Change-centric Improvement of Team Collaboration

Doctoral Dissertation submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Lile Palma Hattori

under the supervision of
Prof. Dr. Michele Lanza

February 2012

Dissertation Committee

Prof. Dr. Matthias Hauswirth	Università della Svizzera Italiana, Switzerland
Prof. Dr. Mehdi Jazayeri	Università della Svizzera Italiana, Switzerland
Prof. Dr. Premkumar T. Devanbu	University of California, United States of America
Prof. Dr. Arie van Deursen	Delft University of Technology, The Netherlands

Dissertation accepted on 13 February 2012

Prof. Dr. Michele Lanza
Research Advisor
Università della Svizzera Italiana, Switzerland

Prof. Dr. Antonio Carzaniga
PhD Program Director

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Lile Palma Hattori
Lugano, 13 February 2012

To Lauro

Abstract

In software development, teamwork is essential to the successful delivery of a final product. The software industry has historically built software utilizing development teams that share the workplace. Process models, tools, and methodologies have been enhanced to support the development of software in a collocated setting. However, since the dawn of the 21st century, this scenario has begun to change: an increasing number of software companies are adopting global software development to cut costs and speed up the development process.

Global software development introduces several challenges for the creation of quality software, from the adaptation of current methods, tools, techniques, *etc.*, to new challenges imposed by the distributed setting, including physical and cultural distance between teams, communication problems, and coordination breakdowns. A particular challenge for distributed teams is the maintenance of a level of collaboration naturally present in collocated teams. Collaboration in this situation naturally drops due to low awareness of the activity of the team. Awareness is intrinsic to a collocated team, being obtained through human interaction such as informal conversation or meetings. For a distributed team, however, geographical distance and a subsequent lack of human interaction negatively impact this awareness.

This dissertation focuses on the improvement of collaboration, especially within geographically dispersed teams. Our thesis is that by modeling the evolution of a software system in terms of fine-grained changes, we can produce a detailed history that may be leveraged to help developers collaborate. To validate this claim, we first create a model to accurately represent the evolution of a system as sequences of fine-grained changes. We proceed to build a tool infrastructure able to capture and store fine-grained changes for both immediate and later use. Upon this foundation, we devise and evaluate a number of applications for our work with two distinct goals:

1. *To assist developers with real-time information about the activity of the team.* These applications aim to improve developers' awareness of team member activity that can impact their work. We propose visualizations to notify developers of ongoing change activity, as well as a new technique for detecting and informing developers about potential emerging conflicts.
2. *To help developers satisfy their needs for information related to the evolution of the software system.* These applications aim to exploit the detailed change history generated by our approach in order to help developers find answers to questions arising during their work. To this end, we present two new measurements of code expertise, and a novel approach to replaying past changes according to user-defined criteria.

We evaluate the approach and applications by adopting appropriate empirical methods for each case. A total of two case studies – one controlled experiment, and one qualitative user study – are reported. The results provide evidence that applications leveraging a fine-grained change history of a software system can effectively help developers collaborate in a distributed setting.

Acknowledgements

A PhD is a long and though journey, with plenty of ups and downs, and that can only be accomplished with the support of many people. I am deeply grateful to have had the support from several different people in this challenging four-year journey.

I would like to thank my advisor Michele Lanza for his ever-present guidance. Thank you for all the constructive discussions, for always being available for a talk, and for trusting that I would keep myself in the right track, even with so many trips across the Atlantic. Over the years, I have learned that behind this strongly opinionated and sometimes controversial person hides a very generous heart that embraces his students as his own family. Thank you for accepting the challenge of guiding this hard headed person here.

I thank the members of my dissertation committee, Mehdi Jazayeri, Matthias Hauswirth, Premkumar T. Devanbu, and Arie van Deursen for dedicating their time to evaluate my thesis, as well as helping me throughout this journey. Mehdi, thank you for providing your opinion each time I approached you with a new idea. Matthias, thank you for the valuable feedback you have given since the prospect review, and thank you for letting me execute my experiment in your course. Prem, thank you for providing such a detailed feedback on my dissertation. Arie, thank you for receiving me in Delft, where I had a great time and was able to learn so much with our discussions.

Many thanks to my former colleagues from the REVEAL research group, who were immensely important, each in their own ways. Romain, thanks for the very proactive postdoc attitude – I'm sure you are now a great professor. Mircea, thanks for bringing merriness to the office every day. Ricky, thanks for keeping the office full of atmosphere with your incredible jokes. Marco, thanks for always having a solution for everything, even when it came with a joke. Alberto, thanks for being this amazingly kind and humble person, always ready to help everyone. Fernando, thanks for all your unexpected demonstrations that a PhD student can be a laid-back, yet competent person. Special thanks to two ladies I'm deeply fond of: Anja and Sylvie. I am so grateful for having been given the chance to co-supervise your master theses, because I have gained two super smart friends who I'm so proud of.

I am also grateful to the people I have met and have had a lot of fun together: Adina (in memoriam), Alessandra, Amir, Anna, Antonio, Chrysa, Cyrus, Daniela, Giacomo, Giovanni, Katharina, Marco, Mehdi, Morgan, Mostafa, Nicolas, Paolo, Parisa, Parvaz, Ricardo, Sara, Sebastian, Shima, Simi, and to you who felt your name was missing from this list. Thanks to all the professors, secretaries, and staff from the Faculty of Informatics. I am thankful for having met many special people during conferences and my visits to research groups; people who have helped me with my research by sharing their opinion or opening the doors of their labs to one of my experiments. My sincere thanks to all the volunteers and those who helped me organize my never-ending experiments.

Most of all, there are three people who were always on my side, giving me strength and confidence throughout this entire journey, and who I dedicate this work to: Lauro, and my parents, Ligia and Likiso. Mom and dad, you are the best role models I could have ever asked for. Thank you for always being there for me whenever I needed. Lauro, your unconditional love and dedication inspired me daily to keep moving forward. No words can describe my love and gratitude to you.

Lile Hattori
February 2012

Contents

Figures

Tables

Part I

Prologue

Chapter 1

Introduction

Team collaboration is intrinsic to the production of quality software. The creation of software has been mainly a team effort since the first software development company was founded [Kubi 94]. In the early days, before either tool support or formal process were available, and when methods of communication were scarce, knowledge about the software under construction was kept with developers and in formal documentation.

A few decades later, software is everywhere, and software development has had to evolve to keep up with the dynamism of the modern world. Software development methodologies such as the traditional spiral model [Boeh 86], or the lighter and more people-centric Extreme Programming (XP) [Beck 01; Beck 04] and Scrum [Risi 00], have been widely adopted by software development companies. Great advances have also been made with software development tools, such as integrated development environments (IDEs) [Ecli 11; Micr 10; Orac 11], software configuration management (SCM) systems [Estu 05; Chac 09; OSul 09; Pila 08; Vesp 06], and bug tracking systems [Mozi 11; Mant 11; Atla 11].

The advent of the Internet and the improvement of network bandwidth since the mid-nineties have enabled the adoption of a distributed software development model (also known as global software development). This model was first embraced by the free/libre/open source community (FLOSS) [DiBo 99; Will 02; Raym 01], and was later adopted by software development companies, who envisioned offshoring and outsourcing as a means to cut down costs [Carm 99; Sang 07]. This distributed software development environment has introduced new challenges to software engineering (*e.g.* strategic and cultural issues, new communication needs, and knowledge and process management issues), and has created a new discipline: Global Software Engineering.

Today, with the growing complexity of software systems, the urge for faster time-to-market, and the new challenges of managing distributed software teams, the global software engineering community has drawn significant attention to the problem of low team coordination and collaboration [Herb 00; Hegd 08; Fros 07; Dour 92; Silv 06; Dami 07; Sarm 08]. Team coordination has been negatively impacted in several ways, affecting the degree of knowledge a developer can access and use to accomplish their work. For instance, the physical distance between members of globally distributed software teams reduces communication and affects developers' tacit knowledge of the current activities of the project [Alle 77; Herb 00]. Cultural differences among sub-teams, such as differences in the proper procedure of communicating a change, can cause breakdowns in team coordination [Herb 01; Dami 07]. The diversity of communication options

currently used (*e.g.* formal meetings, face-to-face interactions, multi-point video conferencing and telephone calls, emails, Internet relay chats (IRC), instant messaging (IM), online forums, and social media) often spread project information over multiple sources [Herb 01].

Researchers have observed that disregarding the importance of team collaboration can cause several problems, leading to delays [Herb 00; Dami 07]. There are several aspects of team collaboration that must be taken into consideration, including:

Management of the team structure. This is an integral task that provides the foundation for a team's collaboration and communication dynamic. Team structure can vary greatly depending on several influential factors. Teams can be collocated or globally distributed. The latter type has the option of adopting a central or globally integrated coordination approach. Team allocation may be module-based – with each sub-team responsible for implementing a module of the system –, phase-based – with sub-teams covering different responsibilities over the entire code base –, or integrated – the entire team shares multiple responsibilities over the entire code base. A team is usually divided into sub-teams, and the more hierarchical the structure is, the harder it is for developers of different sub-teams to collaborate.

Team awareness. This is defined as an understanding of the activities of others, providing a context for one's own activities [Dour 92]. Awareness can be seen as the degree to which team members are aware of the work of others that is interdependent with their current tasks. Raising awareness therefore enables better team coordination of teams [Dami 07]. In a collocated team, awareness is mainly obtained through human interactions, which may include meetings, informal conversations, or helping colleagues. This is specifically called workspace awareness [Gutw 96], because it is the knowledge acquired as consequence of a shared physical workspace. In a distributed team, however, the distance between developers or teams constitutes a barrier for informal interaction. Workspace awareness in this environment is consequently lower than that of collocated teams.

Knowledge management. This discipline has been adopted by approximately 80% of the largest global corporations [Rus 02]. In the field of software engineering, there are several knowledge needs, including domain knowledge; knowledge about new techniques; and knowledge about local laws, policies, and practices. With the dynamism and high turnover of today's development teams, it is important that developers collaborate and share their knowledge.

Team collaboration can be severely hindered when any of the aforementioned factors are overlooked. For instance, the distance of geographically distributed teams can cause a decrease in communication between team members [Sang 07], consequently affecting team awareness. Similarly, flawed team allocation may isolate sub-teams or individuals. Consequences range from low willingness to help out [Herb 00; Grin 96] to change in the behavior of developers, such as a tendency to rush to check in changes to avoid merging code [Souz 03]. The lower a team's awareness, the more isolated its developers become, and the more difficult it is for them to share knowledge. There are several ways to promote team collaboration, such as increasing the number of communication strategies or venues that a team uses, or the adoption of tools to manage collaboration, knowledge, and coordination.

We focus on leveraging software change history to address two of the three aforementioned aspects of team collaboration: team awareness and knowledge management. For the latter, we

focus on providing tools for developers to acquire useful information that can be extracted from the source code change history.

The mining software repositories (MSR) research community has explored the source code change history, among other repositories (e.g., bug tracking repositories or email archives), for various purposes [Mock 00; Gall 03; DAmb 09; Girb 05a; Holt 96; Xing 05; Rapu 04; Lung 07; Wett 08; Flur 07b]. These approaches commonly leverage source code change history stored in mainstream SCM systems. The software systems that these approached have analyzed had often passed the initial phase at the time of the analyses. This means that these systems already had sufficient change history in terms of number of commits to yield valuable results for the analyses.

One concern is that very few approaches are targeted at using change history on the fly to help developers during coding activities [Zimm 05; Bacc 10], because usually the data (*e.g.* SCM commits) is too coarse-grained to provide immediately valuable information to developers.

We want to effectively support team collaboration on the fly and at any stage of the software lifecycle. To do this, we need to use a change history that is finer-grained than SCM commits. Thus, we first propose an approach to model the evolution of software systems in terms of its fine-grained source code changes. We then leverage the data produced by our model to help developers maintain a high level of awareness of the activity of the team, and to acquire knowledge about the evolution of the system and the team dynamics, all within a collaborative context.

We formulate our thesis as:

Modeling the evolution of a software system in terms of fine-grained changes produces a detailed history that can be leveraged to support developer collaboration.

With the aim of helping developers collaborate, share knowledge, and accomplish their work, we intend to extract relevant information from a detailed history of the evolution of a software system. To achieve this, we first extend Robbes's Change-based Software Evolution (CBSE) model [Robb 08a] into our Collaborative Change-based Software Evolution (CCBSE) model. Our CCBSE model represents the fine-grained evolution of a software system by:

1. modeling the state of an object-oriented system at the level of source code entities by defining an abstract syntax tree down to method and instance variable level;
2. modeling changes as tree operations;
3. explicitly modeling developers as part of the evolution history of the system.

Utilizing this model, we elicit the necessary requirements for a tool infrastructure able to track fine-grained changes and store them for both immediate and later use. The tool infrastructure should complement SCM systems, be integrated on the IDE, be non-intrusive and lightweight, and enrich the history of changes. Once developed, the tool infrastructure allows us to evaluate our thesis by proposing and evaluating four applications that leverage the change history for two different purposes:

1. **Increasing real-time awareness.** These applications concern how the immediate use of change information may improve the awareness of the team. We initially focus on showing developers information about who is currently (or has recently been) changing to the system, and which parts of the system they have been working on, at the level of

object-oriented classes. This information allows them avoid duplicated work by identifying whether two or more people are working on the same parts of the code, in addition to granting them the ability to quickly identify people knowledgeable about specific artifacts, and in general be more aware of the current activity of the team.

2. **Assisting information acquisition.** These applications explore the fine-grained source code change history to assist developers find the information necessary to conduct their work. The applications are targeted to a distributed development environment, in which information that would usually be acquired through informal interactions becomes difficult to access remotely. We exploit different measurements of code expertise to help developers find experts, and provide a user interface for developers to explore and understand past changes of the system.

We validate our thesis by individually evaluating each proposed application. We use several methodologies for the evaluation: case studies, a qualitative user study, and a controlled experiment. We report on the design and the analysis of the results in Part III (p.59) chapters.

1.1 Contributions

This dissertation's contributions to the field of software engineering can be classified into three categories: models and tools, applications and techniques, and evaluations.

Models and Tools

1. **The definition of the Collaborative Change-based Software Evolution (CCBSE) model** [Hatt 09a; Hatt 10a; Hatt 11a] to represent the detailed evolution of a shared software system in terms of fine-grained source code changes. The CCBSE model applies the reification principle on changes to object-oriented systems, forming a detailed history of their evolution.

2. **The creation of the Syde toolset** [Hatt 09a; Hatt 10b; Hatt 11a]. We implement a collection of tools to support the CCBSE approach, allowing for the detailed evolution of a system to be stored and used to help developers accomplish different tasks. Syde is publicly available as an extension to the Eclipse IDE.

Applications and Techniques

3. **The application of the CCBSE approach to awareness of team activity** [Lanz 10]. We propose different visualizations, integrated on the IDE, to deliver information on the team activity in real time. The visualizations always show the most recent activity by updating themselves in a non-intrusive way.

4. **The application of the CCBSE approach to preemptive conflict detection** [Hatt 11d]. We defined a technique to detect potential conflicts that might emerge due to concurrent modifications of source code elements, and to notify developers of their existence before check-in time. This information allows developers to take preventive action to avoid complex merging at check-in time. In addition, we devise different ways to deliver such information in the Eclipse IDE, including enriching the Java editor with visual cues and building views dedicated to showing emerging conflicts.

5. **The creation of two measurements of code expertise based on fine-grained changes** [Hatt 09b; Hatt 10c]. One measurement takes into account the number of small changes performed by a developer on a source code entity in the IDE. Because it takes into account every edit interaction a developer has with a source code entity, this measurement reflects the effort input by a developer working on an entity more precisely than do the measurements based only on commits. The second measurement considers the natural effect of forgetting to compute code expertise.

6. **The application of the CCBSE approach to understanding software evolution** [Hatt 10d; Hatt 11c]. We devise an application to allow developers explore the rich history of changes created with the use of the CCBSE approach. By investigating past changes, developers acquire more knowledge to answer different program comprehension questions related to the evolution of the system.

Evaluations

7. **The empirical validation of our preemptive conflict detection application through a qualitative user study** [Hatt 11d]. We design and conduct a qualitative user study to understand how the behavior of developers is influenced by the presence of preemptive conflict detection. We report on the design and analysis of the data collected from several different sources (questionnaires, observations, interviews, documents). In addition, we publish the experimental data to facilitate the execution of similar studies.

8. **The empirical validation of our replay application through a repeatable controlled experiment** [Hatt 11c]. We design a controlled experiment for the empirical validation of our replay application and conduct it with participants from diverse educational and technical backgrounds. We provide a full report on the design, operation, and analysis of the data from this experiment. Moreover, we publish the entire experimental data set and everything required to make the experiment replicable and transparent.

1.2 Structure of the Dissertation

The remainder of this dissertation is organized as follows.

Part I: Prologue. In the remainder of this part, we place our work in the broader context of collaborative software engineering and software evolution.

- In **Chapter 2** **(p.11)** we lay down the historical context of our research. We then review the state of the art of software evolution. We start by discussing the state of the art of software configuration management. We review approaches that analyze the evolution of a software system, discussing the benefits and drawbacks of different granularity levels of data analysis. We conclude by reviewing existing change-based approaches that place change in the center of the development process by modeling it explicitly.

Part II: Supporting Team Collaboration. In this part of the dissertation, we present the models and tools we devised to support team collaboration.

- In **Chapter 3** (p.29) we present the foundation of our work: the Collaborative Change-based Software Evolution (CCBSE) model. We first discuss the principles of the CCBSE model, and then describe how the model is designed to represent a software system and its evolution. We finish by discussing the CCBSE model's uses in the field of collaborative software development.
- In **Chapter 4** (p.41) we describe Syde, the tool infrastructure that supports the use of the CCBSE approach in practice. We start by determining the requirements for building a tool able to manage fine-grained changes performed by multiple developers building a software system. We present the architecture of Syde and discuss the strategies used to gather changes in the IDE. To conclude, we present the applications that use the change history information collected by Syde to accomplish multiple goals.

Part III: Applications to Support Team Collaboration. In this part, we validate our approach by applying it to help developers in different activities. We describe two possible uses for the change history collected by Syde: real-time use, and change evolution analysis.

- In **Chapter 5** (p.59) we explore how different visualizations may be used to inform developers about the current activity of the team. This includes displaying which team members are changing the system, and in which parts of the system they are currently (or have recently been) working. We present the visualizations and discuss the results of the case study performed to evaluate the usefulness of the different visualizations.
- In **Chapter 6** (p.77) we investigate how the behavior of developers changes when they are exposed to preemptive conflict detection. We first discuss the empirical studies conducted so far and explain how our study complements them. We present our preemptive conflict detection algorithm devised to detect potential direct and indirect conflicts at an earlier stage than check-in time. We describe the design of the qualitative user study we conducted, and perform a meticulous analysis of the data, concluding that preemptive conflict detection is useful for developers working in a collaborative context.
- In **Chapter 7** (p.125) we describe how we use the change history collected by Syde to refine code expertise measurements and help developers on find experts on artifacts in the system. We first discuss techniques previously proposed by researchers, and then present two new measurements based on Syde's fine-grained changes. We evaluate our measurements by comparing them to a specific measurement that uses SCM information, and observe that our measurements reflect a developer's effort more precisely than the SCM-based measurement.
- In **Chapter 8** (p.151) we present a controlled experiment we conducted to measure whether Syde's change history can help developers to answer common questions that arise while they are working. We explore questions that can benefit from our detailed change history. We present the design of the controlled experiment and report on the results, showing that Replay (Syde's application to exploit past changes) leads to a decrease of completion time compared to the use of an SVN client to explore the SVN history.

Part IV: Epilogue. In this part, we examine and draw conclusions about the work as a whole.

- In **Chapter 9** (p.179) we reintroduce the context of our thesis to reflect on the contributions this work makes to the field of collaborative software development. We discuss the general

limitations of our work and the directions in which it might be advanced in the future, and conclude.

Part V: Appendices. This part contains the experimental data of the qualitative user study and the controlled experiment conducted as part of the evaluation of our thesis.

- In **Chapter A (p.189)** we publish the experimental data of the qualitative user study reported in Chapter 6 (p.77) to facilitate the execution of similar studies.
- In **Chapter B (p.197)** we publish the entire experimental data set and everything that is required to make the experiment presented in Chapter 8 (p.151) replicable and transparent.

Part VI: Bibliography. The last part lists the bibliography used in this dissertation.

In the next chapter we review the state of the art of software evolution. First, we look at the history of software development with emphasis on software evolution, software configuration management, and collaboration. We discuss the state of the art of software configuration management and review applications that analyze the evolution of a software system. We focus on reviewing approaches that place change in the center of the development process by modeling it explicitly.

Chapter 2

State of the Art

2.1 Introduction

In our dissertation we explore how team collaboration during the software development life cycle can be enhanced by exploiting how the software evolves. The term “software evolution” became known after the work of Lehman and Belady [Lehm 85], who identified a set of behaviors in the evolution of software. These behaviors are known as Lehman’s Laws, and among them are the observations that a software system must continuously change to remain satisfactory, and that its complexity tends to increase and its quality decrease unless maintenance is done to prevent it. Parnas named this degrading phenomenon “software aging” [Parn 94]. Nierstrasz observed that the only way to control software aging is to place change and evolution in the center of the software development process [Nier 04].

For the last twenty years, researchers have investigated the software evolution phenomenon by exploring the history produced by software development support tools. Different repositories containing the development history of a system exist, including bug tracking archives, email, other communication archives, and software configuration management (SCM) repositories. Among these, SCM repositories remain the most explored by the mining software repositories (MSR) research community [Hass 08].

Our approach reifies changes by modeling them as concrete objects, allowing change and evolution to take a central role in the software development process. Consequently, our approach creates and exploits a new source code repository. Therefore, SCM repositories and approaches that explore source code repositories are closely linked to our work. This chapter will therefore review the literature on SCM systems and discuss approaches that explore source code repositories for the purpose of software evolution analysis.

2.2 Genesis of our Approach

Before reviewing the approaches that analyze source code evolution of a software system, it is important to understand why we rely on source code evolution to improve team collaboration. The two share a tight relationship that has been present since the creation of the first programs. We look at the history of software development, focusing on the relation between source code evolution and team collaboration.

2.2.1 Pre-Eighties

In the 1940s, the earliest days of software development, the term "program" was not yet associated with software. Programmers individually wrote routines to do specific computations. These routines could be combined to achieve more complex goals, and through this process the first software programs were born [Hopp 80]. Even in these early days evidence can be found for the relationship between software development and team collaboration. On an interview for the Computer History Museum, Hopper's comments of her first days working as a programmer in 1944 provide evidence that collaboration already existed in the form of code exchange: "If I needed a sine routine, angles less than five over 4, I'd call over Dick Bloch and say 'Can I copy your sine routine?' We'd copy pieces of coding." [Hopp 80].

Two decades later, in the 1960s, the first configuration management (CM) systems were created to place documents under control [Estu 05]. There is little information or documentation available describing the first CM systems, because they were largely integrated into the operating systems of that time, now obsolete. From the available information, it is known that the CMs were originally created to manage textual documents, but it became clear over time that similar techniques could be used to manage any software system.

In 1970s, software configuration management (re-)emerged as a discipline to track and control changes in the software with the introduction of the first SCM applied to software, called Source Code Control System (SCCS) [Roch 75]. From that point on it became feasible for programmers to work on different parts of the same program, though the SCM systems had several limitations, such as an inability to handle concurrent modifications of the same part of a program.

In the same decade, the first effort to make software development a systematic process appeared in the form of the Waterfall process [Royc 70], which introduced a methodology for developers to work in parallel on the production of software. A few more years signaled the appearance of the first argumentation for incremental software development involving continuous user participation [Mill 76], emphasizing the need for greater support for user collaboration and parallel development.

2.2.2 Eighties and Nineties

In 1980, Lehman introduced five laws of software evolution [Lehm 80a; Lehm 80b] based on a study performed to understand the life cycle of IBM's OS 360 operating system. In 1985, Lehman and Belady [Lehm 85] extended Lehman's initial study by confirming the laws of software evolution on other software systems and deriving additional laws from the new data. These studies were the first to analyze the change history of software systems in detail, and to argue that software must continuously change to adapt to the changing world.

The 1980s saw a growth in the popularity of SCM, which took its place as the central point for source code synchronization with the advent of tools such as Make [Feld 79], RCS [Tich 85], and Sablime [Bell 97]. Each system targeted a specific functionality and focused either on version control or on supporting a build process for the generation of an executable program from source files. In 1990, the first version of CVS was released. CVS supported distributed, multi-site and offline operations, and became the most popular SCM for software development for almost two decades.

The initial launch of open source software also occurred during the 1980s. In 1985, Stallman published the *GNU Manifesto*, launching the GNU project and explaining the importance of free software [Stal 85]. The movement gained another important contributor in 1992, when Torvalds

relicensed the Linux project under the GNU General Public License [Torv 92]. In the late 1990s, with the widespread adoption of the Internet, open source software became so popular that researchers started to mine source code repositories. The seminal research works on mining software repositories include the first article to point out the potential of release histories for software evolution support tools [Gall 97], the work on how to find clusters of files frequently changed together [Ball 97] and the approach to infer the effort to make a change [Grav 98].

Regarding development process models, in the late 1980s Boehm proposed the spiral model [Boeh 86], which considered iterative risk analysis as part of the process. This was particularly suited to building large-scale complex systems. The advances on the SCM and process model front in these two decades were pushed by pressure in the ever-changing business environment, advances in hardware and software technology, and demands for more tool support for the development and maintenance of software systems. Collaborative software development was starting to become a reality, pushing the advances on SCM even further. The Capability Maturity Model (CMM) [Paul 93; Paul 95] was established in the 1990s as an important framework of recommended practices to enhance software development and maintenance. The CMM emphasizes the importance of SCM by stating that unless the team utilizes a change management system as a basic tool, the development environment will become chaotic.

2.2.3 The 21st Century

Today, with the popularity of the Internet, low disk storage price, and fast CPUs, the major concerns regarding SCM systems have changed, but been accompanied by great advances. For instance, deltas are no longer required to save space; new SCM systems simply adopt compression. One important advance was in the surge of distributed version control systems, such as Git [Chac 09] and Mercurial [OSul 09], motivated by the highly distributed development of open-source systems.

Development process models have also seen considerable innovation, with the appearance of evolution as a stage in the software life cycle [Rajl 00] and the launch of the Agile Manifesto [Beck 01], along with further establishment of agile methodologies such as eXtreme programming [Beck 04] and Scrum [Risi 00]. These methodologies aim to speed up the development process by improving software quality and responsiveness to changing customer requirements.

On the research front, software evolution has become a well-established research field in software engineering, and mining software repositories has matured into a research area of its own. The first International Workshop on Mining Software Repositories (MSR) was held in 2004 [Hass 04]. In ensuing years, the topic has gained increasing attention in the field, with researchers exploiting different types of repositories (*e.g.* versioning systems, bug tracking systems, email archives) to better understand the software evolution phenomenon and to propose methods and techniques to aid software development and maintenance.

Finally, with the dynamism of today's market, software development companies are increasingly adopting global software development, targeting offshoring and outsourcing as methods of cutting costs [Carm 99; Sang 07]. Global Software Engineering (GSE) is surging as a discipline of its own, introducing numerous challenges to the developing environment such as physical and cultural distances between teams, communication problems, and breakdowns in coordination.

In our dissertation, we focus on the challenge of improving collaboration within development teams, especially within geographically dispersed teams. Instead of mining existing repositories, have chosen to create and mine a new source code repository containing a more detailed change history than the ones produced by current mainstream SCM systems. In the next section we

review the state of the art related to our proposed model to create a fine-grained source code repository.

2.3 Software Configuration Management

Software Configuration Management is the discipline of managing change in software systems, and is thus an important discipline for professional software development and maintenance [Estu 05]. It has seen great advancement over the last three decades to accompany the increase in the complexity of software process models. SCM systems have diverse features varying from configuration identification and control to build and process management. The characteristics that directly relate to our approach are change versioning and workspace management. We discuss both research prototypes and tools used in practice that fit the aforementioned scope by identifying their characteristics and comparing their differences.

2.3.1 Version Space of SCM

The different available paradigms of versioning have distinct characteristics. Conradi and Westfechtel [Conr 98] conducted an extensive survey that provides an overview and classification of different versioning paradigms, relating fundamental concepts such as revisions, variants, configurations, and changes. This section focuses instead on the following characteristics:

- **Generic versus domain-specific versioning.** A generic versioning system uses a common method to version any type of resource, which is treated as text or binary file. A domain-specific versioning system, for example one that is language-dependent, takes advantage of knowing the common characteristics of the domain to use a more precise versioning method.

  Since software systems are seldom written in only one programming language, the versatility of generic versioning systems constitutes a great advantage in practice over domain-specific ones. Most modern SCM systems, such as SVN [Pila 08], Git [Chac 09], and Mercurial [OSul 09], adopt a generic approach. Domain specific versioning systems still have their advantages, however. For instance, they are able to use more precise merge algorithms that take into consideration the syntax of the domain. Early examples of domain specific versioning systems are Gandalf by Habermann and Notkin [Habe 86], and Perry's Inscape [Perr 87]; a later example is Robbes's Spyware [Robb 08b].

- **State-based versus change-based/operation-based versioning.** In state-based versioning, versions are described in terms of revisions and variants [Conr 98], with the system storing the states of a resource on a version graph. For greater space-efficiency, only the first or last version of a resource is stored together with the deltas. Earlier or later versions must therefore be computed from the deltas. In change-based versioning, a version is described in terms of the changes applied to some baseline. Thus the changes are stored and the versions of a resource are computed from them [Robb 08a]. Figure 2.1 illustrates the difference between state-based deltas and change-based deltas.

  State-based versioning has been largely adopted by state of practice SCM systems [Pila 08; Chac 09; OSul 09]. Early adopters are SCCS [Roch 75] and RCS [Tich 85]. Change-based versioning has mainly been used by research prototypes, such as ICE by Zeller and Snelting

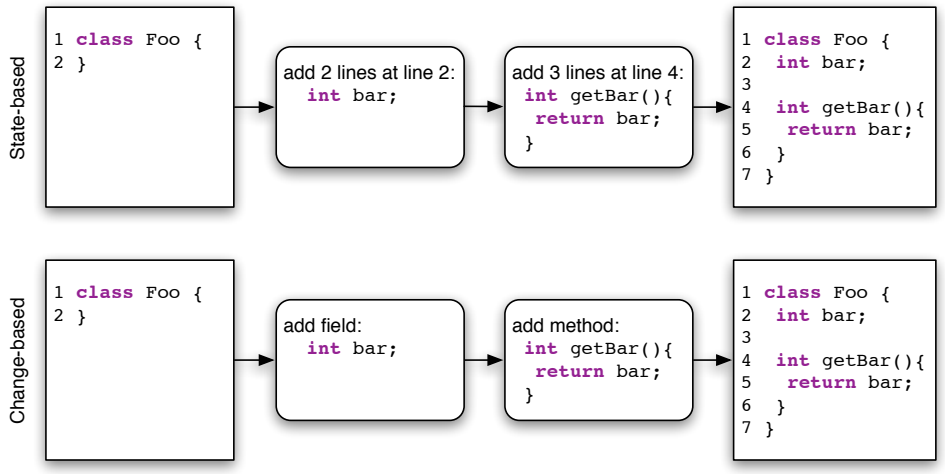


Figure 2.1. Difference between state-based deltas and change-based deltas

[Zell 95], and MolhadoRef by Dig *et al.* [Dig 07b]. One of the advantages of change-based versioning is the the simplicity with which it can express change sets, which can have a logical meaning attached (*e.g.* a change request, or a bug fix). It is also possible for state-based versioning to utilize the concept of change sets, however, as is the case with the recent Rational Team Concert (RTC) Source Control [IBM 11].

Operation-based versioning is a subtype of change-based versioning. This versioning is usually language-dependent (thus domain specific), and takes advantage of knowing the language syntax to version the system as a sequence of syntactic operations. The system is usually modeled at a more fine-grained level than that of files (*e.g.* abstract syntax trees for object-oriented systems), and the sequence of operations can be replayed or rewound to move the system from one state to another. In operation-based versioning, merge conflicts can be resolved with operation-based merge algorithms [Lipp 92]. A few research prototypes that are not full-fledged SCM systems have adopted operation-based versioning. Examples are Robbes's Sypware [Robb 08b], MolhadoRef by Dig *et al.* [Dig 07b], and OperationRecorder by Omori and Maruyama [Omor 08].

Table 2.1 shows a summary of the SCM systems and prototypes cited in this section, plus two additional widely used ones: open-source CVS [Vesp 06], and commercial Perforce [Perf 11]. Each is classified according to the versioning methods discussed above.

The majority of widely adopted SCM systems (the first half of the table from CVS to RCS) is full-fledged (*i.e.* they contain the necessary features to effectively version source code in practice), and adopts generic, state-based versioning. From these, CVS, Perforce, SVN, RTC Source Control, and RCS are centralized, while Git and Mercurial are distributed; only Perforce and RTC Source Control are commercial SCM systems. RTC Source Control is the only full-fledged SCM that implements the concept of change sets. SCCS is a full-fledged SCM system that originated as a research prototype, but follows the characteristics of the widely used SCM systems.

The second half of the table (from Gandalf to OperationRecorder) contains research prototypes that are domain-specific. Gandalf and Inscape are state-based; while ICE, Spyware, and

Table 2.1. Summary of SCM system and prototype characteristics

SCM Name	Full-fledged	Prototype	Generic	Domain-specific	State-based	Change-based	Operation-based
CVS [Vesp 06]	√		√		√		
Perforce [Perf 11]	√		√		√		
SVN [Pila 08]	√		√		√		
Git [Chac 09]	√		√		√		
Mercurial [OSul 09]	√		√		√		
RTC Source Control [IBM 11]	√		√		√		
RCS [Tich 85]	√		√		√		
SCCS [Roch 75]	√		√		√		
Gandalf [Habe 86]		√		√	√		
Inscape [Perr 87]		√		√	√		
ICE [Zell 95]		√		√		√	
Spyware [Robb 08b]		√		√		√	√
MolhadoRef [Dig 07b]		√		√	√	√	√
OperationRecorder [Omor 08]		√		√		√	√

OperationRecorder are change-based and implement the concept of change sets. MolhadoRef combines state- and change-based versioning, with refactoring versioned as change operations and other types of changes versioned as consecutive states.

2.3.2 Workspace Management

Another important topic for our research is workspace management, which defines how users interact with the SCM system. Two aspects of workspace management are of particular interest for us: how users deliver changes, and how they become aware of new changes. In the following section we discuss the most commonly adopted interaction models: pessimistic versus optimistic version control, check-out/check-in, advanced process support, distributed version control, and operation-based version control. We focus on the two aforementioned aspects of interest.

Pessimistic versus optimistic version control. Pessimistic version control allows only one user to access a resource at a time. This user locks the resource before editing it, and releases the lock when committing the new version of the resource. This model eliminates conflicts, but prevents parallel modification of a resource. Optimistic version control allows parallel modification of a resource, relying on the assumption that conflicts are infrequent. However, it is therefore possible for a conflict to occur, support for conflict resolution must be implemented.

In pessimistic version control, a user is only informed about the existence of new code when updating her copy of the system or when trying to acquire the lock of a resource. In optimistic version control, the user will be informed about new code either when updating her copy of the

system or when her attempt to commit a change fails due to merge conflicts. In both cases, the user has to take action to be informed of new changes.

Check-out/Check-in model. This model describes how the user acquires a whole or partial copy of the system, and how modifications are delivered. A user typically checks out a copy of the parts of the system the wants to change, edits them, and checks them back into the repository. This model can be combined with either pessimistic or optimistic version control, and most centralized SCM systems (*e.g.* CVS, SVN, Perforce) support both configurations.

In the check-out/check-in model, a user will only be informed of about new changes when trying to check out, updating her local copy, or committing/locking a resource. The model is therefore similar in this way to both pessimistic and optimistic version controls.

Advanced process support. Some SCM systems support more advanced policies than those of pessimistic and optimistic version control. One such policy is hierarchical broadcast: changes are broadcast to the members of the sub-team as users submit them, and are only published to other sub-teams at a pre-defined rate (*e.g.* daily integration). An early prototype that supported advanced policies is Celine by Estublier and Garcia [Estu 06]; more recent one is the modern centralized RTC Source Control.

Advanced policies open up possibilities for more proactive solutions to notify users of new changes in the repository. RTC Source Control, for instance, has a configuration option to send a notification message to other users as a user submits new changes [IBM 11]. If hierarchical broadcast is on, immediate notifications will be sent only to sub-teams; interested users can also register themselves to be notified.

Distributed version control. Modern decentralized SCM systems (*e.g.* Git, Mercurial, Bazaar [Cano 11]) offer additional support for parallel work. These do not rely on a central repository. Instead, to get a copy of the system, a user has to clone the entire repository locally, with subsequent commits being performed on the user's local repository.

In decentralized version control, commits are local, preventing other users from automatically seeing them. Local commits only become visible to a user after they perform a merge [Bird 09] to commits either pushed by the owner or fetched by the user who is performing the merge. A scalable distributed model therefore comes with the price of lowered user awareness of what is being changed in the system. To counterbalance this, these models provide more advanced support for versioning and merging than centralized SCM systems do. Distributed versioning is widely used by the FLOSS community, and has been gradually adopted by the industry.

Operation-based version control. This alternative model records the changes performed directly in the user's environment. It is employed by operation-based versioning systems, which can have either a local or centralized repository. Changes are automatically recorded as a developer modifies the system, producing a repository that contains all of the changes performed in the system, rather than revisions containing deltas produced only when a developer decides to submit them. Early adopters of this approach were Lippe and van Oosterom in the CAMERA system [Lipp 92], with recent adopters being Spyware [Robb 08b] and OperationRecorder [Omor 08].

This version control model is fundamentally different from the models discussed above: it records changes implicitly, rather than doing it explicitly when commanded by a user. This

fundamental difference allows for the creation of a real-time change notification system that informs other users of what is being modified on the fly.

Discussion. From the interaction models described above all but the last share a common characteristic: users have to explicitly submit their changes for others to be informed about them. It is possible to build a notification system on top of this base functionality, but change notifications can only be broadcast after commit time, which may be too late to prevent some problems (*e.g.* duplicated work, complex merging) depending on the frequency with which users commit changes.

With operation-based version control, however, change notifications can be broadcast in real time. This can be beneficial for alerting the team members of who is working on what; however, it must be possible for a developer to control what is broadcast, as propagating changes from a development session where a developer is experimenting on the code might send a misleading message to her coworkers.

2.4 Software Evolution Analysis with Source Code Repositories

Over the last decades, researchers have proposed techniques for software evolution analysis that explore source code repositories both exclusively or combined with other types of repository. We consider as source code repositories not only those produced by SCM systems, but also any repository, produced by any tool, that contains the source code change history (*e.g.* any repositories produced by monitoring tools connected to the IDE).

The proposed techniques span a variety of different purposes, such as the analysis of source code entities that are logically related [Ball 97; Gall 98; Gall 03; Ying 04; Ratz 05; Beye 06; DAmb 09; Zou 07], analysis of the architectural evolution of a system [Holt 96; Xing 05; Rapu 04; Lung 07; Wett 08], and support for developers' work [Zimm 05; Robb 10a; Robb 10b; Sing 05; DeLi 05; Parn 06; Kers 05; Kers 06]. We discuss the approaches that relate to our thesis by focusing on the advantages and drawbacks of the granularity level of the data analysis in each approach. A summary of the approaches discussed in this section can be found in Table 2.2 (p.19).

2.4.1 Coarse-grained Evolution Analysis

These techniques consider that the meta-data about the SCM transactions (or commits) is sufficient for whatever sort of analysis they perform. The meta-data of each commit is stored in a commit log and usually contains: the name/path and revision number of committed files, the author of the commit, the date, and a message/comment.

Analysis of commit messages. Several researchers have analyzed the information contained in commit messages to infer the activity to which they relate. Mockus and Votta [Mock 00] used word frequency and semantic analysis techniques to classify commit comments according to Swanson's [Swan 76] classification of maintenance activities. Purushothaman and Perry [Puru 05] used the same classification to analyze small changes (one or more modifications to single/multiple lines).

We have proposed a new classification for commit comments and have applied it to open source projects. This classification uses the frequency of keywords to classify a commit into development (forward engineering) or maintenance (reengineering, corrective engineering, or

Table 2.2. Summary of the discussed approaches that analyze source code repositories

Approach	**Repository**	**Change granularity**	**References**
Classification of commits into software lifecycle activities	SCM logs (CVS, SVN)	Coarse	[Mock 00; Puru 05; Hatt 08]
Automatic generation of work description from logs	SCM logs, other logs, notes	Coarse	[Maal 10]
Change coupling analysis	SCM logs	Coarse	[Ball 97; Gall 98; Gall 03]
Predicting related changes based on change coupling	SCM logs	Coarse	[Ying 04]
Change coupling visualizations	SCM logs	Coarse	[Ball 97; Ratz 05; Beye 06; DAmb 09]
Versatile meta-model for evolution analysis	SCM logs, source code, bug reports	Coarse	[Fisc 03; Girb 05a; DAmb 10a]
Analysis of the architectural/design evolution of systems	SCM logs, source code	Coarse	[Holt 96]
Analysis of the design evolution of object-oriented systems	SCM logs, source code	Coarse	[Xing 05]
Recommender to guide changes to correlated source code entities	SCM logs, source code	Coarse	[Zimm 05]
Analysis of design problems over time	SCM logs, source code	Coarse	[Rapu 04]
Analysis of the evolution of relationships between packages	SCM logs, source code	Coarse	[Lung 07]
Visual analysis of the evolution of software systems	SCM logs, source code	Coarse	[Gall 99; Wett 08]
Refinement of SCM change history	SCM logs, source code	Coarse/ medium	[Schn 04; Flur 07b]
Change-based evolution analysis for reverse and forward engineering	Change-based repository	Fine	[Robb 08a; Robb 10a; Robb 10b]
Analysis of interaction coupling	Interaction data	Fine	[Zou 06; Zou 07]
Navigation support based on past interactions	Interaction data	Fine	[Sing 05; DeLi 05]
Creation of task context based on a programmer's navigation and interaction	Interaction data	Fine	[Parn 06; Kers 05; Kers 06]
Measuring developer's familiarity with source code elements	Interaction data	Fine	[Frit 10b]
Analysis of the influence of task on a programmer's behavior	Interaction data	Fine	[Ying 11]

management), and number of files per commit to group commits into sizes. The complete classification can be found in our study on the nature of commits [Hatt 08]. We applied this classification to nine open source systems, discovering that corrective actions generate mainly small commits, whereas development activity and management are spread over all sizes of commits, especially larger ones.

Maalej and Happel performed further analysis on commit comments (and other logs and personal notes) to create an ontology of terms and a set of heuristics to automatically generate a description of the work done [Maal 10].

Analysis of committed files. Software artifacts that have been frequently changed together (in this case, committed) during the evolution of a system have an implicit dependency upon each other, called change (or logical) coupling [Gall 98]. This co-change information may either be present in the versioning system, when the files committed together are versioned receiving a common identification number, or must be logically inferred.

The concept of co-change in versioning systems was first introduced by Ball *et al.* [Ball 97], who used it to detect clusters of classes that often changed together. The authors discovered that classes belonging to the same cluster held a semantic relationship. Gall *et al.* used it to detect implicit relationships between modules [Gall 98], and later between files [Gall 03]. These works have shown that co-changes reveal implicit module and subsystem dependencies, which ultimately aid in understanding the architecture of a system. They can also reveal architectural weaknesses, such as poorly designed interfaces. Ying *et al.* applied a data mining algorithm on co-changes to recommend potentially relevant source code files to a developer performing a change task [Ying 04].

Change couplings have also been visualized, most often as graphs. In these visualizations, nodes represent files or modules, and edges represent the change coupling between two nodes. While some visualize static graphs [Ball 97; Ratz 05], Beyer and Hassan [Beye 06], and D'Ambros *et al.* [DAmb 09] visualize the evolution of the co-change dependencies of a system through dynamic graphs.

2.4.2 Snapshot-based Evolution Analysis

These approaches analyze snapshots of the system at varying intervals, either coinciding with each new version committed to the repository or occurring at fixed intervals (*e.g.* every 3 months). As some of the approaches do not analyze every committed version, they are even more coarse-grained than others in terms of number of versions analyzed. The listed approaches also often have the characteristic of performing source code analysis that involves parsing and reconstruction of previous versions of the system.

Analysis of complete history. The analysis of a system's full history usually involves the use of a meta-model describing the system's entities (for object-oriented systems, entities are packages, classes, methods, *etc.*) and the history of these entities stored in versioning systems. Other kinds of information complementing the evolution analysis can also be modeled: for example, bug reports.

The *Release History Database* (RHDB) models the CVS history of systems, along with problem reports and program features [Fisc 03]. In addition to RHDB, Fischer and Gall proposed a multi-dimensional scaling method to visualize the evolution of features, with the aim of uncovering

hidden dependencies between software features [Fisc 04]. Source code information was later added on top of RHDB and other types of analysis were performed. These included the visualization of source code metrics across different releases together with change coupling information [Pinz 05], and a visualization that combines release history data with source code changes to assess structural stability and recurring modifications [Fisc 06].

Hismo, defined by Gîrba, is a meta-model of software evolution in which a time layer is added on top of every software entity, such as a class or method [Girb 05a]. Hismo combines information extracted from CVS logs with information parsed for each version of each software entity, allowing for different software evolution analyses: detecting candidate artifacts for reverse engineering activities [Girb 04], visualizing code ownership over time [Girb 05b], and enriching Lanza's polymetric views [Lanz 03] with time information [Girb 05c].

Mevo, defined by D'Ambros, is a meta-model that combines versioning system data, source code, and software defects [DAmb 10a]. It is capable of performing several software evolution analyses, including the ones that use versioning systems and source code data. The analyses performed studied the relationship between the evolution of software artifacts and the way they are affected by problems [DAmb 06b], as well as the prediction of defects with the use of source code metrics extracted over the lifetime of the system [DAmb 10b].

Other approaches combine SCM history with information collected from parsing the source code for specific purposes. Holt and Pak's GASE visualizes the evolution a system's architecture between two versions with the aim of helping developers understand and control architectural change [Holt 96]. Xing and Stroulia analyze the design evolution of object-oriented software systems, focusing on detecting evolutionary phases of classes. These phases include: rapidly developing, intense evolution, slowly developing, steady-state, or restructuring [Xing 05]. Zimmermann *et al.* took this a step further with change couplings, parsing the system's versions to build a recommender that guides developers to change-related source code entities [Zimm 05].

Sampling snapshot analysis. These are techniques for which a smaller selection of snapshots is sufficient for analysis. These approaches thus choose not to parse all the versions stored in the SCM, increasing time efficiency.

Rațiu *et al.* defined an approach that enriches the detection of design problems by combining the analysis of a single version with the information related to the evolution of the suspected flawed classes over time [Rapu 04]. Based on Hismo, Lungu and Lanza built an approach to analyze the evolution of relationships between packages. They extracted a set of patterns from the evolution of inter-module relations and used them to recover information about the architecture of software systems. [Lung 07].

Gall *et al.* proposed an approach to visualize a software's release history using color and tridimensional visualization [Gall 99], demonstrating the potential for visualizations to describe the evolution of a software system. Wettel and Lanza proposed a tridimensional visualization based on the metaphor of a city to analyze the evolution of systems primarily at the system level, while also considering the evolution of classes and methods [Wett 08].

Accurate evolution reconstruction. A few approaches tackle the problem of the coarse granularity of changes in traditional SCM systems by reconstructing a more detailed change representation of subsequent versions.

Schneider *et al.* devised a tool called ProjectWatcher that tracks source code changes on the IDE to automatically commit them into a shadow repository [Schn 04]. This solution commits

new changes at a fixed interval of three minutes, thereby increasing the granularity of changes committed to the repository. Information about changes and the relationships between Java constructs are used to build awareness visualizations.

Fluri *et al.* built the ChangeDistiller tool to parse source code files and use an AST differencing algorithm to build a more accurate change representation of the system at every commit [Flur 07b]. The tool reports the changes down to control flow level, including instructions such as loops and iterations. ChangeDistiller has been used for a number of software evolution analyses, including the study of the co-evolution of comments and code [Flur 07a], and the analysis of change type patterns [Flur 06].

2.4.3 Fine-grained Evolution Analysis

These approaches are fine-grained in terms of the granularity of changes they analyze, examining changes beyond commit level. Because they surpass commit level, other types of repositories must be explored in place of the traditional versioning systems.

Change-based evolution. To our knowledge, the work of Robbes [Robb 08a] was among the first to perform diverse change-based evolution analyses, tackling both reverse and forward engineering. In the context of reverse engineering it has multiple applications, abstracting fine-grained changes to uncover high-level relationships between entities in a software system [Robb 07c], characterizing development sessions based on change patterns [Robb 07b], and defining change-based metrics to find logically coupled code entities [Robb 08c]. In the context of forward engineering, it uses change history to improve both code completion [Robb 10a] and change prediction [Robb 10b].

This work shows how a rich change history can be used to improve software development and evolution. It is modeled only on software systems with a single developer, however, neglecting the fact that the development of a software system is seldom the work of a single person.

Interaction and navigation-based evolution. These approaches explore interaction and navigation data collected directly from the IDE. They therefore explore an alternative repository with information oriented mainly toward developer actions while programming, rather than software changes.

Singer *et al.* proposed an approach to support navigation during software maintenance by suggesting related files often navigated together in the past, and thus likely to be navigated together in the future [Sing 05]. TeamTracks, by DeLine *et al.*, is a similar approach that uses the navigation history to display a filtered view of the project based on the entity in focus [DeLi 05]. The major difference between these two approaches is that TeamTracks combines interaction data from all team members.

Zou and Godfrey devised a monitoring tool that records selection and edit events on classes, methods and files within the Eclipse IDE [Zou 06]. In a follow-up work, Zou *et al.* proposed an alternative to change coupling, which they called interaction coupling [Zou 07]. This approach uses both interaction and navigation data to compute the coupling between two entities.

Parnin and Görg introduced a set of models and techniques for building context from a history of programmer interactions with source code [Parn 06]. Kersten and Murphy devised Mylyn, a task and application lifecycle management framework, and the degree-of-interest (DOI) model, allowing navigation and interaction information to be combined to build context for a task [Kers 05; Kers 06].

Fritz and Murphy combined interaction data from Mylyn with authorship (determined from edits to source code elements) to propose a degree-of-knowledge (DOK) model meant to indicate a developer's familiarity with source code elements [Frit 10b].

Ying and Robillard explored Mylyn's interaction repository to understand how the type of a task (*e.g.* enhancement, bug fix) influences programmer behavior [Ying 11]. They envision that this information can be used to automatically customize the IDE based on the type of task as well as the programmer's knowledge.

2.4.4 Discussion

We have reviewed a large spectrum of approaches that use source code repositories to perform software evolution analysis. Though this is not an exhaustive review, it illustrates the richness of information that can be extracted from source code repositories to help developers during software development and maintenance. The approaches exploit different levels of change granularity, each level having various advantages and disadvantages. These will serve the next topic of our discussion.

- **Commit-based analysis is lightweight, but error-prone.** The major advantage of using commit logs is that parsing them is a lightweight process, and it gives information on who committed changes to which files, and when. However, the information linked to commits can drive analyses to inaccurate results. For instance, an analysis that links bugs to files by inspecting commit messages might erroneously link files that are not related to a specific bug but were committed together with files related to the bug. Similarly, if a developer has the habit of working on several tasks and committing all changes after a day of work, files that are not logically related will appear as related in the change coupling analysis. Chen *et al.* conducted a study to assess the accuracy of information on commit logs, and found that up to 78% of commit messages omitted files that had actually been changed, concluding that experimenters should carefully check for omissions and inaccuracies in commit logs before using them in their analyses [Chen 04].

- **Parsing all versions of the system requires intensive data processing.** Parsing all versions of the system and applying a differencing algorithm to compute the changes between subsequent versions requires intensive data processing and further storage of the results. When the analysis of a system is post mortem, the algorithm need only be executed once for the entire history. However, if the system under analysis is evolving, the analysis tool needs to be able to update the generated data with results from newly committed versions. One example of a fine-grained differencing algorithm is ChangeDistiller, by Fluri *et al.* [Flur 07b], which specifically computes changes between subsequent versions. Approaches that analyze the complete history of a system (*e.g.* RHDB [Fisc 03], Hismo [Girb 05a], and Mevo [DAmb 10a]) concentrate on extracting software evolution metrics instead of performing differencing.

- **Snapshot-based analysis loses data.** Snapshot-based analyses range from those that take all committed versions of the system into account, to those that consider only major releases. The larger the intervals between snapshots, the more data is lost in the analysis. This problem becomes evident in program comprehension activities, when the analyst is using a snapshot-based approach to try to understand the rationale behind a sequence of changes. This happens because snapshots (even those on the level of individual commits) aggregate

the changes performed within that timeframe, thus losing information about the changes' precise chronological sequence.

- **Interaction data does not contain the semantic meaning of changes.** Interaction data is excellent for quantifying a developer's level of interaction with the artifacts of the software under development or maintenance, whether they be edit or selection interactions. However, data about edit actions lacks sufficient context: the edit event is captured, but the content of the change is not. For this reason, interaction data is useful for contextualizing a user's work by identifying the artifacts related to a task or another artifact, but is of little use for program comprehension.
- **Change-based evolution analysis trades the flexibility of language independency for greater accuracy of change information.** The major advantage of these approaches is that, being language dependent, they have access to more precise information about the software system's change history. This precise information is readily available for use, and does not need to be reverse engineered. This comes at the price of being domain specific, reducing their use in comparison to language independent approaches, as modern systems are seldom built using only one programming language. However, change-based solutions have targeted uses other aside from evolution analysis. We review these solutions in the next section.

2.5 Change-based Approaches

So far, we have reviewed approaches that analyze the evolution of a software system. In this section we focus on change-based approaches, but broaden the review to purposes other than evolution analysis. We review these approaches to identify similarities and differences with the approach we propose later in this thesis.

Changeboxes by Zumkher *et al.* is a meta-model with the goal of representing changes as a first-class entities and making multiple versions of a system coexist at runtime [Zumk 07; Denk 07]. The Changeboxes prototype, devised to model Smalltalk constructs, encapsulates the semantics of a change process as well as its effects, and models the entire change history of a software system. It captures changes at the level of the runtime system and the IDE, and records both low-level changes and complex transformations such as refactorings.

Each Changebox provides a scope for dynamic execution. Several Changeboxes may be concurrently active and thus enabling different views of the same software artifact within a single running system. Changeboxes can be arbitrarily extended, enabling a single one to explore several development trails simultaneously. Multiple Changeboxes may be merged, allowing their changes to be combined depending on a customizable conflict resolution strategy.

Changeboxes features basic SCM capabilities, such as merging, however its authors do not explicitly mention its ability to support multiple developers. Rather, when merging is discussed, the authors refer to branching from a single developer. Neither do the authors explore the history created by Changeboxes.

Cheops first-class change model aimed at runtime evolution of systems [Ebra 07]. Cheops models a set of features that must be transitionally applied to a software system in order to add functionality. This approach enables bottom-up feature-oriented programming and consists of

three phases: capturing change operations as first-class entities; classifying these operations into separate sets, each implementing a single distinct functionality; and recomposing those modules to form software variations providing different functionalities. The authors did not explore the history created by Cheops. In addition, there is no evidence that Cheops can be used by multiple developers.

Spyware is another tool modeling changes as first-class entities [Robb 08a]. Its main goal is to record changes, allowing for software evolution analysis based on a complete list of the changes performed on a software system. Spyware actively monitors the changes performed by a developer on the IDE, in contrast with the approaches that use change information passively collected by SCM systems.

The Change-based Software Evolution (CBSE) meta-model defined by Robbes is domain specific, modeling object-oriented constructs, but language independent. Like the other tools discussed above, it is limited to modeling changes made by a single developer to a software system. We use the CBSE meta-model as a base for our own work and extend it to represent changes made by multiple developers.

2.6 Summary

We began this chapter by laying down the historical context of our research. We examined the history of software development, focusing on the relation between source code evolution and team collaboration. We traced the history of software development to its origin in the 1940s and listed the major points related to our approach: software evolution, SCM systems, development processes and methodologies, open-source development, collaborative and distributed development, mining software repositories, and global software engineering.

Global software engineering is currently seeing significant growth as a discipline, bringing new challenges to software development. One of these challenges is the maintenance of collaboration in a distributed scenario equivalent to what it would be for a collocated team. We tackle this challenge by leveraging a fine-grained change history of a system to promote collaboration.

We have spent this section reviewing the state of the art relating to the solution we will propose. We started by discussing the state of the art of software configuration management system, focusing on those directly related to our approach. We also reviewed approaches to analyzing the evolution of a software system, discussing the benefits and drawbacks of different granularity levels of data analysis. We concluded by reviewing change-based approaches: those that place change in the center of the development process, modeling it explicitly.

In the next part of this dissertation we present our approach for modeling the evolution of a software system, called the Collaborative Change-based Software Evolution (CCBSE) model. We have devised it to accurately model the changes performed by developers in their IDEs and, as a result, record the evolution of a system as fine-grained changes. The CCBSE model and its implementation, both presented in the next part, serve as the foundation of our work – the starting point for us to build applications to aid developer collaboration.

Part II

Supporting Team Collaboration

Chapter 3

Modeling Software Evolution for Team Collaboration

3.1 Introduction

In the previous chapter, we reviewed the existing approaches for system evolution analysis, especially those that were change-based. In this chapter, we present our approach to model software evolution for team collaboration support, focusing on two aspects: team awareness and knowledge sharing.

Our proposed approach, Collaborative Change-based Software Evolution (CCBSE), is an extension of Robbes's CBSE model [Robb 08a] to represent the fine-grained evolution of a software system developed by a single user. Our extension provides support for systems with multiple developers. First, we review the principles of the CBSE approach and define one additional principle related to team collaboration. We then present our CCBSE approach, detailing how it models a software system and its associated changes. The rest of the chapter is dedicated to discussing the CCBSE approach.

Structure of the chapter. In Section 3.2 (p.29) we present the principles of our CCBSE model. In Section 3.3 (p.32) we show how a software system is represented in our model. In Section 3.4 (p.36) we describe how we model the evolution of a system in terms of source code changes. Lastly, in Section 3.5 (p.38) we elicit the potential uses of our CCBSE model to support team collaboration.

3.2 Collaborative Change-based Software Evolution

The goal of the Change-based Software Evolution (CBSE) model is to accurately represent object-oriented programs and their evolution. To achieve that, the CBSE model applies the reification principle to the changes performed on a software system. Reification is the process by which an abstract concept is turned into a concrete object. In the CBSE model, changes are modeled as first-class entities, and their explicit representation allows for the evolution of a software system to be expressed as a sequence of changes.

The three principles described below, originally defined by Robbes [Robb 08a], serve as a foundation to the CCBSE model. We discuss the strengths and weakness of each principle.

Principle 1: Programs instead of text. In order to accurately model and analyze the evolution of a program, we must adopt the most accurate method of data representation available. The state of an object-oriented program is most accurately described as an Abstract Syntax Tree (AST). The CCBSE model, as its predecessor, models the structure of the system as an AST, having the project as root and methods and instance variables as leaves.

Strengths:

- An accurate representation is multifaceted. It can be used for more detailed analyses, which need highly accurate information, as well as for less detailed ones, which only require the metadata of the changes.
- No parsing is needed to analyze the data produced by this model. In contrast, approaches that use data from language-independent SCM systems need to parse all versions of the system for an accurate analysis, requiring intensive data processing. The CCBSE model's accurate representation can be used directly on analyses, with no need for parsing.

Weaknesses:

- AST representation requires heavy memory use. Maintaining a full system's AST occupies more memory than a simpler model encoding only file names and number of lines. The allocated space required for an AST is $O(de)$[1], where d is the number of developers (we keep one AST per developer, as explained in Section 3.3.2 (p.33)), and e is the number of entities (package, class, method, field) in the system. This means that space grows polynomially in relation to the number of developers and number of entities. The scalability of our approach could therefore be an issue for large systems and large teams. We think, however, that with the amount of memory available in today's machines, this difficulty can be overcome.

Principle 2: Changes instead of versions. With an AST representing the state of a software system, we model changes as executable tree operations, the most basic type of changes we capture. The tree operations (insertion, deletion, and change) can be executed to take the system from one state to another. Our model supports composition of tree operations to form higher-level changes, such as refactorings [Fowl 99].

Strengths:

- First-class changes are more accurate than versions. They model every change between two arbitrary versions of the system, preserving their chronological order. According to Parnin, developers prefer to resume interrupted tasks by looking at the chronology of recent changes than by inspecting an aggregated version of it, which is what commits produce [Parn 10].

[1] The space required also depends upon the size of each entity, which is not a constant value. To simplify this analysis, we ignore this variable.

- First-class changes are a superset of versions. Since they can be executed and undone, they are capable of producing a version of the system as an AST. First-class changes are the deltas between versions of an operation-based versioning system.

Weaknesses:

- Storing every change made to every entity of the program is space inefficient. The space required to store every change instead of the aggregated textual deltas depends on the frequency with which developers perform changes, which is usually two orders of magnitude higher than the frequency with which they commit. For instance, the space occupied by our own tool in the SVN repository is 78MB, while the space it occupies in the repository created by the CCBSE model is 3GB3GB: two orders of magnitude higher. With the cost of storage space continually lowering, however, we believe that having the complete change history recorded is affordable.
- The execution of changes to produce versions may not be scalable, especially for a system developed by a large team. This scalability issue may be one of the greatest problems that the CCBSE model faces, though it is possible to optimize the procedure in different ways. We have investigated an approach involving the storage of snapshots at several points in time, allowing for a hybrid method between changes and versions.

Principle 3: Record instead of recover. Changes are recorded directly from the IDE, instead of being reconstructed from SCM archives. Monitoring the IDE gives access to a large amount of information lost in the check-out/check-in model of SCM systems. To retain access to this information, we decided to record changes as they happen in the IDE, rather than attempting to recover them through differencing algorithms.

Strengths:

- IDE integration is necessary. The IDE has become the single tool a developer uses to work on a project, with an increasing number of alternate tools being migrated into the IDE. UML modeling, testing, and versioning are a few examples of activities that have changed from stand-alone tools to tools integrated in the IDE.
- Recording is simpler than reconstructing. With change-based versioning, the difference between two system states is a tree operation, analogous to a delta between two versions with state-based versioning. The fundamental difference between these two methods is that with the latter, in order to determine what has changed between versions in terms of software constructs, a differencing algorithm must be used.

Weaknesses:

- Our approach requires the developer to use an IDE. However, because IDEs bring such diverse facilities to developers nowadays, those who do not use them are growing more and more exceptional.
- Our approach is IDE-specific. Because we rely on IDE monitoring, the portion of the work done directly on the IDE must be re-implemented to work on a different IDE, although it should not be difficult to make the code retargetable.

- Changes performed outside the IDE are lost. Sometimes developers perform quick changes outside the IDE, which will not be recorded by our monitoring tool, and will consequently appear as a gap on the change history of the system.

To introduce the collaborative aspect of our CCBSE model, we define a fourth principle that complements the three principles defined by Robbes.

Principle 4: Collective instead of individual. In the CCBSE model, we model developers as entities that are part of the evolution of the system, though not part of the system itself. The representation of the system in each developer's workspace is modeled as an AST, meaning that at any point in time, the representation of the state of a system is the combination of the ASTs of all developers. Analogously, the evolution of the system is a combination of the sequences of changes performed by all developers.

Strengths:

- Teamwork is the rule rather than the exception; a software system is seldom the product of one person. The CBSE does not account for this fact, instead modeling the evolution of a system as a sequence of changes performed by a single person. We extend the model to reflect a more realistic scenario: a multi-developer project.

- Emphasis on people facilitates helping them. The ultimate goal of evolution analyses is to help developers better understand the evolution of the system. We believe putting not only changes, but also people in the center of the process can improve these approaches and more effectively help developers.

Weakness:

- The change history produced by the CCBSE model is non-linear. If we consider the history of changes of a single developer, the history is linear. In considering the combination of all sequences of changes performed by a team, however, it becomes non-linear; multiple developers may perform conflicting changes to an artifact that cannot be linearly combined. This generates the question of how to handle the combination of these sequences, whose impact on the applications will depend on the type of analysis performed. That is, some data analyses, such as the identification of code experts, are not closely related to the combination of sequences, while others, such as identification of conflicting changes, are tightly dependent on them.

3.3 System Representation

In the following we describe the generic AST defined by the CBSE approach and its specific version defined by the CCBSE approach. We opt for a language-dependent representation of the system, focusing on the particularities of object-oriented systems developed in Java. We thus trade the model's potential to be used flexibly with multiple object-oriented languages in favor of obtaining greater accuracy in its representation of Java systems.

3.3.1 Generic Abstract Syntax Tree

The first principle of the CBSE model is to adopt a domain-specific representation of the system. To do this, the CBSE approach defines a generic AST for object-oriented programs: the simplest possible representation to be easily used for different types of analysis.

