImpl.java-getDailySpentEffortByInterval-QDateTime-QDateTime
HourEntryBusinessImpl.java-getDailySpentEffortByIteration-QIteration-QDateTimeZone
HourEntryBusinessImpl.java-getDailySpentEffortByWeek-QLocalDate
HourEntryBusinessImpl.java-getDailySpentEffortForHourEntries-QList-QDateTime-QDateTime
HourEntryBusinessImpl.java-getEntriesByUserAndDay-QLocalDate
IterationBurndownBusinessImpl.java-getBurndownTimeSeries-QList-QLocalDate-QLocalDate
IterationBurndownBusinessImpl.java-getCurrentDaySpentEffortSeries-QList-QDateTime
IterationBurndownBusinessImpl.java-getEffortSpentDataItemForDay-QDailySpentEffort
IterationBurndownBusinessImpl.java-getEffortSpentTimeSeries-QList-QDateTime-QDateTime
ProjectBusinessImpl.java-getProjectData
ProjectBusinessImpl.java-retrieveLeafStories-QStoryFilters
StoryHierarchyBusinessImpl.java-moveUnder-QStory-QStory
HourEntryDAOHibernate.java-getHourEntriesByFilter-QDateTime-QDateTime
DailySpentEffort.java-setDay-QDateTime
StoryHierarchyAction.java-moveMultipleAfter
StoryHierarchyAction.java-moveMultipleBefore
StoryHierarchyAction.java-moveMultipleUnder
StoryHierarchyAction.java-moveStoryUnder
StoryHierarchyAction.java-recurseHierarchyAsList
Bernard's Method Trace for Task ...
BacklogHistoryEntryBusinessImpl.java-setStoryHi-erarchyDAO-QStoryHierarchyDAO
BacklogHistoryEntryBusinessImpl.java-updateHistory
ProductBusinessImpl.java-store-QProduct
ProjectBusinessImpl.java-persistProject-QProject
ProjectBusinessImpl.java-store-QInteger-QProject-QSet
ProjectBusinessImpl.java-retrieveLeafStories-QStoryFilters
StoryBusinessImpl.java-createStoryRanks-QStory-QBacklog
StoryBusinessImpl.java-store-QInteger-QStory-QInteger-QSet
StoryWidget.java-getStory
UserLoadWidget.java-getUser

the communication that is ongoing in order to resolve the build issues does not necessarily include the people that could have helped to prevent the build from failing.

Nevertheless, the recordings and communication after the merge conflict was resolved allows for a better picture of the people involved. After the issue was resolved, the team fixing the issue voiced the concern in the following retrospective and sprint planning as well as contacted the respective team. From the communication it became apparent that the issue could have been prevented if the authors of the conflicting change-set would have known about each other's change. Using our approach, and given the data we collected from the developer traces, we could have provided a recommendation that would have prevented the build failure by making the developers aware of their overlapping work.

10.4 Discussion

In this chapter, we set out to answer the question of whether it is possible to make recommendations to prevent build failures before they happen. We found that the methods proposed by Blincoe et al. [9] gathering real time code interactions using Mylyn to make the post-mortem analysis performed in the traditional socio-technical congruence context [58] actionable is useful for providing real time recommendations. Although we did not show the usefulness of using the Blincoe et al. [9] approach by implementing a recommender system leveraging the data, nonetheless, the data is in place and shows promise to resolve certain issues before they become problematic. The course experiment showed that there is a link between the build failures that stems from technical dependencies that can be inferred while developers are working to implement their work items.

10.5 Summary

We brought this chapter to a close by returning to the initial research question that we had set out to answer:

RQ 2.3: Can recommendation actually prevent build failures?

Based on data that can be gathered in a team setting, we showed that recommendations preventing build failure can be generated in a classroom setting. The student in the globally distributed software engineering course experienced the very build failures that the approach we presented in Chapter 7. Through interviews and analyzing the students development diaries we could identify a number of build failures that matters to the students. For a representative build, we traced the cause of the build failure through the coding interactions we recorded from the students, and were able to find that the cause for the build

failure stemmed from modifying the same file. This lends further evidence that the technical dependencies we used in Chapter 8 to generate actionable insights. We did not provide the actual recommendations but instead traced the build issue in one representative case back to coding errors that could have been uncovered if the respective developers would have talked about their changes.

Chapter 11

Conclusions

In this chapter, we summarize and discuss our approach and look at how the research we presented in the last three chapters support it (cf. Section 11.1). We will then delve in to our second contribution that takes the form of the individual insights gained by the five studies we presented (cf. Section 11.2). Before ending this dissertation with concluding remarks as well as some future work (cf. Section 11.4), we discuss the threats to validity (cf. Section 11.3).

11.1 An Approach For Improving Social Interactions

We derived the approach presented in Chapter 7 through two case studies that investigate the usefulness of social and socio-technical networks to predict build outcome (cf. Chapters 5 and 6). We conducted a study to see if the approach can generate relevant recommendations in Chapter 5. The studies we conducted in the subsequent Chapters 9 and 10 further explore the usefulness of the information with respect to whether experts expect the level of recommendations to be of use, as well as whether these recommendations could be produced in real time and potentially prevent issues from arising. The approach we presented in Chapter 7 consists of five steps:

1. Define scope of interest
2. Define outcome metric
3. Build social networks f
4. Build technical networks
5. Generate actionable insights

In Chapter 5, we showed that the communication structure of a software development team influences build success. This suggests that there is value in manipulating this structure to improve the likelihood for a successful build. That evidence is further supported by our finding that gaps in the social network constructed from developer communication as suggested by technical dependencies among developers also affect build success (cf. Chapter 6).

In these two studies, we already applied the first four steps of the approach presented in Chapter 7. We defined the build as the scope and the outcome metric as the build outcome. Using the scope we constructed social networks from the communication among developer that can be related to a build as described in Chapter 2 in both Chapter 5 and 6. Chapter 6 used dependencies among change-set committed by developers that are relevant to a given build to construct a technical network to complement the social network forming a socio-technical network.

The three studies we presented in Part III of this dissertation focused on the last step to *generate actionable insights*. The study in Chapter 8 showed that we are able to produce recommendations from available repository data that affect build success. These recommendations take the form of highlighting two developers that have a technical dependency but did not communicate in the context of the build.

