Design and programming are human activities; forget that and all is lost.

– Bjarne Stroustrup

University of Alberta

Analyzing Individual Contribution and Collaboration in Student Software Teams

by

Fabio Rocha de Pinho

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Spring 2013
Edmonton, Alberta

Abstract

Software development is an inherently team-based activity, and many software-engineering courses are structured around team projects, in order to provide students with an authentic learning experience. In these courses, student developers define, share and manage their tasks, relying on collaborative-development tools. These tools support all lifecycle activities and generate a detailed record in the process, which can provide valuable insight into the team's work. In this sense, these tools enable instructors to monitor the team's progress and to understand and evaluate the contributions and overall performance of each individual team developer. This thesis describes an analysis and visualization framework, designed to enhance the usefulness of such collaborative tools for instructors, by enabling them to interactively explore interesting facets of the activities and contributions of each individual developer within the team. We have used our framework to analyze and understand how a student team worked through a term project. At the same time, we collected data about the students' own perception of their work processes through individual questionnaires and team interviews. Our analyses suggest that the inferences supported by our framework are congruent with the individual-developer feedback, while there are some discrepancies with the reflections of the team as a whole. Furthermore, in order to improve the learning experience of software engineering students, we studied the human and social aspects of a student team and we developed a grounded theory that helps on the understanding of how students behave during a course project.

Acknowledgements

First and foremost I would like to thank my supervisor, Dr. Eleni Stroulia, from whom I learned more than I ever thought I could learn from somebody, for her infinite patience, which was put to test multiple times, and for her genius. I am honored and flattered to have been your student and I will be forever grateful.

I would also like to thank Dr. Mario Nascimento, you are a true source of inspiration as a person, thank you for being a friend. Dr. Nikolaos Tsantalis, thank you for your friendly advices, criticisms, and encouragements. I will never forget the endless battles we had with Jazz and I would not have made this far without you. Dr. Nelson Amaral, thank you for seeing and believing in my potential, I would not be here without your help two years ago.

To Lucio Gutierrez and Himanshu Vashishtha, thank you for being the go-to persons whenever I had doubts about anything, from obscure technical stuff to please-just-google-it questions, I was lucky to have you around. To Natalia Negara, Morteza Ghandehari and Dan Han, thank you for hours and hours of conversations when I could not see the other side of the tunnel. Thank you all for being such wonderful colleagues, it was nothing but a pleasure being around you.

Thiago Arruda, Toluwanise Adesanwo, Kyrylo Shegeda and Peter Wood, I was blessed to have had all of you as my roommates over the past two years. Thank you for listening to me whine, for keep telling me I was nearly finished when I knew I was months away from it and, above all, feeding me when I was too busy for anything. You were my family in Edmonton and I am sure your spots in heaven are guaranteed. Thanks to Humberto Moreira and Bruno Lima, for believing in me when I did not. Your words kept me going when things were tough and you are the best friends I could have ever asked for. Some people are really angels on Earth. Thank you Jessica Castro, for being an angel in my life and putting up with me being such a big baby.

I would need to write a book to explain how grateful I am to my family. Thank you mom, for supporting all my decisions and always being there for me. I can feel your love even when we are far apart. Thank you dad, for being a friend and a role model in my life. I want to be like you when I grow up. Thank you both for all the sacrifices that you have made so I could have this opportunity. My brothers, Thiago and Delano, thank you for always being there for me when I need it, and for being there for mom and dad when I can not be.

To all of you and many others, you have deeply and positively impacted my life. This is for you.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Today, we are noticing two interesting trends relevant to the features and architecture of collaborative software development tools. On one hand, the new generation of these tools are no longer "closed" and tightly coupled with particular IDEs (Integrated Development Environments); instead, they are designed to be flexibly available, with browser-accessible architectures. At the same time, we are observing the increased adoption of "dashboards", i.e., analysis and visualization components that aim to help (a) developers become more aware of their team-members' work and how it may impact their own tasks, and (b) project managers monitor the project status and progress and evaluate the contributions and performance of each individual developer.

The construction of new tools to support software teams in their activities of developing software and managing their artifacts and process requires a solid understanding of the questions that the developers ask during these activities. Essentially, it becomes the objective of the tools to provide the right information so that these questions become easier to answer. The challenge then becomes to identify these questions. Several empirical studies have identified questions regarding different aspects of the software development process. Fritz and Murphy [11], for instance, through interviews with professional developers, identified 78 questions that span eight domains of information: source code, change sets, teams, work items, web sites and wiki pages, comments on work items, exception stack traces and test cases.

Myers and LaToza, on the other hand, focused on questions that developers find hard to answer (e.g., how did this run-time state occur?) and identified 67 questions that had not been previously reported [23].

Looking at these questions, as they arise from these two empirical studies, one can recognize several high-level categories emerging, corresponding to the different roles of the team members primarily interested in answering them. For example, since much software development is team-based, and often times distributed, some questions refer to the need of individual developers' to understand what other developers are currently doing to specific software assets. Yet others are of interest to project managers who inspect the project at a different scale: while developers may want to know whether a particular class has gone through testing, the project manager may want to know

how the team's progress compares against the project plan and whether the overall project is on time. In this thesis, we have yet another perspective: namely that of the instructor of a software-engineering course whose students participate in a team software project.

1.1 The Research Problem

More specifically, the research problem that this thesis investigates is "how student software teams work". In this investigation, we focus on identifying the questions of interest to the course instructor, that is, questions that are motivated by the need to understand the roles and the contributions of each individual team member to the project; the patterns of communication among the team members; and the dynamics of the team. More specifically we are interested in the following questions:

1. Was there a clear team leader throughout the project or did all members contribute equally to the project management?
2. What was the role of each member in the project?
3. How much effort did each member put into the project? How much time did each team member spend working on the project and what work did s/he specifically do?
4. How much did the team members communicate with each other? What tools did they use for communication?
5. How was the progress of the team members throughout the project lifecycle (i.e., did they do everything in the last minute or activities were fairly consistent during the project life span?)

In principle, the above questions can be (at least partially) answered through the information collected by the collaborative software-development tools mentioned above. These tools include a variety of components, each used by developers in the context of specific activities, such as issue-tracking systems which also serve for task management, source-code repositories, mailing lists and others. The usage history of these components is an information-rich data set that we can leverage to answer the questions above. And as these components are all integrated in a common framework, which is flexibly and therefore regularly accessible by the developers, their usage history contains relationships among developers and between developers and artifacts that would have not been otherwise possible to obtain. For example, a developer may identify a problem during testing which he may report as a bug in the issue-tracking system, including the stack-trace of the failed execution, and also communicate through the mailing list to the whole team. The discussion through the mailing list may identify a related bug, already in the issue-tracking system and may help yet another developer may resolve the two bugs through a change to a set of software modules, whose new versions are stored in the repository. Given an integrated collaborative software-development tool, the

links between the various elements of the above scenario (developer identities, bug IDs, and module identifiers) make it, in principle, possible to reverse engineer the complete story of how the team worked together to resolve the two bugs. One such tool that integrates several components for developers, managers and stakeholders is the IBM Jazz Collaboration platform[1]. The main objective of Jazz is to provide an infrastructure to enhance the quality of software development process for both individuals and teams. Therefore, products built on Jazz platform help teams from the following points of view[1]:

- *Collaboration:* Jazz tries to overcome the geographical and temporal gaps in the software development lifecycles in which distributed team members are involved. To that end, it leverages social networking capabilities to enable richer participation and collaboration between team members.

- *Automation:* Jazz enables automatic individual and team workflows by automating repetitive tasks to improve team efficiency. It also allows to standardize processes without adding overhead.

- *Reporting:* Jazz provides real-time insight into the software development process and supports the comparison of the project's progress with desired business outcomes. Accordingly, it can help team members to make better decisions at the time.

The Jazz platform is aimed to support the complete software lifecycle and to provide seamless integration of tasks across all phases of the software development process. Therefore, Jazz is considered as a framework that can integrate and synchronize people, processes, and assets involved in the software development process. Consequently, Jazz can make the software development process more collaborative, productive and transparent. Considering the extensibility of Jazz platform and the services that are provided for its extension, in this project we have targeted Jazz as the collaborative software development IDE we leverage in this work. Nevertheless our architecture is generic and could easily be tailored to support any similar collaborative software development IDE. Essentially, the technical objective of this thesis is to develop an *analysis-and-visualization service* for instructors of team-based project courses on software engineering. This is accomplished through an API that must be implemented by the chosen collaborative-development tool to extract relevant information from its various components (e.g., bug tracking, source code repository, etc). Once this API has been implemented, the data is consumed by the services and presented to consumers through another set of (REST) APIs. This layered architecture provides a simple model for developers to easily extend the default set of operations, utilize data from other tools (or even separate systems), and output data for different types of visualizations other than the ones directly provided by the dashboard of our service.

[1] http://www.jazz.net - Jazz Community Website

The analysis-and-visualization service of this thesis has been integrated with the Jazz platform to enable instructors using it in their software-engineering courses to answer some of the questions above. As the team uses Jazz, the collected usage data is analyzed to intuitively communicate information to the instructor (and the team members themselves) through a variety of interactive dashboards. The visualizations provide an overview of the status of the project, contributions of team members in terms of time and work, and communication channels used by the team, while still providing a way to drill down the data and trace back to the actual artifacts for further investigation if necessary.

In parallel to the technical objective of developing an analysis service for software-engineering course instructors, in the context of this thesis, we empirically studied the work of a student team through interviews, questionnaires and from the emails they exchanged throughout the term. We analyzed this qualitative data using a grounded-theory approach to study the social and human aspects of how students behave during a software engineering project. This kind of study is relatively new in the field of software engineering and may benefit the discipline not only by the results themselves but also by providing stepping stones to conducting such studies in a technical field.

1.2 Thesis Contributions

This work makes two important contributions. First, it presents a flexible software service for analyzing and visualizing the data collected by a collaborative software development tool, such as IBM Jazz. The toolkit is designed as a modular component that can be integrated with other similar tools, as long as they support exporting of similar data as required by our extraction mechanism, and it easily presents dashboards that help answering questions asked by developers, ultimately helping increasing team awareness. Furthermore, we present an empirical study in which we used the above toolkit to make inferences about the relative productivity and contribution of individual members in a software team. The findings of our study provide evidence that, indeed, an instructor inspecting the work of a student software team through the lens of our toolkit could draw conclusions with regard to which students assumed team-leadership roles and how the team may be further organized in smaller tightly collaborating subgroups.