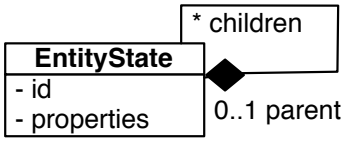


Figure 3.1. A node of the generic AST

Figure 3.1 illustrates an AST node of the CBSE model. The node's attributes are detailed below.

Id. Each AST node has a unique identifier.

Parent. The parent of an entity is another entity. Each node keeps a link to its parent for ease of navigation. The only entity that does not have a parent is the root of the AST.

Children. The collection of an entity's children. Leaves of the tree have no children. The model does not impose any restriction on the number or order of children.

Properties. All other properties of a node are domain-specific, and are thus not specified in the generic model. A node has a dictionary of key-value pairs for these properties, allowing the model to accommodate any type of property.

Figure 3.2 illustrates an example of the generic AST representation of an object-oriented system. It can model software entities down to statement level, thus providing a precise representation of the system at hand.

3.3.2 Specific Abstract Syntax Tree

Though the generic AST could be directly used to model Java systems, this would allow many of the language's peculiarities to be lost. We have therefore adapted and extended the generic model to specifically model Java systems. Though we studied the possibility of directly adopting existing AST models, such as the Eclipse Java Development Tooling (JDT) model [2], their Application Programming Interfaces (API) offer support to many activities that we do not focus on (*e.g.* write, compile, test, and debug programs). Adopting one of the existing models would require extra work to deal with a model that is more complex than what we needed.

The requirements for our model are:

- **Be sufficient.** The model should capture the specific characteristics of a Java program while being capable of modeling systems with multiple developers.
- **Be complementary.** Our approach is not meant to replace SCM systems, but to complement them. In addition, the model is not meant to support implementation activities (*e.g.* compile, test, and debug), but to model the fine-grained evolution of the system.

[2] For more information on the JDT API, refer to `http://help.eclipse.org/`

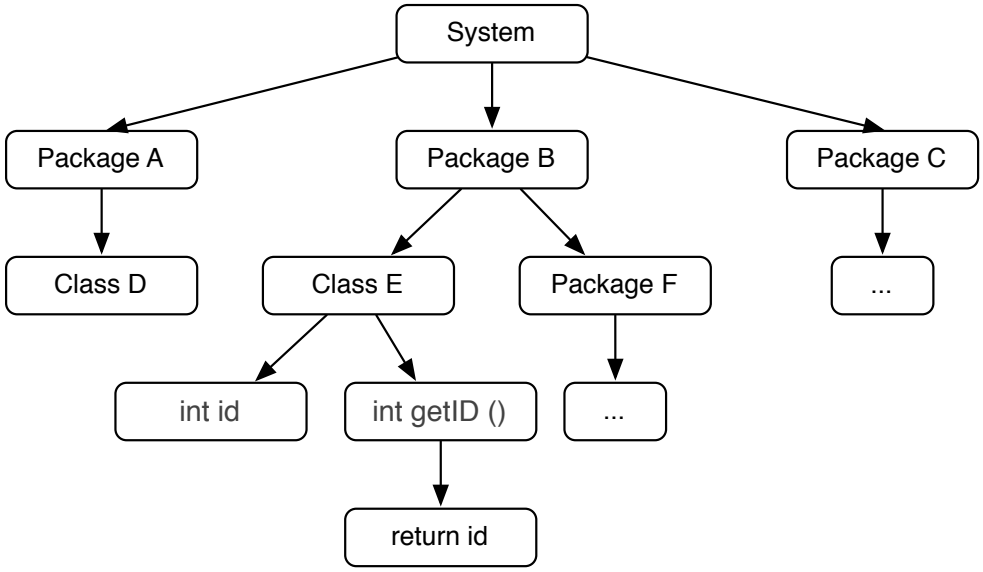


Figure 3.2. An example of the AST representation of a system

Figure 3.3 Shows the UML model of the nodes of the AST we define. We describe each class (or node) of this model below.

EntityState. This is the abstract representation of a node of the specific AST, or the super-class of the hierarchy that defines the node types of the AST. The difference between the generic EntityState and the specific one we define is the presence of the attribute *author*, which shows who was responsible for the change that brought this entity to its current state. Apart from this, we modeled the most commonly used properties of an entity (name, path and revision) as attributes to facilitate the manipulation of the model. The revision attribute allows us to store the current revision number of this entity in the SCM, if the shared system is under some version control system. An EntityState can have children, with the allowed types of children depending on their level in the AST. We explain the EntityState hierarchy from the highest to the lowest level of AST node.

PackageState. This is the only entity that can be the root of an AST. A package can have other packages and compilation units as children.

CompilationUnitState. A compilation unit is not an object-oriented entity, but the representation of a Java file that will always contain at least one class. We decided to model it as a tree node because it contains important information, such as import declarations, that would be otherwise lost if it were ignored. A CompilationUnitState always contains at least one ClassState.

ClassState. Java classes can inherit from only one abstract class, and from multiple interfaces. We model both abstract classes and interfaces as ClassState. This entity's children are instances of MethodState and FieldState: two subtypes of LeafState.

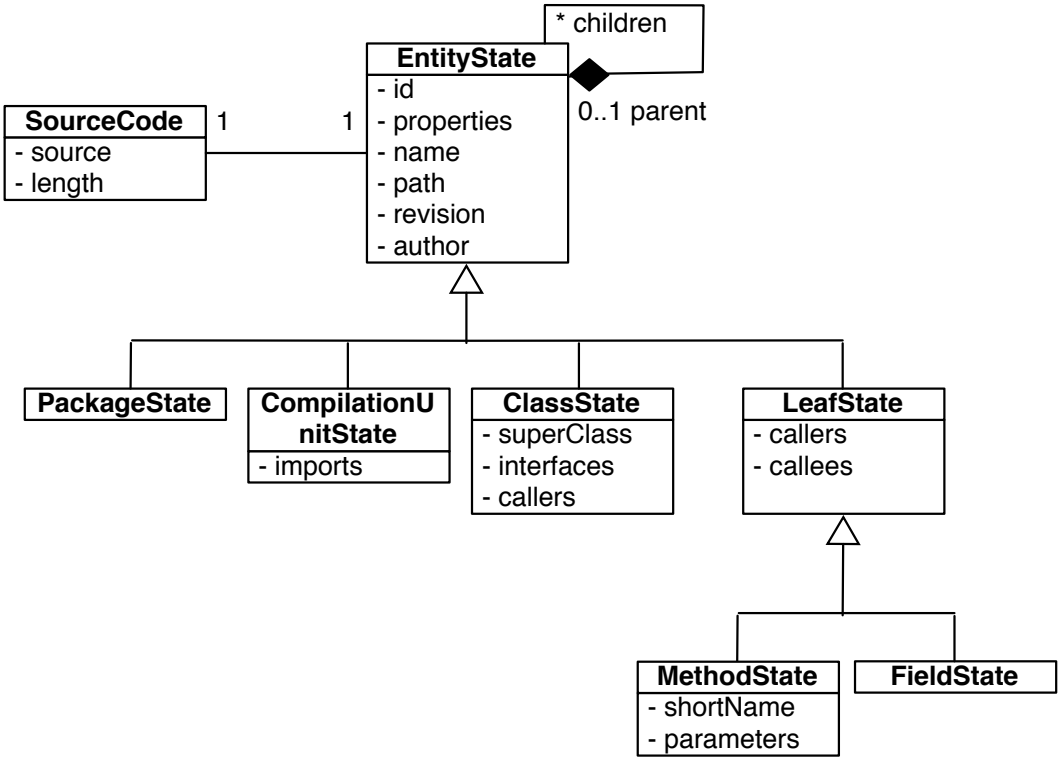


Figure 3.3. UML model of the nodes of the CCBSE's AST

LeafState. We chose to model Java systems down to a method and instance variable level of granularity. This level is sufficient to perform the analyses we target in this work, yet is also manageable in terms of the storage space and memory it requires. Leaves are the only entities that do not have children. They store lists of references to the entities they call and entities that call them.

MethodState. This stores the short name of the method in addition to the full name (the name combined with its parameters). It also holds the list of parameters, separately from the short name.

FieldState. This entity models a class' instance variable (field in Java terminology).

SourceCode. It stores the raw text of the entity's source code. This is necessary, because we stop the model at method level. To be able to reconstruct the system, we must have access to the content of the methods.

We defined the AST to represent the state of a system at a single developer's workspace. However, in a multi-developer project, the current state of a system is different for each developer, and depends on the changes each has performed after a checkout. We studied a number of possibilities for representing this scenario, all of which involved modeling a developer as an explicit entity of the representation. One option was to use an extended AST containing nodes able to store a list of the current states of a system's entity for each developer.

A model more suitable for our primary goal – that of tracking and storing the fine-grained history of a software system – and thus, the one we adopted, was to keep one AST per developer. Figure 3.4 illustrates the state of a two-developer system under this model, highlighting the differences between two developers' ASTs.

With our model, history is represented as linear for a single developer, but non-linear for a whole team, representing the reality of software development: a system can be in different states in individual developers' workspaces. These states are synchronized when a developer updates her code with the repository. Hence, there are points of synchronization, but as a developer modifies her copy of the system, the copies become different until another synchronization is performed.

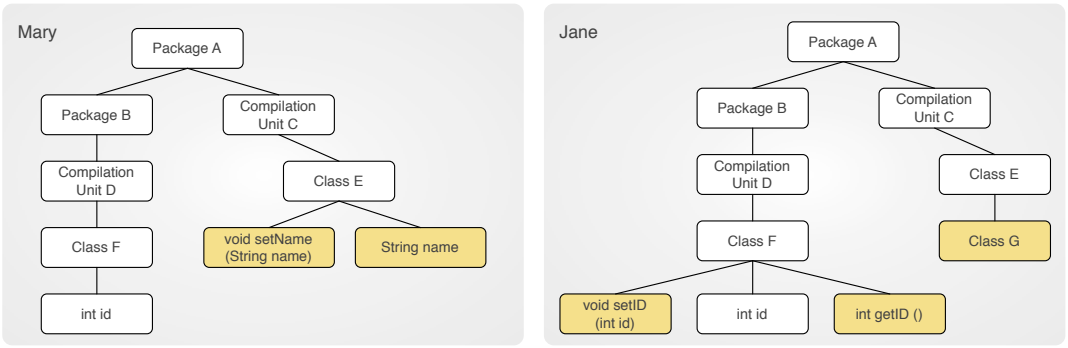


Figure 3.4. An example of the state of a system in two developers' workspaces, highlighting the differences between the two ASTs

3.4 Change Representation

The change representation complies with the second principle of CBSE, which describes the evolution as sequences of changes instead of versions, adopting a change-based approach rather than a state-based approach.

We model a change as a tree operation that takes the representation of a system from one state to the next. Our approach supports two types of change operation: atomic operations and composite operations, shown in Table 3.1.

Atomic operations. These are the finer-grained operations on the system's AST. An atomic operation contains all the necessary information to update the AST model to the next state. The atomic operations we define are: insertion, deletion, and property change.

Composite operations. A composite operation is a sequence of atomic operations applied by a developer for a known purpose (*e.g.* refactoring [Fowl 99]). We model the two most common refactorings: rename, and move. The model can be easily extended to support other composite operations.

In Figure 3.5 we present our meta-model of an evolving system. We first describe the change operations, and then explain how they are related to the other entities of the model.

Table 3.1. Operations on the system's AST

Operation	Type	Description
Atomic	Insertion	Insert a node n as a child of parent p.
Atomic	Deletion	Delete a node n from its parent p.
Atomic	Property Change	Change the value v of property r of node n.
Composite	Rename	Change the name of a node n and change all references of n from the old name to the new one.
Composite	Move	Move a node n and all its descendants from old parent p_{old} to new parent p_{new}.

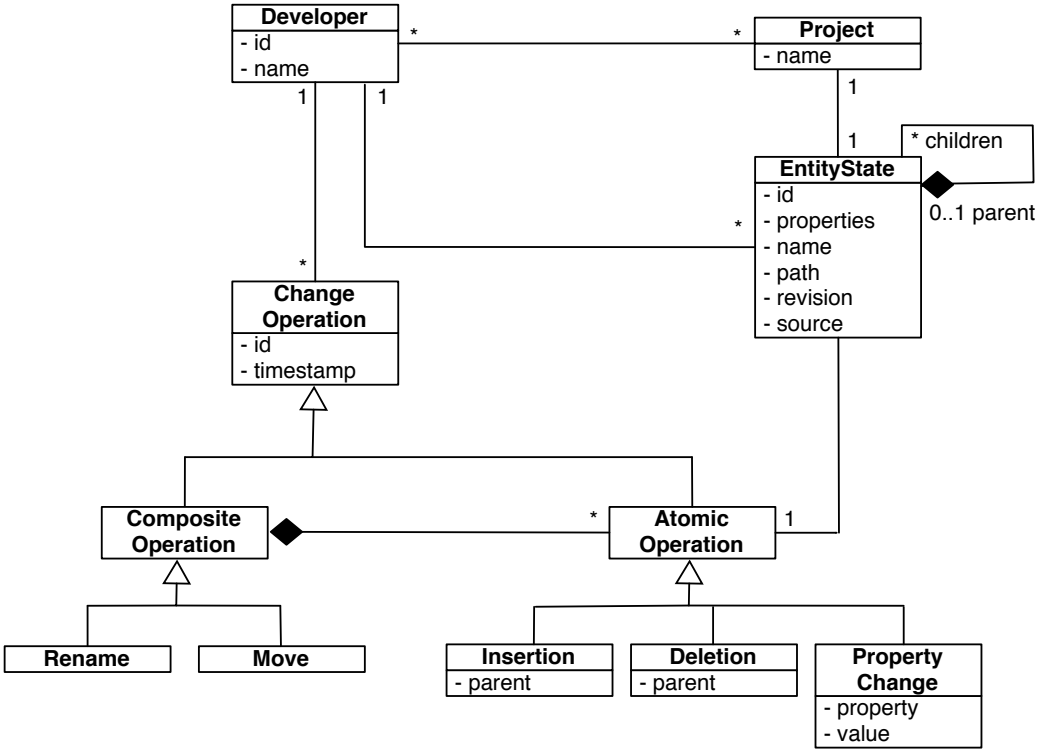


Figure 3.5. The meta-model of an evolving system

ChangeOperation contains a unique identifier and the timestamp of the change. It includes a reference to the developer who performed it, and a collection of atomic operations.

AtomicOperation contains a reference to the resulting EntityState. Insertion and Deletion know the parent of the entity inserted or deleted, and PropertyChange contains the property and its new value.

CompositeOperation is composed of multiple atomic operations. Each subtype of a composite operation records the sequence of its atomic operations.

Project contains its name, a reference to the developers who contributed to the project, and another reference to the root of the AST.

EntityState has a reference to a developer and a project, as described in Section 3.3.2 (p.33). In this diagram, we hide its class hierarchy and the SourceCode class.

Developer contains a unique identifier and a name. A developer can contribute to multiple projects, and can perform multiple changes, generating multiple entity states.

3.5 Uses of the Collaborative Change-based Software Evolution

In this section, we elicit the potential uses for our CCBSE model to support team collaboration. We specifically target applications to support team awareness and knowledge sharing. We evaluate and detail these applications in the next chapters.

3.5.1 Real-Time Awareness of Team Activity

In software development, the term "awareness" is used to describe a team member's level of knowledge of what other members are working on that could potentially impact their own changes to the system. The SCM system plays an important role for the awareness of non-collocated teams, because it is the central point for code synchronization. Developers become aware of new changes introduced by others once they check in their code to the SCM. However, only becoming aware of what others have been working on as late as at check-in time may create problems such as duplicated work, or dealing with complex merging of conflicting code. Fine-grained changes can improve awareness in the following ways:

Awareness of changes. As changes are tracked in a developer's workspace, notifications of these changes can be broadcast to the other team members. This can aid developer awareness of who is changing which parts of the system, who is currently active, what parts of the system are being concurrently modified, *etc.* We discuss our solution and the ways it differs from previous approaches in Chapter 5 (p.59).

Awareness of conflicts. In addition to knowing what is changing in real time, a developer might be interested in knowing when concurrent changes to the same code artifact might create a merge conflict. We discuss our conflict detection approach and compare it with other approaches in Chapter 6 (p.77).

3.5.2 Assisting Knowledge Acquisition

We concentrate on applications that assist developers in acquiring knowledge that can be extracted from the source code change history. This includes solutions that target a distributed development environment, in which information that would usually be acquired through informal interactions becomes difficult to access remotely.

Expertise finding. Developers who have performed a significant amount of change to an artifact can be considered familiar with it. Similarly, those who have changed an artifact recently will retain some knowledge about it. In collocated teams, developers usually have tacit knowledge of who is responsible for or knowledgeable about various artifacts; it is difficult to maintain this knowledge in a distributed team, however. Fine-grained information on source code changes allows us to easily identify those who are knowledgeable about a specific artifact. Our solution for how to measure knowledge based on the CCBSE model is presented in Chapter 7 (p.125).

Understanding past changes. Ideally, a team's structure should remain the same from beginning to end of a software development project. In practice, however, there is turnover, sometimes with as much as the complete team changing over the course of the project [Neu 11]. If a developer is struggling to understand an artifact of the system, and the former expert on this artifact has left the team, an ability to identify this expert from memory will not help the developer understand the artifact. We have instead devised an approach, presented in Chapter 8 (p.151), allowing one to replay past development sessions to understand the changes that lead an artifact to a specific state.

3.6 Summary

Collaborative software development has been practiced for decades, and several methods, processes, and tools have been developed to support the different phases of the software lifecycle. A phase that often becomes hectic, but during which it is critical that collaboration and coordination be maintained, is implementation. We have proposed a Collaborative Change-based Software Evolution approach to aid in developer collaboration. Our approach aims to help developers maintain awareness of the current activity of the team, and obtain essential knowledge of the evolution of the system to better perform their work.

In this chapter we presented our CCBSE approach, which models the fine-grained change history of a software system developed by multiple people. We revisited Robbes's CBSE approach [Robb 08a] and its three principles, then defined an additional principle related to team collaboration. We then proceeded to describe our CCBSE model, presenting the UML model of the AST to represent the state of a system. We also presented the meta-model of an evolving system, including a model of software changes. Finally, we discussed the potential uses of the data that would be generated by applying the CCBSE approach to an evolving system.

The Collaborative Change-based Software Evolution model is the foundation of our work, and the first step toward the evaluation of our thesis. In the next chapter we take another necessary step by describing the implementation of the CCBSE model into a toolset called Syde.

Chapter 4

Syde - A Tool Infrastructure for Team Collaboration

4.1 Introduction

We have devised a tool infrastructure called Syde to support the validation of our approach. Our goal in building such a tool is to implement the CCBSE approach, and thereby make it possible to track and store fine-grained changes for the following two purposes:

- to use the information about ongoing changes to improve awareness of team activity in real time;
- to exploit the change history to help developers share knowledge, and consequently better understand the system and its evolution.

With these two purposes in mind, we use the tool infrastructure to illustrate how such a fine-grained history of a system can help developers improve collaboration, and consequently productivity.

Structure of the chapter. In Section 4.2 (p.41) we elicit the requirements for building the Syde tool to describe its design and implementation in Section 4.3 (p.42). In Section 4.4 (p.44) we present the applications that we built on top of Syde's core structure to support our approach's major goal of improved team collaboration. In Section 4.5 (p.52) we briefly describe a prototype to visualize awareness on a city-based visual representation of the system. We summarize the presentation of the tool in Section 4.6 (p.54).

4.2 Requirements

In the following section we elicit and discuss the requirements for building our tool infrastructure, which we will use to record the fine-grained evolution of a software system.

Complement SCM systems. We do not intend to implement a full-fledged SCM system. Instead, our goal is to complement file-based SCM systems by collecting information that would not be possible to obtain using the mechanisms of these SCM systems.

Be integrated in the IDE. Today's most important development tool is the IDE. Developers use it not only to code, but to model, to test, to access the SCM system, to debug, to analyze the code, *etc.* Today's IDEs are highly extensible, allowing them to better attend developers' needs and reduce the number of applications necessary for a developer to perform their work. Our tool must therefore be integrated into the IDE to facilitate its adoption. It will also be necessary for Syde to track changes performed by developers directly in the IDE. In addition, any extra applications created to help developers must be built as extensions to the IDE, so that they may be used directly inside the IDE.

Be non-intrusive and lightweight. Syde must be able to follow changes directly from the IDE, but in a non-intrusive way to avoid disturbing the developer from his work. In addition, the plug-in extensions should be lightweight to avoid compromising the usability of the IDE's main functionalities and taking the developer's focus away from coding.

Enrich history of changes. Similar to the history logs of CVS or Subversion, Syde should produce a change history with the following information: changed artifact, author, and timestamp. The fundamental difference between these history logs and Syde is that Syde produces a historic entry for every change performed on an object-oriented entity, even if these changes were not checked in the SCM repository later.

4.3 Design and Implementation

Syde is a client-server application in which the server records change operations, maintains the current state of a project, and publishes information about team activity. The client is a collection of plug-ins that enrich the Eclipse IDE to track changes, display awareness information to developers, and help developers share knowledge so they might better understand the evolution of the system.

Figure 4.1 demonstrates the architecture of Syde. In this section, we proceed to describe Syde's components.

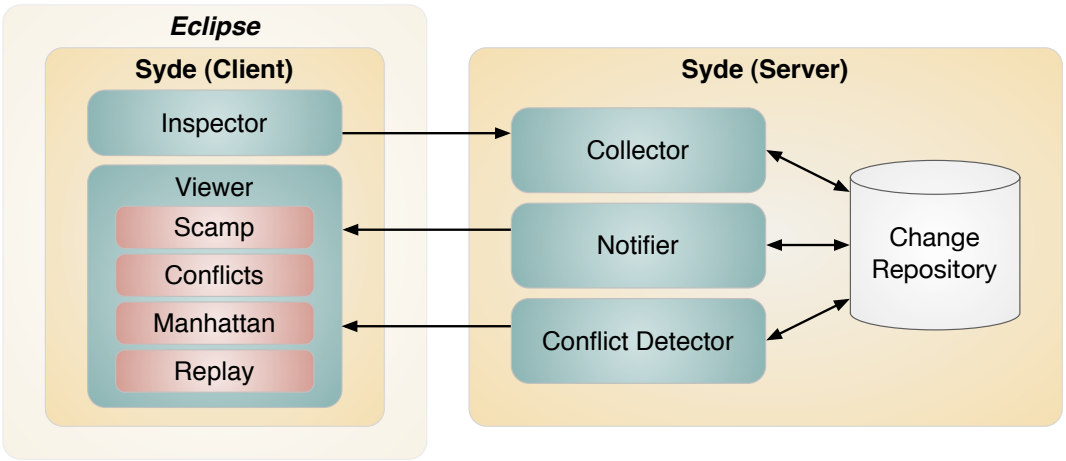


Figure 4.1. Syde's architecture

Inspector. This is the core plug-in, responsible for inspecting the changes made on the IDE, translating them to change operations, and sending them to the server. We explain in detail how changes are gathered in Section 4.3.1 (p.43). Apart from tracking changes, the *Inspector* provides an API for gathering awareness information from the server, facilitating the creation of plug-in extensions.

Collector. This module receives the change operations from the *Inspector*, saves them in the repository, and keeps in memory the state of the system in each developer's workspace. When changes arrive at the *Collector*, they are put in a queue to be processed and are given a second timestamp. The function of this timestamp is to impose a global ordering of the changes as their order of arrival to the server. This avoids potential synchronization problems that might arise (*e.g.* due to different time zones of developers).

This module is also responsible for registering and managing projects and developers, as a developer can work on more than one project.

Notifier. This module is responsible for broadcasting awareness information about ongoing changes. Every time a new change arrives at the *Collector*, it communicates with the *Notifier*, who broadcasts this information to the clients. Events are not filtered on the server side; the *Notifier* broadcasts all change events of a certain project to all clients currently connected to Syde. Event filtering is is implemented by the clients.

Conflict Detector. The *Conflict Detector* searches for potential conflicts that might be arising between ongoing changes. The conflict detector algorithm takes as input the ASTs kept by the *Collector* and the recent operation, and with this information searches for new conflicts and updates existing ones. For detailed information on the conflict detection algorithm, refer to Chapter 6 (p.77).

Viewer. This is a collection of Eclipse plug-ins that use the change information available on the Syde server to visually show the team's current activity, or to assist developers with navigating the change history. It is composed of three plug-ins, presented in more detail in Section 4.4 (p.44):

- **Scamp** shows which users are changing each part of the system, or are the experts of specific artifacts of the system [Lanz 10];
- **Conflicts** displays potential merge conflicts when two developers are changing related parts of the code;
- **Replay** allows developers to replay past development sessions and search for explanations of past changes [Hatt 10d; Hatt 11c].

4.3.1 Strategies for Gathering Changes

Syde uses an operation-based change tracking technique that constantly monitors a developer's workspace to record operations on the fly. To gather changes, we use the following IDE dependent strategies:

Gathering textual edits. When a developer is making textual edits on the body of a class or a method, we capture the textual changes performed between two compilation (or build) actions. In Eclipse, this usually means changes are captured at every save action, given that most developers set Eclipse to build the project at every save. Changes are therefore not captured at every keystroke, nor at each insertion of a new source code entity. The inspector runs a differencing algorithm[1] to detect the changed entities between two subsequent builds. It then transforms these changes into change operations and sends them to the server. This group of change operations has the same timestamp, though the individual changes might have been inserted into the code at different times.

One might argue that the use of a differencing algorithm at this level (to compute the differences between two build actions) is just as subject to error as the application of such an algorithm at commit level. However, a study conducted by Fluri *et al.* [Flur 07b] has shown that the smaller the delta between the versions to which this algorithm is applied, the lower the resulting error. We rely on this finding and on the frequency of compilation actions to capture the change operations and reconstruct the evolution of the system with a high degree of precision.

Gathering refactorings supported by the Eclipse IDE. A number of refactorings are automated or semi-automated by the Eclipse IDE, such as *rename* or *move* of packages or classes. Eclipse sees a rename as a deletion (of text or of a file) followed by an insertion. In our model, a rename is a change to the name property of a node. To correctly identify the rename, we use observers that identify when Eclipse starts and finishes a rename refactoring to transform the subsequent insertion and deletion into a property change operation, rather than a deletion of a node and insertion of a new node. We then inspect the changes that have been made to references of the renamed node and group them with the renamed node to form a rename operation. The solution for the move refactoring is similar to that of rename, given that it is also a semi-automated refactoring action in Eclipse.

Gathering creation and deletion of files and packages. The same observer used to identify changes made inside a Java file also identifies the creation and deletion of files and packages. The *Inspector* transforms these events into change operations, transforming a Java file into a CompilationUnitState object.

4.4 Syde Views

Up to now we have shown the architecture of Syde and explained how the server works to enable the delivery of information to the client side. In this section we present Syde's user interface in the form of different Eclipse plug-ins.

Figure 4.2 shows a screenshot of Eclipse enriched with some of the views available through Syde plug-ins. The *Inspector* (Point 1) acts whenever new changes are added or the project is built. Syde also adds visual cues to the package explorer (Point 2) and outline view, which are both standard components of Eclipse. Most of Syde's plug-ins present view modes at the lower section of Eclipse, however (Point 3). From here, they can display awareness information without distracting the developer from coding, performed in the Java editor (Point 1).

Next, we briefly explain each Syde's plug-ins.

[1] We use the differencing algorithm provided by Eclipse's org.eclipse.compare API.

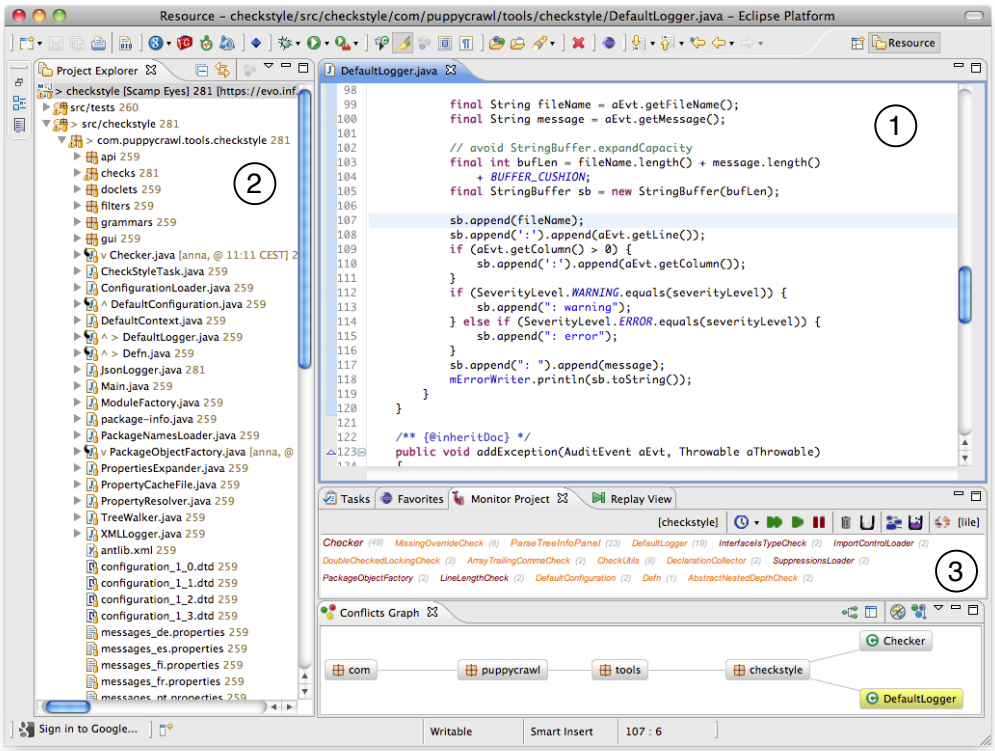


Figure 4.2. Screenshot demonstrating examples of the views Syde can provide (lower half), as well as visual cues on the package explorer

4.4.1 Scamp

Scamp is the plug-in dedicated to showing developers recent and current change activity. It is composed of two views as well as decorations on the package explorer and outline view (see Figure 4.3), as presented below. A detailed description of each view can be found in Section 5.3 (p.61).

WordCloud View (1) notifies developers of which classes have been recently changed by sorting them by date of change from most recent to least recent. It visually marks the person who last performed a change to a class, using different text colors to represent developers. Lastly, it conveys the amount of work that has been done on a class by mapping a larger number of changes made to a larger text size.

Buckets View (2) demonstrates the amount of effort each developer has put into each class by representing a class as a "bucket", and a change as a single square inside the bucket. Each square takes on the color of the developer who performed the change. Once the bucket is filled, this results in a visual display of who might be the experts on a particular class. In Chapter 7 (p.125) we further exploit the notion of code expertise.

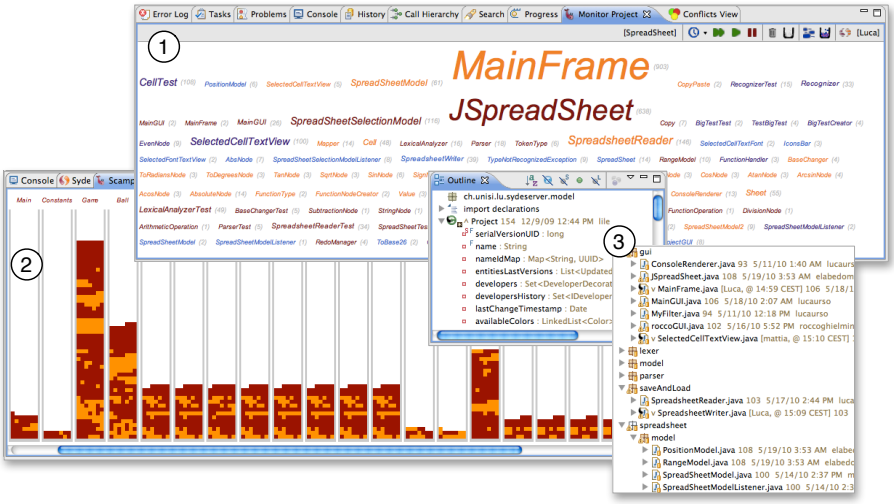


Figure 4.3. The views and annotations presented by Scamp

Decorations (3) are visual cues displayed in the package explorer to indicate that a class is (or has recently been) undergoing a change, as well as who the last person was to change it.

The *Monitor Project* container, in which the views are displayed, provides a place for developers to interact with the views using the toolbar, detailed in Figure 4.4.

Figure 4.4. The toolbar of Scamp's Monitor Project

1. Shows the project name. Scamp allows for focus on a single package; in that case, the package's name will be shown rather than that of the project.
2. These buttons allow the user to control how the information in the view pane arrives. The *time* button lets user set the time window of displayed changes. The *fast-forward* button loads past changes if they were not previously loaded. Clicking on *play* displays changes as they occur. Clicking on *pause* stops the arrival of new changes. Pressing play after having paused causes all past queued changes to flow.
3. The *trash bin* allows one to clear all past changes, and the *empty bucket* clears only those in the buckets visualization.
4. These buttons allow one to switch between the Word Cloud and Buckets views.
5. Shows whether the developer is connected to Syde, also displaying her username.

The views and the visual cues on the package explorer show the recent activity of the team, with the aim of helping a developer become more aware of who is working on which parts of the system. In essence, Scamp provides developers with information on who is currently working on the code, and which artifacts of the code are being changed. The views show more targeted information, such as which classes have undergone the most changes (via bigger words in the WordCloud view), or which developers are experts on a specific class in the system (the most prevalent developer color in the Buckets view). In Chapter 5 (p.59), we discuss how Scamp can be used to improve workspace awareness and to stimulate team communication and coordination.

4.4.2 Conflicts

The Conflict Plug-in displays information about emerging conflicts as developers' copies of the system become inconsistent with one another. Conflict alerts are classified into:

- red – considered severe, because they involve at least one version of an entity that is outdated according to the SCM system;
- yellow – considered moderate, but can easily become severe if the involved developers do not try to resolve them before checking in their code to the SCM.

Conflicts can be displayed in two views: List and Graph. They are also displayed as annotations on the left side of the Java Editor.

List View shows all the potential conflicts that a developer might be involved in (see Figure 4.5). It shows the method or class in which the conflict is located, the name of the other developer involved, and the conflict status (yellow/red). If a developer clicks on one of the conflicts in the list, a Compare View[2] opens showing the differences between the conflicting versions. This view is similar to the ones presented in previous tools [Hegd 08; Sarm 08]; ours can thus be considered an adaptation.

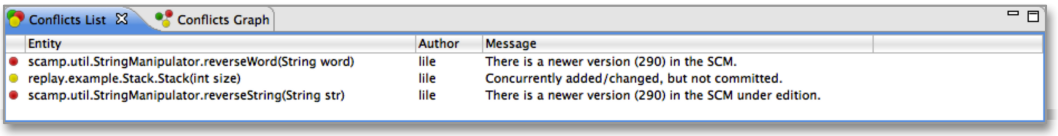


Figure 4.5. The Conflicts List View showing three potential conflicts: two red, and one yellow

Graph View shows a graph representation of the system, with packages and classes as nodes and containment or call dependency as edges (see Figure 4.6).

This view notifies developers about emerging conflicts by adding a color layer on the class node. If a developer desires more information about the conflict, they can hover over the node to identify the method in which the conflict is located, the name of the other involved developer, and a short description of the conflict. A developer can investigate further by double-clicking on the node, which will open a Compare View demonstrating the differences between the conflicting versions.

[2]The Compare View is part of the org.eclipse.compare API.

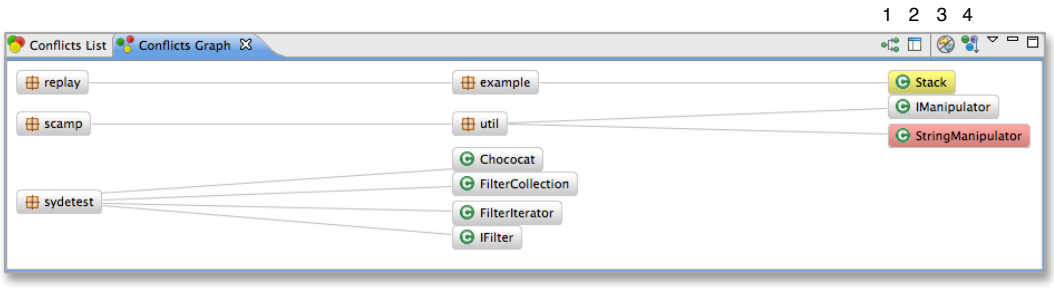


Figure 4.6. The Conflicts Graph View showing potential conflicts in classes Stack and StringManipulator

The Graph view provides several graph customization options that can be accessed through the controls on the toolbar or by right-clicking on a node. Right-clicking a node provides options for collapsing/expanding it – if the node is a package – as well as hiding it. The toolbar allows developers to:

1. change the type of relationship shown by the edges from containment to call dependency;
2. change the graph layout to one of: horizontal or vertical tree, grid or radial;
3. start/stop updating the graph according to the changes being made (*e.g.* if a class is added, a node will also be added to represent it if the option to update the graph is selected);
4. enable/disable emerging design, a function that will cause the graph to only show the classes that a developer opens while he is working.

The concept of emerging design was inspired by previous tools [Silv 06; Proe 10; Serv 10] that specialize in showing developers the design of the software as it emerges.

Annotation on the Java Editor indicates a potential conflict by adding color to the left ruler. This allows a developer to receive hints about potential conflicts even if he does not want to keep the views open, though the hints will only appear for the file on which he is currently working on (see Figure 4.7).

The left ruler of the Java editor showing a conflicting entity receives a yellow or red color. The developer can hover over the annotation to read the conflict's description and the name of the other developer involved.

The aim of the Conflicts plug-in is to complement the information about current and recent change activity provided by Scamp, thereby helping the team maintain a high level of workspace awareness. In Chapter 6 (p.77) we discuss the trade-offs of this approach by presenting a qualitative study evaluating how the behavior of developers changes when information about emerging conflicts is present.

4.4.3 Replay

Replay is the plug-in dedicated to exploring the fine-grained change history of a project. With Replay, developers can search for fine-grained changes made by a set of people to a set of artifacts and watch them in chronological order as they had originally been performed in the IDE.

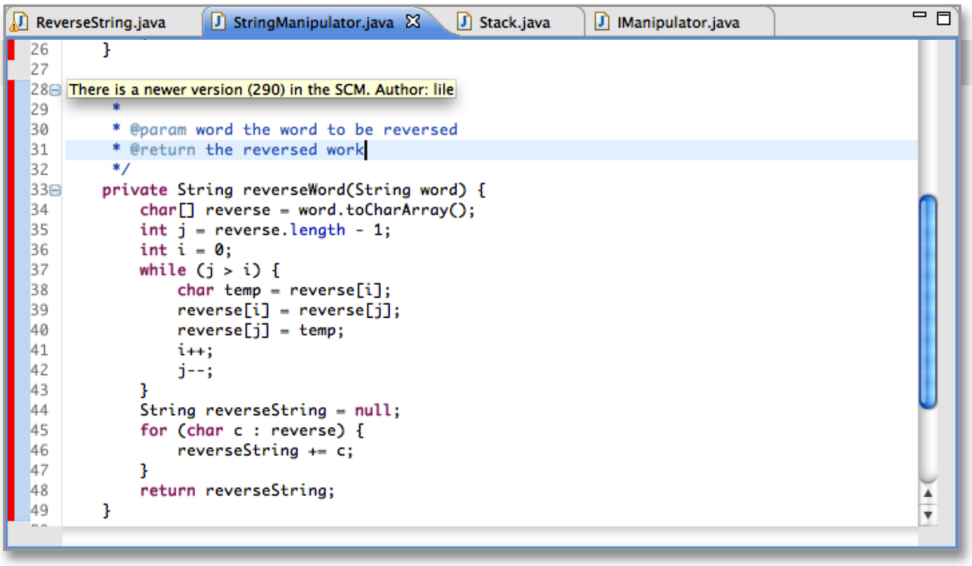


Figure 4.7. Annotation on the Java editor showing a potential conflict on the method reverseWord of class StringManipulator

Since the atomic changes collected by Syde are too fine-grained to be shown individually, Replay groups them by timestamp, author and artifact (package or class), *i.e.*, all the changes that were performed by a developer on a class between two subsequent builds are grouped together based on the last build's timestamp. The granularity of the changes is preserved by the fact that there cannot be more than one change to a single artifact within these groupings.

Figure 4.8 presents the main view of the Replay plug-in. This view has the following components:

The Replay View. The list at the bottom of the window (Point 1) contains all the changes that are currently loaded and are ready to be viewed. These change groups are sorted and displayed chronologically. Each change group is labeled with the name of the artifact from the group appearing uppermost in the AST hierarchy. By navigating through the change groups and following the details of each change in the Replay Editor, one can eventually replay all the activity in the system.

The list provides the following information about each change:

- The artifact that was changed
- The type of change
- The date and time of the change
- The developer who performed the change
- The SVN revision that was the baseline for the change

The Replay Editor. When a change is selected in Replay View, a user can view the effect of that change on the system by looking at the Replay Editor (Point 2), which marks different types

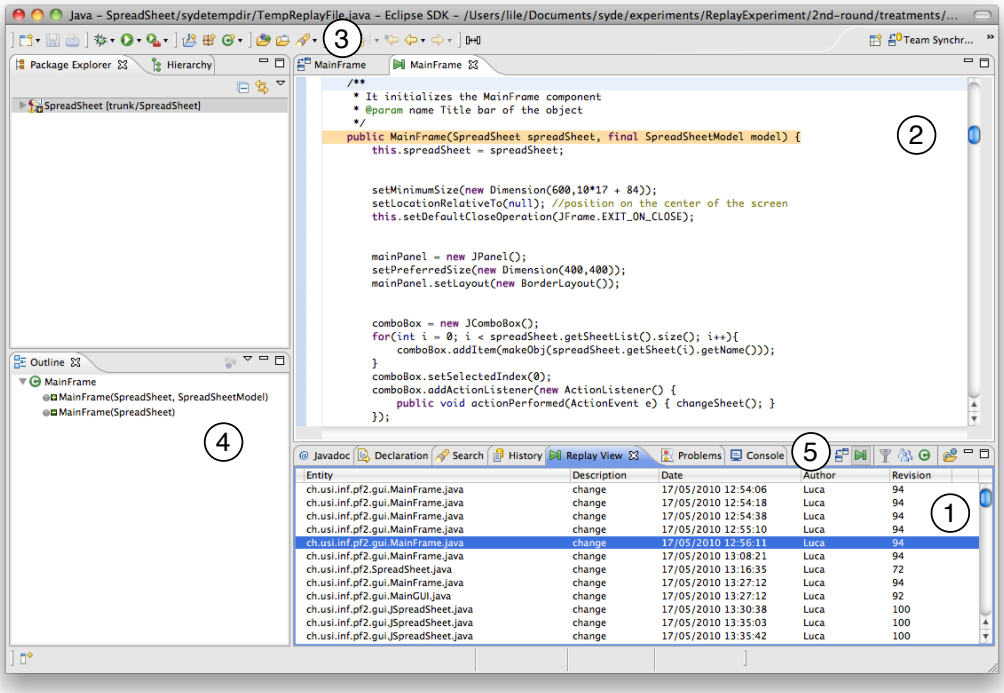


Figure 4.8. The Replay user interface

of changes with different colors. In the example from Figure 4.8, the orange-highlighted text indicates that a change was made on the signature of the constructor in class `MainFrame`. Alternatively, the user can view changes in the Compare Editor[3] (Point 3), which shows a structural and textual comparison between the change selected on the Replay View and the previous change.

The Filters Toolbar. This toolbar allows a subset of the changes to be selected.

There are three orthogonal categories of filters that can be applied to the changes, representing the underlying model of the plugin:

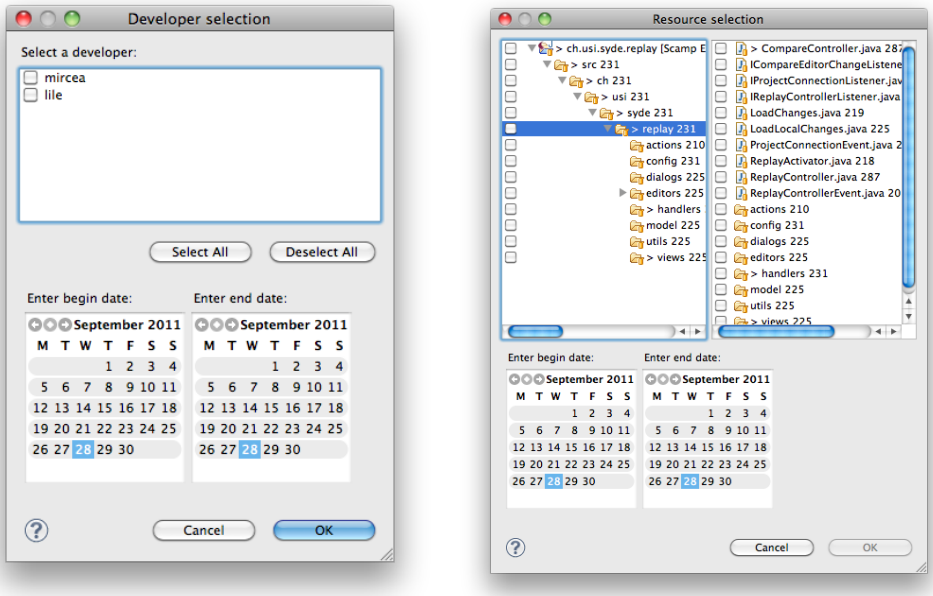
- **Time-based Filters.** Their aim is to filter the changes based on the time period in which they were performed. The time period can be specified as a combination of begin and end time.
- **Artifact-based Filters.** Their aim is to focus the replay on a subset of the artifacts in the system. The artifacts can be methods, classes, compilation units, or packages.
- **Developer-based Filters.** Their aim is to focus the change replay on the activity of a subset of the developers in the system. Such a subset may be a team of developers or an individual developer.

[3]We use the CompareUI already implemented as part of the Eclipse org.eclipse.compare API, leveraging the user's familiarity with the interface.

Combining various filters can support different activities:

- **Remembering previous work.** A developer must remember what she was working on in a previous session before she can move on and continue her work. One way of doing this may be by chronologically replaying the changes she has already performed. This can be accomplished by combining a time-based filter with a developer-based filter.
- **Understanding changes made to a given resource.** When aiming to understand a given resource, it is useful to replay only the changes applied to that resource, using an artifact-based filter combined with a time-based filter.
- **Determining the origin of defects and reason for their existence.** This can be done using a time-based filter by replaying and analyzing all the changes performed on the system since a prior version known to work.
- **Understanding changes made by a given team.** This can be accomplished by using a developer-based filter that filters out all the changes not performed by the team of interest.

Figure 4.9 presents two dialogs supporting the first two activities. In the dialog of Figure 4.9(a) the user can select one or more developers whose changes they would like to see, and select a time interval for further filtering. In the dialog of Figure 4.9(b) the user can select multiple artifacts, including entire packages, and select a time interval to filter these changes further.



(a) Developer selection (b) Resource selection

Figure 4.9. Replay Dialogs

The Replay plug-in differs from the three plug-ins presented before because it does not target real-time awareness, but rather leverages the fine-grained change history collected by Syde to

assist developers in searching for information related to the evolution of the system. This is the only plug-in allowing developers to exploit the change history and watch past development sessions.

This is the first step toward mining Syde's detailed history to help developers better perform their work. The next steps toward accomplishing this goal would be to build recommendation systems able to directly answer common questions that developers ask. One such recommendation system that we have investigated in this work is the indication of source code artifact experts (see Chapter 7 (p.125)).

4.5 Intermezzo: The Manhattan View

In the course of our work we have evaluated the plug-ins previously presented; reports on these evaluations are presented in the next chapters. Manhattan, another Eclipse plug-in integrated with Syde, is a recent work whose primary evaluation indicates its usefulness for developers. However, due to Manhattan's early stage of development, we report only on the plug-in's functions and visualizations and do not present a full evaluation.

Manhattan, devised by Rigotti [Rigo 11; Bacc 11], displays information on recent changes and emerging conflicts with a visual representation of an evolving system. Manhattan models projects in the Eclipse workspace using the 3D city metaphor devised by Wettel *et al.* in CodeCity [Wett 10; Wett 11], which exploits the similarities between software constructs and the urban landscape of a city. This city metaphor gives a visible shape to otherwise intangible software, exploiting the human capacity to build mental models in order to aid in comprehension of a system.

Figure 4.10 shows a project visualized with Manhattan. The city metaphor maps architectural elements to software entities: projects are represented as cities, packages as districts, and classes as buildings. The dimension of each building is mapped to the number of fields (NOF) and the number of methods (NOM) of any class. Interfaces are represented by cylindrical buildings in a special color so that they may be more clearly distinguished.

The 3D city representation of a project requires more screen space than the views of the plug-ins previously presented. We therefore recommend working with a dual-monitor layout when using Manhattan, where one of the two displays shows the interactive city visualization.

A user can interact with the city in different ways. She can navigate the city through the *orbital* mode, where the camera is fixed on a dome centered on top of the city and users can move the view along the dome, or the *first-person* navigation mode, which gives the user full movement control and allows any kind of translation or rotation. By mouse-hovering over any building, a a tooltip appears describing the metrics (*i.e.* NOF, NOM, and LOC) of the corresponding class; by right-clicking, Manhattan opens the class in the Java editor.

Figure 4.11 shows Manhattan's visualization of awareness information. Manhattan identifies:

- emerging conflicts in which the developer is involved;
- classes that have been changed or deleted by other developers;
- developers who have modified a class more frequently.

For illustrative purposes, let us consider a team of three developers working together and analyze their interaction from the perspective of developer 1. When a class is changed by someone other than 1, she sees the corresponding building turn yellow (Point 1). If one of her colleagues

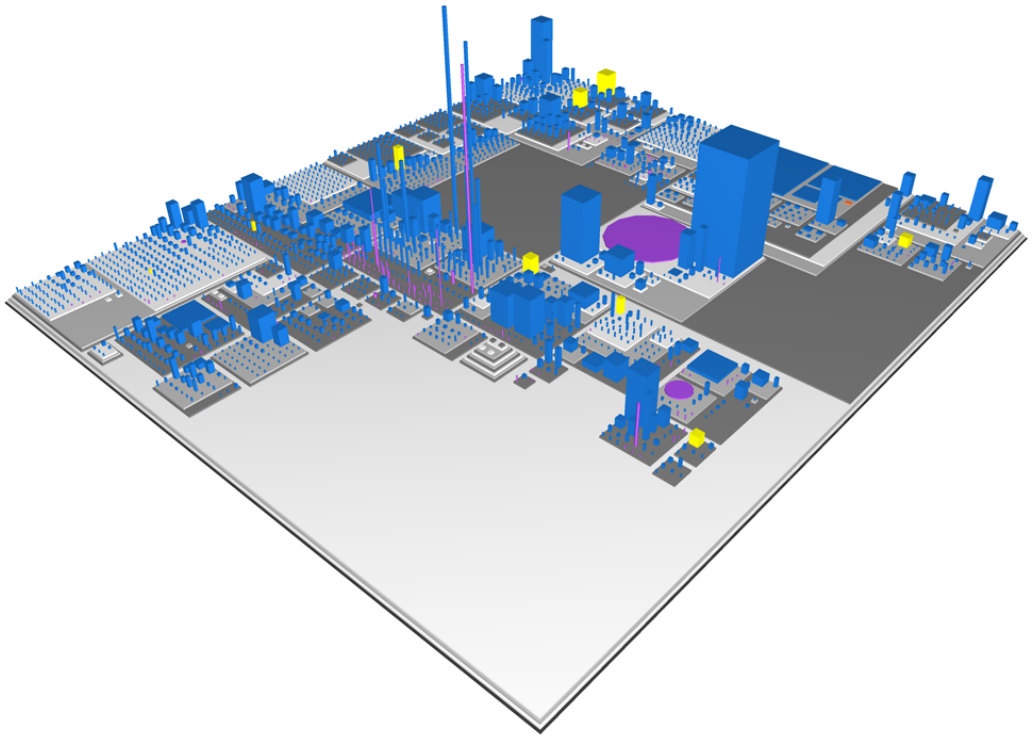


Figure 4.10. Manhattan visualizing a software system – blue buildings are classes, purple cylinders are interfaces, and yellow buildings are classes that have been modified recently

removes a class, the building does not disappear in her view, but becomes orange (Point 2). Supplementary change information, such as the names of developers who have performed a recent modification to a class, is included in the tooltip description for classes (Point 3). Developers are grouped according to the type of change they performed, with any developer who performed a deletion being placed first.

When a conflict emerges within a class, a sphere is put on top of its building (Point 4). The color of the sphere depends on the severity of the conflict: emerging conflicts are shown with a yellow sphere, while committed conflicts (those in which one of the involved developers has committed his changes) are shown with a red sphere. We also use the concept of a *conflict beacon*: a spotlight positioned above a conflict sphere, pointing towards the ground. These beacons illuminate an area of the city around their associated conflict spheres, augmenting the visibility of conflicting classes. After the user notices an alert and hovers the mouse over the conflict sphere, the beacon is deactivated.

Manhattan is an alternative to two of the plug-ins previously presented (Scamp and Conflicts) with a target of augmenting workspace awareness with information about ongoing changes to the system under development. Its city metaphor gives shape to otherwise intangible software artifacts (*e.g.* classes, packages), and the way it represents the evolution phenomenon allows developers to be aware of how – not only how much – the system is growing. A particularity of Manhattan is its need for a dual-monitor setup; the city representation of the system requires significant screen space to achieve a satisfactory degree of comprehensibility.

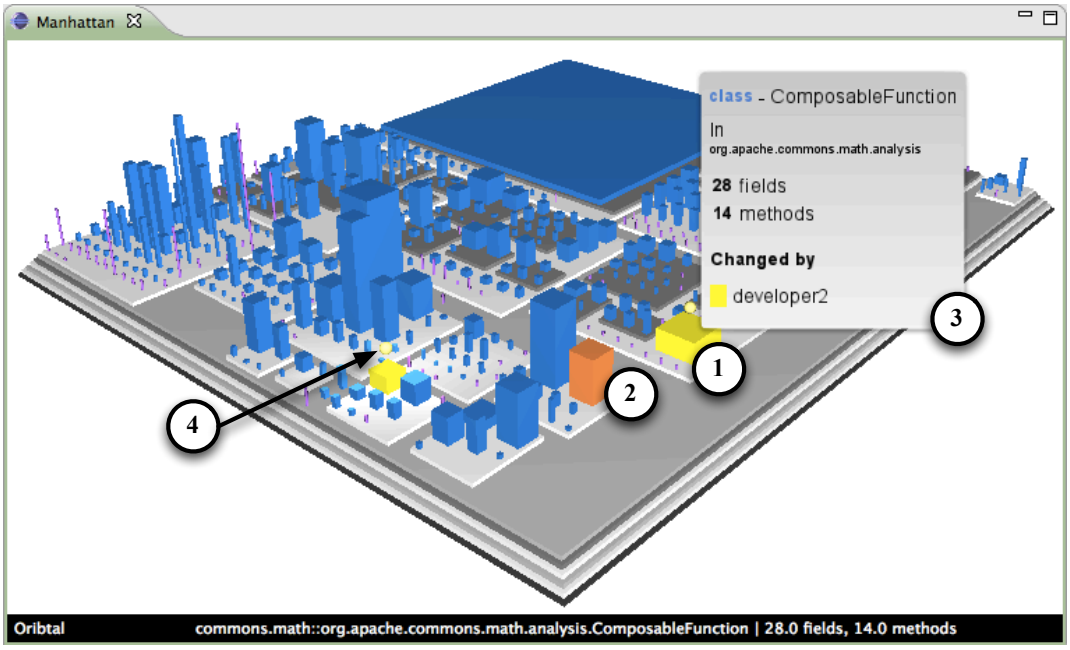


Figure 4.11. Awareness information displayed in Manhattan

An initial evaluation of the use of Manhattan in such a setup is reported in Rigotti's master thesis [Rigo 11]. However, further evaluation is required to assess whether Manhattan's visual metaphor and dual-monitor setup are beneficial enough to developers for it to be worth dedicating a second monitor solely to the display of workspace awareness information.

4.6 Summary

To apply the CCBSE approach in practice, we devised Syde, a tool infrastructure for recording fine-grained changes performed by multiple developers in separate workspaces. The detailed history recorded by Syde makes it possible for the creation of applications that use the available data to help developers directly within the IDE.

In this chapter we presented Syde and its applications to support collaboration. First we elicited the requirements laid down by the CCBSE approach for such a tool: to complement SCM systems, to be integrated in the IDE, to be non-intrusive, and to enrich SCM history. We then presented Syde's architecture, detailing both its server and client components. The server receives, manages, and stores the changes performed by clients, and is additionally responsible for broadcasting change information. The client is divided into two parts: (i) the *Inspector*, which constantly tracks changes made at a developer's workspace; (ii) the *Viewer*, which contains the applications responsible for delivering different types of information to developers.

We also discussed the different strategies used to gather changes on the IDE. There are three strategies that differ according to the type of changes that must be gathered.

1. Textual changes are tracked at every compilation action using Eclipse's observer of build actions. A differencing algorithm detects their corresponding structural changes.

2. Refactorings of type *rename* and *move* are inspected with the help of refactoring observers.

3. Changes at the level of files or packages are detected with the help of Eclipse's observer of build actions, and are translated to change operations.

We detailed Syde's three plug-in applications: Scamp, Conflicts, and Replay. Scamp promotes awareness of changes, and is validated in detail in Chapter 5 (p.59). Conflicts focuses on developer awareness of potential conflicts that emerge due to concurrent changes to software artifacts. It is validated in Chapter 6 (p.77). Replay allows for the history of a project to be explored, and is validated in Chapter 8 (p.151).

Finally, we described Manhattan, a proof-of-concept plug-in that combines the information displayed by Scamp and Conflicts in a city-based visualization of an evolving system. Evaluation of this plug-in is beyond the scope of this thesis.

This part of the dissertation served as a space for us to present the models and tools we devised in support of our thesis: that a detailed change history can be leveraged to help developers collaborate. In the next part we use these tools to investigate our thesis in detail over the course of four chapters. The first two chapters evaluate the use of change history in real time to promote workspace awareness, and the next two chapters evaluate the post mortem use of the change history to assist developers seeking help.

Part III

Applications to Support Team Collaboration

Chapter 5

Supporting Real-time Awareness of Team Activity

5.1 Introduction

Recent years have seen growth in the area of global software development, in which teams of developers do not share the same office but are rather spread over different locations. Collaboration issues, such as *awareness*, *communication* and *coordination*, are often negatively affected in such a setting [Sang 07; Herb 00]. This has a direct impact on parallel development, since an uncoordinated team – one that lacks communication – tends to lose track of who is changing which parts of the system (awareness). Indeed, a drawback of the current mainstream SCM systems that becomes evident when awareness drops is the standard model of change propagation: only once a developer checks in her changes will her colleagues have access to them. As a consequence, there is a general increase in the number of concurrent changes to the same artifact, resulting in merge conflicts and duplicated work.

Recently there has been increased interest in addressing such collaboration issues, which has resulted in full-fledged collaborative software development environments, such as IBM's jazz.net [Fros 07] or Microsoft's CollabVS [Hegd 08]. These are industrial environments especially suited for distributed and collaborative software development. IBM's jazz.net focuses on artifact traceability and on a clear division of activities to prevent synchronization problems, whereas Microsoft's CollabVS creates and exploits a number of communication channels to promote team activity coordination.

We argue that the key to promoting coordination among distributed teams is to increase their level of awareness. We define awareness as an understanding of the activities of others, providing a context for one's own activities [Dour 92]. In a collocated team, awareness is mainly obtained through human interaction such as meetings, informal conversations, overhearing conversations, and helping colleagues[Herb 00; LaTo 06]. In a distributed team, however, the distance between developers or teams constitutes a barrier for these informal interactions, and the team's awareness of member activity is consequently lower than it would be for a collocated team. A low team awareness level can bring about problems such as breakdowns in communication [Dami 07], a lack of team member willingness to help each other, and delays on deliveries [Herb 00].

In this context, we promote awareness by visually enriching the IDE with real-time information about *who* is changing *what* in the system (*i.e.* by instantly propagating change information as the changes happen, rather than waiting for a developer to initiate the synchronization of his code with the SCM system so that he may access changes that other developers have checked in). We argue that working in an IDE enriched with visual cues about what is currently being developed in the system ultimately increases a developer's awareness and helps him coordinate his activities with his team members. However, the creation of such an IDE does not come without its own set of technical and conceptual challenges. For example, one problem that must be addressed is how to display analysis results that do not compete with a developer's main task: writing programs. To evaluate the strength of our thesis against these challenges we present Scamp [Guzz 09], a plug-in built by Guzzi to visually provide awareness information of team activity.

Our work focuses on improving workspace awareness [Gutw 96] through lightweight and non-intrusive techniques, without requiring the use of overly complex environments (*e.g.* Rational Team Concert [IBM 11]) but rather building on Eclipse, an IDE widely used by developers. We implemented Scamp to offer three types of visual options to support developer understanding of changes as they are made to the system, superseding and complementing mainstream software configuration management (SCM) systems like CVS and SVN.

We tested Scamp in a user study with two multi-developer projects spanning the course of several weeks, and report on our findings. Our results show that such "on-the-fly" aids succeed in augmenting awareness among developers, leading to a deeper understanding of the system.

Structure of the chapter. In Section 5.2 (p.60) we discuss previous relevant work that approaches the topic from the point of view of workspace awareness. In Section 5.3 (p.61) we detail our tool infrastructure, composed of the Syde and Scamp Eclipse plug-ins. Scamp offers three custom views to augment awareness: the WordCloud view, the Buckets view, and the Decoration view. We showcase these three views in Section 5.4 (p.65), and also detail a qualitative evaluation performed in the form of interviews. In Section 5.5 (p.73) we discuss the limitations of this approach and future directions that might be taken to improve it. We summarize our findings and conclude in Section 5.6 (p.74).

5.2 Related Work

Recently, significant efforts have been made to craft tools and techniques for the promotion of awareness in a collaborative context [Silv 06; Sarm 08; Schn 04; Hegd 08; Bieh 07].

ProjectWatcher is an Eclipse plug-in that silently auto-commits changes to the source code at pre-defined time intervals [Schn 04]. These changes are mined and information is displayed in the form of two visualizations: activity awareness, which shows past and current activity on project artifacts; and proximity awareness, which illustrates the notion of distance among team members in terms of the system's structure and dependencies.

Lighthouse is an Eclipse plug-in that aims to avoid conflicts by propagating change events from Eclipse and SCM among workspaces, and displaying them on a view of the emerging design representation of the system [Silv 06]. Lighthouse requires a side-by-side presentation of the design representation and the code, which is only feasible if developers work with two screens. Because Lighthouse would require developers to change their programming methods and workspaces, developers will likely resist its adoption.

Palantír [Sarm 08] and CollabVS [Hegd 08] dynamically track code changes to give support for workspace awareness, although they do not focus on the visual display of which artifacts are being edited. Rather, both add support for merge conflict detection and resolution, and CollabVS additionally focuses on communication. Similar to Scamp, Palantír adds decorations to the Eclipse package explorer to mark when a Java file has been recently changed. This visual cue is combined with a textual view listing all emerging conflicts with the goal of warning developers about their existence at a stage earlier than check-in. CollabVS contrastively focuses on creating communication channels to help developers cooperate, have joint working sessions, and resolve conflicts. In neither case is information available to the developer about the number of recent changes per artifact, or the order in which they were applied.

FASTDash aims at augmenting developers' awareness by visually displaying the code base. The information visually presented includes that of which team members have source files checked out, files are being viewed, and classes and methods are currently being modified [Bieh 07]. Analogous to Scamp, FASTDash shows when two developers are working concurrently on a file, allowing them to preempt merge conflicts. However, FASTDash's visualization front-end is built separately from the IDE, and occupies the entire screen, which can generate adoption resistance from developers.

We believe that, in the context of collaborative development, it is important for developers to be immediately aware of who is working on which artifacts. With access to this simple information, developers are able to coordinate their activities and avoid, among other problems, duplicated work and conflicts. The challenge is in making this information available using a representation of the system that is lightweight, non-intrusive, and easily understandable.

5.3 Scamp's Visualizations

Scamp is an Eclipse plug-in that promotes workspace awareness by providing lightweight extensions to Eclipse with the goal of helping developers collaborate. In Figure 5.1 we see the Eclipse IDE enriched with Scamp, which manifests itself in both the Eclipse package explorer and the outline view (A), and additionally uses the bottom space (B) to display three different types of visualizations.

Scamp analyzes one project at a time, focusing on displaying changes made to compilation units. Because Scamp visualizes all changed units, it also displays units that have not yet been versioned, and thus do not yet appear in the user's local working copy of the project. Scamp provides three kinds of visualizations for changes to these compilation units: a *Word Cloud* view, a *Buckets* view, and a *Decoration* on the package explorer.

These visualizations are multivariate and multiscale, and therefore embed all the data they receive from Syde in to the IDE, including entity name, author, and timestamp. Scamp's visualizations show the changes that have been made to an entity during a given period of time. This may correspond to a single day, a week, or a month. This is beneficial in several ways: it allows for different perspectives on how the system is evolving, and additionally flexibility for the different kinds of projects or development phases that a user might wish to view (*e.g.* when the team is in an intensive development phase, a timespan of one day is preferred, while if the project is under maintenance, a weekly or monthly change history might be better suited).