We decided to focus on generating recommendations, enticing developers to communicate in order improve build success over recommendation, that would suggest code changes, changing the dependencies among developers for two reasons: (1) proper code changes are more difficult to suggest without a sufficient understanding of the program requiring more in-depth program analysis and (2) developers need to trust the recommendation, which is easier to achieve by limiting ourselves to suggesting people that are affected by a change.

In Chapter 9 we explored the developer's view with respect to recommendation systems and if and when recommendation on a change-set level would be appropriate. The feedback we received generally welcomed such recommendations as long as they were not seen as irrelevant, thus, corroborating Murphy and Murphy-Hill's [79] point. Developers, in fact, discuss change-sets in general, but specifically towards the end of a release cycle, as each change becomes more important with respect to the stability of the overall project.

Through an in-class study with several students in Canada and Finland we investigated whether we can collect the necessary data to compute recommendations at the appropriate time as well as if those recommendation actually could prevent builds from failing (cf. Chapter 10). We recognize that a student project is substantially smaller in size and com-

plexity than an industrial project such as Rational Team Concert, nevertheless, the students worked on an existing open source project that is used by several companies. Furthermore, the in class study gave us the chance to see the possible effect of our recommendations more clearly as the smaller complexity allowed builds to fail for few reasons and therefore making the impact of the recommendations clearer.

Overall, we gave evidence to the usefulness of our approach by selecting a specific scope and outcome metric, builds and build success respectively, and defined the construction of the social and technical networks in detail (cf. Chapter 2). Furthermore, we showed that the approach could generate actionable insights that are acceptable to developers. In a final study involving students from two countries, we found evidence that we are both able to generate recommendation early enough to be acted upon, as well as demonstrated that such recommendation could actually prevent build failures.

11.2 Contributions through Empirical Studies

Each study by itself contributed to the overall body of knowledge of software development team coordination. We present, in the order of our five research questions, the contribution of each study.

11.2.1 Using Build Success as Communication Quality Indicator

We started our investigation by exploring whether there exists a relationship between build success and communication by using prediction models (Chapter 5). Our models can be used by Jazz teams to assess the quality of their current communication in relation to the result of their upcoming integration. If a team is currently working on a component, and an integration build is planned in the near future, the measures of the current communication in the team can be provided as input to our prediction model and the model will predict whether the build will fail with a precision shown in Table 5.4. For example, if team P is working towards a build, and our model predicts that the structure of its current communication leads to a failed build, the team can have a 76% (see Chapter 5 Table 5.4) confidence that the build is going to fail. This information can be used by developers in monitoring their team communication behaviour, or by management in decisions with respect to adjusting collaborative tools or processes towards improving the integration.

11.2.2 Unmet Coordination Needs Matter

The relationship between communication structure and build failures however significant, has only a small effect on the overall success rate of software builds, the outcome metric we studied. This lead us to include information about the system by adding technical dependencies, as expressed by the source code among software developers. Backed up by findings in the research area of socio-technical congruence, we hypothesized that the technical relationships help to zero in on the important relationships among developers that relate to build failures. As the relationship between socio-technical congruence and productivity suggested influence on software quality, we showed in Chapter 6 that it actually predicts build failures with varying accuracy depending on the type of build. Thus, not meeting coordination needs, as demanded by technical dependencies among software developers, has a negative effect on build success.

11.2.3 Developers That Induce Build Failures

Being able to predict whether a build fails already helps developers to plan ahead with respect to future work, such as stabilizing the system in contrast to working on new features. However, ultimately we want to be able to prevent builds from failing. For that purpose, we need to influence the socio-technical network such that it takes a structure that is more favourable to build success. We found that certain constellations within a socio-technical network, to be more precise pairings of software developer and their respective relationship, seem to be correlating with build success (Chapter 8). This evidence can be used to recommend action before the build is commenced in the sense that developers can investigate their relationship. For example, they can do this by discussing the code changes that created a technical relationship between them.

Our findings have several implications for the design of collaborative systems. By automating the analyses presented here, we can incorporate the knowledge about developer pairs that tend to be failure related in a real-time recommender system. Not only do we provide the recommendations that matter to the upcoming build, we also provide incentives to motivate developers to talk about their technical dependencies. Such a recommender system can use project historical data to calculate the likelihood that an upcoming build fails given a particular developer pair that worked on that build without communicating to each other.

For management, such a recommender system can provide details about the individual developers in, and properties of, these potentially problematic developer pairs. Individual

developers may be an explanation for the behaviour of the pairs we found in Rational Team Concert. This may indicate which developers are harder to work with or too busy to coordinate appropriately, prompting management to reorganize teams and workloads. This would minimize the likelihood of a build to fail by removing the underlying cause of a pair to be failure related. Similarly, as another example in our study demonstrates, most developer pairs consisted of developers that were part of different teams. In such situations, management may decide to investigate reasons for coordination problems that include factors such as geographical or functional distance in the project.

11.2.4 Recommender System Design Guidelines

In our first qualitative study (cf. Chapter 9) we explored whether developers would accept recommendations produced by our approach. Our results showed that developers are generally open to recommendations on a low level, such as on a change-set basis, but it depends on external factors such as the development process. For instance, we found that depending on how close a development team is to a software release, the more they focus on the implications of individual changes, whereas developers focus more high level reusability issues at the beginning of a release cycle.

Nakakoji et al. [83] formulated nine design guidelines for systems that support seeking information in software teams. Some of these guidelines deal with minimizing the interruptions experienced by the developers who are asked for information, while others focus on enabling the information-seeker to contact the right people. Our findings help us refine Nakakoji et al. guidelines:

Guideline #1 *Recommender systems should adjust to the development mode.* Our first finding presented in Chapter 9 strongly suggests that a developer's information needs can dramatically change between development modes. When in normal iteration mode, developers act upon planned work and can therefore anticipate the information they need, but in endgame mode, developers react to unplanned incoming work, such as bug reports or requests for code reviews.

