Doctoral Thesis

Understanding and Improving Collaboration in Distributed Software Development

Author(s):
Estler, Hans-Christian

Publication Date:
2014

Permanent Link:
https://doi.org/10.3929/ethz-a-010439447

Rights / License:
In Copyright - Non-Commercial Use Permitted
Understanding and Improving Collaboration in Distributed Software Development

A thesis submitted to attain the degree of
DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by
HANS-CHRISTIAN ESTLER
Master of Science, University of Paderborn, Germany
Dipl.-Wirt.-Inf., University of Paderborn, Germany

born on
June 25th, 1983

citizen of
Germany

accepted on the recommendation of

Prof. Dr. Bertrand Meyer, examiner
Prof. Dr. Natalia Juristo, co-examiner
Dr. Martin Nordio, co-examiner
Prof. Dr. David Redmiles, co-examiner

2014
Many people helped bring this thesis into existence. I cannot mention all of them, but I’d like to explicitly express my gratitude to some of those who accompanied me over the last years.

I thank Bertrand Meyer for supervising me, for allowing me to join his exceptional group at ETH Zurich, and for giving me the freedom and the opportunity to develop and pursue my own research ideas and interests.

I’m also grateful to Natalia Juristo and David Redmiles for serving as co-examiners on my committee. I was fortunate to get to know both of them not only as examiners but as researchers and experts in their field. Their work improved my skills as a researcher and I’m thankful for their insightful comments, and their well-advised feedback on this thesis and my research in general.

Martin Nordio has been my day-to-day accomplice over the last four years and I cannot overemphasize the impact he had on this work. He’s been a supervisor, mentor, colleague, and friend, and I consider myself very lucky that he has never hesitated to share and explore his “crazy ideas” together with me.

I’d also like to thank Carlo A. Furia for joining Martin and me on numerous research projects. Carlo’s brilliance and ever-lasting optimism have been instrumental to the work presented in this thesis.

Nadia Polikarpova and Scott West, my long-term office mates, were the best colleagues and friends I could have wished for. Their intelligence, humor, kindness, and incredible tolerance for my jokes\textsuperscript{1} made it a pleasure to come to work every day.

In addition to the people I’ve already mentioned, I’d like to thank all other members of the Chair of Software Engineering at ETH. It’s been a pleasure to work alongside such a phenomenal group of people and many of them have helped me out along the way more than once. Thank you Cristiano Calcagno, Georgiana Caltais, Claudia Günthart, Alexey Kolesnichenko,\textsuperscript{1}Verdienst!
Benjamin Morandi, Sebastian Nanz, Đurica Nikolić, Marco Piccioni, Chris Poskitt, Andrey Rusakov, Mischael Schill, Roman Schmocker, Jiwon Shin, Stephan van Staden, Marco Trudel, Julian Tschannen, Yu Pei, and Yi Wei.

I’m grateful to the numerous students (there are too many to name them all), at ETH and at other universities all around the world, who have contributed to my research, some of them in the form of theses or semester projects, others by participating in experiments that provided us with valuable data.

Finally, I’d like to thank my family for their unconditional support and encouragement, not only during the time of this PhD, but throughout all my life. And I’d like to thank Ann for proving, together with me, that a relationship can work in a distributed setting, after all.

**Financing**

This work was partially supported by the Gebert-Rüf Stiftung under the project *CloudStudio: Software Development on the Cloud* (GRS-057/09) and by the Swiss SNF under the project *Models and Tools for Collaboration in Distributed Software Engineering* (153512).
Contents

1 Introduction 1
  1.1 Motivation and Goal 1
  1.2 Challenges and Contributions 2
  1.3 How to Read this Thesis 5
  1.4 Outline 5

2 Communication in Distributed Teams 7
  2.1 Introduction 7
  2.2 Research Questions 8
  2.3 Context of this Study 9
    2.3.1 The DOSE Course 9
    2.3.2 Team and Group Structure 10
    2.3.3 Project Organization 11
    2.3.4 Project Outcomes 12
  2.4 Data Collection and Classification 12
    2.4.1 Data Collection 12
    2.4.2 Data Classification 12
  2.5 Results 14
    2.5.1 Data Transformation 14
    2.5.2 Communication in two and three-location projects 15
    2.5.3 Communication in projects in different time zones 17
    2.5.4 Email reply time in different time zones 20
  2.6 Students’ Feedback 23
  2.7 Threats to Validity 25
  2.8 Related Work 26
  2.9 Summary 28

3 The Role of Development Processes in GSE 29
  3.1 Introduction and Overview 29
4 Conflicts and Awareness

4.1 Introduction .................................................. 59
4.2 Design of the Empirical Study ............................... 61
  4.2.1 Data Collection ......................................... 62
  4.2.2 Participants ............................................. 64
4.3 Results of the Empirical Study .............................. 66
  4.3.1 Merge Conflicts: Frequency and Origin .................. 66
  4.3.2 Workflow Interruptions: Frequency and Origin .......... 68
  4.3.3 Conflicts and Interruptions: Impact and Effects ........ 71
  4.3.4 Awareness: Preferences and Granularity ............... 74
  4.3.5 Correlation Analysis ................................... 76
4.4 Threats to Validity ........................................... 77
4.5 Related Work ................................................ 77
4.6 Summary .................................................... 78

5 A Tool for Conflicts and Awareness .......................... 79

5.1 Introduction .................................................. 79
5.2 Related Work ................................................ 81
5.3 A Session with CloudStudio ................................. 85
5.4 How CloudStudio Unifies Configuration Management, Conflict Detection, and Awareness ............. 87
  5.4.1 Software Configuration Management Model ............... 88
  5.4.2 Real-time Awareness System .............................. 92
  5.4.3 Collaborative Editing .................................... 94
  5.4.4 Conflict Detection and Prevention ....................... 94
  5.4.5 Limitations of CloudStudio’s SCM ....................... 95
5.5 CloudStudio’s Prototype Implementation .................... 95
  5.5.1 Limitations of Current Implementation ................. 96
5.6 Case Study .................................................. 96
  5.6.1 Development Tasks ...................................... 96
  5.6.2 Subjects and Experimental Setup ....................... 97
  5.6.3 Results ................................................ 98
  5.6.4 Discussion ............................................. 99
5.7 Summary .................................................... 100

6 Contracts in Practice ........................................... 101

6.1 Introduction .................................................. 101
6.2 Study Setup ................................................ 103
6.3 How Contracts Are Used .................................... 105
  6.3.1 Writing Contracts ....................................... 107
  6.3.2 Contracts and Project Size .............................. 109
6.3.3 Kinds of Contract Elements ........................................ 110
6.3.4 Contract Size and Strength ....................................... 113
6.3.5 Implementation vs. Specification Changes ...................... 114
6.3.6 Inheritance and Contracts ........................................ 116
6.4 Threats to Validity ..................................................... 117
6.5 Related Work .......................................................... 118
6.6 Concluding Discussion & Implications of the Results .......... 119

7 Collaborative Debugging ................................................. 121
7.1 Introduction ............................................................ 121
7.2 Collaborative Debugging in DSD Teams ......................... 123
7.3 Collaborative Debugging with CDB ............................... 126
7.4 Empirical Study: Experimental Design ......................... 129
  7.4.1 Critical Debugger Features: RQ1 ............................. 132
  7.4.2 Control Policies: RQ2 ........................................... 133
  7.4.3 Debugging Experience: RQ3 ................................... 133
7.5 Empirical Study: Execution and Results ....................... 134
  7.5.1 Critical Debugger Features: RQ1 ............................. 135
  7.5.2 Control Policies: RQ2 ........................................... 140
  7.5.3 Debugging Experience: RQ3 ................................... 142
7.6 Threats to Validity ..................................................... 143
7.7 Related Work .......................................................... 143
  7.7.1 Tools for Distributed Software Development ............. 144
  7.7.2 Empirical Studies of Collaborative and DSD ............. 145
7.8 Summary ............................................................... 145

8 Conclusion .............................................................. 147
ABSTRACT

Distributed software development—in which developers are located in different geographical locations—is common practice in today’s software industry. Teams are distributed over different countries, work in different time zones, may speak different languages and have different cultural backgrounds. All these conditions make it a challenge to collaborate effectively and efficiently in a distributed team. This thesis aims to provide new insights on how teams are affected by distribution, what the consequences are for the projects they work on, and how new techniques and tools could help to improve collaboration.

The thesis investigates the impact of distribution on the communication amongst team members, showing that a substantial amount of the overall project time is spend on communication between the distributed parties of the development team. It further investigates the impact of different distribution scenarios, in particular, comparing distribution over two locations with distribution over three locations, and the role different time zone ranges have on the communication behavior of developers.

The thesis also analyzes if and how the choice of development process affects distributed projects. We study metrics such as the overall success, economic savings, or motivation of the development teams, depending on the type of software development process used. For this, we use data from industry projects developed in Europe, Asia, and the Americas. Our findings suggest that choice of process alone is not a critical factor to succeed in globally distributed development.

Another contribution of this thesis adds to our understanding of how often developers in distributed teams are insufficiently aware of the changes performed by other team members, and how often the changes of individual developers conflict with each other.

Additionally, this thesis presents novel approaches on how configuration management systems can be combined with tools to detect and prevent conflicts between individual developers’ changes. It also introduces a debugging technique and an integrated tool, specifically designed to support effective
collaboration among developers during shared debugging sessions. We dis-
cuss the design of both these tools and evaluate them in case studies, demon-
strating their usefulness in collaborative development.

Finally, this thesis presents an analysis on how programs are evolved over
time. Knowledge on how programs are changed is instrumental when building
tools that report changes to developers. In this contribution we focus on
programs and languages that support contracts, a form of lightweight formal
specifications, and analyze 21 contract-equipped projects written in Eiffel,
C#, and Java. Our findings suggest that awareness tools, which monitor
and analyze changes from developers, would benefit from adopting contracts
as a language element that should be monitored.
ZUSAMMENFASSUNG


In dieser Arbeit untersuchen wir den Einfluss von Verteilung auf die Kommunikation zwischen Teammitgliedern und zeigen auf, dass ein erheblicher Anteil der Zeit, die in ein Projekt investiert wird, für die Kommunikation zwischen verteilten Mitgliedern verwendet wird. Weiterhin untersucht diese Arbeit den Einfluss von verschiedenen Arten der Verteilung und vergleicht dabei insbesondere eine Verteilung auf zwei Standorte mit einer Verteilung auf drei Standorte und analysiert den Einfluss unterschiedlicher Zeitzonen auf die Kommunikation.


Ein weiterer Beitrag dieser Arbeit liefert neue Erkenntnisse darüber, wie oft Entwickler in verteilten Teams unzureichend darüber informiert sind, welche Änderungen von anderen Teammitgliedern vorgenommen werden und wie oft die Änderungen von einzelnen Entwicklern miteinander im Konflikt stehen.

CHAPTER 1

INTRODUCTION

1.1 Motivation and Goal

Many of today’s software projects are globally distributed. The development teams are located in different countries, different time zones, or even different continents. The need to reduce development costs or gain access to more talent causes companies to accept the challenges and risks that follow from introducing geographical distance into a project team.

Despite all advances in communication technology, aspects of collaboration, such as communication, coordination and control, are still difficult to manage in a distributed setting [15, 79].

Researchers have therefore attended to the phenomenon of distributed software engineering with two main goals: first, to improve our understanding of how distributed projects are carried out, and which factors make them succeed or fail. Second, to develop new approaches, methods and tools that allow teams to carry out distributed projects more effectively and efficiently.

Following these goals, this thesis contributes to the body of research on distributed software development by analyzing distributed projects as they are carried out in today’s software industry. Additionally, this thesis contributes novel ideas on how distributed teams can improve their collaboration by using tools that aim to make it easier to be aware of what the various developers of a team are working on, and improve how developers can work collaboratively on finding and fixing faults in programs.

While this thesis has a strong focus on distributed software development—and in most cases that connotes globally distributed development—several of the contributions presented are also applicable to co-located teams that are merely separated by working in different offices or on different computers. This is particularly true for the collaboration tools that we’ll present in the
following chapters.

1.2 Challenges and Contributions

To improve the status quo of how distributed teams work, it is important to understand the processes and factors involved. Gaining such understanding through empirical studies is a challenge; the course and success of a distributed project is influenced by many different factors, ranging from technical decisions over managerial aspects all the way to human factors such as language or cultural background. Furthermore, data sets for empirical studies are often difficult to obtain, especially in the presence of global geographical distance. This thesis benefits greatly from having access to a rare university setting that allowed us to study distributed development in an environment that resembles how distributed projects are carried out in industry: the ETH Zurich course on Distributed and Outsourced Software Engineering (DOSE). In this annual course about a dozen of universities from various countries enable students to work on a software project with a distributed team over the course of a semester. The data we collected from this academic setting complements the data sets we acquired from industry or from mining software repositories. The DOSE course will feature in several chapters of this thesis and each chapter will provide the details on how data was collected.

Communication in distributed teams. The first contribution of this thesis provides insights on how distributed teams and, in particular, their communication is affected if a team is split up over different time zones and different numbers of locations (teams split into two vs. team split into three). Based on data collected in three consecutive years of the DOSE course, we analyze the amount of time spend on communication relative to overall project time, how distribution and time zones affect teams’ communication behaviors and how email reply times change for teams that are distributed over different time zones.

The role of development processes in GSE. With the wide-spread adoption of agile processes in the software industry there is a trend among distributed teams to also adopt agile methods to their development. Such an adoption is often not a simple undertaking because agile processes demand strong communication and interaction amongst developers, as well as with product customers. A distributed setting thus requires modifications to the way agile teams usually work and several researchers have proposed and studied such modifications. It is unclear, however, if a distributed agile team performs better than a team that follows a more traditional, structured
process. We contribute an empirical investigation of this question and show that despite a fairly large sample size of 66 industry projects and many success-metrics analyzed we can not identify the choice of process as a sole critical factor to a distributed project’s success.

Conflicts and awareness. Improving collaboration amongst developers is a main goal of this thesis. One contribution that adds to this goal is an analysis on the frequency of merge-conflicts and a lack of awareness about ongoing changes in distributed teams. The findings of our study demonstrate that our work on awareness and conflict tools, as well as the works of other researchers, address a real problem that occurs frequently.

A tool for conflicts and awareness. Another contribution of this thesis—which directly relates to the previous one—is a novel approach on how to combine a version control system (e.g. Git or SVN) with a tool to detect merge conflicts among developers early and even prevent them, and a tool that informs developers about ongoing changes by other team members. We unify these tools in a single approach that integrates unobtrusively into an IDE. To demonstrate and evaluate our approach, we have developed CloudStudio IDE, a web-based IDE for collaborative development. Using a small-scale case study we show the potential benefits of our approach.

Contracts in practice. When building awareness tools, i.e. tools that inform developers about the changes of other team members, it’s critical to have a concept about which changes are important and thus deserve to be reported; after all, not every change is of interest to every other developer. A trade-off that researchers have proposed and implemented in other awareness tools is to report changes at the level of the API (i.e. a change in the signature of a routine) but not to report changes on the lower-level implementations of routines. Assuming we have a programming language that allows to specify an API not only through method signatures but through contracts—examples are Eiffel, JML for Java, Code Contractors for C#—the question arises if these more expressive API specifications could be utilized for reporting changes in an awareness tool. This thesis provides an answer to these questions by analyzing over 260 million lines of code, originating from 21 projects that use contracts. Our findings provide clear evidence that developers do use contracts as a way to specify program behavior, but also that changes to contracts occur much less frequently than changes to implementations. Our results are a strong indicator that awareness tools, which work on languages that support contracts, can benefit from monitoring the very same.

Collaborative debugging. The final contribution of this thesis takes a fresh look at how distributed teams can collaboratively debug software.
In many situations developers simply debug code on their own. Sometimes, however, debugging code together with other developers is more desirable. Such a collaborative debugging is easily done in a collocated setting—developers sit in front of the same screen—but becomes a technical challenge if the developers are distributed. Today’s common solution is to use a shared-screen approach, in which one developer shares a video of her screen with the others. We’ve developed a new approach to this problem in which developers share a common debugging session but remain in control over their own development environment. We performed a study with 38 users and found that developers prefer this new approach of collaborative debugging and identified which features of our debugger were considered particularly useful.

Summary of Contributions. The contributions presented in this thesis have been initially published at conferences and in journals. The following list provides a comprehensive overview of the contributions and their associated publications:

1. Communication in distributed teams: an empirical study on the impact of time zones and team distributions in GSE projects. A first version of this study was published in ICGSE’11 [75]. The study presented in this thesis builds upon that work but extends it with a much larger data set and different analyses.

2. The role of development processes in GSE: an empirical study to identify if and how the type of development process impacts distributed projects [36, 37].

3. Conflicts and awareness: an empirical study investigating the frequency and impact of merge conflicts and insufficient awareness about other team members’ code changes in distributed projects [35].
[Estler H.-C., Nordio M., Furia, C. A., Meyer B, Awareness and Merge Conflicts in Distributed Software Development, ICGSE’14]
1.3 How to Read this Thesis

The chapters two to seven contain the main contributions of this thesis. The first three of those chapters provide insights on how distributed projects are carried out and thus provide motivations for the tool- and solution-focused work presented in the later chapters. Nevertheless, each chapter is self-contained and can be read independently of others.

Whenever appropriate, we highlight the main findings in result boxes. These result boxes are high-level summaries and the reader should interpret their content within the context of the current experiment or case study.

Finally, we use a number of abbreviations throughout this thesis. The terms distributed software development, distributed software engineering and global software engineering are widely used in literature and we’ll abbreviate them with DSD, DSE, and GSE.

1.4 Outline

The rest of this thesis is structured as follows: chapter 2 describes our study on the effect of timezones and distribution to a team’s communication behavior. Chapter 3 describes the study comparing agile and structured processes in distributed projects. In chapter 4 we present our work on understanding how often developers encounter merge-conflicts and a lack of awareness when
working in a DSD setting, followed by a detailed explanation of CloudStudio and its approach to awareness in chapter 5. Chapter 6 summarizes the work on how contracts are used in practice and why they present a good proxy for detecting and reporting important program changes. Finally, we present our work on a new approach for collaborative debugging in chapter 7 and close with a conclusion and directions for future work in chapter 8.
Chapter 2

Communication in Distributed Teams

2.1 Introduction

Most of today’s software is not created by single individuals but by teams of developers. These developers need to coordinate their work to ensure that individual contributions integrate with each other, ultimately producing a single software product. Effective communication therefore plays a critical role in the success of software development. Introducing geographical distance into a team impedes effective communication [15]. The reasons for this impediment can be manifold, ranging from a lack of trust, to difficulties of interacting due to time zones differences, to misunderstandings caused by different languages or cultural backgrounds amongst team members.

To evaluate the effect of distribution on software development, studies [10, 52, 53, 30] have been performed. For example, Bird et al. [10] have studied the development of Windows Vista, and compared the failures of the components developed distributed with the failures of the components developed locally. Other studies [53, 30] focus on how time zones affect software development.

In this chapter, we are interested in the question of how distribution and time zones affect the communication amongst distributed teams, especially when the teams are located in different countries and continents.

We present an analysis of distributed software development projects, which were performed during three iterations of a university course. In this course, “Distributed and Outsourced Software Engineering” (DOSE), students develop software in a distributed setting, collaborating with teams in Europe, Asia, and South America. Our study is based on the editions of DOSE in the years 2010, 2011, and 2012. In DOSE 2010, the projects were implemented
jointly with eleven universities (located in ten different countries); in DOSE 2011 eleven universities participated (located in nine different countries), and in DOSE 2012 twelve universities took part (located in ten different countries).

Our study analyzes the whole development process: from scope definition, to requirements, to interface specification, to implementation, to testing. In total, we collected data from 112 teams, where each team had between two and five students.

In our study we focus on the communication of distributed projects, analyzing the time spend on communication compared to the total time spend on executing a software project. We compare the communication of teams distributed over two countries with the communication of teams distributed over three countries. We also investigate how communication behavior is affected by time zone differences.

Our results show that almost one quarter of the overall project time is contributed to communication between the distributed team members. On average, teams working in two locations expended about 20% of the project time on communication, while teams working in three locations expended about 23%. We also compare the communication in projects with big time zone differences (more that four hours apart), medium and short time zone difference (two to four hours apart and less than two hours apart, respectively) and observe a trend towards less communication in bigger time zones.

2.2 Research Questions

Communication in distributed projects is difficult. A lack of face to face meetings and synchronous communication can make communication more tedious, can produce misunderstandings in the teams, and result in delays of the project. Conventional wisdom has it that higher degrees of distribution lead to more challenges in communication. Similarly, it’s believed that distributed development is less risky if differences in time zones are kept small, a strategy sometimes referred to as nearshoring in the context of outsourced software development. We analyze these beliefs in our empirical study where projects are developed at different level of distribution and over different ranges of time zones. Our research questions are:

RQ1: Is the amount of communication in three-location projects higher than in two-location projects?

RQ2: Do projects distributed in farther time zones have more communication than projects distributed in closer time zones?
2.3. CONTEXT OF THIS STUDY

RQ3: Is the average reply time for e-mails of projects in farther time zones higher than the average reply time for e-mails in closer projects?

2.3 Context of this Study

2.3.1 The DOSE Course

The study was developed during the Fall semesters in 2010, 2011, and 2012 in the “Distributed and Outsourced Software Engineering” (DOSE) course [77, 76]. The DOSE course targets master students with good experience in programming and some prior knowledge in software engineering. Since 2007, the course incorporated the development of a project with a distributed team. In DOSE 2010 [104], the projects were done in collaboration with the following eleven universities:

University of Debrecen, Hungary; University of Delhi, India; ETH Zurich, Switzerland; Hanoi University of Science and Technology, Vietnam; Korea Advanced Institute of Science and Technology, Korea; Politecnico di Milano, Italy; State University of Nizhny Novgorod, Russia; Odessa Polytechnic National University, Ukraine; University of Rio Cuarto, Argentina; Wuhan University, China; University of Zurich, Switzerland.

In DOSE 2011 [105], the projects were again done in collaboration with eleven universities:

IT University of Copenhagen, Denmark; University of Debrecen, Hungary; ETH Zurich, Switzerland; Hanoi University of Science and Technology, Vietnam; ITMO University, Russia; Universidad Politecnica de Madrid, Spain; Politecnico di Milano, Italy; State University of Nizhny Novgorod, Russia; Odessa Polytechnic National University, Ukraine; University of Rio Cuarto, Argentina; University of Zurich, Switzerland.

In DOSE 2012 [106], the following twelve universities collaboratively taught the course:

Cairo University, Egypt; University of Crete, Greece; University of Debrecen, Hungary; ETH Zurich, Switzerland; ITMO University, Russia; Universidad Politecnica de Madrid, Spain; Politecnico di Milano, Italy; State University of Nizhny Novgorod, Russia; Odessa Polytechnic National University, Ukraine; Pontificia
Universidade Catolica do Rio Grande do Sul, Brazil; University of Rio Cuarto, Argentina; University of Zurich, Switzerland.

The number of students in each year and their geographic distribution is shown in Figure 2.1.

![Figure 2.1: Geographic distribution and number of students participating in DOSE 2010, 2011, and 2012.](image)

### 2.3.2 Team and Group Structure

The projects in the DOSE course were organized around the notion of *groups*, where a group consists of two or three teams, each team located at a different university (and therefore, most likely, in a different country). By the end of the project a group had to deliver a software product that integrates the components developed by its teams. A common breakdown of a group’s project into components for three teams comprised a *database* component, a *logic* component, and a *user interface* component.

Each team wrote its own scope and requirements documents for their component, and each designed and implemented their component based on these documents. Collaboration and coordination was, of course, necessary throughout all phases to ensure that a group can integrate each team’s component.

We strongly encouraged all groups to make consistent use of collaboration tools such as configuration management tools, issue trackers, wiki pages, or google docs. Furthermore, students used the same development environment
with the same projects settings that we project at the start of the project. This minimized collaboration problems due to different project setups.

The organization in groups and teams, as well as the project’s component structure were decided at the beginning of the course. We think that this project organization contributed to reduce communication overhead, and to get achieve outcomes across different projects.

2.3.3 Project Organization

In DOSE 2010, the course projects comprised the development of language learning tools, which help users in learning a foreign language. Each group developed a language learning application in the form of a vocabulary trainer. For each group, one team implemented the logic component of the application, one team the graphical user interface, and one team the database component.

In DOSE 2011 and 2012, the course projects comprised the development of card and board games. Each group developed a game that was pre-approved by the teaching staff. The approval ensured that each group’s project would be comparable in scope and workload. Projects were split into a logic & network component, a graphical user interface component, and (for group’s with three teams) an artificial intelligence component.

The projects in 2010, 2011 and 2012 were organized in the following five phases:

- Phase 1: Scope document (∼ 2 weeks)
- Phase 2: Requirements document (∼ 2 weeks)
- Phase 3: API - Interface specification (∼ 2 weeks)
- Phase 4: Implementation (∼ 5 weeks)
- Phase 5: Testing (∼ 1 week)

In the first week of the course we provided project descriptions, outlining the general structure and componentization of the projects. In Phase 1, the students then developed their own scope documents, with each team defining the scope of their component and the role each student would have in the team. The scope documents helped students to avoid misunderstandings regarding the scope of their applications. In the requirements phase, each team wrote a complete requirements specification document, focusing on the description of the functional requirements. The interface specifications were written in Eiffel using Design-by-Contract. The last two phases covered the
implementation and testing of the projects where the implementation was built on top of the interface specifications developed in phase 3.

2.3.4 Project Outcomes
In all three years of the DOSE course, all groups fully implemented their projects. The final outcomes were inspected and graded by the teaching staff. In all cases the results of the projects were found to satisfactory or exceeded the expectations.

2.4 Data Collection and Classification

2.4.1 Data Collection
The data used in this study has been collected during the three editions of the DOSE course in 2010, 2011, and 2012. Each team reported at the end of each phase (scope, requirements, interface specification, implementation, testing) how much time they (as a team) spent working on their project, how much time they used for communication, and what the average reply time for e-mails to other teams was. Each team collected this data independently of other teams. Furthermore, teams did not have access to the data of other teams, so it is unlikely that answers were influenced by the replies of other teams.

To enable the data collection in a distributed setting, we used online questionnaires. Students were explicitly instructed at the beginning of each phase to collect the data they’d have to report at the end of phase. If a team did not report their numbers at the end of a phase we send them a reminder email asking them to fill in the questionnaire.

All collected data is available for download [25].

2.4.2 Data Classification
We classify projects by two criteria: first, by the level of geographical distribution, measured as the number of countries working together in a group. Second, by time zone difference, measured as the largest distance of time zones between the countries of a group.

Geographical Distribution
The classification by number of countries per group results in the following:
2.4. DATA COLLECTION AND CLASSIFICATION

- **Two locations**: Groups geographically distributed over two countries, with one team per country. During the project each team worked on different components of the application. We have collected information from 35 teams (103 students). Typical group configurations were:
  
  - Switzerland - Ukraine
  - Italy - Hungary
  - Switzerland - Vietnam

- **Three locations**: Groups geographically distributed over three countries, with one team per country. During the project each team worked on different components of the application. We have collected information from 77 teams (239 students). Typical group configurations were:
  
  - Argentina - Switzerland - Vietnam
  - India - Switzerland - Hungary
  - Korea - Italy - Switzerland

**Time Zone Difference**

The range of time zone differences in our data set stretches between zero and twelve hours. For our classification, we simplify this setup and only distinguish three different types of time zones differences: small, medium, and large. A small time zone difference covers 0-2 hours of difference, a medium one covers 2 to 4 hours, and a large one covers more than 4 hours. This classification is based on the idea of how many hours of a working day—from 9am to 5pm—overlap between teams: within a small time zone difference the overlap for teams is large, which might simplify communication and coordination amongst teams; for a medium time zone difference, the overlap only covers between 50% to 75%, whereas for a large time zone difference the overlap is less than 50%. The time zone ranges used in this study are:

- **Small**: Projects distributed over time zones with two or less hours apart. Examples of such group configurations are:
  
  - Switzerland - Italy
  - Denmark - Hungary - Spain

  For this category we collected data from 28 teams.
• **Medium:** Projects distributed over time zones more than two hours and up to four hours apart. Examples for such group configurations are:

  - Italy - Switzerland - Russia
  - Argentina - Switzerland

  In this time zone range, we collected data from 40 teams.

• **Large:** Projects distributed over time zones more than four hours apart; this include teams in South America, Europe, and Asia. Examples for such group configurations are:

  - Argentina - Italy - China
  - Argentina - Switzerland - Vietnam

  For large time zone ranges, we collected data from 44 teams.

### 2.5 Results

#### 2.5.1 Data Transformation

Before performing our analysis, we prepared the raw data in two ways. First, we normalized the absolute numbers reported by each team to account for the number of students in the team. In particular, for values such as "hours spent on communication" or "hours spent developing", the reported values are sensitive to the number of students in the team. Over the course of the three DOSE courses, teams had different sizes, ranging from two to five students. The normalization accounts for these differences. It is noteworthy, however, that the vast majority of all data points, 69%, originated from teams with three members; teams with two or four members each contributed 14% of the data points; and only 3% of the data points originated from teams with five members.

Second, we calculated relative values for the absolute numbers reported by students. We use the ratio of the time expended on communication due to distribution over the total time expended in the project. The use of the ratio allowed us to account for small variations in the lengths of phases in different years. Furthermore, it allowed us to account for the different motivations and different levels of prior knowledge that students had. Some groups invested and engaged more in the projects than others and thus spend, in absolute hours, more time on all aspects of the project. We found, in fact, a
moderate to strong correlation (Pearson $r = 0.64$, $p \approx 0$) between how much time students spent on communication and how much time they spent on other tasks such as writing the requirements documents or implementing the system. For these reasons, we decided to use the ratio in the analysis for $RQ1$ and $RQ2$ to allow for a fair comparison of the different data points. Using the ratio was not necessary for $RQ3$ because the reported value "average reply time (hrs) to emails" is independent of the influential factors mentioned above.

2.5.2 Communication in two and three-location projects

To answer $RQ1$ we analyzed the ratio of the time expended on communication due to distribution over the total time expended in the project for teams in two and three locations, respectively.

**Analysis.** The analysis was preceded by the removal of outliers. For the two samples "2-locations" and "3-locations" we calculated the 5th and 95th percentile. Values below or above these percentiles were removed in the subsequent analysis. This systematic removal of outliers was based on our experience that in every year and every phase of the projects there are teams that struggle with collaboration and communication due to atypical circumstances. Such problems were usually resolved through intervention by the teaching staff in a timely manner. Nevertheless, such incidents can distort the data in both extremes (very low amount of communication or very high amount of communication) which is why we removed such outliers.

![Histogram for 2−locations](image)

![Histogram for 3−locations](image)

Figure 2.2: Histograms and density functions of the average ratio of communication between teams distributed in two and three locations. The red line is the density function, the dashed blue line is the normal distribution with the same mean and standard deviation.
Subsequently, we checked if the data satisfies the assumptions for Welch’s t-test. Figure 2.2 shows the density functions for the 2-locations and 3-locations samples and compares them to normal distributions with the same mean and standard deviation (dashed blue line). Similarly, we checked normal quantile plots and concluded that our two samples approximate normal distributions. Because we used Welch’s t-test we did not have the requirement of equal variances for our samples, and it was unproblematic that the samples are of different size. A final requirement for the t-test is the interdependence of observations. In the context of DOSE we cannot guarantee full independence because on team’s communication might trigger communication for another team (e.g. a reply to an email). Likewise, a lack of communication in phase $n$ might result in a higher amount of communication in phase $n + 1$. We believe, however, that the potential effects of such dependencies are not a dominating factor and thus decided to perform the analysis using a t-test.