The second contribution of this thesis is a grounded-theory study that we conducted to understand how senior software engineer students behave throughout the term. Historically, there has been limited research on the human and social aspects of software engineering. This work contributes to the field by presenting one case study of grounded theory in software engineering, which is important to the understanding of social interactions and behavior that affect the activities of individuals in our field. The result of this study was a set of pedagogical guidelines that may help instructors making students make the most of out software engineering courses.

1.3 Thesis Outline

The rest of this dissertation is organized as follows. In Chapter 2, we review the related work to this study. We start by surveying studies on analysis of programmers questions, and the tools that go about trying to answer them. Next we present some related work that uses data to understand the roles and performance of individuals in software teams. Finally, we present other studies that use grounded theory in software engineering.

In Chapter 3, we describe the our extensible framework's architecture, the REST services required for our architecture to work and the REST services we currently provide. We also discuss in details how we manipulate data and how we create different visualization facets around the same data items. We then discuss limitations of our framework and suggest future directions on improving this.

In Chapter 4, We provide a use case of how we leveraged IBM Jazz services to study a team of senior software engineering students and how we can use our visualizations to better understand the team instead of manually inspecting artifacts. Furthermore, we describe our grounded theory study, where we study the human and social behavior of a team of students to understand how they behave during a software engineering course project. We discuss the steps involved in conducting such a study, its results and we present a set of guidelines we think may be pedagogically useful for instructors of software engineering courses.

In Chapter 5, we conclude with a summary of our work and a review of the contributions of this study. We wrap-up the chapter by pointing out our plans for further studies.

Chapter 2

Background and Related Research

The work of this thesis touches several different but closely related areas, which are reviewed in this chapter. In order to understand how to improve existing collaborative software-development tools, it is important to know the questions commonly asked by team members as they work on their tasks. This thesis focuses on student software teams and the instructor's perspective in monitoring the team's work and progress through the use of interactive visualizations. To our knowledge, there has not yet been an empirical study looking at this perspective; of course, this perspective overlaps with that of the team developers and managers, and therefore previous empirical studies are clearly relevant, even though they may have not been investigating student teams. Section 2.1 provides a review on studies that identify different types of questions developers ask and tools that go about solving them. We focus, however, on the subset of questions to which answers helps on the understanding of the role and performance of individuals. Section 2.2 surveys studies that also have as goal contribution and productivity assessments. Finally, more closely related to our case study, in section 2.3 we present previous studies that also used grounded theory as a methodology to study the human and social aspects of software engineering.

2.1 Analysis of Programmer's Questions and Tools

The creation of new tools and IDEs that facilitate development and maintenance of software requires a deep understanding of the questions asked by developers during coding activities. Besides, almost all software development is team-based. Knowing these questions also helps one to understand how to increase awareness and productivity of software development teams.

Several empirical studies have identified questions regarding different aspects. Fritz and Murphy [11], for instance, through interviews with professional developers, identified 78 questions that span eight domains of information: source code, change sets, teams, work items, web sites and wiki pages, comments on work items, exception stack traces and test cases. These questions were divided into several high-level categories that correspond to areas of interest for developers (i.e., who's working on what, changes to the code, work item progress, etc). They then introduced an

information fragment model and a tool that automate the composition of different kinds of information (e.g.: source code, work items, change sets, etc) and help developers answer most of the questions through different (tabular) views of the composed information. Myers and LaToza, on the other hand, focused on questions that developers find hard to be answered (e.g., *how did this runtime state occur?*), rather than general questions, and identified 94 hard-to-answer questions, 67 of which that had not been previously reported. These questions were clustered into 21 categories. For each category they discussed examples of hard-to-answer questions and speculated tools that might help answer them. For instance, in the category *Architecture*, one hard-to-answer question is *What is the architecture of the code base?* and they suggest that although research tools exist to reverse engineer UML sequence diagrams, tool support to synchronize the diagrams with evolving code is limited [23].

Our visualizations help answer questions in the high-level categories described by Fritz and Murphy related to team members performance: *who's working on what*, *work item progress* and *other questions*. We chose questions that we believe may indicate performance of team members and their specific roles within the team which instructors, and team members themselves, would want to know about a project (e.g., the process followed by the team in practice) without having to manually inspect large amounts of data. The questions of interest for the purpose of this thesis are discussed in Section 4.1.4.

Many other projects have been developed with the purpose of directly (i.e., specifically addressing these questions) or indirectly (i.e., raise awareness and productivity) answering these questions. Atlassian JIRA[1], for instance, is a project-tracking tool for bugs/defects that allows to link issues to related source code, plan agile development, monitor activity and report on project status. Although JIRA also provides tools to generate and visualize reports, some of its reports are not well designed to produce overviews about individual team members, but rather the project as a whole. In our framework, this is quintessential, since we want to be able to address the performance of each student in the team.

DrProject [24] is a web-based portal for software project management that integrates revision history with issue tracking, mailing lists, wiki, and other features. It was designed to introduce students to the tools that professional development teams use to coordinate their work, and although some of the questions commonly asked by developers can be answered through the different integration views (textually), DrProject does not support visualizations.

At about the same time that Dr. project was being developed, our group was developing the WikiDev 2.0 Collaboration tool [2, 17, 9, 8, 10], which adopted a wiki as the central platform for integrating information about the various artifacts of interest (e.g., source code, bug entries in bugzilla), clustering this information in groups of relevant artifacts, and presenting views on the current project status and artifacts that cut across the individual tool boundaries. Much of the

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

conceptual groundwork for our current work on Jazz has been laid in WikiDev 2.0; in the context of the Jazz platform, we have been aiming at a much improved set of visualizations, with a general and flexible mechanism for configuring them to the various data that can be extracted from Jazz, to help answering a wider range of the questions interesting for developers.

The tools above help to indirectly answer some of the questions developers ask, by showing for instance, which code was committed by whom, or the history of the tasks of the project, but we focus on directly providing instructors with tools that help on productivity and contribution assessments, through the use of visualizations that show the performance of different team members, while still providing the ability of drilling down into the data and tracing back to the artifacts for further details, if necessary.

Hackystat[2] is an open source framework for collection and analysis of software development process and product data, where the developers have to install sensors (plug-ins) to their development tools (e.g., CVS, Eclipse, Atlassian Jira, etc), which will collect data and send it to a centralized repository accessible through web-services to form higher level abstractions of the data. The default analyses provided by Hackystat are mainly concerning metrics such as project coverage, complexity, coupling, commit frequency, commit size, etc. These analyses differ from ours, where we are interested primarily on metrics that help to assess individual contributions to the project, such as hours dedicated to certain types of activities, number of tasks completed and others.

Finally, the WorkItemExplorer by Treude et al. [26] is very similar in design and intent to our toolkit, as it provides an interactive environment to visually explore data from software development tasks of Jazz. WorkItemExplorer currently supports the analysis of these elements and the relationships between them: work items themselves, developers, iterations, project areas, team areas, tags, and comments. Our framework, however, presents interactive visualizations from a broader set of sources of data, besides work items, such as source code files, emails and about the process itself.

In table 2.1, we compared the tools mentioned above on a few key requirements that our analysis and visualization service has been designed to meet, described on the next chapter. The *reports* requirement refers to the ability of the tool to present information that can help answering programmer's questions in the form of visual reports, not textual data. It is also important that these reports are *customizable* by the user to answer slightly different questions than the default view. Another requirement checks whether the tools present *interactive* reports, where the user can interact with them through capabilities such as filtering, zooming and others. The *artifact* requirement analyzes the capability of the tool to link back to the original artifacts on its reports. Last, we analyze if the proposed solutions are integrated with collaborative software-development environments, either natively or through plugins. As we see from the table, it is not easy to find a single tool that satisfies all of the requirements above. Our toolkit meets all these requirements, which are crucial to delivering an easy, intuitive and powerful experience to a course instructor.

[2]http://code.google.com/p/hackystat/

Table 2.1: Comparison of tools for analysis of programmer's questions.

	DrProject	HackStat	Jazz	Jira	Wikidev 2.0	WorkItemExplorer
Reports	-	✓	✓	✓	✓	✓
Customizable	-	✓	✓	✓	✓	✓
Interactive	-	-	-	-	-	✓
Artifacts	-	-	✓	-	-	-
Integrated with IDE	-	-	✓	✓	-	-

2.2 Studies on Roles and Performance in Software Teams

Recently, with the increase of the amount of artifacts produced by team-based software projects, numerous studies have tried to facilitate or automate the task of measuring individual performance of team members as well as discover actual roles, as opposed to assigned roles, played by the individuals in the project. The primary goal of such studies is to answer more managerial questions, such as (a) if the project is going well (b) how the team members have distributed responsibilities among themselves and (c) who is contributing what and to what degree.

SEREBRO, by Hale et al. [16], for example, is an integrated tool that captures social (e.g., comments, agreements, disagreements) and development artifacts (e.g., SVN commits, wiki updates, file uploads) automatically and provides real time rewards (i.e., badges and titles) indicating a user's progress towards predefined goals. Using a variety of automated assessment measures (e.g.: *SEREBRO score*, *SEREBRO events*, etc), which evaluate the content and contribution of an individual on a project within a team, it ranks students according to their contributions. *SEREBRO score*, for instance, concentrates points on ideas, submitted on the SEREBRO tool, that are well received by the team and generate further discussion. It propagates points from child to parent, discounting them as the distance from the considered node increases. As reward thresholds are met, badges become publicly visible in the users profile. They show that such automated assessment measures are indeed performance indicators. We are interested, however, in showing the contribution of the different team members to the instructor through different interactive visualizations and let them rank students according to their own perception.

Serce et al. [25] use cluster analysis on communication artifacts of a globally distributed team to identify groups with similar communication patterns and suggest that particular patterns of communication behaviors are associated with higher performance.

Damian et al. [27] conceptualize coordination outcome by the result of their code integration build processes (successful or failed) and, using data from Jazz, study team communication structures with social network measures. They leveraged the combination of communication structure measures into a predictive model that indicates whether an integration will fail.