Scamp's focus is to help developers increase their awareness of others' work. We assign each developer a different color in the visualizations, which remains consistent between the different views. The color attribute of visualization has a fundamental role as it can be applied to almost

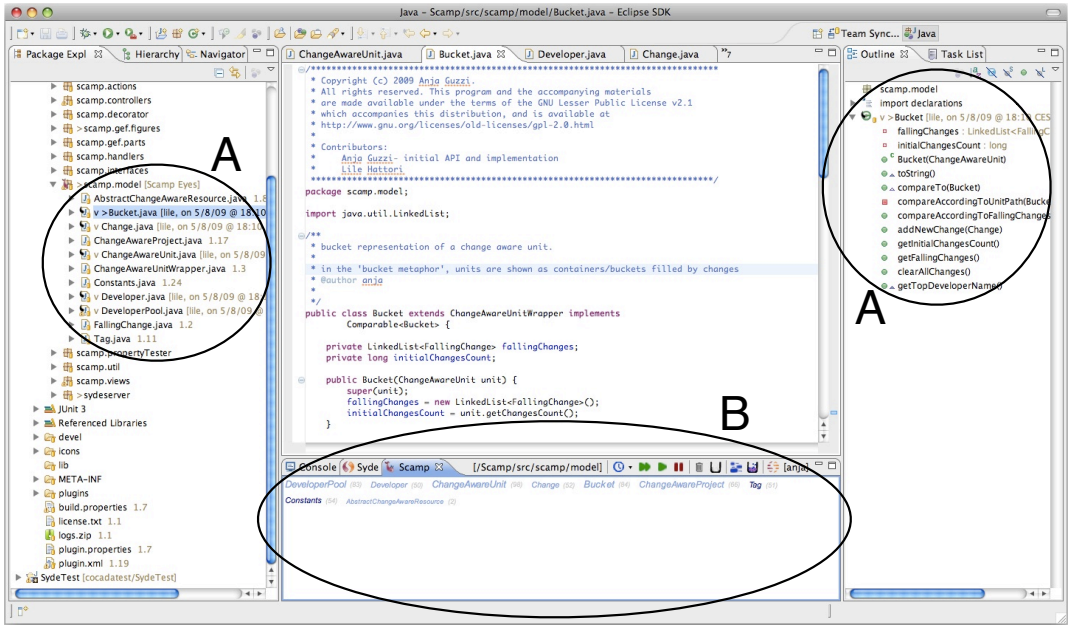


Figure 5.1. Screenshot of the Eclipse IDE featuring the Scamp plug-in

any visual element. Color can be easily assigned to text, figures, lines, *etc.*, while other attributes may be more difficult to assign (*e.g.* assigning a shape to a textual element). The disadvantage to using colors is that only a limited number of them (around twelve) are distinct enough to use simultaneously. This may be a drawback for projects with a large number of developers.

Another aspect of Scamp's visualization approach that enforces awareness is its automatic update feature, which causes the visualization to move to reflect any new changes. We exploit this preattentive attribute (motion) to make the information stand out in the visualization. The motion is fast and happens only once for a single change (*i.e.* it does not blink nor have a transitional time), but is distinct enough for the iconic memory to process the visual information.

In the remainder of this section, we present the three visualizations that compose Scamp.

5.3.1 WordCloud View

A word cloud is a list of words that have been weighted, colored, and sorted according to specific metrics (not necessarily the same). This visual technique was first introduced by the popular photo-sharing website Flickr[1], where words are used to tag pictures.

In Figure 5.2 we see the word cloud of Scamp's own vocabulary, where the size and the color of the words are mapped according to the frequency of words in the Scamp source code. The number in parentheses near each word is the actual number of occurrences of the word.

In Scamp, we display the names of the classes present in a project. The number of changes that have been performed on each class is used as size metric, while the order of the classes indicates how recently they have been changed, with the most recently changed classes at the

[1] See `http://www.flickr.com`

Figure 5.2. WordCloud of Scamp's vocabulary

top. Each word in the cloud is colored according to the developer who performed the most recent change to the class in question. Clicking on a word will take the user to the source code.

Figure 5.3. WordCloud of Scamp's changes

It is possible for a user to select a timespan for the change events they would like to view. For instance, a developer might be interested in seeing which classes were changed in the past week, or month. In such a case, classes that have undergone heavy work will stand out from the others, helping developers spot where the major changes have occurred. Choosing a smaller time window – a single day, for example – allows developers to focus more precisely on what is happening in the present.

There is therefore a constant reshuffling of the words to maintain their chronological order, and changing of colors to reflect the last developer to change a class. This allows developers to quickly spot when concurrent work is taking place: if a developer is changing a class but its word is painted with someone else's color, she immediately knows that someone else is also changing it.

Figure 5.3 presents an example of this word cloud of changed classes. All class names

have the same color, as Scamp was developed almost entirely by a single person. This view depicts the development process at a certain moment in time: here, the developer was focused on changing the classes `BucketsViewManager`, `Constants`, `ScampLightweightDecorator`, and `SydeServer`. The most labor-intensive classes in terms of number of changes made are `ScampLightweightDecorator` and `ScampController`, as is denoted by their large size.

Although this view excels at depicting the entities of most recent interest, it does not display the extent to which various developers have contributed to entities. The buckets view was devised to display this information.

5.3.2 Buckets View

The Buckets view is so named due to the fact that source entities – classes, in our case – are displayed as "buckets", which are progressively filled with single changes depicted as small squares. The color of each change denotes the developer responsible for it. Changes are placed into the buckets in chronological order, with older changes at the bottom and more recent changes at the top. The corresponding class name for each bucket is colored according to the developer with the most expertise with that particular code. Expertise is here assigned according to the number of changes a developer has made to the code. The definition of the expertise measurement can be found in Section 7.3.2 (p.128). The time window can also be adjusted as the developer wishes, with the fill pattern of each bucket reflecting the effort that each developer spent on it within that window.

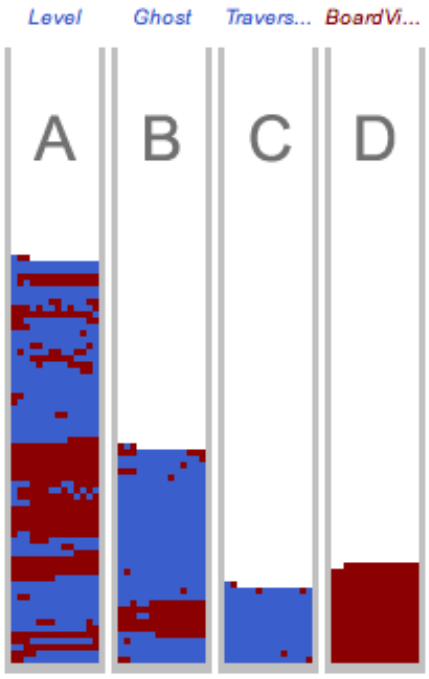


Figure 5.4. Example patterns of buckets

In Figure 5.4 we see four potential patterns that the buckets view can reveal. In case A this

class was repeatedly changed by two developers (denoted by the two colors). However, rather than this work being split evenly over time, the red developer was most active in the middle of the relevant time window. In case B the pattern is similar, but from the height of the bucket it can be noted that the class in B was modified much less than the one in A. In case C the blue developer was most responsible for changes, with only a few being performed by red, while in case D this class was edited exclusively by the red developer.

Both the Wordcloud and the Buckets view occupy the bottom space of the Eclipse user interface. Some developers prefer to maximize the text editor, however. To take this into account, we have also added decorations to the package explorer.

5.3.3 Package Explorer Decoration

Scamp provides decorations in the form of small annotations within the Eclipse package explorer, where the project classes are displayed. If a developer is changing a class and is using Syde, the class' representation in the package explorer is annotated in three different ways to express that "something is going on" with the class. Scamp's decorations are (A) an overlay icon, (B) an arrow, and (C) a textual annotation, respectively marked in Figure 5.5.

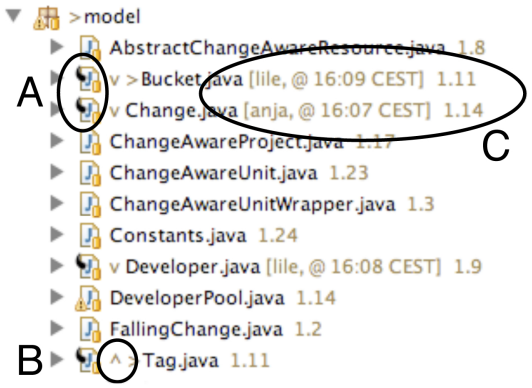


Figure 5.5. Package explorer decorations

The overlay icon (A) denotes the classes that have been changed by some developer using Syde since the beginning of the viewing developer's working session.

An arrow (B) is placed between the class icon and the class name. The arrow points up (∧) if the class has been changed by the developer himself, and points down (∨) if the person to perform the most recent change was another developer.

If the most recent change made to a class was performed by another developer, an annotation (C) is displayed after the class name, showing the username of the developer responsible, and the timestamp of the change.

5.4 Case Study

To validate Scamp, we ran a case study involving two projects developed by students at our department for the "Programming Fundamentals 2" course. Each project was developed by a pair

of students; four students in total were involved in the study. The projects lasted approximately five weeks, and the students, who were recruited for the study on a voluntary basis (*i.e.* they were invited, but not forced to use the tool), used Scamp for the entire duration. At the end of the project, we conducted interviews with each participant to collect feedback on the usage of the tool.

In the next section we present the projects and discuss the answers to the interview questions.

5.4.1 The jArk Project

At the end of development, this project counted at the end 12 packages, 83 classes, and 315 methods, for a total of 7,200 lines of Java code. The number of SVN commits was 300, while the number of changes recorded by Syde was 60,166.

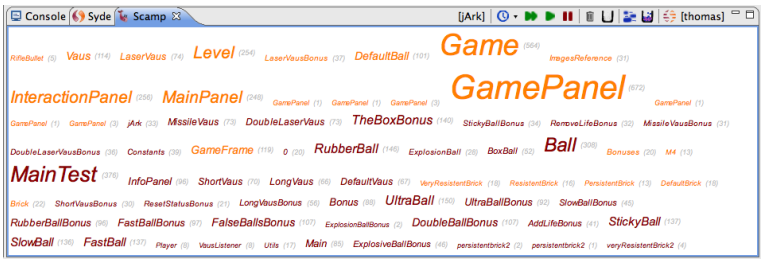


Figure 5.6. WordCloud view of the jArk project

In Figure 5.6 we see a WordCloud view of this project taken during April 2009. This view tells us which classes have changed the most, *i.e.,* `Game` and `GamePanel`, as well as the most recently changed classes, *i.e.,* `RifleBullet`, `Vaus`, and `LaserVaus`. A distinct pattern is visible here: all the most recent changes were performed by the same developer (orange), who is currently the owner of many of those classes; while the red developer worked on many classes, but not as recently.

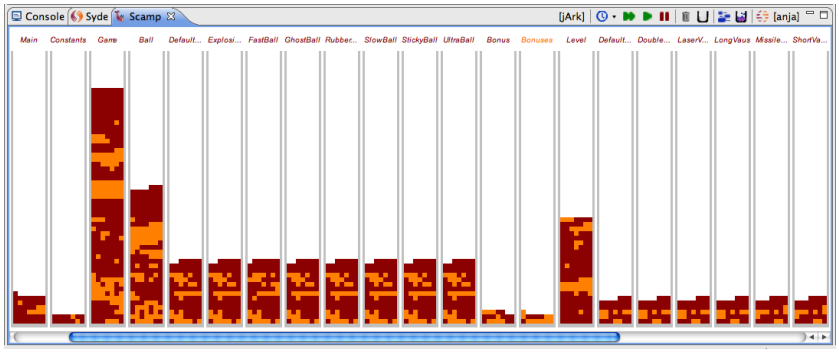


Figure 5.7. Buckets view of the jArk project

Figure 5.7 displays the Buckets view of the project in its last few weeks of development. The class `Game` is still the developers' main focus, but development now is equally divided between

the two developers (red and orange). The classes `Ball` and `Level` have also undergone many changes. We also see that many buckets are identical. These are all sibling classes with a common superclass. This pattern shows a current shortcoming of Syde: if a superclass is changed and compiled, Eclipse automatically recompiles all subclasses, even if they have not individually changed.

5.4.2 The PacMan Project

At the end of development, this project counted 5 packages, 59 classes, and 382 methods, for a total of 3,978 lines of Java code. The number of SVN commits was 170, while the number of changes recorded by Syde was 35,066.

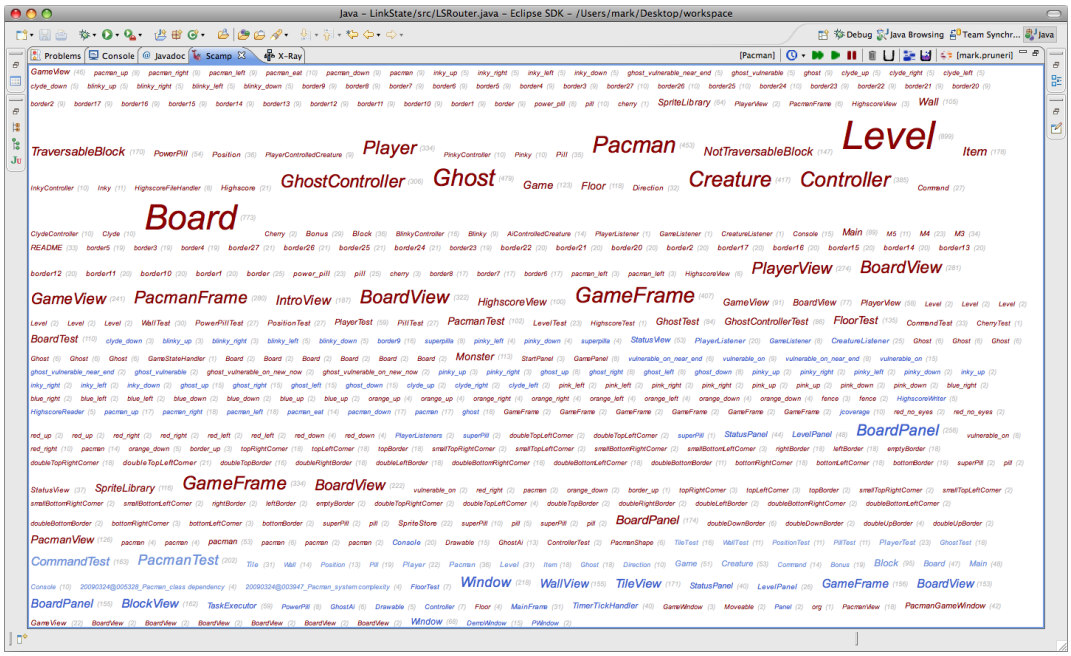


Figure 5.8. WordCloud view of the PacMan project

In Figure 5.8 we see the WordCloud view depicting the entities changed during the last month of the project. The developers confirmed that the most visible classes in this view, such as `Board`, `Level`, `Ghost`, *etc.* were indeed the central domain classes in the system. The view additionally reveals that the project is, at this stage, undergoing many changes on entities that have seldom changed previously. This is noted by the small size (denoting few changes) of the entities at the top of the cloud (denoting recent changes). The developers also mentioned that some classes, such as `GameFrame` had been moved from one package to another. The class names of these entities are visible in more than one place in the cloud, with the top word corresponding to the newest path of this class.

As the project continues to evolve, words representing old paths will sink further to the bottom of this cloud until they disappear from the view. One possible solution for this duplication would be to grey out the name to imply that the old words do not have a link to the class anymore.

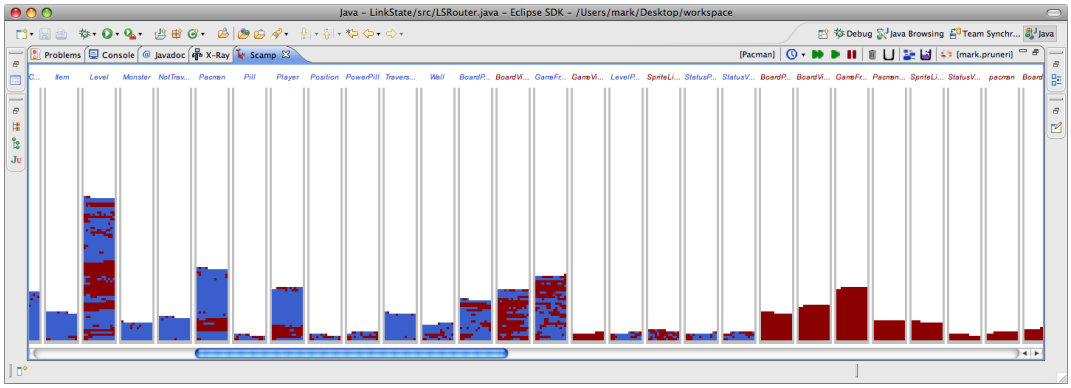


Figure 5.9. Buckets view of the PacMan project

In Figure 5.9 we see a Buckets view of the last month of development on the PacMan project. Some patterns are visible, such as collaboratively edited files (exhibiting a blue-red pattern), heavily edited files (the tallest buckets), or files edited by one developer only (on the right side).

5.4.3 Qualitative Evaluation

The low number of projects and people participating in those projects (2 people per system) does not allow us to perform any statistical evaluation. We instead present a qualitative evaluation performed in the form of interviews, using the questionnaire listed in Table 5.1.

The interviews were conducted individually on a face-to-face basis, and each lasted approximately 30 minutes. Before presenting the questions, we asked the students to describe Scamp in a few words. Their answers indicate that they had a good understanding of Scamp:

"These tools show me which classes we are changing and how much we changed." (Dev 1)

"With Syde, which is a plug-in for Eclipse, you can track who is modifying a piece of software, a file, and you can also view what others modify." (Dev 2)

"An Eclipse plug-in that shows when your colleague modifies some file in the project." (Dev 3)

"You can see the files that you are working on, also which ones your partner is working on, and the last one who edited a file. You can see them as text or as buckets." (Dev 4)

Q1: How often did you use Syde and Scamp? Three developers reported that they used it most of the time, with some exceptions when they occasionally forgot to activate Scamp, especially when they were doing small changes. One developer reported that he experienced an initial phase of acceptance, during which he more or less consciously avoided using the tools. With time, he got used to the new tool and incorporated it on his "development process".

Q2: Were the Scamp views always visible? Most developers answered positively; only in some cases would they minimize the views at the bottom of the IDE to obtain a larger text area.

No.	Question
1	How often did you use Syde and Scamp?
2	Were the Scamp views always visible?
3	Did you usually work physically close to your team mate(s)?
4	Were you aware of what the other(s) were doing by looking at the information shown through the views?
5	Was the constant refreshing of the views disturbing?
6	When you noticed that the other(s) were working on the same class did you: (a) talk to each other, (b) wait and stop editing, (c) ignore and continue editing, (d) rush to commit changes?
7	How often did you commit your code?
8	How often did you have to merge code that was already committed?
9	How were you aware that certain file commits would generate merge conflicts?
10	When you were not physically close, was Scamp more useful than when you were working nearby, and if yes, why?
11	What is the most useful feature of Syde/Scamp and why?
12	What improvements can you suggest?
13	Classify the following statements on a scale from 1 (= strongly disagree) to 7 (= strongly agree) a. It was easy for me to see what my co-worker was doing b. It was not useful to see what my team mate(s) were doing c. I liked seeing my team mate(s) presence even when it did not have any direct benefit such as conflict detection d. I was not comfortable with others seeing information about my activity
14	Classify the following improvements in terms of usefulness on a scale from 1 (= completely useless) to 7 (= very useful) a. Create a filter to allow developers to choose which file types and which folders they are interested in b. Allow a developer to compare the most up-to-date version of a file with his version c. Add an option to automatically login to Syde and load the Scamp views (a.k.a. auto-connect) d. Show the status (online/offline) of each developer e. Provide a mechanism to resolve merge conflicts automatically f. Show who is responsible for each method inside a class that is being edited

Table 5.1. Questionnaire used for the interviews with the developers

We guess this was the case when they were inspecting some large piece of code.

Q3: Did you usually work physically close to your team mate(s)? One pair of developers worked in pair programming the first couple of weeks, but worked mostly from home after that, coordinating their activities through the Scamp views, instant messaging, and through SVN as a last resort. The other pair worked mostly side-by-side, in a setup where each could actually see the other's screen.

Q4: Were you aware of what the other(s) were doing by looking at the information shown through the views? The answers were a clear "yes" for the majority of situations. In some borderline cases where too much happened at once, the developers talked to each other to clarify:

"I just knew which file he was modifying, so I could ask him what he was doing." (Dev 3).

Q5: Was the constant refreshing of the views disturbing? The answers were a clear "no" in three cases. One of the developers reported that he felt a little bit disturbed, but that this feeling disappeared as soon as he concentrated on programming.

Q6: When you noticed that the other(s) were working on the same class did you (a) talk to each other, (b) wait and stop editing, (c) ignore and continue editing, (d) rush to commit changes? The reactions were unanimously that the developers talked to each other. Any other action would have created a delay.

Q7: How often did you commit your code? The answers varied from one a day to several times a day. This is fairly in line with common practice.

Q8: How often did you have to merge code that was already committed? This seems to have happened mainly in the beginning of the project. Later on, the developers gained the ability to more smoothly integrate their individual efforts, as well as use the Scamp views more effectively.

Q9: How were you aware that certain file commits would generate merge conflicts? The developers who worked separately generally used the Scamp views to preempt conflicts, which would then additionally spawn discussions through instant messaging. Even for the pair working together, Scamp proved useful for avoiding emerging conflicts in some situations. However, it was easier for these developers to coordinate their activities by constantly talking to each other.

Q10: When you were not physically close, was Scamp more useful than when you were working nearby and if yes, why? The pair that worked separately answered yes, and explained that the Scamp views were helpful to trigger clarifying discussions:

"It just showed me that Dev 1 was working on a file, and sometimes I asked him what he was doing." (Dev 2)

Q11: What is the most useful feature of Syde/Scamp and why? Two developers said that it was most useful to see which files the other person was editing, so that he might work on something else if he saw that a certain file was being worked on. Another developer said that it was useful to have the possibility to see "live" who was doing what, which minimized redundancy and helped to incite focused discussion about what needed to be done next. One developer really appreciated the WordCloud view, because it allowed him to quickly spot the most important classes in the system (those on which the most effort was expended) within a certain period.

Q12: Which improvements can you suggest? One developer mentioned "auto-connect", i.e., that Syde and Scamp should automatically start instead of needing to be switched on. This is a good sign: the two tools had become a integral part of the working environment, to such a degree that having to repeatedly and explicitly switch them on was seen as a nuisance. Two developers mentioned that it would be good to see even more fine-grained real-time change information, down to the level of single lines of source code. The developer who really appreciated the WordCloud view suggested the addition of a scroll bar, and to limit the growth of words to a maximum threshold. This reinforces the assumption that this view was used heavily throughout the project's development.

Q13: Classify the following statements according to the scale: 1 = strongly disagree, 2 = disagree, 3 = partially agree, 4 = neutral, 5 = partially agree, 6 = agree, 7 = strongly agree. This question was extracted from the questionnaire of a previous study conducted in the context of CollabVS [Hegd 08]. In their survey, the series of questions was classified as "presence (awareness) stream related questions". They performed this survey after conducting a user study with 16 participants grouped into pairs, where each pair had 60 minutes to implement a given task and could freely use CollabVS to accomplish it.

In Table 5.2 we present the answers indicated by the developers in our case study beside the mean value of the answers given by the CollabVS users.

Answers	Dev1	Dev2	Dev3	Dev4	CollabVS
It was easy to see what my team-mate(s) were doing.	7	6	2	3	5.88
It was not useful to see what my team mate(s) were doing.	2	3	2	2	2.22
I liked seeing my team mate(s) presence even when I did not have any direct benefit such as conflict detection.	5	4	6	7	5.56
I was not comfortable with others seeing information about my activity.	2	3	1	2	2.00

Table 5.2. Presence (awareness) related questions extracted from [Hegd 08]

A large gap is apparent between the opinions of the two pairs in answer to the first question. This is due to the fact that developers 3 and 4 worked side-by-side for the duration of the project, a situation in which Syde and Scamp have little added value. Developers 1 and 2, who worked remotely, had the opportunity to benefit from information broadcast by our plug-ins to increase their awareness of the other's activities.

The results obtained from the first pair of developers are similar to the results of the CollabVS study, which emphasized the need for awareness support in situations where developers lose access to face-to-face communication.

Even though the second pair was not sure whether the plug-ins helped them maintain a high level of awareness, as indicated in question 1, their answers to question 2 indicate that they still found the information useful. This agrees with the answers of the first developer pair from our study and with the CollabVS results.

Where privacy is concerned, the answers indicate that the developers were not bothered by the fact that someone else could see what they were doing, again matching the CollabVS results.

One could argue that any instant messaging application like Skype or iChat displays the same degree of private information, and people have gotten used to them.

Q14: Please classify the following improvements in terms of usefulness according to the scale: 1 = completely useless, 2 = useless, 3 = probably useless, 4 = neutral, 5 probably useful, 6 = useful, 7 = very useful. We summarize the answers indicated by the developers in Table 5.3.

Answers	Dev1	Dev2	Dev3	Dev4
Create a filter to allow developers to choose which file types and which folders they are interested in.	6	7	7	7
Allow a developer to compare the most up-to-date version of a file with his version.	6	6	7	7
Add an option to automatically login to Syde and load the Scamp views (a.k.a. auto-connect).	7	7	7	7
Show the status (online/offline) of each developer.	6	6	5	5
Provide a mechanism to resolve merge conflicts automatically.	6	7	3	7
Show who is responsible for each method inside a class that is being edited.	5	6	6	6

Table 5.3. Answers given to question 14

All developers' answers indicated a desire for more functionalities: a strong argument in favor of the usefulness of tools like Syde and Scamp.

Observations. Overall, the developers greatly appreciated the additional information provided by the Scamp views, and were not bothered by their presence. When analyzing the data and conducting the interviews, we observed that the developers' behavior changed after using Scamp for some time. Though the developers initially ignored Scamp's views, with time they began to look at them not only to discover what was going on when they were physically distant, to react to events that the views indicated were happening. These reactions were usually in the form of instant messages sent back and forth requesting explanations. This is much in line with our intentions for Scamp, which are not only to augment awareness, but also to modify the behavior of programmers to increase efficiency and minimize conflicts. This was exactly what occurred for both projects, where using Scamp became second nature.

5.4.4 Reflections

A number of patterns emerged from the views offered by Scamp while the two projects were monitored.

At every moment during development, the WordCloud view highlights the current classes of interest. At first glance, one may think that this view is less interesting for a developer than the Buckets view, as a developer usually knows which parts of the system other users are modifying. This is a misconception, however, and without the help of Scamp, developers can only be aware of changes after they have been committed to the repository. According to the feedback given by

the developers in our case study, the information presented by the WordCloud was fundamental for spotting when a source code element started to be edited concurrently, immediately driving them to coordinate their work by contacting each other through instant messaging.

The developers also drew great benefits from the Buckets view and package explorer decorations. The Buckets view supports the notion of "real time" by immediately updating to reflect each change made by anyone working on the project. This helps to instantaneously preempt conflicts that would emerge at the level of an SCM commit: if a bucket quickly receives several differently colored items it is obvious that a problem is in the making. The drawback of the Buckets view is that its visual symbolism must be internalized for easy understanding, while the other two views are more immediate and intuitive.

The decoration is certainly the least intrusive view method, and the developers appreciated that it smoothly merged into the development environment without occupying any extra screen space. Of all views, this one would likely see the least resistance to adoption, even from non-visual individuals.

5.4.5 Threats to Validity

Although this case study has shown that Scamp's lightweight and non-intrusive visualizations are capable of increasing workspace awareness, the limited size of the projects and number of developers involved prevent us from drawing strong conclusions.

A number of other characteristics about the developers may have influenced the results. First of all, working in pairs, they only need to keep track of one communication path. In addition, one of the groups reported that they worked physically near each other most of the time, sometimes even engaging in pair programming. Scamp's views are not helpful in this situation, and indeed the developers deactivated them while engaging in these practices.

Another noteworthy issue is that the developers in question were fairly new to the Eclipse IDE, *i.e.* they had not yet developed a specific working process with the IDE, and it was not problematic for them to incorporate a new addition like Scamp. It is probable that more expert programmers will present more resistance to a change in their working environment, but this is a reality that any recommender system must face.

Lastly, the qualitative evaluation may have been biased by the fact that one of the authors of these tools also evaluates the participants' coursework. However, it is important to note that the professor responsible for the course "Programming Fundamentals 2" for which these programs were written is not involved in this work, and that the students volunteered knowing that participation in the study would in no way influence their grades in the course.

5.5 Limitations and Future Work

Scamp's visualizations only just scrape the problem of how to augment workspace awareness in a geographically dispersed team, taking the approach of informing developers in real time about changes made to a project's source code. This work has a number of limitations, but opens doors to future work on workspace awareness. In the following section we discuss these topics, suggesting how to overcome some of the limitations, and indicating new research directions.

Scalability issues of current visualizations. Scamp's views have a couple of scalability issues that must be addressed to make it functional for teams larger than a dozen developers. The use of one color per developer limits the amount of users it is possible to distinguish. A

possible solution for this issue would be to assign colors among sub-teams, or to assign distinct colors only within a sub-team while assigning a default color to the rest of the team. The Buckets view shows a limited number of recently changed classes: currently twenty percent of all classes, to a maximum of twenty classes. Once the Bucket view is full, these classes stay visible until the view is closed. While this is a sufficient approach to keep the view legible, other options should be available to tailor the view to the developer's needs. For example, it should be possible to filter the visible classes according to some relevant criteria, or to update the view by replacing the least-recently changed class with a newly modified one.

Subscription system for effective information filtering. Scamp's visualizations show change information for the entire system and all developers. It currently has an option to filter out information by package, allowing a developer to be notified only of changes to a specific package. This filtering option is limited and seldom used, however, as developers are usually interested in changes to multiple packages. For this information filtering feature to be effective, a subscription system should be created allowing developers to choose which users they would like to follow, and what software artifacts are of interest to them. It should also allow information to be filtered with relevance to the team's structure (*e.g.* only listen to changes of a sub-team).

Diversification of the notification system. The workspace awareness of collocated teams is acquired through different sensory channels, *i.e.* sight and sound. Scamp currently exploits sight by visually notifying developers of new changes, but notification sounds could also be incorporated. In addition, different visualizations and notification strategies should be evaluated in terms of their usability and effectiveness at conveying workspace awareness. One example of a different visualization option is Manhattan, presented in Section 4.5 (p.52), though this currently requires further evaluation. A potential notification system that could be integrated is the pop-up based method commonly used by instant messaging applications.

Activity dashboard for managers. Currently, notifications inside the IDE target software developers who are actively coding. However, the same data could be exploited to provide managers with information about the progress of the team. We suggest that a web-based dashboard be created containing charts following the progress of the team, its sub-teams, and individuals, as well as the progress made on specific artifacts (*e.g.* classes, packages). Such a dashboard may also alert managers of potential problems. For example, if a developer is spending much more time than planned implementing one feature, this may be an indication that she needs help. Additionally, if an artifact is concurrently modified by multiple developers for an extended period of time, this might suggest that it is in need of a re-design.

5.6 Summary

Collaborative and distributed software development is a growing phenomenon. It is standard practice in open-source development and has been pushed into practice commercially by the surge of project outsourcing and offshoring, where development teams distributed across the globe develop the same system. Such a phenomenon is believed to have a number of benefits

compared to on-site software development: increased productivity, cost reduction, knowledge sharing, *etc.* However, a number of collaboration issues that are trivial to solve for co-located, become problematic when teams are separated by distance.

In this chapter we specifically tackled one of these issues: lowered team awareness. We proposed a lightweight and non-intrusive approach to augment team awareness, *i.e.* to increase developers' level of knowledge about ongoing change implementation in the project. We aimed to overcome some of the drawbacks caused by the lack of physical workspace awareness, such as the loss of awareness of who is knowledgeable about a class. The case study indicated that developers benefited from the increased level of awareness provided by use of our Syde tool, Scamp. They were able to avoid any duplication of work, and to reduce the number of merge conflicts.

We have therefore successfully leveraged the real-time use of the change data collected by our Syde tool in promotion of workspace awareness. In the context of our thesis, we have demonstrated the usefulness of having immediate access to a detailed change history for augmenting workspace awareness and, consequently, promoting collaboration. In the next chapter, we continue to investigate ways to promote workspace awareness by introducing the concept of preemptive conflict detection, and evaluating whether it can benefit developers.

Chapter 6

Supporting Preemptive Conflict Detection

6.1 Introduction

Recently, there has been a significant effort from the software configuration management (SCM) community to increase the awareness of distributed teams by supporting coordination between multiple developers working in parallel from the same code base [Bieh 07; Guim 10; Hegd 08; Sarm 08]. These recent approaches to promote *workspace awareness* preempt SCM systems by detecting concurrent modifications to software artifacts in real time, especially those modifications that can potentially cause merge conflicts when changes are committed.

A limited number of studies [Bieh 07; Dewa 07; Sarm 08] have been conducted to evaluate whether the adoption of tools to promote workspace awareness are beneficial to developers. Their initial findings suggest that the introduction of preemptive conflict detection increases the frequency of communication, reduces overlapping work, and increases the detection and resolution of conflicts. However, some fundamental questions concerning these approaches remain open. Were the changes noted by these studies beneficial to developers? Were there any changes in the strategies developers used to deal with merging? Is the delivery of this information disruptive to the development process? Would developers prefer to receive the information a different way?

We have conducted a qualitative user study to observe how developers communicate and coordinate with each other to deal with conflict merging, how this behavior changes when they are exposed to preemptive conflict detection, and whether this change is ultimately beneficial. We have also investigated how developers prefer this information to be delivered by exposing them to two different visualizations for emerging conflicts and collecting their opinions. This study investigates the following three research questions:

RQ1: How do developers behave when they have to merge code and resolve conflicts?

RQ2: How does developer behavior change when information is present about emerging conflicts?

RQ3: How do developers perceive different methods of delivering information about emerging conflicts?

We have collected data from observations, interviews, and questionnaires, and analyzed it iteratively by abstracting the findings to obtain a summary of the most important points. Two highlights of our findings are a change that we have observed in the frequency and depth of communication between developers, as well as a change in the strategies used to merge code when developers are exposed to preemptive conflict detection. These changes proved beneficial to developers, who were able to successfully deal with merging, in contrast to their earlier struggles without the emerging conflict information provided by our tools.

Structure of the chapter. In Section 6.2 (p.78) we review related work on tools and conflict-detecting mechanisms in order to discuss what has been evaluated so far. In Section 6.3 (p.81) we present our technique for detecting emerging conflicts, and the different ways it can be visualized on the IDE. In Section 6.4 (p.84) we describe the design of the qualitative user study we conducted. In Section 6.5 (p.90) we present our analysis of the data we obtained, and discuss our findings further in Section 6.6 (p.114). We finally summarize our results and discuss what was learned in this study in Section 6.8 (p.122).

6.2 Related Work

Although efforts have been made to help developers to detect and preempt merge conflicts earlier than check-in time, few of them evaluate developers' behavior or the strategies that they use when merging and using preemptive conflict tools. In this section, we first present the existing related work on tools and mechanisms for detecting conflicts, then discuss the evaluation of these mechanisms that has been performed so far.

6.2.1 Conflict Detection Tools

To develop software in parallel, software developers use coordination tools, such as SCM systems and bug tracking systems, as well as communication tools, such as emails and IRC channels. SCM systems make it possible for developers to code in parallel by allowing them to check out the project to their private workspace, perform changes, and check them into the repository. This coordination model provided by SCM systems is a tradeoff between working in isolation and being aware of the team activity.

Conflict detection tools aim at complementing SCM systems by increasing a developer's awareness of the activity of the team, while maintaining workspace isolation – other developers will only have access to a developer's new code once he checks it in. These tools can be classified into two categories: those that function *after* check-in or *before* check-in.

Conflict Detection After Check-in

Tools belonging to this category detect conflicts after one has checked new code into the repository. This has the advantage of reducing the number of false positives (conflicts that might be detected but do not exist at check-in time), with the price of only detecting conflicts at a later stage.

The seminal work of O'Reilly *et al.* [ORei 03] improves conflict detection in Concurrent Versioning System (CVS) by extending the *watches*: an existing mechanism that permits users to request notification of edit, un-edit and commit actions on files managed by CVS that originate from other users. Their tool, Night Watch, collects notifications from a total of twelve events

(introducing nine more on top of the three native ones) and checks for possible conflicts across workspaces.

Brun *et al.* [Brun 11] propose a similar solution with a stand-alone tool, called Crystal, which detects conflicts in Mercurial. Crystal detects seven distinct states a repository can be in: same, ahead, behind, merge, textualX, buildX, and testX. The latter three states occur when distinct change sets cannot be automatically merged, when a build fails after a merge, and when tests break after a merge, respectively.

Conflict Detection Before Check-in

These tools aim to detect conflicts as early as possible, before new changes are checked into the repository. They are advantageous in that they warn developers as soon as conflicts arise, allowing them to take preventive measures to avoid dealing with conflict resolution at check-in time. However, these tools can produce false positives or short-lived conflicts that might disappear before the check-in. Some of the existing work on preemptive conflict detection was presented earlier in Section 5.2 (p.60), but we reiterate it here to give special emphasis to the techniques that have been developed for detecting emerging conflicts.

Palantìr [Sarm 07; Sarm 08] is an Eclipse plug-in that monitors source code changes at a developer's workspace and checks for emerging conflicts. Palantìr is able to detect direct conflicts (*i.e.* concurrent changes to a single file) as well as indirect ones (*i.e.* changes to a file that might impact a dependent file). It enriches the Eclipse workspace by adding visual cues to the package explorer to show files that might be conflicting, and providing an additional view to detail each potential conflict with information including the authors involved, location, and severity of the conflict. The difference between Palatìr and Night Watch is that the first can preempt conflicts before changes are checked in, while the latter can only detect them afterwards.

FASTDash [Bieh 07] and CollabVS [Hegd 08] offer different types of awareness information, one of which is conflict detection. FASTDash shows who has source files checked out, which files are being viewed, and what methods and classes are currently being changed. When two programmers are editing the same source file, FASTDash immediately notifies them of a potential conflicting situation. CollabVS contributes to awareness by providing different communication channels, including instant messaging, collaborative code editing, and audio/video sharing. It notifies developers of concurrent dependent changes so that they may use their knowledge to prevent conflicts. The main difference between these two tools is that FASTDash is a stand-alone application, modeled to be displayed in a separate monitor, while CollabVS is an extension for VisualStudio, thus allowing all work to remain inside the IDE.

Guimarães and Rito-Silva [Guim 10] propose a solution that goes beyond awareness of potential conflicts. When the project is checked out, *real-time integration* creates a merge workspace that is shared among developers, which continuously integrates the changes they make in their private workspaces. As conflicts emerge, the merge workspace reports them to the affected developers. Conflicts are identified in two different ways: direct conflicts are detected when parallel revisions of the same element cannot be merged; and other conflicts are detected every time the merged system is rebuilt (recompiled and retested).

Lighthouse [Silv 06; Proe 10] and its extension, CASI [Serv 10], are two loosely related works. Their philosophy in approaching merge conflicts is to prevent them from happening, doing this by displaying an emerging design of who is changing which parts of the system down to method level. With this information, developers can proactively avoid concurrent modification to the same source code entities. These tools do not explicitly detect emerging conflicts.

6.2.2 Evaluation of Coordination Strategies when Merging

Though significant effort has been made to proactively detect conflicts and notify developers, few steps have been made toward understanding how developers behave when facing merge and conflict resolution. What are developers' needs in these situations? A few field studies [Grin 96; Souz 03] have been conducted to observe this phenomenon and report on it at a general level. Other questions remain unexplored, however. Does developer behavior change when information about emerging conflicts is available, and if so, how? The solutions proposed to support conflict detection and resolution suffer from a lack of evaluation, and thus little is known about whether the extra information they provide is beneficial to developers.

Grinter conducted the first field study that investigated developers' coordination strategies [Grin 96]. The study is composed of field observations and semi-structured interviews conducted on two development teams from distinct software companies. The study's specific aim was to determine how developers use SCM tools to coordinate their work. Grinter observed that it is sometimes difficult for a developer to merge without communicating with the other person who worked on a module. When this is the case, developers tend to discuss what changes they made and work together to successfully resolve conflicts and merge the code. Grinter also observed a dilemma that developers are constantly faced with: on one hand they want to rush to finish their work first to avoid merging; on the other hand, they want to produce quality code. One might think that experienced developers have no problems handling merge. However, Grinter observed that even experienced developers face problems merging code.

The second study, conducted by de Souza *et al.*, was an eight-week observation of the coordination practices of a software team at NASA [Souz 03]. The team followed a formal software development process that included, among other things, communication via email after code reviews and after files were checked in. Similar to Grinter's findings, de Souza observed that developers tend to speed up to finish their activities earlier in hopes of avoiding merging. Another observation involved files that are frequently changed by many developers. When working with these files, developers tend to perform partial check-ins, a practice in which some files are checked in even when a developer has not finished performing all the changes. It was also observed that some developers tend to hold their check-ins until the end of a workday because it takes so long to compile a system under development. It is also common practice to send an email to the team before performing the check-in, briefly explaining the impact of their changes upon others' work. This demonstrates that in some cases developers think individually trying to avoid merging, while in others they think collectively by holding check-ins and explaining their changes to their team members.

The work of Sarma *et al.* [Sarm 08] focuses on preemptive conflict detection and the variation in developers' abilities to detect and resolve conflicts. They conducted a 90-minute laboratory study with 40 participants with the intent to investigate: (i) whether workspace awareness improves users' abilities to identify a larger number of conflicts; (ii) whether workspace awareness affects the completion time of tasks with conflicts; and (iii) whether workspace awareness promotes coordination. Their results show that participants who used Palantìr detected and resolved a larger number of conflicts than those with no conflict detection tool. When it came to completion time, the participants using Palantìr took less time to resolve direct conflicts in comparison to the control group, but more time to resolve indirect conflicts. In general, participants using Palantìr coordinated more than those who did not use it, with their main coordination actions being SCM operations and chat. This initial evaluation shows promising results, but we believe that it remains important to investigate whether these improvements bring

fundamental changes in developers' behavior, and whether the benefits of added coordination weigh out over the costs.

Another related study was conducted by Biehl *et al.* [Bieh 07] , evaluating the impact of change awareness and conflict detection. The tool under evaluation, FASTDash, not only informs developers of potential conflicting situations, but also which files are being viewed and which methods and classes are being changed. They conducted an observational study of a development team before and after the introduction of the tool. The study involved six participants who were observed for four afternoons. The study's most important findings were of an increase in communication when using the tool, which the authors attribute to raised awareness, and a reduction in overlapping work.

The final related study on this topic was conducted by Dewan and Hedge [Dewa 07] to evaluate the usefulness of CollabVS's mechanism to detect and fix conflicts at editing time. The tool proved useful in increasing the developer's ability to detect and resolve direct conflicts, and improved communication for the resolution of indirect conflicts.

6.2.3 Motivation for the User Study

Our user study complements the existing empirical studies in several respects. First, it studies the behavior of developers when performing SCM operations (*e.g.* check-in, check-out, merge) in detail, analyzing the different strategies used and relating them to the developers' experience. Second, it investigates how developers change their behavior when exposed to preemptive conflict detection, and whether or not this change is beneficial for them. Lastly, it provides a collection of developers' opinions of how to present emerging conflict information without disturbing their main focus: the production of quality code.

6.3 Preemptive Conflict Detection

To study the impact of preemptive conflict detection on developers' behavior when merging code, we have implemented an application to detect conflicts in real time and to notify developers about them, presented in Section 4.4.2 (p.47). The goal of this application is to support developers in detecting emerging conflicts at an earlier stage than during check-in, and consequently, in coordinating their activities to avoid complex merging. In this section, we present the conflict detection algorithm behind this application.

6.3.1 Conflict Detection Algorithm

Our conflict detection algorithm, which resides on the Syde server, detects structural conflicts [Mens 02] related to the change operations that Syde tracks. It can detect both direct [Perr 01; Sarm 07] and indirect conflicts [Sarm 07; Serv 10]. Direct conflicts refer to changes concurrently made to the same program artifacts, while indirect conflicts refer to changes made to an artifact that impact an interdependent artifact. Indirect conflicts can be further classified as syntactic or semantic. An example of a syntactic conflict would be a change to the method signature that another method calls, causing a compilation error. A semantic conflict would involve changing the behavior of a method, potentially introducing runtime errors or unexpected behaviors to its callers. So far, our algorithm addresses direct conflicts and indirect syntactic conflicts.

The conflict detector is triggered every time a new change operation arrives at the server. It receives as input the AST node belonging to the author who performed the operation, and compares this new node with the corresponding nodes from the ASTs of the other developers currently working on the project, one at a time. If it detects a potential conflict, it always refers to the two developers with the conflicting entities. In our model, a conflict never involves more than two developers. Although we could have modeled it accommodating conflicts between multiple developers, we decided to keep it simple to avoid the overhead in complexity of the algorithm, and in potential coordination of developers.

Detected conflicts are classified into two categories:

Yellow. When there is a structural conflict between two entities, but neither has been checked into the SCM system.

Red. When there are structural differences between two entities, and one of them has been checked into the SCM system.

Algorithm 1: The direct conflict detection algorithm

Input: author *(the author of the change)*, node *(the changed node)*, developers *(set of all developers for the project)*
Output: conflicts *(set of all conflicts generated by the change)*

```
1  conflicts := Initialize()
2  i := 0
3  foreach element dev of developers do
4      if (dev ≠ author) then
5          // Search for the corresponding node in the other developer's AST.
6          devNode := FindNodeInAst(node,dev.Ast)
7          // Check for differences in the implementation of the nodes.
8          if node ≠ devNode then
9              if node.Revision ≠ devNode.Revision then
10                 // If revisions are different, a red conflict is created.
11                 newConflict := CreateRedConflict(node,author,devNode,dev)
                   conflicts[i] := newConflict
12             end
13             else
14                 newConflict := CreateYellowConflict(node,author,devNode,dev)
                   conflicts[i] := newConflict
15             end
16             i := i + 1
17         end
18     end
19 end
20 return conflicts
```

Algorithm 1 shows the pseudocode of the direct conflict detection algorithm. This algorithm is triggered by operations of type insertion and property change. It checks whether the modified node contains differences in relation to the corresponding node of the other developers. If such

differences are found, it checks whether they correspond to the same revision currently in the SCM to create a conflict. Finally, the algorithm returns a list of potential conflicts.

Algorithm 2 shows the pseudocode of the indirect conflict detection algorithm. Every AST node of type *MethodState* stores a list of its callers to facilitate the detection of conflicts due to signature change. Therefore, this algorithm is only triggered by operations of types property change and deletion on nodes of type *MethodState*. The algorithm verifies whether the other developers' callers of the changed node are actually calling it. If a caller does not have a call to this node, but was supposed to, then the algorithm creates an indirect conflict.

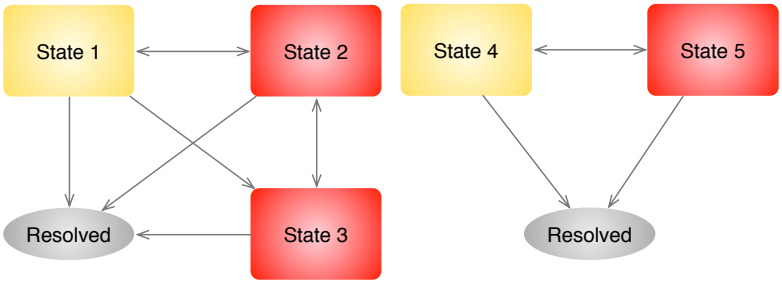
Algorithm 2: The indirect conflict detection algorithm

Input: author *(the author of the change)*, node *(the changed method)*, developers *(set of all developers for the project)*
Output: conflicts *(set of all conflicts generated by the change)*

```
1  conflicts := Initialize()
2  i := 0
3   // Each caller of the method is impacted by a change to its signature.
4  foreach element caller in node.Callers do
5      foreach element dev of developers do
6          if (dev ≠ author) then
7              // Search for the corresponding node in the other developer's AST.
8              devCaller := FindNodeInAst(caller,dev.Ast)
9              /* Check whether the caller has a reference to the changed node by
                  comparing the signature name and the parameters' quantity, names,
                  and types                                                        */
10             if not CallerHasCallee(devCaller,node) then
11                 if devCaller.Revision ≠ caller.Revision then
12                     // If revisions are different, a red conflict is created.
13                     newConflict :=
                       CreateRedIndirectConflict(node,author,devCaller,dev)
14                     conflicts[i] := newConflict
15                 end
16                 else
17                     newConflict :=
                       CreateYellowIndirectConflict(node,author,devCaller,dev)
18                     conflicts[i] := newConflict
19                 end
20             end
21             i := i + 1
22         end
23     end
24 end
25 return conflicts
```

Once created, a conflict follows a workflow (see Figure 6.1) until its resolution. Direct conflicts can be in three states, while indirect conflicts can be in two. Each state is described in the figure

below. To summarize, a conflict can change to another state, stay in the same state or be resolved. Once the detection algorithm returns the list of conflicts, we check the workflow state of each conflict and update the lists.



(a) Direct conflicts (b) Indirect conflicts

State 1	Conflicting changes on two versions of same entity, both are up-to-date with SCM.
State 2	Conflicting changes on two versions of same entity, one is outdated with SCM.
State 3	Conflicting changes on two versions of same entity, both are outdated with SCM.
State 4	A developer changed/deleted a method's signature, another developer has to update its caller.
State 5	A developer changed/deleted and committed a method's signature, another developer has to update its caller.

Figure 6.1. Conflicts workflow

6.4 User Study Design

We aimed to conduct a qualitative study exploring how developers behave, in terms of communication and coordination strategies, when dealing with merging code, and how this behavior changes when they are exposed to preemptive conflict detection. As a secondary goal, we aimed to investigate how developers prefer the information on emerging conflicts to be delivered to them.

We formulate the following research questions:

RQ1: How do developers behave when they have to merge code and resolve conflicts?

RQ2: How does this behavior change when information is present about emerging conflicts?

RQ3: How do developers perceive different methods of delivering information about emerging conflicts?

An ideal setup for observing developers' behavior would be a field study with a team of developers that adopt the preemptive conflict detection tool. This would allow us to identify how their behavior changes while they develop software in practice. However, as it is difficult to convince practitioners to change their programming environment without existing evidence for the usefulness of a tool, our user study has been designed for a laboratory setting, to allow for the simulation and close observation of the scenario of developers working in parallel.

Even in a laboratory setting, conducting such a user study is complex because it requires that different developers be observed at the same time, and additionally requires that conflicts be

guaranteed to occur. We designed the user study as a programming assignment to be performed by a pair of developers, consisting of three tasks. With two developers working on these related tasks at a time, this would simulate parallel development while simultaneously provoking conflicts. To simulate a distributed environment, developers are forbidden to speak to each other or to see each other's screen. All communication between the pair must be done over instant messaging (IM). The participants are given an Eclipse installation with the necessary tools to perform the assignment (Subversive, SVN connectors, Syde's plug-ins) and an IM client for communication.

6.4.1 Data Collection

We follow the guidelines of Creswell [Cres 07] and Barbour [Barb 08] in using mixed data collection methods for a better interpretation of our findings. The data sources we have collected are:

- **questionnaires** regarding the assignment and the participants' experience levels;
- **observation** through video recording;
- **interviews** to gather participants' feedback on their experience with preemptive conflict detection;
- **documentation** in the form of the participants' IM logs, and the list of changes and conflicts generated during the implementation assignment.

In the following we explain how each source of data was collected and the procedures we used to analyze them.

Questionnaires

Participants were asked to answer two questionnaires for different purposes:

1. **Screening questionnaire.** Before the assignment session, participants were asked to answer a screening questionnaire (see Appendix A.1) that collected personal information (name, email address) and information on technical experience (level and number of years of experience with the tools and concepts used in the assignment). Personal information was used for contact purposes, while the information on the participants' experience level was used to characterize the participants and take their knowledge into consideration for the purpose of data analysis.

2. **Debriefing questionnaire.** After the assignment, participants were asked to answer a short questionnaire (see Appendix A.2) to give immediate feedback about the assignment. First they were asked about their experiences in performing the experiment; then they rated some statements related to the usability of the views that showed emerging conflicts. In the second half of the questionnaire, participants were asked to rate statements regarding each task according to their opinions.

Observation

Observation was not performed directly during the assignment; we instead recorded each participant's interaction with the tool as a screencast (a digital recording of computer screen output) and later analyzed them. To properly analyze the screencasts, we developed a codebook (see Table 6.1) that we used to annotate the videos. The codes created can be classified into four categories: communication, interaction with the SCM system, interaction with Syde's views of preemptive conflict detection, and testing.

To code the screencasts, we used a tool called VCode [Hage 08], which provides a timeline, two different types of annotations (range and mark), and keyboard shortcuts that allow a viewer to place annotations, play or pause the video, or add annotation descriptions. After the screencast has been annotated, VCode exports all the annotations to a .csv file, which we used for both qualitative and quantitative analyses.

Interview

At the end of each partnered session, we conducted a semi-structured interview with the two participants at the same time to collect more feedback from them. Table 6.2 shows the questions that guided the interviews.

However, since the goal of the interview was to collect the participants' opinions regarding preemptive conflict detection, the different ways to visualize this information, and the user study itself, each interview had an additional different set of questions that were asked in reaction to the answers participants had given to previous questions.

In most cases, there was a break of about 10 to 30 minutes between the assignments and the interviews. The participants therefore had some time to think about their experience with the tool, the advantages and disadvantages being provided with preemptive conflict detection information, and how this information could be better presented. On occasions when participants could not meet with the experimenter at a later time, the interview was conducted immediately after the assignment.

The interviews were recorded and later on transcribed to be used in the data analysis. We decided to not develop a codebook for the interviews, because they were usually short (around 10 minutes). We therefore mainly used direct quotes from the participants to illustrate their opinions, and support the findings that were emerging from the data analysis.

Documents

A few documents were also collected at each session; however, these were not generally used for the purpose of data analysis except where some other piece of data was missing. The documents we collected were each participant's chat logs, and the list of changes that they performed (collected through Syde). The list of changes indicates who changed what entities of the system and in which order (see Figure 6.2).

The list of changes was more frequently used by the experimenter to monitor the assignment and to take action whenever unexpected problems occurred. The chat logs were only used during data analysis when there was a problem with a participant's screencast, which happened three times, or the chat window was located outside the recorded area, which happened once.

Table 6.1. Codebook used to annotate screencasts of the participants' coding sessions

Category	Code	Description	Type
Communication	Communicating	When the chat window is in focus	range
	Message notification	When notification of a chat message appears, but the chat window is not in focus	mark
Interaction with SCM system through the IDE	Synchronizing	Synchronizing the project or a part of it with the codebase in the repository	range
	Updating	Updating the project or a part of it with the latest version from SVN	range
	Checking in	Checking in the changes made to the project	range
	Merging	Merging the code with the latest version from SVN	range
	Resolving conflict	Resolving conflict during merging	range
	Viewing changes	Viewing changes in the Compare editor	range
Interaction with Syde	Conflict graph	When the Conflict graph is visible	range
	Conflict list	When the Conflict list is visible	range
	Interaction with graph	When the user interacts with the Conflict graph	mark
	Interaction with list	When the user interacts with the Conflict list	mark
	Yellow conflict	When a yellow conflict regarding the task appears	mark
	Red conflict	When a red conflict regarding the task appears	mark
Testing	Testing	Running the JUnit tests	range
Task delimiter	Task starts	User starts a task	mark
	Task ends	User finishes a task	mark

6.4.2 Object System

The system we chose as the object of our experiment is *Checkstyle*[1]. We used version 5.3, which consists of 341 classes distributed across 22 packages, for a total of 46 KLOCs[2]. Our choice was motivated by the following factors: Checkstyle's small size allows for the performance of a user study session, yet is representative of real life programs. It is written in Java, with which many potential participants are sufficiently familiar. It has been used in previous experiments [Corn 10; Guzz 11; Romp 08; Zaid 11], one of which was conducted by the authors of this work.

[1] See `http://checkstyle.sourceforge.net/`

[2] Measured using `http://eclipse-metrics.sourceforge.net/`

Table 6.2. Questions that guided the semi-structured interview at the end of each session

	Questions
1	Did you have to resolve conflicts during the experiment? In which situations?
2	Do you think being aware of emerging conflicts at implementation time helped you to prevent conflicts or to reduce their complexity at check-in?
3	Did the information about emerging conflicts help you to be aware of what your colleague was doing?
4	Were emerging conflicts an incentive for you to talk with your colleague and to coordinate your tasks?
5	Did you get to know the other participant's tasks?
6	Was the Conflict list useful? Was it intuitive? What are the advantages and disadvantages of this view?
7	Was the Conflict graph useful? Was it intuitive? What are the advantages and disadvantages of this view?
8	Which one of the two visualizations do you prefer? Why?
9	Can you think of any other ways that this information could be visualized?
10	Can you think about situations in your everyday coding where these visualizations could be helpful?
11	Do you have any other comments or suggestions about the tool?
12	Do you have any comments or suggestions about the experiment?

6.4.3 Tasks

The assignment was composed of three coding tasks that had to be done by two collaborating participants. Each participant had a different set of tasks, but they were complementary. For a task to be considered finished, each participant had to implement what his task asked him to do, coordinate with his partner, check in his changes, update the code with the changes made by his partner, and make sure all the tests related to the task passed.

In this section we describe each task, showing what each of the participants was required to code (refer to Appendix A.2 to see the complete instructions one participant's set of tasks).

Task 1 – Improve class MethodCountCheck. In this task, participants were asked to make different modifications to the method `checkCounters`, which basically counts the number of (private, public, package, and default) methods in a class. One of the participants was asked to refactor the method `checkCounters` by creating a utility method, allowing for repetition in the `checkCounters` code to be removed. The other participant was asked to implement a few checks missing from `checkCounters`.

Task 2 – Finishing class PlainTextLogger. In this task, participants were asked to finish the implementation of class `PlainTextLogger`. For one participant, this required completing the implementation of method `addError`, and implementing method `fileStarted`. For the second participant, this required completing the implementation of method `addError` with a different functionality than the other participant was tasked with, and implementing method `fileFinished`.

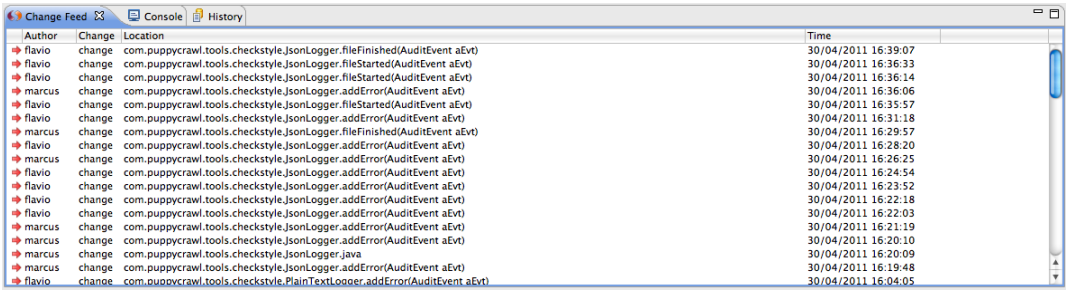


Figure 6.2. Example of a list of changes

Task 3 – Finishing class JsonLogger. In this task, participants were asked to finish the implementation of class `JsonLogger`. For one participant, this meant completing the implementation of method `addError`, and implementing method `fileFinished`. For the second participant, it meant completing the implementation of method `addError` with a different functionality than the other participant, and implementing method `fileStarted`. Though the description for this task is similar to that of task 2, the changes that participants were asked to implement to the method `addError` were different in all four cases.

Each task was accompanied by instructions to help prepare for it, containing a list of the tools and views that participants were or were not allowed to use for each task.

6.4.4 Pilot Studies

It took several iterations to successfully plan the complex task of a user study in which two participants are required to collaborate through IM, and merge conflicts are guaranteed to appear. In our first pilot study, we initially tried to reduce the complexity of the study by simulating the second member of a developer pair, so that only one human developer would be involved at a time. This proved to be unfeasible; the simulation of the second participant suffered several limitations. For instance, it was difficult to simulate the decisions an average developer would take, as well as make sure the participant did not notice the second person is actually a simulation.

We then redesigned the assignment by creating a set of complementary tasks for two participants, and ran a second pilot study. Though the complexity of the user study initially discouraged us from trying it out at first, this second pilot study showed that it was possible to run it. The merge conflicts appeared when we planned them to, and the participants, who in some cases were located on different continents, were able to communicate over chat and solve the tasks collaboratively.

6.4.5 Operation

The user study is composed of runs, with each run involving two participants who must change the object system in parallel to solve the tasks. The set of tasks each of the two participants receives is different, but closely related to that of their partner for the run.

Each run consists of a training session of approximately 15 minutes and one experiment session. The training session consists of a tutorial on the views provided by Syde, a hands-on session with a toy system where the participants can intentionally cause conflicts and experiment

with the views, and a warm-up task to allow the participants to get used to the object system. The experiment session is composed of three programming tasks, which participants are given unlimited time to solve. The first task is performed without preemptive conflict detection, while the second and third tasks are performed with one of either the list or the graph view.

Six experiment runs were conducted, each with slightly different conditions, which are summarized in Table 6.3.

In runs 1-3 and 6, the participants used laptops with 2 GB and 4 GB of RAM, running Mac OS X, with Eclipse and Skype installed. In runs 4 and 5, the participants used their personal computers to run a VirtualBox image with Ubuntu 10.04 and 1.5 GB of RAM, with Eclipse installed and Gmail chat for communication. The VirtualBox image might have constituted a disadvantage to the participants of runs 4 and 5, since it runs more slowly than the laptops with Mac OS X.

In runs 1-3, the participants had distinct sets of tests to fix. However, we observed that not being explicitly informed of this proved to be an additional barrier to communication while they tried to understand that they had different tasks and tests. For runs 4-6, the two set of tests were combined together, so a participant could see that there were failures that the other participant was responsible for fixing, allowing them to immediately infer that they had different tasks.

In runs 1-3 the participants used the list view for task 2 and the graph view for task 3. In runs 4-6 the participants used the graph view for task 2 and the list view for task 3. This swap of the views avoids serial position effects.

Runs 1-3 and 6 took place in a reserved room at the University of Lugano, while runs 4 and 5 took place in a laboratory (shared with other people) at the University of British Columbia.

Table 6.3. Summary of settings for the experiment runs

Task group	Configuration	Test set	List view	Graph view	Location
1-3	2GB or 4GB RAM, Mac OS X	Distinct	Task 2	Task 3	Univ. of Lugano
4-5	VirtualBox, 1.5GB RAM, Ubuntu 10.04	Equal	Task 3	Task 2	Univ. of British Columbia
6	2GB or 4GB RAM, Mac OS X	Equal	Task 3	Task 2	Univ. of Lugano

6.5 Data Analysis

In this section we analyze each run individually and address the research questions in Section 6.6. For each run (R), we first describe the participants' backgrounds based on the screening questionnaire. , and then describe any unexpected events or problems that happened in the run. We then summarize the observations for each participant (P), and finally correlate it to the data collected from the debriefing questionnaire and the interview. The complete answers to the questionnaires can be found in Appendix A.3. For neutrality reasons, all participants will be referred to with the masculine pronoun, which does not imply that all participants are male.

6.5.1 Run 1

The participants (P1, P2) of this run are Master students in Computer Science, each with at least 6 years of experience in Java development and use of Eclipse. They consider themselves advanced users in the subject. P1 has 4 years of experience with SCM systems and 3 years of experience working in teams, while P2 has 2 years of experience with SCM, and considers himself knowledgeable about working with teams, though he has had little direct experience with them.

P1 indicated that he usually works in teams of 3-5 people, and uses Git and SVN. He checks in his code more than once per day, but does not have to resolve conflicts frequently, because the members of the team he works with take different roles and tasks. P2 does not work in teams, and seldom uses SCM systems.

Problems. A couple of problems happened in R1 that directly influenced the behavior of the participants throughout the user study, as well as their opinions about emerging conflict and the study itself.

First, the participants did not follow the instructions given both by the experimenter and the handout regarding when to use the views of emerging conflicts for each task. Both participants used the list and the graph to visualize emerging conflicts for task 1, while P2 used the list for part of task 3 rather than the graph.

In addition, P1 did not understand how to use the graph, and thus was unable to see anything with it. This indicated that participants require more time to get comfortable with using the views, even though the steps describing how to open and load the graph view were given in the preparation instructions for task 3.

Observations from Videos

Table 6.4 shows the frequencies for participant P1's observed activity, taken from a 60-minute screencast of P1's computer. There are a total of 93 events, the majority of which are communication events (22). The annotations of the events of P1's programming session are incomplete for two main reasons: (i) the video recording application crashed when P1 was at the end of task 1, so most of the events for this task were lost; (ii) we restarted the video recording before the beginning of task 2 with a reduced capture window that followed the cursor, in hopes of avoiding further crashes. Some of the notifications (especially message notifications) that were located outside the window .