**Hypothesis.** The hypotheses used in the analysis of $RQ1$ consisted of the null hypothesis $H_1$ and its alternative hypothesis $\overline{H}_1$. Let $\bar{c}_i$ denote the expected mean communication ratio in projects distributed in $i$ locations. For each phase, and for the overall span of the projects (covering all phases), we define the following hypotheses:

$$H_1 \triangleq \bar{c}_{2\text{-location}} = \bar{c}_{3\text{-location}}$$

$$\overline{H}_1 \triangleq \bar{c}_{2\text{-location}} \neq \bar{c}_{3\text{-location}}$$

**Results.** Table 2.1 shows the basic statistics for 2-locations and 3-locations projects. Ignoring phases and comparing all data (column All), we found about 4% more communication in 3-locations projects. This difference is highly statistically significant, but only of small to medium effect (we measured effect size using Cohen’s $d$).

From the five phases we only observe significant differences in the API design and Implementation phases. Here, we found the effect of two vs. three locations on communication to be medium to large ($0.5 < d < 0.8$).

Projects distributed in three locations have, overall, significantly more communication across teams than projects distributed in two locations. The effect is small to medium in size but shows clearly in the project phases of API design and Implementation.

**Discussion.** The finding that projects in three locations required more communication than those in in two locations matches our intuition that
Table 2.1: Comparison of ratios of communication time $c$ for projects with teams in two locations (2-loc.) and teams in three locations (3-loc.). $n$ is the number of observations for each phase (columns: scope, requirements, api design, implementation, test) and the overall number of observations over all phases (all). $\bar{c}$ is the median in %; $\bar{c}$ is the mean in %; $\sigma$ is the standard deviation in %. Row t-test gives the results of Welch’s t-test comparing 2-loc. and 3-loc., differences that are significant at the $\alpha = 0.05$ level are marked using (*); $d$ is the effect size calculated using the measure Cohen’s d.

more distribution requires more coordination, which is carried out through communication. Interestingly, the findings showed us that—at least for our student projects—the difference in amount of communication is strongest during the phases that are vital to a successful completion of a project: interface specification and implementation. The fact that the mean percentage of communication in these two phases is lower than in the other three phases may be contributed to a proportionality larger amount of time invested into designing and writing code.

### 2.5.3 Communication in projects in different time zones

RQ2 required an analysis similar to the one for RQ1 but needs to account for three rather than two samples: groups distributed over small time zone ranges, groups distributed over medium time zone ranges, and groups distributed over large time zone ranges. As mentioned above, a group qualified as small if its teams were distributed over two or less hours apart. A time zone range of more than two and up to four hours qualified a group as medium, groups with a time zone range of more than four hours qualified as large.
Analysis. As a first step of the analysis we, again, removed outliers based on the 5\(^{th}\) and 95\(^{th}\) percentile in the small, medium, and large samples. Our motivation for this removal was the same as in the analysis of RQ1.

In order to compare the three samples we’re using a one-way ANOVA test. The ANOVA test assumes, just like a t-test, that the dependent variable (communication) approximates a normal distribution for each category of the independent variable (time zone). We check this assumption using the density plots shown in Figure 2.3 and normal quantile plots. While not fully matching the theoretical normal distribution, we found the approximation sufficient to justify the use of ANOVA.

![Histograms and density functions of the average ratio of communication between teams distributed in small, medium, and large timezones.](image)

Figure 2.3: Histograms and density functions of the average ratio of communication between teams distributed in small, medium, and large timezones. The red line is the density function, the dotted blue line is the normal distribution with the same mean and standard deviation.

Other assumptions of the one-way ANOVA test are homogeneity of variances and independence of observations. Our samples didn’t all have the same variance but the difference was less than a factor of 2.5. Such a deviation is within the limits that justify the use of ANOVA. For independence of observations we faced the same situation as for RQ1, i.e. we can’t guarantee
independence but believe its effect to be minor.

**Hypothesis.** The hypotheses used in this analysis consisted of the null hypothesis \( H_2 \) and its alternative hypothesis \( \overline{H}_2 \). Let \( \bar{c}_i \) denote the expected mean communication ratio in projects with time zone range \( i \), where \( i \in \{ \text{small}, \text{medium}, \text{large} \} \). For each phase, and for the overall span of the projects (covering all phases), we define the following hypotheses:

\[
H_2 \triangleq \bar{c}_\text{small} = \bar{c}_\text{medium} = \bar{c}_\text{large}
\]

\[
\overline{H}_2 \triangleq \bar{c}_\text{small} \neq \bar{c}_\text{medium} \neq \bar{c}_\text{large}
\]

**Results.** The results of the analysis are shown in Table 2.2. We found that, overall, there’s a significant difference in the mean communication ratio. Looking at individual phases, however, we only found a significant difference in the first phase, the development of the scope document. The effect is large of medium size for the scope phase but only small in the overall comparison.

For the two cases where the ANOVA test led us to reject the null hypotheses, we ran a pairwise t-test to determine which pairs of time zone ranges differ significantly. For the overall analysis we found that the difference between small & medium time zones is significant \( (p < 0.001) \), the difference between medium & large time zones is not significant \( (p = 0.573) \), and the difference between small & large time zones is again significant \( (p < 0.026) \).

We obtained similar \( p \) values for the pairwise comparison in the scope phase: small & medium \( (p < 0.002) \), medium & large \( (p = 0.626) \), and small & large \( (p < 0.029) \).

Projects distributed over small time zone ranges have significantly more communication than projects distributed in medium or large time zones. There’s no significant difference between medium or large time zones. Overall, the effect of time zones on the communication ratio between teams in a group is small.

**Discussion.** An important observation from our data is that time zones ranges do have an effect, but only a small one. In particular, the effect is limited to having a group distributed over a small time zones or not: after a certain threshold—in our case 2 hours—it doesn’t seem no to matter much how wide the time zone range is.

We motivated our split of the time zone ranges ("less than two", "two to four", and "more than four") in Section 2.4. To validate our motivation we ran an analysis which different splits. We found that medium and large time zones don’t show any significant difference, i.e. our results are robust with
Table 2.2: Comparison of ratios of communication time \( c \) for projects developed in small, medium, and large time zone ranges. \( n \) is the number of observations for each phase (columns: scope, requirements, API design, implementation, test) and the overall number of observations over all phases (all). \( \bar{c} \) is the mean in %; \( \tilde{c} \) is the median in %; \( \sigma \) is the standard deviation in %; row ANOVA gives the results of a one-way ANOVA test, differences that are significant at the \( \alpha = 0.05 \) level are marked using (*); \( \eta^2 \) is the effect size calculated using eta square.

<table>
<thead>
<tr>
<th># observ.</th>
<th>Scope</th>
<th>Req.</th>
<th>API</th>
<th>Impl.</th>
<th>Test</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>small</td>
<td># observ. ( n )</td>
<td>26</td>
<td>26</td>
<td>26</td>
<td>19</td>
<td>123</td>
</tr>
<tr>
<td></td>
<td>median ( \tilde{c} )</td>
<td>30.4</td>
<td>22.6</td>
<td>25</td>
<td>19.4</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>mean ( \bar{c} )</td>
<td>33.4</td>
<td>22.5</td>
<td>26.5</td>
<td>18.7</td>
<td>24.3</td>
</tr>
<tr>
<td></td>
<td>std. dev. ( \sigma )</td>
<td>14</td>
<td>10.3</td>
<td>15.1</td>
<td>12.4</td>
<td>8.6</td>
</tr>
<tr>
<td>medium</td>
<td># observ. ( n )</td>
<td>34</td>
<td>37</td>
<td>38</td>
<td>32</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>median ( \tilde{c} )</td>
<td>24.2</td>
<td>19</td>
<td>22.8</td>
<td>19.2</td>
<td>16.7</td>
</tr>
<tr>
<td></td>
<td>mean ( \bar{c} )</td>
<td>22.8</td>
<td>19.1</td>
<td>21.9</td>
<td>20.2</td>
<td>18.7</td>
</tr>
<tr>
<td></td>
<td>std. dev. ( \sigma )</td>
<td>9.2</td>
<td>7.5</td>
<td>8.9</td>
<td>7.5</td>
<td>8.5</td>
</tr>
<tr>
<td>large</td>
<td># observ. ( n )</td>
<td>40</td>
<td>42</td>
<td>40</td>
<td>40</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>median ( \tilde{c} )</td>
<td>24.2</td>
<td>17</td>
<td>20.7</td>
<td>16.3</td>
<td>21.3</td>
</tr>
<tr>
<td></td>
<td>mean ( \bar{c} )</td>
<td>26.1</td>
<td>19.5</td>
<td>22.1</td>
<td>20.3</td>
<td>21.8</td>
</tr>
<tr>
<td></td>
<td>std. dev. ( \sigma )</td>
<td>10.4</td>
<td>9.3</td>
<td>10.6</td>
<td>10.7</td>
<td>8.2</td>
</tr>
</tbody>
</table>

ANOVA

- \( F \) = 6.98
- \( k - 1; n - k \) = 2.97; 2.102; 2.101
- \( p \) = 0.002; 0.295; 0.23
- Effect size \( \eta^2 \) = 0.13

The observed difference of more communication in small time zone ranges, however, vanished if we stretch the small difference beyond more than three hours difference. At that point, we couldn’t find any significant difference between time zones anymore.

A possible explanation for this observation might be that there’s a threshold after which time zone ranges, independent of their size, hamper and, thus, lead to less communication. Based on our data, this threshold is around two to three hours of time zone difference.

### 2.5.4 Email reply time in different time zones

The analysis of our final research question, \( RQ3 \), followed the same setup as the analysis of \( RQ2 \). We differentiate between small, medium, and large time...
zone ranges and analyzed how the average reply time in email communication differed. In contrast to our other analyses, this analysis did not require a normalization of the data or the use of the ratio. Instead, we directly used the "average number of hours for email replies" as provided by all teams.

**Analysis.** The analysis was once more preceded by the removal of outliers from all three samples using the 5th and 95th percentile. Upon inspecting the three sample distributions we found that none of them approximated a normal distribution and all of them were skewed to the right. For this reason we didn’t use an ANOVA test but a non-parametric Kruskal-Wallis test. The density function and histograms of the three samples are shown in Figure 2.4. The Kruskal-Wallis test assumes independence of observations. As argued in the previous analyses we can’t guarantee independence but believe that the impact of violations would be small. After manually inspecting the density functions of the three samples, we concluded that all three samples might originate from the same theoretical distribution but only be shifted in location. With this assumptions, we formulated the hypotheses for RQ3.

**Hypothesis.** The hypotheses used in the analysis of RQ3 consisted of the null hypothesis $H_3$ and its alternative hypothesis $\overline{H}_3$. Let $\tilde{t}_i$ denote the expected median reply time to emails in projects with time zone range $i$, where $i \in \{\text{small}, \text{medium}, \text{large}\}$. For each phase, and for the overall span of the projects (covering all phases), we define the following hypotheses:

\[
H_3 \triangleq \tilde{t}\text{small} = \tilde{t}\text{medium} = \tilde{t}\text{large}
\]

\[
\overline{H}_3 \triangleq \tilde{t}\text{small} \neq \tilde{t}\text{medium} \neq \tilde{t}\text{large}
\]

**Results.** The results of the analysis are shown in Table 2.3. Given that the Kruskall-Wallis test showed significant differences, we extended our analysis to pairwise comparisons. Using pairwise Wilcoxon tests for email reply times over all phases, we found that the difference between small & medium time zones is significant ($p < 0.001$), the difference between medium & large time zones is significant ($p < 0.007$), and the difference between small & large time zones is significant ($p < 0.001$) as well.

Comparing the pairs for individual phases we found that for the three phases scope, requirements, and API, the reply time in small is always significantly shorter than the reply time in large at a significance level of $\alpha = 0.05$. Other significant differences exist in phase requirements for the pairs small & medium ($p < 0.04$) and the pair medium and large ($p < 0.05$). In the API phase, a significant difference also exists for the pair small & medium ($p < 0.04$).
Figure 2.4: Histograms and density functions of the average ratio of communication between teams distributed in small, medium, and large timezones. The red line is the density function, the dotted blue line is the normal distribution with the same mean and standard deviation.

Discussion. The findings for email reply time largely match our intuition: if team members work in a time zone further away, it might take a longer time before they respond to an email. A surprising observation that might warrant additional investigation is the increase of about 100% in the average reply time for small in the implementation phase. A possible explanation for this behavior might be that teams are very busy and stressed in this phase, trying to finish the project on time and therefore become less responsive to email communication.
2.6 Students’ Feedback

At the end of DOSE 2010, we asked the students to fill in a questionnaire. The goal of the questionnaire was to get the feedback of the students and their opinion of the effect of time zones in distributed software development. Students also reported whether cultural differences affected the development. The questionnaire was filled in by 87% of the students.

Table 2.4 summarizes the result of the questionnaire. On average, students’ feedback was that time zones, as well as cultural differences, had little effect on quality or productivity, and have not caused a lot of communication overhead. The answer options ranged from 1 to 5, where 1 represents not at all, and 5 very much. We classified the feedback by time zone ranges. The results follow an order where the small time zone range has lower values, and the large time zone range has higher values. The values for small are likely lower because of the shorter time zone difference and cultural similarity for
Table 2.4: Students’ feedback for the effect of time zones and cultural differences in Doing 2010. Values range from 1 to 5: 1 not at all; 5 very much.

<table>
<thead>
<tr>
<th>Large</th>
<th>Medium</th>
<th>Small</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>3.7</td>
<td>4.0</td>
<td>4.0</td>
</tr>
<tr>
<td>3.8</td>
<td>3.3</td>
<td>4.1</td>
<td>4.0</td>
</tr>
<tr>
<td>3.6</td>
<td>3.1</td>
<td>3.7</td>
<td>3.7</td>
</tr>
<tr>
<td>4.0</td>
<td>3.7</td>
<td>4.1</td>
<td>4.1</td>
</tr>
</tbody>
</table>

Local projects: the communication overhead would be lower.
Local projects: the productivity would be higher.
Local projects: the quality would be better.
Local projects: the development would be easier.

Cultural differences caused communication overhead.
Cultural differences affected productivity.
Cultural differences affected quality.

Time zones caused communication overhead.
Time zones affected productivity.
Time zones affected quality.
groups with teams spread over Europe.

We also asked the students how they think the project development would be affected, if the projects were developed by the same groups but all teams working in the same location. The last four entries in Table 2.4 shows the result of these questions. There is no significant difference in the reply of the different time zones. The result indicates that the students assume the development would be easier, the quality better, the productivity higher, and the overhead of communication would be lower.

The feedback also included a question about the students preference to set up a new software projects: 35% reported that they would prefer a local project, 5% would prefer a distributed project, and 60% reported that both types of setup would be fine for them.

2.7 Threats to Validity

There exist a number of threats to the validity of our study. In this section we discuss the major ones in the area of construct, internal, and external validity.

Construct Validity. The data used in our study was reported by students at the end of each phase by filling in an online questionnaire. We instructed students at the beginning of each phase to keep track of how much time they expand while working on their projects but have no means to verify the correctness of the numbers reported.

Internal Validity. The students participating in the DOSE courses have different educational backgrounds, experiences and motivations. Different histories are a threat to the internal validity of our study as we’ve observed that teams with very good students are usually less affected by distribution than average teams. This threat is, however, counteracted by the large set of data points in our study, which should provide a fair distribution of different histories.

It’s very challenging to account for confounding factors in our study. For example, groups that stretch over large time zone ranges will, by their very nature, have a more diverse cultural background. And different cultural backgrounds may influence the communication behavior. Similarly, a project that’s distributed over three rather than two locations will most likely involve more cultural diversity and languages. The findings from Table 2.4 suggest that the impact of such confounding factors may be limited. Nevertheless, they present a threat to our independent variables being the main cause of the observed differences in the dependent variables.

Our study investigates teams’ communication over the entire life cycle of
a project. The vast majority of students working on the projects don’t have prior experience with DSD. Over the course of the project, they gain experience and develop strategies to improve their coordination and collaboration, i.e. the participation in the study may, over time, change what students report. This threat, while likely to be present in our study, is a phenomena that’s not limited to our study setup but will also occur in any project that distributes its team members over different countries.

**External Validity.** The main external validity threat of our study is generalizability. After all, the scope of the DOSE projects is limited to one semester and students’ experience, motivation, and the processes they use to develop software are likely to differ from those of development teams working in industry. However, as our findings are in line with the results of other researchers, we assume that the DOSE course does, ultimately, not differ too drastically from a small to medium size industry project.

### 2.8 Related Work

The challenges and effects of distributed development have been subject to research for more than two decades, for example analyzing how distributed development influences the quality of software products and the productivity in the development process. Bird et al. [10] present a case study on the development of Windows Vista, comparing the failures of the components developed distributed with the failures of the components developed locally. They have found no difference in the failures. In their study, a considerable number of developers were located in the same campus (in the same city). Our study was done using a totally distributed project, and focuses only on the communication aspects of the distribution. As future work, we plan to extend the study to also analyze the failures produced in these projects.

Ramasubbu et al. [92] also study the influence of distribution on quality. They develop a model of software development, and then use the model to understand the consequences of distribution on performance. In contrast to Bird et al.’s work [10], they found that distribution significantly reduces productivity and affects quality.

A study by Spinellis [97] examined how distributed development affects defect density, code style and productivity in the open source project FreeBSD. The analysis revealed that there is almost no correlation between geographic distribution and defect density or productivity.

In [75] we performed the same study shown in this chapter but only had data available from the DOSE course in the years 2009 and 2010. Our findings then were not conclusive as our analysis didn’t lead to any significant results.
The study shown here builds upon this work and confirms our expeditions described (but not confirmed) in [75] with the exception that we now have evidence for more communication in groups distributed in three locations than in two locations. The data from the original paper didn’t indicate this outcome.

Similar to our study, other researchers have investigated the effect of distribution on communication. Allen [1] reported that the frequency of communication amongst engineers whose offices are more than 30 meters apart dropped to almost the same level of those by engineers with offices separated by several miles. Comparable results were reported by Kraut et al. [61].

Carmel [14] identifies loss of communication as one of four major risk factors which can lead to the failure of global software projects. Carmel and Agarwal [15] proposed to reduce intensive collaboration, temporal distance and national and organizational distance to mitigate the problems of distributed development. An experience report by Battin et al. [5] about a large-scale distributed project at Motorola also emphasis the importance of establishing good relationships amongst the distributed teams.

A study by Herbsleb and Mockus [52] analyzed the effects of globally distributed development with regard to resolving modification requests and differences in communication. They found that on average it takes 2.5 times as long to complete distributed work items. When accounting for factors like number of people working on a modification request or size of change, however, the differences were no longer significant. The analysis of distributed communication revealed that developers communicate much more frequent with three co-located colleagues than with their remote colleagues. Also, the size of the social network (number of colleagues a developer interacts with) were significantly smaller in the distributed case. Similar to Herbsleb and Mockus’ study, we have observed that the communication in two-location projects was higher than the communication in three-location projects. The same result applies to the time zone ranges. As future work, we plan to compare the distributed projects with projects developed locally.

Nguyen et al. [74] report on an empirical study of IBM’s Jazz project which was developed globally distributed at five different sites. The study examined the effects of communication delay as well as task completion time. They found that distance does not have a strong effect on either one. It is noteworthy, however, that the analyzed communication was restricted to comments on work items and did not include other media such as e-mail, chat or voice. Our study was performed in an academic environment, however, it analyzes the amount of communication, especially e-mail, chat and voice, in the whole development process (from requirements to testing) for projects
with different distributions.

To analyze the effects of time zone ranges, Espinosa et al. [30] conducted a study using several identical projects. They performed a laboratory experiment in which groups of two subjects had to collaboratively solve a task; they collected data by surveying students. The task to solve was, however, comparably small (at most one hour work time was given) and not a software task. Time zone ranges were mimicked by introducing work time overlaps (zero, one-third, two-third, full) between the two subjects. Espinosa et al. studied how the time zone ranges affected production speed and production quality only. While an experiment in such a controlled environment reduces the threats to validity, it is questionable if it could be used to study the effects of time zone ranges on communication as they are experienced in software projects ranging over several month.

Another controlled experiment was performed by van Soligen et al. [108] to study the impact of the number of sites on the overall working speed. The experiment consisted of two to four sites. They have reported that when the number of sites increases, the overall working speed of sites increases. Deshpande et al. [23] present a study, based on interviews, on the effect of cultural differences in distributed projects.

2.9 Summary

In this chapter, we’ve presented a study on communication behavior in distributed projects. The study showed that a substantial amount, almost 25% of overall time spend on a project, is attributed to communication between distributed team members. We observed significantly higher ratios of communication in projects distributed over three locations than in projects distributed over two locations, and this difference is particularly present in the phases of API design and implementation.

Our findings also showed that teams distributed over small time zone ranges communicate more. At the same time we found the overall effect of time zones on the ratio of communication to be limited.

Finally, we analyzed the reply time for e-mails of teams located in different time zone ranges. We found that average reply times for emails increase with larger time zone ranges.
CHAPTER 3
THE ROLE OF DEVELOPMENT PROCESSES IN GSE

3.1 Introduction and Overview

The importance of choosing the right development process to ensure the successful and timely completion of distributed software projects cannot be understated... Or can it? This chapter presents an extensive case study analyzing the impact of different development processes on the success of software projects carried out by globally distributed development teams.

3.1.1 Empirical Analyses of Globally Distributed Software Development

Globally distributed software development has become a common practice in today’s software industry; companies cross the barriers introduced by distance, cultural differences, and time zones, looking for the most skilled personnel and the most cost-effective solutions. Globally distributed software development may exacerbate several of the criticalities already present in traditional local software development, and it often generates its own peculiar challenges originating in the difficulty of carrying out the traditional parts of a software development project—requirements elicitation, API design, project management, team communication, etc.—in environments where members of the same team live and work in different countries, or even in different continents.

Given the challenges and peculiarities introduced by globally distributed software development, it is interesting to peruse the standard methods and practices that have been successful in traditional local software development,
determining if they can be applied with positive results also in globally distributed settings. From the perspective of empirical research in software engineering, this general line of inquiry materializes in questions of the form “What is the impact of X on the quality of globally distributed software development projects”, where “X” is a practice, method, or technique, and “quality” may refer to different aspects such as timeliness, customer satisfaction, cost effectiveness, or the absence of problems. Examples of globally distributed software development issues investigated empirically along these lines include the usage of contracts for API design [78], the effect of time zones on various phases of development [53, 30, 75] and on productivity and quality [92, 10], and the impact of geographic dispersion on several quality metrics [93].

3.1.2 Goals of This Study: Impact of Development Processes

The case study presented in this chapter focuses on development processes [44, 91, 86] to find out whether the choice of process has a significant impact on qualities such as programmer productivity and development cost-effectiveness in globally distributed software development. To our knowledge, this is one of very few empirical studies that explicitly investigates the impact of development processes on globally distributed software development.

3.1.3 Software Development Processes: Structured vs. Agile

A software development process is a scheme to structure and manage the various aspects of development: requirements elicitation, design, implementation, verification, maintenance, etc. Software engineering [44, 91, 86] has traditionally targeted so-called structured processes\(^1\), such as the Rational Unified Process (RUP), the waterfall model, or the spiral model. Structured processes are characterized by a focus on rigorously defined practices, extensive documentation, and detailed planning and management. More recently, a surge of agile development processes have been introduced to overcome some of the limitations and unsatisfactory aspects of structured processes. Agile processes [6, 22], such as Scrum or eXtreme Programming (XP), emphasize the importance of effective informal communication among developers, and of iterative improvement of implementations driven by use-case scenarios, and they champion small cohesive development teams over large structured processes.

---

\(^1\)Other names for such processes are: heavyweight, plan-driven, disciplined. In this article, we will consistently use the term “structured” to denote “non-agile” processes; this is merely a terminological convention and does not entail that agile processes have no structure whatsoever, or that structured processes are completely inflexible.
3.1. INTRODUCTION AND OVERVIEW

units. The relative merits and applicability of structured vs. agile processes in local software development are fairly well-understood \[71, 57, 7, 73, 11\]: for example, for applications whose requirements are accurately known and not subject to radical changes, a structured development may offer more controllability and better scalability; on the other hand, agile processes may be preferable when requirements are subject to frequent change and achieving a formal and structured communication with stakeholders is difficult or unrealistic.

3.1.4 The Role of Processes in Distributed Development

The present study re-considers the “structured vs. agile” dichotomy in the context of globally distributed software development, and tries to understand whether one of the two development approaches emerges as more appropriate to organize software development carried out by globally distributed teams. The facts learned about non-distributed contexts may not apply to distributed settings, where it is not obvious how to enforce some of the principles underlying structured or agile methods. Agile processes, in particular, often require that [6]:

- all project phases include communication with customers;
- face-to-face exchanges be preferred as the most efficient and effective method of communicating.

In contrast, structured processes often emphasize the importance of maintaining accurate documentation, which can be problematic when cultural and language differences are in place. Correspondingly, effectively applying the principles of agile rather than structured development in a distributed setting has been the subject of much software engineering research [99, 81, 80].

The question remains, however, of what are the relative merits of structured and agile processes for globally distributed software development, and whether one of them is more likely to be effective. We targets this question with a study involving over 31 companies (of size from small to large) for a total of 66 software projects developed in Europe, Asia, and the Americas. The degree of distribution ranges from merely outsourced projects—where management remains in the company’s headquarters while the actual development team operates in a different country—to highly distributed development projects—where members of the same team reside in different countries.

According to the answers collected through questionnaires and interviews, we have classified the development process used in each project into agile
or structured, and we have analyzed the correlation between process type and measures of achieved overall success, importance for the customer, cost-effectiveness, developer motivation, amount of personal communication, and emerge of several problematic aspects. As we discuss in detail in the rest of this chapter, the data collection was designed so as to reduce potential threats to validity, and specifically the intrinsic fuzziness of ordinal-answer questionnaires (see Section 3.3) and the interviewer effect of structured interviews (see Section 3.7):

- The questionnaire (see Table 3.1) included, as much as possible, quantitative descriptions of the possible answers on a scale (for example, “answer 3 corresponds to 3–5 hours per week”). This helps align answers coming from different participants to a common gauge, so that comparing them is meaningful.

- All participants to the study have considerable experience (managers or senior engineers) and reported data about a recently completed single project. This gives us confidence that the participants were aware of the possible pitfalls of reliably reporting complex data, and that they could report on completed projects, where the differences in process and outcome were sufficiently well defined (as opposed to ongoing projects with unclear developments).

- The quantitative analysis only uses data from the questionnaires. We still rely on the interviews to report qualitative and anecdotal data (see Section 3.5), which corroborate and enrich the overall picture about distributed development.

- The number of projects about which we collected data is fairly large for this type of study (see a comparison with related work in Section 3.8); therefore, correlations should easily emerge if present at all.

### 3.1.5 Summary of Results

The bulk of the results show that the differences in any of these measures between agile and structured processes are negligible and with no statistical significance. Therefore, our study suggests that agile and structured processes can be equally effective (or ineffective) for globally distributed software development, and the sources of significant differences in project outcome should be sought in other project characteristics.

These results should not be misread as suggesting that development processes are irrelevant and bear no impact on project outcome. The real take-home lesson is that the development process is not an independent variable,
and hence its choice cannot single-handedly determine the successful or unsuccessful outcome of a project. Single experiences include both great success stories and utter failures with either structured or agile practices. The choice of development process is thus something to be considered with great care, but based on the characteristics of a project and of the development team working on it, as well as on its overall goals—not in a vacuum based on a priori expectations for one-size-fits-all blue-sky solutions.

3.1.6 Outline

The rest of the chapter is organized as follows. Section 3.2 presents the research questions investigated in the case study. Section 3.3 describes the data collection process and the research methodology. Section 3.4 presents the quantitative results of the study, whereas Section 3.5 is devoted to a somewhat informal discussion of other aspects for which only qualitative data is available. Section 3.6 draws the big picture of the study from a practical standpoint. Sections 3.7 and Section 3.8 respectively describe threats to validity and related work. Section 5.7 summarizes and describes future work.

3.2 Research Questions

While the benefits of deploying structured vs. agile processes have been extensively studied in the context of traditional local development, their applicability to and impact on globally distributed development are still largely unknown. This study contributes to filling this knowledge gap by investigating the impact of using different processes—structured rather than agile—on the outcome of software projects carried out in distributed settings. This leads to two overall research questions, the first focusing on project outcome and the second focusing on the emergence of problematic aspects.

RQ1: In software development carried out in globally distributed settings, what is the impact of adopting structured vs. agile processes on the overall success \( A \), importance for customers \( B \), team motivation \( C \), cost-effectiveness \( D \), and amount of real-time \( E \) and asynchronous communication \( F \) ?
RQ2: In software development carried out in globally distributed settings, what is the impact of adopting structured vs. agile processes on the emergence of communication difficulties \( G \), cultural differences \( H \), ineffective project management \( I \), loss or fluctuation of know-how \( J \), shortage of labor skills \( K \), ineffective reading or writing of documentation\(^2\) \( L \) \( M \), interpersonal conflicts \( N \), difficulties in keeping to the project schedule \( O \), and in protecting intellectual property \( P \)?

The choice of project outcome aspects A–P reflects the major dimensions studied in the research literature on distributed development (reviewed in Section 3.8). The specific questions asked in the questionnaires and interviews (see Section 3.3.1 and Table 3.1) outline the definitions we assumed for aspects A–P targeted by the research questions; in the simplest cases, we can just assume dictionary definitions.

### 3.2.1 Hypotheses for RQ1 and RQ2

For each aspect A–F of RQ1 (overall success, cost-effectiveness, etc.), a null-hypothesis states the absence of correlation between development process type and outcome relative to the aspect; all hypotheses refer to projects developed in distributed settings.

- **H\(^A\)_0** There is no difference in the overall success of projects developed using agile methods vs. projects developed using structured methods.
- **H\(^B\)_0** There is no difference in the importance (i.e., criticality for customers) of projects assigned to development using agile methods vs. projects assigned to development using structured methods.
- **H\(^C\)_0** There is no difference in the motivation of teams following agile processes vs. teams following structured processes.
- **H\(^D\)_0** There is no difference in the estimated economic savings (compared to onshore development) for customers in projects using agile methods vs. projects using structured methods.
- **H\(^E\)_0** There is no difference in the amount of real-time communication (e.g., in person or by phone) required by projects developed using agile methods vs. projects developed using structured methods.

\(^2\)For structured processes, the term “documentation” mainly denotes requirement specifications; for agile processes, it mainly denotes use-case scenarios and test cases.
3.2. RESEARCH QUESTIONS

$H_0^F$ There is no difference in the amount of *asynchronous communication* (e.g., emails or wikis) required in projects developed using agile methods vs. projects developed using structured methods.

Similarly, for each potentially problematic aspect $G$–$P$ of RQ2 (communication difficulties, cultural differences, etc.), a null-hypothesis states the absence of correlation between development process type and the emergence of the problematic aspect; again, all hypotheses refer to projects developed in distributed settings.