These studies are important for us because they validate the assumption that some metrics are indeed indicative of performance, such as student participation on events (e.g., code commit), content

quality and certain types of communication patterns (e.g., communication associated with contributions, such as commit comments), and relate closely to the information we can infer from our visualizations (e.g., creation of work items and time dedicated to the project are indicatives of performance). In our empirical study in the context of our senior software-engineering course, we studied how these objective metrics, as visualized through our toolkit, may correlate with the students' perception of their own work and contribution.

2.3 Grounded Theory in Software Engineering

Grounded Theory is a systematic generation of theory from data acquired by a rigorous qualitative research method [14, 15]. Grounded Theory method was developed by sociologists Barney Glaser and Anselm Strauss in the late 60's [12] and is a complete research method in the sense that it provides procedures that cover every stage of research, sampling participants, data collection and analysis, use of literature and writing.

The idea behind Grounded Theory is to investigate the main concern of a number of participants and create a theory to explain how they try to resolve this concern. This concern could be about any particular topic of the field that the researcher is interested in and that is important for the participants involved. The researcher has to uncover this main concern through data collection and analysis and present a theory that explains how people deal or try to resolve it.

Grounded Theory is a useful research method to study the human and social aspects of Software Engineering for many reasons. Firstly, it allows researchers to investigate social interactions and behavior through interviews with the study participants. Secondly, it is useful for research in areas where a new perspective would be beneficial and there has been relatively limited research on the human and social sides of Software Engineering.

Since its inception in 1967, Glaser and Strauss have disagreed on how to perform a grounded theory study, resulting in two distinct paradigms: the Straussian and the Glaserian paradigms. In this work we follow the Glaserian paradigm, which states that data sampling, data analysis and theory generation are not distinct but complementary steps that are to be repeated until one can explain the subject of research. The process of theory generation should stop when new data does not change the emerging theory. A grounded theory, following a Glaserian approach, consists of several steps, as presented in Figure 2.1 and described below.

1. *Light Literature Review:* The study can be initiated with a light literature review, if the researcher is not familiar with the area of interest and needs to acquire a vocabulary and knowledge sufficient to have a good conversation with participants in the data collection phase.

2. *Data Collection*: The data is collected in the field using Theoretical Sampling, which guides the research in the following iterations of data collection. Methods of data collection include interviews, observations and questionnaires.

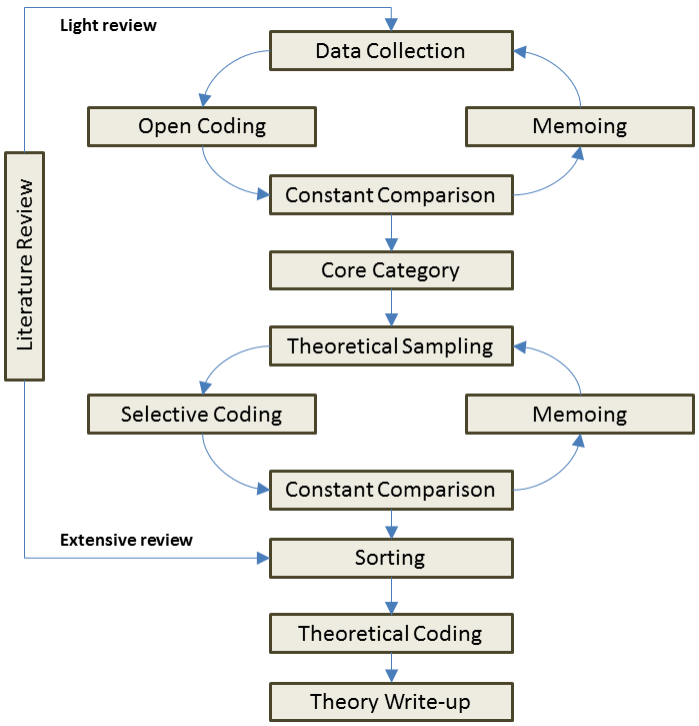

Figure 2.1: The grounded theory method

3. *Open Coding*: The data analysis starts with open coding, where the researcher tags the raw data, summarizing key points in one word or a small sentence.

4. *Constant Comparison*: During the data analysis, the codes arising from the raw data are constantly compared against the codes from the same interview and against the codes from previous iterations of data analysis in order to keep a consistent set of codes throughout the analysis. This constant comparison helps the researcher identifying recurrent and related codes to form a higher level of abstraction called a category.

5. *Memoing*: Memoing is the process of taking notes throughout the data analysis process. These notes (or memos) describe the implicit relationship between different categories and other reflections that the researcher might find useful to the development of the theory.

6. *Core Category Selection*: The theory is built mainly on top of one category called 'core category'. The core category is the one that relates meaningfully and strongly to the other categories that emerge from the data analysis.

7. *Extensive Literature Review*: After the theory starts to appear, the researcher conducts an extensive literature review to see how the literature in the field is related to the theory.

8. *Sorting*: After the theoretical saturation is reached (i.e., data collection stops to provide new insights), the researcher starts sorting out the memos and notes, in order to produce the outline

of the theory.

9. *Theoretical Coding*: Theoretical coding can be described as a framework to describe how categories relate to one another, as a hypothesis to be integrated into a theory [14].

10. *Theory write-up*: The last step it to write up the theory, which should come off naturally after the sorting and theoretical coding.

Grounded Theory has been gaining a lot of attention in the community over the past years with respect to studying the social sides of software engineering [1, 3, 4, 5, 19]. Since this method was formulated in the social sciences, it is still not widely understood in our community, where our strong technical and mathematical backgrounds prevail. Along with the related work mentioned in this section, this thesis contributes to this matter by presenting an empirical study that used grounded theory to study how a team of student developers perceive several factors that are of importance to the instructor when measuring their contribution and performance on the context of a software engineering project.

Adolph et al. [1] used classical (i.e., Glasarian) Grounded Theory in a software engineering context and provide a model for easy comprehension of the social sciences definitions present in Grounded Theory. They employed Grounded Theory to study how people manage the process of software development, how social processes influence the performance of software teams and how software methods influence those processes. They contribute to the community by enumerating a set of fifteen guidelines about the use of Grounded Theory for software engineering research.

Similarly, Coleman et al. [3] emerged in a Grounded Theory study to understand what was happening in practice in relation to software processes and software processes improvement (SPI). SPI studies the software process as it is used within an organization and drives the implementation of changes to that process to achieve specific goals such as increasing development speed. Using the Irish software product industry as subject for their grounded theory study, they found, for instance, that all of the companies are *tailoring* standard software processes to their own particular operating context such as the size of the company, the target market, and project and system type. Furthermore, they also discuss results of selecting and using Grounded Theory, and evaluate its effectiveness as a research methodology for software processes research.

When software engineers join a new project they have to become familiarized with a whole new set of things. Dagenais et al. [5] call this "a project landscape". Using Grounded Theory they studied newcomers of several projects in order to understand key factors that influence the experience of new team members of a software project. They theorize that three factors play a crucial role to a successful early experience for a new comer (and consequently for the people responsible for orienting them): (a) early experimentation (e.g., initial guidance and early feedback), (b) internalizing structures and cultures (e.g., building new relationships) and (c) progress validation (e.g., team feedback).

Finally, Hoda et al. [19], using Grounded Theory investigated how Agile teams, known to be "self-organizing", actually organize themselves in practice. Through interviews and observation of nearly thirty practitioners in several software organizations they suggest that six informal roles exist in the teams: mentor, translator, coordinator, champion, promoter and terminator. These roles could be used to help Agile software development teams organize themselves or guide Agile coaches in working with Agile teams.

2.4 Summary

In this chapter we discussed several areas that are intrinsically related to the work of this thesis. First, we reviewed several studies that identify questions commonly asked by developers and tools that try to answer some of them. Even though some tools have tried to address this particular problem, we try a different approach. Specifically, we want to improve the instructor's awareness through the use of visualizations that easily address some of the questions they ask when trying to understand the roles and the contributions of students to a software engineering project. Instead of using tables or static visualizations, our interactive visualizations enable the instructor to drill down into the data and trace back to the artifacts for further details, if necessary.

Next we reviewed several studies that focus on the same types of questions we are interested in, those to which answers provide some indication of productivity and contribution assessments. We share several things in common with these studies, such as the sources from where we extract information (i.e., source code repositories, task management systems, etc), some metrics that we think are indicative of contribution (e.g.: code commit) and the interest for social aspects of collaboration (e.g.: communication). However, most of these studies try to measure individual performance with techniques like algorithms, models and cluster analysis, presenting a set of results to the user. We are more interested in presenting data in the form of interesting and interactive visualizations in a way that is easy for the user to draw conclusions.

Last, more closely related to our grounded theory study, we surveyed several studies that utilize a grounded theory approach to perform data analysis. While grounded theory is a well known methodology in the social sciences, it is still in its infancy in the software engineering research community. We believe that if we are to understand how people perform and contribute to a software engineering project, we need to understand the social aspects of a software engineering team. We contribute to the field by adding one more study that utilized a grounded theory methodology to investigate how students behave throughout a software engineering course, which is presented in Chapter 4.

Chapter 3

Analysis and Visualization Service

In this work we have developed a framework for analyzing the products of collaborative software development tools for the purposes of providing instructors of software engineering courses with information that can support them in better understanding their students teams and the roles of the individual students in a project.

This framework is integrated with the IBM Jazz collaborative development infrastructure. The reason behind our choice is the wealth of data that Jazz stores about the team software process and the support it provides for consuming this data through a set of Representational State Transfer (REST) [7] APIs. Even though we used Jazz as the default data provider for this work, our framework is built with generality and extensibility principles in mind, meaning that any kind of collaborative software development IDE that provides similar instances of data works perfectly. In fact, data does not even have to be extracted from a single collaborative software system and could be extracted from different places like a source control system (e.g., CVS), a bug tracking system (e.g., Bugzilla) and other development tools.

Our framework consists of several modules, organized in a architecture diagrammatically depicted in Figure 3.1. It queries different services provided by the Jazz Team Server (e.g., work items service, source control service), as well the mailing list used by the team, processes the data and provides new REST services to be used by clients, which in our case are visualization widgets.