Many tools, such as Codebook [4] and Ensemble [113] provide information and recommendations in a fixed way. Codebook enables developers to discover other developers whose code is related. In contrast, Ensemble provides a constant stream of potentially relevant events for each developer. In the Codebook case, this might lead to extra overhead in endgame mode, when developers frequently need to search for information instead being

automatically provided, whereas Ensemble might overload developers during the feature development mode by providing a constant stream of information.

To avoid overwhelming or reducing overhead further for developers, recommendation systems should either automatically adjust to the development mode or feature customizable templates that can easily be switched.

Guideline #2 *Recommender systems should account for perceived knowledge of other developers.* Our second and third findings (cf. Chapter 9) unveiled factors that trigger developers to seek information about a change-set that are not related to its code. Instead, developers pay close attention to the experience level, as well as the quality of previously delivered work, to determine whether to speak to the change-set owner.

Traditional recommender systems in software engineering focus on the source code in order to determine useful recommendations (e.g. Codebook [4] and Ensemble [113]). This might lead to providing developers with information about changes that are of little interest due to the trust placed in more experienced developer.

Since developers often look beyond source code and perform an additional step, namely considering the change-set owner's experience and recent work, information solely created from source code might miss interesting instances where novices to the code made inappropriate changes. Recommender systems might report issues that are of less importance due to the substantial experience of the change-set owner.

Implementing filtering mechanisms based on author characteristics, such as experience and quality of previously delivered work, can help developers to focus on the information that is important to them. Since this information can be difficult to determine automatically, since perceptions are specific to the individual developer, a recommender system should allow users to manually input information.

Guideline #3 *Recommender systems should assist in non-implementation tasks such as code reviews and risk assessment.* We observed, as described in the fourth, fifth, and sixth findings (cf. Chapter 9), that developers are highly engaged in discussions when performing risk assessments or reviews of change-sets.

In software engineering, most recommendations are focused on providing information to support concrete tasks such as bug fixes or re-factorings, but not for tasks such as reviews. To provide information for non coding tasks, recommender systems should be configurable to display relevant information beyond the tasks that they are intended to support, so that

developer can easily access the information provided by recommender systems when performing code reviews or risk assessments.

Guideline #4 *Recommender systems should account for business goals.* Our last finding (cf. Chapter 9) points to internal conflicts within teams and among developers caused by the desire to create a flawless product under the restriction of a set of business goals such as shipping the product on time. Thus, developers often need to be reminded that their efforts need to be focused on fulfilling business goals rather than on polishing the product as they see fit. Existing recommenders that use code-related metrics such as quality or productivity may shift attention away from fulfilling business goals.

To support developers to focus on business goals, systems supporting the information-seeking behaviour of developers should be able to prioritize information related to tasks that are mission-critical to the organization, helping the team focus its attention on the most relevant problems for the upcoming release.

11.2.5 Socio-Technical Congruence in Real Time

Knowing that socio-technical congruence lends itself to producing actionable knowledge that has an acceptable form to support developers in the wild, we are lead to our last study (cf. Chapter 10). In this study we showed the feasibility of generating recommendations at the right time, by gathering data to generate socio-technical congruence in real time. Thus, we showed that socio-technical congruence could be used in real time to create actionable knowledge that might be of use to developers.

11.3 Threats to Validity

In this section, we detail the threats to validity of this dissertation.

External Validity In part of this work we draw on information from observational studies (cf. Chapters 9 and 10) and studies relying on development repositories (cf. Chapters 5, 6, and 8) that cover two development projects. Although this limits the generalizability of the findings presented, as well as the validity of the inferred approach, we believe that the approach still holds merit since the studies that lay the foundation for the validity of generating insights in real time are derived from an industrial project comprised of more than one hundred developers at a large software corporation. This in-depth relationship created

by working together with the IBM Rational Team Concert development team limits the amount of data available for the studies we presented. However, this in-depth relationship enables us to better interpret the collected data as well as gain a deeper understanding of the organization, their processes, and how they influence the data. In the case of the in class study, we aimed to minimize the conclusions we drew to only serve as a feasibility study to demonstrate that technical networks can be constructed in real time as well as offer some evidence that potential recommendations can prevent build failures from occurring.

In our close relationship with the IBM Rational Team Concert team we had the chance to interview ten developers, which represent a fraction of the development team at large. These ten developers were all located at the same site. As a result of this, our interview data could be biased and unrepresentative of the RTC team at large. However, we are confident that this threat is minor, due to the mix of developers we interviewed, including novices, senior developers, and team members that had been part of the group since its beginning. Furthermore, the triangulation with our observations and survey responses increases the confidence in our findings.

Construct Validity In this dissertation, we conceptualized social dependencies among developers using digitally recorded communication artifacts in the form of work item discussions, as well as relied on technical dependencies inferred from developers changing the same source code file. Both constructs are used by the software engineering research community in several studies (e.g. [19]). Nevertheless, both the social and technical dependency characterizations come with the danger that they do not necessarily measure social or technical dependencies of relevance or might as well miss existing dependencies. This leads to the threat that our inferences might be based on inconsistencies in the data, such as meaningless communication among developers, or file changes that are not technical in nature. For instance, due to storage problems, the Jazz teams erased some build results. In the case of nightly builds we expected 90 builds (according to project duration) but found only 15. This could possibly affect our results, but we argue that due to our richness of data the general trend remains preserved. Given that we use data that was generated by highly disciplined professionals, or by students that we monitored, we are confident that the data available for analysis is of high quality.

Internal Validity Chapters 6 and 8 demonstrated that constructing the socio-technical networks is feasible, and in Chapter 8 we showed that there is a relationship between the network configuration and build success that can be used to generate recommendations.

One issue that we will need to address in future work is showing a definite link between the insights presented in Chapter 8 and the actual build failures as well as to what extend the recommendations can actually prevent build failures from happening. To mitigate this threat, we demonstrated some initial evidence of tracing a failed build back to its original failure source and showed that the failure could have been prevented using the socio-technical information available at the point in time when the error was introduced into the code base.

Another threat to the approach, which is related to the previously mentioned lack of tracing the basis of the recommendations back to actual build failures, is that we did not test it in the field to see how the recommendation would affect the development process. In Chapter 9, we presented a study that explored if the recommendations are made at an appropriate level of granularity as well looked at the feedback concerning the usefulness of such recommendations. Furthermore, the study conducted in a classroom setting also suggests that there is value in generating such recommendations.