Table 6.5 shows the frequencies for our observations of participant P2, taken from a 140-minute screencast of P2's computer. The list of events is complete and covers the entire programming session. There are a total of 178 events, the four most frequent of which are communication (29), message notification (22), interaction with the list (22), and interaction with the graph (21).

Task 1. Since P1's screencast for task 1 was almost fully lost, our summary of the events for this task is mostly based on the events from P2's session. Even though the participants were not supposed to use the emerging conflicts views during this task, P2 keeps both views open, with the list in focus. At the beginning of the task, P1 tells P2 that he modified `MethodCountCheck`, and asks whether he has got any issues, to which P2 responds negatively. After P2 also modifies `MethodCountCheck`, a yellow warning appears, notifying him that they are both modifying method `checkCounters`. P1 asks P2 whether he sees the notification, to which he answers

Table 6.4. Observation frequencies for participant P1 – total duration of the events is given in seconds

Events	**T1**	**Dur.**	**T2**	**Dur.**	**T3**	**Dur.**	**Total**	**Dur.**
Communicating	1	19	14	285	7	278	22	582
Conflict graph	0	0	0	0	4	567	4	567
Conflict list	1	592	5	361	5	934	11	188
Checking in	0	0	2	31	2	33	4	64
Merging	0	0	2	197	0	0	2	197
Resolving conflict	0	0	0	0	0	0	0	0
Synchronizing	0	0	6	25	3	14	9	39
Updating	0	0	3	13	0	0	3	13
Viewing changes	0	0	4	371	0	0	4	371
Running tests	0	0	6	152	3	78	9	230
Message notification	1		4		1		6	
Interaction with graph	0		0		2		2	
Interaction with list	0		4		3		7	
Yellow conflict	0		1		0		1	
Red conflict	0		2		0		2	
Task starts	1		1		1		3	
Task ends	1		1		1		3	
Total	5		51		31		93	

positively, but the only further communication between them regarding this notification is P1 telling P2 that he is "waiting for something to happen".

P2 finishes his task and checks in the changes. When P1 tries to update the code, he gets conflicts and has to resolve them. At this moment, they discuss their tasks and find out what each other's tasks are. After P1 merges, checks in the code, and P2 updates it, the yellow warning disappears. P2 tells P1 everything is working, and they finish task 1.

An interesting thing to note is that P2 interacted quite a lot with the two views, and it becomes clear from observation that he had been trying to get used to them while performing the programing assignment. This is evidence that they did not spend enough time during the practice session to feel comfortable with the views before starting the tasks. P1's list of events (see Table 6.4) demonstrates that he also used the list view while solving task 1, which was not allowed.

Table 6.4 and Table 6.5 show that events regarding use of the conflict list appeared in all three tasks for P1 and P2. This is further evidence that the participants did not follow the instructions given about the use of the views for the tasks.

Task 2. Both the participants start working on method `addError` of class `PlainTextLogger`. When P2 is almost done, a yellow warning appears on the conflict list, to which P1 reacts by asking P2 whether he is changing `addError`. P2 has already checked in his changes, and lets P1 know. P1 then merges the changes from P2 and checks in his changes to `addError`. A red warning appears during modification of method `fileFinished`, which leads to a discussion of

Table 6.5. Observation frequencies for participant P2 – total duration of the events is given in seconds

Events	T1	Dur.	T2	Dur.	T3	Dur.	Total	Dur.
Communicating	12	790	12	320	5	70	29	1180
Conflict graph	5	685	0	0	1	964	6	1649
Conflict list	6	699	9	1277	4	308	19	2284
Checking in	1	42	2	101	1	11	4	154
Merging	1	2	1	6	3	20	5	28
Resolving conflict	0	0	0	0	1	70	1	70
Synchronizing	4	30	9	30	4	12	17	72
Updating	1	2	1	2	1	1	3	5
Viewing changes	1	41	1	74	3	289	5	404
Running tests	2	41	8	149	3	62	13	252
Message notification	13		7		2		22	
Interaction with graph	14		0		7		21	
Interaction with list	18		4		0		22	
Yellow conflict	1		1		0		2	
Red conflict	0		3		0		3	
Task starts	1		1		1		3	
Task ends	1		1		1		3	
Total	79		52		34		178	

whether P1 is changing it or not. In the middle of the task, both P1 and the experimenter realize he has the wrong set of tests, which causes some disruption. Another red warning appears for method `fileStarted`, but for both red warnings there is no further need for merging at check-in time.

Table 6.4 and Table 6.5 demonstrate that both participants spent a fair amount of time interacting with the list view, communicating, and viewing changes. The screencasts show that the information about emerging conflicts generated a good amount of communication. However, the participants seldom tried to understand the object of the other's task or how it related to their own solely from that information. This type of communication was only triggered after a first round of check-in and check-out.

Task 3. Unexpected events in this task prevented an effective simulation of parallel development using the graph view. First, P2 does not understand how to load the graph, and does not ask for help. As a consequence, the graph is empty during the entire task. Second, in the middle of the task, P1 switches to the list and uses that instead of the graph for the rest of the task. Due to these issues, P1 and P2 do not communicate for the first half of the task and do duplicated work on method `addError` of class `JsonLogger`. When P2 has to merge the changes previously checked in by P1, he struggles to resolve the conflicts and ends up doing it manually on the Java editor.

The number of communication events in task 3 is the lowest for all of the 3 tasks, if P1's incomplete data from task 1 is ignored. We attribute the lowered degree of communication to the absence of notification of emerging conflicts, which prevented the participants from realizing that

they were implementing the same method.

Discussion

P1 and P2's responses to the questionnaire indicate that P1 had to resolve conflicts for tasks 1 and 2, while P2 had to do the same for task 3. They both indicated that communicating with the other participant was somewhat helpful (agree/neither disagree nor agree); however, their opinions differed with regard to the usefulness of knowing about emerging conflicts, one participant answering "agree" while the other answered "disagree".

During the interview, the participants commented that they had mostly been communicating during check-in time:

"I think it was when we had to commit." (P1)

"So mostly we were communicating if there was trouble in merging." (P2)

In addition, P2 came to the conclusion that he had been rushing to check in his changes to avoid dealing with merging:

"... after the first commit I think that, for the first two tasks I was always focusing on committing first so I don't have to resolve conflicts. I reflected on it..." (P2)

Another issue that became evident through the interview was that the participants struggled to understand the concept of emerging conflicts. Due to this struggle, they were partially unable to benefit from the information with which they interacted. They strongly believed they had misused the tool by not being collaborative enough:

"You know what? When I saw that he was editing I was like 'we're gonna have problems'. So I think I just completely misused the thing because after seeing that done in a collaborative way you'd say 'what are you changing?'. Instead I was like 'we're gonna have problems'." (P1)

The participants mainly communicated when they had to merge code and resolve conflicts, even when information was present to help them while the conflicts were emerging. However, while the extra information might have not influenced the point at which they started talking, it seems to have influenced the amount of communication that they engaged in.

Regarding the different views, P1 affirmed that he prefers the graph view with its color metaphor to inform viewers of emerging conflicts, although he expressed his belief that a defect on the view had prevented him from seeing the information at times. P2 did not have a chance to use the graph, but expressed that the list was useful for identifying which code the other user was editing.

Finally, regarding other ways to visualize the emerging conflicts, P1 suggested changing the background color of the Java editor, or adding markers to its ruler. P2 thought that it would be beneficial to show the information in the editor, because then they would not need to keep a dedicated view open.

6.5.2 Run 2

The participants (P3, P4) of the run are PhD students in Computer Science, with 5 and 2 years of experience with Java development, respectively. They each have 1-2 years of experience in developing industrial-sized systems and in team development. P3 has 3 years of experience with using IDEs and SCM, and 2 years of experience with JUnit. P4 has 2 years of experience with using IDEs, and 1 year of experience with using SCM and JUnit. P3 is somewhat familiar with Checkstyle, while P4 has no familiarity with it.

These participants usually do not work in teams, and currently use language-dependent SCM systems for Smalltalk. They indicate that they check in code frequently (after new tests are created and run, and hourly), but they rarely have to resolve conflicts because of the granularity of the SCM they use (method level).

Problems. A couple of problems happened on R2 that may have influenced the behavior of the participants throughout the programming session, as well as their opinions about preemptive conflict detection and about the study. First, they ran into problems with incompatible tests, which provoked an extra loop of updating, running the tests, and checking in the changes for task 2. During task 3, the participants were unable to see any conflicts in the graph, because participant P4 kept a version of the code with compilation errors for the entire duration of the task, preventing the tool from detecting changes and, consequently, potential code conflicts.

Observations from Videos

Due to technical problems with the video recording application that were not detected during the experiment, we were unable to capture a screencast of P3's session for analysis. This run's analysis is therefore based on the events of P4 and on the communication log.

Table 6.6 displays the frequencies for events observed from P4's session, taken from a 81-minute screencast of P4's computer. There are a total of 205 events, the most frequent of which are message notification events (42) and communication events (39). Annotations from the task 3 events are incomplete because the experimenter interrupted this task as soon as P3 checked in his changes. The goal of task 3 was to provoke conflicts so that the participants might see them with the graph view, but as this did not happen there was no purpose in P4 finishing the task.

Task 1. The participants begin by implementing what is asked in the handout, with P3 finishing and checking in his code first. When P4 finishes and synchronizes his code, he notices that P4 has not added method `checkMax`, and starts to communicate with him. After sharing code over chat, they realize that they had been asked to make different modifications, but think their modifications reflect the same code behavior. They therefore discuss which one should be kept, and decide on P4's. P4 then replaces P3's code with his own, checks in the changes and tells P3 to update. When P3 updates the code, his tests break, and he realizes the functionalities of their codes were different. They exchange messages to clarify that they have different tasks. P3 redoes the modifications, checks in the code and tells P4 to update. P4 updates his code and tells P3 that it works without rerunning the tests.

The communication between the participants is totally concentrated after P3's first check-in and P4's first attempt to check in. Although they had communicated before P4 checked in to try to understand what the other was doing, before P4 checks in, the fact that they had not realized their implementations had different functionalities provoked an extra cycle of check-out,

Table 6.6. Observation frequencies for participant P4 – total duration of the events is given in seconds

Events	T1	Dur.	T2	Dur.	T3	Dur.	Total	Dur.
Communicating	9	568	27	525	3	60	39	1153
Conflict graph	0	0	0	0	7	1106	7	1106
Conflict list	0	0	11	1342	1	1	12	1343
Checking in	1	8	3	61	0	0	4	69
Merging	0	0	0	0	0	0	0	0
Resolving conflict	0	0	0	0	0	0	0	0
Synchronizing	4	11	11	40	0	0	15	51
Updating	1	3	2	7	0	0	3	10
Viewing changes	2	466	5	234	0	0	7	700
Running tests	2	49	8	241	0	0	10	290
Message notification	4		28		10		42	
Interaction with graph	0		0		33		33	
Interaction with list	0		18		2		20	
Yellow conflict	0		3		0		3	
Red conflict	0		4		0		4	
Task starts	1		1		1		3	
Task ends	1		1		1		3	
Total	25		122		58		205	

modification and check-in, performed by P3 after P4's check-in. Before the assignments had started, the experimenter had informed the participants that they had related tasks, which, to them, implied that they were the same. It was not the goal of the study to have participants spend extra time trying to understand that they had distinct tasks. This was a sign that the experimenter would need to be clearer about this in the next runs.

An interesting observation is that the participants twice shared code snippets of their modifications over chat to coordinate between themselves.

Task 2. The participants begin communicating at the beginning of the task, with P3 informing P4 he has to fix 2 tests, to which P4 replies saying that he also has to fix 2 tests. However, they do not verify whether these tests are the same or different. They start implementing, until P3 says he sees some conflicts. P4 replies by saying that he sees P3 is also working on method `addError` method. They continue the conversation by sharing code snippets of their modifications and realizing they are different. P3 asks P4 to check in first, so P3 can handle the merge. When P3 updates and runs the tests, he realizes the changes introduced by P4 break his tests. P3 fixes the code again and checks in the changes. When P4 updates the code, his tests break. At this point the experimenter is called, who realizes that the participants have incompatible tests and asks them to move to the next task.

Several interesting behaviors may be observed from this task, the first being the participants' attempt to coordinate their activities before starting to implement, probably in expectation of conflicts appearing, but with hopes to avoid them. Second, as soon as the participants saw a

conflict emerging, they communicated, and the more experienced developer offered to handle the merge. Third, in the middle of the task, P4 inadvertently overrode the changes introduced by P3 instead of merging them, consequently breaking a test that had previously been passing.

Task 3. The first part of this task is to implement a rather long method. While working on it, P4 keeps the code with compilation errors, inadvertently preventing Syde from capturing his changes and checking for conflicts. In addition, P3 is familiar with what is asked in the task, and quickly finishes it. Thus, no emerging conflicts are generated, and the participants do not get the opportunity to use the graph view as the study had intended.

Discussion

P3 and P4's answers to the questionnaire indicate that both had to merge code and resolve conflicts in task 1, while P3 had to merge and resolve conflicts in task 2. They indicate that communicating with the other participant had been helpful for performing the tasks (strongly agree), and that they communicated with each other as soon as they saw a conflict during task 2 (strongly agree/agree). When asked whether knowing about emerging conflicts helped them to better coordinate, P3 was neutral (neither agree nor disagree), and P4 was positive (strongly agree). This last difference in opinion is due to the fact that during task 2, the participants agreed for P4 to check in first, and P3 be responsible for merging.

When asked in the interviews whether emerging conflict information helped them to coordinate, they answered positively:

"... as soon as I noticed that P4 was changing something, then I asked him 'man, what are you changing?'." (P3)
" We coordinated before committing." (P4)

When asked whether knowing about existing conflicts before checking in was helpful for reducing the complexity of the merge, they were unsure:

"I'm not sure, because at the end... So probably the merge tool would also be helpful. So, it's more like a psychological thing. I would expect some conflicts to be there, so I will merge them. But I'm not sure it reduces the complexity." (P3)
"Maybe it reduces the overhead of the final merging, because you merge by kind of merging in place, right? Via Skype or via the tool. Maybe that complexity, but regarding the code complexity I think that no." (P4)

Indeed, knowing about the conflict in advance on task 3 caused them to deliver the changes in two smaller check-ins instead of a single one comprised of the changes in two methods.

Participant P3 seemed to be convinced of the usefulness of the tool:

"While I think that the usefulness of this tool is that you are working on something you have to do and then you see that someone else is also working on that. So in this case it was useful to coordinate maybe before doing the commit, but the real strength of this tool is that maybe P4 is in another city, we are working, and then I see P4 working in the same method, and then we can start interacting." (P3)

In addition, when asked whether they could think about situations in their everyday coding routine in which emerging conflict information would be useful, P3 expressed that he would feel more comfortable when working with other people if he could be aware of which parts of the system they were modifying.

"Well, in my case for example, I can think about the work we're doing with P4. I would have been more relaxed to know that P4 wasn't working on the same place I was working on. So, that would have been useful... and in the case we were working on the same thing, we would have chatted. No, I think it's useful." (P3)

The results of this run show that emerging conflicts were useful for the participants to communicate, exchange code, become familiar with one another's tasks, and coordinate code check-ins and merging. In addition, the participants (especially P3) felt that the information on emerging conflicts was helpful for the assignment, and would be helpful in their development environment.

Regarding the different views, P3 affirmed that he preferred the graph view, because he was able to use it instead of Eclipse's package explorer. However, the intent of this view is not to replace the package explorer. P4, on the other hand, expressed his aversion to graphs in general, and affirmed that his preference was for the list view.

The participants' suggestions for improving the emerging conflicts views were to add a quick view with a comparison of the conflicting code, or to have a shortcut from the conflict notification to a compare view of the code differences.

6.5.3 Run 3

The participants (P5, P6) of this run were Master students in Computer Science, with 4 and 5 years of experience with Java development, respectively. They each have 4 years of experience with team development, but no experience with developing industrial-sized systems. They have 3-4 years of experience using SCM, IDEs (specifically with Eclipse for Java development), and JUnit, and consider themselves advanced users. Both indicated that they have no experience with Checkstyle.

Both participants indicated that they often work in teams of 2-4 people. P5 uses SVN and Git, checks in the code daily, and rarely has to resolve conflicts. P6 uses SVN, checks in the code once or twice per day, and has to resolve conflicts 1-3 times per week.

Problems. A couple of problems happened during R3 that might have influenced the behavior of the participants throughout the programming session, as well as their opinions about emerging conflicts or the study itself. First, a defect on Syde prevented the participants from seeing the emerging conflicts at the beginning of task 2, and the experimenter did not detect the problem until task 2 was nearly over. This might have influenced the fact that the participants did not communicate until right before checking in changes for this task. Second, P6 did not initialize the graph view correctly until the middle of task 3, at which point he started to see the emerging conflicts that had been present from before that point.

Observations from Videos

Due to technical problems with the video recording application that were not detected during the experiment, we were unable to capture a screencast of P5's session for analysis. The analysis for this run is therefore based on P6's events and communication log.

Table 6.7 shows the frequencies for events observed from P6's session, taken from a 104-minute screencast of P6's computer. There are a total of 188 events, the most frequent of which are message notification events (36), followed by interaction with list (28), communication (26), and interaction with graph (21). Unlike in the recorded sessions from the first two runs, communication events do not appear as the most frequent event type. However, notification events are the first in the list, indicating that P5 tried to communicate several times with P6, but was on many occasions completely ignored.

Table 6.7. Observation frequencies for participant P6 – total duration of the events is given in seconds

Events	**T1**	**Dur.**	**T2**	**Dur.**	**T3**	**Dur.**	**Total**	**Dur.**
Communicating	20	1060	3	43	3	71	26	1174
Conflict graph	0	0	0	0	10	1283	10	1283
Conflict list	1	179	7	993	1	1	9	1173
Checking in	2	15	2	51	3	33	7	99
Merging	2	27	1	24	2	79	5	130
Resolving conflict	0	0	0	0	1	45	1	45
Synchronizing	2	7	0	0	3	6	5	13
Updating	4	25	1	2	1	5	6	32
Viewing changes	4	214	3	151	1	87	8	452
Running tests	3	46	4	58	6	121	13	225
Message notification	25		5		6		36	
Interaction with graph	0		0		21		21	
Interaction with list	0		28		0		28	
Yellow conflict	0		3		2		5	
Red conflict	0		0		2		2	
Task starts	1		1		1		3	
Task ends	1		1		1		3	
Total	65		59		64		188	

Task 1. Before the participants start the assignment, the experimenter explains that they will both be given three related, but not identical, implementation tasks. Despite this, a few minutes before the participants start implementing task 1, P5 asks P6 a technical question related to his task in a manner that assumes P6 has the same task. P6 simply ignores the question and replies saying that he is writing Javadoc. P6 then asks P5 whether method `checkMax` also has to log, to which P5 responds by arguing that there is no such method in his code, and that he only needs to change method `checkCounter`. They have a long discussion about what the task is asking them to do, until they realize they have different tasks. P6 has P6 meanwhile finished the task, and checks in the changes. When it is time for P5 to check in, he tells P6 he has never done it through

Eclipse. This is unexpected information for the experimenter, since in the screening questionnaire, P5 responded stating that he had 3-4 years of experience with using SCM, IDEs, and specifically Eclipse for Java development.

P5 tries merging the code, but ends up copying the code snippet he has implemented into the method `checkMax` that P6 introduced, which breaks the code. Ignoring this, P5 proceeds with the check-in. When P6 updates to the version P5 checked in and sees the errors, he simply reverts the code to the previous version ((the one he had checked in before, prior to P5's addition), checks it in again, and informs P5 he does not see any new checks, seemingly unaware that he has just erased P5's code snippet. P5 says he is going to add it again and asks P6 whether he has added any new code, to which P6 answers "the resolution to the conflict you introduced with the faulty class". With this answer, it is clear that P6 thought he actually resolved the conflicts correctly, which is not true. After re-implementing his changes, now fixing errors and making the tests pass, P5 has difficulties checking in the changes into the repository, and asks the experimenter for help. After the experimenter explains how to proceed, he checks in the code. P6 updates to the new code and successfully runs the tests.

A careful observation of P6's screencast points out a few difficulties he had when dealing with conflicting versions in task 1. He mistakenly erased the changes introduced by P5 without even trying to understand the meaning of these changes. In addition, he clearly thought he had resolved the conflicts, unaware that he had only erased the changes introduced by P5. This behavior indicates that P6 is also fairly inexperienced with merging code, although on his questionnaire he had also indicated having 3-4 years of experience with using SCM, IDEs, and specifically Eclipse for Java development.

Task 2. Unlike what happened in task 1, the participants do not communicate while implementing the changes in this task. Due to a defect on Syde, the list of emerging conflicts remains empty until the participants finish implementing the changes related to this task and restart Eclipse. Meanwhile, P5 finishes his implementation and says he is going to check in the changes. P6 sees that there are three emerging conflicts and waits for P5 to check in. While waiting, P6 tries to investigate the emerging conflicts by clicking on them (probably trying to see the code differences for each conflict).

Once P5 informs P6 that he can update, P6 opens the Compare editor with the latest version from the repository and copies the new code to his local version, instead of using SVN's synchronize and update options. When he tries to commit the merged code, SVN informs him that he does not have the newest version. After that, he updates the code in the proper manner, verifies that the tests still pass, and checks in the changes.

Communication between participants in this task was restricted to coordinating at check in time, unlike what happened in the first task. They finished the task much faster, but still had trouble merging changes.

Task 3. At the beginning of this task P6 does not correctly follow the instructions for initializing the graph. He only realizes his mistake after he finishing his implementation of the changes, at which point he sees the emerging conflicts that had been occurring during the task. Right after this, P5 asks whether P6 has already checked in, to which P6 responds that he is waiting so that he can resolve the conflicts.

After P5 checks in, P6 again opens the Compare editor with the newest revision and copies the new code into his local version. During the copying process, he introduces a code redundancy

that breaks two tests. Meanwhile, P5 tells P6 to ask if he does not understand P5's code, and proceeds to explain what he did. After the explanation, P6 realizes the problem in the code that is breaking the tests, fixes it and tries to check in. Because he had again failed to update, SVN informs him that he has an outdated version. When he updates, SVN performs an automatic textual merge, which breaks the code. P6 performs a straightforward fix: he erases the code that had been copied from the newest version into his own (since he had already done the merge through the Compare editor). Once more, P6 runs the tests and checks in the changes, finishing the assignment.

Discussion

P5 and P6's answers to the questionnaire indicate that P5 was the participant to merge code and resolve conflicts in task 1, while P6 chose to do this for tasks 2 and 3. Overall, they think that communication was helpful for coordinating themselves to perform the tasks (agree), except for P6's answer for task 3, where he disagreed that it was helpful. When asked whether knowing about emerging conflicts was an incentive for them to communicate for each task, they had slightly different opinions: P5 was neutral for task 2, but agreed for task 3; while P6 strongly disagreed for task 2, and disagreed for task 3. Their answers are a reflection of the behavior observed from the screencast: P5 was much more communicative and proactive than P6. When asked more specifically whether knowledge of emerging conflicts helped the participants avoid them at check-in time, P5 was neutral while P6 agreed.

At the end of task 1, when P5 had difficulties merging and checking in, P6 decided to wait during the following tasks so that he could perform the merge. Although the decision was made without a discussion, P5 was happy to accept it. During the interview, the experimenter asked P6 whether he took this decision to complete the tasks faster, to which he answered positively. This demonstrates the participants' expectation that the following tasks would involve conflict resolution, and the fact that they made an effort to coordinate themselves to perform the tasks faster. This decision is positive in that the more experienced participant showed willingness to help the least experienced. It is also negative, however, because it demonstrates a preconceived notion on the part of the participants that conflicts would occur, which likely influenced their behavior for tasks 2 and 3.

Indeed, the communication between the participants in tasks 2 and 3 was concentrated at check-in time. This was the main time when they shared code snippets and discussed their implementations. One interesting statement given by P6 during the interview was that after seeing the emerging conflicts, he could deduce what P5 was doing:

"They were also small tasks, so you can deduce after seeing the conflicts what the other was doing." (P6)

Because the experimenter noticed during the run that the participants had little experience with using SVN through Eclipse, she asked them what they thought of Eclipse's support for resolving conflict, to which P6 answered that it depends on the situation:

"Short answer is it depends. There are situations in which it's straightforward ... there is an arrow that copies from the version that he developed. In other kinds of situations it's tricky because if we are really working on the same piece of code and changing stuff, that is messy because you have to decide whether your version is correct or the other one is correct." (P6)

When asked which visual metaphors they preferred, both answered the list, explaining why:

"It was clear that, ok, it's red, then you have some conflicts. You read, and it specifies where and what's going on. And the dots, the yellow, red, are really useful because you don't spend time reading all of them, but you see, ok the red ones are really the important ones. You go through them." (P5)

When asked to comment about the disadvantages of the graph, P6's explanation evidences that he did not understand how to initialize it:

"The disadvantage I found is that in the third task I missed the view because of the layout of Eclipse, because it was centered, but in the view I was seeing it was black. Then I had to realize that I have to scroll and the center was there and the graph evolves on the right. And then I looked at it just at commit time, I think." (P6)

Though P6 here explained that he thought the graph was not showing up because he had to scroll down to see it, the fact was that he had not followed the steps given on the handout to create the graph when he initialized the view. This could have influenced his preference for the list.

When asked about any other ways that they would like to see information about emerging conflicts, they expressed a preference for highlighting the code on the Java editor directly. They suggested that highlighting should be a layer that could be disabled if it became disturbing for the developer.

6.5.4 Run 4

Participant P7 is a PhD student at the University of British Columbia, and participant P8 is a PhD student at the Federal University of Campina Grande and assistant professor at the University of Feira de Santana. They are experienced developers, with with 5 and 7 years of experience respectively in Java development, 5 and 4 years of experience in team development, and 2 and 3 years of experience in developing industrial-sized systems. Furthermore, they have used IDEs for 5 and 7 years, Eclipse for 3 and 5 years, SCM for 4 and 3 years, and JUnit testing for 2 and 4 years. Neither has previous experience with Checkstyle.

P7 currently uses SVN and works in a team of 2-3 people. He checks in weekly and deals with conflicts once a month. P8 currently uses CVS and sometimes works in teams of 3-4 people, but not at the moment. He checks in daily, but rarely has to resolve conflicts in his current project.

Problems. A couple of problems happened on R4 that might have influenced the behavior of the participants throughout the programming session, or their opinions about emerging conflict information or the study itself. First, there was a unplanned 30-minute break between tasks 1 and 2 because P7 had to meet a student who was working on a project with him. Second, at the beginning of task 2 the connection was lost and the participants were working offline for some time until the experimenter detected the problem. As a consequence, the participants did not see conflicts as they emerged, but instead only after reestablishing the connection. Third, P7 ignored the instructions at the beginning of task 3 and continued using the graph instead of the list.

Another problem occurred when P8 updated the tests to the version in the repository at the beginning of task 3; this removed the test from the user's workspace. After some time

trying to locate the test class, P8 called the experimenter, who had to copy it back to the workspace. This disturbed the start time of P8's implementation, giving P7 enough time to finish his implementation and check in the changes before P8 had effectively started, eliminating the goal of concurrent implementation. Lastly, P7 could not stay for the debriefing interview, so it was conducted only with P8.

Observations from Videos

A couple of problems occurred that influenced the data collected from the videos. First, the video from the screencasts is accelerated in comparison to the audio. Since we do not analyze the audio, and it was not possible to compute the acceleration factor, the duration of events is skewed: the further an event is towards the end of the recording, the faster it is in comparison with the original speed. Second, P7 kept the chat window outside the recording area, so his communication events could not be recorded. Checking the chat log, there are 40 communication events with a couple of exchanged messages in each of them for the entire session.

Table 6.8 shows the frequencies of P7's events, taken from a 98-minute screencast of P7's computer. There are a total of 137 events, the most frequent of which are interaction with graph (41), running the tests (24), use of the graph view (20), and synchronization (13). Communication events were lost, because P7 kept the chat window outside the recording area. Moreover, message notification events do not exist on Gmail chat, which limits the observations drawn from P7's communication strategies to what may be gleaned from the chat log.

Table 6.8. Observation frequencies for participant P7 – total duration of the events is given in seconds

Events	**T1**	**Dur.**	**T2**	**Dur.**	**T3**	**Dur.**	**Total**	**Dur.**
Communicating	0	0	0	0	0	0	0	0
Conflict graph	0	0	13	1177	7	457	20	1634
Conflict list	0	0	4	24	0	0	4	24
Checking in	4	52	1	16	1	30	6	98
Merging	0	0	0	0	0	0	0	0
Resolving conflict	0	0	0	0	0	0	0	0
Synchronizing	4	22	2	20	7	61	13	103
Updating	2	20	2	7	3	14	7	41
Viewing changes	1	29	1	24	0	0	2	53
Running tests	7	81	11	108	6	19	24	208
Message notification	0		0		0		0	
Interaction with graph	0		28		13		41	
Interaction with list	0		9		0		9	
Yellow conflict	0		1		2		3	
Red conflict	0		0		2		2	
Task starts	1		1		1		3	
Task ends	1		1		1		3	
Total	20		74		43		137	

Table 6.9 shows the frequencies for events observed from P8, taken from a 92-minute

screencast of P8's computer. There are a total of 126 events, the most frequent of which are communication (30), interaction with graph (16), and synchronization (16).

Table 6.9. Observation frequencies for participant P8 – total duration of the events is given in seconds

Events	T1	Dur.	T2	Dur.	T3	Dur.	Total	Dur.
Communicating	15	314	6	160	9	149	30	623
Conflict graph	0	0	11	658	1	1	12	659
Conflict list	0	0	3	8	7	780	10	788
Checking in	1	23	1	20	2	23	4	66
Merging	0	0	4	32	2	26	6	58
Resolving conflict	0	0	0	0	0	0	0	0
Synchronizing	8	76	3	25	5	42	16	143
Updating	2	9	0	0	0	0	2	9
Viewing changes	1	26	1	134	2	117	4	277
Running tests	5	88	4	68	4	70	13	226
Message notification	0		0		0		0	
Interaction with graph	0		15		1		16	
Interaction with list	0		0		2		2	
Yellow conflict	0		1		2		3	
Red conflict	0		0		2		2	
Task starts	1		1		1		3	
Task ends	1		1		1		3	
Total	34		51		41		126	

Task 1. The participants begin implementing their tasks; P7 finishes the implementation and tries to communicate with P8, who does not respond. P7 proceeds with trying to check in his code, but P8 has already checked in his changes. Meanwhile, P8 sees that P7 was trying to talk to him, and answers to the chat saying he is going to update the code. Because there is already a new version of `MethodCountCheck` in the repository from P8's check-in, P7 does not manage to check in with his own changes. While P7 updates to the newest version, P7 and P8 discuss their tasks. P7 then examines the differences between the versions and decides to overwrite his local copy, which results in the changes he had implemented being completely erased. It becomes clear that P7 had not realized that the two implementations were semantically different, because after re-running the tests and getting two failures (the same ones he had gotten in the beginning of the task), he says that P8's changes broke his tests, which is not true. He then re-implements his changes, checks them in and informs P8, who updates the code and successfully re-runs the tests.

Task 2. The first half of this task is disturbed by the loss of connection with the server, which prevents the participants from seeing the first emerging conflicts as they occur. When P7 is about to finish and P8 is halfway through the implementation, the experimenter detects the problem and asks them to reconnect. P8 immediately observes the emerging conflict in `PlainTextLogger` and contacts P7 to tell him what methods he is modifying. P8 also says what he is changing,

and they identify that they have conflicts only in method `addError`. They decide that P7 should check in his changes and P8 can handle the merging. After P7 checks in, P8 merges the code straightforwardly, runs the tests, and checks in the code. P7 then updates and successfully re-runs the tests.

Task 3. In this task, P7 keeps on using the graph view instead of using the list as instructed. In addition, at the beginning of the task, P8 updates the tests to the version in the repository, which removes the test he has to fix from the workspace. After some time trying to locate the missing test class, P8 calls the experimenter, who has to copy it back to the workspace. Meanwhile, P7 continues looking at the graph while performing his implementation, but no emerging conflict appears until he has finished and checks in. The same moment P7 checks in, an emerging conflict appears, and P8 asks P7 whether he is changing `JsonLogger`. P7 says he has already checked in and asks P8 whether he had been saving his file, to which P8 answers he was. The task ends with P8 finishing his implementation, merging P7's version into his code, and checking in. Unfortunately, no real parallel coding happened in this task.

Discussion

P7 and P8's answers to the questionnaire indicate that P7 had to merge code for tasks 1 and 2, and P8 had to merge code for tasks 2 and 3. They generally agreed that communication was helpful for coordinating themselves to perform the tasks (strongly agree - P7, agree - P8), except in the case of task 3 for P8, who was neutral about it. For tasks 2 and 3, both participants saw emerging conflicts and communicated as soon as they saw them (agree). However, their opinions differ on whether knowing about conflicts in advance helped them to avoid them at check-in time (agree - P7, disagree - P8). Indeed, P8 was the one to merge code in the last 2 tasks, and their communication after seeing emerging conflicts revolved around coordinating who would check in first, and who would handle the merging.

During the interview, the experimenter asked how communication took place after the participants saw emerging conflicts, to which P8 answered:

"I remember that I was asking him some questions about which methods he was changing... It seemed to be not really hard tasks, so it was easy to solve the conflict, but we still had to talk about it at least to decide who was going to fix first, who was going to commit the code first." (P8)

The coordination was kept shallow, and P8 was never aware of what P7's task was.

"No, not really. I just knew that he was changing some code in parallel with me. That was it." (P8)

At the beginning of task 2 there was a disruption caused by a connection loss. When asked about it, P8 expressed that it disturbed the task:

"After we got the connection back, we saw it (the conflict). That kind of disturbed the whole task. Made it feel a little bit artificial." (P8)

However, despite the disruption, P8 expressed that he preferred the graph view over the list:

"The graph. I think it looks better. Especially the graph that you filter out all the design, and only have the emerging design. That's more interesting to me. I think lists... they can be ok, but they don't allow you to focus on what's going on. When you have the graph, you immediately see the issues. There's not too much information. There's just some colors that just show you like, 'oh, there's something going on here', and if you want to take a look, then you just point to the node and see what's going on. When you have a list, you kind of get lost with so much of information." (P8)

P8 expressed that for a project with very few people involved, he does not think there is a need for preemptive conflict detection, but that he might change his mind for a project with many people and lots of concurrent modification. In addition, when asked whether he could think of other ways to visualize emerging conflicts, he proposed the addition of cues to the package explorer. This way, he could use it in combination with Mylyn, and see emerging conflicts only for the classes he is focusing on without having to use another view.

6.5.5 Run 5

Participant P9 is a PhD student at the Federal University of Campina Grande, and P10 is a PhD student at the Federal University of Minas Gerais. They have 5 and 7 years of experience respectively with Java programming. They both have 3 years experience in team development, and they have 1 and 2 years respectively in developing industrial-sized systems. They have used SCM, IDEs, and specifically Eclipse for 5 and 6 years, and have 5 and 7 years of experience with JUnit. P9 has no previous knowledge of Checkstyle, while P10 has 1 year of experience with it as a user and considers himself comfortable with it.

P9 currently uses SVN and works in a team of 2 people. He checks in the code once a week, and has to resolve conflicts once a month. P10 uses Git, SVN, and Rietveld, and works in teams that vary from 2 to 5 people depending on the project. Currently he is working on a project with one other person, and thus checks in his code every couple of days, but does not have to resolve conflicts frequently.

Observations from Videos

A couple of problems happened that influenced the data collected from the videos. The video of the screencasts is accelerated in comparison with the audio. Because we ignore the audio and because it was not possible to compute the acceleration factor, the duration of events is skewed: the closer to the end of the screencast an event is, the faster it plays in comparison to the original timing. In addition, the experimenter forgot to start recording P9's session, and only realized this at the end of task 1. Thus, no events were recorded for P9 for task 1.

Table 6.10 shows the frequencies for events observed from P9's session, taken from a 58-minute screencast of P9's computer. There are a total of 225 events, the most frequent of which are interaction with graph (63), communication (46), viewing the conflict graph view (31), and viewing the conflict list view (29). Message notification events do not exist on Gmail chat and were therefore not captured.

Table 6.11 shows the frequencies for events observed from P10's observed events, taken from a 96-minute screencast of P10's computer. There are a total of 131 events, the most frequent of which are communication (37), interaction with graph (20), and running test (16). Message notification events do not exist on Gmail chat and were therefore not captured.

Table 6.10. Observation frequencies for participant P9 – total duration of the events is given in seconds

Events	**T1**	**Dur.**	**T2**	**Dur.**	**T3**	**Dur.**	**Total**	**Dur.**
Communicating			22	125	24	210	46	335
Conflict graph			29	679	2	45	31	724
Conflict list			0	0	29	472	29	472
Checking in			1	4	2	14	3	18
Merging			3	26	2	11	5	37
Resolving conflict			0	0	0	0	0	0
Synchronizing			5	20	5	14	10	34
Updating			1	3	1	2	2	5
Viewing changes			2	55	2	32	4	87
Running tests			6	33	9	67	15	100
Message notification			0		0		0	
Interaction with graph			58		5		63	
Interaction with list			0		6		6	
Yellow conflict			2		2		4	
Red conflict			1		2		3	
Task starts			1		1		2	
Task ends			1		1		2	
Total			132		93		225	

Even though P9's screencast does not include the events of task 1, he has generated almost double the number of events P10 did. In particular, P9 has a high number of communication events and interactions with the views. This occurred because P9 had full-screen windows of both Eclipse and the chat window, between which he constantly switched, while P10 had a small chat window that was able to fit in the same screen on top of Eclipse. Indeed, if the duration of events is inspected instead of the number, it can be seen that P10's total event duration is larger than P9's, which is to be expected since the events from P9's first task are missing.

Task 1. The participants begin implementing their tasks, and P9 manages to finish first. He checks in the code and lets P10 know that there is new code in the repository. When P10 finishes his implementation, he checks the changes implemented by P9. Upon viewing the changes, he is unable to merge the code directly, because the compare view detects a conflict. He manually resolves this conflict by copying the code from P10's version into his, making the necessary changes to conform with his refactoring, marking the code as merged, and checking it in. P9 then tells P10 to update and re-run the tests, which P10 does successfully. This task was completed smoothly, with P10 handling the conflict and not breaking any tests.

Task 2. This task starts with both participants initializing the graph view and taking some time to interact with it. P9 spends more time interacting with the graph, but eventually they both move onto the implementation task. As soon as the participants see a warning of emerging conflict they start communicating. They ask each other what methods they are modifying, and discuss for a

Table 6.11. Observation frequencies for participant P10 – total duration of the events is given in seconds

Events	T1	Dur.	T2	Dur.	T3	Dur.	Total	Dur.
Communicating	4	111	13	356	20	241	37	708
Conflict graph	0	0	11	1012	1	1	12	1013
Conflict list	0	0	0	0	6	558	6	558
Checking in	1	16	2	37	2	35	5	88
Merging	1	3	3	17	4	52	8	72
Resolving conflict	1	31	0	0	1	39	2	70
Synchronizing	1	10	3	19	3	10	7	39
Updating	0	0	0	0	0	0	0	0
Viewing changes	1	17	2	46	2	132	5	195
Running tests	3	62	5	91	8	87	16	240
Message notification	0		0		0		0	
Interaction with graph	0		20		0		20	
Interaction with list	0		0		2		2	
Yellow conflict	0		1		1		2	
Red conflict	0		2		1		3	
Task starts	1		1		1		3	
Task ends	1		1		1		3	
Total	14		64		53		131	

while whether it is possible to see this information from the graph until P9 eventually discovers how. They decide P10 should commit the changes to `addError` (the method that both of them had changed) first, so P9 can merge it before they continue implementing the other methods. As soon as P10 checks in, P9 notices that the conflict warning on the graph has turned red. He goes to the chat and confirms that P10 has checked in the code. P9 then merges the code, finishes implementing, checks in the code and tells P10 to update. P10 updates, re-runs the tests and checks in his changes to method `fileFinished`, and tells P9, who successfully updates the code and re-runs the tests. This task also completed smoothly, with the participants taking the strategy of breaking the check-in into smaller chunks when they noticed a potential conflict emerging.

Task 3. For this task, instead of jumping into the code immediately, the participants start by asking each other what methods they will be modifying. They find out there is a common method and decide to adopt the same strategy as they had adopted for task 2: to check in and merge the method they had in common as early as possible, and then continue working on their other implementations. P9 finishes the implementation of `addError` first and checks in so that P10 can merge the code. P10 does this and is not required to resolve any conflicts, and checks it in with all tests referring to `addError` passing. Meanwhile, P9 finishes the implementation of `fileFinished`. As soon as P10 checks in with the merged `addError` code, P9 merges the new code and checks in the changes for `fileFinished`. By this point P10 has finished his implementation for `fileStarted`, so he performs the last merge and checks in so that P9 may do the final update. P9 does so, and successfully re-runs the tests.

Discussion

The participants' answers to the questionnaire indicate that P9 had to merge code in tasks 2 and 3, while P10 merged code in all three tasks. Both participants think that communication was helpful for coordinating themselves to perform the task (agree - P9, strongly agree - P10). When asked about preemptive conflict detection, they had different opinions. P9 saw emerging conflicts and communicated right after seeing them (strongly agree), which he believed was very helpful to avoid needing to resolve them at check-in time (strongly agree). Conversely, P10 did not see emerging conflicts and did not think they triggered communication (strongly disagree). P10 was neutral about whether knowledge of conflicts helped them to be avoided at check-in time (neither agree nor disagree). P10's negative answer about seeing emerging conflicts, however, is not consistent with his responses in the interview, where he talks about details of the two views and how they displayed conflicts emerging while he and P9 were implementing the tasks.

In general, both participants behaved similarly when updating code that had been changed by the other. They first synchronized the code, then viewed the differences between the local and the remote versions. Once they understood the differences, they merged the code using Subversive's built-in automatic merge function. After that, they marked the file as merged and checked it in. This process did not change upon the introduction of preemptive conflict detection. What changed was instead the frequency and granularity of the check-ins – a fact that the participants also recognized. Knowing in advance about the presence and location of conflicts caused them to decide to resolve them before continuing with their coding. By making the check-ins more frequent and the granularity of the changes smaller, the participants believed the conflict resolution process became less complex:

"I think it did help, because we started communicating before actually committing all the code, and before doing other changes in other methods we started communicating when we detected conflicts. So it helped to see and synchronize what things we were adding and committing the conflicts first to first resolve all the conflicts and then continue." (P9)

"I think like it was easier to detect conflicts and do the merge only with small changes instead of looking through all the code." (P10)

When asked about which view they preferred, both said that they preferred the list for this assignment in particular, but that the graph could be useful in other situations (*e.g.* when more people are involved, or when there are conflicts in dependent classes):

"Well, in my opinion, I think the second view (the list) is a bit easier to use, because it shows only the classes that are actually in conflict. So it filters for you, and you don't need to see the whole thing, you don't need to, even in the graph view you can go to the square class (node) that you changed and you can see the same thing, but you have to do it manually. You have to go there and see what is changing. Maybe the good thing is that you can see if dependencies are being checked at the same time... but I think the second view was much more straightforward. You can see exactly what's wrong and resolve conflicts." (P9)

"I would agree that the second view is better, but the first view (the graph), like P9 said, I didn't think about this before like, you could see dependencies being changed, but the test didn't have any. We were always changing one file only. So the second view is better for that, because you have little changes and you can see. Maybe if we were changing larger files the first would be good." (P10)

P10 also reported that the graph was less intuitive, and that they had difficulties realizing that when a node was hovered upon, it showed information of which methods of a class were in conflict:

"I think I didn't know this for a long while. I just saw the square changing colors and I think at the end of the task while I was waiting for P9 to commit something, I started playing around and then I saw that there was like 'oh, if I put the mouse here then I can see the changes', but it wasn't clear that, and it's not intuitive that you were gonna see the changes like that." (P10)

P9 suggested that the graph view could be improved by adding a search option to find a node quickly. When asked about other ways to warn developers of potential conflicts, P9 suggested adding annotation directly into the Java Editor to avoid having to use a separate view.

Finally, when asked whether preemptive conflict detection would be helpful for their everyday coding, both said it would be useful only in the context of a team greater than two:

"... if you code with someone else, there is two people and maybe that's easier to synchronize, but in the past definitely, when we worked on Ourgrid and stuff, there were always conflicts. Maybe having a large project and, maybe not even on the class scale there aren't two people editing the same class, but there's a dependency that both people are editing and there's gonna be a problem. And also, some people just don't want to commit right away, and you're doing the same thing he did or whatever. So that would be helpful." (P10)

"Nowadays the team is only two people working, so we don't really need something like that, because when we plan to do something on the code, we always try to share the most and try not to have conflicts, but on the planning phase, not afterwards when implementing. But actually, sometimes even with two people it happens. It happens sometimes with me, so it would be useful and even more useful on the past when I worked with 5-6 people in the same project, I think it would be even more useful. " (P9)

6.5.6 Run 6

The participants (P11, P12) of this run are Master students with 5 and 4 respective years of experience in Java development, 6 years and 1 year of experience with team development, and no experience with developing industrial-sized systems. They have 5 and 4 years of experience using IDEs, 3 years of experience each with Eclipse, 4 and 2 years of experience with SCM, and 3 years and 1 year of experience with JUnit. They have no previous experience with Checkstyle.

The participants currently use SVN and often work in teams of 3-5 people (P11) and 5 people (P12). P11 reports that his check-in frequency depends on how involved he is in the project. For projects in which he has a relevant role, he prefers to check in many times each day in hopes of avoiding conflicts, though he still has to deal with them once a week. P12 reports that he checks in often, and rarely has to resolve conflicts. He says sometimes he forgets to check out before starting to work, which causes the majority of his conflicts.

Observations from Videos

Table 6.12 shows the frequencies for events observed from P11's session, taken from a 92-minute screencast of P11's computer. There are a total of 72 events, the most frequent of which are communication (21), message notification (11), and running tests (8). The number of events

Table 6.12. Observation frequencies for participant P11 – total duration of the events is given in seconds

Events	T1	Dur.	T2	Dur.	T3	Dur.	Total	Dur.
Communicating	4	1644	5	330	12	1016	21	2990
Conflict graph	0	0	4	1249	0	0	4	1249
Conflict list	0	0	0	0	1	2028	1	2028
Checking in	2	34	2	63	1	22	5	119
Merging	0	0	0	0	0	0	0	0
Resolving conflict	0	0	0	0	0	0	0	0
Synchronizing	0	0	0	0	0	0	0	0
Updating	0	0	1	5	3	11	4	16
Viewing changes	0	0	0	0	0	0	0	0
Running tests	1	15	3	33	4	47	8	95
Message notification	0		6		5		11	
Interaction with graph	0		3		0		3	
Interaction with list	0		0		4		4	
Yellow conflict	0		1		1		2	
Red conflict	0		1		2		3	
Task starts	1		1		1		3	
Task ends	1		1		1		3	
Total	9		28		35		72	

collected from P11's screencast is the lowest of all the study participants; this is because P11 managed to finish the implementation of all his tasks before P12 (and sometimes even before P12 had started). This meant P11 was not required to merge or resolve conflicts at all throughout the assignment. Additionally, after P12 had checked in, P11 was supposed to update his code and re-run the tests, but he did not actually do this for any of the tasks.

Table 6.13 shows the frequencies for events observed from P12's session, taken from a 93-minute screencast of P12's computer. There are a total of 132 events, the most frequent of which are message notification (41), communication (32), and running tests (25).

Task 1. The participants start to implement the task. When P11 finishes, he asks P12 whether he is done, but P12 replies that he is still working. P11 waits for a while, but then decides to check in his changes to the repository. After a while, P12 also finishes and repeatedly tries to check in, but fails each time for various reasons. In his first attempt, P12 tries to check in the entire project, but keeps receiving a message that some folders in his copy are outdated. P12 then updates his code, which introduces conflicts between his new code and P11's changes. Faced with a broken class, P12 tries to understand the changes P11 introduced. Meanwhile, P11 suggests that P12 copy the new code he is attempting to introduce, revert the code, and paste the code back in. P12 reverts the code, but forgets to copy what he had implemented, and thus has to re-implement everything from scratch. Once he re-implements his changes, he tries to check in the code twice with no success. After, telling to P11 he could not check in, P11 alerts him that he should only be checking in the classes from the source folder. After a total of four failed attempts,

Table 6.13. Observation frequencies for participant P12 – total duration of the events is given in seconds

Events	T1	Dur.	T2	Dur.	T3	Dur.	Total	Dur.
Communicating	13	196	9	161	10	106	32	463
Conflict graph	0	0	2	1057	0	0	2	1067
Conflict list	0	0	0	0	1	2031	1	2031
Checking in	5	150	1	30	1	43	7	223
Merging	1	10	1	8	0	0	2	18
Resolving conflict	1	174	1	41	0	0	2	215
Synchronizing	0	0	0	0	0	0	0	0
Updating	2	23	1	7	2	9	5	39
Viewing changes	0	0	0	0	0	0	0	0
Running tests	3	50	5	84	17	293	25	427
Message notification	14		12		15		41	
Interaction with graph	0		3		0		3	
Interaction with list	0		0		2		2	
Yellow conflict	0		1		1		2	
Red conflict	0		0		2		2	
Task starts	1		1		1		3	
Task ends	1		1		1		3	
Total	41		38		53		132	

P12 checks in the code and informs P11. P11 does not re-run his tests and they finish task 1.

Task 2. In this task, P12 takes the initiative by asking P11 what his task is before they start implementing. However, neither participant takes any extra action after they share their assignments. In addition, P12 starts by fixing a test that is P11's responsibility to fix. P11 implements his changes quickly, but when he is about to finish P12 begins implementing method `addError`. The warning of emerging conflict only appears once P11 is just about to check in, and he immediately calls P12's attention to the existence of the conflict. P11 asks P12 whether he should check in, to which P12 answers positively. P12 directly updates the code without first inspecting the changes introduced by P11, which causes a conflict that breaks the code. At this point P12 realizes that he and P11 had implemented the same check (by referring to the conflict's severity level). P12 continues the implementation, and when all the tests for `PlainTextLogger` are passing, he checks in the code and lets P11 know. Once more, P11 does not update the code, nor re-run the tests.

Task 3. At the beginning of the task P12 again asks P11 what he has to implement, but P11 takes a long time before looking at his chat window. P11 finally answers P12 at the same time that a warning of emerging conflict in method `addError` appears. He asks P12 what he is doing, and draws P12's attention to the warning. P11 says he has fixed the test related to the message, and P12 says he is working on the test related to the security level. They decide that P11 should check in first and P12 merge the code, so P11 checks in. To merge, P12 copies the code he has

implemented, reverts the local copy of `JsonLogger` to the latest version in the repository (the one P11 checked in), and pastes his code into the updated version. This workaround prevents the appearance of conflicts, and can be considered an unconventional way of merging. P12 implements the second method of his task, checks in the code, and they finish the task, once more with P11 not performing the final code update and test re-run.

Discussion

P11 and P12's answers to the questionnaire confirmed that P12 performed all merges for tasks 1 and 2. For task 1 one, they indicated that communication was helpful for them to coordinate and to perform the task (agree), even though P12 had problems merging the code. For task 2 they thought that communication was helpful (agree), and that they communicated with each other as soon as they saw the emerging conflicts warning. They responded that knowing about conflicts in advance helped to avoid them at check-in time (agree - P11, strongly agree - P12), even though it did not prevent P12 from having to merge. For task 3, neither indicated they had to merge, despite the screencast showing that P12 merged code. They indicated that communication was somewhat helpful for performing the task (agree - P11, neither agree nor disagree - P12). Lastly, they communicated with each other as soon as they saw conflicts emerging for this task, and feel it was useful to help them to avoid merging code at check-in time (agree - P11, strongly agree - P12).

When observing the videos of P11 and P12, it became clear that P12 was unfamiliar with using SVN through Eclipse, despite his indication on the screening questionnaire that he currently uses SVN. During the interview, the experimenter asked P12 whether he experienced difficulties using SVN through Eclipse, to which he answered that this was his first time accessing SVN within Eclipse, because he usually code in Objective-C and uses SVN through the command line. This is probably the reason why P12 demonstrated a different behavior from the majority of other participants who performed merges. P12 neither synchronized the code, nor investigated the differences before updating; this resulted in him losing his implementation in the first task and having to redo it.

Apart from P12's lack of experience with using SVN through Eclipse, another issue was P11's speed in resolving the tasks, which almost prevented the emerging conflicts warnings from being useful. These warnings are most useful when both developers are initiating implementation on a conflicting part of the code, but can also be useful when just one developer is beginning.

During the interview P11 and P12 expressed that knowledge of emerging conflicts helped them coordinate, and that they did not need to communicate excessively to know what the other was modifying:

"I think it helped us to coordinate... We asked just 'which methods are you modifying?' just to prevent some modification." (P11)

"We didn't go in the specific like 'well I'm implementing...', only the method. It was sufficient to coordinate because we knew when we were editing the same method." (P12)

When asked when they talked to each other, they confirmed that they communicated when they saw conflicts emerging, but also tried to coordinate in advance:

"I contacted him when I saw a conflict and we started discussing why there was this conflict." (P11)

"And there were times we contacted each other just at the beginning of the task before starting to do anything. Maybe I asked him 'what part of the code are you going to modify?'. Well, for the conflicts we used just the view on Eclipse." (P12)

Both P11 and P12 preferred to view the warnings of emerging conflicts with the list view:

"In this example I don't really have one that I prefer, but I think that if we had to work for hours in a project, then the second one (list) for me is better because in a table you have all conflicts organized. In the other I didn't see, because I didn't have many files open and so on, but maybe it becomes a really huge graph and it might be difficult to scroll." (P11)

"The last one (list) maybe could be better because we are just working in pairs. If we were working in a group of one hundred developers, maybe the graph is not so easy to understand when you work." (P12)

The participants suggested adding visual cues directly into the Java Editor, which could be a mark on the ruler or a background color on the code.

6.6 Discussion

In the following section we discuss the research questions based on our analysis of the data collected from the six experimental runs. We first summarize the important findings derived from each experimental run, taking note of the behavior of each participant. Then, we analyze changes made to communication and coordination strategies, taking into account the participants' feedback collected on the interview and on the debriefing questionnaire. We note that the participants are naturally split into two distinct groups:

- **Beginners:** The participants of runs R1, R3, and R6 are MSc students with no experience developing industrial-sized systems. Though they reported experience with SCM systems, some of them clearly had little experience, and struggled with performing certain SVN operations or using them within Eclipse.
- **Advanced:** The participants of runs R2, R4, and R5 are PhD students with at least one year of experience developing industrial-sized systems. Most of them reported to have previously worked in the industry, and thus have practical experience in the field.

6.6.1 RQ1: How do developers behave when they have to merge code and resolve conflicts?

We derive the answer to this question from the data collected from task 1, which involved the improvement of a method for one developer, and the code refactoring from the same method into a second one for the other developer.

Beginners. The participants of run R1 seldom tried to coordinate, and concentrated their communication to after the first check-in. When they communicated, their intentions were to let the other know there were new changes in the repository, rather than to explain what these changes were. One participant (P2) observed that he rushed his work to avoid dealing with merge.

The participants of R3 actively communicated in the beginning of the first task to understand what the other's task was. They began communicating before check-in time. They were inexperienced at using SVN through Eclipse and struggled to merge code and resolve conflicts. P6 demonstrated an unconventional strategy at dealing with merge: he did not synchronize files before inspecting the changes, instead manually copying the new code into his local (and outdated) version, which provoked a failed merge. This behavior cost him time and increased the complexity of a simple merge.

In R6, the participants started communicating at check-in time, but kept this communication shallow, only sharing which methods they had to modify. This was the only run in which only one participant's (P12) merging behaviors could be observed, because he took responsibility for merging in all three tasks. Although he had experience with using SVN, he had no previous experience with using it within Eclipse. Because of that, he used a naïve strategy to merge his code. First, he updated his code directly, which introduced compilation errors. After unsuccessfully trying to solve these errors, he then reverted to the version with the other participant's changes and lost his own, requiring that he reimplement them.

Advanced. In R2, the participants started to communicate after the first failed check-in, by coordinating and sharing code snippets over chat. After this initial communication, they started to communicate, they actively coordinated with each other. Regarding the actual merge, they followed the expected behavior: synchronizing the code, inspecting the conflicting classes, and merging the code by resolving the conflicts either manually or with the help of Subversive. They then checked the code in.

The participants of R4 started to communicate at check-in time. They entered into discussion in order to identify each others' tasks, but did not realize that their changes were semantically different, and the subsequent removal of one of the changes broke the related tests. This generated an extra round of check-out, implementation, testing, and check-in. Regarding the actual merge, they followed the expected behavior.

The participants of R5 also started to communicate at check-in time. Their merging strategy was the same as for the participants of R2. Additionally, one participant (P10) was observed following the behavior to rush to check in first in order to avoid merging. The other participant had no difficulties understanding the semantic differences of the two implementations, and merged the code successfully.

Discussion. A common behavior observed was that while participants did communicate for coordination purposes, they only started to effectively communicate after the first check-in in all runs but one (R3), where participants communicated earlier. Communication generally started with one participant telling the other that there was new code in the repository, sometimes also providing an explanation of his changes. In some cases, this communication evolved into a discussion to better understand each other's implementation. In one instance of this, developers even shared code snippets to aid their partner's comprehension (R2).

Participants' coordination strategies were therefore concentrated around alerting their partner to new code, or understanding the necessary changes to merge successfully.

Regarding the merge strategies themselves, there was a clear distinction between beginners and advanced developers. While the advanced users demonstrated a common behavior, the beginners showed unexpected strategies to merge and resolve conflicts in two cases. Instead of synchronizing the code and inspecting it before merging, they directly updated the code, which

performed a textual merge and introduced conflicts in the form of compilation errors. In one of the cases, to avoid compilation errors, the participant reverted the code to the latest version his colleague had checked in, which completely erased his implementation.

A struggle to merge is expected when developers are inexperienced. However, a notable observation that some might not expect is that in most cases the advanced developers had problems when merging as well. In two cases while analyzing the differences in the code and discussing with their counterparts, developers failed to understand that their implementations were semantically different, and consequently failed to successfully merge. This observation confirms what Grinter observed through a field study [Grin 96]: even experienced developers face problems when merging code. Another behavior observed by Grinter and Souza [Souz 03; Grin 96] that we observed twice, when it was explicitly stated by a participant, is developers' tendencies to rush to check in first to avoid dealing with merge.

Summary. Our observations of developers' behavior when they have to merge code can be summarized as follows:

Beginners adopt various naïve strategies. Developers who are inexperienced with SVN tend to use different naïve strategies to merge code. These strategies seem to derive from strategies taken when using SVN through the command line. They struggle to understand how to properly merge the code, sometimes erasing their own changes, or erasing the changes of others.

Advanced developers share a working strategy, but struggle with merge. Experienced developers usually follow the strategy of synchronizing the code, inspecting the conflicting classes to understand their differences, merging the code by resolving the conflicts, and checking the merged code in. However, in many cases in our data they had difficulty understanding how to properly merge the code, introducing errors that broke the tests. This confirms Grinter's observation that even experienced developers face problems when merging code [Grin 96].

Developers communicate after the first check-in. With one exception, all developers in this study started to communicate after the first check-in or after the first failed attempt to check in.

Communication is kept shallow. In most cases, communication is kept at the bare necessity. Developers tend to communicate only when it is really necessary to continue the coding task; otherwise, they only code.

A minority of developers communicate more deeply. A few developers from this study communicated more, trying to explain the changes they had introduced or to understand the changes introduced by others. Some even shared code snippets over IM.

Some developers rush to check in first to avoid dealing with merge. Our observations of this behavior confirm previous findings that developers try to avoid dealing with merge by checking in changes before their colleagues [Souz 03; Grin 96].

6.6.2 RQ2: How does developer behavior change when information is present about emerging conflicts?

The answer to this question is derived from the data collected for tasks 2 and 3. In these tasks, developers had the aid of preemptive conflict detection to alert them of emerging conflicts in real time.

Beginners. We observed changes in how participants communicated compared to before they had access to preemptive conflict notification. The participants of R3 and R6 initiated communication earlier, when conflicts started to emerge, rather than at check-in. In the last task of R6, one of the participants even attempted to start communicating as soon as the task started. Again in R6, we observed a slight change in the manner that participants coordinated. Rather than merely informing the other that there were new changes, they exchanged messages that actually explained what changes they had performed. However, we did not observe any change in behavior from participants in R1. The participants did not initiate communication before check-in time, probably because they did not understand the concept of preemptive conflict detection.

Participants' behavior regarding how to perform merges with the new information also varied between runs. In R1 there were no significant changes in behavior compared to before emerging conflict information was available, while in R3 and R6 a few changes were observed. Participants in R3 decided to assign the task of conflict resolution to the most experienced developer with the goal of speeding up task completion, although he also struggled to perform the merges despite his experience. In R6 we observed a gradual change in the behavior of the participant who resolved the conflicts. Initially, he was aware of existing conflicts, but did not know how to benefit from this knowledge, and took the same steps to merge code as he had in the first task, which introduced compilation errors. Then, he adopted the following strategy to successfully avoid breaking the code when merging: he copied his code, overrode and updated the changed class, and pasted the code back into the class. This merge strategy is unusual to those who use SVN through Eclipse, but very similar to what would be expected from someone using SVN through the command line.

Advanced. We observed a change in both communication and merge behavior in all three runs with advanced users. In three instances participants started to communicate immediately when conflicts emerged, earlier than they had previously. For coordinating the merge itself, they had different strategies. In R2 participants decided to let the most experienced developer deal with merging, with the other user explaining the changes they had made in detail. They also split the check-ins into smaller chunks to reduce the complexity of the merge. The participants of R4 coordinated to decide who would change first and who would merge; however, they did not give preference to one person to merge code. In R5 the participants also split the check-ins into smaller sections, and again did not assign a specific user to handle the merge. In general, participants had no difficulties in merging the code that they had prior knowledge was conflicting.

Discussion. We observed several changes in developer behavior, with some commonalities among the runs. Excepting R1, participants started to communicate when conflicts emerged rather than initiating communication at check-in time. Some of them also intensified the explanation of their changes to their counterparts, helping them to better understand what needed to be merged.

Analyzing the changes in coordination, we observed that participants adopted two new strategies: (1) to let the most experienced developer deal with merging, with the goal of speeding

up the completion of the task; (2) to split the check-ins into smaller chunks to reduce the complexity of the merge.

Analyzing the change in coordination strategies according to each group of participants, we observed that the beginners who adopted the first strategy still had problems with merging, because the most experienced participants, who were responsible for the merge, still struggled to resolve conflicts. One of the participants confirms he had difficulties: *"In other kinds of situations it's tricky, because if we are really working on the same piece of code and changing stuff, that is messy because you have to decide whether your version is correct on the other one is correct."* (P6). The participants who only heightened communication and intensified their explanations of new changes demonstrated a change in merge strategy as well, finishing the last task more efficiently than in the first two tasks.

In the case of advanced developers, all intensified their coordination by discussing who should check in first and who should be responsible for merging. In two runs, they also decided to split the check-ins into smaller ones. The increase of their awareness of existing conflicts along with their increased coordination helped them perform the merge more successfully, which the participants themselves recognized: *"I think it did help, because we started communicating before actually committing all the code, and before doing other changes in other methods we started communicating when we detected conflicts. So it helped to see and synchronize what things we were adding and committing the conflicts first to first resolve all conflicts and them continue."* (P9). The participants still used the same merge technique as they had in the first task: to synchronize the code, inspect the conflicting classes to understand their differences, merge the code, and check it in.

Summary. Our observations of changes in the developers' behavior upon exposure to notifications of emerging conflicts can be summarized as follows:

Developers started to communicate earlier and had more meaningful communication. With the exception of one run, all developers started to communicate before check-in time. Most of them also put more effort into explaining to their counterpart the changes they had introduced.

Developers coordinated more effectively. The developers in different runs introduced two distinct new coordination strategies: to discuss and determine who should check in first and who should merge (giving preference to the most experienced developer for the task of merging); and to split the check-ins into smaller chunks to reduce the complexity of the merge.

Beginners' behavior changed more slowly. Beginner-level developers had some difficulty understanding the concept of preemptive conflict detection. Consequently, their behavior changed more slowly. In one case this gradual change of behavior led the participants to successfully deal with merging.

Advanced developers succeeded in merging code. Unlike what happened in task 1, developers managed to merge code successfully in tasks 2 and 3. We attribute this success to a higher level of awareness and improved coordination.

6.6.3 RQ3: How do developers perceive different approaches to deliver information on emerging conflicts?

The goal of preemptive conflict detection is to raise developers' awareness about changes that others have made which might impact their own work [Sarm 08]. This information, however, must be delivered in a non-intrusive way to avoid diverting the developers' attention from coding.