3.1 The Data Extraction and Analysis Layer

In this work we provide framework for analyzing and visualizing the data collected by collaborative software development tools. For these purposes, however, the raw data provided by collaborative software development tools tell us very little about team collaboration. To create an analysis of collaboration out of the data returned by these tools, we needed to go to a higher level of abstraction and take advantage of the links available in the data to provide a new set of services focused on team collaboration. This is done in the data extraction and analysis layer, which forms the bottom part of our framework.

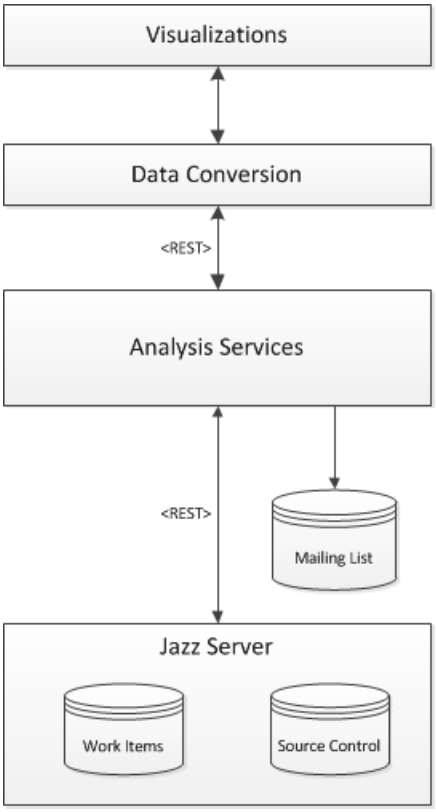

Figure 3.1: Design view of our architecture.

3.1.1 The Data Services

There are two main data services that the collaborative software development tools have to implement for our framework to work as intended. Tools that produce sets of related resources need the ability to look back in time to see how these resources developed throughout the project lifecycle. This is especially necessary if there is a need to present an aggregation of resources to the user as if it were a single resource. A good example of this is a specification document with its associated resources and its comments. It may be necessary for a tool to provide a view of the updates to the specification, its associated resources and comments for each revision of the specification. This defines a "history" service (i.e., a "revisions" service) that enables clients to retrieve these related resource event histories and corresponding resource revisions in such a way that they could determine what the resource states had been current at each revision of each resource.

In addition, similar to software artifacts, other lifecycle products, such as work items, are also expected to change over time, having multiple states along the entire software development process. This "work item" service provides the entire history of every work item stored in the repository, as well as pointers to different revisions of the same work item.

The *history* and *work item* services are the building blocks necessary for us to provide a useful

analysis service. Hence, collaborative software development Tools (or even different isolated tools) that want to take an advantage of our analysis services must provide these two services by implementing the data provision API. Figure 3.2 shows a UML diagram containing the data assumed by our framework. Appendix A describes each property separately.

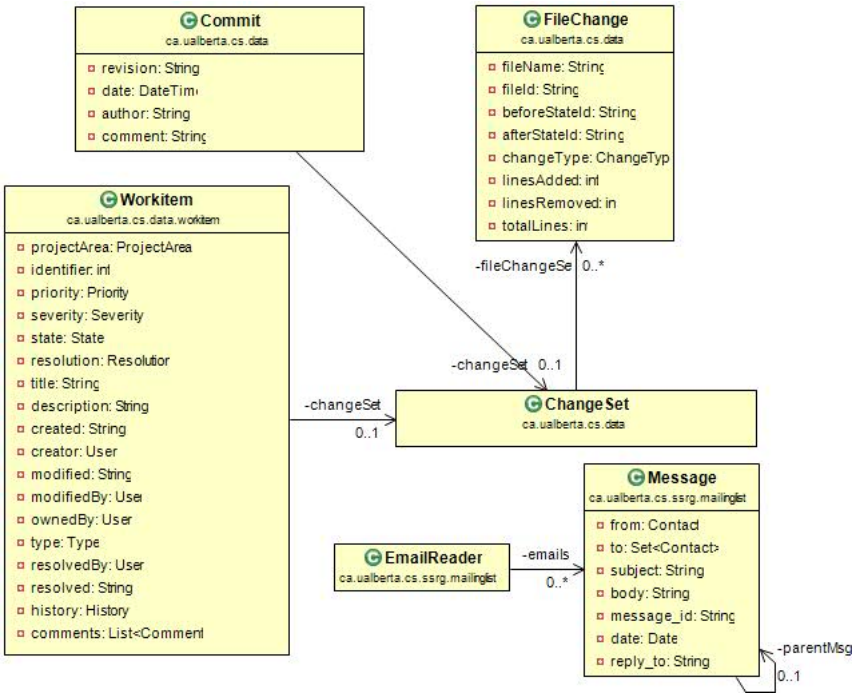

Figure 3.2: UML Diagram containing the data properties used in our framework.

3.1.2 The Data Provision API

The Data Provision API is conceptually simple and is represented by a set of objects and operations used by the analysis services. In this context, the main component is the abstract class *DataExtraction*, which encapsulates the mechanism of extraction of data from the tools to two different ends: (a) our object model or (b) a database. Figure 3.3 shows the implementation of the *DataExtraction* class. Note that there are two abstract methods that need to be implemented, *extractDataFromRepositoryToObjectModel* and *extractDataFromRepositoryToDatabase*. They are conceptually equal, the only difference is that while the first retrieves the data from the sources and stores everything in objects, the second stores everything in a database that is designed in the same way as our object model. Our object model is presented in Figure 3.4 and describes the common objects that must be instantiated and populated of when implementing a *DataExtraction* mechanism. Note that our object model is composed by: (a) the *DataExtraction* abstract class, which must be implemented; (b) a few helper classes such as *Repository, SourceControl and WorkItems*, that provide a few useful methods that are recurrently used in our analyses, such as *getAllUsers*, which retrieves all the users

involved in a project; and (c) the classes that model the data extracted from the data services.

For the purposes of this work, and as a means of verifying the flexibility of our framework, we implemented the *DataExtraction* API twice. The first for Jazz, *JazzDataExtraction*, was used for the empirical study discussed in the next chapter. The second was called *CustomDataExtraction* and extracted data from different tools, such as SVN and Bugzilla. These two different implementations were useful on identifying problems with our initial data model and making it more flexible for further implementations.

```
public abstract class DataExtraction {

    protected final String DB_CLASS = Constants.getValue("DB_CLASS");
    protected final String DATABASE_SERVER_ADDRESS = Constants.getValue("DATABASE_SERVER_ADDRESS");
    protected final String DATABASE_SERVER_STRING = "jdbc:mysql://" + DATABASE_SERVER_ADDRESS +"/";
    protected final String USERNAME = Constants.getValue("USERNAME");
    protected final String PASSWORD = Constants.getValue("PASSWORD");
    protected final String SVNPASS = Constants.getValue("SVNPASS");
    protected final String DATABASE_NAME = Constants.getValue("DATABASE_NAME");
    protected Connection connection = null;
    protected String URL;
    protected Repository repository;

    public DataExtraction(String url, boolean extractToObject) {

    public abstract Repository extractDataFromRepositoryToObjectModel();

    public abstract void extractDataFromRepositoryToDatabase();

    protected Connection getConnection() {

    protected static void InsertToDB(Connection dbconnection, String query) {

}
```

Figure 3.3: The abstract class *DataExtraction*. The base of our data extraction API.

3.2 The Data Conversion Layer

The top two layers of our architecture (as shown in in Figure 3.1) are responsible for translating the JSON output of the REST services into interactive visualizations that can be used by the instructors to recognize individual contribution in the context of a team project in order to more effectively support the project management and performance assessment.

Our visualization framework has been designed in terms of the three P's involved in a successful project management: *People, Products and Process*, similar to the PCANS model, proposed by Krackhardt and Carley [21] to represent the structure of an organization through the different relationships among three domain elements (individuals, resources and tasks). Following this concept, data within or across these dimensions is combined to provide a set of analyses that increase awareness about the project and create insight for instructors and students.

The *People vs. People* scenario, for instance, shows how team members compare to each other, both in an aggregated way or relatively over the time of the project. The remaining dimensions, in this context, become different views of analysis. For example, people can be compared in terms

of different product artifacts (work items, source code, documentation..) or process attributes (e.g., time spent on documentation).

The *People vs. Product* analyses provide information about the contribution of individuals in terms of software artifacts, making it possible to infer high-level roles played by each person in the project (i.e., developers, project manager, etc).

One of the most important requirements we set out for ourselves in developing our visualization toolkit is *high interactivity*. All visualizations are designed to represent an aggregated high-level view of different project data; however, it is always possible for the users to drill down and trace the aggregate views back to the individual data elements that compose the aggregate. This is done through the use of tooltips that appear when the user passes the mouse over items of a visualization (e.g, a bar in a bar chart), with links to the original individual resources stored in Jazz, as well as with filters that provide the user the ability to visualize specific contents of interest.

The second requirement is *flexibility*. Given that a multitude of visualizations may potentially be used to visualize a data set, we did not want to commit to a one-to-one mapping between our data and views. This is why, in the data conversion layer, fields from the JSON object (as returned by the REST calls to our visualization service) are mapped to fields of the visualizations, which translate directly to aspects of the visualization (e.g., node size, color, title, etc). This mapping, which we call a *visualization strategy*, can be conceptualized using a simple diagram (Figure 3.5) where all the fields of the data are listed on the left, all the fields required by the visualization are listed on the right and arrows represent the data mappings. Any given set of mappings for a particular dataset and visualization represents a different facet of the three P's described above.

The data-conversion layer is also responsible for maintaining a certain level of *consistency* across the various visualizations. Each different visualization widget is by default configured with different attributes, for example with respect to colors and sizes of elements. The data-conversion layer guarantees that data elements of interest (such as individual student names, for example) are represented with the same attribute (such as a consistent color) across all visualizations.

Another important functionality supported by this layer relates to the *scaling* of numerical data elements. Certain visualizations require numbers as input, (e.g. the size of a node or the width of an edge) but these numbers may require a specific format to make sense in the context of the visualization. Consider for example the case of numbers, to be represented by the size of nodes in a graph-like visualization; sometimes the numerical data is so large that, if it were to be passed directly to the size field of the node without alteration, the resulting node would occupy the entire screen. The data conversion layer handles this issue by scaling the input numerical data to have a desired mean and standard deviation (i.e., normalizing the values), thus formatting the numbers to make sense in the context of the visualization.