The surveys we deployed in our qualitative studies (cf. Chapter 9 and 10) survey asked developers to answer closed questions with a pre-defined list of answers that might introduce a bias. This bias poses a threat to our findings due to the possibility that we were missing important items. We mitigated it by developing the survey iteratively by piloting and discussing it with one of the development teams to identify the most important items, and by relying on our other two sources of data to triangulate our findings.

11.4 Future Work

In this dissertation, we illustrated an approach to leverage the concept of social-technical congruence in order to generate actionable knowledge. This five step approach focuses on defining two key parameters up front: (1) the scope of interest and (2) the outcome metric of interest. The first parameter scope helps with constructing the social networks (the third step) and constructing the technical networks (the fourth step) by supporting the selection of the best data sources. The outcome metric guides the analysis in producing actionable knowledge in the form of indicators that positively or negatively influence the outcome metric (step 5).

We see four promising directions we can take based on the research we presented in this dissertation:

- Implement and deploy the recommender system

- Extend the recommender system with more technical dependencies
- Extend the recommender system by generalizing the recommendations
- Investigate architectures that better fit organizational structures

11.4.1 Implement and Deploy the Recommender System

The approach we presented and validated in this dissertation needs to be implemented and deployed to test its effects in a real development environment. This implementation would also follow the recommendation we mentioned in Section 11.2.4, by allowing the user to manually input data that cannot be automatically inferred from recorded data.

The purpose of this deployment is twofold. The deployment creates a baseline for future improvements on the recommendation. Furthermore, the deployment of the system to actual developers will provide developers with insights, and support the development effort by reducing the risk of build failure.

11.4.2 Extend the Recommender System with more Technical Dependencies

In this dissertation, we defined two developers as sharing a technical dependency if they modified the same file. This technical dependency is very basic and missed several code dependencies that define dependencies among code artifacts. For instance, Schröter et al. [101] showed that usage relationships point to failures. Other measures that rely on dependencies among software artifacts are also good predictors for failures (e.g. [81]).

Using multiple types of technical dependencies will not only increase the accuracy of our approach, but it also enables us to provide recommendations that offer more insights into the reason for failure. Leveraging the different technical dependencies allows us to further prioritize recommendations based on the predictive power of a technical relationship. Moreover, developers using the recommender system can decide which technical dependency is most relevant for their work and filter recommendations based on that information.

11.4.3 Extend the Recommender System by Generalizing the Recommendations

Currently, recommendations are specific to developers. To further explore patterns of coordination related to failures and to extract more general rules, we plan on generalizing the developers. Developer can be characterized in several ways; such as their roles, positions or experience levels.

The generalization of developers into roles or positions allows for the knowledge of one project to be transferred to other projects. We plan to extract these more general patterns from several projects in order to find general rules that can be used to guide new projects that cannot rely on a rich history to generate recommendations generated by our approach.

11.4.4 Investigate Architectures that Better Fit Organizational Structures

Another interesting avenue to explore is what software architecture can support what types of communication and organizational structure. So far, the research surrounding socio-technical congruence is leading in the direction of changing how software developers coordinate their work. However, we propose returning to the original observation Conway made in that the software architecture will change to accommodate the communication structures in an organization. Therefore, analyzing software architectures with respect to the project properties, such as distribution of the development team, or the organizational hierarchy, might yield valuable insight in guiding design decisions of the software product that not only take into account properties to increase the feature richness or maintainability of the software product, but is optimal with respect to properties of the organization and the development team in order to increase productivity and quality.

Bibliography

[1] Roberto Abreu and Rahul Premraj. How Developer Communication Frequency Relates to Bug Introducing Changes. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, (IWPSE-Evol 2009), pages 153–158. ACM, 2009.

[2] Jorge Aranda and Gina Venolia. The Secret Life of Bugs: Going Past the Errors and Omissions in Software Repositories. In *Proceedings of the 31st ACM/IEEE International Conference on Software Engineering*, (ICSE 2009), pages 298–308. ACM, May 2009.

[3] Sarita Bassil and Rudolf K. Keller. Software Visualization Tools: Survey and Analysis. In *Proceedings of the 9th IEEE International Workshop on Program Comprehension*, (IWPC 2001), pages 7–17. IEEE Computer Society, 2001.

[4] Andrew Begel, Yit Phang Khoo, and Thomas Zimmermann. Codebook: Discovering and Exploiting Relationships in Software Repositories. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, (ICSE 2010), pages 125–134. ACM, May 2010.

[5] Andrew Begel and Beth Simon. Struggles of New College Graduates in Their First Software Development Job. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, (SIGCSE 2008), pages 226–230. ACM, 2008.

[6] Robert M. Bell. Predicting the Location and Number of Faults in Large Software Systems. *IEEE Transactions on Software Engineering, (TSE 2005)*, 31(4):340–355, 2005.

[7] Christian Bird, Nachiappan Nagappan, Premkumar Devanbu, Harald Gall, and Brendan Murphy. Does Distributed Development Affect Software Quality? An Empirical

Case Study of Windows Vista. In *Proceedings of the 31st ACM/IEEE International Conference on Software Engineering*, (ICSE 2009), pages 78–88. IEEE Computer Society, 2009.

[8] Christian Bird, Nachiappan Nagappan, Harald Gall, Brendan Murphy, and Premkumar Devanbu. Putting It All Together: Using Socio-technical Networks to Predict Failures. In *Proceedings of the 20th IEEE International Symposium on Software Reliability Engineering*, (ISSRE 2009), pages 109–119. IEEE Computer Society, November 2009.

[9] Kelly Blincoe, Giuseppe Valetto, and Sean Goggins. Proximity: A Measure to Quantify the Need for Developers' Coordination. In *Proceedings of the 26th 2012 Conference on Computer Supported Cooperative Work*, (CSCW 2012), pages 1351–1360. ACM, 2012.

[10] Barry Boehm. A Spiral Model of Software Development and Enhancement. *SIGSOFT Software Engineering Notes, (SEN 1986)*, 11(4):14–24, August 1986.

[11] Barry Boehm, Chris Abts, and Sunita Chulani. Software Development Cost Estimation Approaches: A Survey. *Annals of Software Engineering, (ASE 2000)*, 10:177–205, 2000.