$H_0^G$ There is no difference in the emergence of *communication difficulties* across geographically distributed units in projects using agile methods vs. projects developed using structured methods.

$H_0^H$ There is no difference in the emergence of *cultural differences* in projects using agile methods vs. projects developed using structured methods.

$H_0^I$ There is no difference in the emergence of *ineffective project management* in projects using agile methods vs. projects developed using structured methods.

$H_0^J$ There is no difference in the emergence of *loss or fluctuation of know-how* in projects using agile methods vs. projects developed using structured methods.

$H_0^K$ There is no difference in the emergence of *shortage of labor skills* in projects using agile methods vs. projects developed using structured methods.

$H_0^L$ There is no difference in the emergence of problems with *ineffective reading of documentation* in projects using agile methods vs. projects developed using structured methods.

$H_0^M$ There is no difference in the emergence of problems with *ineffective writing of documentation* in projects using agile methods vs. projects developed using structured methods.

$H_0^N$ There is no difference in the emergence of *interpersonal conflicts* in projects using agile methods vs. projects developed using structured methods.

$H_0^O$ There is no difference in the emergence of difficulties in *keeping to the project schedule* in projects using agile methods vs. projects developed using structured methods.
There is no difference in the emergence of difficulties in protecting intellectual property in projects using agile methods vs. projects developed using structured methods.

If the collected data manages to falsify, with a degree of statistical significance, any null-hypothesis, there is evidence supporting the corresponding alternative hypothesis: the choice of agile rather than structured development processes has an impact on the outcome of a certain aspect or the emergence of a certain problem in globally distributed projects.

3.2.2 Discussion of the Hypotheses

Familiarity with software development practices might suggest a priori that not all investigated aspects have the same importance or the same dependence on the development process. For example, among the issues pertaining RQ1, the amount of asynchronous communication may be less relevant than the overall success of a project, which is arguably the most important overall goal. Among the aspects mentioned in RQ2, the difficulty of protecting intellectual property may seem to be largely independent on the choice of development process compared to, say, issues with project management. Nonetheless, we investigate all hypotheses independently, to determine to what extent intuition is supported by hard evidence. This is also helpful to reduce the chance that we miss any significant correlation between process type and other aspects due to incorrect prior expectations.

The following sections describe how data was collected to support or falsify the hypotheses above: Section 3.3 discusses the data collection process and Section 3.4 quantitatively analyzes the data. Section 3.5 complements the quantitative analysis with a qualitative presentation of issues related to RQ1 and RQ2.

3.3 Research Methodology

The data was collected in two phases. In the first phase, we sent out questionnaires to companies in Europe, Asia, and the Americas, about their offshore and distributed development projects. In the second phase, we interviewed representatives of several companies located in Switzerland about their distributed development efforts. Both the questionnaires and the interviews targeted distributed projects, collecting data about: their success

\footnote{The dataset is available at \url{http://se.inf.ethz.ch/data/icgse12.zip} Statistical analysis was performed using IBM SPSS v. 20.}
for the companies, their cost-effectiveness, team motivation, importance for customers, amount of communication, problematic aspects that emerged, and whether they were organized according to a structured or agile process. As we discuss in the rest of the section, the questionnaires included, whenever possible, descriptive answers with quantitative references, so as to make comparable the answers coming from different participants.

We did not distinguish among different types of agile (e.g., Scrum vs. extreme programming) or different types of structured (e.g., RUP vs. waterfall) processes. This is consistent with the observation that, while different processes may involve different practices, the principles underlying agile rather than structured methods are normally visibly different and straightforward to identify in practice; after all, agile processes emerged in explicit contrast with traditional structured processes [6]. The complete data set contains information about 66 distributed projects (details about the distribution are in Section 3.3.1), of which 36 deployed agile methods and 30 structured methods.

### 3.3.1 Questionnaires

In the first phase, we contacted through questionnaires companies in various countries and continents worldwide; each questionnaire targeted one software project, containing several questions about the project.

We sent out the questionnaires to over 60 contacts worldwide, and we received replies about 48 projects developed by companies in the USA (14 projects), Nordic countries (12 projects), Germany (6 projects), the UK (4 projects), Russia (1 project), the Netherlands (1 project), Latin America (1 project), and Switzerland (3 projects, from companies other than those involved in the interviews of the second phase); the countries of origin of the remaining 6 projects were unspecified. 22 of the 48 projects were in collaboration with remote units in Russia, 20 in India, 2 in Argentina, and 1 in each of China, Bulgaria, Hungary, Romania.

The information in the questionnaires was provided by people with significant experience: 19 high managers, 19 project leaders, and 10 software engineers, architects, and researchers. 19 projects out of 48 followed structured processes, and 29 applied agile processes.

Table 3.1 lists the questionnaire questions measuring aspects $A$–$P$ to assess hypotheses $H_0^A$–$H_0^P$. As shown in the rightmost column, the ordinal range of available answers for each question came, whenever possible, with a description that refers to quantitative data (to make comparable answers to the same questions coming from different participants). As it is common practice in empirical research based on questionnaires, we used closed ques-
tions (that is, questions with predefined answers), which can be answered in a moderate amount of time. We selected the possible answers and their ranges based on our intuition and previous experience—but not on a pilot study. Participants could choose to not answer individual questions if they did not have access to or were not allowed to disclose the relevant data. Tables 3.2 and 3.3 report the number of valid answers received for each question.

3.3.2 Interviews

In the second phase, we contacted 13 Swiss software companies including: 3 large companies with more than 10'000 employees worldwide; 8 mid-size companies with 200 to 900 employees each; and 2 small companies with less than 100 employees. 6 of the companies also develop hardware products.

We individually interviewed 18 employees from these 13 companies, that have experience with globally distributed development. Of the 18 employees, 9 were high managers (CEOs, CTOs, or business unit leaders), 9 were project managers and senior software engineers.

Each interview discussed a recently completed software project the interviewee had been involved with. All the projects were in collaboration with distributed units from companies in Western Europe, Eastern Europe, Russia, or Asia. In 12 of the 18 projects, the units outside Switzerland were subsidiaries of the main company; the collaboration in the other 6 projects can be characterized as off-shore development provided by external companies. Finally, 11 projects out of 18 followed structured processes, and 7 applied agile processes.

The questions asked during the interviews were similar to those used in the questionnaires; it was much harder, however, to get quantitative data from the interviews, because the interviewees were often evasive when asked to characterize precisely measures such as the success or economic savings of a project, and only gave generic answers such as “the project was successful” or “the savings were small”. For this reason, we used the data from the interviews only in the qualitative analysis of Section 3.5.

3.4 Quantitative Results

We now present the quantitative data analysis of the hypotheses presented in Section 3.2 on the data of the questionnaires (see Section 3.3). Subsection 3.4.3 discusses the six hypotheses $H_0^{A}$ to $H_0^{F}$ related to RQ1 about project outcome; subsection 3.4.4 collectively presents the findings for the ten hypotheses $H_0^{G}$ to $H_0^{P}$ related to RQ2 about problematic aspects.
3.4. QUANTITATIVE RESULTS

<table>
<thead>
<tr>
<th>VARIABLE</th>
<th>QUESTION</th>
<th>ANSWERS AND NUMERIC MAPPING</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Rate the success of the last completed project.</td>
<td>1 (Complete Failure) ... 10 (Full success)</td>
</tr>
<tr>
<td>B</td>
<td>How important was the last completed project for the customer?</td>
<td>1 (Unimportant) ... 10 (Very critical)</td>
</tr>
<tr>
<td>C</td>
<td>How motivated were you working in an outsourcing project?</td>
<td>1 (Not at all) ... 10 (Very much)</td>
</tr>
<tr>
<td>D</td>
<td>What was the financial outcome for the customer compared to on-shore development?</td>
<td>- (Don’t know) 1 (Lost more than 25%) 2 (Lost 10% to 25%) 3 (About even: −10 to 10%) 4 (Saved 10–25%) 5 (Saved 25–50%) 6 (Saved more than 50%)</td>
</tr>
<tr>
<td>E</td>
<td>How often did you have real-time communication with the outsourcing partner?</td>
<td>1 (Never) 2 (1 to 2 times per year) 3 (3 to 5 times per year) 4 (6 to 9 times per year) 5 (10 to 14 times per year) 6 (15 to 30 times per year) 7 (more than 30 times per year)</td>
</tr>
<tr>
<td>F</td>
<td>How often did you have asynchronous communication with the outsourcing partner?</td>
<td>1 (&lt;1 hour per week) 2 (1 to 2 hours per week) 3 (3 to 5 hours per week) 4 (6 to 9 hours per week) 5 (10 or more hours per week)</td>
</tr>
<tr>
<td>G</td>
<td>Was communication due to distance a problem?</td>
<td>- (Don’t know) 1 (Not at all) 2 (A little) 3 (Medium) 4 (Severe)</td>
</tr>
<tr>
<td>H</td>
<td>Were cultural differences a problem?</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>Was project management a problem?</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>Was loss or fluctuations of know-how a problem?</td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>Was a shortage of labor skills a problem?</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>Was reading specifications a problem?</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>Was writing specifications a problem?</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>Were personal conflicts a problem?</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>Was keeping the project schedule a problem?</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>Was protecting intellectual property a problem?</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Questions from the questionnaire used to collect data for hypotheses $H_0^a$, $H_0^b$. For each measured variable in the first column, the questionnaire formulates a question (second column) and an ordinal range of answers with quantitative references (third column, also given to participants). Each question also included the option not to answer.
The initial data-set included information about 48 projects; we removed one of them, as it consisted of a questionnaire with clearly bogus answers, leaving us with data about 47 projects. Some questionnaires were incomplete, in that answers to some questions were missing. The analysis that follows excludes the missing answers for each question; therefore, the number of projects evaluated may vary from question to question.

The analysis for each of the hypotheses $H_0^A$ to $H_0^P$ aims to determine whether the data shows a statistically significant difference between answers about projects using agile processes and answers about projects using structured processes.

The questionnaire consisted of multiple-choice questions; given the nature of the available choices shown in Table 3.1, we should consider the emerging data as *ordinal* but not interval-scale. For each answer, we visually inspected the distribution of data and we performed a Shapiro-Wilk normality test; none of them gave evidence to consider the underlying distributions as normal. The presence of ordinal data and non-normal distributions suggests to deploy the Mann-Whitney-Wilcoxon U test, a non-parametric statistical hypothesis test [3, 98].

### 3.4.1 Analytical Formulation of Hypotheses $H_0^A$ to $H_0^P$

Each hypothesis $H_{0X}$, for $X=A,...,P$ refers to a certain quantity (overall success, importance, motivation, etc.) or to the severity of a certain potentially problematic aspect (communication difficulties, cultural differences, etc.) measured by the corresponding random variables $X$, $AG^X$, and $ST^X$—respectively in all projects, in agile projects, and in structured projects. For example, $H_{0C}$ tests the motivation of teams, hence $AG^C$ models team motivation in agile projects, $ST^C$ models team motivation in structured projects, and $C$ models team motivation across all projects. With this notation, the null hypothesis $H_{0X}$ is expressible as:

$$H_{0X} : \ P(AG^X > ST^X) = P(ST^X > AG^X) ;$$

that is, the probability that random samples of quantity $X$ are larger in agile projects than in structured projects equals the probability that the samples are larger in structured projects than in agile projects. Correspondingly, the alternative hypothesis $H_{1X}$ is that there is a difference in probability, that is:

$$H_{1X} : \ P(AG^X > ST^X) \neq P(ST^X > AG^X) .$$

We do not directly test for differences in medians and means of the random variables $AG^X$ and $ST^X$ using the U test, because that would require that
3.4. QUANTITATIVE RESULTS

the underlying distributions of $AG^X$ and $ST^X$ have the same shape; however, our data does not meet this requirement.

Given a significance level $\alpha = 0.05$, the U test gives a probability $p$ that the data supports the null hypothesis: if $p < \alpha$, the data gives evidence to reject the null hypothesis, if $p > \alpha$, one can not reject the null hypothesis.

3.4.2 Summary of the Quantitative Findings

The overall outcome of our quantitative analyses is that we never reject the null-hypothesis; specifically, the $p$ values are normally much greater than any reasonable significance level, and hence no correlation emerges as significant in spite of the fairly large sample size. Therefore, the data provides no evidence to distinguish between using agile vs. structured processes as independent variables.

We do not reject any of the hypothesis $H_0^X$, for $X = [A \ldots P]$, implying that the study’s data provides no evidence that choice of process significantly impacts the properties $X = [A \ldots P]$.

The following subsections describe the results in more detail. As an afterthought following the quantitative analysis, Section 3.4.5 shows that the data possesses some statistically significant correlations among certain random variables, such as the overall success and the importance of a project. Such correlations are, however, completely independent of the process type—agile or structured.

3.4.3 Quantitative Analysis of RQ1: Project Outcome

The distribution of the data for hypotheses $H_0^A$ to $H_0^F$ is shown in Figure 3.1. While a cursory visual inspection seems to indicate that some aspects—such as the success, economic savings, and amount of asynchronous communication—are impacted by the choice of process, the quantitative analyses show that the differences are not statistically significant.

Table 3.2 lists the analysis results for hypotheses $H_0^A$ to $H_0^F$. For each hypothesis, the table reports standard descriptive statistics (such as median and mean) split according to the process type; and the outcome of U test between data in the two partitions (as described earlier in this section). The descriptive statistics refer to the ordinal scales assigned to answers of each question, shown in Table 3.1 notice that different questions may have different scales (which is not an issue since each hypothesis refers to a single question). In all, none of the U tests achieves statistical significance; in fact,
3.4.4 Quantitative Analysis of RQ2: Problematic Aspects

The distribution of the data for hypotheses $H_0^A$ to $H_0^F$ is shown in Figure 3.2. Visual inspection may seem to indicate that some problematic aspects emerge more frequently with one process type than with the other type—in particular, communication difficulties are reported as more common with structured processes, and ineffective management and cultural differences tend to be
3.4. QUANTITATIVE RESULTS

Figure 3.2: Bar charts for the data corresponding to hypotheses $H_0^G$ to $H_0^P$, see Section 3.4.4 for a detailed analysis.
more severe with agile processes. The quantitative analyses, however, show that none of the differences is statistically significant.

Table 3.3 lists the analysis results for hypotheses $H_G^0$ to $H_P^0$. For each hypothesis, the table reports the same statistics as Table 3.2 but about variables $G$–$P$. The descriptive statistics refer to the ordinal scale 1–4 of severity shown in Table 3.1, which is the same for all variables related to RQ2. In all, none of the U tests achieves statistical significance; in fact, $p$ is greater than 0.24 in all cases but one (where it still greater than 0.11), which means that we clearly have no statistical significance.

### 3.4.5 Other Correlations Between Variables

The analysis using the Mann-Whitney-Wilcoxon U test did not provide any evidence of statistically significant correlations between the project type (agile or structured) and the reported measures of different quantities (success, team motivation, communication problems, etc.). Are there statistically significant correlations between other pairs of variables, not involving the project type?

To answer this question, we calculated Kendall’s $\tau$ rank correlation coefficient for all pairs of random variables $A$ to $P$ measuring the quantities involved in hypotheses $H_A^0$ to $H_P^0$ across all 47 projects targeted by the questionnaires. Table 3.4 lists the cases of significant correlations: the pairs of

---

**Table 3.2: Quantitative analysis of hypotheses $H_A^0$ to $H_F^0$.** Column T identifies the process type: agile (A) or structured (S); column # reports the number of projects that provided data about the aspect. Columns $U$ and $p$ report the results of applying the U test. For the reported severity of each aspect, the remaining columns report Median, Minimum and Maximum, Mean Rank, Mean, and Standard deviation. See Table 3.1 for the ordinal scales of each variable.
### 3.4. QUANTITATIVE RESULTS

Table 3.3: Quantitative analysis of hypotheses $H_0^G$ to $H_0^P$. Column T identifies the process type: agile (A) or structured (S); column # reports the number of projects that provided data about the aspect. Columns $U$ and $p$ report the results of applying the U test. For the reported severity of each aspect, the remaining columns report Median, Minimum and Maximum, Mean Rank, Mean, and Standard deviation. All variables are on a 1–4 ordinal scale (see Table 3.1).

<table>
<thead>
<tr>
<th>ASPECT</th>
<th>T</th>
<th>#</th>
<th>Median</th>
<th>Min/Max</th>
<th>Mean Rank</th>
<th>Mean</th>
<th>Std. dev.</th>
<th>$U$</th>
<th>$p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>communication</td>
<td>A 29</td>
<td>2</td>
<td>1/3</td>
<td>23.81</td>
<td>1.97</td>
<td>0.778</td>
<td>255.5</td>
<td>0.898</td>
<td></td>
</tr>
<tr>
<td>cultural differences</td>
<td>S 29</td>
<td>2</td>
<td>1/4</td>
<td>26.24</td>
<td>1.83</td>
<td>0.848</td>
<td>196</td>
<td>0.119</td>
<td></td>
</tr>
<tr>
<td>project management</td>
<td>A 29</td>
<td>2</td>
<td>1/3</td>
<td>23.81</td>
<td>1.97</td>
<td>0.778</td>
<td>255.5</td>
<td>0.898</td>
<td></td>
</tr>
<tr>
<td>loss of know-how</td>
<td>S 29</td>
<td>2</td>
<td>1/3</td>
<td>23.81</td>
<td>1.97</td>
<td>0.778</td>
<td>255.5</td>
<td>0.898</td>
<td></td>
</tr>
<tr>
<td>labor skills</td>
<td>A 28</td>
<td>1</td>
<td>1/3</td>
<td>22.71</td>
<td>1.54</td>
<td>1.015</td>
<td>230</td>
<td>0.578</td>
<td></td>
</tr>
<tr>
<td>reading doc.</td>
<td>S 28</td>
<td>1</td>
<td>1/3</td>
<td>22.88</td>
<td>1.54</td>
<td>0.693</td>
<td>234</td>
<td>0.926</td>
<td></td>
</tr>
<tr>
<td>personal conflicts</td>
<td>A 28</td>
<td>2</td>
<td>1/2</td>
<td>22.36</td>
<td>1.45</td>
<td>0.506</td>
<td>213.5</td>
<td>0.242</td>
<td></td>
</tr>
<tr>
<td>project schedule</td>
<td>S 28</td>
<td>2</td>
<td>1/4</td>
<td>24.69</td>
<td>2.07</td>
<td>0.884</td>
<td>241</td>
<td>0.636</td>
<td></td>
</tr>
<tr>
<td>intellectual property</td>
<td>A 26</td>
<td>1</td>
<td>1/3</td>
<td>21.36</td>
<td>1.35</td>
<td>0.464</td>
<td>209.5</td>
<td>0.358</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.4: Correlation analysis based on Kendall’s $\tau$ correlation coefficient. The pairs of random variables are those with a correlation coefficient $\tau \geq 0.4$ or $\tau \leq -0.4$ and with significance $p < 0.01$.

<table>
<thead>
<tr>
<th>PAIR OF QUANTITIES</th>
<th>$\tau$</th>
<th>$p$ (two-sided)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A (overall success) / B (importance)</td>
<td>0.505</td>
<td>&lt; 0.001</td>
</tr>
<tr>
<td>L (reading doc.) / K (labor skills)</td>
<td>0.493</td>
<td>&lt; 0.001</td>
</tr>
<tr>
<td>L (reading doc.) / M (writing doc.)</td>
<td>0.645</td>
<td>&lt; 0.001</td>
</tr>
<tr>
<td>N (personal conflicts) / K (labor skills)</td>
<td>0.407</td>
<td>= 0.003</td>
</tr>
</tbody>
</table>

These correlations may suggest lines for future studies about significant aspects of globally distributed software development. At the same time, none of them is really surprising in hindsight. If a project is important for the customer, it is reasonable that more effort is put into it so as to achieve overall
success; this justifies the correlation between $A$ and $B$. Properly handling documentation (whether formal documents or use-case scenarios) requires skilled developers (correlation between $L$ and $M$), who can also abate the likelihood of interpersonal conflicts due to lack of confidence in other team members (correlation between $N$ and $K$). Finally, poorly written documentation is also hard to read, reflected in the correlation between variables $L$ and $M$.

The study’s data shows significant correlations between different variables in $A, \ldots, P$. These correlations are, however, independent from type of process.

### 3.5 Qualitative Results

This section discusses various aspects of distributed software development that emerged from all the collected data: the costs of nearshore vs. offshore development, the average success and quality achieved in distributed projects, the role of personnel skills and of communication patterns, the issues of personnel fluctuation and intellectual property management, the criteria for choosing development process, and the typical team size.

Unlike the results of Section 3.4, the data is mostly qualitative and deals with aspects complementary to the choice of development process that also affect the outcome and success of projects. Sections 3.5.1 through 3.5.7 report data from the interviews only (for a total of 18 projects, see Section 3.3), whereas the other Sections 3.5.8 and 3.5.9 also incorporate data from the questionnaires (totaling 65 projects).

Unlike the previous Section 3.4, the results in the current section are mainly qualitative. The measures we report in the following subsections should therefore be considered elements to articulate the discussion and corroborate the quantitative picture of the rest of the chapter.

#### 3.5.1 Costs of Nearshore vs. Offshore

The conventional wisdom is that nearshore development (where developers work close to customers, in terms of time zones and distance) makes team coordination and collaboration with customers easier, but is significantly more expensive than offshore development (which can use less expensive developers in countries such as India and China). The interviews with Swiss companies representatives revealed, however, that most companies that prefer nearshore development do it for legal reasons and because it reduces the amount of traveling, rather than directly to reduce costs.
In fact, our data does not show correlation between costs and location: the overall costs in nearshore development are not necessarily higher than in offshore development. While the salaries of programmers in Asia is typically between $\frac{1}{5}$ and $\frac{1}{3}$ of the corresponding positions in Switzerland, the overall project costs are also affected by factors such as productivity, communication and management overhead, and various costs for setting up and maintaining offices, which weakens the dependence between total costs and location.

Similarly, our data shows no significant cost differences between globally distributed projects (where members of the same development team operate in different locations) and outsourced projects (where management is in a location, and all development takes place in a different location): compared with purely local development, globally distributed projects reported savings for an average 28% and a standard deviation of 11%;\footnote{Data about 8 projects.} outsourced projects reported very similar savings for an average of 33% and a standard deviation of 11%.\footnote{Data about 10 projects.} The overhead (for communication, management, and office costs) is also almost identical in globally distributed and in outsourced projects, ranging between 35% and 45%. For example, creating a new unit in an outsourced location requires 2 to 5 months to setup the office, plus another 3 months to become productive; the investment pays back after 3–4 years on average.

### 3.5.2 Project Success

Out of the 18 projects surveyed in the interviews: 11 are considered “complete success”; and 5 are “overall success” which, however, suffered non-trivial problems during development. The major sources of problems and difficulties were: unqualified personnel, cultural and communication difficulties, deficiencies of the infrastructure, insufficient interaction among units. The major sources of success were: skilled personnel and effective team building (after a solid team is established, the members will work proficiently even if distributed in different locations).

The data does not show any visible correlation between the motivation of a development team and the overall success (while Section 3.4.5 showed correlation between project importance and success). Other factors, however, seem to positively affect the final outcome: companies that are comfortable with frequent changes of customer, working on series of short-term independent projects rather than on the long-term incremental development of
a single product, tend to encounter fewer problems. The interviews suggest that companies adapt to frequently changing customers by improving their process and management skills; companies bound to fewer customers, in contrast, accumulate valuable knowledge but tend to have less robust organizational structures. This may suggest that flexibility is a central skill for distributed software development, required to react to and minimize the effects of inevitable problems.

3.5.3 Project Quality

The overall quality of the majority of projects was reported as “good” or better, but our interviews revealed that development problems related to quality are not uncommon, especially at the beginning of projects (where problems were reported in over 50% of the projects). It seems that quality correlates positively with timeliness: late projects are unlikely to achieve a good quality; nonetheless, compromises on quality are often accepted to meet deadlines. We observed no significant differences between nearshore and offshore development.

3.5.4 Personnel Skills

Personnel skills are a major factor of project success. Most interviewees think that personnel skills decrease with distance: the most skilled personnel is in Switzerland, followed by the personnel in nearshore locations (typically, Eastern Europe), and then by personnel in offshore locations (India and China). The deterioration of skills is attributed to difficulties in communication and collaboration, and more generally to the challenges introduced by distributed software engineering.

3.5.5 Communication Patterns

Effective communication is another major factor of project success, and in fact 13 out of 18 projects required a weekly (virtual) meeting among all project members. Figure 3.3 shows the means of communication used in the 18 projects, classified by their richness (i.e., perceived effectiveness) and synchrony. Most of the communication among developers takes place using instant messaging, which is preferred over voice calls and face-to-face communication because it helps bypass communication obstacles—for example, strong accents—and because it is a good compromise between real-time and asynchronous communication. Time zones were not reported as an issue for communication.
3.5. QUALITATIVE RESULTS

Figure 3.3: Communication means categorized by richness and synchrony of different communication types. The diameter of the circles is proportional to the amount of communication (calculated as means) used in the distributed projects examined through interviews.

3.5.6 Personnel Fluctuation

Personnel fluctuation refers to the phenomenon of frequently changing teams, whose members are likely to quit on a short notice (or no notice at all) and have to be rapidly replaced. Fluctuation often causes loss of know-how, delays, and cost increases for training and recruitment. While 6 out of 18 projects report fluctuation levels higher than those they experience in projects developed entirely in Switzerland, only one of them found fluctuation to be a critical problem in distributed development.

The interviewees mentioned different mentalities of work and the availability of many job opportunities as the main reasons for why offshore personnel may fluctuate. In some cases, fluctuation is just a consequence of the less stimulating tasks being often outsourced: the offshore personnel would be motivated were it assigned to interesting jobs.

3.5.7 Intellectual Property

Companies consider intellectual property a key asset, as it provides fundamental protections against competitors. None of the projects encountered problems regarding intellectual property. This was probably the results of two factors: first, intellectual property is project-specific, and hence it is
hardly marketable to others in the case of custom software projects (all the projects targeted by the interviews are custom). Second, the critical components of each project are developed onshore, where the most valuable intellectual property can be adequately protected.

3.5.8 Onshoring vs. Offshoring in Different Phases

For each development phase (requirements, design, implementation, unit and system testing, deployment, and maintenance), we asked which percentage of the time spent on that activity was assigned to the onshore rather than to the offshore units of a distributed project. Figure 3.4 shows the data about 66 projects, discussed during the interviews and reported in the questionnaires (in addition to the data of the questionnaire data analyzed in Section 3.4). The graphs only suggest a slight difference in the system design phase—where offshoring is more common with agile processes—and in the unit testing phase—where offshoring is more common with structured processes. The overall qualitative trends are, however, quite similar in the two plots of Figure 3.4, consistently with the general observations of the study.

![Figure 3.4: Allocation (percentage of time assigned to offshore or onshore units) of development phases in agile and structured processes.](image)

---

6We did not provide a definition of the various activities to interviewees, relying on their conventional understanding of the terms in each context. While the same phase may have fairly different connotations in agile rather than structured processes (e.g. requirements in agile projects might refer to user stories), no interviewee voiced doubts about how to assign activities within their projects or asked for clarifications about this aspect.
3.5.9 Team Size

Agile development practices focus on small teams, whereas structured processes often are designed with larger teams in mind; the data we collected about team size confirms these expectations. Figure 3.5 shows the team size of 66 projects, discussed during the interviews and reported in the questionnaires (in addition to the data of the questionnaire data analyzed in Section 3.4); agile projects tend to have smaller teams than structured projects: most agile projects deploy teams of 30 people or fewer, whereas structured projects may deploy large teams, up to 120 people or more. It is noteworthy, however, that agile practices have been scaled up to teams of more than 60 developers in a couple of cases; and that structured processes have been followed even with quite small (<4) teams.

![Figure 3.5: Team size in agile and structured projects.](image)

3.6 Discussion: Practical Implications

The bulk of our study showed no statistically significant correlations between variables measuring outcomes of distributed processes and the type of development project followed—agile or structured. What are the take-home lessons of this overall result for practitioners?

During the informal discussions following the presentation of an initial version of this article [36], our results were often perceived as provocative, as they seemed to clash with the direct experience of many experienced practitioners of distributed development. Everyone had their success stories,
reporting noticeable improvements as soon as they switched from a process type to another; how can our results so blatantly contradict their experience?

A common thread we noticed in all of such success stories is that they were mostly anecdotal and, even in the few cases where they were based on rigorously collected quantitative data, they invariably involved only a handful of projects (in many cases, only one) managed and performed by the same advocate of the particular choices made. Our study, in contrast, targets a substantial number of different projects, involves quantitative data (even taking into account the limitations of questionnaires discussed in Section 3.7), and does not target any projects in which we were even remotely involved. The implications of sharpening the experimental conditions are well-known: if there are significant correlations, they should be magnified by better designed experiments and larger sample sizes.

In our study, we observed the opposite: the claimed superiority of one or the other development process for distributed development vanished as we looked at correlations of many variables and over a larger number of projects. As discussed in Section 3.4, we do not get even close to statistical significance: most of our $p$ values are larger than 50%. Rather than resuscitating the hopes that “one process type is intrinsically better”, the qualitative analysis of Section 3.5 corroborates the quantitative results whenever it targets similar aspects, since there is no distinctive trend that emerges based on the structured vs. agile dichotomy. While experiments can never establish a negative, the only reasonable conclusion is that there are no significant intrinsic correlations between development process type and the numerous variables we considered to measure project outcome and problematic aspects.

While we maintain that this result is sound, it should not be misinterpreted as to suggest that “processes don’t matter”. The only sensible way to reconcile the individual personal successes of many practitioners with the big picture drawn by our study is concluding that the process is not an independent variable. We could not find any obvious aspect whose outcome is single handedly influenced by the choice of process. Therefore, we cannot expect to positively affect the chances of success (or other dimensions) of a project just by choosing to be agile rather than structured. Practitioners should choose the process carefully, based on the project characteristics, organizational structure of the company, and their previous successful or unsuccessful experiences, rather than a priori expecting that a single choice can be a silver bullet. To summarize with an alternative slogan: “processes do matter, but they’re not all that matters”.
3.7 Threats to Validity

We discuss the threats to validity in two categories: internal and external. Internal validity refers to whether the study supports the findings. External validity refers to whether the findings can be generalized.

3.7.1 Internal Validity

A number of threats to internal validity may surface in studies based on surveys and interviews. Interviews feature a trade-off between minimizing “interviewer effects” [42]—the interviewer giving subtle clues about preferred answers—and ensuring quality of answers. One of the authors of this study carried out the interviews using brief and schematic questions (like those used in the questionnaires) in order to minimize interviewer effects. With some interviewees, however, this resulted in insufficiently clear or vague answers. For example, several interviewees responded to multiple-choice questions with open answers that did not stick to the available choices. In these cases, the interviewer sometimes tried to improve the quality of answers by using a more dialectical style of inquiry, possibly at the risk of introducing interviewer effects. We cannot guarantee that the optimal trade-off was achieved in all cases.