This layer ensures a consistent set of features supported by all widgets, by extending or modifying their default behavior through libraries. Namely, legends, filtering of the visualized data and

tooltips to support data drill-down are designed to be consistent across all widgets.

3.3 The Visualization Layer

The uppermost layer of the framework is the visualization layer, which lies on the client-side and is composed by a set of visualization widgets. It consumes data created by our analysis services and treated by our data conversion layer, which is implemented on the server-side as a set of analysis services that are invoked by the widgets.

3.3.1 The Analysis Services API

The Analysis Services API can be implemented to provide analysis services for different types of clients, which in our case are visualization widgets. The analysis Services API can be visualized in Figure 3.6 and consists of several methods for different types of services, which are relevant for different types of visualization widgets. For instance, the *getWIDistributionStats* method is part of the temporal distribution service and is relevant for temporal visualizations. The different types of analysis services represented in our API are detailed below.

Operation Distribution Service The *Operation Distribution Service* works for both artifacts and lifecycle resources. It is responsible for transforming the raw data returned by both the work item service and history service into an aggregated data composed by each type of operation on artifacts (e.g., add, delete, modify) or work items (e.g., open, close, modify).

Temporal Distribution Service The evolution of any kind of artifact or work item is crucial to understand how the team is collaborating and how the project is evolving. The *Temporal Distribution Service* transforms the raw data into data aggregated by periods of time (e.g., day, week, month).

Communication Service The informal communications among the members of the team about their project is a key determinant to its success or failure and has been deeply studied in recent years [6, 18]. In the *Communication Service*, we provide an initial exploration of the communication artifacts of the team project, both formal (e.g., work item comments) and informal (e.g., emails) to give instructors the possibility to understand how the team coordinated activities on the project lifecycle.

Evolution Service The *Evolution Service* finds patterns of co-evolving software-artifacts (e.g., classes that are committed together in some fraction of the overall commits), providing information such as class inter-dependencies and highly coupled designs, which in turn could point out opportunities for design improvement.

We expose the implementation of our analysis services as a REST API to be consumed by clients. After the implementation is deployed to a web server, the analysis services are accessible via the HTTP method GET. For instance, the *getRepositoryDistributionStats* method is accessible through:

```
https://localhost:9443/jazz/service/ca.ualberta.cs.ssrg.common.internal.
rest.ITeamAnalysesRestService/repositoryDistributionStats
```

Parameters can be passed normally through the URL. In the example below we pass the workspace identification number as the ID of the repository to get the repository distribution analysis from.

```
https://localhost:9443/jazz/service/ca.ualberta.cs.ssrg.common.internal.
rest.ITeamAnalysesRestService/\\repositoryDistributionStats?workspaceId=_r_
0oYIQcEeGhI-XoVcR95g
```

All methods return JSON objects containing the results of the analyses, which can be consumed by clients of any type. In our case, our JSON results are handled by the data conversion layer, and ultimately by the visualization widgets.

3.3.2 Visualization Widgets

This part of the framework takes the converted and formatted data (output of the data conversion layer) and turns it into something visual. The actual visualization can be any widget that takes a JSON as input and displays it in a visual manner. We have already developed a collection of widgets, which we present below and discuss in the context of our empirical study in the next chapter. Nevertheless, it is our intention to expand our widget collection, and to explore which mappings from data to views users find more informative. In this work, we have implemented three different types of visualizations: (a) Aggregation Visualizations, (b) Temporal Visualizations and (c) Relationship Visualizations.

Aggregation Visualizations Instructors usually want to see the contribution of students in a format that is easy understand. One way of presenting such information is by aggregating each individual interaction of a student with the tools. For instance, we can present an aggregation of all code commits performed by the students, in a way that it is easy for the instructor to visualize *how much* students are contributing. These *Aggregation Visualizations* consist of several widgets that present aggregated data on different facets of software development assets. We have implemented aggregation visualizations for the following assets: source code files, documentation files, design files and work items.

Figure 3.7 presents one such visualization, which consists of the aggregated time spent by students on different project tasks, such as documentation, coding and testing. In this figure, we see that *Student3* logged 175.25 hours of activities in the project, where a great chunk of it was in coding activities, mostly in work items 206 and 205, but also contributed to documentation activities. *Student5*, however, indicates very little contribution to documentation activities. In fact, so little that the portion of it is insignificant in the image, which leaves its title not visible.

Temporal Visualizations Another important information for team members and instructors is how people contribute to the project *over time*. These *Temporal Visualizations* can be helpful in many ways. On assessing the performance of one student, for example, it is important to know if that particular student contributed to the project on all of its phases or just on a particular part. Another example is to understand the actual process followed by the team, by looking at how different assets were touched during the lifetime of the project.

In this work we implemented a few different temporal visualizations, such as the temporal distribution of work items, source code files, documentation files and design files. Figure 3.8 presents one example of a temporal visualization, namely, the temporal distribution of work items, with information about their creation, modification and resolution by students. Each event is displayed in a timeline, where the user can navigate to see how often and how much students performed certain tasks throughout the project. This snapshot, in particular, presents the last three days before the last deadline imposed by the instructor. It is visible that students were heavily working on these days.

Relationship Visualizations A third essential type of information to understand how team members collaborate is the *relationship among project entities*. The *Relationship Visualizations* are used to present this information in an intuitive way. Our implementations of this family of visualizations include the relationship of individuals based on the emails exchanged by them during the course, as shown in Figure 3.9, and relationships of people and artifacts (i.e., artifacts owned by students). In the image presented we see the amount of emails exchanged (i.e., replied to in the mailing list) of one of *Student5* with the other team members. We perceive that *Student5* replies to more emails from *Student2* and *Student4*.

3.3.3 Dashboards

The visualizations mentioned above are useful on creating insight from different kinds of data. More often than not, however, visualizations create even more value when presented together, forming a *dashboard*. For our purposes, it is no different. Instructors need to know different kinds of information before making a decision on the contribution of a student. For instance, while a student might have closed more work items, he or she might have not contributed so much to the code part of the project. It is also important for the instructor to be able to visualize these things in a easy way,

and this is where our conversion layer plays a key role on keeping things consistent throughout all visualizations.

On the other hand, students also might need different kinds of information when trying to gain awareness about the project and other team mates. These, might or might not be congruent with the information needed by instructors, which are usually more interested in a less granular view of the data, as opposed to team members. This creates the need for different kinds of dashboards, which are essentially the same but with different filter and granularity default settings.

In this work we implemented the instructor dashboard, which was used in the context of our empirical study. It is formed by all the visualizations described in the previous section and contains consistent colors for the students and filters that have effect on all visualizations.

In the future we want to extend our dashboard with other functionalities for both instructors and students. For students, we envision an annotation feature, where students can annotate points of interest in the visualizations, possibly leading to interesting discussions with the other team members. For instructors, we want, for instance, to provide the ability of comparing (i.e., highlighting) multiple students on visualizations and comparing different teams. This could provide valuable insight for instructors on how different individuals (and teams) are performing during the course.

3.4 Integration with Jazz

IBM Rational Jazz is an extensible technology platform for collaborative software development. It is designed to support seamless integration of tasks across all phases of software lifecycle and to be extensible both on the client and the server. In this work we take advantage of its extensibility on both ends to provide a new set of services that can be consumed by visualizations.

One of the main components of the Jazz architecture is the Jazz Team Server (JTS). The JTS provides fundamental services that enables easy integration, and an uniform experience for users, of all software built upon the Jazz architecture. These fundamental services are called Jazz Foundation Services (JFS) and include user management, security, data query and other generic capabilities.

There are many products which are built on the Jazz platform, each of which leverages a set of capabilities for collaborative team-based software development. A few examples are: IBM Rational Team Concert, Rational Quality Manager, Rational Requirements Composer and Rational Asset Manager. In this work we focus on Rational Team Concert (RTC) which is a collaborative work environment for developers, architects and project managers, with unified work items, source control, build management, process management and iteration planning. RTC stores rich sets of information, created by developers throughout the development cycle of an application, which can be leveraged in order to increase team awareness.

The Jazz REST Services (JRS), both foundation and tool-specific, feature a Resource-Oriented (i.e., RESTful) Architecture and provide web services designed for general consumption by arbitrary clients. These web services provide stable programmatic web-based “APIs” for directly accessing

the facilities and data offered by the various Jazz components. Furthermore, they use a combination of URIs, HTTP methods, and standard representation languages such as XML and JSON, that work like the rest of the web. The protocol is stateless meaning that client state is managed in the client and server state is reflected directly in the resources. This kind of architecture is important for us because ensures that our REST services always receive updated information about team artifacts from the server.

A REST API is required to provide three key things: stable URLs for the tool data resources, documentation for those data resources, and a protocol and operations for manipulating those data resources based on standard HTTP methods (i.e., PUT, POST, GET, DELETE). Providing URLs for the tool data resources enables these resources to be linked from anywhere, which distinguishes REST from SOAP/WS-*. In a REST architecture, data resources become "hyper-data" just like hypertext enables fully connected, flexible content (e.g., text, images, etc). As with all APIs, REST APIs must be carefully designed with both stability and future evolution in mind. The long-term stability of URLs is of great importance since without it broken links, among other things, prevent data resources from being accessed, which is a serious issue.

3.4.1 Extending the Jazz Team Server

A JTS Extension is a tool-specific functionality which is affiliated with a particular JTS instance. Each tool uses certain operations provided via Jazz Foundation Services (JFS) in order to interact within the JTS, and exposes its own services and data through RESTful web services, as shown in Figure 3.10. These, when combined with the REST services of JFS, give a JTS a uniform way of exposing all available functionality to clients. This allows clients to treat a Jazz Team Server as a unit (i.e., a set of interconnected REST services), without having to be concerned with how the services are provided.