This study's secondary goal is to understand which views developers prefer to use to receive information of emerging conflicts. We have exposed developers to two different approaches: a view that shows conflicts as a list of items, with most of the information delivered as text; and a view that shows a graph of classes that can be easily reduced only the classes that the developer opens, which delivers conflict information visually by changing the color of the graph node. During the interview we asked the participants' opinions on the views and also sought suggestions for other ways this information might be visualized.

Participants' opinions. In the following we present a summary of the participants' preferences and comments about the two views.

One participant of R1 preferred the graph, because he thought the color metaphor on the graph nodes (representing classes) was a good way to visualize the information. The other participant preferred the list, though it must be noted that he did not have a chance to use the graph.

One of the participants of R2 preferred the graph view because he was able to use it in place of the package explorer (which he does not like) to keep track of the classes he had recently opened. The participant's choice of preferred view is therefore not based on how well it displays emerging conflict information in comparison with the list view, but rather in its potential to replace the package explorer metaphor. The other participant preferred the list because of his aversion to graphs in general. Hence, both participants' preferences were based on reasons other than the view's actual visual presentation of the conflicts.

Both participants of R3 preferred the list metaphor because one can identify the severity and the location (class and method) of a conflict just by looking at the information on the list, whereas with the graph a user is required to hover over a node to see the location. In short, the participants preferred the metaphor that provides for less effort to get the most information in a shorter period.

One of the participants of R4 worked more with the graph view, but his preferences could not be recorded because he could not stay for the interview. The other participant preferred the graph, because it enabled him to focus on the classes he is working on, meaning that he will not see warnings of conflicts in classes that he was not working on by displaying only the classes that he had opened, though the consequences of using this option meant that he could not see warnings for conflicts in other classes. He also preferred the visual layer added to the graph when there was an emerging conflict, which was easy for him to spot. He thought that the list presented too much information.

Both participants of R5 preferred the list for this assignment, but thought that the graph could be useful in other situations. They thought that the list was more advantageous because it was straightforward; it shows only the classes that are actually in conflict. However, they thought the graph might help demonstrate when dependencies are in conflict, because the edges are markers of call dependency (or inheritance). They also thought the graph might be more useful when changing larger files. One of the participants reported that the graph was less intuitive, and that they had difficulties realizing that additional information about the method where the conflict

originated could be accessed by hovering over a node. This demonstrates that it was easier for the participants to get used to the list view than the graph.

Both participants from R6 preferred the list, because it shows all conflicts and details in a compact manner. They presented the concern that the graph might grow to a huge size if a system is large, and that it would become difficult to see the conflict warnings.

Summary. The majority of the participants preferred the list (eight of them), with their strongest reasoning being that the list presents the important information in a direct and condensed manner. Thus, developers only need to quickly look at the view to get all the information they need about an emerging conflict. To get such detail about emerging conflicts from the graph view, a user needs to hover over a node.

Only three of the participants preferred the graph, each presenting different rationale. One preferred it because he could just replace the package explorer with it. Another one preferred the graph because he thought the color metaphor in the nodes of the graph (representing the classes) is a good way to visualize the information. The last participant preferred it because it enabled him to focus on the classes he was working on, though this meant that he would not see warnings of conflicts in classes that he was not currently working on. He also preferred the visual layer added on the graph when there is an emerging conflict, which is easy to spot. This participant also argued that the list shows too much information at once.

Suggestions of other ways to visualize emerging conflicts. Participants were asked to suggest other ways to visualize information on emerging conflicts. Surprisingly, many of them had the same suggestion: to show this information directly in the Java editor.

The participants of R1 thought it would be beneficial to show the information on the editor, and suggested marking conflicts by either changing the background color or adding markers on the ruler. The participants of R2 and R3 also suggested marking conflicts in the Java editor, but specified that this should be done through a layer highlighting the code, which could be disabled if it became disturbing for the developer. One participant from R4 suggested the addition of cues to the package explorer, so that it might be used in combination with Mylyn to focus on code that is currently being changed right now and avoid the use of an extra view on Eclipse. The participants of R5 and R6 suggested annotations directly on the Java editor to avoid the use of a separate view, and noted that these annotations could be made on the editor as a marker on the ruler or as a background color.

Even though adding information directly into the Java editor is a more intrusive approach, developers showed a strong preference to have it there. Having an enable/disable option should be enough to resolve the problem if a developer feels disturbed by the notifications. However, showing information on emerging conflicts only at the class level in the Java editor would prevent developers from receiving overall information on the system. Hence, we think that it is also helpful to provide the list and graph views as alternative ways to look at conflict notifications.

6.7 Limitations and Future Work

This investigation on the effects of the presence of preemptive conflict detection on the behavior of developers showed evidence of its ability to enhance the coordination and communication of the team. However, for this approach to become adoptable by developers, a number of limitations have to be addressed:

Choice of pairwise conflict detection algorithm. The algorithm we implemented detects conflicts between pairs of developers. This means that, if three developers were to perform conflicting changes to the same artifacts, the algorithm would generate three alerts, and each developer would receive two alerts about the conflicts existing on a single artifact. In addition, these alerts would not correlate with one another, leaving it up to the developers to deduce that they refer to the same code and that all three of them are involved, not just two. While this algorithm is less complex and satisfies most cases (rarely there will be a point of conflict involving more than two developers), other solutions that detect conflicts among multiple developers should be evaluated. One solution would be to add a step on the pairwise algorithm to check for interdependent conflicts.

Scalability of the conflict detection algorithm. Our algorithm is currently triggered at every new arrival of a change to a class or method, and it traverses the AST of each developer to check for new conflicts. The complexity of this algorithm is $O(de)$, where d is the number of developers and e is the number of nodes (entities) in each tree. Since the trees are neither sorted nor balanced, we were not able to optimize the tree traversal, consequently giving us an algorithm with polynomial growth. Considering the rate of the arrival of new changes, which is tied to the number of active developers, this algorithm may show scalability issues when the number of developers and source code entities grows. To assess how scalable this algorithm is, we need to execute performance tests that simulate larger teams and systems than those we have tested in this study.

Improvements on how to deliver information about emerging conflicts. We evaluated how developers perceived two of three possible ways to deliver information on emerging conflicts. The participants' feedback revealed drawbacks in both views that need to be addressed. The graph shows a conflict's details in a pop-up that appears when a developer hovers the mouse over a colored node. A major problem with the pop-up is that it requires too much text to describe the conflict details, which is hard to read before the pop-up disappears. Given that the node represents a class, multiple conflicts in different methods of the class might appear, making the information in the pop-up even more confusing. We need to rethink how to display the details of conflicts for each node. A common problem participants had with both views is that they do not offer a filter. With the number of conflicts that can be produced over time, the number shown can get overwhelming. The views should allow developers to customize what they want to see by filtering out notifications that they judge to be irrelevant.

Conflict resolution support. While detecting conflicts at early stage can prevent them from getting overly complex, a further step that would help developers deal with merge would be to offer support for the resolution of conflicts. Researchers have investigated how to manage change-base merging [Dig 07b], and real-time merging [Guim 10]: the two aspects we need to consider to propose a conflict resolution engine. We believe that the most suitable approach would be to create a stepwise, semi-automatic conflict resolution engine that automatically resolves trivial conflicts, but waits for a developer's decision on complex ones.

It is important to note that this user study also has its limitations. Such a study in a laboratory setting may have been influenced by factors that were not possible to control or predict. For instance, after the first task, the participants were coding under the expectation that conflict would occur, which may have influenced their behavior. Each run additionally had unexpected

issues (*e.g.* problems with the video recorder, network failure, participant not following the instructions) that might also have influenced the results. In an attempt to counterbalance the effect of these unexpected issues upon our analysis, we took them into account in our discussion.

This study is an initial step toward understanding the role preemptive conflict detection can play on collaborative development. Further studies, such as a longitudinal case study, should be conducted to evaluate the usability of this approach and factors that influence its adoption in the long term.

6.8 Summary

In recent years, there has been a significant effort to increase the awareness levels of distributed teams by supporting coordination between multiple developers working in parallel on the same code base [Bieh 07; Guim 10; Hegd 08; Sarm 07; Sarm 08; Brun 11; Silv 06; Proe 10]. However, only a limited number of studies [Bieh 07; Dewa 07; Sarm 08] have been conducted to investigate whether the adoption of tools to promote workspace awareness is truly beneficial to developers. The initial findings of these studies suggest that, when preemptive conflict detection is introduced, the frequency of communication increases, overlapped work is reduced, and the rate of detection and resolution of conflicts increases. However, some fundamental questions concerning the concept of preemptive conflict detection remained, *e.g.* "Were the changes observed in these studies beneficial to developers?", "Did the developers' strategies to deal with merging change?", "Is the information being delivered disturbing?", "Would developers prefer to receive this information a different way?".

In this chapter, we have investigated some of these open questions by focusing on understanding whether the behavior of developers changes when they are exposed to preemptive conflict detection, and whether this change is beneficial to them. To perform the investigation, we first devised different ways to display information about emerging conflicts on the IDE, with the *list view* being the adaptation of previous works [Hegd 08; Sarm 08], and the *graph view* being inspired by tools that show emerging design [Silv 06; Proe 10; Serv 10]. We then designed and conducted a qualitative user study in a laboratory setting with six pairs of developers who had to resolve a number of tasks in collaboration. We collected data from questionnaires, interviews, recorded observations (screencasts), and documentation, and analyzed it in a iterative process.

We reported on the behavior of developers when they were required to merge code without the help of preemptive conflict detection. Our findings conform with those of previous studies: developers, even experienced ones, struggle with merging [Grin 96]; and some developers rush to be the first to check in so that they may avoid merging [Souz 03; Grin 96]. We observed a number of additional important findings: developers only start to communicate, and consequently to coordinate, after the first (attempted) check-in; and in most cases the communication remains shallow, with only a few developers putting in the effort to explain the changes they had performed to their counterpart. Experienced developers shared the same conflict resolution strategies, while beginners had different, often naïve, behavior.

We found significant change in the behavior of developers after the introduction of preemptive conflict detection. Participants started to communicate earlier, usually right after the first information on emerging conflicts appeared. This earlier and more meaningful communication gave them the opportunity to coordinate better by adopting more involved merging strategies that they would not have usually attempted. We observed two new strategies: to discuss and decide upfront who would merge the conflicting code, and to break the check-ins into smaller

chunks to reduce the complexity of the merges.

This study has shown that the changes in the behavior of developers when exposed to preemptive conflict detection are beneficial in terms of having more effective communication and coordination, as well as in terms of resolving conflicts with a higher success rate. In addition, we believe that different ways to visualize this information are complementary to one another, because developers have different preferences, and the fact that they can choose among different options might help them in the adoption process.

In the context of our thesis, this chapter describes and evaluates a more specific application of the same data used in the previous chapter. The information provided about the detection of emerging conflicts proved to effectively help developers to better communicate and coordinate, consequently helping them to better collaborate.

In these two chapters, we proposed applications that make immediate use of the fine-grained changes collected through our CCBSE approach to promote workspace awareness. In the following chapters we address the second goal of our work, which is to help developers acquire information related to the evolution of the system by leveraging the detailed change history that Syde stores.

Chapter 7

Refining Code Expertise Measurement with Fine-grained Changes

7.1 Introduction

In the previous chapters, we explored the use of applications for the real-time change information collected by Syde. Now, we focus on exploring how the rich change history that Syde collects can be utilized for the benefit of developers. Our aim is, as always, to help developers collaborate, share knowledge, and better understand the evolution of the system and its intrinsic dynamics. In this chapter we focus on refining code expertise measurements to help developers locate software artifact experts.

The nature of the information a software repository contains will determine what can be inferred from it [Robb 07a]. File- and content-based SCM systems store snapshots that represent the system's state at points in time, rather than representing a continuous evolution of the changes made to the system to bring it from one state to another. We believe that code expertise measurements derived from file- and content-based SCM systems are subject to the loss of information that comes with these underlying models.

We describe how we use Syde's change history to better understand developers' dynamics, and with this information proceed to refine existing measurements of code expertise and define a new one. We further improve our new measurement by accounting for imperfect memory and integrating the notion of forgetting into our model of expertise: a developer who changed a file early on may have less actual knowledge of it than another developer who changed it more recently, even if it was changed to a lesser degree.

Our new expertise measurement has been integrated into Scamp's Bucket View (see Section 5.3.2 (p.64)). This feature helps developers search for assistance when striving to understand a piece of code. In such cases, the expertise recommender visually displays the developers who are knowledgeable about the code artifact, giving a rank of their expertise according to the amount of change they have performed.

To conduct this study, we used the history of three systems, including a commercial one, recorded with Syde.

Structure of the chapter. In Section 7.2 (p.126) we review related work. In Section 7.3 (p.127) we describe the notion developers' expertise, and the effect of time on their ability to recall information. In Section 7.4 (p.132), we describe a visualization we use as support for interpreting the expertise data: the Expertise Map. In Section 7.5 (p.136) we then evaluate the proposed measurement in a case study with three software systems, before discussing threats to the validity of our study in Section 7.6 (p.145). In Section 7.7 (p.147) we discuss the limitations of this approach and future directions that might be taken to improve it. Finally, we conclude in Section 7.8 (p.148).

7.2 Related Work

There have been several works on determining which developer has the most expertise on a given part of a large software system. The rationale behind these approaches is that since no one can be knowledgeable about the entire system, one can instead identify the right people to contact to get more information about different parts of a system – the experts. To determine developers' levels of expertise, researchers have gathered information from several data sources.

Several approaches use SCM data (authorship) to compute expertise and ownership. They assume that people gain expertise on a part of the system when they change its implementation. McDonald and Ackerman used authorship to determine expertise levels. They also included what technical support data was available to them in their particular case study [McDo 00]. Mockus and Herbsleb in contrast, used only authorship as the data source of their Expertise Browser [Mock 02]. Gîrba *et al.* [Girb 05b] focused more specifically on ownership, determining that the developer with the most expertise on a file is the file's owner. They also used SCM data to compute ownership. Rahman and Devanbu [Rahm 11] computed code expertise on Git repositories and related it to defect introduction. Their findings suggest that experts on a file are less likely to introduce defects than developers who rarely perform changes to it, and that specialized experience is more important than general experience.

A few approaches consider not just the SCM history, but also any code that the developer consults during their work. Schuler and Zimmermann [Schu 08] suggest an approach for analyzing: (1) implementation expertise, accumulated by changing code; and (2) usage expertise, accumulated by using code (*e.g.* calling a method).They were able to create expertise profiles that included data about what APIs a developer might be an expert in according to their usage of those APIs. Fritz *et al.* [Frit 10b] combine authorship with interaction data to propose a Degree-of-Knowledge model for capturing source code familiarity. While authorship data for this model comes from the SCM repository, interaction is measured with the Degree-of-Interest model derived from Mylyn data [Kers 06].

Other approaches use different data sources. Anvik and Murphy used bug archive data to determine implementation expertise, and found that it can serve to replace SCM data in cases where the latter is not accurate [Anvi 07]. Matter *et al.* determined developers' expertise based on the vocabulary they use. They used that expertise information to assign bugs to developers [Matt 09]. Ma *et al.* introduced the concept of usage expertise, which takes the expertise of people who use a given piece of code into account, as opposed to that of people who have implemented that code. [Ma 09].

Our approach differs from the previous ones in the granularity of the authorship data it utilizes. Most of the previous approaches use authorship extracted from SCM commits, mapping one committed file to an author. In contrast, we map one change to a file at the developer's workspace to an author. This mapping is more detailed than the type that uses SCM data and is

almost as detailed as interaction information. The difference between the latter approach and ours is that interaction information concerns code that has been navigated but not changed, while our approach considers only code that was actually edited.

7.3 Code Expertise

Code expertise is a measurement that quantifies the amount of knowledge each developer has. By measuring which developers have accumulated the most knowledge on different artifacts, it is capable of indicating the developer that owns – or has the most expertise with – each artifact in a software system. This notion of code ownership is important in large projects, where developers are generally not familiar with each artifact of the system. Code expertise can be used to answer questions such as "who should fix this bug" [Anvi 06] or "who should I ask about artifact XY" [Zimm 05].

We present the measurements with which we compute code expertise. Our general assumption is that whoever performs the greatest number of changes on a file is the most knowledgeable about it. We use distinct measurements of code expertise based on three different repository sources: CVS, SVN, and Syde. For CVS and SVN, we adopt the measurement previously introduced by Gîrba *et al.* [Girb 05b], whereas for Syde, we present a lightweight approach to compute code expertise by mainly using historical information contained in the Syde change logs.

7.3.1 Measuring Code Expertise with CVS/SVN Logs

The expertise definition based on CVS and SVN logs exclusively uses the information contained in each respective log, with this including the file name, revision, author, and number of lines added and deleted. Since SVN logs do not contain the number of lines added and deleted for each revision of a file, we implemented a parser to extract this information by comparing every subsequent revision of a file in the repository.

According to Gîrba *et al.* [Girb 05b], a developer owns a line of code in a file if he was the one who committed that line. The overall expert of a file is therefore the one who owns the largest percentage of it. To compute the expertise, we need to first estimate the size of a file. Because we only use information contained in the log history[1], we know the number of lines added or deleted but not the initial size of the file, and must therefore compute it. Given a file f, let f_n be a revision of f, σ_{f_n} be the author of that revision, a_{f_n} be the number of lines added, and r_{f_n} be the number of lines removed. The size of a file revision s_{f_n} is given by:

$s'_{f_0} = 0$
$s'_{f_n} = s'_{f_{n-1}} + a_{f_n} - r_{f_n}$
$s_{f_0} = |min\{s'_x\}|$, let s'_x be the set of revision sizes from files f_0 to f_n
$s_{f_n} = s_{f_{n-1}} + a_{f_n} - r_{f_n}$

[1] In our case study, we examine an industrial system that uses CVS and two academic systems that use SVN. For the first, we only had access to the logs, not to the source code. We thus decided to consistently apply the same approach to all three projects, by estimating the file size for each of them.

To exemplify the size estimation of a file, suppose the following sequence of changes:

f_1: 8 lines added, 3 lines deleted;

f_2: 7 lines deleted.

We apply the given formulae to estimate the sizes of the files:

$$s'_{f_0} = 0$$
$$s'_{f_1} = 0 + 8 - 3 = 5$$
$$s'_{f_2} = 5 + 0 - 7 = -2$$

Since there cannot be more lines deleted than there were added in total, the above values must be adjusted.

$$s_{f_0} = |min\{-2, 5\}| = 2$$
$$s_{f_1} = 7$$
$$s_{f_2} = 0$$

Given the size s_{f_n} of a file revision, the percentage $exp^{\sigma}_{f_n}$ of lines in a revision owned by a developer σ is given by:

$$exp^{\sigma}_{f_0} = \begin{cases} 1 & \text{if } \sigma = \sigma_{f_0} \\ 0 & \text{else} \end{cases}$$

$$exp^{\sigma}_{f_n} = exp^{\sigma}_{f_{n-1}} \frac{s_{f_n} - a_{f_n}}{s_{f_n}} + \begin{cases} \frac{a_{f_n}}{s_{f_n}} & \text{if } \sigma = \sigma_{f_n} \\ 0 & \text{else} \end{cases}$$

Knowing the percentage $exp^{\sigma}_{f_n}$ of lines owned by each developer, the greatest expert on a file revision can be determined to be the developer who owns the greatest percentage.

This measurement relies on the assumption that the number of lines of a file owned by a developer reflects the amount of effort he spent writing those lines. However, the development of a code artifact is not a linear action that can be summed up simply as the number of lines added to a file. Developers do and undo changes, try alternatives, refactor the code – which may reduce the size of the file –, *etc.* The knowledge they retain from an artifact depends more on how much effort they expend in implementing it, than it does on the final result. This technique is therefore most effective when developers check their files in frequently. However, if within a team there are some developers who check in their changes frequently and others who work for long periods before checking in, this technique is prone to discrepancies.

7.3.2 Measuring Code Expertise with Syde Logs

We use the history logs provided by Syde to measure code expertise, therefore basing our definition of expertise on every small change that is performed on a system. *Every small change* means every change performed between two compilation actions. One might argue that the definition of expertise using Syde logs, analogous to what happens with CVS and SVN logs, is also biased by the frequency with which developers save their code. However, the way Eclipse works drives developers to maintain their code with no compilation errors, and hence to save and build files often. Eclipse constantly points out where compilation errors are, which discourages developers

from running or testing code with errors, and in general incites them to fix them as soon as possible.

Given Syde logs, let f be a file, f_n a version of this file, and σ_{f_n} be the author of that version. Let the number $exp^{\sigma}_{f_n}$ of changes performed by a developer σ be given by:

$$exp^{\sigma}_{f_0} = \begin{cases} 1 & \text{if } \sigma = \sigma_{f_0} \\ 0 & \text{else} \end{cases}$$

$$exp^{\sigma}_{f_n} = exp^{\sigma}_{f_{n-1}} + \begin{cases} 1 & \text{if } \sigma = \sigma_{f_n} \\ 0 & \text{else} \end{cases}$$

The greatest expert exp_{f_n} of a file at a certain version is the one who has accumulated the largest number of changes from the creation of the file until the date of the considered version:

$exp_{f_n} = max\{exp^{\sigma_1}_{f_n}, exp^{\sigma_2}_{f_n}, ..., exp^{\sigma_m}_{f_n}\}$, where m is the total number of developers.

To exemplify how to compute expertise on a file with Syde's log, we show the change history of a hypothetic file *Foo* in Figure 7.1.

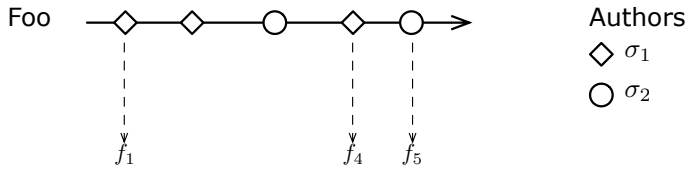


Figure 7.1. Change history of file Foo. This file is currently in version 5 and two developers have changed it.

In this example, we want to find who knows more about *Foo* by the time version f_5 is created. The number of changes performed by each author at this point is:

$exp^{\sigma_1} = 3$
$exp^{\sigma_2} = 2$

The greater number exp_{f_5} of changes performed in the file f at revision f_5 is:

$exp_{f_5} = max\{exp^{\sigma_1}, exp^{\sigma_2}\} = exp^{\sigma_1}$

Hence, the greatest expert of file f at revision f_5 is developer σ_1.

This measurement assumes that developers accumulate knowledge about a class or file but never forget it, even though months or years may have passed. This is a rather naïve assumption, since the content of files might change over time. Suppose a developer creates a file and introduces 10 lines of code with 30 Syde changes. Later, a second developer edits the same file and, with 15 Syde changes, completely changes the 10 lines of code. The current expertise measurement based on Syde changes would still consider the first developer as the most knowledgeable, whereas the measurement based on CVS/SVN commits would consider the second as most knowledgeable

because he currently owns the greater percentage of lines (assuming that the total number of lines remains constant, the second developer is the owner of 100% of the lines).

In addition, developers undergo a natural process of forgetting the content and functionality of a class over time, even though these items may not have changed. To address these issues, we add the concept of forgetting to the code expertise measurement based on Syde's logs. Although the measurement based on CVS/SVN logs also ignores this process, we focus exclusively on investigating the effect of forgetting on our expertise measurement, based on Syde's logs.

The Forgetting Effect on Code Expertise. Forgetting is a natural process in which old memories are unable to be recalled from a human's memory. It has been studied extensively by psychologists, since the pioneering work of Ebbinghaus [Ebbi 13]. The curve that commonly describes forgetting is expressed as $R = e^{-t/s}$, where R is memory retention, s is the relative strength of memory, and t is time. Although there has been continuous discussion about whether forgetting is best described by a power or exponential function [Wixt 07; Murr 09; Carp 08], the differences represented by each curve can be considered minimal for the context of expertise measurement. Thus, we consider the exponential formula introduced above.

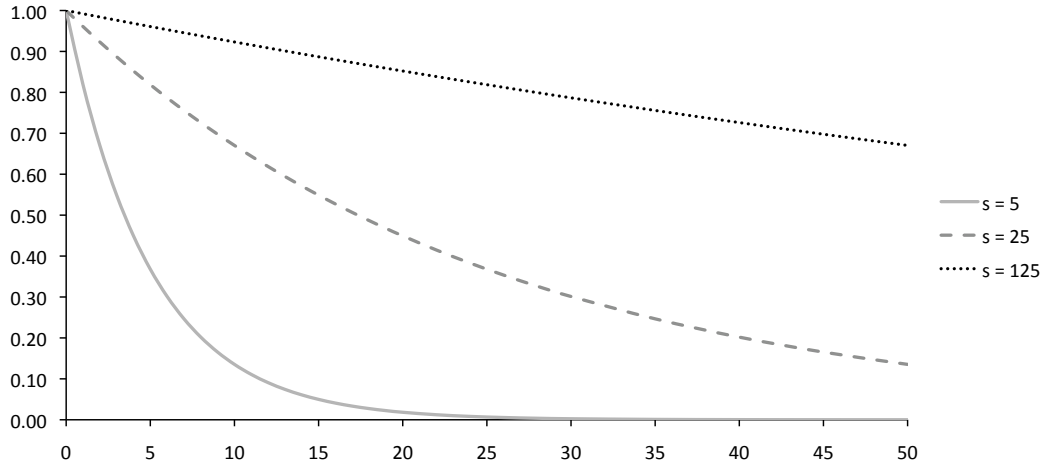


Figure 7.2. Forgetting function $R = e^{-t/s}$, where R is memory retention, s is the relative strength of memory, and t is time. The higher is the value of s, the more likely the person will remember an event for a longer period.

Figure 7.2 shows the plot of the forgetting function for three different values of s (5, 25, and 125). In the context of code expertise, our units of time are days. The s parameter reflects how long a person might remember the contents of a certain file. The smaller s is, the weaker the memory. For instance, for $s = 5$, the probability is only 30% that a person remembers the contents of a file edited 6 days ago (see Table 7.4 for other figures). This situation could be true in a scenario where this person is developing multiple systems at the same time, and only makes small changes, which are not sufficient for him to absorb concrete knowledge of each class. The parameter s is therefore influenced by a number of factors that are specific to each project, *e.g.* the complexity of the project or feature, the level of experience of each individual, the accumulated experience of each individual in the context of the project, the total number

of developers, *etc.* Hence, it is not our goal to determine an ideal s value for all projects, but to explain how it can be adjusted according to each scenario.

We include the concept of forgetting in the code expertise measurement by adding the time factor into the calculation for the expert $exp^{\sigma}_{f_n}$ of a change. In the previous formula, a value of 1 was given to a developer for each change he made. With the forgetting measurement, we use the formula $R = e^{-t/s}$ to weight this value, which now has a range of $(0, 1]$.

To formalize the new measurement, for each version f_n of a file f, we consider the time t_{f_n} of the creation of version f_n. Given a point in time t, the argument t of the forgetting function $R = e^{-t/s}$ is the amount of time elapsed between the creation of version f_n and the current time t:

$$\Delta t_{f_n} = t - t_{f_n}$$

Hence, given a point in time t, the new expertise measurement is computed as follows:

$$\Delta t_{f_0} = t - t_{f_0}$$
$$exp^{\sigma}_{f_0} = \begin{cases} e^{\frac{-\Delta t_{f_0}}{s}} & \text{if } \sigma = \sigma_{f_0} \\ 0 & \text{else} \end{cases}$$

$$\Delta t_{f_1} = t - t_{f_1}$$
$$exp^{\sigma}_{f_1} = exp^{\sigma}_{f_0} + \begin{cases} e^{\frac{-\Delta t_{f_1}}{s}} & \text{if } \sigma = \sigma_{f_1} \\ 0 & \text{else} \end{cases}$$

$$\Delta t_{f_n} = t - t_{f_n} = 0$$
$$exp^{\sigma}_{f_n} = exp^{\sigma}_{f_{n-1}} + \begin{cases} e^{\frac{-\Delta t_{f_n}}{s}} & \text{if } \sigma = \sigma_{f_n} \\ 0 & \text{else} \end{cases}$$

The greatest expert exp_{f_n} of a certain version of a file is the person who has accumulated the greatest value of weighted knowledge from the creation of the file until the time t_{f_n} of the considered version:

$exp_{f_n} = max\{exp^{\sigma_1}_{f_n}, exp^{\sigma_2}_{f_n}, ..., exp^{\sigma_m}_{f_n}\}$, where m is the total number of developers.

Recalling the example from Figure 7.1, we added the notion of time in Figure 7.3 to recompute the expertise considering the forgetting effect. Analogous to the previous example, we want to find the person who is most knowledgeable about *Foo* at the time t_{f_5} of the creation of version f_5.

Hence, the value of weighted knowledge accumulated by each developer, given the strength of memory $s = 2$, is:

Let $t_{f_1} = 1, t_{f_2} = 2, t_{f_3} = 3, t_{f_4} = 4, t_{f_5} = 5$,

$$exp^{\sigma_1}_{f_5} = exp^{\sigma_1}_{f_4} + 0 = e^{\frac{-\Delta t_{f_1}}{s}} + e^{\frac{-\Delta t_{f_2}}{s}} + 0 + e^{\frac{-\Delta t_{f_4}}{s}} + 0 = e^{\frac{-4}{2}} + e^{\frac{-3}{2}} + 0 + e^{\frac{-1}{2}} + 0 \simeq 0.96$$

$$exp^{\sigma_2}_{f_5} = exp^{\sigma_2}_{f_4} + 1 = 0 + 0 + e^{\frac{-\Delta t_{f_3}}{s}} + 0 + e^{0} = 0 + 0 + e^{\frac{-2}{2}} + 0 + 1 \simeq 1.37$$

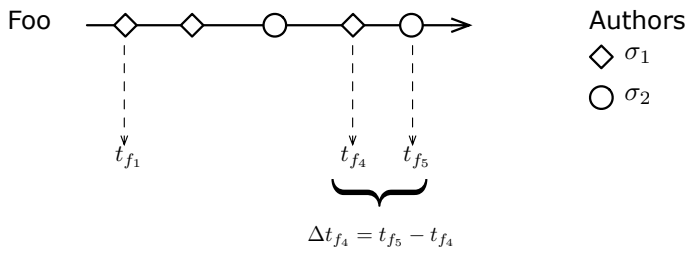


Figure 7.3. Change history of file Foo with additional notion of time

The greater value exp_{f_5} of knowledge accumulation for file f at revision f_5 is:
$exp_{f_5} = max\{exp^{\sigma_1}, exp^{\sigma_2}\} = exp^{\sigma_2}$

Even though developer σ_1 has made the greater number of changes, if we consider the forgetting effect on knowledge retention and the recency of changes, the most knowledgeable about file f at revision f_5 is developer σ_2.

With this new measurement of code expertise, it is possible to adjust memory strength s to prioritize recent changes over old ones. The range of s most suitable for a project depends on a number of factors intrinsically related to its characteristics. In Section 7.5 (p.136) we apply and discuss the influence of the forgetting effect on ownership maps of three projects.

7.4 Expertise Maps

To visually assess the differences in expertise between the measurement using versioning system changes and the measurement using Syde changes, we display the data using expertise maps.

The Expertise Map visualization was first introduced by Gîrba *et al.* [Girb 05b], who used the term *ownership map*, with the aim of characterizing developers' behavioral patterns throughout the life-cycle of a software system. Gîrba's expertise map was inspired by the visualization proposed by Rysselberghe and Demeyer [Ryss 04], where the horizontal axis represented each file of the system, and the vertical axis represented the time that a change was made.

In Gîrba's original expertise map, each line represents a file in the system and time is represented on the horizontal axis. Every change to a file is shown as a colored disc. The color of the disc is representative of the developer who made that change. Finally, the line of the file is colored to represent the most knowledgeable author. A file can have multiple experts throughout its history.

We use three distinct expertise maps:

CVS/SVN Expertise Map presents the expertise of files according to CVS/SVN commits;

Syde Expertise Map shows the expertise of each file according to Syde changes;

Delta Map illustrates the differences in expertise classification of the above two maps.

In the following, we detail each map, and explain how we order the files in the maps.

7.4.1 CVS/SVN Expertise Map

In the CVS/SVN expertise map, each rectangle represents a commit[2]. Each developer is represented by a unique color. These colors are used to indicate the authors of commits by coloring the rectangles, and also indicate the expert of the file at a certain period of time by coloring the corresponding part of the line.

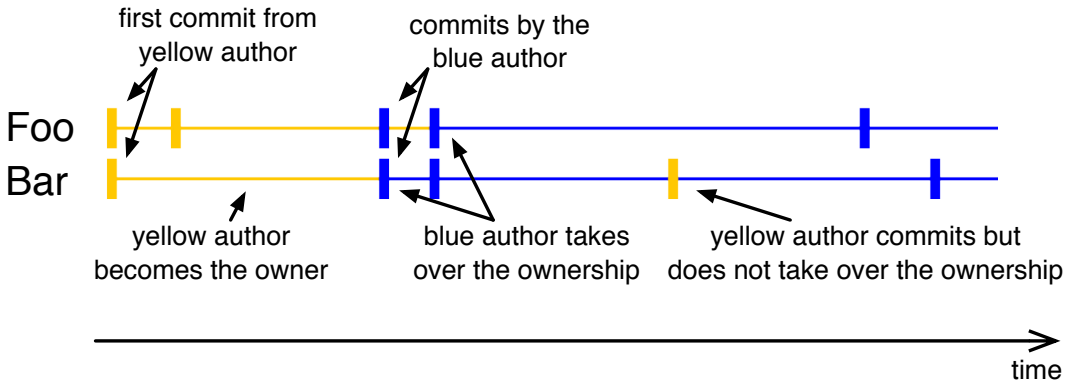


Figure 7.4. Example of CVS/SVN expertise map

Figure 7.4 illustrates an example of this map. Recall that the measurement used to compute expertise for CVS/SVN is based on the number of lines added and deleted from each version, and evenly takes into account changes made throughout the history of the file.

7.4.2 Syde Expertise Map

In the Syde expertise map, a colored disc represents a Syde-level change, with the color of the disc indicating the author of the change. The line of the file is painted with the color of its expert at the relevant period of time.

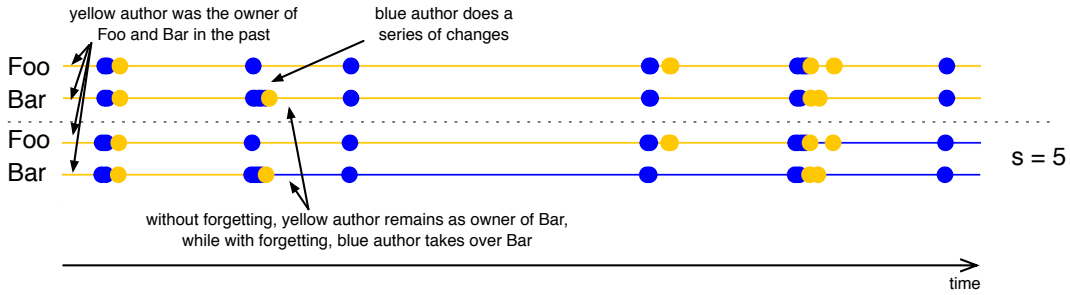


Figure 7.5. Example of Syde expertise map

Figure 7.5 shows two Syde expertise maps. The first uses the original Syde expertise measurement, while the second integrates the forgetting notion, using a value of $s = 5$ to model the

[2]We use rectangles instead of circles for SCM logs so that one can differentiate between SCM changes and Syde changes when both are overlapped on the same map.

strength of memory. It is easy to spot the differences in expertise between the two maps. Consider file *Bar*, which experiences a series of changes from the blue author early on. In the first map, the yellow author remains the owner of *Bar*, but in the second, the blue author takes over. In Section Section 7.5 we discuss the differences in expertise according to how fast a developer might forget the contents of a file. When forgetting is taken into consideration, the developer who is designated as the file's expert can change at any moment, not only when another author implements a change.

7.4.3 Delta Expertise Map

The Delta expertise map is a combination of the previous two, with rectangles representing CVS/SVN commits, and discs representing Syde changes. Discs and rectangles receive the color of the developer who authored their corresponding changes or commits. The lines are either grey, when there is no difference in expertise classification between Syde and CVS/SVN measurements, or red, when there is a difference between the measurements.

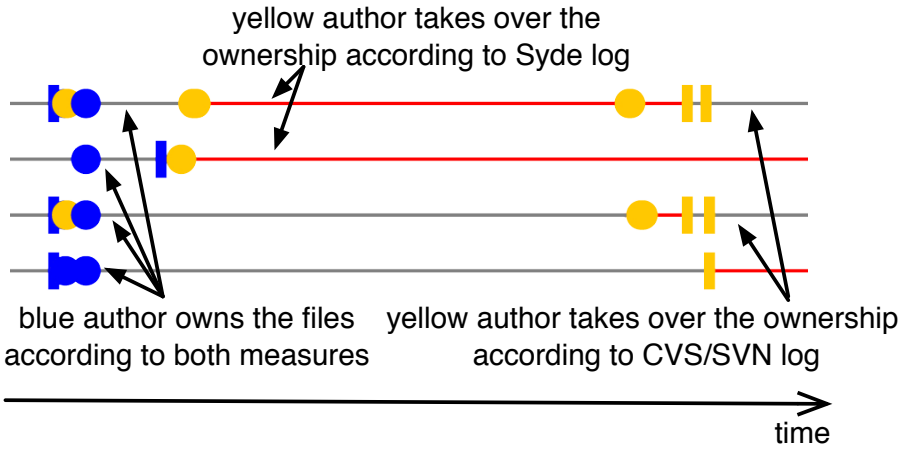


Figure 7.6. Example of delta expertise map

Figure 7.6 shows an example of the delta map. The blue author is initially the owner of the first file, but a difference between the two logs appears when the yellow author makes a Syde change and becomes the expert, before she performs a commit. Once she does this, her expertise is reflected by the CVS/SVN measurement and the difference disappears. For the second and third files, the blue author is initially the expert according to both measurements. For the second file, Yellow becomes the expert according to Syde changes after making a series of changes, but this is not reflected by the CVS/SVN measurement because the changes are not yet committed. Yellow takes over the third file, which causes a difference in classification before she commits. As seen in some of these examples, the difference in expertise classification is oftentimes caused by the instant propagation of changes in Syde against the latency in propagation caused by CVS/SVN commits.

7.4.4 Ordering of the Files

The order of the files in the expertise map has a large effect on its legibility. We order the files by similarities in their change histories, rather than using the standard alphabetical order. With this ordering, the relationships between the files stand out much more, as files changing at similar times – inherently related to one another – are grouped.

To measure the similarity between the change histories of two files, we use a variant of the Levenshtein distance [Leve 66], which measures the similarity between two sequences by counting the number of edit operations necessary to transform one into the other. There are three kinds of operations: Insertion, Deletion, and Modification of a sequence of elements. These operations are defined on abstract sequences of items and are not related to SCM operations.

In our case, we use the Syde change histories of the files as the sequences for this measurement. The Syde-based clustering of the files is also used for the CVS/SVN maps, so that the files are ordered identically in each set of maps, making comparison easier. We split the change history into a series of time intervals, and count the number of changes that affected the file during that interval. This yields a sequence of clustered changes with varying intensities, ordered by time. We tried intervals of 1, 4, 12 and 24 hours, inspecting the clusters produced in each case, and concluded that a 12-hour clustering was most effective at grouping together sets of files that were modified together (*e.g.* files that were modified together were put side by side more often with 12-hour intervals than with other values).

Each edit operation involved in the Levenshtein distance measurement is associated with a cost. To account for the different nature of our data, our definitions of the costs vary from the common definition. The Levenshtein distance is often used to measure the distance between words – where one can assume that the characters are independent –, whereas we are comparing patterns of changes with an intensity of changes in a given interval. Intuitively, the distance between any two characters is constant, but this assumption does not carry over to changes. Measuring the difference in change intensity allows us to use a more precise distance metric.

We retain the standard costs of 1 for insertion and deletion operations on items in the sequence. These operations are primarily used to switch the order of two time intervals when items with the same values are indexed nearly identically in the two sequences (*e.g.* a burst of changes on file $f1$ occurred just before a similar burst of changes on file $f2$).

We use a different definition of the cost for the modification operation. The modification operation is used to alter the value of an item in the sequence so that it corresponds to the value of the item at the same index in the second sequence (*e.g.* a burst of changes on file $f1$ at time $t1$ is slightly smaller than the burst of changes of $f2$ at the same time). We define the cost of a modification operation between two change amounts a and b as:

$$ModificationCost_{a,b} = \begin{cases} 1, & \text{if } a = 0 \text{ and } b > 0 \\ 1, & \text{if } b = 0 \text{ and } a > 0 \\ \frac{|b-a|}{10}, & \text{otherwise} \end{cases} \tag{7.1}$$

The rationale behind our choice is that moderate variations in the intensity of changes during an interval should result in a lower edit cost than larger ones. On the other hand, a transition from no changes to any number of changes is always costly.

After computing the Levenshtein distance, we use a hierarchical clustering algorithm to order the files with respect to their similarity according to this distance. All the maps we show in the remainder of the article are ordered using this scheme.

7.5 Case Studies

In this evaluation, we use the data provided by Syde to tackle the following research question:

How can Syde's history log help to characterize code expertise?

To discuss this question we analyze three projects: Speed, jArk, and Pacman.

7.5.1 Presentation of the Projects

Speed[3] is a commercial project that was under development at the software factory of CPMBraxis [CPMB 11]. This software factory was chosen because of its professional characteristics: it has a well defined production process certified by CMMI-DEV 5 and ISO 9001:2000 standards, and its projects adopt metrics, software reuse practices, and new technologies for delivering high quality products.

Pacman and jArk are group projects developed by students during the "Programming Fundamentals 2" course offered at the University of Lugano. Unlike with Speed, where we collected data for a brief period, the students used Syde for the entire duration of their projects. We therefore have a full development history for these projects.

Table 7.1 presents the data we gathered for the three projects monitored in this study.

Table 7.1. Projects studied, the time period over which each project was analyzed, the number of java files that were committed to the SCM repository, the number of files with changes captured by Syde, the number of SCM commits, and the number of Syde changes

Project	Period	Dev.	Files in SCM	Files in Syde	Commits	Syde Changes
Speed	15 days	4	14	56	26	2,429
jArk	5 weeks	2	146	172	266	23,786
Pacman	5 weeks	2	140	223	149	14,460

The number of files with Syde changes is higher than the number of files committed to CVS or SVN. We examined the source code and observed that the main cause of this disparity is situations where developers create a class and work on it for a while, but change its location before checking it into the repository.

We can immediately observe that the changes recorded by Syde are much more fine-grained than the ones recorded by CVS or SVN; the number of Syde changes collected is two orders of magnitude higher than CVS or SVN commits.

Table 7.2 shows the summary statistics of the number of Syde changes and CVS/SVN commits recorded per day.

To compute the summary, we removed only the days lacking both changes and commits (*i.e.* if there was at least one commit or one change, the day is included for both measurements). For at least 75% of the days, the number of Syde changes is two orders of magnitude higher than SVN commits for jArk and Pacman (2nd, 3rd, and 4th quartiles). For Speed, the same figure holds for at least 50% of the days (3rd and 4th quartiles).

[3] We use pseudonyms in conformation with the non-disclosure agreement.

Table 7.2. Summary statistics for Syde changes and CVS/SVN commits recorded per day

Project		Min.	1st Quartile	Median	Mean	3rd Quartile	Max.
Speed	CVS	2	4.00	6.00	6.00	8.00	10
	Syde	1	48.75	143.50	242.90	367.50	668
jArk	SVN	1	3.00	6.00	7.82	12.75	24
	Syde	1	121.00	454.00	792.90	863.80	9,242
Pacman	SVN	1	1.00	4.00	5.14	8.00	17
	Syde	5	113.00	300.09	437.90	611.00	2,981

Figure 7.7 provides an overall view of the Syde expertise maps of the three projects. Since the full maps are large, we use magnified, readable parts of them for the analyses that follow.

7.5.2 Characterizing Code Expertise with Syde

Table 7.3 presents the total number of Java files contained in Syde's log for each project and compares it to the number of files where there are differences in expertise classification (deltas) between the Syde-based and CVS/SVN-based measurements. According to Table 7.1, the number of Syde changes is two orders of magnitude higher than the number of commits. We therefore expect Syde's expertise measurement to give more fine-grained information about the identity of the current expert on a file in comparison to the measurement based on CVS/SVN. Speed has a low number of files committed – only 14 files –, which influences the low number of files with deltas. Pacman and jArk, however, have a high number of files with differences in classification, which is justified by the significantly larger number of commits and longer period of Syde usage.

Table 7.3. Comparison between total number of files and number of files with differences between Syde and CVS/SVN expertise classifications

	Speed	jArk	Pacman
Total files	56	172	223
Files with deltas	8	130	122

Speed. Figure 7.8 shows Syde, CVS, and delta expertise maps for a set of files from the Speed project. The Syde map suggests that developers dark blue (Bob) and pink (John) are the experts of the majority of the files, whereas the CVS map shows that John has not committed a single file. Figure 7.9 reinforces this observation. While John is responsible for 40% of Syde changes, he has not committed any change to CVS. The situation with Alice is exactly the opposite: she is responsible for less than 20% of Syde changes, but appears as the major contributor according to CVS.

We investigated why John did not commit any change to CVS, and determined that this behavior was influenced by two factors. The first was the late adoption of CVS, which occurred four days after the project had started – towards the end of the first week. The second factor was

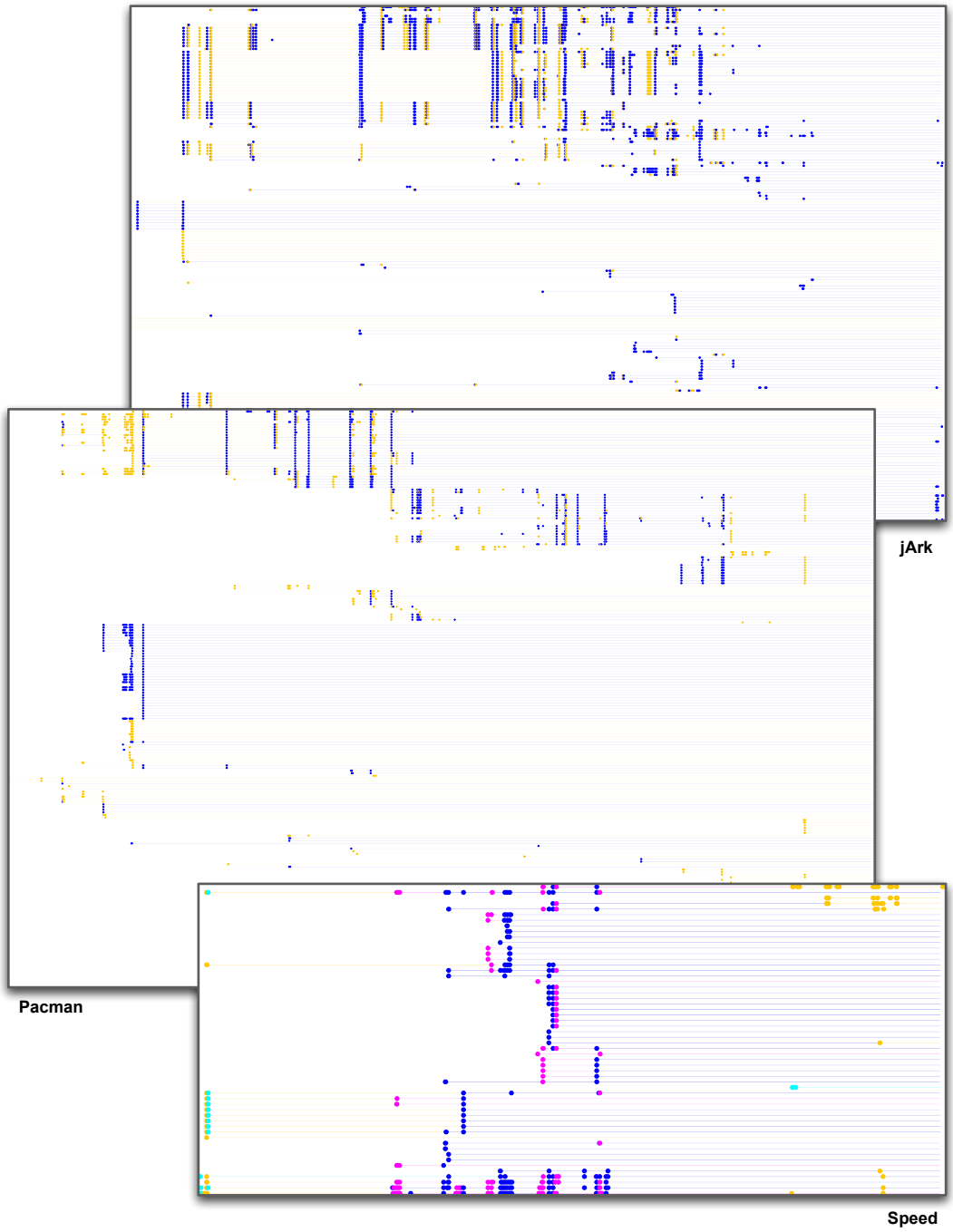


Figure 7.7. Overview of Syde expertise maps of projects Speed, jArk, and Pacman. This picture aims to provide an overview of the expertise maps of the three projects we studied. It is not meant to be inspected in detail by the reader

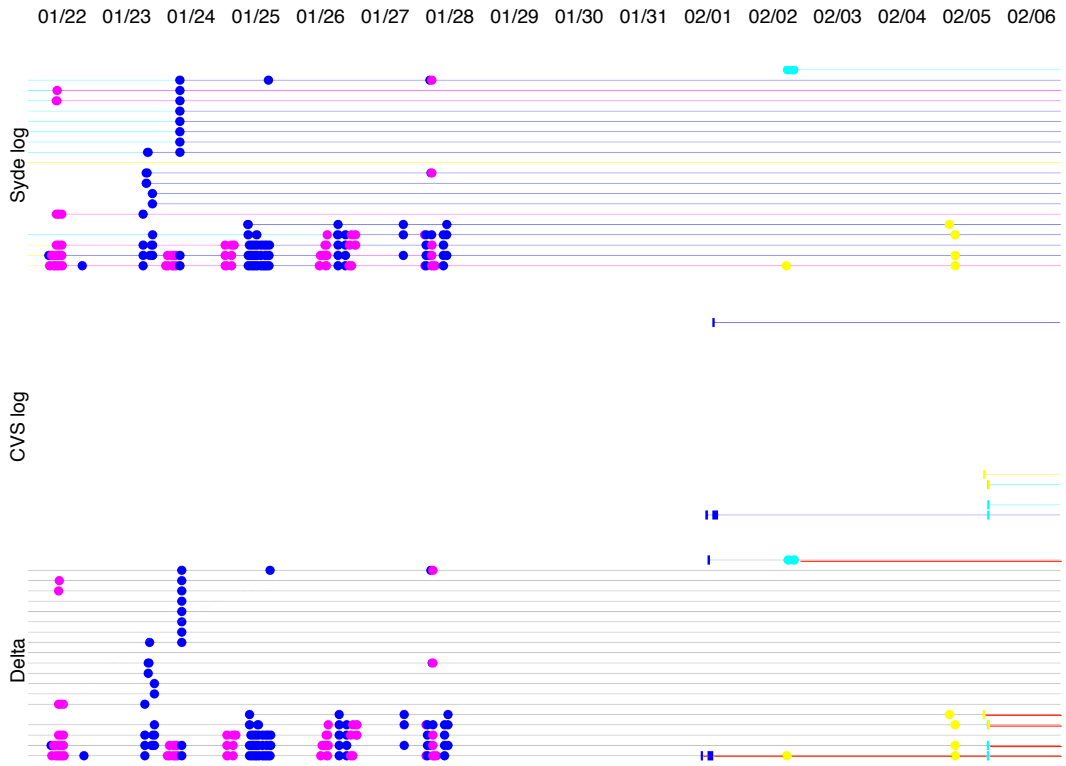


Figure 7.8. Expertise characterization for a set of files in the Speed project. Colors represent developers, lines represent files, circles represent code edits, and rectangles represent commits. In the Delta map, red lines represent differences in expertise classification between the Syde and the SVN maps.

that John had an active role in three other projects within the company, and had higher priority on one of the other projects in the second week of the experiment. Therefore, he was only able to commit his changes to this project after the period of the study.

It is evident from the differences between the Syde and CVS maps that the Speed developers do not commit their code frequently, nor are the same commit behaviors shared by every member of the team. Figure 7.9 leads us to the same conclusion: in the context of Speed, the definition of code expertise according to Syde is more suitable than that of CVS. Based on this, we suggest that the larger the difference between a developer's effort (measured in small changes) and the frequency of his commits, the more suitable our approach is in comparison to that Gîrba *et al.* However, since the developers of this project reported that they were developing multiple projects at the same time, and that some of them occasionally forgot to use Syde, further investigation was needed to support our suggestion. We hence performed the same study on the two other projects at our disposal.

Pacman and jArk. Looking at the distribution of changes/commits per developers of Pacman and jArk, we observe that Syde changes are consistent with SVN commits. The authors who performed the most changes are also the ones who committed more often, even though the

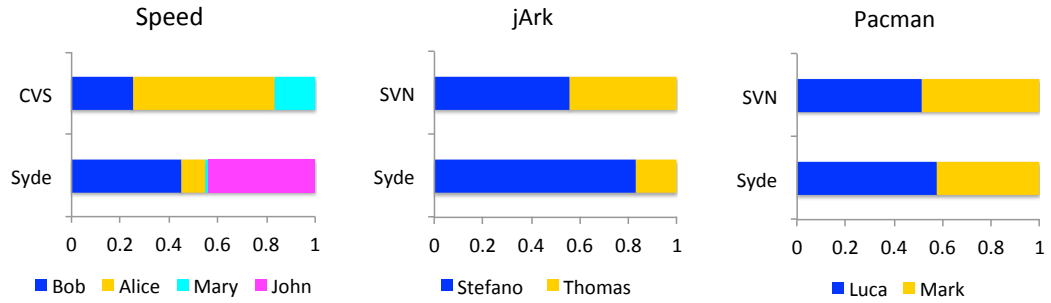


Figure 7.9. Distribution of changes per developer for Syde, and commits per developer for CVS/SVN for each of the three projects. We include all files that had at least one Syde change, or at least one commit for CVS/SVN.

percentage of Syde changes is higher than the percentage of commits in both cases (*e.g.* in jArk, Stefano performed about 80% of Syde changes, and about 55% of the commits). We know that the students worked physically together most of the time, and otherwise worked remotely with frequent communication via instant messaging. Unlike with the Speed project, they were focused on only one project and split the workload equally. We believe that these characteristics influenced the consistent relationship between Syde changes and SVN commits.

Based on the distribution of changes and commits for each developer on these two projects, we expect that the differences between the Syde and SVN-based measurements for expertise classification are influenced by the frequency with which developers commit. In other words, we expect to see differences in classification for the two measurements during the period between a set of Syde changes and a commit, where a takeover happens. To investigate where deltas appear, we take a close look at the maps of one day of work in the Pacman project.

Figure 7.10 shows the expertise maps for one day of work on Pacman. By carefully analyzing the three maps, we notice that in most of the cases a developer works on a couple of files for some time, and commits his changes later. This is the case for files `BoardView` and `GameFrame`; `Creature`, `Pacman`, and `Ghost`; and `Board` and `Level`.

The Syde and SVN maps show that both developers are often working on the same files in parallel (*e.g.* `BoardView`, `GameFrame`, `Creature`, `Pacman`, `Gost`, `Board`, and `Level`). In such cases, the source of any differences between Syde and SVN expertise is related to both the frequency of commits and the nature of the measurements taken. While the measurement based on Syde logs relies on the actual amount of work and time that one spent, the measurement based on CVS/SVN commits relies on the number of lines changed, which does not always directly reflect one's effort.

For instance, imagine that a developer created and implemented a method, tested it, fixed some tricky defects that took him a considerable amount of time, and finally refactored it. Syde records every edit step, reflecting how much effort this developer put on implementing this method, while the CVS/SVN measurement is only informed with the number of lines added to the file corresponding to this method. While the latter can be a good indicator of effort, it is not perfect.

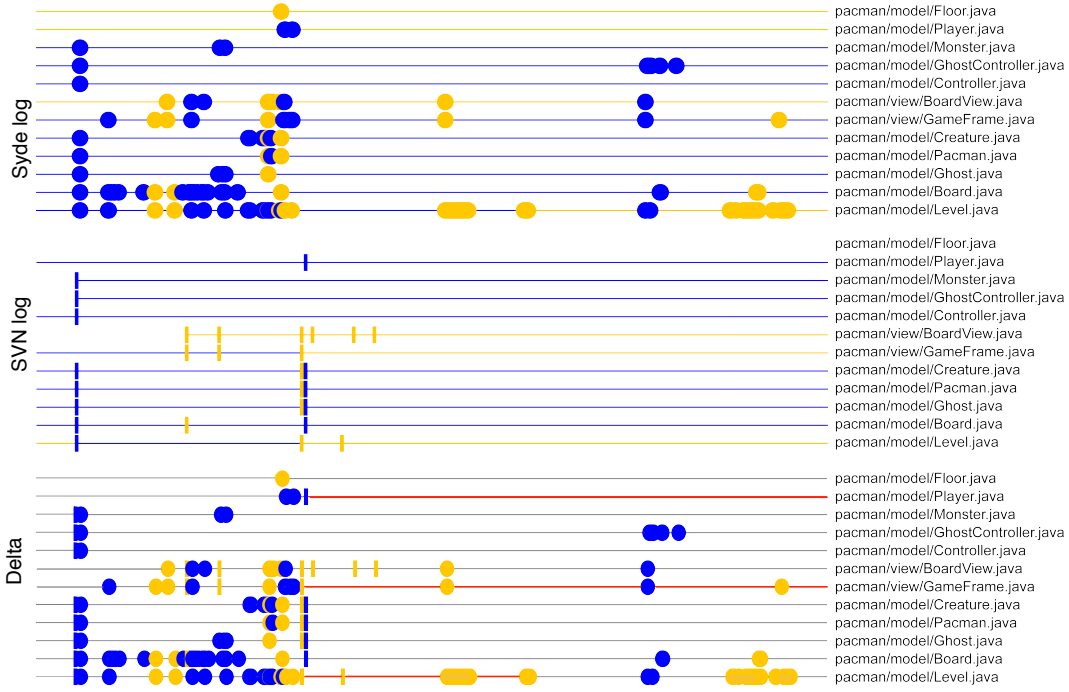


Figure 7.10. Expertise characterization for a set of files from the Pacman project

It is common sense that the more effort someone puts on a task, the more likely it is that he will remember it. Based on this, we reaffirm that the larger the difference between a developer's effort (measured by the number of small changes performed) and the frequency of his commits, the more suitable our approach is in comparison to that of Gîrba *et al.*

7.5.3 Evaluating Code Expertise with Forgetting

In Section 7.3, we introduced the concept of forgetting, and incorporated it into Syde's measurement of expertise. In this section we evaluate the forgetting effect by comparing the results of the measurement with different values of memory strength. The function we adopted to describe forgetting is $R = e^{\frac{-T}{s}}$, where R is the memory retention, T is time, and s is the strength of a developer's memory. Ideally, the value of s would be empirically determined by comparing it to the opinions of the developers themselves. However, we did not have that information at our disposal. We hence rely on heuristics and compared the behavior of expertise for several values of s. The values of s that we selected for comparison are 5, 25, and 125; we chose powers of 5 because the forgetting function is exponential, and with these three values we cover a reasonable range of memory retention. As shown in Table 7.4, a value of $s = 5$ reaches a low memory retention value after 10 days (0.14), while $s = 25$ reaches the same value after nearly two months; a value of $s = 125$ yields a memory retention value that is still strong after that time.

To determine the best memory setting for each project, we use two heuristics: minimizing the ratio of *short-term* switches in expertise among all that occur, and minimizing the overall average number of switches per files. A switch occurs when one developer replaces another

Table 7.4. Percentage of memory retention after a number of days for the values of memory strength chosen for this study: 5, 25, 125.

	days						
Memory strength	**1**	**5**	**10**	**20**	**30**	**40**	**50**
s = 5	0.82	0.37	0.14	0.02	0.00	0.00	0.00
s = 25	0.96	0.82	0.67	0.45	0.30	0.20	0.14
s = 125	0.99	0.96	0.92	0.85	0.79	0.73	0.67

in the position of having the most expertise on a file. A short-term switch is a switch whose overall duration is four hours or less. Finding the value of *s* that minimizes these two values leads to a better expertise description, as it minimizes what can be seen as spurious changes of expertise. Table 7.5 reports both these values for each project and each level of memory strength, including the default Syde-based expertise measurement, which assumes a perfect memory with no forgetting.

Table 7.5. Percentage of short-term expertise switches among expertise switches, and average number of switches per file, according to project and memory strength

	Short switches			**Switches per file**		
Memory strength	**Speed**	**jArk**	**Pacman**	**Speed**	**jArk**	**Pacman**
s = 5	10.0%	29.6%	44.4%	0.70	0.72	0.57
s = 25	32.2%	31.1%	40.5%	1.05	1.08	0.83
s = 125	32.1%	32.5%	40.7%	1.00	0.93	0.63
full memory	31.6%	33.3%	41.7%	1.02	0.81	0.60

For a more fine-grained view of the expertise switches, we also show the distribution of short and long-term expertise switches among files and memory settings in Figure 7.11. This figure shows a series of histograms displaying expertise switches per file. The *x* axis represents the number of switches that may have occurred in a file, and the *y* axis shows the number of files that contained that exact number of switches throughout their lifetime. The first three columns of graphs show histograms that consider the forgetting effect in order of increasing memory strength, whereas the last column does not consider forgetting. Black bars represent short-term expertise switches, while gray bars represent the remainder of expertise switches.

For all three projects, the majority of the files did not have switches: 28 out of 56 files for Speed; 120 out of 172 files for jArk; and 159 out of 223 files for Pacman. We removed them from the histogram in order to increase the resolution of the *y* axis and hence the legibility.

Relationship between number of switches and memory. If we analyze the files by frequency of switches, the histograms indicate that overall, a weak memory yields the lowest number of switches. In other words, there are more files with a low number of switches when *s* has a small value, while when *s* has a large value, there is a greater number of files with switches. Table 7.5 confirms this observation: the lowest ratio of overall switches per file is consistently obtained for

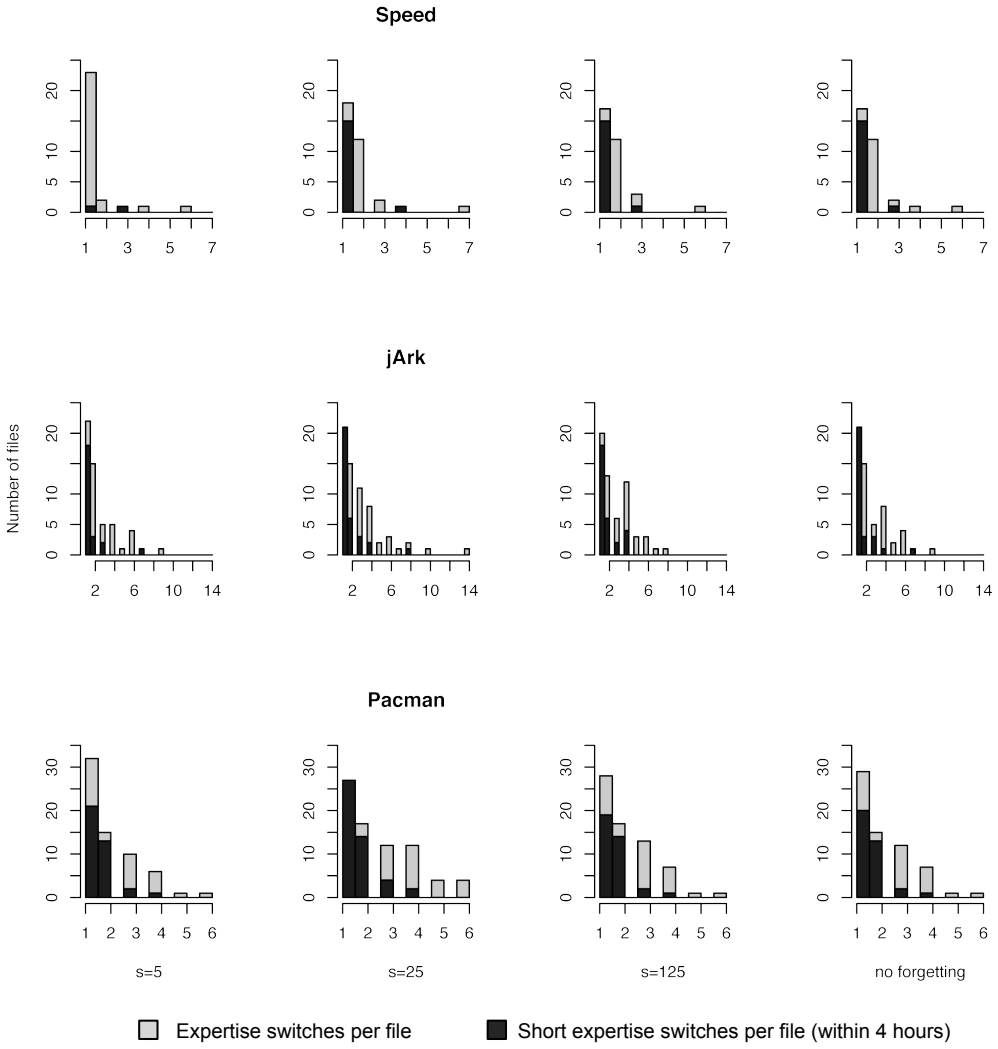


Figure 7.11. Histogram of expertise switches per file according to Syde changes. The first three columns consider forgetting with $s = \{5, 25, 125\}$, while the last has no forgetting effect.