Another potential threat to internal validity is the risk that the granularity of multiple-choice answers (in questionnaires and interviews) is too coarse. In particular, the dichotomy between agile and structured processes does not allow for “hybrid processes”, which may be used in practice. Also, the different backgrounds of study participants could have resulted in different interpretations of the same ordinal scales (for example, the “overall success” of the same project would be ranked at a different level on a scale of 1 to 10 by different individuals). Adding redundancy (multiple questions to assess the same variable) into the questionnaire could have helped here, but we decided against it to limit the overall size of the questionnaire. Finally, the absence of a control group and the fact that we did not have direct access to data about projects make it impossible to evaluate the genuineness of the data collected with interviews and questionnaires: we do not know how precise (and objective) the assessment of quantities such as “overall success” or “economic savings” was, nor whether processes classified as “agile” (or “structured”) properly followed the agile (or structured) principles and practices.

While these threats are inherent in studies based on questionnaires and interviews, we have reasons to assume they had only limited impact. First, participants were asked to report on the latest completed single (agile or...
structured) globally distributed software development project; hence they had a chance to select one well-rounded example that unambiguously fits the agile or structured paradigm, rather than a hybrid. Furthermore, the differences among agile (or among structured) processes are likely to be small compared to the differences between any one agile and any one structured process. In fact, agile processes emerged as a reaction against mainstream development practices, hence the “agile vs. structured” classification is reasonably robust. Second, Table 3.1 shows how we provided descriptions for the ordinal values of answers to questions in the questionnaires; the descriptions typically include quantitative references (e.g., “less than 2 hours per week”). This helps ground the various answers on quantitative data, and reduces the risk of wildly different interpretations by different respondents. Third, we limited the quantitative analysis (Section 3.4) to the more reliable data coming from the questionnaires, whereas we used the possibly somewhat iffy data coming from the interviews only qualitatively, to corroborate the overall picture drawn by the study (Section 3.5). Fourth, the wide array of variables analyzed independently (also see the discussion in Section 3.4.5) increases the chance that, if some significant correlation were present, we would have detected it. About the remaining threats, the rank and experience of most participants to the study positively reflect on the chances of having obtained quantitative estimates of fair and uniform quality.

3.7.2 External Validity

The major threats to external validity for studies based on surveys come from insufficient coverage or responsiveness. Coverage measures to what extent the data-set supports generalization of the findings. Responsiveness quantifies the amount of “non-respondents”, that is contacts who received a questionnaire but did not reply with meaningful data. A low responsiveness is a threat to external validity, as non-respondents may exhibit some characteristics relevant for the study and underrepresented among respondents.

In our study, we sent out questionnaires to over 60 contacts and we received 48 replies (one was discarded). Even though we do not know for sure whether some of the contacts forwarded the questionnaires to others (the replies were anonymous for confidentiality reasons), the figures seem to indicate a low risk of bias due to lack of responsiveness.

Assessing the coverage is harder for our study. The data collected through interviews was limited to projects developed by Swiss companies; the online questionnaires reached 18 different companies worldwide, but the vast majority of these companies have their headquarters in Europe or North America. We cannot prove that the experience of our respondents is representative of
the entire population of distributed software projects, which may affect the generalizability of our findings.

3.8 Related Work

This section presents related work in three areas: empirical studies on agile processes in local development settings (Section 3.8.1), on the issues and challenges raised by distributed development (Section 3.8.2), and on applying agile processes in distributed settings (Section 3.8.3). Section 3.1 listed general references on development processes and their role in the software development life-cycle.

3.8.1 Agile Processes for Local Development

The effectiveness of agile processes in collocated projects has been widely investigated empirically. [71] studied the outcome of extreme programming practices in a programming course for graduate students. Their data characterizes the performance of 12 students grouped in pairs, each pair carrying out a 40-hour programming project and 3 smaller assignments (totaling about 600 minutes of work). The study showed that groups using extreme programming produced high-quality code, but it also exposed some difficulties in applying this agile methodology at best in practice.

[57] also analyzed extreme programming practices, and in particular pair programming. Their study involved both students and professional programmers in a controlled setting, where they developed implementations of size up to 8000 lines of code. The results provided no evidence that pair programming practices improve productivity or the quality of the produced code, compared with programmers working solo.

[7] surveyed the results of pair programming practices at Microsoft. Professional programmers, testers, and manager with about 10 years of experience took part in the survey; the majority reported that pair programming works well for them and produces higher-quality code.

[8] conducted an empirical study about test-driven development—another practice of extreme programming—at Microsoft. The study compared test-driven development against more traditional practices, showing an increase in code quality but also in development time (by about 15%) with the adoption of test-driven development.

[73] compared development with extreme programming against development following CMM level 2. Their study, targeting university students, revealed that CMM implementations are more stable and contain fewer bugs,
but programmers following CMM practices perceive their job as more tedious.

### 3.8.2 Empirical Studies on Distributed Development

Empirical studies on globally distributed development have analyzed different aspects, including communication patterns, the effect of time zones, and achievable quality and productivity.

Several studies focused on the amount and type of communication required by distributed projects. For example, [1] reported that the frequency of communication among engineers whose offices are more than 30 meters apart drops to almost the same level as that of engineers separated by several miles. In the same vein, [13] identified loss of communication as one of four major risk factors that can lead to the failure of distributed software projects.

[52] analyzed the impact of globally distributed development on the amount of communication needed to agree on and implement requests for modifications of existing implementations. They found that, when developers are geographically distributed, the overall time increases by a factor of 2.5 on average. If, however, the effect of other variables such as the number of people involved in a task and the size of the required modifications is properly taken into account, the differences in communication time between distributed and collocated teams are no longer significant. Other findings were that communications is much more frequent among collocated than among remote developers; and that the size of the social network (i.e., the number of colleagues a developer ever interacts with) is significantly smaller for programmers working in distributed teams.

Other studies of distributed development focus on achievable quality. For example, [10] present a case study on the development of Windows Vista, comparing the failures of components developed by distributed teams with those of components developed by collocated teams. Their results show no significant differences in the two cases. [97] examined how distributed development affects defect density, coding styles, and productivity in an open source project. He found that there is little or no correlation between geographic distribution and these quality metrics.

None of these studies targeted the type of processes adopted in distributed development, which is instead the focus of the present work. Taking a different angle, [16] analyzed the mutual impact of process maturity and distribution on project quality. Their study classified companies according to their CMMI level, showing that the advantages of processes at higher maturity levels decrease with the distribution of teams. They do not compare structured and agile processes, as the present study does.
3.8.3 Agile Processes for Distributed Development

There are some clear challenges involved in applying agile processes—such as Scrum and extreme programming—to distributed projects. Researchers have proposed changes to agile practices that render them applicable in globally distributed settings. Correspondingly, the remainder of this section summarizes a few empirical studies that have analyzed distributed projects using agile methods. The present study complements such work, as it compares the impact of agile methods against that of structured methods for distributed development.

[64] studied the communication practices of extreme programming teams distributed between the USA and the Czech Republic. The developers created an environment for informal communication in a distributed setting, which helped develop user-story specifications and solve technical problems quickly and efficiently. Face-to-face communication was effectively replaced by other means of communication.

[81] studied how Scrum is performed in distributed projects. Their interview of 19 team members in 3 companies in Finland identified best practices, benefits, and challenges involved in the various Scrum activities such as “daily scrum”, “sprints”, and “sprint planning meeting”.

[99] developed another evaluation of agile processes, targeting only one company. Besides surveying best practices and lessons learned, they make a brief comparison of the productivity (measured as lines of code over time) of 15 projects using agile and non agile processes, and they report a 10% increase in the productivity with agile methods.

3.9 Summary

We presented a case study analyzing the impact of software processes on distributed development. We have examined a total of 66 industry projects, classified them into agile and structured, and evaluated the correlation between process type and success, importance, economic savings of projects, team motivation, and real-time and asynchronous communication, as well as the emergence of problematic aspects such as management difficulties and personal conflicts. The collected data shows that the correlations between process type and the other measures are negligible and without statistical significance.
Chapter 4

Conflicts and Awareness

4.1 Introduction

An elusive, yet major, issue in collaborative endeavors is awareness. Coordination between parties requires that they be aware of each other’s work, to avoid duplications and redundancies, and to ensure that independently developed modules can function in combination. Conversely, insufficient awareness indicates lack of crucial information, which may disrupt progress and jeopardize efficiency and timeliness.

This chapter presents an empirical study of the problems related to awareness deficiency in the practice of distributed software development, with the goals of understanding their severity and suggesting solutions to allay them or to prevent them from happening. A concrete simplified scenario will help better introduce the terms of the problem.

Motivating scenario. Anita and Bruno are developers who respectively belong to the American and Brazilian development units of a globally distributed software company; they have been working together on a software for trading stocks. As it is customary these days, they use a distributed version control system to keep track of changes and to share them. Following another good practice of software engineering, they have modularized the system so that Anita can work on improving the features to sell stocks while Bruno is working on a GUI for the whole system. Since they work on distinct private copies of the code base, Anita and Bruno can see each other’s work only when they push the local commits to a shared repository. Bruno proceeds with his work until he realizes that he has been refactoring some functionality whose original design was Anita’s. At this point, he has to interrupt his work and try to contact her. In an online meeting, they discuss progress in general terms; Anita is however reluctant to share her work so
far, since she hasn’t had a chance to test some new features that Bruno’s work is using. After a few days, when they are finally ready to merge their changes to the code, it is clear that Bruno’s refactoring work is inconsistent with Anita’s. The two meet again online and reconcile the inconsistencies until merging is possible without conflicts. It turns out that Anita’s conflicting changes were introduced over two weeks ago, but Bruno could not plan his work around them since he was unaware of the location and nature of Anita’s changes.

Research questions. The scenario describes some frequently occurring problems related to insufficient awareness of the work of other developers. In particular, it highlights merge conflicts as a possible outcome, and hints at the fact that insufficient awareness often leads to work interruptions and deteriorates coordination. Based on this, we frame the problem of awareness into the following research questions:

**RQ.A:** How frequent and how significant are merge conflicts? Do they originate more often in the work of co-located or of remote team members?

**RQ.B:** How frequent and how significant is insufficient awareness (of the work of team members)? Does it originate more often in the work of co-located or of remote team members?

**RQ.C:** What are the effects of merge conflicts and of insufficient awareness on project development?

**RQ.D:** What are the frequency and detail level with which awareness information should be provided that are preferred by developers in a distributed setting?

Summary of findings. Section 4.3 describes our study’s findings in details. The most important results are:

- The likelihood of incurring into merge conflicts is not significantly affected by the location (co-located vs. remote) of developers within the same team.

- Interruptions due to insufficient awareness occur frequently for teams of non-trivial size. As for merge conflicts, the location of developers does not significantly affect the likelihood of such interruptions.

- Interruptions impact more negatively than just merge conflicts measures such as productivity, motivation, and keeping to the schedule.
• Developers would appreciate having access to awareness information frequently but not in real time; they have, however, diverse preferences regarding the level of detail in which such information should be made available.

**Awareness applied to development tools.** In the last decade, development tools have been introduced specifically to provide improved awareness of other developers’ work, to reduce the chance that merge conflicts occur, and to facilitate resolution when conflicts do occur. The findings presented in this chapter can help improve such tools by suggesting how awareness-related problems occur in practice and what their impact is. An overview of the state of the art for awareness tools is provided in chapter [5].

### 4.2 Design of the Empirical Study

The subjects participating in our empirical study were students enrolled in the 2013 edition of DOSE, a master-level university course on “Distributed and Outsourced Software Engineering” which we have been organizing for several years [77] [76]. Ten universities across the globe took part in the 2013 edition of the course: the University of Rio Cuarto (Argentina); the University of Adelaide (Australia); the Pontificia Universidade Catolica do Rio Grande do Sul (Brazil); the IT University of Copenhagen (Denmark); Cairo University (Egypt); the Politecnico di Milano (Italy); the State University of Nizhny Novgorod (Russia); the Universidad Politecnica de Madrid (Spain); ETH Zurich (Switzerland); and the University of Zurich (Switzerland).

DOSE teaches globally distributed software engineering in a globally distributed setting: its key component is a software development project carried out by the students in inter-university distributed teams. The 2013 project theme was the development of a platform for networked multi-player games. We defined the overall architecture so that it could accommodate top-level modules for different games. The 171 students from the 10 universities were arranged in 12 development teams; each development team produced a different game for the platform. 134 students were active developers; the remaining 37 students played the role of project customers.

Each team consisted of 3 groups with different responsibilities: one group focused on GUI and networking development; one group on implementing the game logic; and one group on the artificial intelligence component. The requirements elicitation phase was completed before the actual development started.\(^1\)

Each group consisted of 3 or 4 student developers; members of

---

\(^1\)For organizational reasons that are not important here, it mainly involved students...
the same group were located in the same country and attended the same university. But the 3 different groups making up a development team were located in different countries, so as to make it a distributed development effort.

For our empirical study, we collected data about the development that spread over a total of 4 weeks, and produced 12 successful projects. (The average project size was 11,776 lines of code in 68 classes).

4.2.1 Data Collection

We collected data for our study from two sources: a questionnaire and a real-time tracking tool.

While merge conflicts are well-defined events, a general problem with studies of awareness is that it is hard to measure lack of or insufficient awareness directly. Instead, we shifted all questions from awareness to the tightly related—but much more clearly defined and tangible—problem of interruptions due to lack of or inconsistent information. For example, if we cannot proceed because we realize that a function (originally developed by another team member) does not behave as per requirements, it would be useful to acquire the information about whether other developers have noticed the same problem, and whether someone is already in the process of fixing it. Thus, we make the underlying assumption that: interruptions of the development workflow are indicative of insufficient awareness of the work of other developers on the same team. Section 4.3.3 will discuss data that confirms a posteriori the soundness of this assumption.

Questionnaire. The questionnaire consisted of two parts. The first part asked for a self-assessment of the participants’ experience in distributed development and in some of the tools used for the project. Section 4.2.2 outlines the picture that emerged from these questions. The second, and major, part of the questionnaire asked the participants to express their preferences and their experience during project development regarding several aspects:

- Merge conflicts and usage of distributed version control (specifically, Git).
- Interaction and problems during development with the local (intragroup) team members.

from Australia and Brazil.

2The questionnaire and all collected data is available for analysis and replication: [http://se.inf.ethz.ch/data/awareness](http://se.inf.ethz.ch/data/awareness)
• Interaction and problems during development with the remote (inter-group) team members.
• How they coped with interruptions due to interaction.
• Preferences regarding awareness of other team members’ work.

We illustrate the detailed content of the specific questions as we analyze the data in Section 4.3.

Real-time tracking. With the goal of collecting finer-grained quantitative information about interruptions of the workflow (as proxies for awareness problems), we built a web-based session tracking tool called James.3 Whenever starting a programming session, one can also activate James in a browser. James provides a simple interface to enter data about interruptions, shown in Figure 4.1. Whenever a workflow interruption occurs, one increments the counter according to the nature of the problem and to whether the object of the interruption pertains the work of a local or remote team member. James logs all the events in real time, and also offers the option to store free text notes in case something unusual is worth recording.

![Figure 4.1: Screenshot of the web-based tracking tool.](http://cloudstudio.ethz.ch/james)
4.2.2 Participants

Questionnaire. At the end of the course, we invited all 134 student developers of DOSE 2013 to fill in the questionnaire; no reward was offered. 105 of the students accepted the invitation to fill in the questionnaire, giving a solid response rate of 78%.

Background and experience. It is useful to understand the self-assessed experience of the 105 participants in matters of distributed development. We asked to rate their experience on a scale of 1 to 7 (where 1 means no experience and 7 means lots of experience) in: programming in general; programming in teams of two or more developers; programming in distributed teams of two or more developers; and using distributed version control (in particular, Git). Figure 4.2 and Table 4.1 summarize the results. Overall, the participants had fairly extensive (for students) experience in programming and team programming, but little or no experience with distributed development and distributed version control systems. As we will see in Section 4.3, some of the study results reflect this situation (in particular, the lack of experience with Git exacerbated some problems related to merge conflicts).

![Figure 4.2: Experience of the study participants in programming, team programming, distributed team programming, and Git. Experience ranges over a 1–7 scale.](image-url)
Table 4.1: Experience of the study participants in programming, team programming, distributed team programming, and Git. Experience ranges over a 1–7 scale.

**Real-time tracking.** Before beginning the development phase, we invited all the 134 student developers of DOSE 2013 to use the real-time tracking tool during development; no reward was offered other than a public acknowledgment. Since this was a much more burdensome task than filling in a questionnaire, we expected only a minority of motivated students to reply; we made clear in the request that there was no obligation involved, but also that we expected volunteers to do a good job and use the tool throughout the 4 weeks of development. 15 students accepted the invitation and used the real-time tracking tool. Post mortem, we removed the few obviously spurious sessions from the logs, leaving us with data about 106 distinct development sessions, each lasting an average of 2.9 hours. Table 4.2 shows more statistics about the data collected with the session tracking tool.

<table>
<thead>
<tr>
<th></th>
<th>median</th>
<th>min</th>
<th>max</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>programming</td>
<td>5</td>
<td>2</td>
<td>7</td>
<td>4.6</td>
</tr>
<tr>
<td>team programming</td>
<td>5</td>
<td>1</td>
<td>7</td>
<td>4.5</td>
</tr>
<tr>
<td>distributed programming</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>2.1</td>
</tr>
<tr>
<td>Git</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>2.9</td>
</tr>
</tbody>
</table>

Table 4.2: Data collected through the tracking tool.

We discuss in detail the data about real-time tracking in combination with the questionnaire data in the next section.
4.3 Results of the Empirical Study

The data collected in our empirical study demonstrates how merge conflicts and (lack of) awareness impact on distributed software development activities—in particular, coordination between developers on the same team. This section presents the results that emerged from a combined analysis of the questionnaires and the real-time tracking data (described in Section 4.2). Sections 4.3.1 through 4.3.4 describe the main findings corresponding to research questions [RQ.A] through [RQ.D]. Section 4.3.5 discusses correlations between data and how they delineate other potentially disruptive aspects of distributed software development.

4.3.1 Merge Conflicts: Frequency and Origin

Research question [RQ.A] points to the well-known problem of merge conflicts: upon trying to export their local modifications into a shared global code base, developers may introduce inconsistent changes that must be reconciled. This section discusses how frequently merge conflicts occurred in our study; how much time conflict resolution took; and whether changes by co-located or by remote team members were the main source of conflicts.

Frequency. To get a qualitative picture of how frequently merge conflicts occur, we asked how many conflicts occurred over the four weeks of project development; Table 4.3 summarizes the questionnaire responses. Overall, more than 94% of the 105 developers had to deal with some merge conflicts, but only a minority experienced them very frequently.

<table>
<thead>
<tr>
<th># of conflicts</th>
<th># of responses</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>6.7</td>
</tr>
<tr>
<td>1–4</td>
<td>61</td>
<td>58.1</td>
</tr>
<tr>
<td>5–9</td>
<td>19</td>
<td>18.1</td>
</tr>
<tr>
<td>≥ 10</td>
<td>18</td>
<td>17.1</td>
</tr>
</tbody>
</table>

Table 4.3: Number of merge conflicts encountered during a four-week development period.

Significance. To get an idea of the severity of merge conflicts, we asked how much time developers spent to resolve a single conflict, on average and in the worst case; Table 4.4 summarizes the questionnaire responses. Conflicts do not normally seem to take a lot of time to resolve: nearly 70% of the developers spent no more than 10 minutes to deal with a conflict on average. However, the worst cases of conflict resolution can be substantially more time
4.3. RESULTS OF THE EMPIRICAL STUDY

Consuming.

<table>
<thead>
<tr>
<th></th>
<th>average case</th>
<th>worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td>time (min)</td>
<td>responses</td>
<td>%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>≤ 1</td>
<td>4</td>
<td>4.3</td>
</tr>
<tr>
<td>1–5</td>
<td>28</td>
<td>30.1</td>
</tr>
<tr>
<td>5–10</td>
<td>33</td>
<td>35.5</td>
</tr>
<tr>
<td>10–20</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>≥ 20</td>
<td>15</td>
<td>16.1</td>
</tr>
</tbody>
</table>

Table 4.4: Average- and worst-case time (in minutes) spent to resolve one merge conflict.

Origin. To estimate whether merge conflicts are more frequently due to the parallel work of co-located or of remote team members, we asked whether “most conflicts” occurred: with local (i.e., co-located) members of your team; with remote members of your team; or with developers of other teams. The last option was meant to indicate cases of unwanted interaction between the work of independent teams, possible since all teams committed into different directories and branches but in the same physical repository. Table 4.5 summarizes the questionnaire responses. The difference between local and remote team members is not large and probably not significant; this may indicate that the likelihood of conflicts depends more directly on other factors than location. More significantly, the majority of responses indicate the work of other teams as the main source of conflicts suggesting that bad practices in the usage of the Git version control system were the norm rather than the exception. Our intent in using a unique repository was to let developers have read access to the work of every other teams, while using the powerful features of Git to manage their work independently. Unfortunately, the data indicates that this didn’t work in practice as expected; the result is consistent with the fact that most developers indicated a below average experience with Git and distributed version control (see Section 4.2.2). Overall, our study did not find major differences between local and remote team members as origins of source conflicts; but the project setup decreases our confidence in this appraisal.

---

4Statistical significance tests do not seem to be applicable: paired tests (such as the Wilcoxon signed-rank test) typically require samples of the same population in different conditions, and unpaired two-sample tests (such as the U test) typically require independent samples. Neither seems justifiably the case in our experiments of local vs. remote: local and remote conditions in each pair are reported by the same person (hence they are likely dependent); the populations of local and of remote are not directly comparable (in our setup, each local group interacts with two remote groups).
Table 4.5: Origin of most conflicts: local vs. remote team members.

<table>
<thead>
<tr>
<th>most conflicts with</th>
<th>responses</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>local team members</td>
<td>30</td>
<td>30.9</td>
</tr>
<tr>
<td>remote team members</td>
<td>22</td>
<td>22.7</td>
</tr>
<tr>
<td>members of other teams</td>
<td>45</td>
<td>46.4</td>
</tr>
</tbody>
</table>

Merge conflicts will occur but are not very frequent; most of them have limited significance, as they take few minutes to resolve. The location of team member does not seem to have a major effect on the likelihood that their work will introduce conflicts.

4.3.2 Workflow Interruptions: Frequency and Origin

Research question [RQ.4] looks for problems related to insufficient awareness of the work of team members. It is hardly possible to measure (lack of) awareness in real-time, since it is a somewhat elusive notion that emerges mostly as an afterthought: for example, in the scenario outlined in Section 4.1, a merge conflict makes Bruno realize that he previously was unaware of Anita’s work.

Given this nature of awareness, we collected indirect data about possible effects of the lack of awareness in terms of interruptions caused to a developer’s workflow: due to lack of documentation about some functionality; due to some expected functionality missing altogether; and due to functionality present but incorrect (that is, behaving differently than documented). The questionnaire asked to estimate the frequencies of these interruptions; and the real-time tracking system recorded these interruptions as they were flagged.

**Frequency.** To get a qualitative picture of how frequently workflow interruptions (as a sign of awareness deficiencies) occur, we asked to assess their frequencies on an ordinal scale from 1 to 7 (where 1 means never and 7 means very often). Table 4.6 summarizes the questionnaire responses, which were broken down according to kind of interruption (missing documentation, missing functionality, and incorrect functionality) as well as to whether the source of interruptions was related to the work of local or of remote team members. Overall, interruptions to one developer’s workflow seem to be roughly proportional to the number of team members he or she interacts with, as each local group collaborated with two remote groups (and all groups were roughly the same size). This entails that the frequency of interruptions becomes significant (but hardly overwhelming) as soon as more than few
people collaborate closely on the same project.

<table>
<thead>
<tr>
<th>interruption</th>
<th>median</th>
<th>min</th>
<th>max</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>documentation</td>
<td>L 2</td>
<td>1</td>
<td>7</td>
<td>2.5</td>
</tr>
<tr>
<td></td>
<td>R 4</td>
<td>1</td>
<td>7</td>
<td>3.7</td>
</tr>
<tr>
<td>functionality</td>
<td>L 2</td>
<td>1</td>
<td>7</td>
<td>2.4</td>
</tr>
<tr>
<td></td>
<td>R 5</td>
<td>1</td>
<td>7</td>
<td>4.4</td>
</tr>
<tr>
<td>incorrect</td>
<td>L 2</td>
<td>1</td>
<td>7</td>
<td>2.3</td>
</tr>
<tr>
<td></td>
<td>R 4</td>
<td>1</td>
<td>7</td>
<td>3.9</td>
</tr>
</tbody>
</table>

Table 4.6: Frequency of workflow interruptions according to the questionnaire responses. There are three kinds of interruptions: due to missing documentation, missing functionality, and incorrect functionality. The source of interruptions can be the work of local (L) or remote (R) team members. Frequencies range over a 1–7 scale.

To quantitatively look at interruption frequency, we asked to tag interruptions in each category in real time. Table 4.7 summarizes the data about the 15 developers who used the real-time tracking tool over a total of 106 sessions, each lasting 2.9 hours on average. Overall, interruptions occurred in 34% of the 106 sessions; assuming that our sample is representative of the actual distribution of interruptions, we expect a workflow interruption every 2.5 hours. These figures are consistent with the qualitative findings, and suggest that interruptions occur with significant frequency for teams of non-trivial size.

<table>
<thead>
<tr>
<th>interruption</th>
<th>#</th>
<th>ratio R/L</th>
</tr>
</thead>
<tbody>
<tr>
<td>documentation</td>
<td>L 9</td>
<td>1.9</td>
</tr>
<tr>
<td></td>
<td>R 17</td>
<td></td>
</tr>
<tr>
<td>functionality</td>
<td>L 25</td>
<td>1.6</td>
</tr>
<tr>
<td></td>
<td>R 39</td>
<td></td>
</tr>
<tr>
<td>incorrect</td>
<td>L 16</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td>R 19</td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td>L 50</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td>R 75</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.7: Number of workflow interruptions recorded in real time by 15 developers working for a total of 307 hours. There are three kinds of interruptions: due to missing documentation, missing functionality, and incorrect functionality. The source of interruptions can be the work of local (L) or remote (R) team members.

**Kind.** The collected data suggests that interruptions due to missing functionality—that is, a functionality described in the requirements but not
Figure 4.3: Frequency of workflow interruptions according to the questionnaire responses. The top row shows interruptions caused by the work of local team members; the bottom row shows interruptions caused by the work of remote team members. Frequencies range over a 1–7 scale.

available in the code base—are the most frequent kind among the three considered (missing documentation, missing functionality, and incorrect functionality). Overall, problems with functionality (missing or incorrect) account for nearly 80% of the interruptions according to the real-time tracking data. This is consistent with the emphasis on the requirements elicitation phase in project development, where we insisted that teams agree on a reasonable requirements specification document before they start the implementation phase.

**Origin.** As we mentioned when discussing interruption frequencies, remote team members are twice as numerous as local team members. Therefore, *if location plays no dominant role in determining interruptions*, it is to be expected that the *origin* of interruptions be the work of remote team members more frequently than the work of local team members. This is what we observe in Table 4.6 where median and means for remote team members are consistently higher than the corresponding measures for local team members. To have a visual confirmation of this difference, Figure 4.3 shows the
distribution of answers summarized in Table 4.6: the distributions for co-located team members are slanted towards low frequencies, whereas those for remote team members are more uniform and include higher frequencies. The quantitative data of Table 4.7 confirms this trend: remote interruptions are overall 50% more numerous than local ones.

If we refine the picture by weighing in additional factors, we can understand the fine-grain differences between local and remote shown in Table 4.7. First, while remote team members are roughly twice as numerous as local ones, the modular architecture of the systems and the matching division of labor were such that local team members interacted more closely on the same piece of code. Second, problems related to functionality (missing or incorrect) determine interruptions that are likely more prominent than missing documentation: in the latter case, one can often resort to exploration and testing to understand the behavior; in contrast, if functionality belonging to another component is missing or incomplete, one probably has to check with the responsible developer what the issue really is. Therefore, we expect the closer interaction associated with local team members to affect more strongly interruptions related to functionality. These observations help explain the data in Table 4.7: location is essentially irrelevant for missing documentation (where an almost perfect 2-to-1 ratio of interruptions is observed); whereas it affects problems related to functionality to some extent, as a result of local team members interacting more closely on the same module.

Workflow interruptions (related to insufficient awareness) occur fairly frequently for teams of non-trivial size. Interruptions due to functionality (missing or incorrect) are more frequent than interruptions due to documentation problems. Developer location seems to play no major role in determining the frequency of interruptions.

4.3.3 Conflicts and Interruptions: Impact and Effects

Research question [RQ.C] asks for the effects of merge conflicts and of insufficient awareness on project development. To address these issues, we asked to estimate the impact of merge conflicts and of workflow interruptions (which we use as a signal of insufficient awareness) on various dimensions of success; and to describe what actions are normally undertaken when workflow is interrupted.

Impact. We asked to assess the negative impact of both merge conflicts and workflow interruptions on one’s productivity, motivation, and keeping to the schedule using a scale of 1 to 7 (where 1 means had no negative
Figure 4.4: Impact of conflicts (top row) and of workflow interruptions (bottom row) on productivity, motivation, and ability to keep to the assigned schedule. Impact ranges over a 1–7 scale, where higher rank means more negative impact.

impact at all and 7 means had very much negative impact). Table 4.8 summarizes the questionnaire’s responses; Figure 4.4 details the distribution of responses. While there seems to be no significant difference between the effects on productivity, motivation, and keeping to the schedule, there is a systematic bias in favor of merge conflicts; that is, the negative impact of merge conflicts is consistently lower than the negative impact of workflow interruption. This is confirmed (with different confidence in different categories) by a Wilcoxon signed-rank test\(^5\) comparing conflicts and interruptions in each category, whose \(p\)-values are given in the last column. This data suggests that interruptions that are ultimately related to awareness problems are more pernicious or however have a broader impact than “run-of-the-mill” merge conflicts.

Effects. To get an idea of how workflow interruptions are dealt with, we asked to indicate one or more actions typically done when the workflow is interrupted (due to lack of awareness). The questionnaire included four

\(^5\)In this case, data in each pair are from comparable populations.
4.3. RESULTS OF THE EMPIRICAL STUDY

<table>
<thead>
<tr>
<th>impact of</th>
<th>on</th>
<th>median</th>
<th>min</th>
<th>max</th>
<th>mean</th>
<th>( p )</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC</td>
<td>productivity</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>3.3</td>
<td>0.001</td>
</tr>
<tr>
<td>WI</td>
<td>productivity</td>
<td>4</td>
<td>1</td>
<td>7</td>
<td>4.0</td>
<td></td>
</tr>
<tr>
<td>MC</td>
<td>motivation</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>3.5</td>
<td>0.08</td>
</tr>
<tr>
<td>WI</td>
<td>motivation</td>
<td>4</td>
<td>1</td>
<td>7</td>
<td>3.8</td>
<td></td>
</tr>
<tr>
<td>MC</td>
<td>schedule</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>3.0</td>
<td>10^{-7}</td>
</tr>
<tr>
<td>WI</td>
<td>schedule</td>
<td>4</td>
<td>1</td>
<td>7</td>
<td>4.3</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.8: Negative impact of merge conflicts (MC) and workflow interruptions (WI) on: productivity, motivation, and keeping to the schedule. Impact ranges over a 1–7 scale.

predefined answers: “I check the repository for a recent commit”; “I contact the responsible developer asynchronously (e.g., via email)” ; “I contact the responsible developer in real time (e.g., via Skype)” ; “I switch to a different task that does not depend on the interruption”; as well as the option to write down an open answer. The open answer option was not used by the respondents; Table 4.9 summarizes the results for the predefined answers. The least popular option is switching to a different task; this indicates that most workflow interruptions require to be dealt with by acquiring missing information and cannot be worked around by adjusting the workflow. This is an indirect confirmation of our assumption that workflow interruptions are often due to missing awareness; and that tools enhancing the awareness of other developers’ work may help reduce disruptive interruptions.