[12] Francesco Bolici, James Howison, and Kevin Crowston. Coordination without Discussion? Socio-Technical Congruence and Stigmergy in Free and Open Source Software Projects. In *Proceedings of the Second International Workshop on Socio-Technical Congruence*, (STC 2009), pages 1–10. ACM, 2009.

[13] Arber Borici, Adrian Schröter, Daniela Damian, Kelly Blincoe, and Guiseppe Valetto. ProxiScientia: Toward Real-Time Visualization of Task and Developer Dependencies in Collaborating Software Development Teams. In *Proceedings of the 5th International Workshop on Cooperative and Human Aspects of Software Enginering*, (CHASE 2012), pages 932–935. ACM, 2012.

[14] Ronald Burt. *Structural Holes: The Social Structure of Competition*. Harvard University Press, August 1995.

[15] Scott Carter, Jennifer Mankoff, and P. Goddi. Building Connections among Loosely Coupled Groups: Hebb's Rule at Work. *Computer Supported Cooperative Work, (CSCW 2004)*, 13(3-4):305–327, 2004.

[16] Marcelo Cataldo, Matthew Bass, James D. Herbsleb, and Len Bass. On Coordination Mechanisms in Global Software Development. In *Proceedings of the 2nd IEEE International Conference on Global Software Engineering*, (ICGSE 2007), pages 71–80. IEEE Computer Society, 2007.

[17] Marcelo Cataldo, James D. Herbsleb, and Kathleen M. Carley. Socio-Technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity. In *Proceedings of the Second ACM International Symposium on Empirical Software Engineering and Measurement*, (ESEM 2008), pages 2–11. ACM, 2008.

[18] Marcelo Cataldo and Sangeeth Nambiar. Quality in Global Software Development Projects: A Closer Look at the Role of Distribution. In *Proceedings of the fourth IEEE International Conference on Global Software Engineering*, (ICGSE 2009), pages 163–172. IEEE Computer Society, July 2009.

[19] Marcelo Cataldo, Patrick A. Wagstrom, James D. Herbsleb, and Kathleen M. Carley. Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools. In *Proceedings of the 20th ACM Anniversary Conference on Computer Supported Cooperative Work*, (CSCW 2006), pages 353–362. ACM, 2006.

[20] Melvin E. Conway. How do Comittees Invent? *Datamation*, 14(4):28–31, 1968.

[21] Thomas Cortina and Ellen Walker, editors. *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education, (SIGCSE 2011)*. ACM, 2011. 457110.

[22] Davor Cubranic, Gail C. Murphy, Janice Singer, and Kellogg S. Booth. Hipikat: A Project Memory for Software Development. *IEEE Transactions on Software Engineering, (TSE 2005)*, 31:446–465, 2005.

[23] Bill Curtis, Herb Krasner, and Neil Iscoe. A Field Study of the Software Design Process for Large Systems. *Communication of the ACM, (CACM 1988)*, 31(11):1268–1287, 1988.

[24] Michael A. Cusumano and Richard W. Selby. How Microsoft Builds Software. *Communication of the ACM, (CACM 1997)*, 40(6):53–61, 1997.

[25] Daniela Damian, Luis Izquierdo, Janice Singer, and Irwin Kwan. Awareness in the Wild: Why Communication Breakdowns Occur. In *Proceedings of the Second IEEE International Conference on Global Software Engineering*, (ICGSE 2007), pages 81–90. IEEE Computer Society, 2007.

[26] Daniela Damian, Sabrina Marczak, and Irwin Kwan. Collaboration Patterns and the Impact of Distance on Awareness in Requirements-Centered Social Networks. In *In Proceedings of the 15th IEEE International Requirements Engineering Conference*, (RE 2007), pages 59–68. IEEE Computer Society, October 2007.

[27] Cleidson R. B. de Souza and David Redmiles. The Awareness Network: To Whom Should I Display My Actions? And, Whose Actions Should I Monitor? In *Proceedings of the 21st ACM European Conference on Computer Supported Cooperative Work*, (CSCW 2007), pages 325–340. ACM, September 2007.

[28] Cleidson R. B. de Souza and David F. Redmiles. An Empirical Study of Software Developers' Management of Dependencies and Changes. In *Proceedings of the 31st ACM/IEEE International Conference on Software Engineering*, ICSE 2009, pages 241–250. IEEE Computer Society, May 2008.

[29] Cleidson R. B. d. de Souza, David Redmiles, Li-Te Cheng, David Millen, and John Patterson. How a Good Software Practice Thwarts Collaboration: The Multiple Roles of APIs in Software Development. *Software Engineering Notes (SEN 2004)*, pages 221–230, 2004.

[30] Kevin C. Desouza, Yukika Awazu, and Peter Baloh. Managing Knowledge in Global Software Development Efforts: Issues and Practices. *IEEE Software, (Software 2006)*, 23(5):30–37, 2006.

[31] Nicolas Ducheneaut. Socialization in an Open Source Software Community: A Socio-Technical Analysis. *Computer Supported Cooperative Work, (CSCW 2005)*, 14(4):323–368, 2005.

[32] Kate Ehrlich and Klarissa Chang. Leveraging Expertise in Global Software Teams: Going Outside Boundaries. In *Proceedings of the First IEEE International Conference on Global Software Engineering*, (ICGSE 2006), pages 149–158. IEEE Computer Society, 2006.

[33] Kate Ehrlich, Mary Helander, Guiseppe Valetto, Stephen Davies, and Clay Williams. An Analysis of Congruence Gaps and Their Effect on Distributed Software Development. In *Proceedings of the First International Workshop on Socio-Technical Congruence*, (STC 2008), pages 1–10. IEEE Computer Society, 2008.

[34] Alberto Espinosa, Sandra A. Slaughter, Robert E. Kraut, and James D. Herbsleb. Team Knowledge and Coordination in Geographically Distributed Software Development. *Journal of Management Information Systems, (JIMS 2007)*, 24(1):135–169, 2007.

[35] Samer Faraj and Lee Sproull. Coordinating Expertise in Software Development Teams. *Journal of Management Science, (JMS 2000)*, 46(12):1554–1568, 2000.