$s = 5$, although larger values give a close ratio for the Pacman project. The maximum ratio of switches per file is found for $s = 25$, and it decreases slightly after that.

Speed has a very low number of files with two or more switches for $s = 5$. A stronger memory (higher s value) noticeably increases the number of files with two switches. Pacman and jArk contain a greater number of files with expertise switches, but have an overall smaller ratio of conflicting switches per file. Speed is an overall smaller project (at least the package that we focused on), with twice the number of developers; hence the potential for conflicts increases. On the other hand, jArk and Pacman both have several files with a large number of switches, indicating that a small number of files in the project have an increased rate of conflict.

Relationship between memory and short-term switches. Consulting Table 7.5, we observe that for Speed and jArk, the proportion of short-term switches among overall switches increases as the strength of the memory increases. For Pacman, this trend is true for $s = 25$, $s = 125$, and no forgetting, but the value for $s = 5$ is, surprisingly, the highest. $s = 5$ gives the fewest number of switches across all three projects, but a value of $s = 125$ or a measurement lacking the forgetting component provide a comparable number of short-term switches for the Pacman project;

With a closer look at the history of Pacman, we found that all 56 short-term expertise switches happened in the first half of the project, on a restricted set of 33 files. The developers of Pacman worked together at the beginning on the model of Pacman, and then split: one stayed on the model, while another developed the view. This close collaboration early on caused a large number of unavoidable short-term switches: 23 short-term switches occurred on 10 files containing unit tests; 13 short-term switches happened on 7 files on 20/04/2009; and 10 more occurred on 10 files on 06/05/09. This shows that the short-term switches are clustered around specific dates and denote bursts of activity. As such, they are not artifacts of the usage (or non-usage) of the forgetting effect. We can conclude that the best setting is again the lower memory value ($s = 5$) as it diminishes the overall quantity of switches, even though it does not reduce the number of short-term ones, which hence increase in proportion.

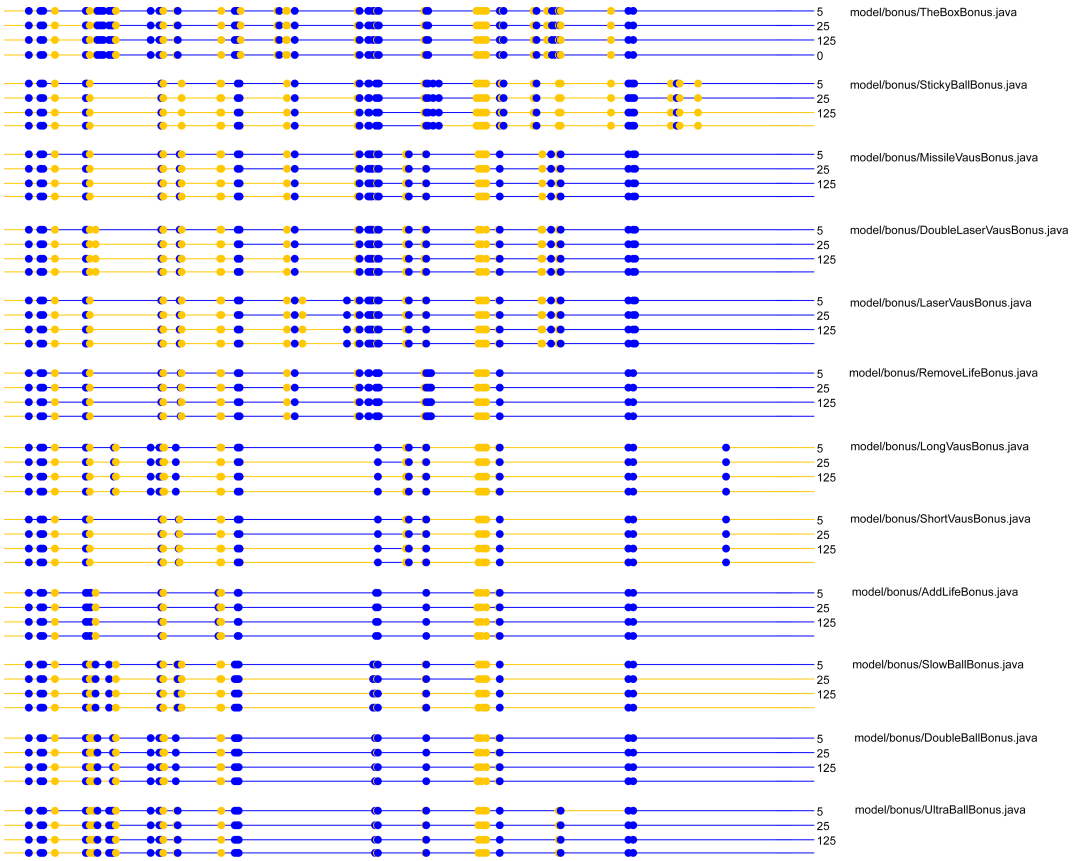


Figure 7.12. Expertise map of jArk with forgetting settings

A detailed view on expertise switches. With these observations in mind, we focus on investigating the behavior of the switches for the different rates of forgetting. To do so, closely examine a set of classes from jArk. Figure 7.12 shows a Syde expertise map where, for each class, there is one line for each s value, including a fourth line for the original measurement (with no forgetting). The main pattern that we observe is that the stronger the memory is, the longer it takes for an expertise switch to happen. Furthermore, when $s = 125$, the behavior is extremely similar to when developers have a perfect memory. For the files represented in Figure 7.12, they are in fact equal, though this is not the case in jArk overall. This might be an indication that $s = 125$ is too high, at least for the context of jArk. Indeed, Figure 7.2 shows that with this value developers would be expected to have a 70% probability of remembering a file 50 days after they changed it. In the context of jArk, which lasted for approximately 35 days, this value is already too high.

Conclusions. As previously stated, the strength of memory is a subjective value that depends on the various characteristics of each project. Therefore, there is an optimal range of values for each project, but there is no optimal value for all projects. In the case of jArk a low memory strength value is more suitable, since it was a project with a short duration performed by two developers with no clear method for task division, thus giving it more dynamic characteristics. In the case of Speed, though there was a clear division of tasks, more developers were involved, which increased the number of conflicts; it is therefore also more suitable to consider a lower memory strength for this project, at least until it grows in size. In the case of Pacman, a low value of memory reduces the overall number of switches, but keeps the number of short-term expertise switches nearly constant, and is hence preferable as well. Even though the three projects appear to be best represented with the same memory settings, it is too early to generalize beyond them.

7.6 Threats to Validity

7.6.1 Threats to Construct Validity

Construct validity refers to the extent with which our variables are correctly measured. We identified two potential threats to construct validity:

Syde Change Recording. Although Syde checks for compilation errors when a source code is changed, at the time the experiment took place, Syde did not compute the structural differences between two subsequent versions. In consequence, any edit to a file was considered a change, including addition or deletion of comments and blank lines.

Syde records every change made by a developer as long as he is connected to the Syde server. Speed was monitored with an early version of Syde that had some limitations. The history log collected from Speed is not complete, because some of the developers reported that they forgot to connect to Syde a couple of times. We minimized this issue by offering the option to automatically connect to the server, but this initial version of Syde did not enable automatic connection by default. The early version also lacked a buffer in the plug-in to save the changes performed while a developer was offline and to send them to the server when he connects. Thus a number of changes may have been lost, which may have influenced the accuracy of our measurement.

The jArk and Pacman projects were monitored with a later version of Syde, featuring both auto-connect enabled by default and a buffer to record offline changes. This allowed us to record a large portion of the changes that would have been lost in the previous version. There is,

however, a slight probability that the offline buffer was full in some instances, leading to the loss of a few changes.

SCM Usage. Syde was used at the initial implementation phase of Speed, but CVS was only adopted four days later (01/26). This delay in recording could have influenced expertise measurements at the beginning of the project. This was a decision taken by the team and hence beyond our control. Pacman and jArk adopted Subversion at the beginning of the project, so this threat does not apply for them.

7.6.2 Threats to Internal Validity

Internal validity refers to the validity of our causal conclusions. We identified two potential threats to internal validity:

Developer behavior under observation. Since developers knew they were observed, they may have altered their behavior in ways that we cannot predict. For example, they may have committed more or less often than usual. However, because we monitored our subjects for relatively long periods of time using non-intrusive tools (SCM change logs and Syde), we think they had time to get used to it; hence that effect should be weak.

Match of expertise measurements with developers' opinion. Ideally, we should have collected the developers' opinions of how much they knew about particular files they edited. This collection should have been done at fixed intervals throughout the data collection process, so that we would have points in time where we could compare our findings to developers' opinions. We did not collect this information, however, and it would be impossible to gather in retrospect. Showing the map to developers several months after their performance on the project would require them to have perfect memories to accurately check whether the map conforms to their notion of expertise.

7.6.3 Threats to External Validity

External validity refers to how much our results can be generalized to other circumstances. We identified two potential threats to external validity:

Number of systems. We monitored three projects (one industrial and two student projects) featuring a total of 8 developers over a combined time period of 28 weeks. This is still a relatively low number of projects and a short period of overall observation time (although two of the projects were monitored from start to completion). Moreover, all the projects were implemented using the same toolset: the Java programming language and the Eclipse IDE. These restrictions prevent us from deriving stronger conclusions at this time.

Styles of developers. It must also be considered that developers might present diverse patterns of saving and compiling, which could influence the results of code expertise measurement, since it is based on the number of changes each developer produced. We believe the usage of an IDE such as Eclipse, which outlines the errors still present in the code, encourages one to compile more often, thus mitigating this threat. In the same fashion, developers have different patterns of SCM usage. These usage patterns can be influenced by the development process that the team

adopts: agile development encourages developers to check in their code frequently, while more traditional processes encourage developers to maintain the consistency of the repository, which may delay their check-ins.

Since we monitored eight developers, we do not know if we account for all the variability they may have displayed; however, we did notice large differences between developers in SCM usage behavior with respect to the actual number of changes performed, which reinforced our opinion that our expertise metric is more resilient than the one based on SCM system usage.

7.7 Limitations and Future Work

Our work on code expertise advances the state of the art in two respects. The first is the creation of a measurement based on fine-grained changes, and the second is the integration of the notion of forgetting into this measurement. We obtained promising results in our evaluation; however, some limitations of our work need to be addressed to allow us to make further advances on helping developers to find code experts. In the following, we discuss these limitations and propose future directions for this work.

Comparison with other existing measurements. We only compared our measurements with the commit-based one proposed by Gîrba *et al.* [Girb 05b] because of the facility with which it could be applied to the history data of the object systems of our experiment. However, further evaluation should take other measurements – based on other data sources – into consideration, such as the interaction- and authorship-based Degree-Of-Knowledge model [Frit 10b], the bug-based measurement of Anvik and Murphy [Anvi 07], and the vocabulary-based model proposed by Matter *et al.* [Matt 09].

Creation and evaluation of new measurements. Fine-grained changes are a new source of authorship data, positioned between traditional authorship (commit-based) and interaction data. It could, however, be combined with interaction data to create an expertise measurement that might be closer to reality than our current measurements. In addition, we can also take the learning curve into consideration to further refine our measurements. There are a total of four factors to be combined: authorship, interaction, learning, and forgetting. To find out which combinations yield the best results, we need to perform a thorough evaluation collecting data from systems with longer histories and greater numbers of developers than those used in the evaluation reported in this chapter.

Comparison with developers' perception of expertise. An important future work would be to match the results of the measurement with the developers' perceptions of their expertise. An ideal scenario to be used in such an evaluation would be a middle-sized system implemented by a collocated team. This would be a safe choice to guarantee that the team maintains a reasonable level of awareness, consequently having the tacit knowledge of who knows what about various parts of the system. Additionally, we could ask developers to use the measurements and rank them according to their perception. This evaluation might indicate which measurements are more likely to be adopted.

Creation of a recommendation system. Syde currently features one view dedicated to displaying code expertise: Scamp's Buckets view (see Section 5.3.2 (p.64)). We believe that the need to locate experts, especially for newcomers, is high and that developers would benefit from having a recommendation system dedicated to this information. We plan to implement

a recommender in the form of a Eclipse plug-in to help developers locate those who are knowledgeable about an artifact of the system. The recommender should allow a developer to query for experts of a class, a package, or a method, and provide a rank of experts.

7.8 Summary

We have used the change history recorded by Syde to measure code expertise and to compare the result with the expertise computed exclusively with SCM-level logs. Similar to mainstream SCM systems, such as CVS, Syde produces history logs containing useful information about changes, which can be mined in the same context as the widely mined CVS logs. The fundamental difference between them is that Syde's logs are the result of continuous edits performed by developers, who do not need to stop their work to submit the changes. In contrast, CVS logs are the result of explicit check-ins of changes, the frequency of which can vary according to team culture, developer habits, and the likelihood of merge conflicts. Hence, we argue that Syde's logs reflect what happened in the past more accurately than those provided by mainstream SCM systems.

We defined a new expertise measurement based on the frequency with which developers change the code of each file; we subsequently refined our new measurement to incorporate the concept of memory loss to our definition of code expertise. That is, a developer who has performed the majority of code edits to a file, but has not touched it for a long period (over which the file has undergone significant changes), can be understood to have lost some knowledge of it. In the meantime, the developer who performs the recent changes becomes more knowledgeable, even though he may not have performed as many edits as the first developer.

To validate the Syde expertise measurement, we used the data collected by Syde, and the CVS/SVN logs from the development of three distinct projects: Speed, for a period of fifteen days and jArk and Pacman for a period of five weeks. We monitored eight developers for a total of 28 man/weeks, or 7 man/months.

We compared the results of the variants of expertise measurements with the help of the Expertise Map, a visualization introduced by Gîrba *et al.* [Girb 05b] that we extended to fit our data. The results showed differences between the two classifications, especially when active developers did not check in their changes frequently; in one case, a developer did not commit any code for two weeks, significantly skewing the measurement based on SCM data. Based on this finding, we suggest that our code expertise classification is more accurate than the one proposed by Gîrba *et al.* [Girb 05b], as it is less sensitive to developers' commit habits.

In addition, we suggest that the use of the concept of memory loss when measuring expertise reflects a more realistic scenario than the assumption that a developer remembers everything regardless of the amount of time that has passed. We found that models based on smaller memory retention in general satisfied the two heuristics of minimizing the overall number of expertise switches and of minimizing the number of short-term (possibly spurious) expertise switches. However, it is important to emphasize the subjective nature of forgetting, and thus, that the ideal rate of forgetting for each project is subject its own distinct characteristics.

The analysis of Syde's change history to measure code expertise, reported in this chapter, is the first evidence that the data made available by Syde opens new perspectives for several types of analysis, such as the understanding of developers' roles and activities, code expertise, and detection of unstable code. We also believe that since the data is being collected in real time, we can provide new types of "developer assistance" [Zell 07] or recommendation systems, especially

with respect to the collaborative aspects that Syde supports. In the next chapter we continue to investigate how Syde's change history may be used to assist developers by helping them to answer common questions whose answers can be derived from a system's evolution history.

Chapter 8

Helping Developers Answer Software Evolution Questions

8.1 Introduction

When evolving a code base developers keep a mental model of the system – an internal working representation of the software under consideration [Mayr 95] –, during software development or software maintenance. This individual understanding of the system is constantly being updated through the developer's interactions with the code and the team, and through the developer's search for answers to various questions [Sill 06; Alwi 08; Frit 10a; Ko 07]. These questions span multiple areas such as program comprehension, software evolution, collaborative software development, and program analysis [Robb 07b]; they therefore require a variety of information sources (*e.g.* colleagues, code bases, issue trackers, documentation, communication history) and multiple tools (*e.g.* [Stor 05; Lanz 05; Corn 10; Wett 11]) to satisfy them.

Although there are a number of resources (data and tools) available to ease the comprehension of a system and its evolution, the resources actually used by developers are often limited to talking to colleagues and exploring the code.

In an exploratory study [LaTo 06], LaToza *et al.* report that: 1) almost all teams have a *team historian*, who is the go-to person for questions about the code; 2) most team members subscribe to check-in messages to keep themselves updated with regard to code evolution, though many of them expressed dissatisfaction with the low level of detail provided by their teammates when describing the changes in commit messages.

We argue that this *low level of detail* is a fundamental problem for understanding software evolution, *i.e.* changes made by other developers. The problem is related to the coarse granularity with which changes are checked in and, consequently, seen by others. When trying to understand the evolution of the code, the delta between subsequent changes can be complex enough to prevent developers from inferring the design decision behind the changes in the code. Moreover, as indicated by previous studies [Grin 96; Souz 03], large commits can also lead to merge conflicts, duplicated work, and conflicting design decisions.

We have developed Replay (see Section 4.4.3 (p.48)), an Eclipse plug-in that allows developers to explore the rich change repository created by Syde. Developers can search for fine-grained changes made by a set of people to a set of artifacts and watch them in the chronological order

as originally performed in the IDE. This makes for a better user experience [Parn 10] than the aggregated form of commit-based Software Configuration Management (SCM) tools, such as CVS and Subversion.

We conducted a controlled experiment to assess whether Replay is at least as effective and efficient as the current state of the practice at providing support to developers with questions related to software evolution. The design of the experiment involved the selection of a set of common questions that developers ask, collated from previous catalogues [Sill 06; Alwi 08; Frit 10a; Ko 07]. We converted them into a set of tasks in order to measure both the correctness of the task solutions and their completion time.

The results of the experiment show that Replay leads to a decrease in completion time with respect to a set of software evolution comprehension tasks. We conclude that there are benefits in using Replay rather than the state of the practice tools for answering questions that require fine-grained change information and questions related to recent changes.

Structure of the chapter. In Section 8.2 (p.152) we present existing work related to the tool, and to the controlled experiment. In Section 8.3 (p.153) we describe the design and operation of our controlled experiment. In Section 8.4 (p.158) we analyze the experiment results, and in Section 8.5 (p.170) we discuss the threats to validity. In Section 8.6 (p.172) we illustrate the usefulness of replaying past development sessions in a slightly different context: education. In Section 8.7 (p.173) we discuss the limitations of this approach and future directions in which it might be improved. In Section 8.8 (p.174) we present the concluding remarks. In Appendix B we present the complete dataset that makes this experiment replicable.

8.2 Related Work

We discuss the work related to ours according to two perspectives: (1) the tool, and (2) empirical evaluation of software evolution and comprehension by means of controlled experiments.

8.2.1 Approaches Related to Replay

To our knowledge, Replay is the first tool to support replaying development sessions in a collaborative environment. However, other tools support programmers with their questions. Fritz and Murphy propose a prototype that combines different information fragments (source code, work items, change sets, teams, comments, wiki) to support 78 questions software developers commonly ask about a development project [Frit 10a]. The tool Ferret combines four different sources of information (static, dynamic, evolutionary, and Eclipse PDE) to build a knowledge base for answering conceptual queries [Alwi 08]. James is another knowledge base, composed of IDE interactions and micro-blogging, to support developers [Guzz 10].

Looking at our work in the broader context of software evolution, there are various lines of research that relate to ours. In the software evolution analysis context, several approaches make use of the changes performed to a system over its lifetime to support comprehension: Lanza, Gîrba *et al.*, and Lungu *et al.* respectively summarize and visualize respectively the evolution of classes [Lanz 01], the evolution of class hierarchies [Girb 05c], and the evolution of inter-module relationships [Lung 07]. In these works the history is not replayed, bur rather summarized, and the order in which the changes are performed is lost. We specifically focus on replaying the change events in the order in which they happened.

A few approaches focus on replaying the changes that happened in a system. Wettel and Lanza visualize the evolution of the entire system by allowing the user to travel in time and observe the changes of the system as they are represented in a 3D city metaphor [Wett 08]. Hindle *et al.* present an animation of the evolution of the architecture of a system [Hind 07] with the Yarn tool. The animation presents the evolution of the relationships between the modules of the system. While these approaches allow for the animation of the changes, they present the changes at a high level of abstraction from which the code is not easily accessible.

One major difference between this work and the aforementioned tools is the level of detail of the data. In most existing approaches the data is extracted from commit-based SCM systems, with the result that changes between versions can be arbitrarily complex. An approach that uses fine-grained change information was proposed by Robbes [Robb 08a]. Although he collects fine-grained information from software systems, Robbes exploits it for other purposes than for the replaying of changes, *e.g.* to detect and characterize development sessions [Robb 07b]. Dig, however, uses a change-centric approach to record sequences of refactorings, and to replay them on other library-based applications [Dig 07a].

8.2.2 Empirical Studies

There are relatively few empirical studies in the form of controlled experiments in software engineering. Further, there is no controlled experiment that directly relates to our aim: answering developers' questions related to software evolution. However, there are a number of controlled experiments related to software evolution and program comprehension.

Quante evaluates, through a controlled experiment, the support provided by dynamic object graphs to answer a set of program comprehension tasks [Quan 08]. Cornelissen *et al.* performs a controlled experiment to evaluate the use of Extravis, an execution trace visualization tool, to answer program comprehension tasks [Corn 10]. Wettel *et al.* assess the use of CodeCity in performing program comprehension and quality assessment tasks [Wett 11].

The major difference between these controlled experiments and ours is that they evaluate tools that support program comprehension by visualizing data other than source code (*e.g.* dynamic graphs, execution traces, and system models). We evaluate a tool that allows a developer to investigate the history of the system by looking directly at its source code.

8.3 Experiment Design

We want to quantitatively evaluate Replay's effectiveness and efficiency at helping developers answer the questions they ask while developing software. The developer questions we focus on are related to the evolution of the system and can be answered by analyzing data from source code repositories.

8.3.1 Research Questions and Hypotheses

We raise the following research questions:

RQ1: Does the use of Replay reduce the **time** it takes to answer software evolution questions compared to SCM-based tools?

RQ2: Does the use of Replay increase the **correctness** of the answers to software evolution questions compared to SCM-based tools?

RQ3: Does the user's **experience level** affect the potential benefits of using Replay in terms of correctness and time?

RQ4: Which **type** of questions benefit most from the use of Replay?

Research questions 1 and 2 cover the quantitative evaluation of efficiency and effectiveness, respectively. The goal of RQ3 is to discover whether the experience co-factor influences the results on efficiency and effectiveness, where experience is measured by assessing the subject's number of years of experience with using the technologies relevant for the experiment. We specifically chose this co-factor because having at least a minimal experience level was a pre-requisite for eligibility to participate in the experiment. Thus, it is guaranteed that all subjects will have some experience. In addition, some commonly used co-factors, such as gender and domain, are either irrelevant or inapplicable to this study. The goal of RQ4 is to qualitatively investigate how the benefits of Replay are related to the type of questions asked, which can guide future improvements. The null and alternative hypotheses associated with the first two research questions are formulated in Table 8.1.

Table 8.1. Null and alternative hypotheses

	Null Hypotheses		**Alternative Hypotheses**
$H1_0$	The tool does not impact the time required to answer software evolution questions.	$H1$	The tool impacts the time required to answer software evolution questions.
$H2_0$	The tool does not impact the correctness of the answers to software evolution questions.	$H2$	The tool impacts the correctness of the answers to software evolution questions.

To test the hypotheses $H1_0$ and $H2_0$ we define a series of tasks that have to be performed by the control and the experimental groups. The control group (Eclipse+SVN) uses an Eclipse installation with default development tools and Subclipse[1] to answer the questions by accessing the change history from SVN. The experimental group (Eclipse+Replay) uses the same Eclipse installation with default development tools and Replay to access Syde's change history.

We maintain a between-subject design, meaning that each subject is part of either the control or the experimental group. Although this choice of design faces a number of disadvantages, such as the need for a larger number of subjects to generate useful results, or the difficulty of maintaining homogeneity across groups, it suffers little contamination by extraneous factors, thus generating more reliable results.

To answer RQ3 we analyze the data within blocks to check whether a participant's experience level influences their performance. For the last question (RQ4) we perform a separate qualitative analysis of correctness and completion time for each task.

[1] Subclipse provides support for SVN in Eclipse `http://subclipse.tigris.org/`

8.3.2 Object System

The system we chose as the object of this experiment is called *SpreadSheet*. Developed by a team of four BSc students, it is a simple spreadsheet application with support for basic mathematic formulae. Its development lasted for six weeks, and at the end, the project counted 13 packages, 77 classes, 286 methods, and totaled 1,882 lines of Java code. The number of SVN commits was 137, while the number of recorded Syde changes was 11,661. The choice of this specific system was constrained by our need to have the change history recorded with both Syde and SVN for the entire development cycle. Thus, it was not possible to choose an open-source system, nor did we have at our disposal a team working on a commercial system. This choice implies some threats to the validity of the experiment, discussed in Section 8.5 (p.170).

8.3.3 Task Design

We want to evaluate whether Replay outperforms a baseline (Subclipse) for assisting developers in answering comprehension questions related to a system's evolution.

We considered previous catalogues of questions [Alwi 08; Frit 10a; Ko 07; Sill 06], selected those that could be answered by investigating the change history of a system, and created corresponding tasks. Table 8.2 provides a short description of each task together with its goal and rationale. Each task is an adapted version of a question from a previous catalogue [Frit 10a].

In the handout distributed to the subjects, the descriptions of the tasks are integrated in the following hypothetical scenario: the subject is joining a team to replace a developer that left. The scenario makes the subjects feel as if they have become part of the team and are gradually learning the system while solving the experimental tasks.

8.3.4 Subjects

We conducted the experiment with 45 subjects: 18 MSc students, 25 PhD students, one postdoc, and one professor. The participants' average age was 27.84, comprising 14 different nationalities. The MSc students, the postdoc, and the professor have a software engineering background. The PhD students additionally have other backgrounds, such as information retrieval, human-computer interaction, or security. Participation was voluntary. None of the participants had previous experience with Replay.

8.3.5 Operation

The operation was composed of several experimental runs. Each run included a training session of approximately 15 minutes and one experimental session. Each training session consisted of a tutorial about how to use the tool, given by the experimenter, followed by a hands-on session where the subjects perform some small tasks and could ask clarification questions. The experimental session was composed of six tasks, with time limits as follows: 10 minutes for each of the first four tasks, 20 minutes for task five, and unlimited time for task six.

The sessions were conducted using the subjects' laptops, with instructions provided to configure them for the experiment. The control group had Eclipse and Subclipse installed; a local SVN server was provided. For the experimental group, the subjects were asked to install Eclipse and Replay.

There were twelve experimental runs in four locations: one run with seven participants at the University of Berne; one with six participants at the University of Zurich; four with two

Table 8.2. Tasks' description, goal and rationale

Id	Description	Goal	Rationale
1	Imagine that you are joining the project's team to replace a member. Find out what he was working on, so you can start from what he left unfinished. Identify the two classes he changed the most in the past days.	Becoming familiar with someone else's work	It is not uncommon for developers to leave and join the team/company during the development of a software system. The goal of this task is to simulate when a newcomer has to take over the responsibility of a developer who just left the team [Ko 07].
2	You have just started to work on a set of classes, and want to find out whether someone else has recently changed them before you commit your changes. Identify the methods that someone else has also changed.	Becoming aware of team activity	Developers are not simply interested in knowing who is working on what, but are rather interested in knowing who is working on parts of the system that can impact on their work (or that their work may impact on) [LaTo 06; Ko 07; Frit 10a].
3	You have identified one of the main classes of the system but cannot quite understand it. Look for experts who can help you by searching who has recently changed it the most.	Finding experts at the class level of abstraction	Developers often find themselves trying to understand a part of the code that was written by someone else [LaTo 06]. The goal of this task is to simulate how a developer would find out who to ask for help on a class.
4	You are taking over the responsibility of a class and your first task is to refactor the code to improve its design and readability. You want to start with the most complicated feature, because it will need most of your effort. From the list below, identify the feature that provoked the largest number of changes.	Relating a feature to code changes	Developers are often confronted with the task of making a part of the source code more readable and maintainable [Fowl 99; Gamm 95], usually through refactorings. In order to do so, they need to understand the code, and identify its most problematic parts, its design flaws, *etc.* This task tackles the identification of parts of the code that need more attention.
5	You are given the description of a defect and instructions to reproduce it. Find out the origin of this defect, when and by whom it was introduced. Propose a fix.	Tracking back the introduction of a defect	Resolving defects is part of every developer's job. The goal of this task is to resolve a defect by tracking back when it was introduced and reverting the changes.
6	Before you joined the team the system underwent a major refactoring, which involved the deletion of a class and restructuring of other classes. Investigate why this refactoring took place.	Understanding the rationale behind past refactorings	Decisions taken during the development of a system are seldom documented [Sing 98; Oezb 07]. The goal of this task is to simulate a situation in which one needs to understand the rationale behind an undocumented refactoring.

participants, two with one participant, one with six participants and one with thirteen participants at the University of Lugano; one with two participants, and one with one participant at the University of British Columbia. Table 8.3 summarizes the twelve experimental runs.

Table 8.3. Summary of the experimental runs

Date	Location	Participants
10.12.2010	University of Lugano	1 PhD
13.12.2010	University of Lugano	4 MSc, 2 PhD
17.01.2011	University of Berne	1 MSc, 5 PhD, 1 Professor
13.01.2011	University of Lugano	1 PhD
13.01.2011	University of Lugano	1 PhD, 1 Postdoc
18.01.2011	University of Lugano	2 PhD
19.01.2011	University of Lugano	2 PhD
25.01.2011	University of Lugano	2 PhD
28.01.2011	University of Zurich	6 PhD
23.08.2011	University of British Columbia	1 PhD
24.08.2011	University of British Columbia	2 PhD
21.10.2011	University of Lugano	13 MSc

8.3.6 Pilot Studies

To refine the experiment design and make Replay mature enough to guarantee an operation without technical impediments, we ran 8 pilot studies involving 19 people over the course of 4 months. The participants and the data points of the pilot studies were totally discarded from the data analysis. The summary of the pilot runs is presented in Table 8.4.

Table 8.4. Summary of the pilot runs

Date	Location	Participants
07.2010	University of British Columbia	1 PhD
09.2010	University of Lugano	2 PhD
09.2010	Carnegie Mellon University	1 PhD
09.2010	University of Sannio	1 PhD
10.2010	Free University of Brussels	1 MSc
10.2010	University of Antwerp	4 PhD, 1 Postdoc
10.2010	Delft University of Technology	4 PhD, 3 Postdoc
12.2010	University of Lugano	1 PhD

The first three pilot runs were planned, while the last five were potential experimental runs that had to be discarded. The major concern after running the planned pilots was over improving Replay's performance at retrieving the change history to be comparable to Subclipse.

The fourth and fifth runs were executed without problems, but during the sixth run we identified that one of the tasks to be answered by the experimental group was unfeasible. Therefore, we had to change the task and discard the previous runs. For the seventh run, we prepared a VirtualBox[2] image with the tools to be used in the experiment, and had the subjects

[2]See http://www.virtualbox.org/

install the images onto their laptops. Because some settings were not controlled for, this led to several problems, including slowdowns caused by the restricted memory space of the running VirtualBox image, poor navigability caused by the small screen size of some subjects' laptops, and slowdowns in fetching the change history for both groups (experimental and control). To solve these problems, we removed the virtual image from the experiment package, included the repositories into the package to allow for local access, and specified that participants were required to use a minimum of a 15" screen. In the last pilot run, configuration problems affected the completion time of the tasks, so we decided not to include it in the analysis.

The pilot studies had a fundamental role in making the experimental runs as homogeneous as possible. We discarded multiple potential runs until we reached a point where no more modifications in the experimental setup were conducted. With this approach, we minimized the effects of having multiple experimental runs by ensuring a solid and homogeneous setup.

8.3.7 Data Collection

We collected four different types of data during the experiment:

1. **Personal information.** Before the experiment, we collected information about the subject (*e.g.* age, affiliation) and the subject's experience with Java, Eclipse and Subversion. This was done with a screening questionnaire.

2. **Timing data.** To time the participants, we adopted two strategies. When the session involved up to two subjects, the experimenter timed them manually. When the session had more than two subjects, the experimenter used a timing web application to time each subject, and also to show them their remaining time. In either case, the experimenter notified the subjects when they went over time, and allowed them to write down their findings before going to the next task.

3. **Correctness data.** To convert the solutions into quantitative values, we established a grading system. Each task was worth 1 point, evenly distributed according to the number of correct answers that subjects were required to enter, *e.g.* if there were 4 correct answers, each is worth 0.25, while each wrong answer counts as -0.25. The correct answers were determined by the experimenter, and double-checked by two other people.

4. **Participant feedback.** The experiment ended with a debriefing questionnaire, where the subjects assessed the time pressure, the difficulty of the tasks, and whether the tasks were realistic. The subjects were also given the opportunity to write down their opinions about the experiment and the tools.

8.4 Analysis and Interpretation

We performed a preliminary analysis on the subjects' opinions regarding the tasks in order to check for exceptional conditions.

Difficulty. We asked the subjects to indicate how much time pressure they felt during the experiment, ranging from 1 (no pressure) to 5 (too much pressure). The average time pressure reported was 2.75 (stdev. 1.02) for the control group and 2.55 (stdev. 1.05) for the experimental

group, who felt slightly less time pressure. We sorted the tasks in increasing order of difficulty throughout the experiment, which was confirmed by the subjects' assessment, shown in Figure 8.1.

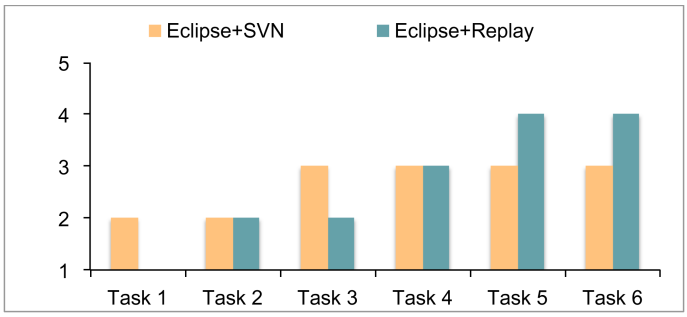


Figure 8.1. Median difficulty values: 1 - trivial, 2 - simple, 3 - intermediate, 4 - difficult, 5 - impossible

Although there is a small difference on the perceived difficulty between control and experiment groups in tasks 1, 3, 5, and 6, this perception did not characterize a high discrepancy in terms of either completion time or correctness; we therefore decided to maintain these tasks in the analysis. Since task 6 required a subjective answer in the form of a short essay, it is not included in the statistical test.

Realism. As shown in Figure 8.2, the participants felt that the tasks reflected situations that happen in real development scenarios.

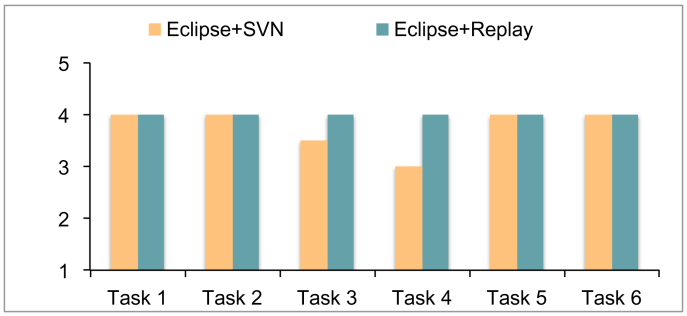


Figure 8.2. Meadian: 1 - strongly disagree, 2 - disagree, 3 - undecided, 4 - agree, 5 - strongly agree

Task 4 received the lowest grade, especially from the control group. We believe this was due to the formulation of the task description rather than the task goal.

8.4.1 Subject Analysis

We followed the suggestions of Wohlin *et al.* [Wohl 00] regarding the removal of outliers caused by exceptional conditions before performing our statistical test. We registered a number of outliers.

One subject from the control group was unable to finish the experiment in the allotted time due to lack of experience with Eclipse. One subject from the experimental group did not follow the instructions provided in the handout regarding the tools he was allowed to use, and used the tools reserved for the control group instead. One subject from the experimental group did not understand the concept of fine-grained changes provided by Replay. His answers clearly demonstrated that he did not use the tool, but rather answered randomly, characterizing himself as an outlier both in terms of correctness (low grading score) and time (low completion time). Finally, two subjects (one from each group) failed to register the completion time of at least one of the tasks.

We excluded these five cases from the statistical analysis, and were left with 40 subjects. We previously assigned subjects to the control versus experimental group using a combination of randomization and blocking according to their experience level. We asked the subjects to indicate the number of years of experience they had with programming in Java, using Eclipse, and using SVN. The criterion used for the blocking was: a subject is considered advanced only if he has at least four years of experience with Java and Eclipse, and at least one year of experience with SVN. If one of these criteria is not met, the subject is classified as beginner. As a result of the random assignment and after the removal of the outliers, we obtained a fair distribution of subjects, as shown in Table 8.5.

Table 8.5. Subject distribution

	Eclipse+SVN	**Eclipse+Replay**	**Total**
Beginner	12	11	23
Advanced	8	9	17
Total	20	20	40

8.4.2 Interpretation of the Results

Our experiment has a balanced between-subjects design with one independent variable, *i.e.* the tool. The choice of the hypothesis test depends on whether the sample distributions are normal and have equal variances. If these two requirements are met, we can choose the parametric Student's t-test; otherwise, we should use the nonparametric Mann-Whitney U test.

We performed the Shapiro-Wilk test of normality (see Table 8.6), which rejected the hypothesis that the experimental sample for correctness is normal (p-value $= 0.022 < 0.05$). For completion time, we also performed the Levene test (see Table 8.7) and verified that the samples have equal variances. The descriptive statistics related to correctness and completion time are presented in Table 8.7.

The results of the statistical tests are presented in Table 8.8. Since the completion time results are normally distributed with equal variances, we use the Student's t-test to analyze them. For correctness we must use the Mann-Whitney U test.

8.4.3 Results on Completion Time

We first test the null hypothesis $H1_0$, which states that the use of the tool Replay does not impact the time required to complete the assigned tasks.

Table 8.6. Results of the Shapiro-Wilk test of normality

	Group	p-value
Time	Eclipse+SVN	0.213
(minutes)	Eclipse+Replay	0.576
Correctness	Eclipse+SVN	0.669
(points)	Eclipse+Replay	0.022

Table 8.7. Descriptive statistics of the experiment results

	Group	Mean	Diff.	Min.	Max.	Stdev.
Time	Eclipse+SVN	47.71		36.20	55.55	5.98
(minutes)	Eclipse+Replay	42.19	-11.56%	32.00	53.92	5.24
Correctness	Eclipse+SVN	3.52		1.00	5.00	1.04
(points)	Eclipse+Replay	4.11	+16.76%	2.33	5.00	0.81

Table 8.8. Results of the statistical tests

	Group	S-W		Student's t-test			MWU
		p-value	Levene	t	df	p-value	p-value
Time	Eclipse+SVN	0.091					
(minutes)	Eclipse+Replay	0.855	0.144	2.762	38	0.009	
Correctness	Eclipse+SVN	0.065					
(points)	Eclipse+Replay	0.046					0.062

Table 8.7 shows that the experimental (Eclipse+Replay) group took on average 11.56% less time to complete the tasks than the control (Eclipse+SVN) group, and that this result is statistically significant at the 99% confidence interval (p-value $= 0.009 < 0.01$ for the t-test). With these results, we can reject the null hypothesis $H1_0$ in favor of the alternative hypothesis $H1$, and positively answer RQ1.

Figure 8.3 shows a box plot[3] of the total time (in minutes) the subjects spent on the first five tasks. As can be seen, the 50^{th} percentile of the experimental group is at roughly at the same level as the 25^{th} percentile of the control group. This means that almost 50% of the experimental group subjects completed the tasks before 75% of the subjects from the control group did, *i.e.* 50% of the subjects completed the tasks before (roughly) 43 minutes, while 75% of the subjects from the control group took more than (roughly) 43 minutes to complete the tasks.

The variability (interquartile range) of completion time is lower in the experimental group than in the control group. A series of factors gave advantages to the control group. For instance, Replay equally unknown to all participants, while most of the subjects had some experience

[3]The right end of the box represents the 75^{th} percentile, the left end of the box the 25^{th} percentile, and the line in the middle the 50^{th} percentile (median).

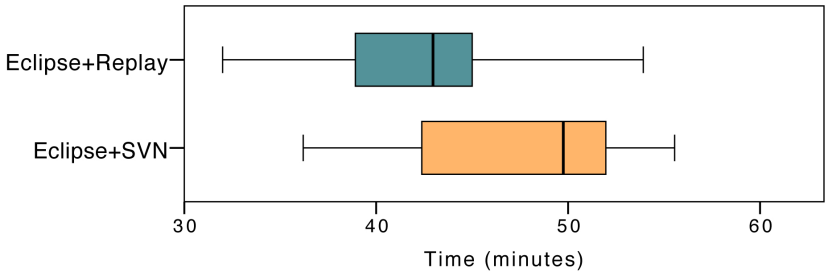


Figure 8.3. Box plots on completion time

with SVN. In addition, the experimental group subjects' previous experiences using other tools (including SVN itself) and their dedication to understanding Replay before starting the tasks may have impacted their completion time. Lastly, the experimental group had to look through finer-grained and larger numbers of changes than the control group did to find answers. However, these "drawbacks" of the Replay tool did not prevent the experimental group from outperforming the control group in terms of completion time and variability.

8.4.4 Results on Correctness

Table 8.7 shows that the experimental group obtained, on average, a score 16.76% higher than the control group. However, this result is not statistically significant at the 95% confidence interval (p-value $= 0.062 > 0.05$ for MWU test), but it is at the 90% confidence interval. We are unable to fully reject the null hypothesis $H2_0$, and partially answer RQ2.

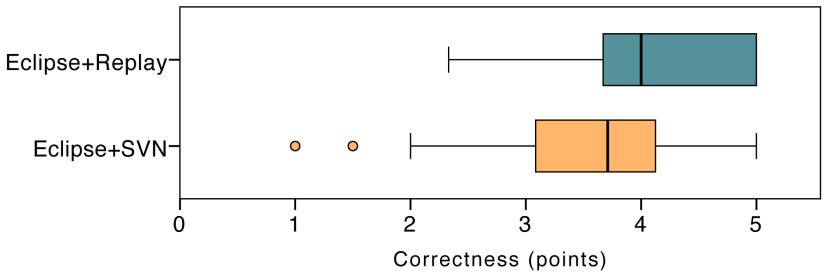


Figure 8.4. Box plots on correctness

Even though the results are not statistically significant at the 95% confidence interval, Figure 8.4 shows evidence that the experimental group's performance was superior to that of the control group. The 25^{th} percentile of the experimental group is at the same level as the 50^{th} percentile of the control group, *i.e.* 75% of the subjects from the experimental group obtained higher scores than 50% of the control group subjects.

Figure 8.4 shows that there were two outliers in the control group, who scored 1 and 1.5 out of 5. These subjects (C16 and C19) are classified as beginners, as both have little experience with the tools used (1 to 3 years of experience with Java and Eclipse, and less than 1 year of experience with SVN). They rated themselves as beginners with using SVN, which is evidence

that they might not have been familiar with the tool, although they had the option of indicating that they had no familiarity with it (when a developer indicated no previous experience with SVN, he was automatically assigned to the experimental group).

Upon closer inspection we found no sign that their answers were randomly selected, because some of the mistakes they made were also common to other subjects. For instance, for task 4 both answered that the changes were most related to "handling mouse events", which was the most frequently made mistake in both groups. In addition, when they went over time, they left the answer sheet blank (*e.g.* task 5 for C16 and task 1 for C19), which supports our analysis that they did not answer randomly, but rather needed more time to answer some questions because of their low level of experience with the tool. Therefore, these subjects were not classified as outliers and were kept in the analysis.

Non-parametric tests (*e.g.* MWU, Wilcoxon signed rank, Kruskal-Wallis) are more conservative than parametric tests (*e.g.* Student's t-test), meaning that non-parametric tests require larger samples or greater difference in the values to yield statistically significant results. We argue that this is the main reason that the MWU test retained the null hypothesis at 0.05 significance level. There is evidence for the tendency of results to achieve statistical significance with larger samples; the results reported here – with 40 samples – have a greater degree of statistical significance in comparison with previous results, when there were 26 samples [Hatt 11c] (p-value decreased from 0.072 to 0.062).

8.4.5 Influence of the Experience Level

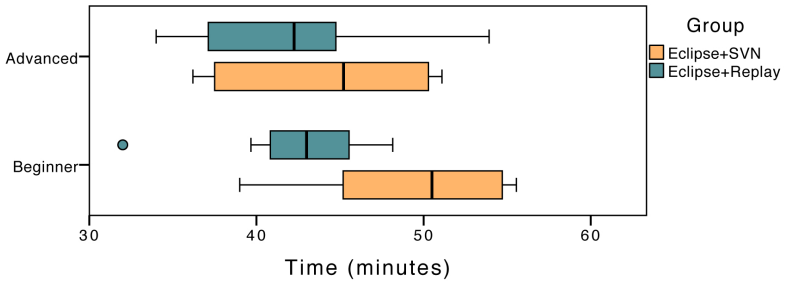
We compared correctness and completion time across the two levels of experience, i.e., beginner and advanced. Figure 8.5 shows that the experimental group outperformed the control group in both correctness and completion time, regardless of the experience level. Though the number of subjects per experience level is too low to yield statistically significant results, we can draw some observations.

The variability (interquartile range) of the experimental group was lower than that of the beginners and experts in the control group, although beginners in the experimental group had the lowest variability by a slight amount. Our assumption is that for beginners, because both Replay and SVN are new to them, this difference in variability represents that the learning curve of SVN is steeper than that of Replay. For those unfamiliar with SVN, we gave a tutorial on how to use it and allowed the subjects to get used to it before starting to perform the tasks. Although some experts felt that their performance was hindered by the fact that they were so used to looking at changes the "SVN way" that it was not easy to adapt to Replay, the variability difference of experts and beginners for experimental and control groups was not significant.

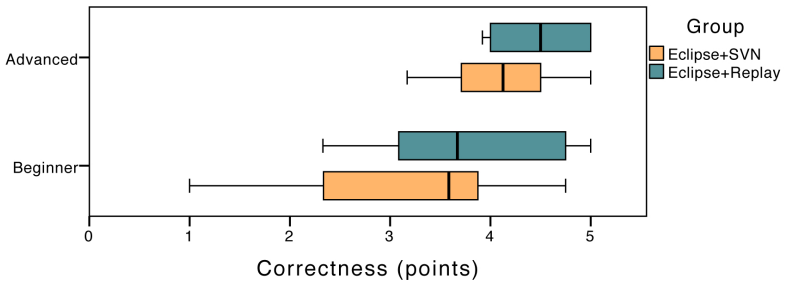
One outlier appears in the beginner-experimental group for completion time. This subject answered all questions correctly. Despite his lack of experience, he was efficient and effective, which might characterize him as a fast learner.

In terms of correctness, both beginners and experts from the experimental group had slightly higher variability than those with the same respective experience level in the control group. One factor that may have influenced the higher variability in both cases is that Replay presented a new method for investigating changes. Therefore, its effectiveness may have been influenced by the subjects' ability to learn how to interpret the information present in Replay.

We can answer RQ3 by stating that the users' experience level does not affect the potential benefits of using Replay in terms of correctness and completion time.



(a) Time (minutes)



(b) Correctness (points)

Figure 8.5. Beginner versus advanced

8.4.6 Individual Task Analysis

To identify which type of tasks can benefit most from the use of Replay (RQ4), we examine the performance of the two tools for each individual task. Table 8.9 shows the descriptive statistics for the individual task analysis, while Figure 8.6 shows box plots of completion time and correctness for each task. Note that the quantitative data for task 6 is not included in the table and figure aforementioned, because it was a subjective task.

Task 1 – Becoming familiar with someone else's work. The subjects were asked to familiarize themselves with recent changes made by one developer, and to identify the two classes this developer worked on the most. The experimental group achieved an excellent performance, while the control group took more time and had lower scores, with both results being statistically significant. Parnin and DeLine have observed that when a developer resumes one of his interrupted tasks, he prefers to see past changes chronologically rather than in aggregated form [Parn 10]. Our findings complement theirs by demonstrating that developers who see others' changes chronologically understand and perform better.

Task 2 – Becoming aware of team activity. The goal of this task was to identify which methods were recently changed and by whom. Although we have developed a plug-in that directly targets awareness [Lanz 10], Replay can also be used for this activity. The results show that Replay as efficient as the baseline and slightly more effective, though the results are not statistically significant.

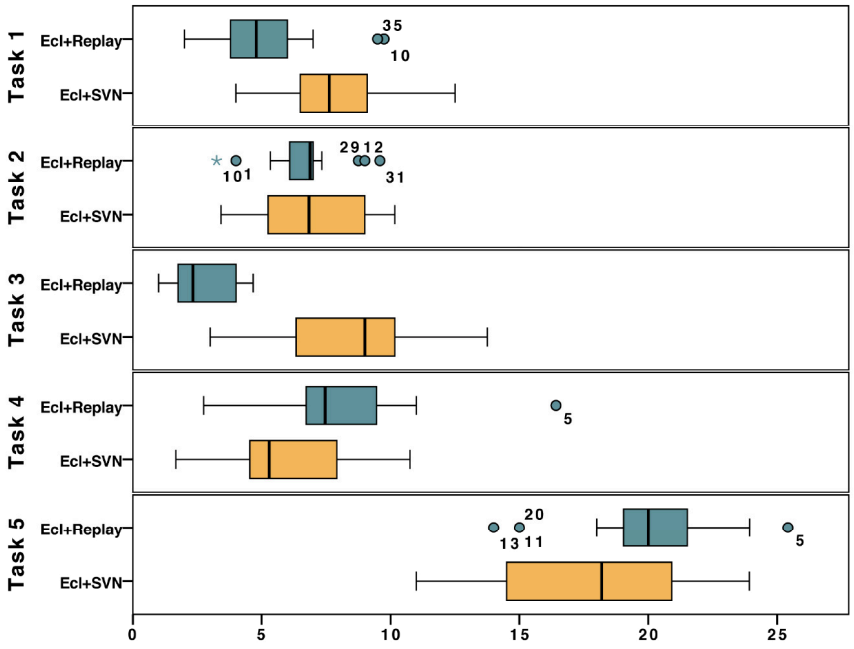
Table 8.9. Descriptive statistics of the individual task analysis results

		Group	Mean	Diff.	Min.	Max.	Stdev.	MWU p-value
Time (seconds)	**Task 1**	Ecl+SVN	7.89		4.00	12.50	2.35	
		Ecl+Replay	5.05	-35.99%	2.00	9.75	2.04	0.000
	Task 2	Ecl+SVN	6.91		3.42	10.17	2.19	
		Ecl+Replay	6.66	-3.62%	3.25	9.58	1.49	0.745
	Task 3	Ecl+SVN	8.56		3.00	13.75	2.83	
		Ecl+Replay	2.61	-69.50%	1.00	4.67	1.20	0.000
	Task 4	Ecl+SVN	6.11		1.67	10.75	2.45	
		Ecl+Replay	7.91	+29.45%	2.75	16.42	2.85	0.038
	Task 5	Ecl+SVN	17.85		11.00	23.92	3.75	
		Ecl+Replay	19.97	+11.88%	14.00	25.42	2.88	0.101
Correct-ness (points)	**Task 1**	Ecl+SVN	0.68		0.00	1.00	0.44	
		Ecl+Replay	1.00	+47.05%	1.00	1.00	–	0.002
	Task 2	Ecl+SVN	0.82		0.00	1.00	0.31	
		Ecl+Replay	0.90	+9.76%	0.33	1.00	0.19	0.449
	Task 3	Ecl+SVN	0.80		0.00	1.00	0.37	
		Ecl+Replay	1.00	+25.00%	1.00	1.00	–	0.019
	Task 4	Ecl+SVN	0.70		0.00	1.00	0.47	
		Ecl+Replay	0.70	–	0.00	1.00	0.47	1.000
	Task 5	Ecl+SVN	0.53		0.00	1.00	0.44	
		Ecl+Replay	0.51	-3.77%	0.00	1.00	0.45	1.000

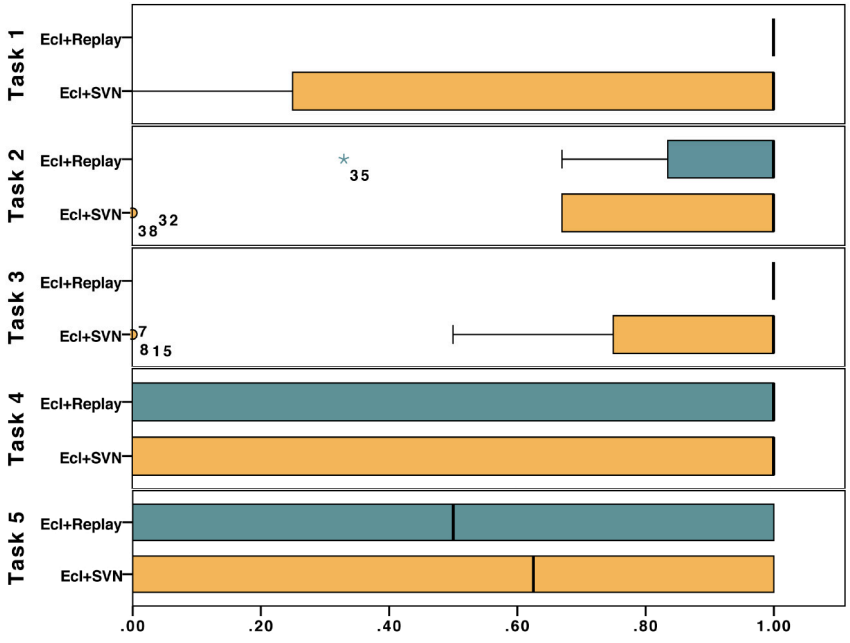
Task 3 – Finding experts at the class level. In this task, the subjects were asked to identify experts for a class based on the recent changes it had undergone. The results are very similar to those from task 1, and show that Replay outperforms the baseline for this task, with the results being statistically significant.

Task 4 – Relating a feature to code changes. This task involved identifying the different features inside a class and identifying the one that had changed the most. The experimental group took, on average, more time than the control group to complete the task, but had equal effectiveness. The results for completion time are statistically significant, validating the better performance of the control group. A possible explanation for this result is that number of fine-grained changes the subjects from the experimental group had to look at was higher than the number of SVN commits the control group had to inspect.

Task 5 – Tracking back the introduction of a defect. This task was added to the experiment after collecting suggestions from the pilot study which indicated that Replay could be useful for identifying the change from which a defect originated. The control group performed better than the experimental group in terms of completion time, and the two groups had similar performances with regard to correctness. The results and the feedback collected from the experimental group indicate that a potential reason that they took longer than the control group was because there were too many fine-grained changes to inspect for this task, which ended up being counterproductive. However, because the results for completion time are not statistically significant, we cannot affirm that the control group outperformed the experimental group.



(a) Completion time



(b) Correctness

Figure 8.6. Box plots of completion time and correctness per task

Task 6 – Understanding the rationale behind past refactorings. The goal of this task was to understand the design decisions behind a major refactoring performed in the past. Since this was a task requiring a descriptive answer, we do not analyze any quantitative data for it in our results. Furthermore, we did not impose a time limit for this task, so the results for completion time have a high variability.

We have qualitatively graded the answers, shown in Figure 8.7. The essays confirm that the control group was able to better understand the reasons behind the refactoring, with the majority of the answers considered "very satisfactory" being from them. The answer below exemplifies such an answer given by a control group subject:

"The functionality of Sheet was moved to SpreadSheetModel.java. It contains now the cell-grid. Before it only stored the cell contents as String rather than the cell objects. For that all other classes that referred to Sheet before had to be changed: e.g. from "Sheet sheet" to "SpreadSheetModel sheet" in SpreadSheetWriter.java." (C13).

It shows that the subject understood more than just the immediate goal of the refactoring, which was to replace the functionality of `Sheet` in `SpreadSheetModel`. The subject also understood the design decision behind the refactoring, and how this change affected the system's related classes. Similarly, the answer below shows a very satisfactory answer from a subject of the experimental group:

"Before refactoring, the classes Sheet and SpreadsheetModel were separated, and one could convert a Sheet to a SpreadsheetModel by calling a function. The refactoring eliminated the need for these two separated classes, by merging the functionalities of both only in one of them (SpreadsheetModel). I guess it was done to keep consistency of the state of these entities, which were treated separately, but their state depended on each other." (E14).

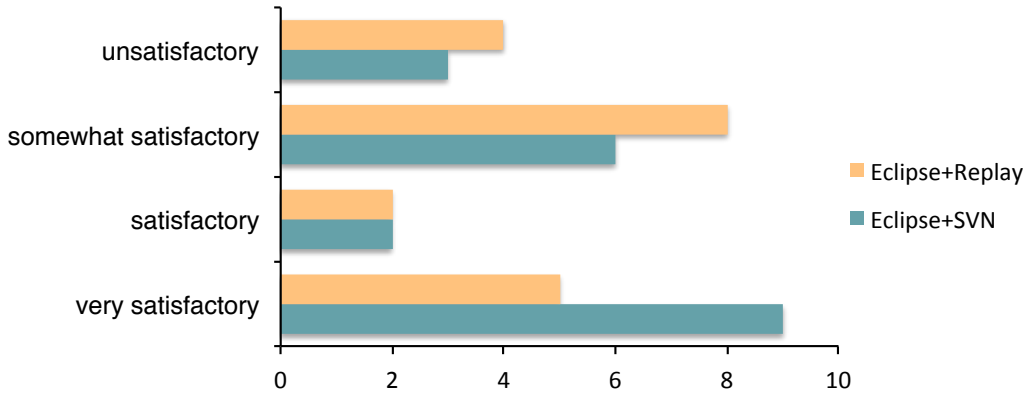


Figure 8.7. Grading of the answers to task 6

The answers classified as satisfactory and somewhat satisfactory are those in which the subjects identified the classes directly involved in the refactoring (`Sheet` and `SpreadSheetModel`), but did not understand why the refactoring happened. The major differences between these two grades were that somewhat satisfactory answers also identified classes that were indirectly

involved in the refactoring and somewhat satisfactory answers expressed doubts about the refactoring. An example of a satisfactory and a somewhat satisfactory answer are shown below:

"Sheet and SpreadSheetModel were essentially duplicates. So the former was deleted and replaced by the latter. The code from Sheet was merged into SpreadSheetModel. MainFrame changed a lot, and as a result, the model classes had to be modified. That's how the duplication received attention." (E7 - satisfactory).

"In the Sheet class, the convert to SpreadSheetModel has been changed multiple times. SpreadSheetModel and SpreadSheet class have been changed. I think the reason was because of the way sheets were kept in an array and converting them to a SpreadSheet. I did not understand very well." (E12 - somewhat satisfactory).

Finally, the unsatisfactory answers are those that failed to identify the classes directly involved in the refactoring, as in the example below:

"The refactoring seems to include the solution of more comments on the classes (i.e., the authors). Sheet has been replaced by SpreadSheetSelectionModel." (C15).

As previously stated, the results from Figure 8.7 and the content of the essays suggest that subjects from the experimental group had more difficulty understanding the refactoring than those of the control group. Further evidence comes from the perceived difficulty level of the task collected in the debriefing and shown in Figure 8.1. This data indicates that the experimental group struggled to understand the refactoring (the average difficulty was higher than 4 - difficult), while the control group found the task to be easier (the average difficulty was 3 - intermediate).

Like with task 5, the experimental group complained that the information provided by Replay was too fine-grained to allow them to see the "big picture" of the refactoring.

Summary. For tasks 1 and 3, which required fine-grained information about recent changes, Replay was more efficient and effective than the baseline. For task 2, when the subjects had to relate recent changes to authors, there was a tendency for Replay to be more effective than the baseline. For task 4, in which measuring the amount of work done in different parts of the code was important, Replay proved to be less efficient but just as effective as the baseline. In task 5, which required analyzing information over a long time span, there was a tendency for Replay's fine granularity to slow the subjects down in comparison to the baseline. To overcome this potential issue, as part of our future work, we plan to add a feature to Replay that offers the possibility to aggregate fine-grained changes using a customizable aggregation factor.

8.4.7 Subjects' Feedback

In the debriefing questionnaire we asked the subjects from the experimental group to leave comments and suggestions about the Replay plug-in. In this section we summarize their feedback, which can be classified into three groups: (i) defects or flaws in the plug-in; (ii) suggestions strongly related to the tasks; (iii) suggestions to improve the tool.

Flaws and defects. In general, the defects or flaws detected by the subjects impaired the user experience, but did not impede their performance on the tasks. One such flaw that disturbed the subjects was the fact that sometimes, when they were browsing through the changes in the view, the tool appeared to freeze. When this happened, the tool was actually requesting data to display in the view, but the response time was too long for the user's perception. A solution for this would be the addition of a progress bar, which would indicate for the user's benefit that the process will take a few seconds. A second flaw was that the tool did not allow subjects to open multiple editors. This was a design decision taken by the authors to allow the simulation of a video replay feature by always showing the sequences of changes in the same editor. However, upon practical usage of the plug-in it became clear that opening multiple editors could be useful in situations that require the comparison of non-subsequent changes.

Defects detected by the subjects included: when switching between editors (Replay and Compare), the tool failed to focus on the newly selected editor; and some descriptions of the changes did not correspond to the actual change. The latter is actually a defect of the tool responsible for capturing the changes, which is only shown when we replay them. The first defect was detected during the first experimental runs, and was fixed for later runs.

Suggestions strongly related to the tasks. Some suggestions were strongly tied to the tasks and if addressed would have helped the subjects in their search for the solution. However, they would bring little benefit to the tool. For instance, one subject suggested adding the ability to save snapshots of the view so that data would not have to be re-requested. This could instead be implemented as a history of recent requests to allow for quicker selection of identical requests. Another example of a task-related suggestion was one request that the Replay editor be made editable to allow for in-place code change (and specifically bug fixing for task 5). However, the main idea behind the Replay editor is to let users view the history of changes, but not to change this history. This is the reason why the Replay editor is non-editable.

Suggestions to improve the tool. The subjects provided a rich list of suggestions to improve the tool and make it more usable. We will comment on them by separating the suggestions per component.

In the Replay view, the most common suggestion was to provide a search bar to facilitate the search for a specific change. To be more useful, the search should include the content of the changes, so users can, for instance, search for a specific method name. Another request was for a navigation mode that would let the user navigate through a specific developer's changes while still showing the changes of the other developers in the list. Two more important requests point in opposite directions. Some subjects suggested adding a feature (*e.g.* a time slider control) to allow for the grouping of changes, because in some instances the changes were too fine-grained. The second request was to show even finer-grained changes, to the level of method, to facilitate identification of what has changed between subsequent versions of a class.

For the filters, the most common request was for the tool to memorize the previous dates selected by the user and add a reset button to reset the date at the user's will. In addition, the subjects requested a search bar to more quickly search for specific classes and users. Users also suggested keeping a history of recent searches to facilitate the loading of similar requests.

The only suggestion for the Replay Editor was to add markers onto the ruler to indicate the lines that the changes are in. This would facilitate navigation when multiple changes are present.

8.5 Threats to Validity

In this section we discuss the threats to construct, internal, and external validity. Threats to construct validity concern the generalization of the result of the experiment to the concept or theory behind the experiment [Wohl 00]. Threats to internal validity refer to influences that may have affected the independent variable with respect to causality, without the researcher's knowledge [Wohl 00]. Threats to external validity are conditions that limit our ability to generalize the results of our experiment [Wohl 00].

8.5.1 Construct Validity

Measurement of the dependent variables. We did not differentiate between the completion time of those who answered correctly and those who did not. It is possible that those who did not find the answer to a question used all the time available for that task before going to the next task, thus skewing the time measurement to a higher value. We limited the time available for each task with the goal of minimizing this issue. However, it might have occurred that subjects who answered wrongly gave up quickly and moved to the next task. We inspected the answer sheets seeking for such a pattern and only encountered one clear instance in which it occurred, eliminating this subject from the analysis.

Co-factors. We explicitly modeled the experience level of the subjects as a controlled variable for the analysis of RQ3. However, we might have not considered all the relevant co-factors that influenced the results. Some of the co-factors we did not consider that may have influenced the results are the details of the subjects' backgrounds (whether the subjects have experience in the industry), the settings of the laboratories (different laboratory configurations per run, such as number of participants and different computers), and the subjects' attitudes towards learning (whether the subject looked for help when faced with difficulties with the tool).

Power analysis. One weakness of our experimental design is the fact that we did not calculate the minimum sample size required to detect significant results before running the experiment.

Subject behavior towards the experiment. Some of the subjects were curious about the purpose of the study. To avoid having their knowledge of the study influencing their behavior during the experiment, we made ourselves available to explain the study and experiment design after a subject's experiment run. Some subjects demonstrated anxiety about being evaluated. To mitigate this effect, we assured them that the evaluation process was anonymized throughout.