<table>
<thead>
<tr>
<th>action</th>
<th>responses</th>
<th>% of respondents</th>
</tr>
</thead>
<tbody>
<tr>
<td>check repository</td>
<td>67</td>
<td>64.4</td>
</tr>
<tr>
<td>contact via email</td>
<td>52</td>
<td>50.0</td>
</tr>
<tr>
<td>contact via Skype</td>
<td>46</td>
<td>44.2</td>
</tr>
<tr>
<td>switch task</td>
<td>24</td>
<td>23.1</td>
</tr>
</tbody>
</table>

Table 4.9: Actions developers taken when their workflow is interrupted due to missing/incorrect functionality or documentation. Developers could choose more than one option.

Workflow interruptions (related to insufficient awareness) have a more negative impact than merge conflicts on productivity, motivation, and keeping to the schedule. Most interruptions require acquisition of missing information to continue, and thus disrupt the planned workflow.
4.3.4 Awareness: Preferences and Granularity

The answers to \( RQ.C \) in Section 4.3.3 suggest that workflow interruptions are a serious issue and originate in insufficient or missing awareness information. Research question \( RQ.D \) investigates the follow-up problem of whether developers would welcome tools that make awareness information available more often than it is available using standard tools. To answer this question, we first investigate the overall preference for or against awareness information; then we try to figure how frequently and with what detail awareness information should be made available.

**Overall preference.** We asked to assess how helpful having access to information about the work of other developers before they push to the repository would be. Helpfulness ranges on an ordinal scale from 1 to 7 (where 1 means *not helpful at all* and 7 means *very helpful*). Table 4.10 summarizes the answers. Overall, there is some preference but not a very strong one.

<table>
<thead>
<tr>
<th>median</th>
<th>min</th>
<th>max</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>How helpful?</td>
<td>4</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 4.10: How helpful would it be to have awareness information? Usefulness ranges over a 1–7 scale.

**Time granularity.** Given that lack of awareness is a significant source of nuisances and problems (Section 4.3.3), we partly attribute the limited enthusiasm for awareness information to the difficulty of evaluating a fairly abstract idea. Developers have a hard time imagining exactly how awareness information would be displayed and made available, which would determine whether it would be helpful.

To get into more concrete options, we asked to choose one or more preferred “times” when information about other developers in one’s own team should be displayed: in real-time as if having complete access to the other developers’ computers; whenever a developer completes a successful compilation; whenever a developer commits locally; or whenever a developer pushes committed changes to the shared repository. Table 4.11 summarizes the answers. The option of being notified whenever a developer in one’s team successfully completes a version of the system is the most popular one, chosen by over half of the respondents. In contrast, being notified in real time as developers type in their changes is the least popular option. Overall, the responses reinforce the timid preference for awareness information indicated by the previous generic question. They also suggest that real-time awareness can be distracting rather than helpful: the real-time collaboration features
highlighted by web-based IDEs such as those discussed in chapter 5 are probably mostly useful only for practices that require a tight interaction between very few developers (such as remote pair programming), but would not scale to larger teams and more complex interactions.

<table>
<thead>
<tr>
<th>when?</th>
<th>responses</th>
<th>% of respondents</th>
</tr>
</thead>
<tbody>
<tr>
<td>real time</td>
<td>28</td>
<td>26.7</td>
</tr>
<tr>
<td>compile time</td>
<td>57</td>
<td>54.3</td>
</tr>
<tr>
<td>commit time</td>
<td>34</td>
<td>32.4</td>
</tr>
<tr>
<td>push time</td>
<td>39</td>
<td>37.1</td>
</tr>
</tbody>
</table>

Table 4.11: When should awareness information be made available? Developers could choose more than one option.

**Detail granularity.** Having ascertained that awareness information is helpful to have as long as it’s not displayed too frequently, we assessed the orthogonal dimension of “detail”. We asked to choose one or more preferred detail levels for the information about other developers in your team: the “full detail” of all changes and additions to the code base; which routines have been affected (but now how they have been modified); which classes have been affected (but not which routines within those classes); which packages or other modules have been affected (but not which classes within those packages); and no information about the change at all. Table 4.12 summarizes the answers. We observe varied preferences, but with a trend towards more detail. Overall, this suggests that as much information as possible should be available within an awareness tool, but it should also be possible for users to customize what is displayed according to their individual preferences of the moment: one size does not fit all.

<table>
<thead>
<tr>
<th>what?</th>
<th>responses</th>
<th>% of respondents</th>
</tr>
</thead>
<tbody>
<tr>
<td>full detail</td>
<td>37</td>
<td>35.2</td>
</tr>
<tr>
<td>changed routines</td>
<td>32</td>
<td>30.5</td>
</tr>
<tr>
<td>changed classes</td>
<td>25</td>
<td>23.8</td>
</tr>
<tr>
<td>changed packages</td>
<td>5</td>
<td>4.8</td>
</tr>
<tr>
<td>no information</td>
<td>6</td>
<td>5.7</td>
</tr>
</tbody>
</table>

Table 4.12: What should the detail of awareness information be? Developers could choose more than one option.
Table 4.13: All variable pairs that correlate (Kendall’s $\tau > 0.5$) with high significance ($p < 10^{-3}$).

Most developers prefer awareness information to be available at significant events (such as successful compilation) rather than in real time. Different developers have different preferences regarding the level of detail that awareness information should have.

4.3.5 Correlation Analysis

To look for dependencies between responses, we performed a correlation analysis between all pairs of variables corresponding to the answers to the questionnaire, including those for the self-assessment of the respondents’ experience with programming and collaboration (see Section 4.2.2). Table 4.13 shows all variable pairs that exhibit a significant correlation.

The correlations are not particularly surprising in hindsight. For example, the correlations between the number of interruptions due to incorrect and missing functionality (both with local and with remote team members) reflect the fact that well-coordinated teams (with well-understood requirements) will have fewer problems with both incorrect and missing functionality, whereas badly-coordinated teams encounter problems with both. On the other hand, missing correlations bring more insight. Specifically, the lack of significant correlations with variables measuring the level of experience with distributed version control and other techniques for collaborative software development suggests that the findings distilled from the questionnaire are not a direct result of the background and pre-existing skills of the respondents. This reflects positively on the generalizability of our findings.
4.4 Threats to Validity

Internal validity. To minimize threats to internal validity, we manually went through all data (105 questionnaire responses, and 106 real-time sessions from 15 subjects) looking for obviously inconsistent or bogus information. For example, we discarded the data of respondents that declared that they “never experienced conflicts” and took “over 20 minutes to resolve conflicts on average”; and very short tracking sessions that increased and decreased the interruption counts without sensible pattern. The fact that only 8% of the data points had to be discarded suggests a fundamentally good quality of the sanitized data. We also reran our analysis on data that only contained the top-50 developers (in number of commits). All the qualitative findings applied to this subset as well. This gives us confidence that our findings are largely independent of the development time invested by participants.

External validity. The major threat to generalizability comes from the majority of participants’ low level of experience with Git, which is something not to be expected of experienced professional developers. Nevertheless, in spite of their limited experience, our study participants did not find merge conflicts an overall dominant problem; therefore, it is reasonable to speculate that experienced developers would strengthen our findings about the severity of workflow interruptions. Another threat comes from only involving student developers, albeit at the master’s level. The fact that the results showed no correlation linking developers’ experience to the significant measured variables indicates, however, that this threat is unlikely to be severe.

4.5 Related Work

Other challenges and issues in globally distributed development have been investigated empirically; for example, the effect of time zones [53, 50, 75]; the role of development processes [36]; the impact on productivity and quality [92, 14]; the usage of contracts [78]; and the role of dispersion [93].

There is a broad body of research about awareness in distributed software development [109], mostly targeting aspects related to project management—orthogonal to those present in this chapter which focuses on the practical impact on development activities. [2, 58], for example, focus on high-level aspects such as personal information or level of expertise; similarly, [28] shows the importance for developers of information about requirements, planning, and project status.

Research on awareness and coordination tools dates back for more than two decades [29]. More recently, however, researchers proposed a number of
tools which have a particular focus on software development (more details listed in chapter 5). The research on these development tools is often accompanied by experiments to test the tools and compare them against traditional version control systems. These experiments [50, 34, 94] normally involve small (2 or 3 people) teams working on controlled artificial exercises (such as refactoring) that can be carried out in a short programming session. In other cases [12], the experiments used data from open-source software repositories to estimate the effectiveness of their conflict detection mechanisms. The present study complements both kinds of experiments by targeting real software development over a considerable time span, beyond the information stored by version control systems, and independent of specific solutions (as they were implemented in the evaluated tools).

4.6 Summary

This chapter presented an empirical study of the significance of merge conflicts and awareness-related problems in distributed software development. To achieve general findings independent of specific tool protocols, we studied interruptions to the workflow as proxies for situations of insufficient awareness. The findings confirm the benefits of improving awareness information; suggest that different developers often have different preferences regarding the frequency and detail that awareness information should have; and do not indicate developer location as a major factor in determining merge conflicts or awareness problems.
Chapter 5

A Tool for Conflicts and Awareness

5.1 Introduction

Software development is overwhelmingly a group activity. Sure, innovative software products may still be conceived by—or even require—the stroke of genius of a solitary demiurge; but, if they eventually become widely used and successful, it is only through the contributions of multiple developers over several years.

With collaborative development becoming the norm, a number of standard processes and tools have emerged that support multiple developers working on the same codebase. Integrated Development Environments (IDEs) and Software Configuration Management systems (SCMs) have thus become the software developer’s central tools. Traditional IDEs are essentially personal tools, where every member of a project works on a local copy of the software under development and periodically undergoes a process of synchronization with the other members using the functionalities offered by the SCM.

The development paradigm embodied by the combination of IDEs and SCMs has been incrementally refined and has successfully scaled up to very large development efforts. Its fundamental structure and mode of operation, however, have not changed significantly since their introduction, and its shortcomings are becoming more evident as the magnitude of software development efforts is ever increasing. Since each developer works off-line on a local copy of the codebase, conflicts between her changes and another developer’s may emerge. Conflicts complicate and slow down collaborative development, because they require resolution: an often painful process of
The software engineering research community is well aware of these shortcomings and, in response, has proposed a number of advanced techniques to support conflict detection as soon as possible, and to improve every developer’s awareness of what his colleagues are doing to the codebase that may affect him. Conflict detection and awareness can both be instrumental in reducing the likelihood and severity of conflicts. The major limitation of the existing approaches (reviewed in Section 5.2) is that they have been conceived and implemented in isolation: conflict detection systems remain centered around the notion of conflict and resolution without actively promoting conflict avoidance; awareness systems are typically oblivious of the abstractions used by the SCMs, and hence are of limited help to resolve conflicts when they cannot be avoided.

This chapter describes CloudStudio, our proposal for a collaborative development framework where the software configuration management, conflict detection, and awareness systems are unitarily conceived and tightly integrated. CloudStudio’s configuration management system continuously operates in the background to automatically share every developer’s changes with everybody else. On top of this, the real-time awareness system lets each user decide to selectively display, notify, or hide the changes introduced by others. The whole development environment is aware of the current view, and can compile, execute, and debug the project accordingly. This tight integration of different features makes it possible to synergically avail their combined benefits: direct conflicts are prevented in most situations, but at the same time a developer’s work need not block others. Real conflicts occur only when a developer deliberately decides to branch out a new version of the code independent of the others’ work.

CloudStudio is also the name of a web-based IDE prototype (Figure 5.1) that we have implemented to demonstrate the ideas of the CloudStudio framework. You can try it out online at cloudstudio.ethz.ch. Using this prototype, we have conducted a case study where three teams of two programmers worked on collaborative development tasks either with CloudStudio or with traditional a IDE and SCM (EiffelStudio and Subversion). Within the limits given by its scope, the case study substantiates our claims that the CloudStudio framework can facilitate collaborative development without interfering with the habitual practices of programmers.

The main contributions of the chapter are as follows:

- A novel software configuration management model that automatically maintains multiple synchronized versions of the codebase integrating analysis and coordination to produce a unique consistent version that merges the conflicting views.
the changes of different developers. This facilitates conflict prevention even in situations where multiple developers work closely on the same portion of code. The software configuration management model does not rely on the details of any specific SCM and can be applied to existing repositories.\footnote{In the current prototype, it is implemented on top of Git.}

- A real-time awareness system that supports multiple views on the project, including or excluding other developers’ changes, and unobtrusively making programmers aware of each other’s work before conflicts occur.

- A prototype implementation of the CloudStudio collaborative development framework into a publicly available web-based IDE.

- A case study that gives preliminary evidence of the advantages brought by the integrated CloudStudio approach.

5.2 Related Work

Several proposals for awareness and advanced conflict detection systems have been put forward in the last decade, sharing the same goals of facilitating collaborative development and improving over the standard practices based on traditional SCMs. This section discusses the main features of these systems, with focus on those that are directly relevant for the CloudStudio framework discussed in the rest of this chapter; Table 5.1 gives a synoptic overview. The distinction between awareness and conflict detection is not sharp as most systems include some of both features; it is, however, useful to highlight the focus of each approach and to discuss how CloudStudio targets the tight integration of these two naturally related aspects. Since CloudStudio is currently available as a web-based IDE, we also discuss some outstanding examples of IDEs for collaborative work (Collabode \cite{Collabode} and Cloud9 \cite{Cloud9})—even though the innovations of CloudStudio are in the underlying collaboration model rather than in its specific implementation in a web-based IDE.

The standard approach to software configuration management (implemented by tools such as CVS and Subversion) uses a client/server architecture with a central repository and local working copies with every developer. Synchronization between developers takes place indirectly through the central repository by explicit request of the clients: a client’s commit operation propagates her local changes upward into the central repository; a client’s
Figure 5.1: Main CloudStudio user interface, showing the integration of the awareness and conflict detection mechanism (balloons are not part of the user interface but of this figure’s caption).

update operation copies the central repository’s content downward to her local copy. Whenever the local and central content diverge in irreconcilable ways, there is a conflict that must be addressed by merging the two different copies. While committing becomes a local operation with distributed SCMs (such as Git and Mercurial), where every client maintains a complete commit history of her changes, this does not makes conflicts less likely to occur, nor obviates the need for merging whenever a developer periodically pushes her local changes to the other team members’ repositories.

This mode of operation can introduce two kinds of conflicts: direct conflicts—two users editing the same piece of code in different ways—and indirect conflicts—the edits of a user in a portion of the code break another portion of the code written by another user, such as when changing a method signature’s requires the clients to change their invocations.

Awareness systems. address the problem of conflicts by allowing each
Table 5.1: Main features of awareness systems and conflict detection frameworks.
For each system, we report: whether it supports detection of direct conflicts and of indirect conflicts; whether conflict reports may include false positives; whether conflicts are available in real-time or upon commit; the granularity of the awareness system (line, class, branch, and whether it is customizable); whether collaborative editing supports shared sessions a la Google Doc and automatic merging of versions; whether there are mechanisms to transfer the changes of one user to another; and to compile the version of the project under the current view; and the main limitations of the approach.
developer to see the changes introduced by others and what artifacts have been affected by the changes. If developer $u$ can see that developer $v$ has changed a line, $u$ is aware that modifying the line may introduce a conflict, and that he should coordinate with $v$ to avoid it. Significant examples of awareness systems are FastDash [9], Palantir [94, 95], Syde [63], and CollabVS [50].

Awareness mostly targets direct conflicts, but only simple indirect conflicts such as changes to method signatures; this is the case of Palantir and CollabVS, whereas FastDash and Syde do not detect indirect conflicts at all. Since awareness systems are normally not integrated with the SCM, they fail to provide good support to resolve and merge conflicting versions when conflicts do occur. Exceptions are CollabVS and Syde, but their support for inspection of conflicting versions is comparable to textual diffs like those offered by traditional SCM.

Advanced conflict detection. Following the principle that problems are easier to fix the sooner they are discovered, conflict detection and conflict prediction systems monitor the activities of all members of a development team searching for causes of conflict and reporting them to the interested users as soon as possible. Tools such as Crystal [12] and WeCode [49] work by continuously (and transparently) trying to merge the local copies of the various users; whenever a speculative merge fails, it signals that a conflict may occur in the future. This mode of operation covers one of the main weaknesses of awareness systems: the detection of complex indirect conflicts, since the speculative merge may include compilation or even regression testing.

The evaluation of Crystal [12] shows that this information is quite useful to anticipate problems during development and to make merging less painful. On the other hand, since conflict detection systems are not fully integrated with the rest of the development environment, they do not offer flexible ways to inspect the work of others when conflicts are detected in order to facilitate resolution. A related issue is that conflict detection works only on stored or committed files, but user activity in-between such operations is invisible to others.

IDEs for collaboration. A diversity of tools are used to simplify collaboration among distributed teams, including some commercial products such as IBM’s Jazz [18] and Microsoft’s Team Foundation [41]; a complete review is beyond the scope of this work.

Recognizing the centrality of the IDE among development tools, tools such as CodeRun [21] have brought IDEs to the web. Using these tools requires no software installation or configuration but only a browser. Other features, however, are replicated as in traditional IDEs: every developer
works on a different copy of the code, stored on a server, and SCM follows traditional practices. Thus, IDEs have followed the industry’s trend to move to the web, but they do not address awareness or merge conflicts in any specific way.

In a few cases, for example Cloud9 and Collabode, web-based IDEs support collaborative development through real-time code sharing: developers can simultaneously work on the same piece of code with the same view, as if they were editing a GoogleDoc shared document. Such an unrestricted form of collaboration is, however, useful only in certain circumstances where direct tight collaboration is required, such as in pair programming practices; during general development practices it is instead necessary to follow a more disciplined approach that integrates with standard SCMs.

5.3 A Session with CloudStudio

This section gives an overview of the CloudStudio framework from the perspective of two users—Claudia and Stu—who are working on the same project using the CloudStudio web-based IDE. The use case scenario is shown in Figure 5.2, to which the following description repeatedly refers.

After logging in on cloudstudio.ethz.ch and selecting a project, Claudia (rightmost column in Figure 5.2) starts working on a class PARAGRAPH. CloudStudio displays the class current base version as plain text in Claudia’s editor (C1).

Claudia is editing class PARAGRAPH concurrently with Stu, who is working at a different location. At any time, Claudia can show or hide Stu’s changes to the code by toggling a button. When changes are shown, vertical bars of different colors mark each line of code according to its edit status: blue for lines changed or added by the current user; orange for lines changed or added by others; lines without a colored bar are unchanged by anyone. Claudia starts modifying class PARAGRAPH by adding a method set_font_size (C2 and C3), whose code is marked in blue in her editor.

In the meanwhile, Stu (leftmost column in Figure 5.2) starts editing the same PARAGRAPH class. Stu’s setup is using another visualization option offered by CloudStudio: it displays only the locations of Claudia’s current changes (marked with red arrow tips), but not the actual content of her changes. Stu notices that the “to do” comment line is marked (S1) and realizes that Claudia has modified that line. Stu switches view to see exactly Claudia’s work (the implementation of set_font_size), which now appears marked in orange in his editor (S2). When Stu compiles the project, he can target the base version of the code (only unchanged lines), or include his or Claudia’s
Figure 5.2: A scenario demonstrating how the CloudStudio framework supports collaborative development.
Orange marks lines modified by others; blue lines modified by self; red arrow tips lines modified but not shown.
changes, or both. This mechanism makes Claudia and Stu aware of each other’s work; they do not have to block and immediately resolve conflicts, but they can continue working without stomping on each other’s feet.

Fully aware of Claudia’s concurrent editing, Stu does some light refactoring, consisting of renaming attribute size to font.size (S3). Claudia is aware of the change, because attribute size’s line becomes marked by a red arrow in her editor (C4). She decides to fully display Stu’s changes (C5), so that she can check that Stu has diligently modified the body of set.font.size consistently with the refactoring.

At this point, Claudia and Stu continue with their concurrent editing without need for explicit synchronization (C6 and S4); this prevents conflicts during concurrent editing. What they see in their editors at any time is, however, only a real-time view constructed based on their visualization preference. Underlying the awareness system there is a full-fledged software configuration management system that maintains personal development branches for Stu and for Claudia. CloudStudio offers support to automatically synchronize and merge personal branches into the base version.

Claudia “approves” Stu’s latest changes by storing them in her personal branch (C7) and by pushing the final set of changes to the master repository (C8). Since Stu has enabled automatic synchronization, his personal repository gets immediately synchronized with the latest version committed by Claudia through the master repository (S5).

5.4 How CloudStudio Unifies Configuration Management, Conflict Detection, and Awareness

CloudStudio’s software configuration management model combines flexibility and automation, and supports fine-grained conflict prevention and real-time awareness within an environment that facilitates collaborative development. This section describes the main technical details of its configuration management, conflict prevention, and awareness systems.

The CloudStudio framework builds on two fundamental abstractions: tasks and views. A task is a project branch organized around the activities of groups of developers. A view is the version of a project (or a subset thereof) obtained by combining the contributions of multiple developers.
5.4.1 Software Configuration Management Model

The software configuration management model maintains information about files and folders in combination with the task structure, which is used to consistently update the tasks and, by the awareness system, to automatically extract views on user demand. The rest of this subsection describes in some detail these abstractions and how they are implemented in CloudStudio; in order to do that, it first briefly revises standard notions used in SCMs (repositories, branches, and push/pull operations), on top of which CloudStudio’s configuration management model is built.

Repositories

A repository is a collection of revisions (or snapshots) of a software project, consisting of files organized in folders. Repositories are organized in DAG structures: when it is created, a repository only contains an empty root branch. As new revisions are committed, the root grows linearly, until a new branch is created. Branches grow independently of the root until they possibly merge back into it. New branches can also spawn off other branches, thus creating subbranches. Figure 5.3 shows the structure of a repository where the root branch has revisions $r_{0.0}$ to $r_{0.6}$ and two branches originating in $r_{1.0}$ and $r_{2.0}$; the first branch has a subbranch at revision $r_{1.1.0}$ and the second branch merges back into the root at revision $r_{0.6}$.

![Diagram of a repository with branches and subbranches.]

Synchronization between repositories occurs via push and pull operations, usually through a master repository. The push of a branch $b$ from a repository $R$ takes the content of $b$ in $R$ and merges it with the content of $b$ in the master; if no conflicts occur, this corresponds to copying and appending $b$ into the master. Conversely, the pull of a branch $b$ from a repository $R$ takes the content of $b$ in the master and merges it with the content of $b$ in $R$. 
5.4. HOW CLOUDSTUDIO UNIFIES CONFIGURATION MANAGEMENT, CONFLICT DETECTION, AND AWARENESS

Tasks and Subtasks

Every CloudStudio project maintains a master repository plus personal repositories for each developer. Developers can work on the predefined root task, present in the master as well as in every repository and conventionally denoted $T_0$, or create new tasks. A task corresponds to a branch managed according to the synchronization policy of CloudStudio, which provides seamless and consistent synchronization among branches. Whenever a user $u$ creates a new task $T$, both $u$’s repository and the master spawn off a new branch for $T$; if, later, another user $v$ joins task $T$, $v$’s repository is updated with a branch for $T$ as well. Tasks can also spawn subtasks, which are implemented as subbranches in the repositories. Figure 5.4 shows a CloudStudio project involving three developers—Claudia, Stu, and Ann—and three tasks—$T_1$, $T_2$, and $T_3$; $T_1$ involves Claudia and Stu, $T_2$ involves Stu and Ann, and $T_3$ involves Claudia and Ann; the root task $T_0$ is shared by everyone as usual.

Line Change Model

CloudStudio manages tasks according to a model of the changes introduced by the developers working on each task. The change model is fine-grained as it logs changes to individual lines in every source file. For each line of every source file, CloudStudio stores a line tree of changes such as those in Figure 5.5. Line trees are ordered sequentially, according to how lines follow one another in the repository’s files.

The root node of a tree initially stores the line in the latest version that all developers have pulled into their personal repositories; this is the base version initially shared by all developers on the project. Figure 5.5(a) shows the base version of a line with the assignment instruction $\text{sum} := 3$. The line is annotated with the tag “$M: T_0$”, denoting that the line is also stored in the master repository $M$ and belongs to the root task $T_0$.

When a developer modifies a line, the corresponding line tree gets extended with a new node that stores the changed line and who did the change. The root node, instead, still stores the version that all “other” users—that...
is, all users not mentioned in other nodes in the tree—have. If Stu initializes $\sum$ to 1 instead of 3, the line tree becomes as in Figure 5.5(b), where the new node is tagged “Stu: $T_0$” because it represents Stu’s editing the root task.

The details of how a new node is added to a line tree depend on the awareness level of the developer introducing the change. Suppose developer $u$ did the latest change to a line $\ell$; CloudStudio stored $u$’s change in a node $N_u$ in $\ell$’s line tree. If another developer $v$ modifies $\ell$ again, the new node $N_v$ with $v$’s changed line is added as a child of $N_u$ if and only if $v$’s awareness system is showing $u$’s changes in real-time. In this case, there is no need to create a conflict because $v$ is aware of $u$’s work, and therefore we can expect that its own changes are consistent with and incremental over $u$’s. Figure 5.5(c) shows an example of this where Ann modifies the assignment to $\sum$ again but she is aware of Stu’s changes. In contrast, if $v$’s awareness system is configured not to show $u$’s changes in real-time, $N_v$ is added as a sibling of $N_u$: $v$ will be able to coordinate with $u$ only later, but for the moment the two activities are separate. This is what happens in Figure 5.5(d), where we assume that, unlike Ann, Claudia is not displaying Stu’s changes in real time.

If $v$ is displaying $u$’s changes in real-time but still prefers to branch off, it can either disable awareness of $u$, or switch to another task $T'$ thus forcing the node $N_v$ (with tag “$v$: $T'$”) to become a sibling of $N_u$ and to start a new series of independent changes. This is the case of Figure 5.5(e), where Ann spawns a new task $T_1$ to accommodate her change even if she is displaying Stu’s changes.

**Task Synchronization**

CloudStudio uses the change model to organize the pushes of changes in tasks, in a way consistent with the information on awareness used in constructing the line trees to minimize conflicts.

To present the synchronization in some detail, we need the notion of **master ancestor**: the master ancestor of a node $N$ for task $T$ in a line tree is the ancestor node $N_0$ of $N$’s such that $N_0$ has the line version stored in the master repository $M$ for task $T$; in other words, the unique path from $N_0$ to $N$ does not contain any node with tag “$M$: $T$” other than $N_0$. Suppose a node $N_u$ stores a developer $u$’s latest change $\ell_u$ to a line $\ell$ on task $T$, and let $N_0$ be the master ancestor of $N_u$ for $T$ in $\ell$’s line tree. Consider the subtree $\tau$ of $\ell$’s line tree rooted at $N_0$; and let $\tau_T$ be $\tau$ with all nodes not tagged with task $T$ pruned. If $\tau_T$ is a linear sequence of nodes (every node has exactly one child), then $N_u$ is **conflict free**. In this case, if developer $u$ pushes its edits to $\ell$, CloudStudio enforces a sequence of pushes, one for each
In this section so far, the presentation has focused on integration of the awareness system and configuration management to automatically control the synchronization of versioning, and views, the CloudStudio framework includes mechanisms to automatically control the uplift of options to trigger synchronization between developers automatically following a compilation event. When enabled, automatic synchronization.
node in $\tau_T$ in that order, and collapses the branch $\tau_T$ by replacing $N_0$ with the node $(\ell_u; M, u, U; T)$\(^2\) while discarding the rest of $\tau_T$ until $N_u$ included. All users other than $u$ tagged in $\tau_T$ are aware of $u$’s latest change, therefore CloudStudio notifies them and updates their personal repositories to coincide with the master on task $T$. The details of the notification are implementation dependent and customizable; for example, users may be asked to approve the push or may be lazily notified only when they want to edit line $\ell$ again.

Continuing the examples of Figure 5.5 consider again the line tree in Figure 5.5(c). The node with $\text{sum} := i$ is conflict free; if Stu pushes his changes, the line tree becomes as in Figure 5.5(f), where the node with $\text{sum} := i + j$ is also conflict free. If Ann pushes next, the tree becomes as in Figure 5.5(g); Stu is aware of Ann’s changes, which have been automatically pulled into Stu’s personal repository. The line tree ends up as in Figure 5.5(g) also if Ann pushes first (from the setup in Figure 5.5(c)). Using standard configuration management models, where there is no notion of who’s aware of whom and whose changes can be merged without rising a conflict, a similar situation would force a conflict.

Branches in $\tau$ tagged with tasks other than $T$ are joined with the collapsed $\tau_T$ so as to preserve the original information (this requires duplication of nodes in some cases, whose details are straightforward). If $N_u$ is not conflict free, CloudStudio cannot push $u$’s edits to $\ell$ automatically; in this case, a conflict is unavoidable, and the interested users are notified and required to resolve it before pushing is possible. They can do so by turning on awareness of each other’s work, and then agreeing on a conflict free version of the line in question.

5.4.2 Real-time Awareness System

CloudStudio’s awareness system displays the content of the project files based on the information in the software configuration management model (in particular, line trees) and displays it according to user preferences. A view is the version of the project determined by the current user preferences.

In the basic view, the editor shows the current user’s edits, and, for each line not modified locally since the last pull, its base version as stored in the root node if its tree. On top of this, CloudStudio provides options to display the changes introduced by other developers in the current view (Figure 5.6 shows how the user interface of the CloudStudio IDE). The current user can select any other developer $v$ and choose to:

- display all changes introduced by $v$ in real time;

\(^2\)U denotes the (possibly empty) list of all other users tagged in $\tau_T$. 
5.4. HOW CLOUDSTUDIO UNIFIES CONFIGURATION MANAGEMENT, CONFLICT DETECTION, AND AWARENESS

Figure 5.6: Details of the CloudStudio IDE awareness system (left to right): hovering over the color-marker of a line shows tool-tip information about the changes to that line by other users, additional lines currently not displayed are indicated by a yellow triangle between line numbers; User can selected which changes of others’ should be displayed; Compilation be include or exclude the changes of other developers.

- display all changes introduced by \( v \) up to the latest successful compilation;

- display where \( v \) introduced changes but do not show them;

- do not display changes by \( v \) at all;

- display all changes introduced in the current task by any other user.

Unlike most related work—where only committed changes are available to the awareness system [12, 45]—CloudStudio updates the information about changes in real-time and makes it available to all users.

CloudStudio’s real-time awareness system integrates with the rest of the development environment through views. The compiler and every other tool working on the project files (such as debuggers or testing environments) have access to the project in the current view, even if the underlying changes have not been pushed to the personal or master repositories yet. This makes for a seamless integration of the awareness system and software configuration management within the developers’ overall activity.

\(^3\)To extract the changes up to the latest successful compilation, all nodes in the line tree that denote the current version are tagged after every successful compilation. This is a straightforward extension of the line tree model discussed previously.
5.4.3 Collaborative Editing

On top of the mechanisms supporting tasks, fine-grained versioning, and views, the CloudStudio framework includes functionality to automatically control the synchronization of tasks between developers and facilitate collaborative editing.