[36] Linton C. Freeman. Centrality in Social Networks: Conceptual Clarification. *Social Networks*, 1(3):215–239, 1979.

[37] Randall Frost. Jazz and the Eclipse Way of Collaboration. *IEEE Software, (Software 2007)*, 24:114–117, November 2007.

[38] Susan R. Fussell, Robert E. Kraut, Javier Lerch, William L. Scherlis, Matthew M. McNally, and Jonathan J. Cadiz. Coordination, Overload and Team Performance: Effects of Team Communication Strategies. In *Proceedings of the 12th ACM Conference on Computer Supported Cooperative Work*, (CSCW 1998), pages 275–284. ACM, 1998.

[39] Peter A. Gloor, Rob Laubacher, Scott B. C. Dynes, and Yan Zhao. Visualization of Communication Patterns in Collaborative Innovation Networks - Analysis of Some W3C Working Groups. In *Proceedings of the 12th ACM International Conference on Information and Knowledge Management*, (ICIKM 2003), pages 56–60, 2003.

[40] Claudia N. Gonzalez-Brambila and Francisco Veloso. Social Capital in Academic Engineers. In *Compendium Portland International Center for Management of Engineering and Technology*, pages 2565–2572, 5-9 Aug. 2007.

[41] Anandasivam Gopal, Tridas Mukhopadhyay, and Mayuram S. Krishnan. The Role of Software Processes and Communication in Offshore Software Development. *Communication of the ACM, (CACM 2002)*, 45(4):193–200, April 2002.

[42] Abbie Griffin and John R. Hauser. Patterns of Communication Among Marketing, Engineering and Manufactoring–A Comparison Between Two New Product Teams. *Journal of Management Science, (JMS 1992)*, 38(3):360–373, 1992.

[43] Rebecca E. Grinter, James D. Herbsleb, and Dewayne E. Perry. The Geography of Coordination: Dealing with Distance in R&D Work. In *Proceedings of the 2nd ACM International Conference on Supporting Group Work*, (GROUP 1999), pages 306–315. ACM, 1999.

[44] Carl Gutwin, Reagan Penner, and Kevin Schneider. Group Awareness in Distributed Software Development. In *Proceedings of the 18th ACM Conference on Computer-Supported Cooperative Work*, (CSCW 2004), pages 72–81, November 2004.

[45] Mary Hall, editor. *PLDI '2011: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2011. 548110.

[46] Andrew Hargadon and Robert I. Sutton. Technology Brokering and Innovation in a Product Development Firm. *Administrative Science Quarterly, (ASQ 1997)*, 42(4):716–749, 1997.

[47] Ahmed E. Hassan and Ken Zhang. Using Decision Trees to Predict the Certification Result of a Build. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering*, ASE 2006, pages 189–198. IEEE Computer Society, 2006.

[48] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Verlag, 3 edition, July 2003.

[49] James D. Herbsleb. Global Software Engineering: The Future of Socio-technical Coordination. In *In Proceedings of Workshop on the Future of Software Engineering*, (FOSE 2007), pages 188–198, 2007.

[50] James D. Herbsleb and Rebecca E. Grinter. Architectures, Coordination, and Distance: Conway's Law and Beyond. *IEEE Software, (Software 1999)*, 16(5):63–70, 1999.

[51] James D. Herbsleb and Rebecca E. Grinter. Splitting the Organization and Integrating the Code: Conway's Law Revisited. In *Proceedings of the 21st ACM/IEEE In-*

ternational Conference on Software Engineering, (ICSE 1999), pages 85–95. ACM, 1999.

[52] James D. Herbsleb and Audris Mockus. An Empirical Study of Speed and Communication in Globally Distributed Software Development. *IEEE Transactions on Software Engineering, (TSE 2003)*, 29(6):481–494, 2003.

[53] James D. Herbsleb, Audris Mockus, Thomas A. Finholt, and Rebecca E. Grinter. An Empirical Study of Global Software Development: Distance and Speed. In *Proceedings of the 23rd ACM/IEEE International Conference on Software Engineering*, (ICSE 2001), pages 81–90. IEEE Computer Society, 2001.

[54] James D. Herbsleb, Audris Mockus, and Jeffrey A. Roberts. Collaboration in Software Engineering Projects: A Theory of Coordination. In *Proceedings of the International Conference on Information Systems*, (ICIS 2006), pages 59–69. Association for Information Systems, 2006.

[55] Pamela Hinds and Cathleen McGrath. Structures that Work: Social Structure, Work Structure and Coordination Ease in Geographically Distributed Teams. In *Proceedings of the 20th ACM Anniversary Conference on Computer Supported Cooperative Work*, (CSCW 2006), pages 343–352. ACM, 2006.

[56] Jesper Holck and Niels Jørgensen. Continuous Integration and Quality Assurance: A Case Study of Two Open Source Projects. *Australasian Journal of Information Systems, (AJIS 2003)*, pages 40–53, 2003.

[57] Liaquat Hossain, Andrè Wu, and Kenneth K. S. Chung. Actor Centrality Correlates to Project Based Coordination. In *Proceedings of the 20th ACM Anniversary Conference on Computer Supported Cooperative Work*, (CSCW 2006), pages 363–372. ACM, 2006.

[58] Mik Kersten and Gail C. Murphy. Mylar: A Degree-of-Interest Model for IDEs. In *Proceedings of the 4th ACM International Conference on Aspect-Oriented Software Development*, (AOSD 2005), pages 159–168. ACM, 2005.

[59] Sunghun Kim, James Whitehead, and Yi Zhang. Classifying Software Changes: Clean or Buggy? *IEEE Transactions on Software Engineering, (TSE 2008)*, 34(2):181–196, 2008.

[60] Sunghun Kim, Thomas Zimmermann, James Whitehead, and Andreas Zeller. Predicting Faults from Cached History. In *Proceedings of the 29th ACM/IEEE International Conference on Software Engineering*, (ICSE 2007), pages 489–498. IEEE Computer Society, 2007.

[61] Tim Klinger, Peri Tarr, Patrick Wagstrom, and Clay Williams. An Enterprise Perspective on Technical Debt. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, (MTD 2011), pages 35–38. ACM, 2011.