At the heart of our framework lies its data-collection mechanism. In our Jazz Extension, we implemented this by extending the *DataExtraction* API discussed earlier. The *JazzDataExtraction* communicates to the different native JFS provided by the Jazz Server through the OSLC REST Consumer. The OSLC REST Consumer is responsible for the HTTP calls to retrieve the Resource Description Framework (RDF) [22] based documents stored in the Jazz Server. Moreover, it is responsible for parsing the RDFs, extracting the required fields, and encapsulating them in our Java objects. The subsections below describe the services used by our framework.

One example of the data returned by Jazz can be found in Figure 3.11, that shows a segment of the XML document representing the history of one artifact in Jazz, which in this case is a requirements document. From the Jazz *History Service* we extract the following set of information: (a) files committed to the repository; (b) author (i.e.: committer), date, and type (e.g., addition, modification) of each commit; (c) work item associated and comments made by the author at the time of commit.

The second service necessary for our framework is the *Work Item* service. Figure 3.12 shows a

segment of the XML document representing the history of one work item in Jazz. The information extracted from this service includes (a) the description of the work item, its type and author, (b) the type, date and comments around each modification of the work item; and (c) the time each developer spent working on this work item. Table 3.1 summarizes the information returned by the Jazz REST services and used in our framework.

Table 3.1: The Jazz Services used by this work, the information of interest and one sample entry returned by each of them.

REST Service	Info Returned	Example
History Service	File	CMPUT 401 Project Summary.docx
	Committer	Student4
	Date of modification	Jan 30, 2012 1:19 PM
	Type of modification	Addition
	Work Item associated	96
	Comment	First draft is done. A little over 500 words, but I don't see that being an issue.
Work Item Service	Work Item Id	96
	Description	Project Requirements Document - Project Summary
	Author	Student4
	Type of Work Item	Task
	Type of modification	Creation
	Date	Jan 27, 2012 9:16 PM
	Comment	(Comment omitted – too long)
	Time estimated	(No time estimated)
	Time per developer	Student4 – 5 hours

3.5 Summary

The work presented in this chapter presents a flexible framework for analyzing and visualizing the data collected by a collaborative software development IDE. However, it is necessary that the collaborative software development IDE implements two APIs, the *Data Provision API* and the *Analysis Services API*. The first is responsible for collecting the data required to answer questions commonly asked by developers, managers and instructors: repository-history and work-item data. The second is responsible for creating the analyses and providing the visualization layer with meaningful information, rather than raw data.

Furthermore, we presented the three types of visualizations we implemented in this work: aggregation visualizations, temporal visualizations and relationship visualizations. The visualizations provide an overview of the status of the project, contributions of team members in terms of time and work, and communication channels used by the team, while still providing a way to drill down the data and trace back to the actual artifacts for further investigation, if necessary.

Last, we presented the IBM Rational Jazz Framework and explained how we extended it with our analysis and visualization services.

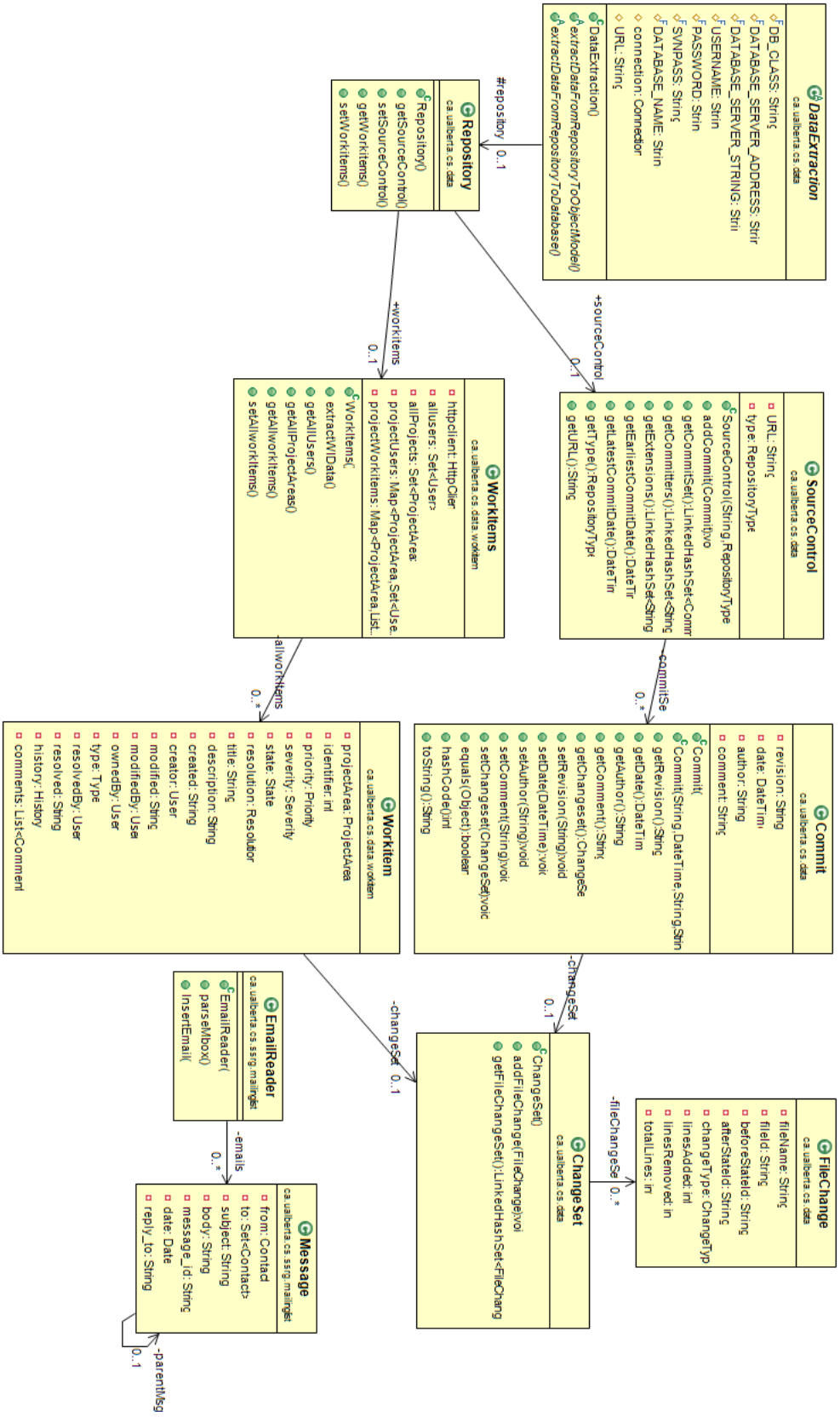

Figure 3.4: Our object model.

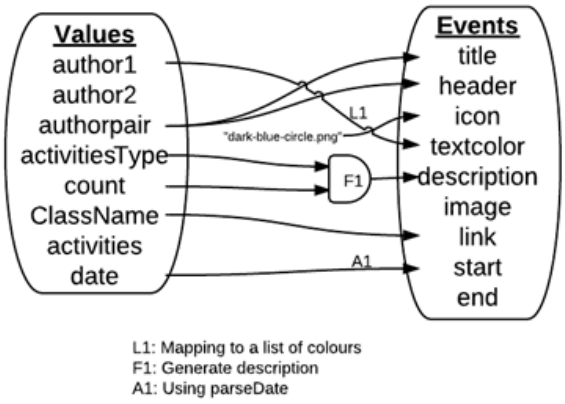

Figure 3.5: A mapping of the JSON attributes returned by the data layer into the attributes required by a visualization tool.

```
public interface ITeamAnalysesRestService {

    public String getHelp() throws TeamRepositoryException;

    public RepositoryOperationDP[] getRepositoryOperationStats (ParmsAnalyses params) throws TeamRepositoryException;

    public WIOperationDP[] getWIOperationStats (ParmsAnalyses params) throws TeamRepositoryException;

    public RepositoryDistributionDP[] getRepositoryDistributionStats(ParmsAnalyses params) throws TeamRepositoryException;

    public ArtifactDistributionDP[] getArtifactDistributionStats(ParmsArtifacts params) throws TeamRepositoryException;

    public WIDistributionDP[] getWIDistributionStats(ParmsAnalyses params) throws TeamRepositoryException;

    public CoEvolutionDP[] getCoEvolutionStats(ParmsCoEvolution params) throws TeamRepositoryException;

    public DesignDiffDP[] getDesignDiff(ParmsDesignDiff params) throws TeamRepositoryException;

    public ActivitiesStatsDP[] getActivitiesStats(ParmsAnalyses params) throws TeamRepositoryException;

    public static final class ParmsArtifacts implements IParameterWrapper {

    public static final class ParmsDesignDiff implements IParameterWrapper {

    public static final class ParmsCoEvolution implements IParameterWrapper {

    public static final class ParmsAnalyses implements IParameterWrapper {

}
```

Figure 3.6: Analysis Services API.

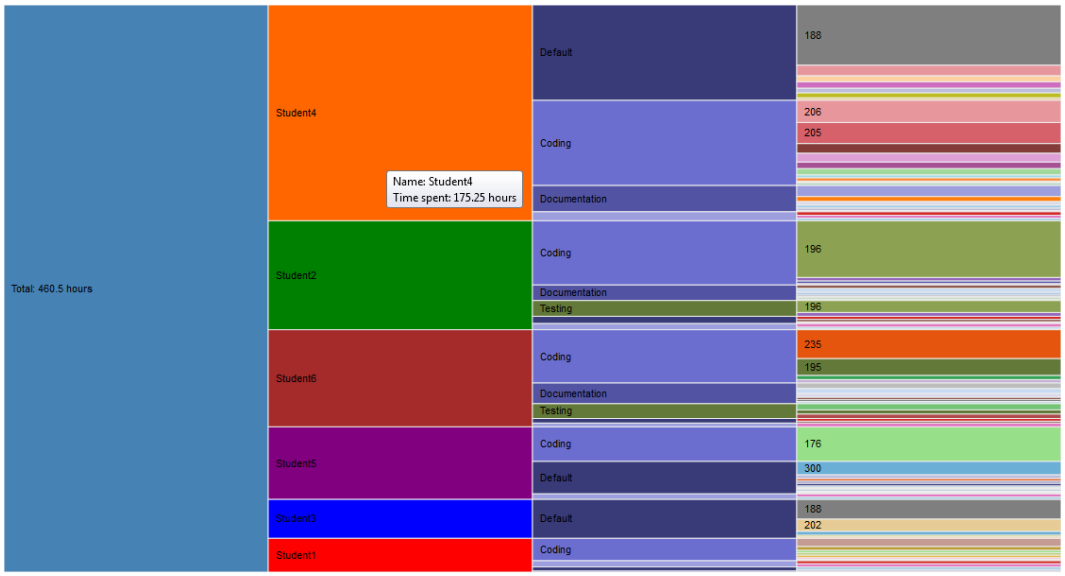

Figure 3.7: Example of one aggregation visualization.