Using the awareness system, it is possible to import code from one user’s personal repository to another’s and to start collaborating. CloudStudio supports the importing function that clones a portion of code (such as a method or a whole class) from a user u’s personal repository to another user v’s and turns on u and v’s awareness systems so that they can work together on the code by relying on the conflict prevention mechanisms described above.

The CloudStudio development environment includes a set of options to trigger synchronization between developers automatically following a compilation event. When enabled, automatic synchronization triggers the following actions whenever a user u saves and successfully compiles the project under its current view:

1. it pushes u’s current task to the master repository;
2. it pulls these changes to the personal repositories of all other users collaborating with u (and who have enabled automatic synchronization).

5.4.4 Conflict Detection and Prevention

In this section so far, the presentation has focused on direct conflicts, also known as syntactic conflicts, which occur when two developers modify the same line. CloudStudio’s tight integration of awareness system and configuration management prevents direct conflicts by letting developers see each other’s work in collaborative editing and by branching out when they deliberately decide to work in parallel ignoring each other.

In contrast, indirect conflicts, also known as semantic conflicts, occur when a developer’s change to a portion of the code breaks the dependency with the work of another developer in another portion of the code. For example, changing a method’s signature by adding a new argument introduces an indirect conflict in all clients that invoke the method anywhere in the code, which have to change their calls to conform to the new signature. CloudStudio supports indirect conflict detection through views. Users can compile the project under the current view, thus getting an error if any of the other users’ changes included in the view has introduced an indirect conflict. The option to include in the view only the changes that can be successfully compiled adds another degree of flexibility, where developers collaborate on “stable”
versions but are still free to experiment changes on their own that may break compilation. In all such cases, CloudStudio’s real-time awareness features can be useful to let two developers collaborate with the goal of removing an indirect conflict that involve their work.

5.4.5 Limitations of CloudStudio’s SCM

CloudStudio’s change model works at the level of individual lines of code. This achieves language-independence but also does not take the structure and semantics of the code into account [63]; sound refactorings, in particular, may still be considered conflicting. While CloudStudio mainly aims at preventing conflicts and supports compilation using other developers’ changes, a mechanism for conflict detection based on speculative compilation [13] could still provide an additional dimension of automation. We will consider these extensions in future work.

5.5 CloudStudio’s Prototype Implementation

The CloudStudio web-based IDE is a prototype implementing the CloudStudio framework described in Section 5.4. The prototype is freely available at cloudstudio.ethz.ch since it is web-based, using it does not require downloading any software. The implementation combines an editor written in Java using the Google Web Toolkit v. 2.3, and leverages a MySQL database and the Git SCM as back-ends.

CloudStudio currently supports development of projects in Java, JavaScript and Eiffel, but its architecture is extensible to other programming languages. Besides the innovative integration of the configuration management and real-time awareness system described in Section 5.3, CloudStudio offers some of the basic functionalities of traditional IDEs such as VisualStudio or Eclipse: an editor with syntax highlighting, a class browser to navigate the project, integration with the compiler, minimal support for execution, and a debugging environment (currently available only for JavaScript projects).

The awareness system uses the color code described in Section 5.3 to highlight lines changed in real-time by other users that have been selected by the current user. If the awareness system is configured to display change locations without showing the actual changes, red arrows in the margin mark the location of changes. Hovering over a colored bar or an arrow shows the user who has introduced the change, as well as different versions of the line as edited by other users active on the same task.

In continuity with our related work on formal verification [101], the
CloudStudio IDE also integrates verification tools to help developers improve software quality. It currently supports testing with the AutoTest framework \cite{AutoTest}, and formal correctness proofs with AutoProof \cite{AutoProof}. AutoTest performs random testing of object-oriented programs with contracts, and it has proved extremely effective in detecting hundreds of errors in production software; AutoProof provides a static verification environment for Eiffel. Both tools are fully automatic and integrated with CloudStudio’s SCM system: testing and proving sessions work on the current view selected by CloudStudio users, which flexibly may or may not include concurrent edits by other developers (as described in Section 5.4).

5.5.1 Limitations of Current Implementation

The current CloudStudio prototype implementation has a number of limitations, which restrict its used for real-world applications. In particular, the IDE still misses user interfaces to easily compare and merge different versions of code for those were manual conflicting resolution is still needed or desired. Furthermore, due to the nature of its prototype status, the current implementation is not optimized for scalability and the performance suffers if larger numbers of users access the server concurrently.

5.6 Case Study

In order to have a preliminary assessment of the CloudStudio framework and its advantages for collaborative development over traditional IDEs and SCM techniques, this section presents a case study of two-programmer teams working on collaborative development tasks. While small in extension, the case study provides preliminary evidence that CloudStudio can improve the performance of programmers working collaboratively.

5.6.1 Development Tasks

The case study included three program development tasks, two focused on refactoring and one on testing; all applications were written in Eiffel.

\textbf{R1:} Task R1 targets an application implementing a card game (the card deck and the game logic); the complete application includes 210 lines of code over 4 classes. Task R1 requires refactoring of three classes, and development of new functionalities by extending the refactored classes; the task is collaborative because the new functionalities must work with the classes after refactoring. Refactoring included: method and field
renaming; enforcement of Eiffel coding standards (e.g., capitalization, comments); re-arrangement of methods in groups (marked by the *feature* Eiffel keyword) according to their functionalities; code extraction into a new class.

**R2:** Task R2 targets an application modeling a coffee vending machine; users of the application have basic options to select coffee and can pay and receive change. The application includes 230 lines of code over 3 classes. Task R2 is similar to R1 except that it targets the coffee machine application: R2 requires refactoring and development of new functionalities by extending the refactored classes.

**T1:** Task T1 targets the same coffee machine application as task R2. It requires development of new functionalities (namely, the option to add milk to the coffee, and the dispatching of different cup sizes) and writing of test cases that achieve 100% code coverage on the new code. Task T1 is also inherently collaborative as the development of new functionalities and of test cases occur concurrently, according to the concept of *test-driven pair programming* \[47\].

### 5.6.2 Subjects and Experimental Setup

The subjects used in the study were six PhD students from our research group. All of them are experienced Eiffel programmers who frequently develop with EiffelStudio and Subversion (SVN) as part of their PhD research; none of them had used CloudStudio before the study, had taken part in its development, or has much experience with collaborative development.

We randomly arranged the six subjects in three pairs: Team1, Team2, Team3. Team1 first performed task R1 with CloudStudio and then task T1 with EiffelStudio and SVN. Team2 first performed task R1 with EiffelStudio and SVN and then task T1 with CloudStudio. Team3 first performed task R1 with CloudStudio and then task R2 with EiffelStudio and SVN.

Each team performed its sessions according to the following protocol. The two team members sat at the opposite corners of a large table with their laptops connected to the network. Before beginning, they were given a brief (5 min.) introduction to the CloudStudio SCM and web-based IDE and were shown how to log-in and what the basics of the SCM system are; reference to the development tasks were not given during this explanations. Subsequently, the team members were each given a sheet of paper with a description of the task they had to perform (the second task was introduced only after completion of the first). The two programmers received identical
Table 5.2: Number of words used by each team to solve a respective task. \textit{CloudStudio} indicates the task was solved using the CloudStudio IDE, \textit{SVN} indicates the task was solved using a regular IDE and a SVN as to reconcile changes.

During the study nobody other than one experimenter and the two programmers was in the room. The programmers were only allowed to use instant messaging to communicate; their position in the room and the experimenter ensured that no other communication channel was available. The experimenter did not interfere with the programmers other than to clarify possible unclear points in the task description (but this was never necessary).

There was no time limit to complete the tasks: each session continued until the current task was completed (the experimenter checked completeness \textit{a posteriori} by manual inspection of the codebase). After each session, the experimenter recorded the total number of words exchanged via instant messaging and the overall time spent to complete the task. An \textit{a posteriori} analysis of the communication logs, discussed in Section 5.6.4, supports the hypothesis that these two measures (words and time) are reasonable proxies for the actual amount of communication between the two programmers that took place during the experiments.

5.6.3 Results

Table 5.2 reports the amount of communication between programmers while performing the various tasks. While all participants are competent programmers, their speed and development style vary significantly; as a result, the random assignment formed heterogeneous groups which may not be directly comparable. The results in Table 5.2, however, show a consistent advantage for teams using CloudStudio over teams using SVN: the difference is sometimes small (as for task T1), sometimes conspicuous (as for task R1 between Team2 and Team1); in all cases, CloudStudio required less communication for the same task than SVN, even if the study’s programmers used it for the first time. Let us now describe the performance of the various teams in more detail.
When using CloudStudio, all three teams required similar or less communication to complete their programming task compared to using SVN.

Team1 delivered the best overall performance and was fluent both with SVN and with CloudStudio; the two programmers worked well together and required a limited amount of communication to synchronize properly. The comparison with Team2 on the same tasks suggests that using CloudStudio is beneficial: Team1 outperformed Team2 almost by an order of magnitude when using CloudStudio on task R1, whereas their performance became similar on task T1 where Team1 used SVN. It was clear that Team1 was overall faster than Team2, but the peculiarities of task R1 magnified the difference in favor of who could rely on better collaboration tools.

The programmers in Team2 had the greatest communication problems in the study, as shown by their performance in task R1. The log of their message exchanges shows that they had to debate several points of disagreement about how to perform the refactorings, and that not being able to see in real-time what the other was doing (as it happened when working with SVN) exacerbated their disagreement and frustration.

Unlike the members of the other teams, the two programmers in Team3 worked with wildly different speed, to the point that in both tasks R1 and R2 a programmer completed his part of the task when the other was still exploring the system and understanding the instructions. The overall performance of Team3 required little communication in all cases, but this is mostly a result of the fact that the different programmer speed forced a serialization between the two programmers; hence, synchronization was not a big issue because the development was not really collaborative and interactive.

We do not discuss in detail the time taken by programmers because the assignments emphasized correctness of the solution and did not pressure the teams for time. Anyway, and perhaps unsurprisingly, the overall time turned out to be correlated with the amount of communication, and hence all the experimental data point to the same qualitative conclusions.

5.6.4 Discussion

A post mortem analysis of the instant messaging logs shows recurring patterns of communications between programmers. The initial part of every session starts with a discussion of the task, after which the two programmers negotiate a division of the labor and agree on some synchronization mechanism. During development with SVN, messages such as “Did you update your project?” and “I’m done with implementing X and have committed”
are frequent. With CloudStudio, the same messages occurs much more spar-
ingly, and some of the remaining instances can probably  be attributed to
the programmers’ limited familiarity with CloudStudio and how it works (in
fact, in some cases of redundant notification messages using CloudStudio,
the recipient replied with sentences such as “Just go ahead, I can see your
changes live”).

After the case study, we asked the participants to complete a simple
questionnaire about their experience and with requests for feedback. The
participants unanimously appreciated CloudStudio’s mechanisms for the real-
time awareness of other people’s changes, and for the prevention and easy
resolution of conflicts. Disagreement existed on how severe a problem merge
conflicts are in everyday’s software development: four programmers consider
it a serious hassle and appreciate better mechanisms to prevent or manage
conflicts; the other two maintained that merge conflicts can be reduced to a
minimum with a little coordination.

In all, the participants to the study tend to agree with our conclusions
that CloudStudio offers valuable features for collaborative development and
a more flexible paradigm of SCM. The generalizability of our results is neces-
sarily limited by the case study’s scope and size, as well as by its reliance on
specific development tasks that emphasize real-time collaboration but may
affect only a limited part of large software projects. In this sense, the re-
action of one of the programmers in our study to task R1 is instructive:
he was initially skeptical and remarked that he “would never do refactoring
while another programmer is implementing new functionalities”; after using
CloudStudio, however, he acknowledged that, with the right tools, such tasks
can indeed be performed in parallel.

5.7 Summary

We described the CloudStudio framework that integrates software configura-
tion management with real-time awareness to help detect and prevent merge
conflicts in collaborative software development. We implemented the Cloud-
Studio framework in a web-based IDE, and conducted a small case study to
assess its usefulness compared to traditional IDEs and SCMs. Within the
limited scope of the case study, the result are promising and suggest that the
CloudStudio approach can improve how developers carry out collaborative
development tasks.
Chapter 6

Contracts in Practice

6.1 Introduction

In this chapter, we’re studying contracts, a kind of lightweight formal specification in the form of executable assertions (preconditions, postconditions, and class invariants). Using specifications as an integral part of the software development process has long been advocated by formal methods pioneers. With contracts, developers are not only able to automatically test and verify their programs but they also have a form of unambiguous API documentation. Researchers have argued that distributed teams should use contracts over non-formal specifications to avoid misunderstandings of requirements [78]. The work presented here takes a closer look at how contracts are used in practice.

Despite verification and documentation, contracts are particularly interesting for awareness tools, i.e. tools that inform developers about the changes performed by other team members. An important requirement of an awareness tool is that it must not bring every type of change to a developer’s attention but only those changes that are relevant. Otherwise, many false positives—notifications about changes that a developer is not interested in, e.g. small changes to a method’s implementation that does not affect its behavior—make an awareness system impractical and creates a cognitive overload for the developer. A common trade-off used by awareness tools is to inform developers about changes at the API level: whenever a method signature or class interface changes, all developers using that method or class get notified; any finer-grained changes are, however, not reported. This trade-off has the disadvantage of false-negatives where a change might not modify a signature but the behavior of the changed methods. Contracts allow developers to do exactly that: specify the intended behavior regardless
of the actually implementation. Thus, it seems reasonable that awareness tools should monitor APIs at the level of contracts, assuming that (a) developers actually write sufficiently many contracts and (b) contracts change less frequent than implementations. The present work investigates if these assumptions hold.

The empirical study of this chapter analyzes 21 projects written in Eiffel, C#, and Java, three major object-oriented languages supporting contracts, with the goal of studying how formal specifications are written, changed, and maintained as part of general software development. Eiffel has always supported contracts natively; the Java Modeling Language (JML [65]) extends Java with contracts written as comments; and C# has recently added support with the Code Contracts framework [39]. Overall, our study analyzed more than 260 million lines of code and specification distributed over 7700 revisions. To our knowledge, this is the first extensive study of the practical evolving usage of simple specifications such as contracts over project lifetimes.

The study’s specific questions target various aspects of how contracts are used in practice: Is the usage of contracts quantitatively significant and uniform across the various selected projects? How does it evolve over time? How does it change with the overall project? What kinds of contracts are used more often? What happens to contracts when implementations change? What is the role of inheritance?

The main findings of the study, described in Section 6.3, include:

- The projects in our study make a significant usage of contracts: the percentages of routines and classes with specification is above 33% in the majority of projects.
- The usage of specifications tends to be stable over time, except for the occasional turbulent phases where major refactorings are performed. This suggests that contracts evolve following design changes.
- There is no strong preference for certain kinds of specification elements (preconditions, postconditions, class invariants); but preconditions, when they are used, tend to be larger (have more clauses) than postconditions. This indicates that different specification elements are used for different purposes.
- Specifications are quite stable compared to implementations: a routine’s body may change often, but its contracts will change infrequently. This makes a good case for a fundamental software engineering principle: stable interfaces over changing implementations [83].
- Inheritance does not significantly affect the qualitative findings about specification usage: measures including and excluding inherited contracts tend to correlate. This suggests that the abstraction levels pro-
vided by inheritance and by contracts are largely complementary. As a supplemental contribution, we make all data collected for the study available online as an SQL database image [20]. This provides a treasure trove of data about practically all software projects of significant size publicly available that use contracts.

**Positioning: what this study is not.** The term “specification” has a broad meaning. To avoid misunderstandings, let us mention other practices that might be interesting to investigate, but which are not our target in this chapter. We do not consider formal specifications in forms other than executable contracts. We do not look for formal specifications in generic software projects: it is well-known [82] that the overwhelming majority of software does not come with formal specifications (or any specifications). Instead, we pick our projects among the minority of those actually using contracts, to study how the few adopters use formal specifications in practice. We do not study applications of contracts; but our analysis may serve as a basis to follow-up studies targeting applications. We do not compare different methodologies to design and write contracts; we just observe the results of programming practices.

**Extended version.** For lack of space, we can only present the most important facts; an extended version [31] provides more details on both the analysis and the results.

### 6.2 Study Setup

Our study analyzes contract specifications in Eiffel, C#, and Java, covering a wide range of projects of different sizes and life spans developed by professional programmers and researchers. We use the terms “contract” and “specification” as synonyms.

**Data selection.** We selected 21 open-source projects that use contracts and are available in public repositories. Save for requiring a minimal amount of revisions (at least 30) and contracts (at least 5% of elements in the latest revisions), we included all open-source projects written in Eiffel, C# with CodeContracts, or Java with JML we could find when we performed this research. Table 6.1 lists the projects and, for each of them, the total number of revisions, the life span (AGE, in weeks), the size in lines of code (LOC) at the latest revision, the number of developers involved (i.e., the number of committers to the repository), and a short description.

**Measures.** To support analysis of large amounts of program code in multiple languages, we developed COAT—a “COntract Analysis Tool”. The raw
Table 6.1: List of projects used in the study. "Age" is in weeks, "# LOC" is lines of code.

<table>
<thead>
<tr>
<th>Project Name</th>
<th>Language</th>
<th># LOC</th>
<th>Age</th>
<th># Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutoTest</td>
<td>Eiffel</td>
<td>195</td>
<td>30</td>
<td>65,625</td>
</tr>
<tr>
<td>EiffelBase</td>
<td>Eiffel</td>
<td>106</td>
<td>100</td>
<td>61,922</td>
</tr>
<tr>
<td>EiffelProgramAnalysis</td>
<td>Eiffel</td>
<td>114</td>
<td>8</td>
<td>40,750</td>
</tr>
<tr>
<td>GoboKernel</td>
<td>Eiffel</td>
<td>747</td>
<td>119</td>
<td>53,316</td>
</tr>
<tr>
<td>GoboStructure</td>
<td>Eiffel</td>
<td>716</td>
<td>19</td>
<td>21,941</td>
</tr>
<tr>
<td>GoboTime</td>
<td>Eiffel</td>
<td>524</td>
<td>30</td>
<td>10,840</td>
</tr>
<tr>
<td>GoboUtility</td>
<td>Eiffel</td>
<td>716</td>
<td>7</td>
<td>6,131</td>
</tr>
<tr>
<td>GoboXML</td>
<td>Eiffel</td>
<td>285</td>
<td>13</td>
<td>163,552</td>
</tr>
<tr>
<td>Boogie</td>
<td>C#</td>
<td>108</td>
<td>88</td>
<td>88,284</td>
</tr>
<tr>
<td>CCI</td>
<td>C#</td>
<td>30</td>
<td>29</td>
<td>20,602</td>
</tr>
<tr>
<td>Dafny</td>
<td>C#</td>
<td>106</td>
<td>19</td>
<td>29,700</td>
</tr>
<tr>
<td>LabsFramework</td>
<td>C#</td>
<td>30</td>
<td>1</td>
<td>14,540</td>
</tr>
<tr>
<td>Quickgraph</td>
<td>C#</td>
<td>100</td>
<td>4</td>
<td>40,820</td>
</tr>
<tr>
<td>Rxx</td>
<td>C#</td>
<td>68</td>
<td>49</td>
<td>55,932</td>
</tr>
<tr>
<td>Shweet</td>
<td>C#</td>
<td>7</td>
<td>2</td>
<td>2,352</td>
</tr>
<tr>
<td>DirectVCGen</td>
<td>Java</td>
<td>119</td>
<td>119</td>
<td>13,294</td>
</tr>
<tr>
<td>ESCJava</td>
<td>Java</td>
<td>366</td>
<td>27</td>
<td>73,760</td>
</tr>
<tr>
<td>JavaFE</td>
<td>Java</td>
<td>389</td>
<td>18</td>
<td>35,013</td>
</tr>
<tr>
<td>Logging</td>
<td>Java</td>
<td>106</td>
<td>3</td>
<td>5,963</td>
</tr>
<tr>
<td>RCC</td>
<td>Java</td>
<td>350</td>
<td>7</td>
<td>10,872</td>
</tr>
<tr>
<td>Umbra</td>
<td>Java</td>
<td>169</td>
<td>8</td>
<td>15,538</td>
</tr>
</tbody>
</table>

Total: 7,756 LOC, 6,392 revisions, 830,747 lines of code.
measures produced by COAT include: the number of classes, the number of classes with invariants, the average number of invariant clauses per class, and the number of classes modified compared to the previous revision; the number of routines (public and private), the number of routines with non-empty precondition, with non-empty postcondition, and with non-empty specification (that is, precondition, postcondition, or both), the average number of pre- and postcondition clauses per routine, and the number of routines with modified body compared to the previous revision.

Measuring precisely the strength of a specification (which refers to how constraining it is) is hardly possible as it requires detailed knowledge of the semantics of classes and establishing undecidable properties in general. In our study, we count the number of specification clauses (elements anded, normally on different lines) as a proxy for specification strength. The number of clauses is a measure of size that is interesting in its own right. If some clauses are changed, just counting the clauses may measure strength incorrectly. We have evidence, however, that the error introduced by measuring strengthening in this way is small. We manually inspected 277 changes randomly chosen, and found 11 misclassifications (e.g., strengthening reported as weakening). Following [66, Eq. 5], this implies that, with 95% probability, the errors introduced by our estimate (measuring clauses for strength) involve no more than 7% of the changes.

6.3 How Contracts Are Used

Our study targets the following main questions, addressed in the following subsections.

Q1. Do projects make a significant usage of contracts, and how does usage evolve over time?

Q2. How does the usage of contracts change with projects growing or shrinking in size?

Q3. What kinds of contract elements are used more often?

Q4. What is the typical size and strength of contracts, and how does it change over time?

Q5. Do implementations change more often than their contracts?

Q6. What is the role of inheritance in the way contracts change over time?

\[1\text{We consider all concrete syntactic changes, that is all textual changes.}\]
Table 6.2: Selected plots for projects EiffelBase, AutoTest, ESCJava, and Boogie. Each graph from left to right represents the evolution over successive revisions of: (1) and (2), percentage of routines with precondition (pre in the legend), with postcondition (post), and of classes with invariant (inv); (3), average number of clauses in contracts; (4), number of changes to implementation and specification (body+spec), to implementation only (body only), and change to specification only. When present, a thin gray line plots the total number of routine in the project (scaled). Similar plots for all projects are available [31, 20].
6.3. HOW CONTRACTS ARE USED

Table 6.3 shows the essential quantitative data we discuss for each project; Table 6.2 shows sample plots of the data for four projects. In the rest of the section, we illustrate and summarize the data in Table 6.3 and the plots in Table 6.2 as well as much more data and plots that, for lack of space, are available elsewhere [20, 31].

6.3.1 Writing Contracts

In the majority of projects in our study, developers devoted a considerable part of their programming effort to writing specifications for their code. While we specifically target projects with some specification (and ignore the majority of software that doesn’t use contracts), we observe that most of the projects achieve significant percentages of routines or classes with specification. As shown in column % ROUTINES SPEC of Table 6.3, in 7 of the 21 analyzed projects, on average 50% or more of the public routines have some specification (pre- or postcondition); in 14 projects, 35% or more of the routines have specification; and only 3 projects have small percentages of specified routines (15% or less). Usage of class invariants (column % CLASSES INV in Table 6.3) is more varied but still consistent: in 9 projects, 33% or more of the classes have an invariant; in 10 projects, 12% or less of the classes have an invariant. The standard deviation of these percentages is small for 11 of the 21 projects, compared to the average value over all revisions: the latter is at least five times larger, suggesting that deviations from the average are normally small. Section 6.3.2 gives a quantitative confirmation of this hint about the stability of specification amount over time.

The EiffelBase project—a large standard library used in most Eiffel projects—is a good “average” example of how contracts may materialize over a project’s lifetime. After an initial fast growing phase (see the first plot in Table 6.2), corresponding to a still incipient design that is taking shape, the percentages of routines and classes with specification stabilize around the median values with some fluctuations that—while still significant, as we comment on later—do not affect the overall trend or the average percentage of specified elements. This two-phase development (initial mutability followed by stability) is present in several other projects of comparable size, and is sometimes extreme, such as for Boogie, where there is a widely varying initial phase, followed by a very stable one where the percentages of elements with specification is practically constant around 30%. Analyzing the commit logs around the revisions of greater instability showed that wild variations in the specified elements coincide with major reengineering efforts. For Boogie, the initial project phase coincides with the porting of a parent project written in Spec# (a dialect of C#), and includes frequent alternations of adding and re-
moving code from the repository; after this phase, the percentage of routines and classes with specification stabilizes to a value close to the median.

There are few outlier projects where the percentage of elements with specification is small, not kept consistent throughout the project’s life, or both. Quickgraph, for example, never has more than 4% of classes with an invariant or routines with a postcondition, and its percentage of routines with precondition varies twice between 12% and 21% in about 100 revisions (see complete data in [31]).

In two thirds of the projects, on average 1/3 or more of the routines have some specification (pre- or postconditions).

**Public vs. private routines.** The data analysis focuses on contracts of public routines. To determine whether trends are different for private routines, we visually inspected the plots and computed the correlation coefficient\(^2\) \(\tau\) for the evolution of the percentages of specified public routines against those of private routines. The results suggest to partition the projects into three categories. For the 9 projects in the first category—AutoTest, EiffelBase, Boogie, CCI, Dafny, JavaFE, Logging, RCC and Umbra—the correlation is positive (0.51 \(\leq \tau \leq 0.94\)) and highly significant. The 2 projects in the second category—GoboStructure and Labs—have negative (\(\tau \leq -0.47\)) and also significant correlation. The remaining 10 projects belong to the third category, characterized by correlations small in absolute value, positive or negative, or statistically insignificant. This partitioning seems to correspond to different approaches to interface design and encapsulation: for projects in the first category, public and private routines always receive the same amount of specification throughout the project’s life; projects in the second category show negative correlations that may correspond to changes to the visibility status of a significant fraction of the routines; visual inspection of projects in the third category still suggests positive correlations between public and private routines with specification, but the occasional redesign upheaval reduces the overall value of \(\tau\) or the confidence level. In fact, the confidence level is typically small for projects in the third category; and it is not significant (\(p = 0.418\)) only for EiffelProgramAnalysis which also belongs to the third category. Projects with small correlations tend to be smaller in size with fewer routines and classes; conversely, large projects may require a stricter discipline in defining and specifying the interface and its relations with the private parts, and have to adopt consistent approaches throughout their lives.

\(^2\)All correlation measures in this chapter employ Kendall’s rank correlation coefficient \(\tau\).
6.3. HOW CONTRACTS ARE USED

In roughly half of the projects, the amounts of contracts in public and in private routine correlate; in the other half, correlation vanishes due to redesign changes.

6.3.2 Contracts and Project Size

The correlation between the number of routines or classes with some specification and the total number of routines or classes (with or without specification) is consistently strong and highly significant. Looking at routines, 10 projects exhibit an almost perfect correlation with $\tau > 0.9$ and $p \sim 0$; only 3 projects show medium/low correlations (Labs and Quickgraph with $\tau = 0.48$, and Logging with $\tau = 0.32$) which are however still significant. The outlook for classes is quite similar: the correlation between number of classes with invariants and number of all classes tends to be high. Outliers are the projects Boogie and JavaFE with the smaller correlations $\tau = 0.28$ and $\tau = 0.2$, but visual inspection still suggests that a sizable correlation exists for Boogie (the results for JavaFE are immaterial since it has only few invariants overall). In all, the absolute number of elements with specification is normally synchronized to the overall size of a project, confirming the suggestion of Section 6.3.1 that the percentage of routines and classes with specification is stable over time.

Having established that, in general, specification and project size have similar trends, we can look into finer-grained variations of specifications over time. To estimate the relative effort of writing specifications, we measured the correlation between percentage of specified routines or classes and number of all routines or all classes.

A first large group of projects, almost half of the total whether we look at routines or classes, show weak or negligible correlations ($-0.35 < \tau < 0.35$). In this majority of projects, the relative effort of writing and maintaining specifications evolves largely independently of the project size. Given that the overall trend is towards stable percentages, the high variance often originates from initial stages of the projects when there were few routines or classes in the system and changes can be momentous. GoboKernel and DirectVCGen are specimens of these cases: the percentage of routines with contracts varies wildly in the first 100 revisions when the system is still small and the developers are exploring different design choices and styles.

Another group of 3 projects (AutoTest, Boogie, and Dafny) show strong negative correlations ($\tau < -0.75$) both between percentage of specified routines and number of routines and between percentage of specified classes and number of classes. The usual cross-inspection of plots and commit logs points
to two independent phenomena that account for the negative correlations. The first is the presence of large merges of project branches into the main branch; these give rise to strong irregularities in the absolute and relative amount of specification used, and may reverse or introduce new specification styles and policies that affect the overall trends. As evident in the second plot of Table 6.2, AutoTest epitomizes this phenomenon, with its history clearly partitioned into two parts separated by a large merge at revision 150. The second phenomenon that may account for negative correlations is a sort of “specification fatigue” that kicks in as a project becomes mature and quite large. At that point, there might be diminishing returns for supplying more specification, and so the percentage of elements with specification gracefully decreases while the project grows in size. (This is consistent with Schiller et al.’s suggestion [96] that annotation burden limits the extent to which contracts are used.) The fatigue is, however, of small magnitude if present at all, and may be just a sign of reached maturity where a solid initial design with plenty of specification elements pays off in the long run to the point that less relative investment is sufficient to maintain a stable level of maintainability and quality.

The remaining projects have significant positive correlations ($\tau > 0.5$) between either percentage of specified routines and number of routines or between percentage of specified classes and number of classes, but not both. In these special cases, it looks as if the fraction of programming effort devoted to writing specification tends to increase with the absolute size of the system: when the system grows, proportionally more routines or classes get a specification. However, visual inspection suggests that, in all cases, the trend is ephemeral or contingent on transient phases where the project size changes significantly in little time. As the projects mature and their sizes stabilize, the other two trends (no correlation or negative correlation) emerge in all cases.

**The fraction of routines and classes with some specification is quite stable over time. Local exceptions are possible when major redesign changes take place.**

### 6.3.3 Kinds of Contract Elements

Do programmers prefer preconditions? Typically, one would expect that preconditions are simpler to write than postconditions (and, for that matter, class invariants): postconditions are predicates that may involve two states (before and after routine execution). Furthermore, programmers have immediate benefits in writing preconditions as opposed to postconditions: a
6.3. HOW CONTRACTS ARE USED

routine’s precondition defines the valid input; hence, the stronger it is, the fewer cases the routine’s body has to deal with.

Contrary to this common assumption, the data in our study (columns % Routines Pre and Post in Table 6.3) is not consistently lopsided towards preconditions. 2 projects show no difference in the median percentages of routines with precondition and with postcondition. 10 projects do have, on average, more routines with precondition than routines with postcondition, but the difference in percentage is less than 10% in 5 of those projects, and as high as 39% only in one project (Dafny). The remaining 9 projects even have more routines with postcondition than routines with precondition, although the difference is small (less than 5%) in 5 projects, and as high as 45% only in RCC.