[62] Jens Knoop, editor. *Proceedings of the 20th International Conference on Compiler Construction (CC 2011): Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2011)*. Springer-Verlag, 2011.

[63] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information Needs in Collocated Software Development Teams. In *Proceedings of the 29th ACM/IEEE International Conference on Software Engineering*, (ICSE 2007), pages 344–353. IEEE Computer Society, 2007.

[64] Robert E. Kraut and Lynn A. Streeter. Coordination in Software Development. *Communications of the ACM, (CACM 1995)*, 38(3):69–81, March 1995.

[65] Irwin Kwan, Adrian Schröter, and Daniela Damian. A Weighted Congruence Measure. In *Proceedings of 2nd International Workshop on Socio-Technical Congruence*, (STC 2009), pages 1–10, May 19 2009.

[66] Irwin Kwan, Adrian Schröter, and Daniela Damian. Does Socio-Technical Congruence Have An Effect on Software Build Success? A Study of Coordination in Software Project. *IEEE Transactions on Software Engineering, (TSE 2011)*, 2011.

[67] Craig Larman and Victor R. Basili. Iterative and Incremental Development: A Brief History. *IEEE Computer, (Computer 2003)*, 36(6):47–56, June 2003.

[68] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining Mental Models: A Study of Developer Work Habits. In *Proceedings of the 28th ACM/IEEE International Conference on Software Engineering*, (ICSE 2006), pages 492–501. ACM, 2006.

[69] Timothy C. Lethbridge, Susan Elliott Sim, and Janice Singer. Studying Software Engineers: Data Collection Techniques for Software Field Studies. *Empirical Software Engineering, (ESE 2005)*, 10:311–341, 2005.

[70] Wayne G. Lutters and Carolyn Seaman. The Value of War Stories in Debunking the Myths of Documentation in Software Maintenance. *Information and Software Technology, (IST 2007)*, 49(6):576–587, 2007.

[71] Alan MacCormack, John Rusnak, and Carliss Y. Baldwin. Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code. *Journal of Management Science, (JMS 2006)*, 52(7):1015–1030, July 2006.

[72] Vincent Maraia. *The Build Master: Microsoft's Software Configuration Management Best Practices*. Addison-Wesley, 2005.

[73] Sabrina Marczak, Daniela Damian, Ulrike Stege, and Adrian Schröter. Information Brokers in Requirement-Dependency Social Networks. In *Proceedings of the16th IEEE International Conference on Requirements Engineering*, (RE 2008), pages 53–62. IEEE Computer Society, 2008.

[74] Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. Predicting Failures with Developer Networks and Social Network Analysis. In *Proceedings of the 16th ACM International Symposium on Foundations of Software Engineering*, (FSE 2008), pages 13–23. ACM, 2008.

[75] Tom Mens and Tom Tourwe. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering, (TSE 2004)*, 30(2):126–139, February 2004.

[76] Tim Menzies, editor. *Proceedings of the 7th ACM International Conference on Predictive Models in Software Engineering, (Promise 2011)*. ACM, 2011.

[77] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Transactions on Software Engineering Methodology, (TSEM 2002)*, 11(3):309–346, 2002.

[78] Kjetil Molkken and Magne Jørgensen. A Review of Surveys on Software Effort Estimation. In *Proceedings of the IEEE International Symposium on Empirical Software Engineering*, (ISESE 2003), pages 223–230. IEEE Computer Society, 2003.

[79] Gail C. Murphy and Emerson Murphy-Hill. What is Trust in a Recommender for Software Development? In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, (RSSE 2010), pages 57–58. ACM, 2010.

[80] Nachiappan Nagappan and Thomas Ball. Use of Relative Code Churn Measures to Predict System Defect Density. In *Proceedings of the 27th ACM/IEEE International Conference on Software Engineering*, (ICSE 2005), pages 284–292. ACM, 2005.

[81] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining Metrics to Predict Component Failures. In *Proceedings of the 28th ACM/IEEE International Conference on Software Engineering*, (ICSE 2006), pages 452–461. ACM, 2006.

[82] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. The Influence of Organizational Structure on Software Quality: An Empirical Case Study. In *Proceedings of the 30th ACM/IEEE International Conference on Software Engineering*, (ICSE 2008), pages 521–530. ACM, 2008.

[83] K. Nakakoji, Y. Ye, and Y. Yamamoto. Supporting Expertise Communication in Developer-Centered Collaborative Software Development Environments. In A. Frinkelstein, J. Grundy, A. van der Hoek, I. Mistrik, and James Whitehead, editors, *Collaborative Software Engineering*, pages 219–236. Springer-Verlag, 2010.

[84] Thanh Nguyen, Timo Wolf, and Daniela Damian. Global Software Development and Delay: Does Distance Still Matter? In *Proceedings of the 3rd International Conference on Global Software Engineering*, (ICGSE 2008), pages 45–54. IEEE Computer Society, August 2008.

[85] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. Recurring Bug Fixes in Object-Oriented Programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, (ICSE 2010), pages 315–324. ACM, 2010.

[86] Tuomas Niinimäki and Casper Lassenius. Experiences of Instant Messaging in Global Software Development Projects: A Multiple Case Study. In *Proceedings of the First IEEE International Conference on Global Software Engineering*, (ICGSE 2008), pages 55–64, July 2008.

[87] Gary M. Olson and Judith S. Olson. Distance Matters. *Human-Computer Interaction, (HCI 2000)*, 15:139–178, September 2000.

[88] Dewayne E. Perry, Nancy A. Staudenmayer, and Lawrence G. Votta. People, Organizations, and Process Improvement. *IEEE Software, (Software 1994)*, 11(4):36–45, 1994.

[89] Mary Beth Pinto and Jeffrey K. Pinto. Project Team Communication and Cross-Functional Cooperation in New Program Development. *Journal of Product Innovation Management, (JPIM 1990)*, 7(3):200–212, 1990.

[90] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. Can Developer-Module Networks Predict Failures? In *Proceedings of the 16th ACM International Symposium on Foundations of Software Engineering*, (FSE 2008), pages 2–12. ACM, 2008.

[91] David Raffo, editor. *Proceedings of the 2011 International Conference on Software and Systems Process (ICSSP 2011)*. ACM, 2011.