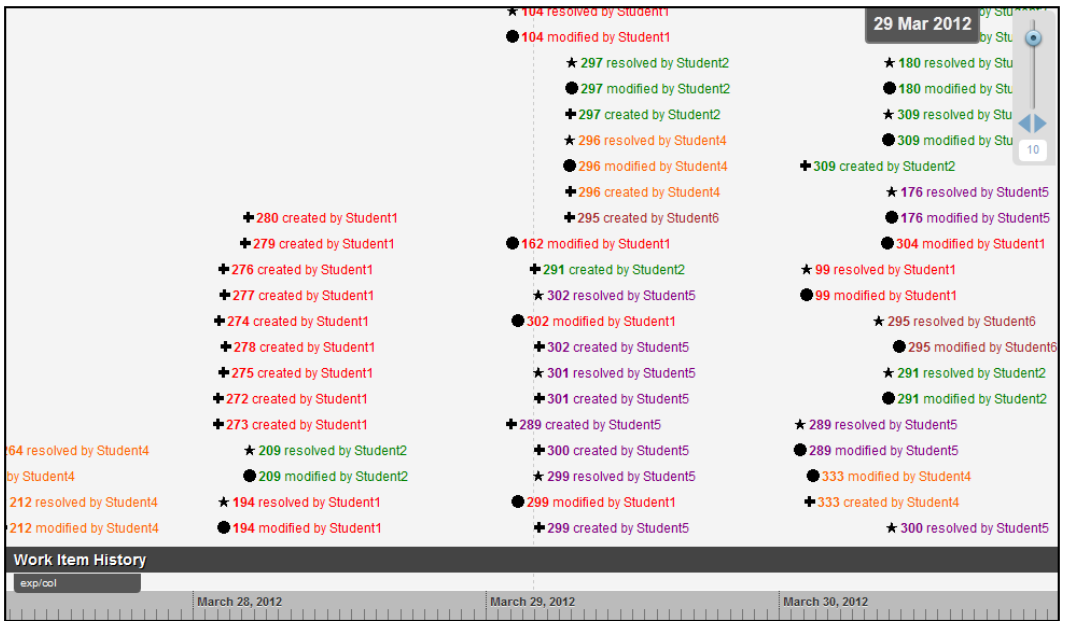

Figure 3.8: Example of one temporal visualization.

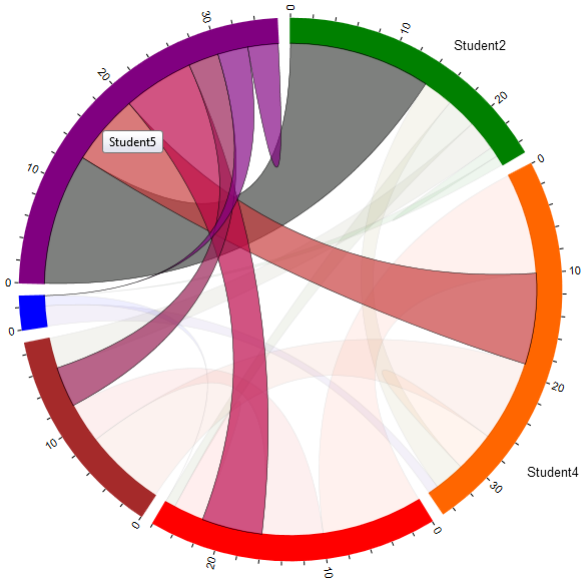

Figure 3.9: Example of one relationship visualization.

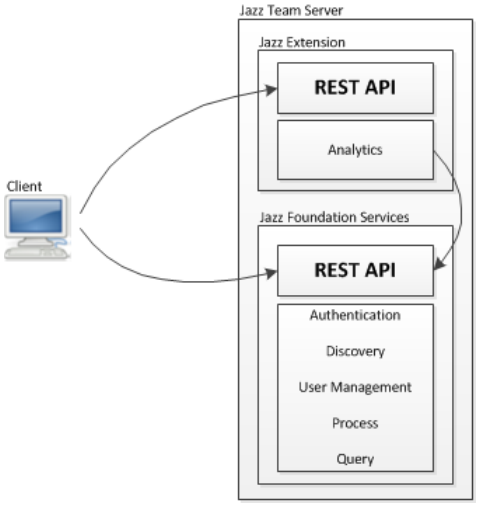

Figure 3.10: Interactions in Jazz.

Figure 3.11: Sample history data from Jazz.

```
− <interface>
    com.ibm.team.workitem.common.internal.rest.IWorkItemRestService
  </interface>
− <returnValue xsi:type="com.ibm.team.repository.common.services:ComplexDataArg">
    <type>COMPLEX</type>
  − <value xsi:type="workitem.restDTO:WorkItemDTO">
      <itemId>_IpINKklvEeG8D7D2dAuFEQ</itemId>
      <stateId>_36busFEjEeGphKqWvEWC4Q</stateId>
    − <locationUri>
        https://localhost:9443/jazz/resource/itemName/com.ibm.team.workitem.WorkItem/96
      </locationUri>
    + <linkTypes></linkTypes>
    + <linkTypes></linkTypes>
    + <linkTypes></linkTypes>
    + <linkTypes></linkTypes>
    + <attributes></attributes>
    + <attributes></attributes>
    + <attributes></attributes>
    + <attributes></attributes>
    − <attributes>
        <key>creationDate</key>
      − <value xsi:type="workitem.query.restDTO:UIItemDTO">
          <label>Jan 27, 2012 10:16 PM</label>
        + <iconUrl></iconUrl>
          <id>2012-01-28T05:16:51.551Z</id>
        </value>
      </attributes>
    − <attributes>
        <key>creator</key>
      − <value xsi:type="workitem.restDTO:ContributorDTO">
          <itemId>_u8pcVz1DEeG2fYHGNp-f3g</itemId>
          <userId>          </userId>
          <name>                    </name>
          <emailAddress>                          </emailAddress>
        + <locationUri></locationUri>
        </value>
      </attributes>
    − <attributes>
        <key>description</key>
```


Figure 3.12: Sample work item data from Jazz.

Chapter 4

The Empirical and Grounded Theory Studies

Having developed our Jazz-based data analysis and visualization framework, we used it to study whether and how it may support an instructor in better understanding students' collaboration in the context of a senior undergraduate software engineering course (i.e., CMPUT401). We also examined how the students themselves perceive the contributions of, and collaboration with, their peers. More specifically, we studied a six-member student team that committed to using Jazz and its many functionalities to manage their collaborative software development.

The senior software engineering course is four months long and is structured with three hours of lecture per week covering fundamental software engineering topics. The major objective is to apply software engineering principles, methodologies, and tools in the creation and release of a significant piece of software. Students form teams and interact with a client to gather requirements, develop prototypes and deliver a fully functional product. The clients are users who need a software solution to a real problem they have.

Furthermore, the course traditionally includes a sequence of (two or three) questionnaires over the term, as a means of collecting the perceptions of individual students on how they and their teammates perform in the team project. In addition to these questionnaires, we collected data through team interviews and we compared and synthesized the information extracted through these students' reports with the information implied by our Jazz-based visualization toolkit. We used this qualitative data using a grounded-theory approach to study the social and human aspects of how students behave during a software engineering project.

This chapter is organized as follows. Section 4.1 presents our empirical study, how data was collected and analyzed. Section 4.2 presents our Grounded Theory and the analysis of the data following grounded theory procedures. We then present the theory that emerged from our grounded theory study, a few examples and some discussion. Lastly, we present related work to the theory that emerged and set of pedagogical guidelines that may help instructors making students make the most out of software engineering courses. Finally, this chapter is concluded in Section 4.3.

4.1 The Empirical Study

This section presents our empirical study, where we used our toolkit to draw conclusions with respect to five different questions about role and performance of software engineering team members. The rest of this section details the five activities of our study: (a) automatic data collection, that spanned the entire project lifecycle; manual data collection, that consisted of (b) interviews and (c) questionnaires, conducted halfway through the project and at the end; (d) analysis, where we compare insights provided by the visualizations against the students' perceptions gathered from the interviews and questionnaires; and (e) our findings and our interpretation of them.

4.1.1 Automatic Data Collection

Using our toolkit, throughout the course term, we collected the history and work-item data from the team Jazz area and examined it through its visualization widgets.

Regarding work items, we were interested on which types of work items students were working (e.g, tasks, defects, milestones), and the types of activities students were performing on them (e.g., opening, closing, resolving). This helps instructors to better understand the role of different students on the project. For instance, a student that opens a relatively high number of work items related to tasks when compared to others might be the project manager of that particular project, even if no explicit roles are defined within the team. Another example is a student that opens many *defect* work items. In this case this student probably plays the role of a tester in the team. We also examined information about different types of artifacts (e.g., source code files, UML files, UI files, etc) as well as activities performed on them (e.g., creation, deletion, modification). This information helps to identify more fine grained roles, for instance, the work items tell us that if two students are working on work items of the type task, those students are probably developers, but the information from different artifacts might show which kind of specific role (i.e., tester, documenter, code writer) the students are performing.

We also collected information about the process followed by the team, such as delivery dates, milestones and time spent on work items. This type of information helps the understanding of certain behaviors during the project lifecycle (i.e., a peak on number of commits). More importantly, the time information helps instructors to verify the amount of time put into the project by different students. While this information is submitted by students themselves and may or may not be correct, when put in context of the information provided by all visualizations, they tend to be consistent and quite useful in helping to measure effort. It is important to note that this information is provided by the work item service, through work items of certain types (e.g., milestone, deliverable, etc).