On the other hand, in 17 projects the percentage of routines with some specification (precondition, postcondition, or both) is higher than both percentages of routines with precondition and of routines with postcondition. Thus, we can partition the routines of most projects in three groups of comparable size: routines with only precondition, routines with only postcondition, and routines with both. The 4 exceptions are CCI, Shweet, DirectVCGen, and Umbra where, however, most elements have little specification. In summary, many exogenous causes may concur to determine the ultimate reasons behind picking one kind of contract element over another, such as the project domain and the different usage of different specification elements. Our data is, however, consistent with the notion that programmers choose which specification to write according to context and requirements, not based on a priori preferences. It is also consistent with Schiller et al.’s observations that contract usage follows different patterns in different projects, and that programmers are reluctant to change their preferred usage patterns—and hence patterns tend to remain consistent within the same project.

A closer look at the projects where the difference between percentages of routines with precondition and with postcondition is significant (9% or higher) reveals another interesting pattern. All 6 projects that favor preconditions are written in C# or Java: Dafny, Labs, Quickgraph, Shweet, ESCJava (third plot in Table 6.2, after rev. 400), and JavaFE; conversely, the 3 of 4 projects that favor postconditions are in Eiffel (AutoTest, GoboKernel, and GoboTime), whereas the fourth is RCC written in Java. A possible explanation for this division involves the longer time that Eiffel has supported contracts and the principal role attributed to Design by Contract within the Eiffel community.

| Preconditions and postconditions are used equally frequently across most projects. |
Class invariants. Class invariants have a somewhat different status than pre- or postconditions. Since class invariants must hold between consecutive routine calls, they define object consistence, and hence they belong to a different category than pre- and postconditions. The percentages of classes with invariant (% classes inv in Table 6.3) follow similar trends as pre- and postconditions in most projects in our study. Only 4 projects stick out because they have 4% or less of classes with invariant, but otherwise make a significant usage of other specification elements: Quickgraph, EiffelProgramAnalysis, Shweet, and DirectVCGen.\(^3\) Compared to the others, Shweet has a short history and EiffelProgramAnalysis involves students as main developers rather than professionals. Given that the semantics of class invariants is less straightforward than that of pre- and postconditions—and can become quite intricate for complex programs\(^4\)—this might be a factor explaining the different status of class invariants in these projects. A specific design style is also likely to influence the usage of class invariants, as we further comment on in Section 6.3.4.

Kinds of constructs. An additional classification of contracts is according to the constructs they use. We gathered data about constructs of three types: expressions involving checks that a reference is `Void` (Eiffel) or `null` (C# and Java); some form of finite quantification (constructs for `∀`/`∃` over containers exist for all three languages); and `old` expressions (used in postconditions to refer to values in the pre-state). `Void/null` checks are by far the most used: in Eiffel, 36%–93% of preconditions, 7%–62% of postconditions, and 14%–86% of class invariants include a `Void` check; in C#, 80%–96% of preconditions contain `null` checks, as do 34%–92% of postconditions (the only exception is CCI which does not use postconditions) and 97%–100% of invariants (exceptions are Quickgraph at 20% and Shweet which does not use invariants); in Java, 88%–100% of preconditions, 28%–100% of postconditions, and 50%–77% of class invariants contain `null` (with the exception of Umbra which has few contracts in general). `Void/null` checks are simple to write, and hence cost-effective, which explains their wide usage; this may change in the future, with the increasing adoption of static analyses which supersede such checks\(^70, 24\). The predominance of simple contracts and its justification have been confirmed by others\(^96\).

At the other extreme, quantifications are very rarely used: practically never in pre- or postconditions; and very sparsely (1%–10% of invariants) only in AutoTest, Boogie, Quickgraph, ESCJava, and JavaFE’s class invariants.\(^3\) While the projects CCI and Umbra have few classes with invariants (4%–6%), we don’t discuss them here because they also only have few routines with preconditions or postconditions.
This may also change in the future, thanks to the progresses in inferring complex contracts \cite{51,111,110}, and in methodological support \cite{SS}.

The usage of \texttt{old} is more varied: C\# postconditions practically don’t use it, Java projects rarely use it (2\%–3\% of postconditions at most), whereas it features in as many as 39\% of postconditions for some Eiffel projects. Using \texttt{old} may depend on the design style; for example, if most routines are side-effect free and return a value function solely of the input arguments there is no need to use \texttt{old}.

\begin{quote}
\textbf{The overwhelming majority of contracts involves \texttt{void/}null checks.} \\
\textit{In contrast, quantifiers appear very rarely in contracts.}
\end{quote}

\subsection{Contract Size and Strength}

The data about specification size (and strength) partly vindicates the intuition that preconditions are more used. While Section \ref{sec:prepost} showed that routines are not more likely to have preconditions than postconditions, preconditions have more clauses on average than postconditions in all but the 3 projects GoboTime, ESCJava, and Logging. As shown in columns \textsc{avg} \textsc{routines pre} and \textsc{post} of Table \ref{tab:contracts}, the difference in favor of preconditions is larger than 0.5 clauses in 9 projects, and larger than 1 clause in 3 projects. CCI never deploys postconditions, and hence its difference between pre- and postcondition clauses is immaterial. GoboTime is a remarkable outlier: not only do twice as many of its routines have a postcondition than have precondition, but its average postcondition has 0.66 more clauses than its average precondition. ESCJava and Logging also have larger postconditions on average but the size difference is less conspicuous (0.25 and 0.32 clauses). We found no simple explanation for these exceptions, but they certainly are the result of deliberate design choices.

The following two facts corroborate the idea that programmers tend to do a better job with preconditions than with postconditions—even if they have no general preference for one or another. First, the default “trivial” precondition \texttt{true} is a perfectly reasonable precondition for routines that compute total functions—defined for every value of the input; a trivial postcondition is, in contrast, never satisfactory. Second, in general, “strong” postconditions are more complex than “strong” preconditions \cite{SS} since they have to describe more complex relations.

Class invariants are not directly comparable to pre- and postconditions, and their usage largely depends on the design style. Class invariants apply to all routines and attributes of a class, and hence they may be used extensively and involve many clauses; conversely, they can also be replaced by pre- and
In over eighty percent of the projects, the average **preconditions** contain more clauses than the average **postconditions**.

Section 6.3.1 observed the prevailing stability over time of routines with specification. Visual inspection and the values of standard deviation point to a qualitatively similar trend for specification size, measured in number of clauses. In the first revisions of a project, it is common to have more varied behavior, corresponding to the system design being defined; but the average strength of specifications typically reaches a plateau, or varies quite slowly, in mature phases.

Project Labs is somewhat of an outlier, where the evolution of specification strength over time has a rugged behavior (see [31] for details and plots). Its average number of class invariant clauses has a step at about revision 29, which corresponds to a merge, when it suddenly grows from 1.8 to 2.4 clauses per class. During the few following revisions, however, this figure drops quickly until it reaches a value only slightly higher than what it was before revision 29. What probably happened is that the merge mixed classes developed independently with different programming styles (and, in particular, different attitudes towards the usage of class invariants). Shortly after the merge, the developers refactored the new components to make them comply with the overall style, which is characterized by a certain average invariant strength.

One final, qualitative, piece of data about specification strength is that in a few projects there seems to be a moderate increase in the strength of postconditions towards the latest revisions of the project. This observation is however not applicable to any of the largest and most mature projects we analyzed (e.g., EiffelBase, Boogie, Dafny).

**The average size (in number of clauses) of specification elements is stable over time.**

### 6.3.5 Implementation vs. Specification Changes

Contracts are **executable** specifications; normally, they are checked at runtime during debugging and regression testing sessions (and possibly also in
production releases, if the overhead is acceptable, to allow for better error reporting from final users). Specifically, most applications and libraries of our study are actively used and maintained. Therefore, their contracts cannot become grossly misaligned with the implementation.

A natural follow-up question is then whether contracts change more often or less often than the implementations they specify. To answer, we compare two measures in the projects: for each revision, we count the number of routines with changed body and changed specification (pre- or postcondition) and compare it to the number of routines with changed body and unchanged specification. These measures aggregated over all revisions determine a pair of values \((c_P, u_P)\) for each project \(P\): \(c_P\) characterizes the frequency of changes to implementations that also caused a change in the contracts, whereas \(u_P\) characterizes the frequencies of changes to implementations only. To avoid that few revisions with very many changes dominate the aggregate values for a project, each revision contributes with a binary value to the aggregate value of a project: 0 if no routine has undergone a change of that type in that revision, and 1 otherwise.\(^4\) We performed a Wilcoxon signed-rank test comparing the \(c_P\)'s to the \(u_P\)'s across all projects to determine if the median difference between the two types of events (changed body with and without changed specification) is statistically significant. The results confirm with high statistical significance \((V = 0, p = 9.54 \cdot 10^{-7}, \text{and large effect size—Cohen's } d > 0.99)\) that specification changes are quite infrequent compared to implementation changes for the same routine. Visual inspection also confirms the same trend: see the last plot in Table 6.2 about Boogie. A similar analysis ignoring routines with trivial (empty) specification leads to the same conclusion also with statistical significance \((V = 29, p = 4.78 \cdot 10^{-3}, \text{and medium effect size } d > 0.5)\).

When specifications do change, what happens to their strength measured in number of clauses? Another Wilcoxon signed-rank test compares the changes to pre- and postconditions and class invariants that added clauses (suggesting strengthening) against those that removed clauses (suggesting weakening). Since changes to specifications are in general infrequent, the results were not as conclusive as those comparing specification and implementation changes. The data consistently points towards strengthening being more frequent than weakening: \(V = 31.5\) and \(p < 0.02\) for precondition changes; \(V = 29\) and \(p < 0.015\) for postcondition changes; \(V = 58.5\) and \(p = 0.18\) for invariant changes. The effect sizes are, however, smallish: Cohen’s \(d\) is about 0.4, 0.42, and 0.18 for preconditions, postconditions, and invariants. In all,

---

\(^4\)Using other “reasonable” aggregation functions (including exact counting) leads to qualitatively similar results.
the effect of strengthening being more frequent than weakening seems to be real but more data is needed to obtain conclusive evidence.

\[\text{The implementation of an average routine changes much more frequently than its specification.}\]

6.3.6 Inheritance and Contracts

Inheritance is a principal feature of object-oriented programming, and involves contracts as well as implementations; we now evaluate its effects on the findings previously discussed.

We visually inspected the plots and computed correlation coefficients for the percentages and average strength of specified elements in the flat (explicitly including all routines and specification of the ancestor classes) and non-flat (limited to what appears in the class text) versions of the classes. In the overwhelming majority of cases, the correlations are high and statistically significant: 16 projects have $\tau \geq 0.54$ and $p < 10^{-9}$ for the percentage of routines with specification; 17 projects have $\tau \geq 0.66$ and $p \sim 0$ for the percentage of classes with invariant; 12 projects have $\tau \geq 0.58$ and $p < 10^{-7}$ for the average precondition and postcondition strength (and 7 more projects still have $\tau \geq 0.33$ and visually evident correlations); and 15 projects have $\tau \geq 0.45$ and $p \sim 0$ for the average invariant strength. The first-order conclusion is that, in most cases, ignoring the inherited specification does not preclude understanding qualitative trends.

What about the remaining projects, which have small or insignificant correlations for some of the measures in the flat and non-flat versions? Visual inspection often confirms the absence of significant correlations, in that the measures evolve along manifestly different shapes in the flat or non-flat versions; the divergence in trends is typically apparent in the revisions where the system size changes significantly, where the overall design—and the inheritance hierarchy—is most likely to change. To see if these visible differences invalidate some of the findings discussed so far, we reviewed the findings against the data for flat classes. The big picture was not affected: considering inheritance may affect the measures and offset or bias some trends, but the new measures are still consistent with the same conclusions drawn from the data for non-flat classes. Future work will investigate whether this result is indicative of a mismatch between the semantics of inheritance and how it is used in practice [100, 90]. (See the extended version [31] for details.)
Qualitative trends of measures involving contracts do not change significantly whether we consider or ignore inherited contracts.

6.4 Threats to Validity

**Construct validity.** Using the number of clauses as a proxy for the strength of a specification may produce imprecise measures; Section 6.2, however, estimated the imprecision and showed it is limited, and hence an acceptable trade-off in most cases (also given that computing strength exactly is infeasible). Besides, the number of clauses is still a valuable size/complexity measure in its own right (Section 6.3.4).

**Internal validity.** Since we targeted object-oriented languages where inheritance is used pervasively, it is essential that the inheritance structure be taken into account in the measures. We fully addressed this major threat to internal validity by analyzing all projects twice: in non-flat and flat version (Section 6.3.6).

**External validity.** Our study is restricted to three formalisms for writing contract specifications. While other notations for contracts are similar, we did not analyze other types of formal specification, which might limit the generalizability of our findings. In contrast, the restriction to open-source projects does not pose a serious threat to external validity in our study, because several of our projects are mainly maintained by professional programmers (EiffelBase and Gobo projects) or by professional researchers in industry (Boogie, CCI, Dafny, and Quickgraph).

An important issue to warrant external validity involves the selection of projects. We explicitly targeted projects that make a non-negligible usage of contracts (Section 6.2), as opposed to the overwhelming majority that only include informal documentation or no documentation at all. This deliberate choice limits the generalizability of our findings, but also focuses the study on understanding how contracts can be seriously used in practice. A related observation is that the developers of several of the study’s projects are supporters of using formal specifications. While this is a possible source of bias it also contributes to reliability of the results: since we are analyzing good practices and success stories of writing contracts, we should target competent programmers with sufficient experience, rather than inexpert novices. Besides, Schiller et al.’s independent analysis 96 of some C# projects using CodeContracts also included in our study suggests that their developers are hardly fanatic about formal methods, as they use contracts only to the extent
that it remains inexpensive and cost-effective, and does not require them to change their programming practices.

Nevertheless, to get an idea of whether the programmers we studied really have incomparable skills, we also set up a small control group, consisting of 10 projects developed by students of a software engineering course involving students from universities all around the world. In summary (see \cite{31} for details), we found that several of the trends measured with the professional programmers were also present in the student projects—albeit on the smaller scale of a course project. This gives some confidence that the big picture outlined by this chapter’s results somewhat generalizes to developers willing to spend some programming effort to write contracts.

### 6.5 Related Work

To our knowledge, this study is the first quantitative empirical study of specifications in the form of contracts and their evolution together with code. Schiller et al. \cite{96} study C# projects using CodeContracts (some also part of our study); while our and their results are not directly comparable because we take different measures and classify contract usage differently, the overall qualitative pictures are consistent and nicely complementary. In this chapter we also highlighted a few points where their results confirm or justify ours. Schiller et al. do not study contract evolution; there is evidence, however, that other forms of documentation—e.g., comments \cite{40}, APIs \cite{59}, or tests \cite{114}—evolve with code.

A well-known problem is that specification and implementation tend to diverge over time; this is more likely for documents such as requirements and architectural designs that are typically developed and stored separately from the source code. Much research has targeted this problem; specification refinement, for instance, can be applied to software revisions \cite{43}. Along the same lines, some empirical studies analyzed how requirements relate to the corresponding implementations; \cite{54}, for example, examines the co-evolution of certain aspects of requirements documents with change logs and shows that topic-based requirements traceability can be automatically implemented from the information stored in version control systems.

The information about the usage of formal specification by programmers is largely anecdotal, with the exceptions of a few surveys on industrial practices \cite{17,113}. There is, however, some evidence of the usefulness of contracts and assertions. \cite{62}, for example, suggests that increases of assertions density and decreases of fault density correlate. \cite{72} reports that using assertions may decrease the effort necessary for extending existing programs and increase
their reliability. In addition, there is evidence that developers are more likely to use contracts in languages that support them natively [17]. As the technology to infer contracts from code reaches high precision levels [29, 111], it is natural to compare automatically inferred and programmer-written contracts; they turn out to be, in general, different but with significant overlapping [87].

6.6 Concluding Discussion & Implications of the Results

Looking at the big picture, our empirical study suggests a few actionable remarks. (i) The effort required to make a quantitatively significant usage of lightweight specifications is sustainable consistently over the lifetime of software projects. This supports the practical applicability of methods and processes that rely on some form of rigorous specification. (ii) The overwhelming majority of contracts that programmers write in practice are short and simple. This means that, to be practical, methods and tools should make the best usage of such simple contracts or acquire more complex and complete specifications by other means (e.g., inference). It also encourages the usage of simple specifications early on in the curriculum and in the training of programmers [60]. (iii) In spite of the simplicity of the contracts that are used in practice, developers who commit to using contracts seem to stick to them over an entire project lifetime. This reveals that even simple specifications bring a value that is worth the effort: a little specification can go a long way. (iv) Developers often seem to adapt their contracts in response to changes in the design; future work in the direction of facilitating these adaptations and making them seamless has a potential for a high impact. (v) A cornerstone software engineering principle—stable interfaces over changing implementations—seems to have been incorporated by programmers. This also positively answers the question is then whether awareness tools should leveraged contracts to monitor and report changes, therewith improving collaboration between programmers in a development team. (vi) Somewhat surprisingly, inheritance does not seem to affect most qualitative findings of our study. The related important issue of how behavioral subtyping is achieved in practice [90] belongs to future work, together with several other follow-up questions whose answers can build upon the foundations laid by this chapter’s results.
average (μ) and standard deviation (σ) across all revisions.}

<table>
<thead>
<tr>
<th>Project</th>
<th>µ pre</th>
<th>σ pre</th>
<th>µ post</th>
<th>σ post</th>
<th>µ classes inv</th>
<th>σ classes inv</th>
</tr>
</thead>
<tbody>
<tr>
<td>Umbra</td>
<td>1</td>
<td>1.04</td>
<td>1.05</td>
<td>0.02</td>
<td>0.04</td>
<td>0.01</td>
</tr>
<tr>
<td>Logging</td>
<td>1.21</td>
<td>1.28</td>
<td>1.36</td>
<td>0.04</td>
<td>0.03</td>
<td>0.07</td>
</tr>
<tr>
<td>JavaFE</td>
<td>0.04</td>
<td>0.52</td>
<td>0.54</td>
<td>0.23</td>
<td>0.02</td>
<td>0.56</td>
</tr>
<tr>
<td>Dafny</td>
<td>0</td>
<td>0.01</td>
<td>0.02</td>
<td>0.01</td>
<td>0.02</td>
<td>0.04</td>
</tr>
<tr>
<td>CCI</td>
<td>1</td>
<td>1.02</td>
<td>1.02</td>
<td>0.01</td>
<td>0.02</td>
<td>0.03</td>
</tr>
<tr>
<td>Rxx</td>
<td>1</td>
<td>1.18</td>
<td>1.36</td>
<td>0.46</td>
<td>0.01</td>
<td>0.03</td>
</tr>
<tr>
<td>Shweet</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.01</td>
<td>0.02</td>
</tr>
<tr>
<td>DirectVCG</td>
<td>0</td>
<td>1.18</td>
<td>1.36</td>
<td>0.46</td>
<td>0.01</td>
<td>0.03</td>
</tr>
<tr>
<td>GoboXML</td>
<td>0</td>
<td>1.8</td>
<td>2.07</td>
<td>0.24</td>
<td>0.12</td>
<td>0.16</td>
</tr>
<tr>
<td>GoboUtility</td>
<td>0</td>
<td>1.62</td>
<td>1.7</td>
<td>0.15</td>
<td>0.12</td>
<td>0.15</td>
</tr>
<tr>
<td>GoboStruct</td>
<td>0</td>
<td>1.17</td>
<td>1.44</td>
<td>1.49</td>
<td>0.10</td>
<td>0.10</td>
</tr>
<tr>
<td>GoboKernel</td>
<td>0</td>
<td>1.23</td>
<td>1.25</td>
<td>0.09</td>
<td>0.25</td>
<td>0.27</td>
</tr>
</tbody>
</table>
Chapter 7

Collaborative Debugging

7.1 Introduction

Debugging is the process of dealing with one of life’s inevitables—programming mistakes. Always a cardinal part of the development process, its importance has prompted the construction of techniques and tools that can assist programmers to make them more productive at finding and fixing errors. Debugging techniques have typically been developed around traditional development practices, and debugging tools have become part of every integrated development environment (IDE) such as Eclipse or VisualStudio. This entails that they are essentially conceived as tools for individual usage, and hence may be a poor match for today’s increasingly collaborative and distributed development processes.

This chapter investigates the problem of deploying debugging techniques and tools in the context of collaborative development, where developers working on the same project cooperate with the goal of finding and correcting errors in their shared codebase. Debugging in collaborative environments is likely to feature as a useful activity in every team development practice, but is especially crucial—if not outright necessary—in the increasingly common distributed development settings: when developers of the same team may be located in physically distinct locations, possibly even in widely different time zones, the fine-grained coordination, required by collaboration on a highly interactive process such as debugging, becomes a real challenge. We offer three main contributions addressing the general problem of collaborative debugging, with an eye towards distributed settings.

Debuggers—such as the GNU debugger or the Microsoft VisualStudio debugger—are normally integrated in IDEs without specific support for shared usage. Programmers working on the same project synchronize indirectly
through shared repositories managed using tools such as Subversion or Git, but their debugging sessions are individual and cannot benefit from collaboration unless they are sitting at the same desk—something hardly possible in a distributed setting. The first contribution of this chapter is a debugging technique, called CDB, designed for remote collaboration. CDB supports multiple programmers, each one sitting at her desk, sharing a common execution of the program under debugging. Every programmer can add or remove breakpoints, inspect variables, and navigate the code, with the others aware of each other’s actions in real-time. We implemented this debugging technique leveraging the features of CloudStudio, the web-based IDE for collaborative development we introduced in previous work [34]. Section 7.3 discusses our collaborative debugging technique and how it can be used in practice. CloudStudio and CDB are available online.

Evaluating the effectiveness of a complex and ultimately human-driven process such as debugging is a challenging problem even in traditional single-user sessions. Parnin and Orso [84], for example, provided evidence that the traditional protocols used to evaluate fault localization techniques often misrepresent or overstate their real effectiveness when used within actual debugging sessions. In collaborative settings, designing sound empirical evaluations becomes even more challenging, as more variables (such as the number of people collaborating) must be controlled and accounted for. The second contribution of this chapter is the design of an empirical study of collaborative debugging in possibly distributed settings. Our experimental protocol accommodates analysis focused on the interaction model—for example, as in CDB vs. using traditional processes. We describe the protocol in Section 7.4.

Our third contribution is an actual empirical study of collaborative debugging, following the experimental design just outlined. The study involved 38 participants, performing 10 debugging tasks of various difficulty using either our collaborative debugging technique CDB or a standard remote-desktop application to follow and interact with their debugging partners. We split the students in different groups to achieve a good trade-off between statistical significance and generalizability of the results; in particular, we experimented with different interaction models and included both pairs and triples of programmers sharing the same debugging session. Section 7.5 describes the details of the study, whereas Section 7.6 analyzes potential threats to validity. The study’s most significant findings are: the two features that are generally perceived the most important for collaborative debugging are:

- being able to independently browse the code under debug;
and add variables to watch.

There is no evidently preferred mode of control, but debugging with CDB provides for collaborative debugging with a more uniform level of involvement, as well as for debugging processes perceived as more efficient and generally preferred for collaborative tasks.

Before presenting our contributions, Section 7.2 gives an overview of the challenges of using debugging tools collaboratively, in a setting where programmers may be displaced at different locations and interact only remotely.

### 7.2 Collaborative Debugging in DSD Teams

Let us introduce our two fine programmers Pippo and Binha. Pippo lives in Italy and has recently joined our team of developers working on a large Java application. His current assignment involves using a binary tree implementation written a few weeks ago by Binha—who belongs to the Brazilian development unit.

To get started and understand the binary tree's API, Pippo writes the client code in class `Client` shown at the bottom of Figure 7.2. His code creates an instance \( t \) of class `BinaryTree` and populates it with a few nodes of class `Node` storing integer values. The rest of Figure 7.2 outlines the essential parts of the `BinaryTree` and `Node` classes written by Binha.

It does not take long before Pippo realizes that something's wrong with the binary tree implementation. Since he has access to the whole codebase, Pippo may simply debug the implementation on his own. This approach has, however, some evident drawbacks. Since Pippo is not familiar with the details of the binary tree implementation, he is likely to be slow at debugging it. Worse, he may not be aware of the other clients' usage requirements, and his fixes may negatively affect them without him realizing it. Finally, touching base with Binha is probably advisable in any case, just to avoid that the two of them introduce conflicting changes which will require a later painful merge process.

So Pippo asks Binha to help him debug the problems he is facing. Since they have no specific tool support for collaborative debugging, the best they can do is using a remote-desktop application: Pippo's computer is running the debugger within Eclipse; and Binha is connected remotely, sees whatever Pippo sees in his computer screen, and can text him with comments or requests for actions.

To demonstrate the problem, Pippo adds breakpoints at lines 36, 37, and 39 and starts the debugger. When execution stops at the first breakpoints (line 36), the debugger displays a pane with the variables in scope as in
at line after fixing the first bug

Figure 7.1: Debugging the binary tree example. (a) Before fixing the first bug. (b) After fixing the first bug. I State

(a) Before fixing the first bug.

(b) After fixing the first bug.
7.2. COLLABORATIVE DEBUGGING IN DSD TEAMS

```java
class BinaryTree<G extends Comparable<G>> {
    Node<G> data;
    BinaryTree<G> left, right;

    public BinaryTree (Node<G> root) {
        data = root; left = null; right = null;
    }

    public void insert (Node<G> d) {
        if (d.lessThan(data)) {
            if (left == null) {
                left = new BinaryTree<G>(d);
            } else { left.insert(d); }
        }
        if (d.greaterThan(data)) {
            if (right == null) {
                right = new BinaryTree<G>(d);
            } else { right.insert(d); }
        }
    }

    public void traverse(int i) { ... }
}
```

```java
class Node<T extends Comparable<T>> {
    T data;
    public Node(T d) { data = d; }
    public boolean lessThan(Node<T> n) { ... }
    public boolean greaterThan(Node<T> n) { ... }
}
```

```java
class Client {
    public void main() {
        BinaryTree<Integer> t =
            new BinaryTree<>(new Node<Integer>(17));
        t.insert(new Node<Integer>(10));
        t.insert(new Node<Integer>(30));
        t.insert(new Node<Integer>(35));
        t.insert(new Node<Integer>(12));
        t.insert(new Node<Integer>(21));
        t.traverse(0);
    }
}
```

Figure 7.2: A Java binary tree implementation.

Figure 7.1a This shows that the node with value 10 has been incorrectly inserted twice in the tree. Binha realizes that the problem is the conditional
at line 15 which should be exercised only if the previous condition on line 10 fails. Binha suggests to change the if on line 15 into an else if. She cannot do the change directly and her view is limited to the code currently displayed in Pippo’s IDE window; instead, she explains the problem to Pippo via voice chat and asks him to deploy the change.

After fixing as suggested by Binha, Pippo re-starts the debugger. Upon reaching the breakpoint at line 36, it now shows the state in Figure 7.1b which looks fine. Pippo issues a step-over command, which continues execution until the next breakpoint at line 37, that is it inserts a node with value 30. It is clear, however, that there is still a problem, as the insertion does not actually change the state of the tree.

After coordinating with Binha again over voice chat, Pippo issues a step-into command, which causes the debugger—now at the second breakpoint at line 37—to show the code executed by the call \( t.\text{insert}(35) \) which inserts a node with value 35. The debugger shows that the condition \( d.\text{lessThan}(\text{data}) \) on line 10 evaluates to false; the next condition \( d.\text{greaterThan}(\text{data}) \) on line 15 is also false given the state of Figure 7.1c. Thus, the problem is with the implementation of lessThan or greaterThan:

```java
43 public boolean lessThan(Node<T> n) {
44     int last = this.data.compareTo(n.data);
45     return last < 0 ? true : false;
46 }
47 public boolean greaterThan(Node<T> n) {
48     int last = this.data.compareTo(n.data);
49     return last > 0 ? false : true;
50 }
```

Evaluating the expressions \( \text{this.data.value} \) and \( \text{d.data.value} \) in the current debugging context gives the values 17 and 35, and variable \( \text{last} \) in library method greaterThan correspondingly evaluates to 1 because \( 35 = \text{d.data.value} > \text{this.data.value} = 17 \). The conditional expression on line 49 however, incorrectly returns false. Pippo and Binha agree that a suitable change is switching the returned values on line 49 which becomes \( \text{return last > 0 ? true : false.} \)

### 7.3 Collaborative Debugging with CDB

Even if the debugging session described in Section 7.2 eventually succeeded, it showed that using a remote-desktop application to collaborate has several shortcomings. In fact, achieving effective collaboration during shared debugging sessions seems to have conflicting requirements. On the one hand, the programmers involved should be able to share a common debugging session; ideally, each should be able to modify the code or interact with the debugger directly, without need to describe actions in speech or via chat and request
someone else to carry them out. At the same time, each programmer should also be able to perform independent activities in parallel, such as browsing parts of the project or testing the effects of small changes to the code, without interfering with or depending on what his colleagues are doing in their IDEs. Remote-desktop applications provide sharing, but subject to a strict discipline where one or more clients have access to a master machine. The clients’ view on the master machine are limited to what is displayed on its screen at any time, and there is no simple way to coordinate or even to switch the role of master with one of the clients.

Based on these preliminary observations, which the empirical study of Section 7.5 will corroborate, we designed CDB, a collaborative debugger that...
facilitates remote coordination. CDB is integrated within the CloudStudio web-based IDE, which we developed in related work [34]. Users of CDB log-in to the CloudStudio server using any web browser. Whenever they select a common project to work on, they can start shared debugging sessions using CDB. A shared session has a unique thread of execution on the project under debugging; all participating users observe the program state throughout the shared session, for example by displaying the values of certain variables. Figure 7.3 shows a screenshot of CDB running on the example of Section 7.2.

CDB offers the standard commands to control debugging sessions: add or remove breakpoints (where execution pauses); step-over a call (execute it as an atomic statement); step-into a call (execute the next of its constituent steps); step-out of a call (switch back to the higher level after a step-into); and resume or terminate the whole session. All collaborating users see the effect of any issued commands on their shared session.

A crucial issue for usability is achieving a suitable balance of flexibility and discipline in how to control debugging sessions: flexibility for users to issue commands without requiring explicit coordination—thus reducing the communication overhead—and discipline to avoid haphazard debugging sessions—thus reducing the impact of conflicting strategies. To this end, CDB offers two control modes, which achieve different trade-offs between flexibility and discipline. While all users are allowed to insert or remove breakpoints in either mode, the other commands are managed differently:

**Master mode:** a single user is allowed to issue the commands step-over, step-into, step-out, resume, and terminate; the other users observe the execution controlled by the master. At any time during the session, the master can switch roles with one of other users. This is often useful when the execution reaches parts of the code the master is not familiar with; he may then decide to become an observer and tap in a more knowledgeable collaborator, who takes the lead without having to start over with a new debugging session.

**Peer mode:** all connected users are allowed to issue any of the commands. To avoid some extreme situations where conflicting commands are issued whose net effect is void (for example, a step-into followed by a step-out shortly afterward), whenever a user X issues a command, CDB rejects any new command issued by any other user Y within a couple of seconds. This retains some coordination discipline by implicitly requiring that Y waits at least until X concludes a sequence of arguably closely related commands.

Debugging with CDB offers additional perks that derive from being in-
7.4 Empirical Study: Experimental Design

The overview of Section 7.2 highlighted some issues that are likely to surface when performing debugging in collaborative distributed settings; and the presentation of our collaborative debugger CDB in Section 7.3 suggested an approach that may improve the effectiveness of the debugging experience in such settings. The rest of the chapter describes an empirical study aimed at evaluating issues of and approaches to collaborative debugging.