Experimenter expectations. A couple of actions were taken to mitigate the experimenter's expectations of the results. First, the oracle containing the answers to the tasks was prepared before the start of the experimental runs. Then, during the experimental runs, the experimenter refrained from looking at the handouts as the participants returned them. A factor that might have negatively affected the analysis of RQ4 is that only one experimenter analyzed the subjective answers.

8.5.2 Internal Validity

Subjects. To reduce the threat that the subjects may not have been competent enough to accomplish the tasks, we ensured that they had sufficient skills with the tools used during the experiment. To lessen the threat that the subjects' expertise may not have been fairly distributed, we used randomization and blocking to assign treatments to subjects.

Tasks. The tasks were designed by the authors of this paper, and thus may have been biased in Replay's favor. To mitigate this threat, we have based the tasks on valid questions that developers ask, which were reported in previous catalogues. The tasks might have been too difficult and the allotted time per task may have been insufficient. To alleviate these threats, we conducted several pilot runs to fine-tune them. Furthermore, the task that was classified as too difficult by the experimental group (task 6) was not designed to be included in the statistical analysis.

Experimental runs. There were several runs and the differences between them may have influenced the results. However, having several pilot runs with different numbers of participants allowed us to find a stable and reliable experimental setup.

Training. We provided a training session on Replay to all subjects of the experimental group, while the subjects from the control group were assumed to have familiarity with the baseline tools. To mitigate the fact that lack of proper training might have influenced the results, we also provided a training session on Subclipse whenever a subject from the control group was unfamiliar with the tool.

Learning effect. To neutralize the learning effect on the tasks, we should have randomized the order of the tasks for each subject. Instead, we sorted the tasks in increasing order of difficulty to allow the subjects to build confidence as they cleared the tasks. As a consequence, we might have enforced the learning effect on the last tasks. Because the questions asked in each task were fundamentally different – they had different goals, and required the use of different features of the tools – the strongest potential learning effect regards the usage of the tool, but not to its specific features. We minimized this threat by: (i) giving subjects the time to explore and get familiar with the tool before the start of the experiment; and (ii) asking the subjects to answer a warm-up question before moving to the actual task. A typical learning curve is steep at the beginning, and gradually evens out as a person gets familiar with the subject, so we aimed to give the subjects enough experience with the tool that the learning curve had begun to even out before they started the tasks.

8.5.3 External Validity

Subjects. The fact that the subjects of the experiment were from academia may have limited our ability to generalize the results to the industrial environment. It is difficult to recruit practitioners who are willing to dedicate two hours of their time to do an experiment. To mitigate the lack of practitioners, we assume a relatively high average expertise level of the 40 selected participants. This assumption is sustained by the subjective assessment of the expertise provided by the subjects prior to the experiment. They were asked to rank their perceived knowledge according to the following scale: 1–none, 2–beginner, 3–knowledgeable, 4–advanced, and 5–expert. The

results—Java (avg. 3.65, stdev. 1.05), Eclipse (avg. 3.45, stdev. 0.90), SVN (avg. 2.90, stdev. 1.19)—indicate an average of knowledgeable subjects.

Tasks. Our choice of tasks may not reflect real questions related to software evolution. This threat was neutralized by our reliance on existing catalogues of questions [Sill 06; Alwi 08; Frit 10a; Ko 07], which were mainly constructed through surveys and interviews with practitioners, to elaborate the tasks.

Object system. One important threat is whether our object system is sufficiently representative, because it is a small system that was developed by undergraduate students. Therefore, it may not reflect the complexity of large-scale industrial systems. The use of more than one object system may have yielded different or more reliable results. However, the choice of the object system was constrained by the need for a project whose change history had been recorded by both Syde and SVN.

8.6 Intermezzo: Application of Replay in Education

Our work on replaying fine-grained changes makes a side contribution to the field in the way it may be applied to the context of education [Hatt 11b].

Much of human learning is built on observing, retaining, and replicating behavior witnessed from a model. With this basis, instructors often teach informatics by providing programming examples to be observed and analyzed by learners. By remembering and replicating the steps leading to the final artifacts, students learn.

However, due to resource constraints, professors usually illustrate an example program only once and provide only its finalized version to students. This hinders the students' need for repeated observation and replication.

We adapted the Replay plug-in to overcome the limitations of the current approach to learning by example. Replay records every code edit of a programming session, making it available as an interactive executable "tape". Professors can accurately design the steps of an example, and "play" them as live sessions in class, free of the usual burden of concurrent coding and explaining; students have the complete code history available to them, and can observe it repeatedly, or interact with it at any moment.

The Replay-edu toolkit, illustrated in Figure 8.8, is composed of two Eclipse plug-ins:

1. **Replay Recorder** is an adaptation of Syde's Inspector Section 4.3 (p.42) that locally records a snapshot of the source code every time a file is built, generating an *executable code tape* of the programming session. In addition to the playable change history, the user can enter descriptive information about the development session (see Figure 8.8), and attach annotations to saved changes, functions that would allow professors to add explanations to complex changes.

2. **Replay Player** lets the user load and play a tape. The Player lists every step, and the learner can (1) play and watch the code evolve, (2) pause the replay, and (3) step through the items in the list.

We argue that *Learning by Replaying* helps the learning process in various ways. Learners have access to the source code creation process, where each step is a reminder of the professor's

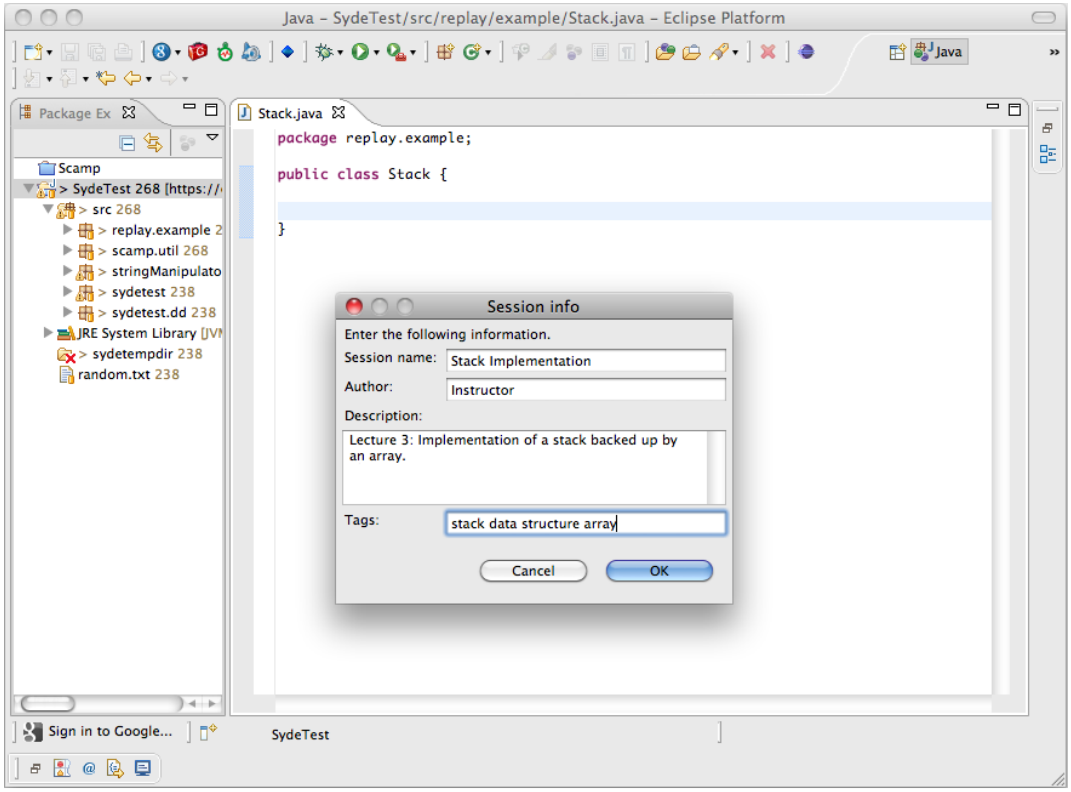


Figure 8.8. Screenshot of Replay-edu showing the creation of a development session

explanation. They can dynamically interact with the code and the whole context at any stage of the development session.

8.7 Limitations and Future Work

The limitations of the controlled experiment are discussed in the threats to validity section (Section 8.5 (p.170)). Therefore, in this section we focus discussing the limitations of the Replay application and future work that may be done to improve it.

Configurable granularity of replayed changes. One of the limitations of Replay that prevented it from outperforming the baseline when answering questions related to analyzing changes over a long time span is the fine granularity level of the replayed changes. To overcome this issue, we plan to add a time slider to overturn control of change aggregation to the users. This feature will work by aggregating the changes that happened within a window of time, but only showing the delta between the first and last versions of the artifacts under analysis to eliminate repetitive and redundant changes.

Improvements on the navigation of changes. The subjects of the experiment indicated a few missing features in the Replay view with the potential to improve the plug-in's usability. The

first one is a search bar, which would help users who are searching for specific information. The searched content should include not only the metadata of the change that is shown on the view, but also the actual source code. Another potential improvement of the view would be the addition of a navigation mode allowing the user to navigate through a specific developer's changes while still showing the changes of other developers in the view's list. The navigation mode will allow the user to focus on watching a single person's development session, yet be simultaneously aware of who else was changing which software artifacts at the same time.

Enhancement of assistance through automated answers. Theoretically, some common questions could be automatically answered by our tool without the need for a user to investigate the code history. Some examples might be "How much work have people done?", which can be calculated based on the amount of Syde changes each person has performed within a period, and "Who changed this code?", which can also be automatically listed by our tool. A future undertaking in this direction would be to identify these questions, survey developers to rank their popularity, and build a recommendation system able to automatically answer them.

8.8 Summary

In this chapter, we presented Replay, a tool that allows developers to explore the evolution history of a system by chronologically replaying the fine-grained changes collected by Syde. We argue that Replay can be useful to help developers find answers to questions that may be raised during the development and maintenance that are related to the evolution of a system.

We conducted a controlled experiment to evaluate whether Replay is at least as effective and efficient as the current state of the practice at supporting developer questions related to software evolution. The results indicate that Replay leads to an improvement in both correctness (16.76%) and completion time (11.56%), with the latter being statistically significant at a 99% confidence interval. As an indication of a superior performance of the experimental group in terms of correctness, 75% of this group performed better than or equal to 50% of the control group. In terms of completion time, 50% of the experimental group was faster than (or equivalent to) 25% of the control group. These results show that there are benefits to using Replay over the state of the practice tools for most of the tasks included in this empirical evaluation.

The per-task analysis of the results provided a number of insights on the type of tasks our approach supports best. For tasks that required fine-grained change information, or for which the inspection of recent changes was important, Replay outperformed the baseline. However, when the tasks required a high-level overview of the changes, Replay did not perform better than the baseline.

We demonstrated the versatility of a tool able to replay the fine-grained change history of a system by adapting the Replay plug-in to be used for education purposes. In this context, we envision professors using the Replay Recorder to record and annotate a coding session to be both used in class and distributed to students. Students can play the tape and watch the code evolve, stop it whenever needed, and reach a deeper understanding of how to bring a program from one state to another, thus experiencing an enhanced learning process.

In the last two chapters, we have leveraged Syde's change history to assist developers in acquiring knowledge related to a project's evolution. The applications provide different and complementary assistance to developers, and open up a new path for building recommendation

systems based on a fine-grained change history. We believe that the knowledge acquired through these assistance tools, together with an increase in workspace awareness, can effectively help developers to better collaborate, especially in distributed teams.

Part IV

Epilogue

Chapter 9

Conclusion

Software development is inherently a team-oriented activity. For the past 60 years, the software industry has mainly built software in a collocated setting, where teams of developers share the workplace. There have been great advances made with process models, tools, methodologies, and techniques to support software development in a collocated setting. However, since the beginning of the 21st century, this scenario has begun to change, with an ever-increasing number of software companies adopting global software development as an alternative to cut costs and speed up the development process.

Global software development comes with several challenges for building quality software, ranging from the adaptation of current methods, techniques, tools, *etc.*; to managing new challenges imposed by the distributed setting, including physical and cultural distance between teams, communication problems, and breakdowns in coordination.

In our dissertation, we focus on one specific challenge for global software development: the improvement of collaboration within development teams, especially within those that are geographically dispersed. Our thesis is that by modeling the evolution of a software system in terms of fine-grained changes, we can produce a detailed history that may be used to help developers collaborate. To validate this claim, we first presented the Collaboration Change-based Software Evolution model, which accurately represents the evolution of a system as sequences of fine-grained changes. We then built a tool infrastructure able to capture and store fine-grained changes for both immediate and later use.

In further validation of our thesis, we evaluated four applications with the intent to improve real-time awareness within teams and of providing assistance for information acquisition. We used several methodologies for this evaluation: case studies, a qualitative user study, and a controlled experiment.

9.1 Contributions

In this dissertation, we have created and leveraged a fine-grained change history for a software system with the intent to improve team collaboration. The series of contributions we have made to the state of the art are summarized below.

9.1.1 Models and Tools

Upon reviewing the state of the art in mining software repositories, we discovered that the data from mainstream SCM systems are too coarse-grained to be used effectively for the improvement of team collaboration. To rectify this, we proposed a new model and a tool to track the fine-grained changes of a software system and to store them to gather a detailed history of the system's evolution.

The definition of the Collaborative Change-based Software Evolution model. Our thesis first contributes to the field with the creation of a model representing the detailed evolution of a shared software system in terms of fine-grained source code changes. The Collaborative Change-based Software Evolution (CCBSE) model applies the reification principle to changes from object-oriented systems, forming a detailed history of their evolution. It models object-oriented systems as abstract syntax trees (AST) where nodes are software constructs (*e.g.* packages, classes, and methods), and changes are tree operations that are applied to the AST to take the model of the system from one state to next.

The concept of representing a system's evolution as a sequence of fine-grained changes is not new [Zumk 07; Ebra 07; Robb 08a], and has been explored for a number of purposes, such as to improve code completion [Robb 10a], and to ease feature composition [Ebra 07]. The innovation of our work is in its extension of an existing model, the CBSE model [Robb 08a], to represent a system developed in a team by making developers an explicit entity of the model.

The creation of the Syde toolset. We implemented a collection of tools to support the CCBSE approach of storing the detailed evolution of the system and using it to help developers with different tasks. Syde is a client-server application whose server is responsible for receiving, managing and storing changes to a software system performed by developers. The client has two distinct responsibilities: tracking the changes made in the IDE, and encapsulating the applications to help developers collaborate. We built four applications as Eclipse plug-ins:

- *Scamp* delivers workspace awareness information. In real time, it shows the developers that are actively working on the system and which parts they are changing, as well as displaying which developers are experts on specific system artifacts.
- *Conflicts* is a plug-in that alerts developers of potential merge conflicts that might be arising due to concurrent modifications to a software artifact.
- *Manhattan* shows both workspace awareness information and potential merge conflicts through a visual representation of an evolving system based on the metaphor of a city.
- *Replay* allows developers to investigate past development sessions by replaying the recorded change history. Its purpose is to help developers search for answers to any common questions they have that are related to the system's evolution.

9.1.2 Applications and Techniques

To evaluate our thesis that a detailed history of a system's evolution can be effectively used to support team collaboration, we built four applications that pursue two specific goals. The first of these goals is to increase the awareness of a development team by broadcasting relevant information on the current activity of the team in real time. The second is to explore the fine-grained change history to assist developers in finding some of the information needed to conduct their work. Each of the four applications we created is summarized below.

The application of the CCBSE approach to awareness of team activity. We proposed a set of visualizations, integrated into the IDE, to deliver information about the team's activity in real time. We defined three visualizations: the WordCloud view, which shows what classes have been changed recently, and by whom; the Buckets view, which shows the number of changes each developer has performed on each class; and decorations in Eclipse's package explorer, which work as visual cues to indicate which parts of the system are changing. The visualizations always show the most recent activity by updating themselves in a non-intrusive way.

The case study that we conducted indicated that developers benefited from the increased level of awareness. They were able to avoid duplicating work and reduce the number of merge conflicts. They were also encouraged to communicate more frequently with their colleagues because they could see what the others were doing.

The application of the CCBSE approach to detect emerging conflicts. We defined a technique for detecting potential conflicts that might emerge due to concurrent modification to source code elements, and for notifying developers about their existence before check-in time. This information allows developers to take preventive measures to avoid complex merging at check-in time. In addition, we devised different ways to deliver this information in the Eclipse IDE, including the enrichment of the Java editor with visual cues, and the use of a graph and a list representation to construct different views.

We conducted a qualitative user study to investigate whether developers' behavior changes when they are exposed to preemptive conflict detection. We observed some change in their behavior, and collected evidence that these changes are beneficial: communication and coordination becomes more effective, and there is a higher rate of successfully resolved conflicts. In addition, we believe that the different ways we provide to visualize this information are complementary to one another, because developers have different preferences. The fact that they can choose between different options might help them in the adoption process.

The definition of two measurements of code expertise based on fine-grained changes. The first measurement we present takes into account the number of small changes performed by a developer on a source code entity in the IDE. Because it takes into account every edit interaction a developer has with a source code entity, this measurement more precisely reflects the effort that a developer spent working on the entity than previous measurements based only on commits. The second measurement combines the number of changes performed by a developer with the natural effect of forgetting to compute code expertise.

The results of the case studies show evidence that the measurements we propose are able to classify expertise more accurately than the one based on CVS history logs [Girb 05c],

especially when there is a high degree of variation in the frequency with which active developers check in code. In addition, the results suggest that the incorporation of the notion of memory loss when measuring expertise reflects a more realistic scenario than assuming a developer remembers everything regardless of the time that has passed. However, it is important to emphasize the subjective nature of forgetting, and that the ideal rate of forgetting for each project is subject to its own individual characteristics.

The application of the CCBSE approach to understanding of software evolution. We developed an application that allows developers to explore the rich history of changes created with the use of the CCBSE approach. By investigating past changes, developers acquire more knowledge to answer different program comprehension questions related to the evolution of the system.

The results of our controlled experiment to evaluate Replay showed that it led to an improvement in both correctness (16.76%) and completion time (11.56%), with the latter being statistically significant at a 99% confidence interval. The results also provided a number of insights on the type of tasks our approach supports best. For tasks that required fine-grained change information, or for which the inspection of recent changes was important, Replay outperformed the baseline. However, when the tasks required a high-level overview of the changes, Replay did not perform better than the baseline. As an orthogonal contribution, we demonstrated the versatility of a tool able to replay the fine-grained change history of a system by adapting the Replay plug-in to be used for educational purposes.

9.1.3 Evaluations

We used several methodologies to evaluate our thesis, including case studies, a qualitative user study, and a controlled experiment. For the sake of completeness and replicability, in the appendix of this dissertation we publish the experimental data of the user study and the controlled experiment.

The empirical validation of our conflict detection application through a user study. We designed and conducted a qualitative user study to understand how developer behavior is influenced by the presence of preemptive conflict detection. We reported on the design and the analysis of a complex amount of data collected from several different sources (questionnaires, observation, interview, documents), and published the experimental data to facilitate the execution of similar studies.

The empirical validation of our replay application through a controlled experiment. We designed a controlled experiment for the empirical validation of our Replay application and conducted it with participants with diverse educational and technical backgrounds. We provided a full report on the design, operation, and analysis of the data from this experiment. Moreover, we published the entire experimental data set and everything required to make the experiment replicable and transparent.

9.2 Limitations and Future Work

Over the course of this dissertation, we identified promising future research directions. Some of them are directions that address the limitations of our approach. In the following, we outline

potential future work and discuss the shortcomings of our work from which some of them originated. Since the limitations and future work of the applications we proposed have been discussed in prior sections, here we take a broader perspective, with a focus on the future work related to our approach.

Addressing scalability issues of the current approach. The CCBSE approach models the evolution of a system as a set of sequences of change operations applied to the AST representation of a system. Each sequence of operations corresponds to the changes performed by one developer in his workspace. This means that at a specific point in time, the current state of a system is described as a set of ASTs, each representing the current state of the system in each developer's workspace. Our server application maintains the current state of the system in memory to facilitate updating and the detection of conflicts. The space required to host the system is $O(de)$, where d is the number of developers, and e is the number of entities in the system. As a concrete example, the project Checkstyle, used in the evaluation of emerging conflicts (see Chapter 6.4 (p.84)), contains 341 classes and 22 packages. Its AST contains approximately 3,300 nodes, and occupies around 6MB of space. The polynomial growth that occurs as new developers are added can reach critical levels, though to this date we have not experienced memory overflow on Syde's server.

One solution to this limitation would be to reduce the number of ASTs kept in memory to represent the state of the workspaces of the active developers only. Another solution would be to migrate Syde's server to a distributed architecture, where each node of the server maintains the ASTs of a sub-team.

Tailoring information to developers' needs. At the moment, Syde broadcasts all awareness information (notifications about who is changing which parts of the system) to all clients, who are individually responsible for filtering out this information. While this design decision gives more flexibility to the applications on the client side, it can be a bottleneck point in the system. For future work, we suggest to adding an application that controls the flow of information between the server and the clients. This application would contain configuration options to tailor the broadcasting of information according to the developers' needs. Some examples of potential configuration options would be to receive change notifications only from a developer's sub-team, to receive notifications of changes to interdependent code (to avoid ripple effect), or to receive notifications of changes in code that a developer owns.

Fine-tuning the granularity of system and change representations. The CCBSE model currently represents a system's constructs down to method and instance variable level. It is possible to extend it to model statements inside a method. One benefit of explicitly modeling statements would be for the simplification of the conflict detection algorithm, which currently has to apply textual differencing to a method's content to detect conflicts inside it. However, it should be evaluated whether the overhead caused by modeling a system down to statement level is worth its adoption. Future work could also be done on the representation of changes to define more composite changes and exploit the benefits of having this information. We can define metrics to automatically characterize development sessions in addition to refactorings, similar to the work of Robbes and Lanza [Robb 07b].

Applying the approach to other programming languages and IDEs. Our tools target Java systems developed with the Eclipse IDE. It is a proof of concept, demonstrating the feasibility

and the usefulness of modeling the evolution of an object-oriented system as fine-grained changes. Future work could be done to extend the implementation of our approach to incorporate other object-oriented languages and to integrate it with other IDEs. The extension of the model to other programming languages would require the generalization of our CCBSE model, which is currently specialized to represent some of the particularities of Java systems (*e.g.* a class can only have one superclass, which is not the case in C++).

Building recommendation systems. The applications to assist developers that we devised in the course of this work are just "the tip of the iceberg". There is a plethora of recommendation systems capable of leveraging the data provided by the CCBSE approach. We discuss a few of them.

Change couplings refer to classes of method that often change together, independently of any syntactic dependencies. Several researchers have proposed change coupling measurements based on SCM commits[DAmb 06a; Flur 06; Zhou 08], and a recommendation system also based on SCM commits has been proposed [Zimm 04]. This application shows a developer changing a software artifact a list of other artifacts that had been changed along with the current one in the past, suggesting that they may also need to be changed. This application, however, proved to require a long history of the analyzed system in order for it to give meaningful suggestions. With the fine-grained history produced by our approach, we can build a recommendation system able to give meaningful suggestions at a much earlier stage. A series of change coupling measurements from the type of data generated by the CCBSE approach have been proposed by Robbes *et al.* [Robb 08c]. Future work would be to implement them into a recommendation system to evaluate its usefulness in suggesting related changes.

Assuming we have defined more composite operations than the ones currently implemented, and have metrics to characterize development sessions, we can extract behavior patterns based on sequences of operations. Examples of these patterns might be a sequence of changes that tends to take the system to an error state, a pattern showing evidence that a developer is struggling with the code, or a pattern that leads to design flaws. A recommendation system based on these patterns would alert a developer when his changes are taking him to unwanted states, and could possibly suggest solutions.

Another recommendation system, discussed in Chapter 8.7 (p.173), would be an application able to automatically answer common questions related to the evolution of a system. Questions such as "How much work has someone done?" could be calculated based on the amount of Syde changes a person has performed within a period. Such a recommendation system would have the potential to save a developer time in understanding the system.

Continuous integration. The final future work we envision would be to provide continuous integration, automating the build and test processes in the server. Our goal would be for the server to continuously build and test the system's most recently stable (with no compilation errors) state and report the problems to the development team. Continuous integration would only be possible if we had a mechanism to transform the model of the system, which is what the server stores, into actual source code. At the moment, because we also store the source code of each artifact, if should be possible to reconstruct the system. In addition, we would also have to solve the problem of combining each developer's sequence of changes to form the current global state of the system.

9.3 Closing Words

In this dissertation, we demonstrated that a detailed change history of a software system, produced by an accurate model of its evolution, can be leveraged to help developers to collaborate. Collaboration, however, is only one of the challenges that accompany global software development. Global Software Engineering is surging as a discipline of its own to address all the challenges imposed by distance, cultural differences, the loss of communication richness, and other unique characteristics of this development model. In addition, the change repository is not the only data source that can be exploited to aid software development activity. There is a current trend to leverage different repository types to build recommendation systems, assistance tools, and to personalize current IDEs to the individual needs of developers with the goal of improving the software development experience.

Our thesis tackles the problem of collaboration in globally dispersed teams, and opens up new perspectives and research directions to improve global software development.

Part V

Appendices

Appendix A

Data of User Study on Preemptive Conflict Detection

In this chapter we present the data of the qualitative user study on preemptive conflict detection, reported in Chapter 6 (p.77). This section contains the screening questionnaire and a sample of the handout used by the participants during the experimental session, as well as the debriefing questionnaire at the end. It also contains the complete answers to the two questionnaires.

A.1 Screening Questionnaire

Using Google Docs[1], we designed an online questionnaire that served both to provide an easily accessible platform for the participants to enroll, and to allow us to capture their personal information and information about their experience. The screening questionnaire is illustrated in Figure A.1 (p.190).

A.2 Handout

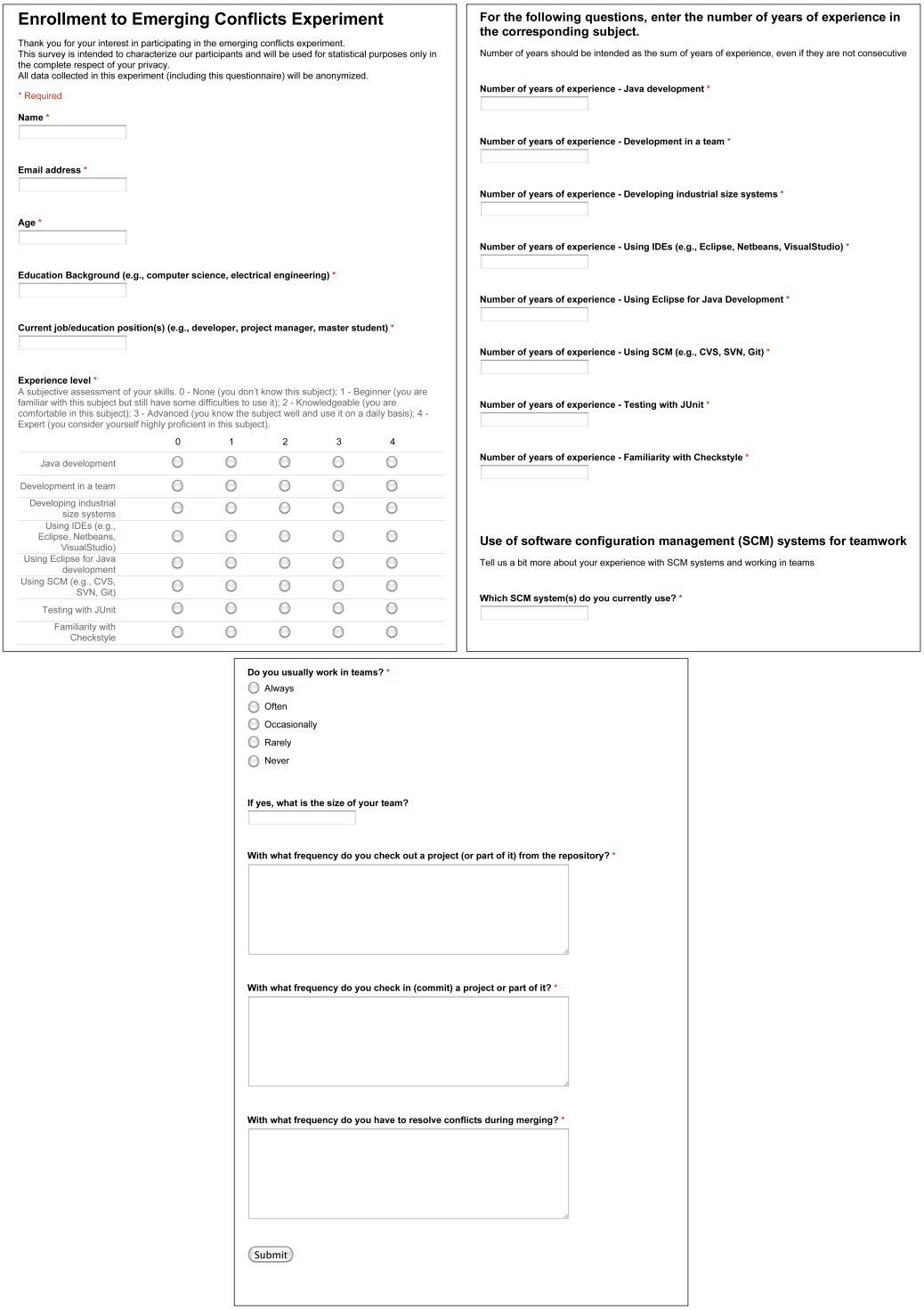
The participants were given a handout with instructions about the assignment and the tasks. Figure A.2 (p.191) and Figure A.3 (p.192) show a complete version of the handout. We used two versions of the handout that differed in their description of the tasks, consistent with the fact that the experiment was done by pairs of developers, and each had a set of different (but related) tasks to implement.

A.3 Data from Questionnaires

For the sake of transparency and repeatability, we provide copies of the participants' answers to the questionnaires.

Table A.1 (p.193), Table A.2 (p.194), and Table A.3 (p.195) contain the answers to the screening questionnaire, while Table A.4 (p.196) contains the answers to the debriefing questionnaire.

[1] `http://docs.google.com`



Enrollment to Emerging Conflicts Experiment

Thank you for your interest in participating in the emerging conflicts experiment.
This survey is intended to characterize our participants and will be used for statistical purposes only in the complete respect of your privacy.
All data collected in this experiment (including this questionnaire) will be anonymized.

* Required

Name *

Email address *

Age *

Education Background (e.g., computer science, electrical engineering) *

Current job/education position(s) (e.g., developer, project manager, master student) *

Experience level *
A subjective assessment of your skills. 0 - None (you don't know this subject); 1 - Beginner (you are familiar with this subject but still have some difficulties to use it); 2 - Knowledgeable (you are comfortable in this subject); 3 - Advanced (you know the subject well and use it on a daily basis); 4 - Expert (you consider yourself highly proficient in this subject).

	0	1	2	3	4
Java development					
Development in a team					
Developing industrial size systems					
Using IDEs (e.g., Eclipse, Netbeans, VisualStudio)					
Using Eclipse for Java development					
Using SCM (e.g., CVS, SVN, Git)					
Testing with JUnit					
Familiarity with Checkstyle					

For the following questions, enter the number of years of experience in the corresponding subject.

Number of years should be intended as the sum of years of experience, even if they are not consecutive

Number of years of experience - Java development *

Number of years of experience - Development in a team *

Number of years of experience - Developing industrial size systems *

Number of years of experience - Using IDEs (e.g., Eclipse, Netbeans, VisualStudio) *

Number of years of experience - Using Eclipse for Java Development *

Number of years of experience - Using SCM (e.g., CVS, SVN, Git) *

Number of years of experience - Testing with JUnit *

Number of years of experience - Familiarity with Checkstyle *

Use of software configuration management (SCM) systems for teamwork

Tell us a bit more about your experience with SCM systems and working in teams

Which SCM system(s) do you currently use? *

Do you usually work in teams? *
- Always
- Often
- Occasionally
- Rarely
- Never

If yes, what is the size of your team?

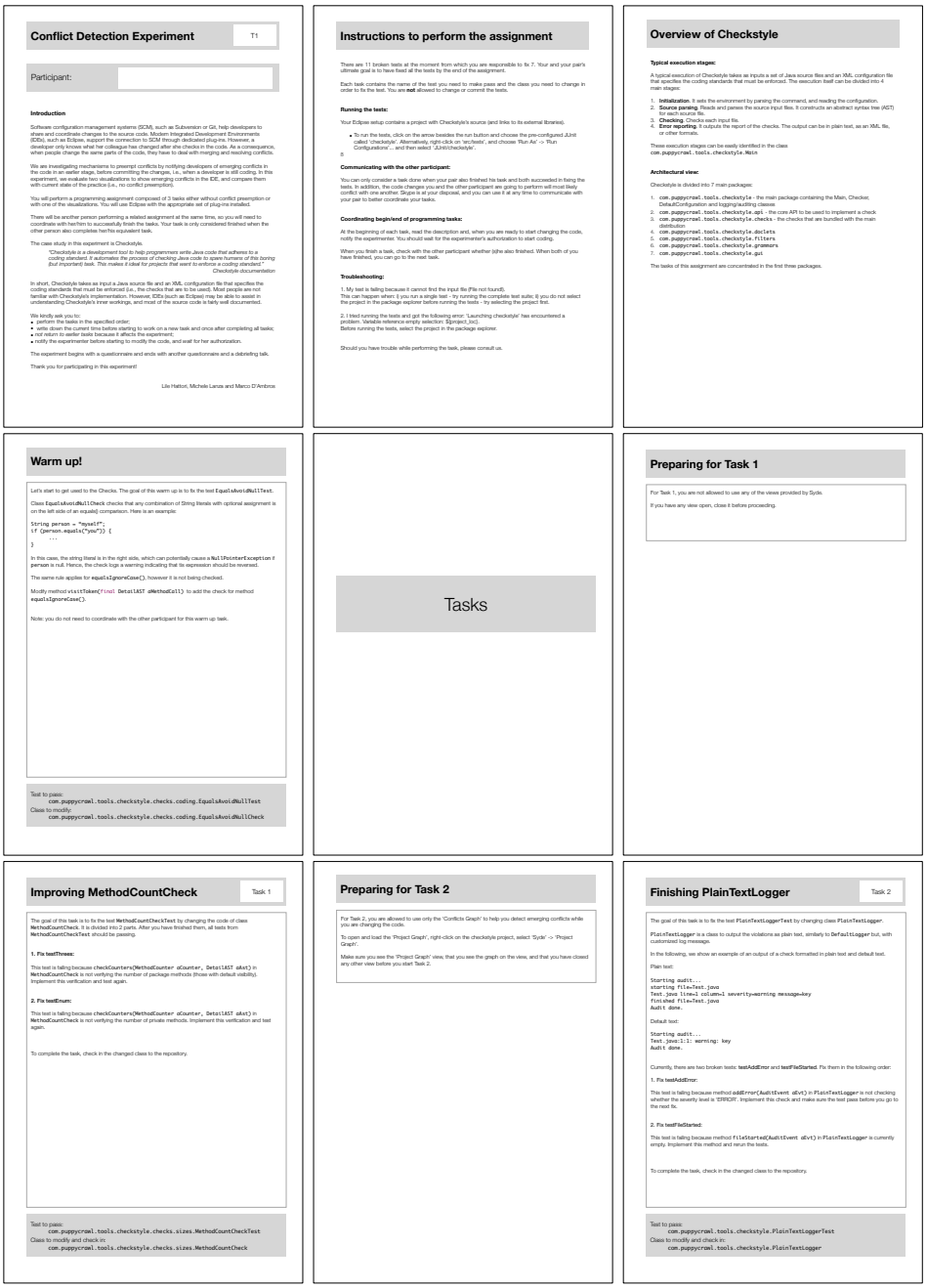
With what frequency do you check out a project (or part of it) from the repository? *

With what frequency do you check in (commit) a project or part of it? *

With what frequency do you have to resolve conflicts during merging? *

Submit

Figure A.1. The online screening questionnaire used to collect the participants' personal information and information about their experience



Conflict Detection Experiment

T1

Participant:

Introduction

Software configuration management systems (SCM), such as Subversion or Git, help developers to share and coordinate changes to the source code. Modern Integrated Development Environments (IDEs), such as Eclipse, support the connection to SCM through dedicated plug-ins. However, a developer only knows what her colleague has changed after she checks in the code. As a consequence, when people change the same parts of the code, they have to deal with merging and resolving conflicts.

We are investigating mechanisms to preempt conflicts by notifying developers of emerging conflicts in the code in an earlier stage, before committing the changes, i.e., when a developer is still coding. In this experiment, we evaluate two visualizations to show emerging conflicts in the IDE, and compare them with current state of the practice (i.e., no conflict preemption).

You will perform a programming assignment composed of 3 tasks either without conflict preemption or with one of the visualizations. You will use Eclipse with the appropriate set of plug-ins installed.

There will be another person performing a related assignment at the same time, so you will need to coordinate with her/him to successfully finish the tasks. Your task is only considered finished when the other person also completes her/his equivalent task.

The case study in this experiment is Checkstyle.

> *"Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard. It automates the process of checking Java code to spare humans of this boring (but important) task. This makes it ideal for projects that want to enforce a coding standard."*
> *Checkstyle documentation*

In short, Checkstyle takes as input a Java source file and an XML configuration file that specifies the coding standards that must be enforced (i.e., the checks that are to be used). Most people are not familiar with Checkstyle's implementation. However, IDEs (such as Eclipse) may be able to assist in understanding Checkstyle's inner workings, and most of the source code is fairly well documented.

We kindly ask you to:
- perform the tasks in the specified order;
- write down the current time before starting to work on a new task and once after completing all tasks;
- *not return to earlier tasks* because it affects the experiment;
- notify the experimenter before starting to modify the code, and wait for her authorization.

The experiment begins with a questionnaire and ends with another questionnaire and a debriefing talk.

Thank you for participating in this experiment!

Lile Hattori, Michele Lanza and Marco D'Ambros

Instructions to perform the assignment

There are 11 broken tests at the moment from which you are responsable to fix 7. Your and your pair's ultimate goal is to have fixed all the tests by the end of the assignment.

Each task contains the name of the test you need to make pass and the class you need to change in order to fix the test. You are **not** allowed to change or commit the tests.

Running the tests:

Your Eclipse setup contains a project with Checkstyle's source (and links to its external libraries).

- To run the tests, click on the arrow besides the run button and choose the pre-configured JUnit called 'checkstyle'. Alternatively, right-click on 'src/tests', and choose 'Run As' -> 'Run Configurations'... and then select 'JUnit/checkstyle'.

8

Communicating with the other participant:

You can only consider a task done when your pair also finished his task and both succeeded in fixing the tests. In addition, the code changes you and the other participant are going to perform will most likely conflict with one another. Skype is at your disposal, and you can use it at any time to communicate with your pair to better coordinate your tasks.

Coordinating begin/end of programming tasks:

At the beginning of each task, read the description and, when you are ready to start changing the code, notify the experimenter. You should wait for the experimenter's authorization to start coding.

When you finish a task, check with the other participant whether (s)he also finished. When both of you have finished, you can go to the next task.

Troubleshooting:

1. My test is failing because it cannot find the input file (File not found).
This can happen when: i) you run a single test - try running the complete test suite; ii) you do not select the project in the package explorer before running the tests - try selecting the project first.

2. I tried running the tests and got the following error: 'Launching checkstyle' has encountered a problem. Variable reference empty selection: ${project_loc}.
Before running the tests, select the project in the package explorer.

Should you have trouble while performing the task, please consult us.

Overview of Checkstyle

Typical execution stages:

A typical execution of Checkstyle takes as inputs a set of Java source files and an XML configuration file that specifies the coding standards that must be enforced. The execution itself can be divided into 4 main stages:

1. **Initialization.** It sets the environment by parsing the command, and reading the configuration.
2. **Source parsing.** Reads and parses the source input files. It constructs an abstract syntax tree (AST) for each source file.
3. **Checking.** Checks each input file.
4. **Error reporting.** It outputs the report of the checks. The output can be in plain text, as an XML file, or other formats.

These execution stages can be easily identified in the class `com.puppycrawl.tools.checkstyle.Main`

Architectural view:

Checkstyle is divided into 7 main packages:

1. `com.puppycrawl.tools.checkstyle` - the main package containing the Main, Checker, DefaultConfiguration and logging/auditing classes
2. `com.puppycrawl.tools.checkstyle.api` - the core API to be used to implement a check
3. `com.puppycrawl.tools.checkstyle.checks` - the checks that are bundled with the main distribution
4. `com.puppycrawl.tools.checkstyle.doclets`
5. `com.puppycrawl.tools.checkstyle.filters`
6. `com.puppycrawl.tools.checkstyle.grammars`
7. `com.puppycrawl.tools.checkstyle.gui`

The tasks of this assignment are concentrated in the first three packages.

Warm up!

Let's start to get used to the Checks. The goal of this warm up is to fix the test `EqualsAvoidNullTest`.

Class `EqualsAvoidNullCheck` checks that any combination of String literals with optional assignment is on the left side of an equals() comparison. Here is an example:

```
String person = "myself";
if (person.equals("you")) {
    ...
}
```

In this case, the string literal is in the right side, which can potentially cause a `NullPointerException` if `person` is null. Hence, the check logs a warning indicating that its expression should be reversed.

The same rule applies for `equalsIgnoreCase()`, however it is not being checked.

Modify method `visitToken(final DetailAST aMethodCall)` to add the check for method `equalsIgnoreCase()`.

Note: you do not need to coordinate with the other participant for this warm up task.

Test to pass:
`com.puppycrawl.tools.checkstyle.checks.coding.EqualsAvoidNullTest`
Class to modify:
`com.puppycrawl.tools.checkstyle.checks.coding.EqualsAvoidNullCheck`

Tasks

Preparing for Task 1

For Task 1, you are not allowed to use any of the views provided by Syde.

If you have any view open, close it before proceeding.

Improving MethodCountCheck

Task 1

The goal of this task is to fix the test `MethodCountCheckTest` by changing the code of class `MethodCountCheck`. It is divided into 2 parts. After you have finished them, all tests from `MethodCountCheckTest` should be passing.

1. Fix testThrows:

This test is failing because `checkCounters(MethodCounter aCounter, DetailAST aAst)` in `MethodCountCheck` is not verifying the number of package methods (those with default visibility). Implement this verification and test again.

2. Fix testEnum:

This test is failing because `checkCounters(MethodCounter aCounter, DetailAST aAst)` in `MethodCountCheck` is not verifying the number of private methods. Implement this verification and test again.

To complete the task, check in the changed class to the repository.

Test to pass:
`com.puppycrawl.tools.checkstyle.checks.sizes.MethodCountCheckTest`
Class to modify and check in:
`com.puppycrawl.tools.checkstyle.checks.sizes.MethodCountCheck`

Preparing for Task 2

For Task 2, you are allowed to use only the 'Conflicts Graph' to help you detect emerging conflicts while you are changing the code.

To open and load the 'Project Graph', right-click on the checkstyle project, select 'Syde' -> 'Project Graph'.

Make sure you see the 'Project Graph' view, that you see the graph on the view, and that you have closed any other view before you start Task 2.

Finishing PlainTextLogger

Task 2

The goal of this task is to fix the test `PlainTextLoggerTest` by changing class `PlainTextLogger`.

`PlainTextLogger` is a class to output the violations as plain text, similarly to `DefaultLogger` but, with customized log messages.

In the following, we show an example of an output of a check formatted in plain text and default text.

Plain text:

```
Starting audit...
starting file=Test.java
Test.java line=1 column=1 severity=warning message=key
finished file=Test.java
Audit done.
```

Default text:

```
Starting audit...
Test.java:1:1: warning: key
Audit done.
```

Currently, there are two broken tests: **testAddError** and **testFileStarted**. Fix them in the following order:

1. Fix testAddError:

This test is failing because method `addError(AuditEvent aEvt)` in `PlainTextLogger` is not checking whether the severity level is 'ERROR'. Implement this check and make sure the test pass before you go to the next fix.

2. Fix testFileStarted:

This test is failing because method `fileStarted(AuditEvent aEvt)` in `PlainTextLogger` is currently empty. Implement this method and rerun the tests.

To complete the task, check in the changed class to the repository.

Test to pass:
`com.puppycrawl.tools.checkstyle.PlainTextLoggerTest`
Class to modify and check in:
`com.puppycrawl.tools.checkstyle.PlainTextLogger`

Figure A.2. Handout for one participant of a pair of participants – Part 1 of 2

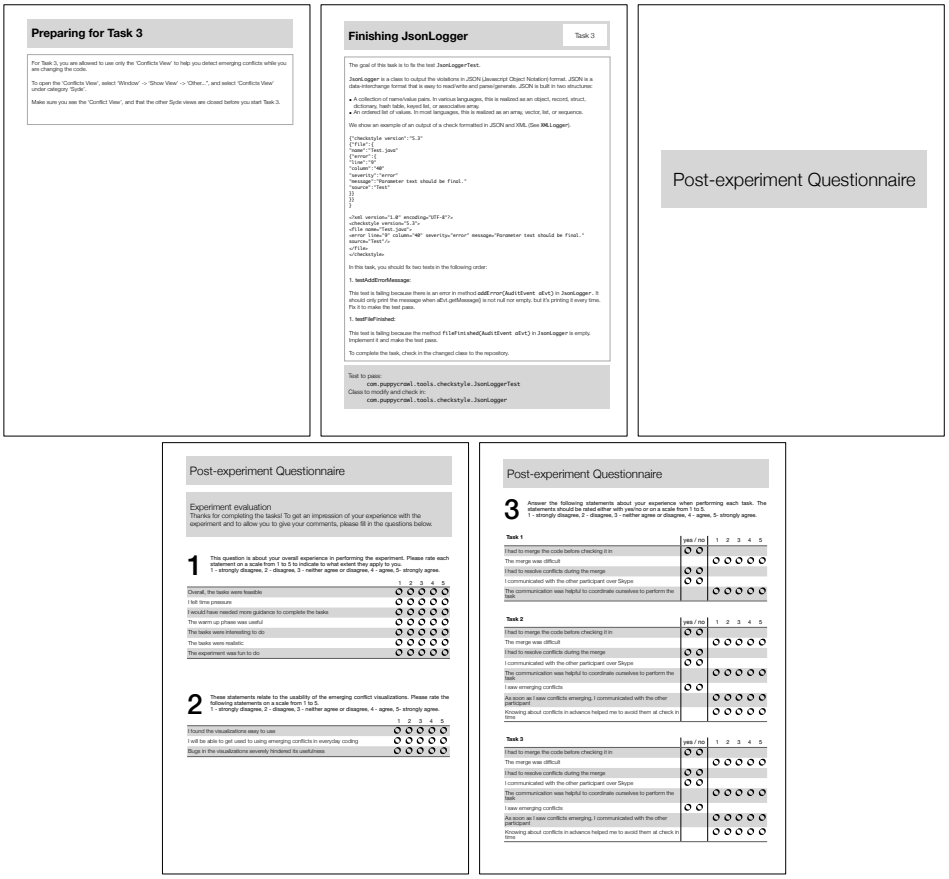


Figure A.3. Handout for one participant of a pair of participants – Part 2 of 2

Table A.1. First part of the answers to the screening questionnaire

Run	Id	Age	Education background	Current position	Experience Level						Familiarity w. Checkstyle
					Java	Team development	Dev. industrial size systems	Using IDEs	Using Eclipse for Java dev.	Using SCM	JUnit testing	
R1	P1	24	Computer science	Master student	advanced	advanced	beginner	advanced	expert	advanced	knowledgeable	none
R1	P2	26	Computer science	Master student	advanced	knowledgeable	beginner	advanced	advanced	knowledgeable	none	none
R2	P3	28	Computer science	PhD student	knowledgeable	knowledgeable	beginner	advanced	beginner	advanced	knowledgeable	beginner
R2	P4	30	Computer science	PhD student	beginner	knowledgeable	knowledgeable	knowledgeable	beginner	beginner	beginner	none
R3	P5	23	Computer science	Master student	expert	expert	beginner	expert	expert	expert	knowledgeable	none
R3	P6	22	Computer science	Master student	advanced	advanced	none	advanced	advanced	advanced	advanced	none
R4	P7	30	Computer science	PhD student	knowledgeable	advanced	knowledgeable	expert	expert	advanced	knowledgeable	none
R4	P8	38	Electrical engineering & Computer science	PhD student & Professor	expert	knowledgeable	knowledgeable	advanced	advanced	expert	knowledgeable	none
R5	P9	24	Computer science	PhD student	expert	knowledgeable	knowledgeable	advanced	advanced	expert	knowledgeable	none
R5	P10	24	Computer science	PhD student	expert	advanced	knowledgeable	advanced	advanced	advanced	advanced	knowledgeable
R6	P11	22	Computer science	Master student	expert	advanced	none	expert	advanced	expert	knowledgeable	none
R6	P12	24	Computer science	Master student	advanced	advanced	none	advanced	advanced	knowledgeable	knowledgeable	none

Table A.2. Second part of the answers to the screening questionnaire

Run	Id	Number of years of Experience							
		Java	Team development	Dev. industrial size systems	Using IDEs	Using Eclipse for Java dev.	Using SCM	JUnit testing	Familiarity w. Checkstyle
R1	P1	5	5	0	5	5	4.5	5	0
R1	P2	6	-	0.2	6	6	2	0	0
R2	P3	5	1.5	1	2	1.5	3	2	0
R2	P4	2	2	2	2	0.5	0	0	0
R3	P5	5	3	0	3	4	4	3	0
R3	P6	4	4	0	4	4	4	4	0
R4	P7	5	5	2	5	3	4	2	0
R4	P8	7	4	3	7	5	3	4	0
R5	P9	5	5	1	5	5	5	5	0
R5	P10	7	4	2	6	6	6	7	1
R6	P11	5	6	0	5	5	6	3	0
R6	P12	4	1	0	4	4	1	1	0

Table A.3. Third part of the answers to the screening questionnaire

Run	Id	SCM System currently used	Work in teams	Size of the team	Frequency of checkout	Frequency of check-in	Frequency of conflict appearance
R1	P1	Git and a bit of SVN	yes	3-5 during UNI, 2-3 outside UNI projects	Various times per day during high activity; once per day during normal activity periods	Various times per day during high activity; once per day during normal activity periods	Not frequently, because the teams I work in have assigned roles/tasks
R1	P2	None	no		Before committing to check for conflicts	As soon as a have a function working, fixed a bug or executed an update	Almost every time
R2	P3	Store (VisualWorks) SVN (mainly for versioning latex files)	Most of the time NO	Maximum of 2 people	Every time (day) I start working on it	After new tests are created and run	Not very often, thanks to store granularity (method level)
R2	P4	Monticello, SVN	No		Once per day	Once per hour	Once per day
R3	P5	SVN	Often	2-4	More than once a day (when working in a team)	1-2 per day	Depends on the team, if well trained once a week, otherwise more than 3
R3	P6	SVN, Git	Often	2	Daily	Daily	Rarely
R4	P7	SVN	Yes	2 to 3	Weekly	Weekly	Once a month
R4	P8	CVS	Sometimes	around 3 or 4	Daily	Daily	It is not so frequent, it usually happens once every 3 months in the present project
R5	P9	SVN	Often	2	Once a week	Once a week	Once a month
R5	P10	Git, SVN, Rietveld	2 to 5 on average	Occasionally	Not often, since currently I am working on small (two people) or personal projects. So there are more checkouts. I would say a month	Once every couple of days	Not common, again small sized teams nowadays
R6	P11	SVN	Often	3-5 people	Usually 3-4 times a day, it really depends on the frequency of updates for the project	It really depends on how much I am involved in the project, usually for projects in which I have a relevant role many times a day. I prefer to commit changes frequently to avoid conflicts.	Once a week
R6	P12	SVN	Often	5	Always before I begin working in order to avoid conflicts when I commit something new or modified.	Often, generally when I modify something that works at all.	Rarely. If it happens, it is generally due to my lack of checking out the repository before start working.

Table A.4. Answers to the debriefing questionnaire

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
Experiment evaluation (1 - strongly disagree to 5 - strongly agree)												
Overall, the tasks were feasible	5	4	5	5	5	4	5	5	5	5	5	5
I felt time pressure	1	2	4	2	3	2	3	1	3	1	1	2
I would have needed more guidance to complete the tasks	1	1	1	1	2	2	2	1	1	4	1	1
The warm up phase was useful	2	5	3	5	3	4	4	4	5	5	4	4
The tasks were interesting to do	2	5	5	5	4	4	4	3	4	3	3	4
The tasks were realistic	5	4	5	5	5	4	4	3	4	2	4	3
The experiment was fun to do	5	5	5	3	5	5	5	3	5	5	5	5
Visualizations evaluation (1 - strongly disagree to 5 - strongly agree)												
I found the visualizations easy to use	2	3	4	4	5	4	4	4	4	3	5	4
I will be able to get used to using emerging conflicts in everyday coding	2	3	4	4	4	3	4	4	4	4	5	4
Bugs in the visualizations severely hindered its usefulness	1	3	2	1	3	3	4	2	2	2	1	4
Evaluation of each task (1 - strongly disagree to 5 - strongly agree; yes/no)												
Task 1												
I had to merge the code before checking it in	yes	no	yes	yes	no	yes	yes	no	no	yes	no	yes
The merge was difficult	4	-	3	3	-	3	1	-	2	1	-	2
I had to resolve conflicts during the merge	yes	no	yes	yes	no	yes	yes	no	yes	yes	no	yes
I communicated with the other participant over Skype	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
The communication was helpful to coordinate ourselves to perform the task	4	3	5	5	4	4	5	4	4	5	4	4
Task 2												
I had to merge the code before checking it in	yes	no	yes	no	yes	no	yes	yes	yes	yes	no	yes
The merge was difficult	4	-	3	2	3	-	1	2	2	1	-	2
I had to resolve conflicts during the merge	yes	no	yes	no	yes	no	yes	yes	yes	yes	no	yes
I communicated with the other participant over Skype	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
The communication was helpful to coordinate ourselves to perform the task	3	4	5	5	4	4	5	4	4	5	4	4
I saw emerging conflicts	yes	yes	yes	yes	yes	yes	yes	yes	yes	no	yes	yes
As soon as I saw conflicts emerging, I communicated with the other participant	4	3	5	4	1	3	4	4	5	1	5	5
Knowing about conflicts in advance helped me to avoid them at check-in time	2	3	3	5	4	3	4	2	5	3	4	5
Task 3												
I had to merge the code before checking it in	no	yes	no	yes	yes	no	no	yes	yes	yes	no	no
The merge was difficult	-	4	-	3	3	-	-	2	2	1	-	-
I had to resolve conflicts during the merge	no	yes	no	yes	yes	no	no	yes	yes	yes	no	no
I communicated with the other participant over Skype	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
The communication was helpful to coordinate ourselves to perform the task	3	2	4	5	2	4	5	3	5	5	4	3
I saw emerging conflicts	no	yes	no	yes	yes	yes	yes	yes	yes	no	yes	yes
As soon as I saw conflicts emerging, I communicated with the other participant	3	3	-	5	2	4	4	4	5	1	5	5
Knowing about conflicts in advance helped me to avoid them at check-in time	3	4	-	5	4	3	4	2	5	3	4	5

Appendix B

Experimental Data for the Controlled Experiment with Replay

In this chapter we present all the additional details necessary to make our experiment repeatable, complementary to the information presented in Section 8.4. This includes questionnaires, oracle sets, and the entire experimental data set collected from our subjects.

B.1 Object System

The Spreadsheet project used in this experiment is an open-source project under the GNU GPL v3 license. Its source code and commit history are available at `http://code.google.com/p/spread-ur-ca-gh-el/`.

B.2 Screening Questionnaire

Using Google Docs[1], we designed an online questionnaire that provided an accessible platform for volunteers to enroll, and allowed for the capturing of the personal information we later used to assign the subjects to treatments (see Figure B.1 (p.203)).

B.3 Experimental Questionnaire

The content of the experimental questionnaires, including variations due to the different combinations of treatments, is presented in text below. The form of the questionnaires as presented to the participants is exemplified in Figure B.2 (p.204) and Figure B.3 (p.205), which show the questionnaire for the treatment T2.

We provide a description of the tasks as well as an oracle detailing the task solutions, specifying how grading was done for each task.

[1] See `http://docs.google.com`.

B.3.1 Introduction

The aim of this experiment is to compare tool efficiency in supporting software practitioners understanding the change history of a software system.

You will use Eclipse with the <toolset> to analyze Spreadsheet, a spreadsheet application written in Java by undergraduate students at the University of Lugano.

You are going to perform six tasks, with limited time to solve five of them. You have 10 minutes to solve each of the first four tasks, and 20 minutes to solve task five.

We kindly ask you:

- to write down your answers in a legible way;
- not to consult any other participant during the experiment;
- to perform the tasks in the specified order;
- to write down the current time before starting to work on a new task (after reading it) and once after completing all the tasks;
- to announce to the experimenter that you are starting to work on a new task (after reading it), in order to reset your allocated timer;
- not to return to earlier tasks because it affects the timing;
- to fill in the required information for each task. In the case of multiple choices, check the most appropriate answer and provide additional information, if requested.

In the following, there is one warm-up task for you to get used to the tool and the experiment. The experiment is concluded with a short debriefing questionnaire.

Thank you for participating in this experiment!
Lile Hattori, Michele Lanza, Mircea Lungu and Marco D'Ambros

B.3.2 Tasks

Task 1: Getting Familiar with someone's code. Imagine that you are on May 22nd and that you are joining this team to replace Omar, who was allocated to another project. This company loosely follows a software process, so Omar did not document what he was doing before he left. Your first task is to find out what he was working on, so you can start from what he left unfinished. Look at the changes Omar has done during the week (May 17th to 21st) and identify the two classes he changed the most. Changes should be quantified with number commits per class.

The two classes Omar (elabedomar) changed the most between May 17th and May 21st are:

1. class .....
2. class .....

Choices:
All classes of the system.

Solution:

- ch.usi.inf.pf2.gui.JSpreadSheet
- ch.usi.inf.pf2.spreadsheet.model.SpreadSheetSelectionModel

(0.5p for each correct class)

Task 2: Awareness of team activity. You are on your second day of work (May 23rd) and you have just started to work on a set of classes. You want to find out whether someone else has recently changed it before you commit your changes. Below are the list of classes and methods you are working on. You know that before Omar left the team, Rocco was pair programming with him, while Luca and Mattia were working on other parts of the code. Hence, focus on Rocco, and identify the methods that he has changed on the previous day (May 22nd).

Identify methods that Rocco (roccoghielmini) changed on May 22nd.

Choices:

- ch.usi.inf.pf2.Cell.getCoordinate()
- ch.usi.inf.pf2.Cell.setValueType(String valueType)
- ch.usi.inf.pf2.Recognizer.findit(String cellValue, String pattern)
- ch.usi.inf.pf2.Recognizer.isFunction(String cellValue)
- ch.usi.inf.pf2.spreadsheet.model.SpreadSheetModel.setCellContents(final int coordY, final int coordX, final String text, boolean endOfText)
- ch.usi.inf.pf2.spreadsheet.model.SpreadSheetSelectionModel.getColumnCount()
- ch.usi.inf.pf2.spreadsheet.model.SpreadSheetSelectionModel.selectCell(int row, int col)
- ch.usi.inf.pf2.spreadsheet.model.SpreadSheetSelectionModel.moveTo(Point point, int width, int height)
- ch.usi.inf.pf2.spreadsheet.model.SpreadSheetSelectionModel.setSelectedRange(RangeModel selectedRange)

Solution:

- ch.usi.inf.pf2.Recognizer.isFunction(String cellValue)
- SpreadSheetModel.setCellContents(final int coordY, final int coordX, final String text, boolean endOfText)
- ch.usi.inf.pf2.spreadsheet.model.SpreadSheetSelectionModel.selectCell(int row, int col)

(0.33p for each correct method)

Task 3: Finding experts of an artifact. You are still on your second day (May 23th) and starting to get familiar with the source code. By now you know that ch.usi.inf.pf2.gui.MainFrame is one of the main classes of the system, but you have difficulties understanding how it works. Look for people who can help you out: search who changed ch.usi.inf.pf2.gui.MainFrame between May 17th and 22nd to find the two developers who changed it the most and rank them (1 for first, and 2 for second). In this case, changes are measured as number of lines of code changes (added/deleted).

Choices:
All developers.

Solution:

1. Luca (lucaurso)
2. Mattia (mattia.candeloro89)

(0.5p for each correct class)

Task 4: Relating code changes to a feature. Until May 20th the implementation of GUI features in ch.usi.inf.pf2.gui.JSpreadSheet was Mattia's responsibility. Now, you are taking over this responsibility and your first task is to refactor the code to improve its design and readability. You want to start with the most complicated feature, because it will need most of your effort. Look at the changes Mattia did on JSpreadSheet on May 20th and, from the list of feature below, identify the one Mattia struggled with the most – the one that underwent heaviest changes in terms of number of lines of code changes (added/deleted).

Choices:

- Handling range selection
- Handling mouse events
- Handling keyboard events
- Handling focus events
- Painting the spreadsheet component

Solution:

Handling keyboard events (1p)

Task 5: Tracking back the introduction of a bug. You have been working with the team for one week now and your next task is to fix a bug. The bug happens when someone tries to open an existing '.csv' file on the spreadsheet. To reproduce the bug, find class ch.usi.inf.pf2.gui.MainGUI, right-click on it and run as 'Java Application'. Then, click on 'Open', and select 'test.csv'. You

should get the exception illustrated below. Rocco told you that the bug wasn't happening with him until last time he changed the code (on May 27th). Based on this information, answer the following questions.

(An image with the exception is shown)

a. When was the bug introduced?

Choices:

Entire period of the development of the system.

Solution:

30.05.2010

b. Who introduced the bug?

Choices:

All developers of the system.

Solution:

Luca

c. What change caused the bug?

Solution:

In free form saying that what caused the bug was the change of the call sheet.getGrid().get(i).add(Cell) to sheet.getGrid().get(i).add(int,Cell).

d. Propose a fix to the bug

Solution:

There are a couple of possibilities, but the easier one is to revert to the old code.

(0.25p for each correct answer)

Task 6: Understanding past refactorings. This system is now on its maintenance phase. The other developers were allocated to new projects and you are maintaining it alone. Since you joined the team almost at the end of the development cycle, when everything is very hectic, you didn't have time to deeply understand the system's architecture. Now, investigating the code, you've noticed that right before you joined the team (around May 17th to 21st) a major refactoring took place, involving the deletion of class ch.usi.inf.pf2.Sheet. This refactoring was mainly done by Luca with small contributions by the other developers. Investigate why Luca removed ch.usi.inf.pf2.Sheet, and what other classes were changed in the same refactoring. Describe the refactoring, giving details about what classes changed and why.

Solution:
Subjective answer: does not count for the qualitative analysis.

B.3.3 Debriefing Questionnaire

Time pressure. On a scale from 1 to 5, how did you feel about the time pressure? Please write in the box below the answer that matches your opinion the most:

1. Too much time pressure. I could not cope with the tasks, regardless of their difficulty.
2. Fair amount of pressure. I could certainly have done better with more time.
3. Not so much time pressure. I had to hurry a bit, but it was OK.
4. Very little time pressure. I felt quite comfortable with the time given.
5. No time pressure at all.

Difficulty. Regardless of the given time, please indicate how difficult would you rate the tasks? Please mark the appropriate difficulty for each of the tasks.

Scale:
1 – trivial; 2 – simple; 3 – intermediate; 4 – difficult; 5 – impossible.

Realism. How realistic were the tasks? Please indicate how much you agree that the tasks were realistic (you can see the situation happening in a real development scenario).

Scale:
1 – strongly disagree; 2 – disagree; 3 – undecided; 4 – agree; 5 – strongly agree.

Comments on the experiment. Enter comments and/or suggestions you may have about the experiment, which could help us improve it.

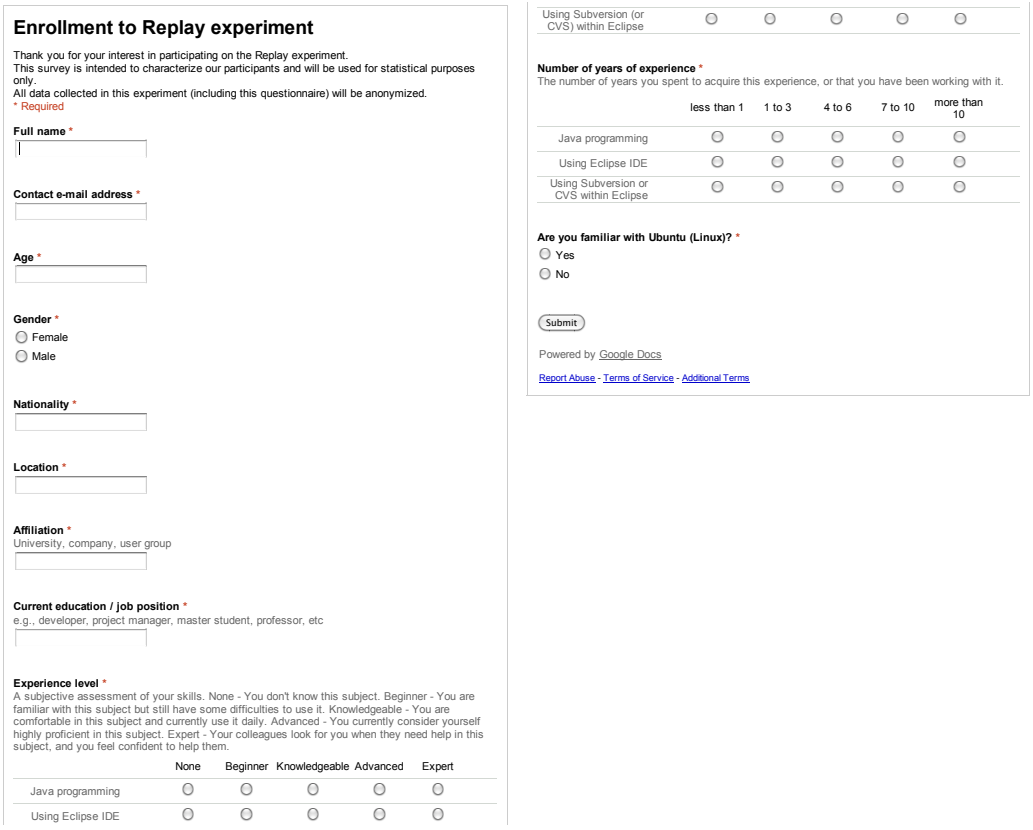
Comments on the Replay tool. Enter comments and/or suggestions to improve Replay (applicable for the experimental group).