[92] Ray Reagans and Ezra W. Zuckerman. Networks, Diversity, and Productivity: The Social Capital of Corporate R&D Teams. *Organization Science, (OS 2001)*, 12(4):502–517, 2001.

[93] David Redmiles, Andre van der Hoek, Ban Al-Ani, Stephen Quirk, Anita Sarma, Silva Filho, Cleidson de Souza, and Erik Trainer. Continuous Coordination: A New Paradigm to Support Globally Distributed Software Development Projects. *Wirtschaftsinformatik, (WI 2007)*, 49:28–38, 2007.

[94] Diane L. Rulke and Joseph Galaskiewicz. Distribution of Knowledge, Group Network Structure, and Group Performance. *Journal of Management Science, (JMS 2000)*, 46(5):612–625, 2000.

[95] Anita Sarma, Larry Maccherone, Patrick Wagstrom, and James Herbsleb. Tesseract: Interactive Visual Exploration of Socio-Technical Relationships in Software Development. In *Proceedings of the 31st ACM/IEEE International Conference on Software Engineering*, (ICSE 2009), pages 23–33. IEEE Computer Society, 2009.

[96] Anita Sarma and Andre van der Hoek. Palantir: Coordinating Distributed Workspaces. In *Proceedings of the 26th International Conference of Computer Software and Applications*, (COMPSAC 2002), pages 1093–1097, 2002.

[97] Anita Sarma and Andre van der Hoek. Towards Awareness in the Large. In *Proceedings of the First IEEE International Conference on Global Software Engineering*, (ICGSE 2006), pages 127–131, 2006.

[98] Steve Sawyer. Software Development Teams. *Communication of the ACM, (CACM 2004)*, 47(12):95–99, 2004.

[99] Kjeld Schmidt and Carla Simone. Coordination Mechanisms: Toward a Conceptual Foundation of CSCW Systems Design. *Computer Supported Cooperative Work, (CSCW 1996)*, 5:155–200, 1996.

[100] Adrian Schröter, Jorge Aranda, Daniela Damian, and Irwin Kwan. To talk or Not To Talk: Factors That Influence Communication Around Changesets. In *Proceedings of the 26th ACM Conference on Computer Supported Cooperative Work*, (CSCW 2012), pages 1317–1326. ACM, 2012.

[101] Adrian Schröter, Thomas Zimmermann, and Andreas Zeller. Predicting Component Failures at Design Time. In *Proceedings of the Fifth ACM International Symposium on Empirical Software Engineering*, (ISESE 2006), pages 18–27. ACM, 2006.

[102] David J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, 2000.

[103] Sidney Siegel. *Nonparametric Statistics for the Behavioral Sciences*. McGraw-Hill Humanities, 1st edition, 1956.

[104] William E. Souder. Managing Relations Between R&D and Marketing in New Product Development Projects. *Journal of Product Innovation Management, (JPIM 1998)*, 5(1):6–19, 1988.

[105] Erik Trainer, Stephen Quirk, Cleidson de Souza, and David Redmiles. Bridging the Gap Between Technical and Social Dependencies with Ariadne. In *Proceedings of the OOPSLA Workshop on Eclipse Technology eXchange*, (ETX 2005), pages 26–30. ACM, 2005.

[106] Giuseppe Valetto, Mary Helander, Kate Ehrlich, Sunita Chulani, Mark Wegman, and Clay Williams. Using Software Repositories to Investigate Socio-technical Congruence in Development Projects. In *Proceedings of the Fourth IEEE International Workshop on Mining Software Repositories*, MSR 2007, page 25. IEEE Computer Society, 2007.

[107] Jilles van Gurp and Jan Bosch. Design Erosion: Problems and Causes. *Journal of Systems and Software, (JSS 2002)*, 61(2):105–119, March 2002.

[108] Andrew H. Van De Ven, Andre L. Delbecq, and Jr. Richard Koenig. Determinants of Coordination Modes within Organizations. *American Sociological Review, (ASR 1976)*, 41(2):322–338, 1976.

[109] Anneliese von Mayrhauser and A. Marie Vans. Program Comprehension During Software Maintenance and Evolution. *IEEE Computer, (Computer 1995)*, 28(8):44–55, August 1995.

[110] Stanley Wasserman and Katherine Faust. *Social Network Analysis: Methods and Applications*. Cambridge University Press, November 1994.

[111] Timo Wolf, Adrian Schröter, Daniela Damian, and Thanh Nguyen. Predicting Build Failures Using Social Network Analysis on Developer Communication. In *Proceedings of the 31st ACM/IEEE International Conference on Software Engineering*, (ICSE 2009), pages 1–11. IEEE Computer Society, 2009.

[112] Timo Wolf, Adrian Schröter, Daniela Damian, Lucas D. Panjer, and Thanh H. D. Nguyen. Mining Task-Based Social Networks to Explore Collaboration in Software Teams. *IEEE Software, (Software 2009)*, 26(1):58–66, 2009.

[113] P. F. Xiang, A. T. T. Ying, P. Cheng, Y. B. Dang, K. Ehrlich, M. E. Helander, P. M. Matchen, A. Empere, P. L. Tarr, C. Williams, and S. X. Yang. Ensemble: A Recommendation Tool for Promoting Communication in Software Teams. In *Proceedings of the International Workshop on Recommendation Systems for Software Engineering*, (RSSE 2008), pages 21–25. ACM, 2008.

[114] Thomas Zimmermann, Valentin Dallmeier, Konstantin Halachev, and Andreas Zeller. eROSE: Guiding Programmers in Eclipse. In *Proceedings of the 20th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, (OOPSLA 2005), pages 186–187. ACM, 2005.

[115] Thomas Zimmermann and Nachiappan Nagappan. Predicting Defects Using Network Analysis on Dependency Graphs. In *Proceedings of the 30th ACM/IEEE International Conference on Software Engineering*, (ICSE 2008), pages 531–540. ACM, 2008.

[116] Thomas Zimmermann and Nachiappan Nagappan. Predicting Defects with Program Dependencies. In *Proceedings of the 3rd IEEE International Symposium on Empirical Software Engineering and Measurement*, (ESEM 2009), pages 435–438. IEEE Computer Society, 2009.