The study’s main goal is testing whether the CDB approach addresses the relevant issues, and whether it brings tangible benefits. The main research questions are correspondingly as follows (the rest of the section gives a characterization of “effective collaboration”):

**RQ1:** Which debugger features (e.g., adding breakpoints) are critical for an effective collaboration in debugging sessions?
CHAPTER 7. COLLABORATIVE DEBUGGING

RQ2: What is the relevance of different control policies (e.g., a single programmer always in control) for an effective collaboration in debugging sessions?

RQ3: How do the two debugging experiences, using remote-desktop vs. using CDB, compare?

Let us now discuss how we collected evidence to answer these general questions.

Assignments and tasks. We evaluate programmers working on two assignments: the List assignment and the Tree assignment. The two assignments have comparable size and complexity, and differ only in the kind of data structure they target—respectively linked lists and binary search trees. Each assignment comes with 100–200 lines of source code written in JavaScript (the example of Section 7.2 is a simplified Java variant of the Tree assignment). We concocted the assignments with the goal of having codebases sufficiently simple, so that programmers can find their ways through them in the limited time allotted by the experiments, but also not entirely trivial.

An assignment consists of five tasks; each task comes with a test case that reveals an error of the assignment’s data structure; the task’s goal is fixing the error through debugging. The participants are also given a written description of the tasks, including detailed instructions and a short tutorial describing the tools and how to use them. They are also allowed to ask for clarification to the supervisor.

The first task in each batch is simpler than the other four, and the programmer performance on it is not evaluated. Since tasks are executed in order, the first task serves as a warm-up: programmers get a chance to acquire some familiarity with the codebase and with the debugging tools at their disposal. This increases the similarity of the experimental setup with real debugging scenarios, where it is likely that programmers already have some knowledge of the codebase and of the tools. It may also reduce the impact of different participants to the study having different previous experience with the tools.

Debugging tools and scenarios. To make the debugging experiences using remote-desktop vs. using our CDB comparable, we need to set up debugging environments that are as similar as possible except for the debugging tools. For example, comparing remote-desktop using the Eclipse debugger (as in Section 7.2) to CDB integrated in CloudStudio would make little sense, given that a widely-used and mature IDE such as Eclipse makes for an overall quite different user experience than the research prototype CloudStudio.

Instead, we set up two usage scenarios for CloudStudio: CD and SS. Under scenario CD (for “CDB” debugging), developers use CDB exactly as described in Section 7.3 with the shared debugging session, the various control modes, as well as all other features of CloudStudio. Under scenario SS (for “shared-screen” debugging), one master developer uses CloudStudio on her machine as if it were a single-user browser; the other developers participating to the debugging session connect to the master machine using a remote-desktop application, can only watch what is displayed on the master machine and interact with the master via text chat, Skype, or voice (the last option is obviously possible when they are in the same room) as described in Section 7.2. In this way, the features offered by the bare IDE are the same, whereas the collaboration means are quite different in the two scenarios.

We randomly split the participants in three groups, using different combinations of CD and SS:

**G1**: programmers in this group debug following scenario SS. Half of the group works on *List* and half works on *Tree*.

**G2**: programmers in this group debug following scenario CD. Half of the group works on *List* and half works on *Tree*.

**G3**: programmers in this group debug following both scenarios CD and SS. This is organized in two subgroups, according to which scenario they follow first:

**G3.A**: programmers first work on an assignment under SS and then work on the other assignment under CD. Half of the group works first on *List* and then on *Tree*; the other half works first on *Tree* and then on *List*.

**G3.B**: programmers first work on an assignment under CD and then work on the other assignment under SS. Half of the group works first on *List* and then on *Tree*; the other half works first on *Tree* and then on *List*.

Group G3 makes it possible to evaluate if the user experience changes when programmers have a chance to try both scenarios CD and SS and to compare them. The split of G3 into G3.A and G3.B helps control for the influence of getting experience with debugging under one scenario on debugging under the other scenario: since both CD and SS use the same basic CloudStudio IDE, the performance on the second assignment may improve just as a result of becoming familiar with the IDE.
Team allocation. We split programmers in each group in debugging teams. Members of the same team interact following scenario SS or CD. Our study does not measure the effect of distribution on debugging performance, even though we had a mixture of teams whose members were in the same room, in the same building, in different locations in the same city, and even in different countries.

To evaluate the effect of increasing levels of collaboration, each group includes debugging teams of different size. In our experiments, we focused on 2-programmer and 3-programmer teams: 2-programmer teams are the baseline for collaborative debugging, whereas 3-programmer teams demonstrate whether more collaboration is achievable with sustainable overhead. Future studies focusing on collaboration may experiment with teams of even larger size, after addressing the criticalities shown by our study.

Previous experience. To control for previous experience of the participants, we ask them to rank on a 1–5 scale (no experience–lots of experience) their experience with programming in general, with JavaScript programming, and with interactive debugging. We also report whether they had already experienced collaborative debugging of any kind before the study.

7.4.1 Critical Debugger Features: RQ1

Research question RQ1 looks for critical features of debuggers used collaboratively. After a team has completed its assignments, we ask each of its members to rank on a 1–5 scale the importance of the following features: adding and removing breakpoints, adding and removing monitored variables and expressions, and browsing the codebase independent of others. An additional free-text question asks to mention any other feature that they consider important.

Answers to questions about the same feature are not directly comparable between teams working under different scenarios, who have or don’t have experienced that feature. In fact, questions for teams following SS are phrased as “How much did you miss feature X?” whereas the corresponding questions for teams following CD are phrased as “How useful was feature X?”. Instead, we can directly compare the same questions about SS between teams in group G1 and in group G3, as well as about CD between teams in group G2 and in group G3. We also analyze correlations between questions about different features for the same group.
7.4.2 Control Policies: RQ2

Research question RQ2 studies the impact of the control policies allowed or enforced in the different sessions. After a team has completed its assignments, we ask each of its members to rank on a 1–5 scale: the importance that everyone on the team can issue commands (e.g., step-into) to the debugger; the level of active involvement in the debugging exercise; the degree of control achieved on the debugger. Again, questions were phrased differently for sessions following SS and sessions following CD: questions for teams following SS are phrased as “How much did you miss that everyone can issue commands?”, whereas the corresponding questions for teams following CD are phrased as “How useful was it that everyone can issue commands?”. We also ask to rank on a 1–5 scale: how often each member issued commands (CD scenarios) or asked the person in charge to issue commands (SS scenarios), where 1 denotes “Never” and 5 denotes “More than 12 times”.

When collecting and comparing data about control policies, it is especially important to account for the role each programmer had in the debugging session and for the number of people involved. To this end, we partition the answers to the questions according to whether the respondent was the master: in SS scenarios, the master is the person who is in control of the IDE; in CD scenarios using master mode, the master is the person who is allowed to issue commands.

For the CD scenarios, we also run sessions where CDB operates in peer mode (see Section 7.3), where there is no single master but all team members can issue commands at any time. Correspondingly, the questionnaires for sessions using CD also included questions about the usefulness of the master or peer modes as control policies.

7.4.3 Debugging Experience: RQ3

Research question RQ3 draws a comparison between the debugging experiences using remote-desktop (as in scenario SS) and using CDB (as in scenario CD). After a team has completed its assignments, we ask each of its members to rank on a 1–5 scale the perceived efficiency of debugging under the assigned scenario. For teams in group G3, who experienced both debugging scenarios, we also ask which one they preferred. Finally, we measure the overall time to complete the tasks (not including the first warm-up task); and the number of tasks successfully completed within a limit of 60 minutes.
Figure 7.4: Previous experience with programming in general (top left), JavaScript programming (to right), debugging (bottom left), and participation in collaborative debugging sessions (bottom right) amongst the study participants. All measures on a 1–5 (no experience–lots of experience) scale, and collaborative debugging which is on a 0–4 scale (0 times (0), 1–3 times (1), 4–10 times (2), 11–20 times (3), more than 20 times (4)).

7.5 Empirical Study: Execution and Results

We performed the empirical study described in Section 7.4 with 38 participants: 19 bachelor’s students, 10 master’s students, and 9 professional programmers, spread across Italy, Switzerland, and Croatia. We distributed the participants into groups G1, G2, and G3 and into 2-person and 3-person teams as shown in Table 7.1. We decided the number of components in each
group, but the specific assignment of people to groups was random.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SCENARIO:</td>
<td>SS only</td>
<td>CD only</td>
<td>SS, then CD</td>
<td>CD, then SS</td>
</tr>
<tr>
<td>≠ 2-person teams</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>≠ 3-person teams</td>
<td>1</td>
<td>2</td>
<td>–</td>
<td>1</td>
</tr>
<tr>
<td>≠ teams debugging List</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>≠ teams debugging Tree</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 7.1: Setup of groups and assignments. Participants used shared screen (SS), collaborative debugging (CD), or both techniques to debug one or both of the assignments List or Tree. Precisely, teams in G1 and G2 debugged one of either List or Tree; teams in G3.A debugged both assignments using SS for List and then CD for Tree; teams in G3.B debugged both assignments using CD for List and then SS for Tree.

As shown in Figure 7.4, the previous experience of the participants spans multiple levels. In particular, nearly all participants have significant programming experience; most of them have repeated debugging experience; about half of them have a little experience with JavaScript, collaborative debugging, or both. The mosaic plot in Figure 7.4 also shows that the distribution of experience is comparable in the various groups, namely those working only with SS, only with CD, and with both SS and CD (in any order).

7.5.1 Critical Debugger Features: RQ1

Which features are most useful during collaborative debugging? The boxplot of Figure 7.5 and the corresponding Table 7.2 report the results of the questions targeting RQ1 concerning debugging sessions under scenario SS (using no specific support for collaboration other than remote desktop). Answers are ranked on a 1–5 scale, with 1 denoting “feature not missed” and 5 denoting “feature much missed”. The possibility of browsing the code independent of other teammates is the most missed feature, followed by being able to add expressions and variables to be monitored.

The most missed feature during a shared-screen debugging session is the ability to browse the source code independently.

The boxplot of Figure 7.6 and Table 7.3 summarize the answers to the corresponding questions concerning debugging under scenario CD (using our collaborative debugger CDB). Answers on a 1–5 scale now denote features from “considered not useful” to “considered very useful”. Browsing code still
is a very popular feature, and so it the possibility of adding variables. Adding or removing breakpoints, which everyone can do at any time with CDB, is not considered particularly useful, nor is often missed in the SS scenario.

Independent browsing of code and adding variables were found to be the most useful features of CDB.

The participants did not report any other generic feature in the specific open-answer questions. In all, browsing the code and modifying the variables and expressions monitored by the debugger emerge as critical features for effective collaborative debugging.

![Box plot of features missed during SS debugging](image)

Figure 7.5: Features missed during SS debugging, on a 1–5 scale (not missed–much missed).

<table>
<thead>
<tr>
<th>FEATURE</th>
<th>#</th>
<th>min</th>
<th>median</th>
<th>max</th>
<th>mean</th>
<th>σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breakpoints</td>
<td>30</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>2.5</td>
<td>1.17</td>
</tr>
<tr>
<td>Expressions</td>
<td>30</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>3.1</td>
<td>1.18</td>
</tr>
<tr>
<td>Variables</td>
<td>30</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>3.33</td>
<td>1.3</td>
</tr>
<tr>
<td>Browsing code</td>
<td>25</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>3.6</td>
<td>1.35</td>
</tr>
</tbody>
</table>

Table 7.2: Features missed during SS debugging (# is the number of answers, σ is the standard error).

**Impact of being the master.** While in debugging scenarios CD with CDB all programmers in the team may browse the code and take control of the debugger, the master is strictly fixed during the SS debugging sessions,
where only the person sitting in front of the computer running the debugger can browse, add variables, expressions, and breakpoints, as well as start and stop the debugger.

<table>
<thead>
<tr>
<th>FEATURE</th>
<th>#M</th>
<th>#O</th>
<th>U</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breakpoints</td>
<td>11</td>
<td>19</td>
<td>82</td>
<td>0.33</td>
</tr>
<tr>
<td>Expressions</td>
<td>11</td>
<td>19</td>
<td>109</td>
<td>0.86</td>
</tr>
<tr>
<td>Variables</td>
<td>11</td>
<td>19</td>
<td>103</td>
<td>0.96</td>
</tr>
<tr>
<td>Browsing</td>
<td>9</td>
<td>16</td>
<td>74</td>
<td>0.91</td>
</tr>
</tbody>
</table>

Table 7.4: Significance test to determine if the role during SS debugging sessions impacts how much a feature is missed. #M is the number of programmers in the role of master, #O the number of all other programmers.

Figure 7.6: Features considered useful during CD debugging, on a 1–5 scale (not useful–very useful).
To understand whether being the master affects the perception of which features are important, we performed statistical significance tests comparing the answers about missing features given by masters vs. the other programmers in the team. Since the data may not be normally distributed and is scaled ordinal but not continuous, we use an independent two-group Mann-Whitney $U$ test, with null hypothesis $H_0$: $P(M < O) = P(O < M)$, where $M$ measures the answers given by programmers in the role of master, and $O$ the answers given by all the others. Table 7.4 shows the results. Since $p \geq 33\%$ for every feature, we do not reject $H_0$: there is no evidence that being the master affects the perception of which features are critical for collaborative debugging.

**Impact of group allocation.** While the answers to the questions for SS scenarios (“Which features did you miss?”) and the corresponding questions for CD scenarios (“Which features did you find useful?”) are not directly comparable, it is interesting to see whether taking part in both debugging scenarios affects the perception of either set of questions.

<table>
<thead>
<tr>
<th>Feature</th>
<th>G1</th>
<th>G3</th>
<th>$U_{SS}$</th>
<th>$p_{SS}$</th>
<th>G2</th>
<th>G3</th>
<th>$U_{CD}$</th>
<th>$p_{CD}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breakpoints</td>
<td>11</td>
<td>19</td>
<td>127</td>
<td>0.33</td>
<td>8</td>
<td>19</td>
<td>74.5</td>
<td>0.96</td>
</tr>
<tr>
<td>Expressions</td>
<td>11</td>
<td>19</td>
<td>134</td>
<td>0.20</td>
<td>8</td>
<td>19</td>
<td>57.5</td>
<td>0.30</td>
</tr>
<tr>
<td>Variables</td>
<td>11</td>
<td>19</td>
<td>92</td>
<td>0.61</td>
<td>8</td>
<td>19</td>
<td>102.5</td>
<td>0.13</td>
</tr>
<tr>
<td>Browsing</td>
<td>11</td>
<td>14</td>
<td>108</td>
<td>0.08</td>
<td>8</td>
<td>19</td>
<td>88</td>
<td>0.46</td>
</tr>
</tbody>
</table>

Table 7.5: Significance test to determine if having experienced both debugging scenarios impacts the importance attributed to features. Columns G1, G2, and G3 list the number of programmers in the corresponding groups.

To this end, we performed statistical significance tests comparing the answers about the debugging scenario SS (respectively, CD) coming from people in group G1 (respectively, G2) and in group G3. Again, the nature of the data suggests a $U$-test with the obvious null hypothesis. Table 7.5 shows the results, with the left-hand half of the table about scenario SS and the right-hand half about scenario CD. Since $p \geq 13\%$ for every feature, we do not reject the null hypothesis: there is no evidence that having experienced both debugging scenarios affects the perception of which features are critical for collaborative debugging.

**Correlation analysis.** Tables 7.6 and 7.7 show the correlation coefficients between the variables measuring the background of participants and the features they considered useful; as usual, Table 7.6 first shows the data about sessions under scenario SS and then Table 7.7 the data about sessions under scenario CD.
Table 7.6: Spearman correlations between variables for sessions under scenario SS. Numbers in bold are statistically significant at a 5% level or better. The variables measure the experience with Programming (P), with Debugging (D), with JavaScript (Js), with Collaborative debugging (Cd), as well as how much adding/removing breakpoints (K), Expressions (E), Variables (V), and independent Browsing (B) was missed.

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>D</th>
<th>Js</th>
<th>Cd</th>
<th>K</th>
<th>E</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>0.58</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Js</td>
<td>0.03</td>
<td>0.20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cd</td>
<td>0.18</td>
<td>0.36</td>
<td>0.35</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>0.14</td>
<td>-0.14</td>
<td>0.40</td>
<td>0.19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>0.15</td>
<td>0.20</td>
<td>0.18</td>
<td>0.03</td>
<td>0.52</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>0.15</td>
<td>0.23</td>
<td>0.22</td>
<td>0.00</td>
<td>0.36</td>
<td>0.60</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0.67</td>
<td>0.59</td>
<td>0.10</td>
<td>0.12</td>
<td>0.17</td>
<td>0.49</td>
<td>0.62</td>
</tr>
</tbody>
</table>

Table 7.7: Spearman correlations between variables for sessions under scenario CD. Numbers in bold are statistically significant at a 5% level or better. The variable names are as in Table 7.6.

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>D</th>
<th>Js</th>
<th>Cd</th>
<th>K</th>
<th>E</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>0.54</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Js</td>
<td>0.12</td>
<td>0.18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cd</td>
<td>-0.16</td>
<td>0.25</td>
<td>0.09</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>-0.27</td>
<td>-0.25</td>
<td>-0.29</td>
<td>-0.11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>-0.29</td>
<td>0.12</td>
<td>-0.17</td>
<td>-0.04</td>
<td>0.40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>0.30</td>
<td>0.16</td>
<td>0.05</td>
<td>-0.33</td>
<td>0.31</td>
<td>0.38</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0.31</td>
<td>0.07</td>
<td>-0.15</td>
<td>-0.50</td>
<td>0.40</td>
<td>0.40</td>
<td>0.71</td>
</tr>
</tbody>
</table>

Looking only at the statistically significant correlations, we find the obvious one between programming experience and debugging experience: it is hard to progress in programming without plenty of debugging involved. Valuing independent browsing correlates with valuing adding variables; after all, these are both consistently ranked as the two most important features. The correlation between adding variables and adding expressions is also significant, probably since the latter can be seen as a generalization of the former. The other significant correlations occur only in one of the two scenarios (SS or CD) and fail straightforward interpretations. For example, the correlation between JavaScript programming and valuing adding breakpoints is reasonable—to the extent that a better understanding of the program makes for a more effective control of the debugging sessions—but it is not clear why it is only significant for SS debugging scenarios. Such open points are good
material for future studies.

7.5.2 Control Policies: RQ2

The second research question studies the impact of control policies, and in particular who is the master and how this role can change.

**Amount of commands.** Table 7.8 shows the amount of commands the programmers requested to the master (when in SS scenarios) or performed themselves (when in CD scenarios). Answers were given in a range from 1–5 where 1 means 0 times, 2 means 1–3 times, 3 means 4–7 times, 4 means 8–12 times, and 5 means >12 times.

The data is very similar under the two scenarios, and in fact a $U$ test does not provide any evidence otherwise ($U = 248$ and $p = 0.86$).

<table>
<thead>
<tr>
<th>Scenario</th>
<th>#</th>
<th>min</th>
<th>median</th>
<th>max</th>
<th>mean</th>
<th>$\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS</td>
<td>30</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>2.26</td>
<td>1.15</td>
</tr>
<tr>
<td>CD</td>
<td>30</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>2.56</td>
<td>1.19</td>
</tr>
</tbody>
</table>

Table 7.8: Amount of commands requested to the master by others (SS scenarios) or performed by any programmer (CD scenarios), on a 1–5 scale.

A correlation analysis shows only one significant correlation: for sessions under scenario SS, the amount of commands requested to the master significantly correlates ($\rho = 0.53$, $p < 0.05$) with the general programming experience of those issuing the requests. This confirms the intuition that general programming experience tends to be an indicator of the capability of being actively involved in debugging processes.

**The amount of commands a programmer sends to the Master of a shared-screen session is similar to the amount of commands a programmer executes herself when using CDB.**

**Impact of control policies.** The leftmost bar in Figure 7.7 shows how much programmers participating in SS debugging sessions missed that everyone can directly issue commands to the debugger. A high variability range suggests that lacking direct control is perceived differently by different people. An obvious guess would be that control is missed the most by who does not have it, that is programmers other than the master in SS scenarios. A $U$ test does not, however, give any support to this guess ($U = 81.5$ and $p = 0.32$). Therefore, the explanation may have more to do with general attitudes towards collaboration; but further investigation is needed to answer conclusively.
Regarding sessions under the CD collaborative scenario, the participants slightly preferred the peer control policy, where everyone can issue commands at any time. The difference is, however, small and not statistically significant: a 2-group Wilcoxon signed rank test (the data is paired) gives $V = 97$ and $p = 0.37$.

In a CDB debugging sessions, there is no preference for using master mode over peer mode or vice versa.

**Involvement and control.** Another set of questions asked what was the level of involvement during the debugging sessions. To see to what extent involvement is influenced by the role and by the debugging scenario (SS vs. CD), Table 7.9 breaks down the data into: SS sessions for programmers in the master role; SS sessions for programmers in another non-master role; CD sessions, where the role of master is interchangeable or collective. With statistical significance, the master is typically more involved than the others in SS debugging sessions ($U = 183$ and $p < 10^{-3}$); and the latter are typically less involved than anyone participating in CD debugging sessions ($U = 155.5$ and $p = 0.02$).

The difference between programmers in CD sessions and masters in SS sessions is, instead, borderline statistically significant ($U = 199.5$ and $p = 0.08$). In all, there is some evidence that a collaborative approach such as
that offered by CDB makes for a more uniform distribution of involvement, which is an important goal in collaborative activities.

\begin{table}
\centering
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
& # & min & median & max & mean & $\sigma$ \\
\hline
SS: master & 11 & 3 & 4 & 5 & 4.56 & 0.69 \\
SS: other & 19 & 2 & 3 & 5 & 3.16 & 0.90 \\
CD & 27 & 2 & 4 & 5 & 3.89 & 0.69 \\
\hline
\end{tabular}
\caption{Involvement of programmers: during SS sessions when in master role; during SS sessions when not in another role; during CD sessions. Data on a 1–5 scale.}
\end{table}

The study provides evidence that a collaborative debugging approach like CDB allows for better involvement of all participants taking part in the debugging session.

### 7.5.3 Debugging Experience: RQ3

The third research question looks at the overall collaborative debugging experience with remote-desktop (SS scenarios) and with our tool CDB (CD scenarios).

Among the 19 programmers who tried both SS and CD (group G3), 17 (or 89%) claimed to prefer the experience with CDB over the interaction using remote-desktop. A related question asked to rate the efficiency of the process in each of the two scenarios. Table 7.10 shows the answers. The difference is still in favor of debugging using CDB, with good statistical significance: $U = 230$ and $p = 0.0037$.

\begin{table}
\centering
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
scenario & # & min & median & max & mean & $\sigma$ \\
\hline
SS & 30 & 1 & 3 & 4 & 2.57 & 0.86 \\
CD & 27 & 1 & 4 & 5 & 3.27 & 1.10 \\
\hline
\end{tabular}
\caption{Process efficiency in each scenario.}
\end{table}

In all, the data gives us some confidence that, even if other variables are not greatly affected, CDB is a step in the right direction of supporting collaborative debugging.

Study participants prefer the collaborative debugging with CDB over a shared screen session and perceive CDB to be the more efficient collaborative debugging approach.
7.6 Threats to Validity

Internal validity. A few shortcomings in the execution of the empirical study (Section 7.5) constitute potential threats to internal validity. We could not always enforce a strict time limit to complete each assignment, mainly due to misunderstandings arising with the geographically distributed programmers who took part to the study. While most sessions completed within one hour, a few went on for longer than two hours. We minimized the impact of this problem by using neither time nor tasks completed as measures in the evaluation, even though we collected this data. Another inconsistency occurred with three two-person teams working under the SS scenario: members of the same team could not use different computers, so they simply sat at the same terminal working together. In these situations, we still applied the protocol that limited their communication to verbal (as if they were connected by Skype) and forbade them from pointing to the screen, or sharing the control of the keyboard or any other input device. This should have limited the impact of the threats to validity in this case. A general limitation of the study originates from the usage of CloudStudio as web-based IDE. Since CloudStudio still is a research prototype, it lacks advanced IDE functionalities, and is not immune from bugs (in particular, the real-time synchronization mechanisms may transiently lose responsiveness due to imperfect load balancing). These may affect how programmers rate their debugging experience. However, we performed all debugging sessions (SS and CD) using CloudStudio, so as to have a common baseline and only evaluate differences relative to it.

External validity. Though we designed the empirical study’s tasks to resemble real-world debugging scenarios, the code examples were necessarily limited in complexity and size, and the bugs to be fixed were introduced on purpose. Such limitations apply to all “laboratory” studies of programming activities, as there is no simple recipe to guarantee that in-the-small tasks are indicative of real-world programming. While debugging is a complex multifaceted process, which we cannot expect to understand with a single empirical study, in-the-small studies such as ours can help single out important factors, which can then be assessed more thoroughly in follow-up larger-scale studies.

7.7 Related Work

As far as we know, this chapter’s contribution is novel, both in presenting a new approach and supporting tool for collaborative debugging where distributed users share a common debugging session in real-time; and in performing an empirical evaluation of debugging in collaborative settings.
This section describes the most relevant related work in two areas: tools for distributed software development (DSD) and the features they offer for debugging; and empirical studies of DSD.

7.7.1 Tools for Distributed Software Development

The arsenal of tools for distributed software development is quickly expanding, and also includes mature commercial tools such as Microsoft Team Foundation [41] and IBM Jazz [18]. These tools support various aspects of the collaboration between developers, such as sharing code and documentation in the early implementation phases. They are also well integrated with the corresponding IDEs—VisualStudio and Eclipse in the case of Team Foundation and Jazz. The VisualStudio debugger offers some support for collaboration: one developer can freeze a debugging session running on her machine and transfer its state to another computer running VisualStudio; there, it can be restored and continue with another user. To be able to take over a debugging session, the two users must have access to the same codebase. IBM Jazz offers a similar debugging functionality.

Such approaches to “transferable” debugging are useful but not truly collaborative, as they offer no support for the real-time sharing that may be needed to have faster and more directed interactions. Our CDB tool also supports transfer of control during debugging sessions, but in real-time without requiring that a session be frozen, transferred, and restored.

With IDEs such as CodeRun [21], Cloud9 [19], and Collabode [45], tools for software development have been following the general trend of moving to the web. These IDEs offer functionalities similar to traditional IDEs, but are usable without installation through a web-browser. Even if the code is stored on a shared server and accessed transparently, every user works on a logically different copy of the code, through standard configuration management practices. Cloud9 and Collabode also support real-time collaboration: multiple users simultaneously edit the same piece of code, as if they were working on a GoogleDoc shared document. None of these web-based IDEs offer collaborative debugging functionalities.

JS Bin [103] is a web-based collaborative tool for developing JavaScript programs, which supports a form of collaborative debugging. The collaboration is achieved by publishing a URL where the current debugging session is shown, providing an experience similar to using remote-desktop applications (as illustrated in Section 7.2 and referred to as SS scenario in the empirical study of Section 7.5). DebugLive [107] offers similar functionalities for passive collaboration. With both tools, only one user is in charge of browsing the code, adding or removing breakpoints, and issuing other commands to
the debugger; the other participants can only watch and give suggestions.

7.7.2 Empirical Studies of Collaborative and DSD

Distributed software development has become a standard practice in today’s software industry, one with many challenges [14, 68, 55]: differences in time zones and cultural backgrounds, increased difficulties of performing requirements engineering, project management, and API design, just to mention a few. Some of these challenges have been investigated empirically. For example, the effect of time zones on various phases of development [53, 30, 75]; the relation between development processes and distribution [36]; the effects on productivity and quality [92, 10]; the usage of contracts for API design [78]; and the impact of geographic dispersion on quality metrics [93].

To our knowledge, there is no study about the collaborative aspects of debugging processes and tools such as those discussed in the present chapter.

7.8 Summary

This chapter presented CDB, a debugging technique and integrated tool to support effective collaborative debugging. We evaluated collaborative debugging—in general and with the CDB approach—through an empirical study whose main findings are:

- The two most critical features useful to improve collaborative debugging are: the possibility of browsing code independent of collaborators, and of changing the watched variables in the running debugging sessions.

- Collaborative debugging tools, such as CDB, that allow collaborators to easily switch the role of who is in control of a debugging session achieve more involvement of all debugging session participants.

- Collaborative debugging with CDB is perceived as more effective than the alternative of sharing a single-user debugging session with only indirect interactions possible.
Collaboration, carried out in an efficient and effective way, is a key ingredient to successful distributed software development. This thesis presented several quantitative analyses of distributed projects, identifying factors that hamper, are negligible for, or enable successful collaboration. It also proposed new approaches for coordinating and reconciling program changes and investigated novel ways of collaboratively debugging software.

We’ve analyzed the role of communication in GSE, specifically how time zones and geographical distribution affect communication between teams. Our findings showed that almost a quarter of the overall project effort is invested into communication with other teams and that communication behavior is affected by the amount of distribution. The observation that communication behavior changes with more distribution motivates the research and development of tools and techniques that minimize non-essential communication — as, for example, in the case of merge conflicts or missing information about ongoing changes — that could be reduced if our collaboration tools were more effective. Our study about merge conflicts and change awareness in distributed projects provides evidence that developers face such questions on a regular basis and typically choose to act, i.e. initiate communication, based on the lack of information at hand. These findings reinforce the need for research and development of awareness tools. With the CloudStudio project described in this thesis we enhance the current state of IDEs and configuration management systems. CloudStudio combines the ideas of traditional software version control systems, awareness tools and merge conflict detection tools in a novel approach, and our prototype implementation demonstrates our vision of a modern collaborative development environment. Building on the ideas of CloudStudio, we’ve shown that it is worthwhile to account for specific concepts of programming languages or methodologies.
(design-by-contract in our study) to build even more powerful awareness tools.

Our results on collaborative debugging demonstrated that there are still aspects of collaborative development that are not widely researched and where simple and yet innovative concepts, such as individual browsing of source code, can have a huge effect on how multiple developers work together.

Finally, our investigation on the impact of choice of development process did not provide any evidence that process alone contributes significantly to any of the manifold factors that determine the success of a distributed project.

**Future work.** Empirical studies are essential to this thesis’s contributions. In some cases, our studies and findings complement the work of other researchers and add to our communal understanding of the characteristics of distributed projects. In other cases, our empirical findings are surprising—as in the case of the study comparing agile and structured development processes—and in contrast to common belief. Such empirical results ask for more research that could aim to either replicate the study, or to address the same research questions from different angles, using different study designs and different data sources.

The tools we have implemented in the CloudStudio IDE, namely conflict-detection, awareness and debugging, are research prototypes. They are sufficient to evaluate our ideas, to demonstrate the underlying approach to collaboration, and run to experiments to check our research hypotheses. An obvious next step is to adapt these ideas and ensure that they scale to large development projects and large numbers of developers. Such performance optimizations will bring the benefits of this research to a wider audience.


[57] Hanna Hulkko and Pekka Abrahamsson. A multiple case study on the impact of pair programming on product quality. In Proceedings of
the International Conference on Software Engineering (ICSE), pages 495–504, 2005.


[95] Anita Sarma, David F. Redmiles, and Andre Van der Hoek. Palantir: Early detection of development conflicts arising from parallel code