B.4 Dataset

To provide a fully transparent experimental setup, we make available our experiment's entire data set.

In Table B.1 (p.206) we present the subjects and the personal information that we relied on when we assigned them to different blocks (*i.e.* based on experience and background). Once the subjects were assigned to the two blocks (beginner/advanced), within each block we assigned the subjects to a treatment using randomization. The assignment of subjects to treatments and blocks is also presented in Table B.1 (p.206).

The correctness level per task for each subject is presented in Table B.2 (p.207). Similarly, the subjects' completion times, both for each task and overall, are presented in Table B.3 (p.208). Finally, Table B.4 (p.209) and Table B.5 (p.210) present the data we collected from the subjects regarding the perceived time pressure, difficulty, and realism for each task, as experienced by our subjects. This data allowed us to determine whether there was a task that was highly unfair for one of the groups. Moreover, it provided us important hints on the type of tasks for which Replay is most beneficial, and for which type of users.



Enrollment to Replay experiment

Thank you for your interest in participating on the Replay experiment.
This survey is intended to characterize our participants and will be used for statistical purposes only.
All data collected in this experiment (including this questionnaire) will be anonymized.
* Required

Full name *

Contact e-mail address *

Age *

Gender *
Female
Male

Nationality *

Location *

Affiliation *
University, company, user group

Current education / job position *
e.g., developer, project manager, master student, professor, etc

Experience level *
A subjective assessment of your skills. None - You don't know this subject. Beginner - You are familiar with this subject but still have some difficulties to use it. Knowledgeable - You are comfortable in this subject and currently use it daily. Advanced - You currently consider yourself highly proficient in this subject. Expert - Your colleagues look for you when they need help in this subject, and you feel confident to help them.

	None	Beginner	Knowledgeable	Advanced	Expert
Java programming					
Using Eclipse IDE					
Using Subversion (or CVS) within Eclipse					

Number of years of experience *
The number of years you spent to acquire this experience, or that you have been working with it.

	less than 1	1 to 3	4 to 6	7 to 10	more than 10
Java programming					
Using Eclipse IDE					
Using Subversion or CVS within Eclipse					

Are you familiar with Ubuntu (Linux)? *
Yes
No

Submit

Powered by Google Docs

Report Abuse - Terms of Service - Additional Terms

Figure B.1. The screening questionnaire used to collect the participants' personal information and information about their experience levels

(a) Introduction (b) Warm-up (c) Time - logged after each task

Figure B.2. Handout for treatment T2 (experimental group) – Part 1 of 2

Figure B.3. Handout for treatment T2 (experimental group) – Part 2 of 2

Table B.1. The subjects' personal information, clustered by treatment combination (control/experimental)

					Experience level			Years of experience		
Subject ID	**Treatment**	**Block**	**Age**	**Job position**	**Java**	**Eclipse**	**SVN**	**Java**	**Eclipse**	**SVN**
E1	Edi+Replay	advanced	27	PhD student	knowledgeable	knowledgeable	advanced	4 – 6	4 – 6	4 – 6
E2	Edi+Replay	advanced	24	Master student	expert	expert	expert	4 – 6	4 – 6	4 – 6
E3	Edi+Replay	advanced	23	Master student	knowledgeable	knowledgeable	knowledgeable	4 – 6	4 – 6	4 – 6
E4	Edi+Replay	beginner	27	PhD student	knowledgeable	knowledgeable	beginner	4 – 6	1 – 3	< 1
E5	Edi+Replay	advanced	53	Professor	advanced	advanced	advanced	> 10	7 – 10	7 – 10
E6	Edi+Replay	advanced	31	PhD student	advanced	advanced	advanced	> 10	7 – 10	7 – 10
E7	Edi+Replay	beginner	27	PhD student	expert	knowledgeable	advanced	7 – 10	1 – 3	1 – 3
E8	Edi+Replay	beginner	30	PhD student	advanced	advanced	knowledgeable	1 – 3	4 – 6	< 1
E9	Edi+Replay	beginner	29	PhD student	beginner	beginner	beginner	1 – 3	1 – 3	< 1
E10	Edi+Replay	advanced	29	PhD student	advanced	advanced	advanced	7 – 10	4 – 6	4 – 6
E11	Edi+Replay	beginner	27	PhD student	knowledgeable	knowledgeable	knowledgeable	1 – 3	1 – 3	1 – 3
E12	Edi+Replay	beginner	27	PhD student	knowledgeable	knowledgeable	none	4 – 6	4 – 6	< 1
E13	Edi+Replay	advanced	30	PhD student	expert	expert	expert	7 – 10	7 – 10	4 – 6
E14	Edi+Replay	advanced	26	PhD student	advanced	advanced	advanced	4 – 6	4 – 6	4 – 6
E15	Edi+Replay	beginner	23	Master student	advanced	advanced	knowledgeable	1 – 3	1 – 3	1 – 3
E16	Edi+Replay	beginner	30	Master student	advanced	knowledgeable	knowledgeable	1 – 3	1 – 3	1 – 3
E17	Edi+Replay	beginner	27	Master student	advanced	knowledgeable	none	7 – 10	4 – 6	< 1
E18	Edi+Replay	advanced	23	Master student	expert	advanced	knowledgeable	4 – 6	4 – 6	1 – 3
E19	Edi+Replay	beginner	25	Master student	advanced	knowledgeable	beginner	4 – 6	1 – 3	1 – 3
E20	Edi+Replay	beginner	26	Master student	knowledgeable	knowledgeable	beginner	1 – 3	1 – 3	1 – 3
C1	Edi+SVN	beginner	25	PhD student	beginner	beginner	none	< 1	< 1	< 1
C2	Edi+SVN	advanced	25	Master student	knowledgeable	knowledgeable	beginner	7 – 10	7 – 10	1 – 3
C3	Edi+SVN	beginner	27	PhD student	advanced	advanced	knowledgeable	4 – 6	1 – 3	1 – 3
C4	Edi+SVN	advanced	34	Post-doc	advanced	advanced	knowledgeable	> 10	4 – 6	7 – 10
C5	Edi+SVN	beginner	26	PhD student	knowledgeable	beginner	beginner	< 1	< 1	< 1
C6	Edi+SVN	beginner	29	PhD student	knowledgeable	knowledgeable	beginner	4 – 6	4 – 6	< 1
C7	Edi+SVN	advanced	26	Master student	advanced	advanced	knowledgeable	7 – 10	7 – 10	4 – 6
C8	Edi+SVN	beginner	25	PhD student	advanced	knowledgeable	none	7 – 10	4 – 6	< 1
C9	Edi+SVN	beginner	30	PhD student	beginner	beginner	beginner	< 1	< 1	< 1
C10	Edi+SVN	advanced	29	PhD student	expert	expert	expert	7 – 10	4 – 6	1 – 3
C11	Edi+SVN	advanced	27	PhD student	advanced	knowledgeable	knowledgeable	7 – 10	7 – 10	4 – 6
C12	Edi+SVN	beginner	33	PhD student	advanced	knowledgeable	knowledgeable	4 – 6	1 – 3	1 – 3
C13	Edi+SVN	advanced	30	PhD student	expert	expert	expert	7 – 10	7 – 10	4 – 6
C14	Edi+SVN	advanced	38	PhD student/Assistant Prof.	advanced	advanced	knowledgeable	4 – 6	4 – 6	4 – 6
C15	Edi+SVN	advanced	31	PhD student	expert	advanced	advanced	4 – 6	4 – 6	1 – 3
C16	Edi+SVN	beginner	33	Master student	beginner	beginner	beginner	1 – 3	1 – 3	< 1
C17	Edi+SVN	beginner	22	Master student	none	beginner	none	< 1	< 1	< 1
C18	Edi+SVN	beginner	23	Master student	advanced	advanced	knowledgeable	4 – 6	1 – 3	4 – 6
C19	Edi+SVN	beginner	27	Master student	knowledgeable	advanced	beginner	1 – 3	1 – 3	< 1
C20	Edi+SVN	beginner	22	Master student	knowledgeable	knowledgeable	none	4 – 6	1 – 3	< 1

Table B.2. The correctness of the subjects' solutions to the tasks

	Correctness per task					
Subject ID	**Task 1**	**Task 2**	**Task 3**	**Task 4**	**Task 5**	**Total**
E1	1.00	1.00	1.00	1.00	0.50	4.50
E2	1.00	1.00	1.00	0.00	1.00	4.00
E3	1.00	1.00	1.00	1.00	0.50	4.50
E4	1.00	0.67	1.00	1.00	0.00	3.67
E5	1.00	1.00	1.00	1.00	0.00	4.00
E6	1.00	1.00	1.00	1.00	1.00	5.00
E7	1.00	1.00	1.00	1.00	1.00	5.00
E8	1.00	1.00	1.00	1.00	1.00	5.00
E9	1.00	1.00	1.00	1.00	0.50	4.50
E10	1.00	1.00	1.00	1.00	1.00	5.00
E11	1.00	1.00	1.00	1.00	1.00	5.00
E12	1.00	0.67	1.00	1.00	0.00	3.67
E13	1.00	0.67	1.00	1.00	0.25	3.92
E14	1.00	1.00	1.00	0.00	1.00	4.00
E15	1.00	1.00	1.00	0.00	0.00	3.00
E16	1.00	1.00	1.00	1.00	0.00	4.00
E17	1.00	0.33	1.00	0.00	0.00	2.33
E18	1.00	1.00	1.00	1.00	1.00	5.00
E19	1.00	0.67	1.00	0.00	0.50	3.17
E20	1.00	1.00	1.00	0.00	0.00	3.00
C1	0.50	0.67	1.00	0.00	0.50	2.67
C2	1.00	1.00	1.00	1.00	0.50	4.50
C3	1.00	1.00	0.00	0.00	1.00	3.00
C4	1.00	1.00	0.00	1.00	1.00	4.00
C5	1.00	1.00	1.00	1.00	0.75	4.75
C6	1.00	0.67	1.00	1.00	0.00	3.67
C7	1.00	1.00	0.00	1.00	0.75	3.75
C8	1.00	1.00	1.00	1.00	0.00	4.00
C9	0.00	1.00	1.00	1.00	1.00	4.00
C10	1.00	0.67	1.00	0.00	1.00	3.67
C11	0.50	1.00	1.00	1.00	1.00	4.50
C12	0.00	1.00	1.00	1.00	0.75	3.75
C13	1.00	1.00	1.00	1.00	1.00	5.00
C14	1.00	1.00	1.00	1.00	0.25	4.25
C15	0.00	0.67	0.50	1.00	1.00	3.17
C16	1.00	0.00	0.50	0.00	0.00	1.50
C17	0.50	1.00	1.00	1.00	0.00	3.50
C18	1.00	0.67	1.00	1.00	0.00	3.67
C19	0.00	0.00	1.00	0.00	0.00	1.00
C20	0.00	1.00	1.00	0.00	0.00	2.00

Table B.3. The subjects' completion time for each task (in minutes)

	Completion time per task							
Subject ID	**Task 1**	**Task 2**	**Task 3**	**Task 4**	**Task 5**	**Task 6**	**Total**	**Total (excl. T6)**
E1	4.00	4.00	1.00	7.00	18.00	17.00	51.00	34.00
E2	4.83	6.33	1.83	2.75	22.42	22.33	60.50	38.17
E3	3.08	7.33	1.67	16.42	25.42	0.00	53.92	53.92
E4	5.50	5.33	2.50	10.08	23.93	11.33	58.68	47.35
E5	9.75	3.25	1.17	9.92	20.67	21.92	66.67	44.75
E6	4.67	5.50	2.25	7.42	15.00	8.08	42.92	34.83
E7	4.75	8.75	2.58	7.50	19.33	8.83	51.75	42.92
E8	7.00	7.00	4.00	11.00	14.00	20.00	63.00	43.00
E9	6.00	7.00	4.00	8.00	20.00	28.00	73.00	45.00
E10	4.00	7.00	2.00	9.00	22.00	50.00	94.00	44.00
E11	2.00	6.00	1.00	8.00	15.00	19.00	51.00	32.00
E12	5.00	7.00	4.00	8.00	20.00	13.00	57.00	44.00
E13	3.08	6.75	1.20	5.25	20.83	20.17	57.28	37.12
E14	6.00	9.00	3.00	7.00	20.00	19.00	54.00	38.00
E15	3.58	6.17	2.08	6.00	23.00	22.00	63.17	40.83
E16	2.25	9.58	2.42	6.92	19.67	24.33	65.17	40.83
E17	9.50	6.42	4.67	6.53	21.03	23.83	71.98	48.15
E18	5.33	7.00	4.25	6.92	18.75	13.33	55.58	42.25
E19	4.50	7.00	4.33	10.25	20.00	28.25	74.33	46.08
E20	6.08	6.75	2.25	4.25	20.33	26.00	65.67	39.67
C1	5.00	4.00	13.00	3.00	14.00	12.00	51.00	39.00
C2	8.33	5.75	10.33	10.75	14.33	27.00	76.50	49.50
C3	7.00	7.00	9.00	10.00	21.00	18.00	72.00	54.00
C4	7.00	10.00	9.00	8.00	16.00	28.00	78.00	50.00
C5	9.18	3.42	10.67	1.67	20.25	14.50	59.68	45.18
C6	7.33	3.77	4.25	5.92	23.92	25.42	70.60	45.18
C7	6.08	5.50	10.00	7.67	21.33	16.42	67.00	50.58
C8	8.00	9.00	10.00	5.00	19.00	15.00	66.00	51.00
C9	9.00	10.00	9.00	5.00	17.00	19.00	69.00	50.00
C10	7.92	6.10	9.83	4.25	23.00	13.50	64.60	51.10
C11	6.92	6.67	7.83	4.83	14.67	23.00	63.92	40.92
C12	7.25	8.00	11.50	6.08	11.00	21.33	65.17	43.83
C13	4.37	7.58	6.67	4.00	13.58	10.50	46.70	36.20
C14	4.00	6.00	3.00	4.00	20.00	21.00	58.00	37.00
C15	6.00	5.00	6.00	5.00	16.00	16.00	54.00	38.00
C16	11.33	9.00	13.75	9.17	12.17	13.33	68.75	55.42
C17	10.50	7.83	9.33	5.25	19.92	8.50	61.33	52.83
C18	8.42	4.33	5.83	5.33	21.58	14.57	60.07	45.50
C19	11.67	10.17	4.95	7.83	20.83	27.75	83.20	55.45
C20	12.50	9.00	7.25	9.42	17.38	17.25	72.80	55.55

Table B.4. The subjects' perceived time pressure and task difficulty

Subject	Time	Difficulty level					
ID	pressure	Task 1	Task 2	Task 3	Task 4	Task 5	Task 6
E1	3	simple	simple	simple	simple	intermediate	difficult
E2	3	trivial	intermediate	simple	intermediate	simple	difficult
E3	1	simple	simple	simple	intermediate	difficult	
E4	3	trivial	simple	trivial	intermediate	difficult	difficult
E5	2	trivial	simple	simple	intermediate	impossible	difficult
E6	4	intermediate	simple	intermediate	intermediate	difficult	difficult
E7	3	trivial	simple	trivial	intermediate	intermediate	impossible
E8		difficult	difficult	simple	impossible	difficult	impossible
E9	2	simple	simple	simple	simple	intermediate	difficult
E10	2	trivial	simple	trivial	simple	simple	difficult
E11	3	trivial	simple	simple	simple	intermediate	difficult
E12	2	difficult	difficult	difficult	difficult	difficult	difficult
E13	2	trivial	simple	simple	intermediate	difficult	difficult
E14	4	trivial	simple	trivial	intermediate	difficult	difficult
E15	4	simple	simple	simple	simple	difficult	intermediate
E16	2	trivial	simple	trivial	simple	intermediate	difficult
E17	2	trivial	trivial	simple	intermediate	difficult	difficult
E18	4	simple	trivial	trivial	intermediate	intermediate	difficult
E19	2	trivial	simple	simple	intermediate	difficult	impossible
E20	3	simple	simple	trivial	intermediate	difficult	difficult
C1	3	simple	simple	intermediate	intermediate	intermediate	intermediate
C2	4	simple	simple	difficult	difficult	intermediate	intermediate
C3	2	trivial	trivial	simple	intermediate	simple	difficult
C4	3	trivial	trivial	trivial	simple	intermediate	difficult
C5	3	intermediate	simple	difficult	intermediate	difficult	simple
C6	3	simple	trivial	intermediate	intermediate	difficult	intermediate
C7	2	simple	trivial	impossible	difficult	intermediate	intermediate
C8	2	intermediate	simple	intermediate	intermediate	impossible	intermediate
C9	3	simple	simple	intermediate	intermediate	difficult	intermediate
C10	3	trivial	trivial	intermediate	intermediate	simple	simple
C11	3	intermediate	simple	difficult	difficult	intermediate	difficult
C12	5	simple	simple	difficult	intermediate	simple	simple
C13	2	intermediate	simple	difficult	intermediate	simple	intermediate
C14	2	trivial	trivial	trivial	simple	difficult	difficult
C15	4	simple	simple	simple	simple	intermediate	difficult
C16	1	intermediate	difficult	impossible	difficult	difficult	impossible
C17	4	intermediate	simple	intermediate	trivial	difficult	difficult
C18	3	simple	simple	intermediate	simple	intermediate	difficult
C19	1	intermediate	intermediate	simple	simple	difficult	difficult
C20	2	difficult	difficult	intermediate	intermediate	intermediate	undecided

Table B.5. The subjects' perceived realism of each task

Subject	Task realism					
ID	Task 1	Task 2	Task 3	Task 4	Task 5	Task 6
E1	disagree	disagree	agree	undecided	agree	strongly agree
E2	agree	agree	agree	agree	undecided	agree
E3	agree	agree	agree	agree	agree	
E4	undecided	undecided	undecided	undecided	undecided	undecided
E5	agree	agree	agree	agree	agree	agree
E6	undecided	agree	agree	undecided	strongly agree	strongly agree
E7	disagree	disagree	agree	strongly disagree	strongly agree	agree
E8	undecided	undecided	agree	disagree	agree	strongly agree
E9	agree	agree	agree	agree	agree	agree
E10	agree	strongly agree	agree	strongly agree	strongly agree	agree
E11	undecided	undecided	agree	agree	agree	agree
E12	agree	agree	agree	agree	agree	agree
E13	strongly agree	strongly agree	strongly agree	undecided	agree	agree
E14	agree	agree	strongly agree	undecided	agree	agree
E15	agree	undecided	agree	agree	strongly agree	undecided
E16	agree	strongly agree	strongly agree	agree	strongly agree	strongly agree
E17	agree	agree	agree	undecided	strongly agree	strongly agree
E18	agree	disagree	undecided	agree	strongly agree	strongly agree
E19	disagree	disagree	undecided	agree	strongly agree	strongly agree
E20	agree	strongly agree	strongly agree	undecided	strongly agree	undecided
C1	agree	agree	agree	agree	agree	agree
C2	undecided	agree	agree	undecided	agree	agree
C3	agree	undecided	agree	agree	strongly agree	disagree
C4	agree	agree	disagree	strongly disagree	agree	strongly disagree
C5	disagree	disagree	strongly agree	undecided	strongly agree	agree
C6	strongly agree	agree	disagree	disagree	agree	strongly agree
C7	agree	agree	disagree	undecided	strongly agree	agree
C8	agree	agree	agree	agree	agree	agree
C9	undecided	undecided	agree	undecided	strongly agree	strongly agree
C10	agree	agree	strongly agree	undecided	strongly agree	strongly agree
C11	undecided	undecided	strongly agree	disagree	disagree	agree
C12	agree	agree	disagree	disagree	agree	undecided
C13	agree	agree	undecided	agree	agree	agree
C14	agree	agree	agree	undecided	strongly agree	
C15	strongly agree	strongly agree	strongly agree	undecided	agree	
C16	undecided	agree	undecided	agree	disagree	disagree
C17	disagree	agree	agree	agree	strongly agree	agree
C18	agree	agree	strongly agree	agree	strongly agree	strongly agree
C19	agree	strongly agree	undecided	undecided	strongly agree	agree
C20	undecided	undecided	undecided	undecided	agree	agree

Part VI

Bibliography

Bibliography

[Alle 77] T. Allen. *Managing the Flow of Technology: Technology Transfer and the Dissemination of Technological Information within the R&D Organization*. MIT Press, 1977.

[Alwi 08] B. de Alwis and G. C. Murphy. "Answering Conceptual Queries with Ferret". In: *Proceedings of ICSE 2008 (30th International Conference on Software Engineering)*, pp. 21–30, ACM Press, 2008.

[Anvi 06] J. Anvik, L. Hiew, and G. C. Murphy. "Who Should Fix this Bug?". In: *Proceedings of ICSE 2006 (28th International Conference on Software Engineering)*, pp. 361–370, ACM Press, 2006.

[Anvi 07] J. Anvik and G. C. Murphy. "Determining Implementation Expertise from Bug Reports". In: *Proceedings of MSR 2007 (4th International Workshop on Mining Software Repositories)*, p. 2, IEEE Computer Society, 2007.

[Atla 11] Atlassian Pty Ltd. "Jira". [Software]. Available: http://www.atlassian.com/software/jira/ [Accessed: Aug. 3, 2011], 2011.

[Bacc 10] A. Bacchelli, M. Lanza, and V. Humpa. "Towards Integrating E-Mail Communication in the IDE". In: *Proceedings of SUITE 2010 (2nd International Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation)*, pp. 1–4, ACM, 2010.

[Bacc 11] A. Bacchelli, F. Rigotti, L. Hattori, and M. Lanza. "Manhattan – a 3D City Visualization in Eclipse". In: *In Proceedings of Eclipse-IT 2011 (6th Workshop of the Italian Eclipse Community)*, pp. 307–310, 2011.

[Ball 97] T. Ball, J. Kim, A. Porter, and H. Siy. "If Your Version Control System Could Talk". In: *Proceedings of the ICSE Workshop on Process Modeling and Empirical Studies of Software Engineering*, 1997.

[Barb 08] R. Barbour. *Introducing Qualitative Research*. Sage, 2008.

[Beck 01] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. "The Agile Manifesto". Feb. 2001.

[Beck 04] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[Bell 97] Bell Labs. “Sablime v5.0 User’s Reference Manual”. 1997.

[Beye 06] D. Beyer and A. E. Hassan. “Animated Visualization of Software History using Evolution Storyboards”. In: *Proceedings of WCRE 2006 (13th Working Conference on Reverse Engineering)*, pp. 199–210, IEEE Computer Society, 2006.

[Bieh 07] J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson. “FASTDash: a Visual Dashboard for Fostering Awareness in Software Teams”. In: *Proceedings of CHI 2007 (25th SIGCHI Conference on Human Factors in Computing Systems*, pp. 1313–1322, ACM, 2007.

[Bird 09] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. “The Promises and Perils of Mining Git”. In: *Proceedings of MSR 2009 (6th IEEE International Working Conference on Mining Software Repositories)*, pp. 1–10, IEEE Computer Society, 2009.

[Boeh 86] B. Boehm. “A Spiral Model of Software Development and Enhancement”. *SIGSOFT Software Engineering Notes*, Vol. 11, pp. 14–24, Aug. 1986.

[Brun 11] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. “Proactive Detection of Collaboration Conflicts”. In: *Proceedings of ESEC/FSE 2011 (European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering)*, pp. 168–178, ACM Press, 2011.

[Cano 11] Canonical Ltd. “Bazaar”. [Software]. Available: http://bazaar.canonical.com [Accessed: Aug. 15, 2011], 2011.

[Carm 99] E. Carmel. *Global Software Teams - Collaborating Across Borders and Time Zones*. Prentice Hall, 1999.

[Carp 08] S. K. Carpenter, H. Pashler, J. T. Wixted, and E. Vul. “The Effects of Tests on Learning and Forgetting”. *Memory and Cognition*, Vol. 36, No. 2, pp. 438–448, 2008.

[Chac 09] S. Chacon. *Pro Git*. Apress, 1st Ed., 2009.

[Chen 04] K. Chen, S. R. Schach, L. Yu, J. Offutt, and G. Z. Heller. “Open-Source Change Logs”. *Empirical Software Engineering*, Vol. 9, pp. 197–210, Sep. 2004.

[Conr 98] R. Conradi and B. Westfechtel. “Version Models for Software Configuration Management”. *ACM Computing Survey*, Vol. 30, pp. 232–282, June 1998.

[Corn 10] B. Cornelissen, A. Zaidman, and A. van Deursen. “A Controlled Experiment for Program Comprehension through Trace Visualization”. *IEEE Transactions on Software Engineering*, Vol. 99, 2010.

[CPMB 11] CPMBraxis. Available: http://www.cpmbraxis.com/ [Accessed: Oct. 5, 2011], 2011.

[Cres 07] J. Creswell. *Qualitative Inquiry & Research Design*. Sage, 2nd Ed., 2007.

[DAmb 06a] M. D’Ambros and M. Lanza. “Reverse Engineering with Logical Coupling”. In: *Proceedings of WCRE 2006 (13th Working Conference on Reverse Engineering)*, pp. 189–198, IEEE CS Press, 2006.

[DAmb 06b] M. D'Ambros and M. Lanza. "Software Bugs and Evolution: A Visual Approach to Uncover Their Relationship". In: *Proceedings of CSMR 2006 (10th IEEE European Conference on Software Maintenance and Reengineering)*, pp. 227–236, IEEE CS Press, 2006.

[DAmb 09] M. D'Ambros, M. Lanza, and M. Lungu. "Visualizing Co-Change Information with the Evolution Radar". *IEEE Transactions on Software Engineering*, Vol. 35, No. 5, pp. 720 – 735, 2009.

[DAmb 10a] M. D'Ambros. *On the Evolution of Source Code and Defects*. PhD thesis, University of Lugano, Switzerland, Oct. 2010.

[DAmb 10b] M. D'Ambros, M. Lanza, and R. Robbes. "An Extensive Comparison of Bug Prediction Approaches". In: *Proceedings of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories)*, pp. 31–40, IEEE CS Press, 2010.

[Dami 07] D. Damian, L. Izquierdo, J. Singer, and I. Kwan. "Awareness in the Wild: Why Communication Breakdowns Occur". In: *Proceedings of the ICGSE 2007 (International Conference on Global Software Engineering)*, pp. 81–90, IEEE Computer Society, 2007.

[DeLi 05] R. DeLine, M. Czerwinski, and G. Robertson. "Easing Program Comprehension by Sharing Navigation Data". In: *Proceedings of the VLHCC 2005 (IEEE Symposium on Visual Languages and Human-Centric Computing)*, pp. 241–248, IEEE Computer Society, 2005.

[Denk 07] M. Denker, T. Gîrba, A. Lienhard, O. Nierstrasz, L. Renggli, and P. Zumkehr. "Encapsulating and Exploiting Change with Changeboxes". In: *Proceedings of ICDE 2007 (International Conference on Dynamic Languages: in conjunction with the 15th International Smalltalk Joint Conference 2007)*, pp. 25–49, ACM, 2007.

[Dewa 07] P. Dewan and R. Hegde. "Semi-synchronous Conflict Detection and Resolution in Asynchronous Software Development". In: *Proceedings of ECSCW 2007 (10th European Conference on Computer Supported Cooperative Work)*, pp. 24–28, Springer, 2007.

[DiBo 99] C. DiBona, S. Ockman, and M. Stone, Eds. *Open Sources: Voices from the Open Source Revolution*. O'Reilly & Associates, Inc., 1999.

[Dig 07a] D. Dig. *Automated Upgrading of Component-based Applications*. PhD thesis, University of Illinois at Urbana-Champaign, 2007.

[Dig 07b] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen. "Refactoring-Aware Configuration Management for Object-Oriented Programs". In: *Proceedings of ICSE 2007 (29th International Conference on Software Engineering)*, pp. 427–436, IEEE Computer Society, 2007.

[Dour 92] P. Dourish and V. Bellotti. "Awareness and Coordination in Shared Workspaces". In: *Proceedings of CSCW 1992 (ACM Conference on Computer-supported Cooperative Work)*, pp. 107–114, ACM Press, 1992.

[Ebbi 13] H. Ebbinghaus. *Uber das Gedchtnis. Untersuchungen zur Experimentellen Psychologie (Memory. A Contribution to Experimental Psychology)*. Leipzig: Duncker and Humblot, 1913.

[Ebra 07] P. Ebraert, J. Vallejos, P. Costanza, E. V. Paesschen, and T. D'Hondt. "Change-oriented Software Engineering". In: *Proceedings of ICDL 2007 (2007 International Conference on Dynamic Languages: in conjunction with the 15th International Smalltalk Joint Conference 2007)*, pp. 3–24, ACM, 2007.

[Ecli 11] Eclipse Foundation. "Eclipse - an Open Development Platform". [Software]. Available: http://www.eclipse.org/ [Accessed: Aug. 3, 2011], 2011.

[Estu 05] J. Estublier, D. Leblang, A. van der Hoek, R. Conradi, G. Clemm, W. Ticky, and D. Wiborg-Weber. "Impact of Software Engineering Research on the Practice of Software Configuration Management". *ACM Transactions on Software Engineering and Methodology*, Vol. 14, No. 4, pp. 383–430, 2005.

[Estu 06] J. Estublier and S. Garcia. "Concurrent Engineering Support in Software Engineering". In: *Proceedings of ASE 2006 (21st IEEE/ACM International Conference on Automated Software Engineering)*, pp. 209 –220, IEEE Computer Society, Sep. 2006.

[Feld 79] S. I. Feldman. "Make – A Program for Maintaining Computer Programs". *Software - Practice and Experience*, Vol. 9, pp. 255–265, 1979.

[Fisc 03] M. Fischer, M. Pinzger, and H. Gall. "Populating a Release History Database from Version Control and Bug Tracking Systems". In: *Proceedings of ICSM 2003 (19th IEEE International Conference on Software Maintenance)*, pp. 23–32, IEEE Computer Society Press, 2003.

[Fisc 04] M. Fischer and H. Gall. "Visualizing Feature Evolution of Large-Scale Software based on Problem and Modification Report Data". *Journal of Software Maintenance and Evolution: Research and Practice*, Vol. 16, No. 6, pp. 385–403, 2004.

[Fisc 06] M. Fischer and H. C. Gall. "EvoGraph: A Lightweight Approach to Evolutionary and Structural Analysis of Large Software Systems". In: *Proceedings of WCRE 2006 (13th Working Conference on Reverse Engineering)*, pp. 179–188, IEEE Computer Society, 2006.

[Flur 06] B. Fluri and H. C. Gall. "Classifying Change Types for Qualifying Change Couplings". In: *Proceedings of ICPC 2006 (14th IEEE International Conference on Program Comprehension)*, pp. 35–45, IEEE Computer Society, 2006.

[Flur 07a] B. Fluri, M. Würsch, and H. C. Gall. "Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes". In: *Proceedings of WCRE 2007 (14th Working Conference on Reverse Engineering)*, pp. 70–79, IEEE Computer Society, 2007.

[Flur 07b] B. Fluri, M. Würsch, M. Pinzger, and H. Gall. "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction". *IEEE Transactions on Software Engineering*, Vol. 33, No. 11, pp. 725–743, 2007.

[Fowl 99] M. Fowler. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, 1999.

[Frit 10a] T. Fritz and G. C. Murphy. "Using Information Fragments to Answer the Questions Developers Ask". In: *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering)*, pp. 175–184, IEE Computer Society, 2010.

[Frit 10b] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill. "A Degree-of-Knowledge Model to Capture Source Code Familiarity". In: *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering)*, pp. 385–394, IEEE Computer Society, 2010.

[Fros 07] R. Frost. "Jazz and the Eclipse Way of Collaboration". *IEEE Software*, Vol. 24, No. 6, pp. 114–117, 2007.

[Gall 03] H. Gall, M. Jazayeri, and J. Krajewski. "CVS Release History Data for Detecting Logical Couplings". In: *Proceedings of IWPSE 2003 (6th International Workshop on Principles of Software Evolution)*, pp. 13–23, IEEE Computer Society, 2003.

[Gall 97] H. Gall, M. Jazayeri, R. Klösch, and G. Trausmuth. "Software Evolution Observations Based on Product Release History". In: *Proceedings of ICSM 1997 (International Conference on Software Maintenance)*, pp. 160–, IEEE Computer Society, 1997.

[Gall 98] H. Gall, K. Hajek, and M. Jazayeri. "Detection of Logical Coupling Based on Product Release History". In: *Proceedings of ICSM 1998 (International Conference on Software Maintenance)*, pp. 190–198, IEEE CS Press, 1998.

[Gall 99] H. Gall, M. Jazayeri, and C. Riva. "Visualizing Software Release Histories: the Use of Color and Third Dimension". In: *Proceedings of ICSM 1999 (IEEE International Conference on Software Maintenance)*, pp. 99–108, IEEE Computer Society, 1999.

[Gamm 95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Girb 04] T. Gîrba, S. Ducasse, and M. Lanza. "Yesterday's Weather: Guiding Early Reverse Engineering Efforts by Summarizing the Evolution of Changes". In: *Proceedings of ICSM 2004 (20th IEEE International Conference on Software Maintenance)*, pp. 40–49, IEEE Computer Society, 2004.

[Girb 05a] T. Gîrba. *Modeling History to Understand Software Evolution*. PhD thesis, University of Berne, Berne, 2005.

[Girb 05b] T. Gîrba, A. Kuhn, M. Seeberger, and S. Ducasse. "How Developers Drive Software Evolution". In: *Proceedings of IWPSE 2005 (8th International Workshop on Principles of Software Evolution)*, pp. 113–122, IEEE CS Press, 2005.

[Girb 05c] T. Gîrba, M. Lanza, and S. Ducasse. "Characterizing the Evolution of Class Hierarchies". In: *Proceedings of CSMR 2005 (9th IEEE European Conference on Software Maintenance and Reengineering)*, pp. 2–11, IEEE CS Press, 2005.

[Grav 98] T. Graves and A. Mockus. "Inferring Change Effort From Configuration Management Databases". In: *Proceedings of the 5th International Software Metrics Symposium*, pp. 267 –273, Nov. 1998.

[Grin 96] R. Grinter. “Supporting Articulation Work Using Software Configuration Management Systems”. *Computer Supported Cooperative Work*, Vol. 5, No. 4, pp. 447–465, 1996.

[Guim 10] M. L. Guimarães and A. Rito-Silva. “Towards Real-time Integration”. In: *Proceedings of CHASE 2010 (2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering)*, pp. 56–63, ACM, 2010.

[Gutw 96] C. Gutwin, S. Greenberg, and M. Roseman. “Workspace Awareness in Real-Time Distributed Groupware: Framework, Widgets, and Evaluation”. In: *Proceedings of HCI 1996 (Conference on People and Computers)*, pp. 281–298, 1996.

[Guzz 09] A. Guzzi. *Supporting Collaboration Awareness in Multi-developer Projects*. Master’s thesis, University of Lugano, June 2009.

[Guzz 10] A. Guzzi, M. Pinzger, and A. van Deursen. “Combining Micro-blogging and IDE Interactions to Support Developers in their Quests”. In: *Proceedings of ICSM2010 (IEEE International Conference on Software Maintenance)*, pp. 1 –5, 2010.

[Guzz 11] A. Guzzi, L. Hattori, M. Lanza, M. Pinzger, and A. van Deursen. “Collective Code Bookmarks for Program Comprehension”. In: *Proceedings of ICPC 2011 (19th IEEE International Conference on Program Comprehension))*, pp. 101–110, IEEE Press, 2011.

[Habe 86] A. N. Habermann and D. Notkin. “Gandalf: Software Development Environments”. *IEEE Transactions on Software Engineering*, Vol. 12, Dec. 1986.

[Hage 08] J. Hagedorn, J. Hailpern, and K. G. Karahalios. “VCode and VData: Illustrating a New Framework for Supporting the Video Annotation Workflow”. In: *Proceedings of the AVI 2008 (Working Conference on Advanced Visual Interfaces)*, pp. 317–321, ACM, 2008.

[Hass 04] A. E. Hassan, R. C. Holt, and A. Mockus. “MSR 2004: International Workshop on Mining Software Repositories”. In: *Proceedings of ICSE 2004 (26th International Conference on Software Engineering)*, pp. 770–771, IEEE Computer Society, 2004.

[Hass 08] A. Hassan. “The Road Ahead for Mining Software Repositories”. In: *Proceedings of FoSM 2008 (Frontiers of Software Maintenance)*, pp. 48 –57, IEEE Computer Society, Oct. 2008.

[Hatt 08] L. Hattori and M. Lanza. “On the Nature of Commits”. In: *Proceedings of EVOL 2008 (4th International ERCIM Workshop on Software Evolution and Evolvability)*, pp. 63–71, IEEE CS Press, 2008.

[Hatt 09a] L. Hattori and M. Lanza. “An Environment for Synchronous Software Development”. In: *Proceedings of ICSE 2009 (31st ACM/IEEE International Conference on Software Engineering - New Ideas and Emerging Results Track)*, pp. 223–226, IEEE CS Press, 2009.

[Hatt 09b] L. Hattori and M. Lanza. “Mining the History of Synchronous Changes to Refine Code Ownership”. In: *Proceedings of MSR 2009 (6th IEEE Working Conference on Mining Software Repositories)*, pp. 141–150, IEEE CS Press, 2009.

[Hatt 10a] L. Hattori. “Enhancing Collaboration of Multi-developer Projects with Synchronous Changes”. In: *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering), Doctoral Symposium*, pp. 377–380, IEEE CS Press, 2010.

[Hatt 10b] L. Hattori and M. Lanza. “Syde: A Tool for Collaborative Software Development”. In: *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering)*, pp. 235–238, 2010.

[Hatt 10c] L. Hattori, M. Lanza, and R. Robbes. “Refining Code Ownership with Synchronous Changes”. *Empirical Software Engineering*, pp. 1–33, 2010.

[Hatt 10d] L. Hattori, M. Lungu, and M. Lanza. “Replaying Past Changes on Multi-developer Projects”. In: *Proceedings of IWPSE-EVOL 2010 (Joint 11th International Workshop on Principles of Software Evolution and 5th ERCIM Workshop on Software Evolution)*, pp. 13–22, 2010.

[Hatt 11a] L. Hattori. “Change-centric Support for Team Collaboration”. In: *Proceedings of GHC 2011 (Grace Hopper Celebration of Women in Computing)*, p. to be published, 2011.

[Hatt 11b] L. Hattori, A. Bacchelli, M. Lungu, and M. Lanza. “Erase and Rewind - Learning by Replaying Examples”. In: *Proceedings of CSEET 2011 (24th International Conference on Software Engineering Education and Training)*, p. 558, 2011.

[Hatt 11c] L. Hattori, M. D’Ambros, M. Lanza, and M. Lungu. “Software Evolution Comprehension: Replay to the Rescue”. In: *Proceedings of ICPC 2011 (19th IEEE International Conference on Program Comprehension))*, pp. 161–170, IEEE Press, 2011.

[Hatt 11d] L. Hattori, M. Lanza, and M. D’Ambros. “A qualitative analysis of preemptive conflict detection”. Tech. Rep. 2011/05, University of Lugano, Sep. 2011.

[Hegd 08] R. Hegde and P. Dewan. “Connecting Programming Environments to Support Ad-Hoc Collaboration”. In: *Proceedings of ASE 2008 (23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 178–187, IEEE CS Press, 2008.

[Herb 00] J. Herbsleb, A. Mockus, T. Finholt, and R. Grinter. “Distance, Dependencies, and Delay in a Global Collaboration”. In: *Proceedings of CSCW 2000 (ACM Conference on Computer Supported Cooperative Work)*, pp. 319–328, ACM Press, 2000.

[Herb 01] J. Herbsleb and D. Moitra. “Global Software Development”. *IEEE Software*, Vol. 18, No. 2, pp. 16 –20, March/April 2001.

[Hind 07] A. Hindle, Z. M. Jiang, W. Koleilat, M. W. Godfrey, and R. C. Holt. “YARN: Animating Software Evolution”. In: *Proceedings of VISSOFT 2007 (4th International Workshop on Visualizing Software for Understanding and Analysis)*, pp. 129–136, IEEE CS Press, 2007.

[Holt 96] R. Holt and J. Pak. “GASE: Visualizing Software Evolution-in-the-large”. In: *Proceedings of WCRE 1996 (3rd Working Conference on Reverse Engineering)*, pp. 163 –167, IEEE Computer Society, 1996.

[Hopp 80] G. Hopper. “Hopper (Grace) Oral History”. Available: http://www.computerhistory.org/collections/accession/102702026 [Accessed: Nov. 29, 2011], 1980.

[IBM 11] IBM. “Rational Team Concert”. [Software]. Available: http://jazz.net/projects/rational-team-concert/ [Accessed: Aug. 11, 2011], 2011.

[Kers 05] M. Kersten and G. C. Murphy. “Mylar: a Degree-of-Interest Model for IDEs”. In: *Proceedings of AOSD 2005 (4th International Conference on Aspect-oriented Software Development)*, pp. 159–168, ACM, 2005.

[Kers 06] M. Kersten and G. C. Murphy. “Using Task Context to Improve Programmer Productivity”. In: *Proceedings of SIGSOFT 2006/FSE-14 (14th ACM SIGSOFT International Symposium on Foundations of Software Engineering)*, pp. 1–11, ACM, 2006.

[Ko 07] A. J. Ko, R. DeLine, and G. Venolia. “Information Needs in Collocated Software Development Teams”. In: *Proceedings of ICSE 2007 (29th ACM/IEEE International Conference on Software Engineering)*, pp. 344–353, IEEE Computer Society, 2007.

[Kubi 94] E. C. Kubie. “Recollections of the First Software Company”. *IEEE Annals of the History of Computing*, Vol. 16, pp. 65–71, June 1994.

[Lanz 01] M. Lanza. “The Evolution Matrix: Recovering Software Evolution using Software Visualization Techniques”. In: *Proceedings of IWPSE 2001 (4th International Workshop on Principles of Software Evolution)*, pp. 37–42, ACM Press, 2001.

[Lanz 03] M. Lanza and S. Ducasse. “Polymetric Views — A Lightweight Visual Approach to Reverse Engineering”. *IEEE Transactions on Software Engineering*, Vol. 29, No. 9, pp. 782–795, Sep. 2003.

[Lanz 05] M. Lanza, S. Ducasse, H. Gall, and M. Pinzger. “CodeCrawler — An Information Visualization Tool for Program Comprehension”. In: *Proceedings of ICSE 2005 (27th IEEE International Conference on Software Engineering)*, pp. 672–673, ACM Press, 2005.

[Lanz 10] M. Lanza, L. Hattori, and A. Guzzi. “Supporting Collaboration Awareness with Real-time Visualization of Development Activity”. In: *Proceedings of CSMR 2010 (14th IEEE European Conference on Software Maintenance and Reengineering)*, pp. 207–216, IEEE CS Press, 2010.

[LaTo 06] T. D. LaToza, G. Venolia, and R. DeLine. “Maintaining Mental Models: a Study of Developer Work Habits”. In: *Proceedings of ICSE 2006 (28th ACM International Conference on Software Engineering)*, pp. 492–501, ACM, 2006.

[Lehm 80a] M. Lehman. “On Understanding Laws, Evolution, and Conservation in the Large Program Life Cycle”. *Journal of Systems and Software*, Vol. 1, pp. 213–221, 1980.

[Lehm 80b] M. Lehman. “Programs, Life cycles, and Laws of Software Evolution”. *Proceedings of the IEEE*, Vol. 68, No. 9, pp. 1060 – 1076, Sep. 1980.

[Lehm 85] M. M. Lehman and L. A. Belady, Eds. *Program Evolution: Processes of Software Change*. Academic Press Professional, Inc., 1985.

[Leve 66] V. I. Levenshtein. "Binary Codes Capable of Correcting Deletions, Insertions and Reversals". *Soviet Physics Doklady*, Vol. 10, pp. 707–710, February 1966.

[Lipp 92] E. Lippe and N. van Oosterom. "Operation-based Merging". *SIGSOFT Software Engineering Notes*, Vol. 17, No. 5, pp. 78–87, 1992.

[Lung 07] M. Lungu and M. Lanza. "Exploring Inter-Module Relationships in Evolving Software Systems". In: *Proceedings of CSMR 2007 (11th IEEE European Conference on Software Maintenance and Reengineering)*, pp. 91–100, IEEE CS Press, 2007.

[Ma 09] D. Ma, D. Schuler, T. Zimmermann, and J. Sillito. "Expert Recommendation with Usage Expertise". In: *Proceedings of ICSM 2009 (25th IEEE International Conference on Software Maintenance)*, pp. 535–538, IEEE Computer Society, 2009.

[Maal 10] W. Maalej and H.-J. Happel. "Can Development Work Describe Itself?". In: *Proceedings of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories)*, pp. 191 –200, IEEE Computer Society, 2010.

[Mant 11] MantisBT Team. "MantisBT". [Software]. Available: http://www.mantisbt.org/ [Accessed: Aug. 3, 2011], 2011.

[Matt 09] D. Matter, A. Kuhn, and O. Nierstrasz. "Assigning Bug Reports Using a Vocabulary-based Expertise Model of Developers". In: *Proceedings of MSR 2009 (6th International Working Conference on Mining Software Repositories)*, pp. 131–140, IEEE Computer Society, 2009.

[Mayr 95] A. von Mayrhauser and A. M. Vans. "Program Comprehension During Software Maintenance and Evolution". *Computer*, Vol. 28, pp. 44–55, 1995.

[McDo 00] D. W. McDonald and M. S. Ackerman. "Expertise Recommender: a Flexible Recommendation System and Architecture". In: *Proceedings of CSCW 2000 (ACM Conference on Computer Supported Cooperative Work)*, pp. 231–240, ACM Press, 2000.

[Mens 02] T. Mens. "A State-of-the-Art Survey on Software Merging". *IEEE Transactions on Software Engineering*, Vol. 28, No. 5, pp. 449–462, 2002.

[Micr 10] Microsoft Corporation. "Microsoft Visual Studio". [Software]. Available: http://www.microsoft.com/visualstudio [Accessed: Aug. 3, 2011], 2010.

[Mill 76] H. Mills. "Software Development". *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, pp. 265 – 273, dec 1976.

[Mock 00] A. Mockus and L. G. Votta. "Identifying Reasons for Software Changes Using Historic Databases". In: *Proceedings of ICSM 2000 (International Conference on Software Maintenance)*, p. 120, IEEE Computer Society, 2000.

[Mock 02] A. Mockus and J. D. Herbsleb. "Expertise Browser: a Quantitative Approach to Identifying Expertise". In: *Proceedings of ICSE 2002 (22nd International Conference on Software Engineering)*, pp. 503–512, IEEE Computer Society, 2002.

[Mozi 11] Mozilla Foundation. “Bugzilla”. [Software]. Available: http://www.bugzilla.org/ [Accessed: Aug. 3, 2011], 2011.

[Murr 09] J. Murre and A. Chessa. “Spurious Power Laws of Learning and Forgetting: Mathematical and Computational Analyses of Averaging Artifacts”. In: *Proceedings of CogSci 2009 (31st Annual Conference of the Cognitive Science Society)*, pp. 1175–1179, Cognitive Science Society, 2009.

[Neu 11] S. Neu, M. Lanza, L. Hattori, and M. D’Ambros. “Telling Stories about GNOME with Complicity”. In: *Proceedings of VISSOFT 2011 (6th IEEE International Workshop on Visualizing Software For Understanding and Analysis)*, pp. 14–21, IEEE CS Press, 2011.

[Nier 04] O. Nierstrasz. “Putting Change at the Center of the Software Process”. In: I. Crnkovic, J. A. Stafford, H. W. Schmidt, and K. Wallnau, Eds., *CBSE 2004*, pp. 1–4, Springer-Verlag, 2004. Extended abstract of an invited talk.

[Oezb 07] C. Oezbek and L. Prechelt. “JTourBus: Simplifying Program Understanding by Documentation that Provides Tours Through the Source Code”. In: *Proceedings of ICSM 2007 (IEEE International Conference on Software Maintenance)*, pp. 64 –73, IEEE Computer Society, oct 2007.

[Omor 08] T. Omori and K. Maruyama. “A Change-aware Development Environment by Recording Editing Operations of Source Code”. In: *Proceedings of MSR 2008 (5th International Working Conference on Mining Software Repositories)*, pp. 31–34, ACM Press, 2008.

[Orac 11] Oracle Corporation. “Netbeans”. [Software]. Available: http://netbeans.org/ [Accessed: Aug. 3, 2011], 2011.

[ORei 03] C. O’Reilly, P. Morrow, and D. Bustard. “Improving Conflict Detection in Optimistic Concurrency Control Models”. In: *Proceedings of the 2001 ICSE Workshops on SCM 2001, and SCM 2003 Conference on Software Configuration Management*, pp. 191–205, Springer-Verlag, 2003.

[OSul 09] B. O’Sullivan and O. Bryan. *Mercurial: The Definitive Guide*. O’Reilly Media, Inc., 2009.

[Parn 06] C. Parnin and C. Gorg. “Building Usage Contexts During Program Comprehension”. In: *Proceedings of ICPC 2006 (14th IEEE International Conference on Program Comprehension)*, pp. 13–22, IEEE Computer Society, 2006.

[Parn 10] C. Parnin and R. DeLine. “Evaluating Cues for Resuming Interrupted Programming Tasks”. In: *Proceedings of CHI 2010 (28th International Conference on Human Factors in Computing Systems)*, pp. 93–102, ACM Press, 2010.

[Parn 94] D. L. Parnas. “Software Aging”. In: *Proceedings of the ICSE 1994 (16th International Conference on Software Engineering)*, pp. 279–287, IEEE Computer Society Press, 1994.

[Paul 93] M. Paulk, B. Curtis, M. Chrissis, and C. Weber. “Capability maturity model, version 1.1”. *IEEE Software*, Vol. 10, No. 4, pp. 18 –27, july 1993.

[Paul 95] M. Paulk. *The Capability Maturity Model: Guidelines for Improving the Software Process*. *The SEI Series in Software Engineering*, Addison-Wesley Pub. Co., 1995.

[Perf 11] Perforce. "Perforce". [Software]. Available: http://www.perforce.com/ [Accessed: Aug. 11, 2011], 2011.

[Perr 01] D. E. Perry, H. P. Siy, and L. G. Votta. "Parallel Changes in Large-scale Software Development: an Observational Case Study". *ACM Transactions Software Engineering Methodology*, Vol. 10, No. 3, pp. 308–337, 2001.

[Perr 87] D. E. Perry. "Version Control in the Inscape Environment". In: *Proceedings of ICSE 1987 (9th International Conference on Software Engineering)*, pp. 142–149, IEEE Computer Society Press, 1987.

[Pila 08] C. Pilato, B. Collins-Sussman, and B. Fitzpatrick. *Version Control with Subversion*. *O'Reilly Series*, O'Reilly, 2008.

[Pinz 05] M. Pinzger, H. Gall, M. Fischer, and M. Lanza. "Visualizing Multiple Evolution Metrics". In: *Proceedings of SoftVis 2005 (2nd ACM Symposium on Software Visualization)*, pp. 67–75, ACM, 2005.

[Proe 10] T. Proenca, N. Moura, and A. van der Hoek. "On the Use of Emerging Design as a Basis for Knowledge Collaboration". *New Frontiers in Artificial Intelligence*, Vol. 6284, pp. 124–134, 2010.

[Puru 05] R. Purushothaman. "Toward Understanding the Rhetoric of Small Source Code Changes". *IEEE Transactions on Software Engineering*, Vol. 31, No. 6, pp. 511–526, 2005. Dewayne E. Perry.

[Quan 08] J. Quante. "Do Dynamic Object Process Graphs Support Program Understanding? - A Controlled Experiment". In: *Proceedings of ICPC 2008 (16th International Conference on Program Comprehension)*, pp. 73–82, IEEE CS Press, 2008.

[Rahm 11] F. Rahman and P. Devanbu. "Ownership, Experience and Defects: a Fine-grained Study of Authorship". In: *Proceedings of ICSE 2011 (33rd International Conference on Software Engineering)*, pp. 491–500, ACM, 2011.

[Rajl 00] V. T. Rajlich and K. H. Bennett. "A Staged Model for the Software Life Cycle". *Computer*, Vol. 33, pp. 66–71, 2000.

[Rapu 04] D. Rapu, S. Ducasse, T. Girba, and R. Marinescu. "Using History Information to Improve Design Flaws Detection". In: *Proceedings of CSMR 2004 (8th European Conference on Software Maintenance and Reengineering)*, pp. 223 – 232, IEEE Computer Society, 2004.

[Ratz 05] J. Ratzinger, M. Fischer, and H. Gall. "Improving Evolvability Through Refactoring". In: *Proceedings of the MSR 2005 (International Workshop on Mining Software Repositories)*, pp. 1–5, ACM, 2005.

[Raym 01] E. S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associates, Inc., 2001.

[Rigo 11] F. Rigotti. *Visualizing Software Systems and Team Activity*. Master's thesis, University of Lugano, Sep. 2011.

[Risi 00] L. Rising and N. S. Janoff. "The Scrum Software Development Process for Small Teams". *IEEE Software*, Vol. 17, pp. 26–32, July 2000.

[Robb 07a] R. Robbes. "Mining a Change-based Software Repository". In: *Proceedings of MSR 2007 (4th International Workshop on Mining Software Repositories)*, p. 15, ACM Press, 2007.

[Robb 07b] R. Robbes and M. Lanza. "Characterizing and Understanding Development Sessions". In: *Proceedings of ICPC 2007 (15th IEEE International Conference on Program Comprehension)*, pp. 155–164, IEEE CS Press, 2007.

[Robb 07c] R. Robbes, M. Lanza, and M. Lungu. "An Approach to Software Evolution Based on Semantic Change". In: *Proceedings of FASE 2007 (10th International Conference on Fundamental Approaches to Software Engineering)*, pp. 27–41, Springer, 2007.

[Robb 08a] R. Robbes. *Of Change and Software*. PhD thesis, University of Lugano, Switzerland, Dec. 2008.

[Robb 08b] R. Robbes and M. Lanza. "SpyWare: A change-Aware Development Toolset". In: *Proceedings of ICSE 2008 (30th ACM/IEEE International Conference in Software Engineering)*, pp. 847–850, ACM Press, 2008.

[Robb 08c] R. Robbes, D. Pollet, and M. Lanza. "Logical Coupling Based on Fine-Grained Change Information". In: *Proceedings of WCRE 2008 (15th IEEE Working Conference on Reverse Engineering)*, pp. 42 – 46, IEEE CS Press, 2008.

[Robb 10a] R. Robbes and M. Lanza. "How Program History Can Improve Code Completion". *Journal of Automated Software Engineering*, Vol. 17, No. 2, pp. 181–212, 2010.

[Robb 10b] R. Robbes, D. Pollet, and M. Lanza. "Replaying IDE Interactions to Evaluate and Improve Change Prediction Approaches". In: *Proceedings of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories)*, pp. 161 – 170, IEEE CS Press, 2010.

[Roch 75] M. J. Rochkind. "The Source Code Control System". *IEEE Transactions on Software Engineering*, Vol. 1, No. 4, pp. 364–370, 1975.

[Romp 08] B. van Rompaey and S. Demeyer. "Estimation of Test Code Changes Using Historical Release Data". In: *Proceedings of WCRE 2008 (15th Working Conference on Reverse Engineering)*, pp. 269–278, IEEE Computer Society, 2008.

[Royc 70] W. W. Royce. "Managing the Development of Large Software Systems: Concepts and Techniques". In: *Proceedings of the IEEE WESTCON*, pp. 1–9, 1970. Reprinted in *Proceedings* of the 9th International Conference on Software Engineering, March 1987, pp. 328–338.

[Rus 02] I. Rus and M. Lindvall. "Knowledge Management in Software Engineering". *IEEE Software*, Vol. 19, No. 3, pp. 26 –38, May/June 2002.

[Ryss 04] F. V. Rysselberghe and S. Demeyer. "Studying Software Evolution Information by Visualizing the Change History". In: *Proceedings of ICSM 2004 (20th IEEE International Conference on Software Maintenance)*, pp. 328–337, IEEE Computer Society, 2004.

[Sang 07] R. Sangwan, M. Bass, N. Mullick, D. Paulish, and J. Kazmeier. *Global Software Development Handbook*. Auerbach Publications, 2007.

[Sarm 07] A. Sarma, G. Bortis, and A. van der Hoek. "Towards Supporting Awareness of Indirect Conflicts Across Software Configuration Management Workspaces". In: *Proceedings of ASE 2007 (22nd IEEE/ACM International Conference on Automated Software Engineering)*, pp. 94–103, IEEE CS Press, 2007.

[Sarm 08] A. Sarma, D. Redmiles, and A. van der Hoek. "Empirical Evidence of the Benefits of Workspace Awareness in Software Configuration Management". In: *Proceedings of FSE 2008 (16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 113–123, ACM Press, 2008.

[Schn 04] K. A. Schneider, C. Gutwin, R. Penner, and D. Paquette. "Mining a Software Developer's Local Interaction History". In: *Proceedings of MSR 2004 (1st International Workshop on Mining Software Repositories)*, pp. 106–110, 2004.

[Schu 08] D. Schuler and T. Zimmermann. "Mining Usage Expertise from Version Archives". In: *Proceedings of MSR 2008 (International Working Conference on Mining Software Repositories)*, pp. 121–124, ACM, 2008.

[Serv 10] F. Servant, J. A. Jones, and A. van der Hoek. "CASI: Preventing Indirect Conflicts Through a Live Visualization". In: *Proceedings of CHASE 2010 (2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering)*, pp. 39–46, ACM, 2010.

[Sill 06] J. Sillito, G. C. Murphy, and K. D. Volder. "Questions programmers ask during software evolution tasks". In: *Proceedings of FSE-14 (14th International Symposium on Foundations of Software Engineering)*, pp. 23–34, ACM Press, 2006.

[Silv 06] I. da Silva, P. Chen, C. V. der Westhuizen, R. Ripley, and A. van der Hoek. "Lighthouse: Coordination Through Emerging Design". In: *Proceedings of ETX 2006 (OOPSLA Workshop on Eclipse Technology eXchange*, pp. 11–15, ACM Press, 2006.

[Sing 05] J. Singer, R. Elves, and M.-A. Storey. "NavTracks: Supporting Navigation in Software Maintenance". In: *Proceedings of ICSM 2005 (21st IEEE International Conference on Software Maintenance)*, pp. 325–334, IEEE Computer Society, 2005.

[Sing 98] J. Singer. "Practices of Software Maintenance". In: *Proceedings of ICSM 1998 (International Conference on Software Maintenance)*, pp. 139–, IEEE Computer Society, 1998.

[Souz 03] C. R. B. de Souza, D. Redmiles, and P. Dourish. "Breaking the Code, Moving Between Private and Public Work in Collaborative Software Development". In: *Proceedings of GROUP 2003 (International ACM SIGGROUP Conference on Supporting Group Work)*, pp. 105–114, ACM Press, 2003.

[Stal 85] R. Stallman. “The GNU Manifesto - GNU Project - Free Software Foundation (FSF)”. [Accessed: Nov. 30, 2011], March 1985.

[Stor 05] M.-A. D. Storey. “Theories, Methods and Tools in Program Comprehension: Past, Present and Future”. In: *Proceedings of IWPC 2005 (13th International Workshop on Program Comprehension)*, pp. 181 – 191, May 2005.

[Swan 76] E. B. Swanson. “The Dimensions of Maintenance”. In: *Proceedings of ICSE 1976 (2nd International Conference on Software Engineering)*, pp. 492–497, IEEE Computer Society Press, 1976.

[Tich 85] W. F. Tichy. “RCS – a System for Version Control”. *Software Practice & Experience*, Vol. 15, pp. 637–654, July 1985.

[Torv 92] L. Torvalds. “Release Notes for Linux v0.12”. The Linux Kernel Archives, 1992.

[Vesp 06] J. Vesperman. *Essential CVS*. *Essentials Series*, O’Reilly, 2006.

[Wett 08] R. Wettel and M. Lanza. “Visual Exploration of Large-Scale System Evolution”. In: *Proceedings of WCRE 2008 (15th IEEE Working Conference on Reverse Engineering)*, pp. 219–228, IEEE CS Press, 2008.

[Wett 10] R. Wettel. *Software Systems as Cities*. PhD thesis, University of Lugano, Switzerland, Sep. 2010.

[Wett 11] R. Wettel, M. Lanza, and R. Robbes. “Software Systems as Cities: A Controlled Experiment”. In: *Proceedings of ICSE 2011 (33rd International Conference on Software Engineering)*, pp. 551 – 560, ACM Press, 2011.

[Will 02] S. Williams. *Free as in Freedom: Richard Stallman’s Crusade for Free Software*. O’Reilly & Associates, Inc., 1 Ed., 2002.

[Wixt 07] T. Wixted John and K. Carpenter Shana. “The Wickelgren Power Law and the Ebbinghaus Savings Function”. *Psychological Science*, Vol. 18, No. 2, pp. 133–134, February 2007.

[Wohl 00] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: an Introduction*. Kluwer Academic Publishers, 2000.

[Xing 05] Z. Xing and E. Stroulia. “Analyzing the Evolutionary History of the Logical Design of Object-oriented Software”. *IEEE Transactions on Software Engineering*, Vol. 31, No. 10, pp. 850 – 868, Oct. 2005.

[Ying 04] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll. “Predicting Source Code Changes by Mining Change History”. *IEEE Transactions on Software Engineering*, Vol. 30, No. 9, pp. 574 – 586, Sep. 2004.

[Ying 11] A. T. Ying and M. P. Robillard. “The Influence of the Task on Programmer Behaviour”. In: *Proceedings of ICPC 2011 (19th IEEE International Conference on Program Comprehension)*, pp. 31–40, IEEE Press, 2011.

[Zaid 11] A. Zaidman, B. V. Rompaey, A. van Deursen, and S. Demeyer. "Studying the Co-evolution of Production and Test Code in Open Source and Industrial Developer Test Processes Through Repository Mining". *Empirical Software Engineering*, Vol. 16, pp. 325–364, 2011.

[Zell 07] A. Zeller. "The Future of Programming Environments: Integration, Synergy, and Assistance". In: *Proceedings of FOSE 2007 (2nd Conference on the Future of Software Engineering)*, pp. 316–325, IEEE CS Press, 2007.

[Zell 95] A. Zeller and G. Snelting. "Handling Version Sets Through Feature Logic". In: *Proceedings of the 5th European Software Engineering Conference*, pp. 191–204, Springer-Verlag, 1995.

[Zhou 08] Y. Zhou, M. Würsch, E. Giger, H. C. Gall, and J. Lü. "A Bayesian Network Based Approach for Change Coupling Prediction". In: *Proceedings of WCRE 2008 (15th Working Conference on Reverse Engineering)*, pp. 27–36, IEEE Computer Society, Washington, DC, USA, 2008.

[Zimm 04] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. "Mining Version Histories to Guide Software Changes". In: *Proceedings of ICSE 2004 (26th ACM International Conference on Software Engineering)*, pp. 563–572, IEEE CS Press, 2004.

[Zimm 05] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. "Mining Version Histories to Guide Software Changes". *IEEE Transactions on Software Engineering (TSE)*, Vol. 31, No. 6, pp. 429–445, 2005.

[Zou 06] L. Zou and M. W. Godfrey. "An Industrial Case Study of Program Artifacts Viewed During Maintenance Tasks". In: *Proceedings of the WCRE 2006 (13th Working Conference on Reverse Engineering)*, pp. 71–82, IEEE Computer Society, 2006.

[Zou 07] L. Zou, M. W. Godfrey, and A. E. Hassan. "Detecting Interaction Coupling from Task Interaction Histories". In: *Proceedings of ICPC (15th IEEE International Conference on Program Comprehension)*, pp. 135–144, IEEE Computer Society, 2007.

[Zumk 07] P. Zumkehr. *Changeboxes – Modeling Change as a First-class Entity*. Master's thesis, University of Bern, Feb. 2007.