The last type of information collected from students was related to communication. We monitored the mailing list used by the team, as well as comments made by students at the time of code commits and work item updates. We used this data to extract information of whom each student communicated more with and how much, as well as discover what was the main form of communi-

cation among team members. This type of information can be used in different ways, for instance, to find out which students are more communicative with the customer or individuals who are collaborating more closely in a remote way. Lastly, information about communication patterns within teams can be used to find out how team members organize themselves in smaller teams to perform the tasks of a project.

4.1.2 Interviews

During the course of our study, we conducted two interviews with the student developers, each for with different objectives. First, halfway through the project, we conducted interviews with each of the six team members individually. The objective of this round was to validate the visualizations widgets we had chosen as a means of intuitively communicating information about the project; essentially, we were interested in assessing whether the visualizations were meaningful (from the students' perspective) and to identify potentially new analyses and visualizations to include in the toolkit. Each interview with a team member was conducted by two interviewers (one of them exclusively taking notes), lasted 30 minutes and was audio recorded. The recordings were then transcribed and used for investigation, along with the notes taken by the interviewers.

The second interview was conducted at the end of the project, with the team as a whole, and had two main goals. The first objective was to help students reflect on their post-mortem analysis of the project by walking them through the various visualizations and discussing with them their implications. The second objective was to evaluate the usefulness of these visualizations in providing insights about individual team member's roles and effort throughout the project. To that end, before the interview, the author of this work (who was neither an instructor nor a TA for the course) reviewed the visualizations and, based on his interpretation of the views, he answered the same questions that the student team was to answer. In this manner, we established an unbiased baseline of what our interpretation of the visualizations was. This baseline was later compared with the interviews and the questionnaires submitted individually by the students to the instructor. The interview was conducted by one interviewer (the author of this work), lasted one hour and was also audio recorded. The recordings were also transcribed and used for investigation, along with the notes taken during the interview and the questions answered by the interviewer a priori. The interview questions of this study can be found in Appendix B.

4.1.3 Questionnaires

The use of questionnaires is one of the traditional ways for the instructor to receive feedback of the students on the course as well as on their peers. For the course in question, questionnaires were made available after each major course milestone. The first was the deliverable of the initial project documentation (i.e., requirements documents, project plan, etc), a few weeks into the course, and the second, after the end of the project, consisted of the complete documentation and final

demonstrations in class.

From the questionnaires we wanted students to answer questions related to their roles in the project, their impression on the work of their peers and about the process (i.e., what went wrong, what could be improved, what they would have done differently), which is basically what we want to address with our visualizations. The questionnaires used in this study can be found in Appendix C.

4.1.4 Analysis

With the analysis of the data we collected (Jazz-derived data, questionnaires and interviews), we wanted to answer the following questions:

1. Was there a clear team leader throughout the project or did all members contribute equally to the project management?
2. What was the role of each member in the project?
3. How much effort did each member put into the project? How much time did each team member spend working on the project and what work did s/he specifically do?
4. How much did the team members communicate with each other? What tools did they use for communication?
5. How was the progress of the team members throughout the project lifecycle (i.e., did they do everything in the last minute or activities were fairly consistent during the project life span?)

The first three questions are interesting from the instructor's perspective because they help the instructor understand the role, as well as the individual contribution of each student, to the project. The last two questions help the instructor to understand the dynamics of the team and identify possible issues early on. These questions were answered by the author, based solely the visualizations, and then compared to: (a) a post-mortem interview with the complete team and (b) the individual peer review from the students submitted to the instructor.

4.1.5 Results and Discussion

As we have already mentioned, our primary goal in the first round of interviews was to get some initial feedback on the validity of the visualizations (as a means of understanding the relative contribution of individuals within a team). From that round of interviews, we learned two lessons. First, absolute numbers are not necessarily a good proxy measure of the effort an individual put into the project (i.e., team members that commit more or more frequently are not necessarily the ones who work harder). Second, the tooltips that enabled the user to drill down the data and inspect the details around the tasks performed by the team members were very useful in improving our inferences regarding teamwork and individual productivity (i.e., not only we could tell that a student was a developer, but also that the student was working on a specific target platform, for instance, the Android

platform). We used this feedback we obtained from the students as a guide for improving the visualizations, before the second interview, where one main goal was to identify roles and contributions of each team member to the project.

A very interesting observation we made during our second round of interviews was that what the students said often times did not match what they wrote on their individual peer reviews. Since this study was not connected in any way to their marks in the course, we suspect that this happened because students did not feel comfortable answering some of the questions of the interview in front of their peers (e.g., if one of the team members acted as a leader throughout the project or members contributed equally to the project management).

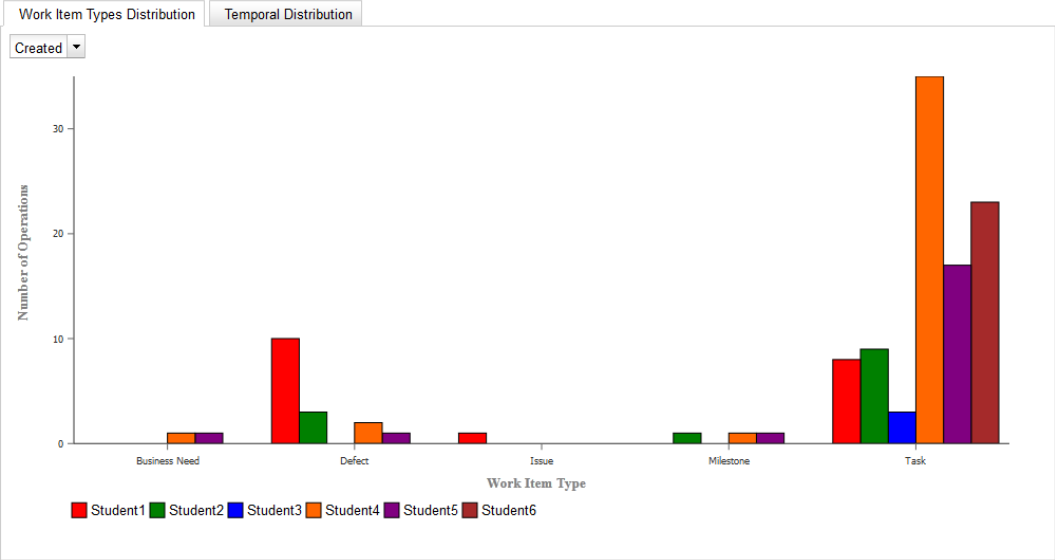

Figure 4.1: View of the amount of work items created by each student segmented by work item type. *student4* created most of the tasks of the project.

1. Was there a clear team leader throughout the project or did all members contribute equally to the project management?

When taking a look at the work item visualizations (Figure 4.1) we inferred that student4 was contributing more to project management by creating most of the tasks of the project (and assigning them to different persons, based on the tooltips, not shown in the image for clarity). However, on their peer reviews, the students did not see student4 as a leader, and pointed out that all decisions were made through a democratic process and consensus. Whether the instructor would see the person with a higher amount of created work items as a leader for creating most of the tasks is a point of debate, but the visualizations do point out that specific team members may have played distinct management roles and could be perceived as leaders.

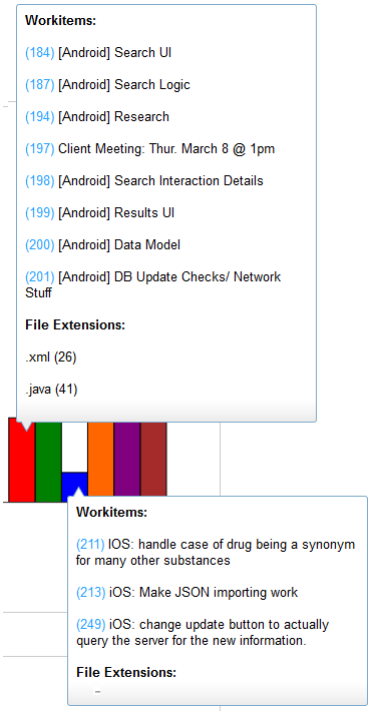

Figure 4.2: Tooltip details of the work items owned by the users. The tooltips display links to the work items, their description and extensions of files associated with the work items (if any). This image shows that two different students were involved in two different parts of the system (Android and iOS).

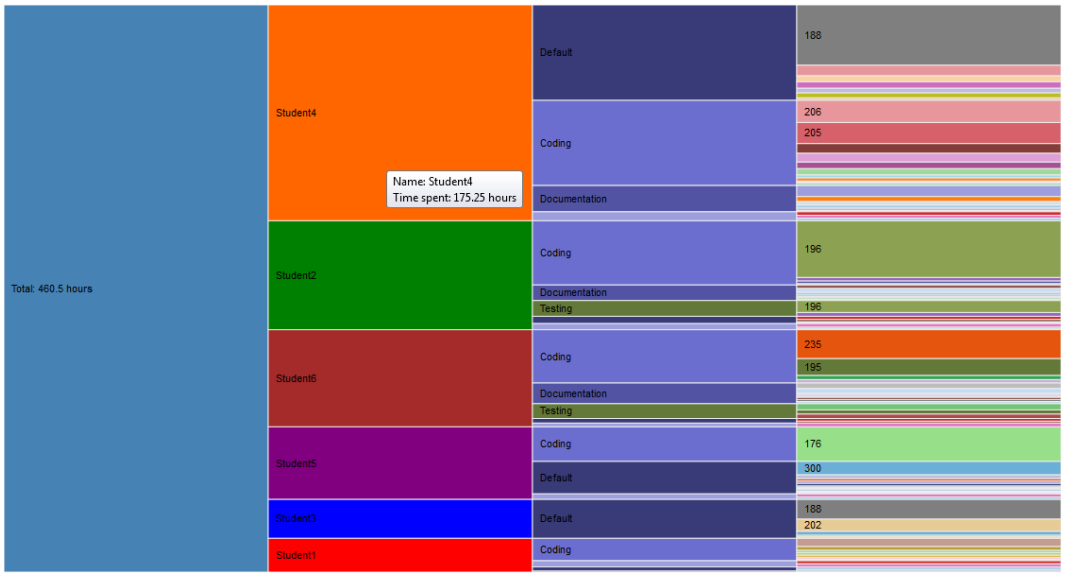

Figure 4.3: Visualization that illustrates the time spent by each student in the project, segmented by activity and work items. *student4* logged 175.25 hours of the 460.5 hours logged by the team.

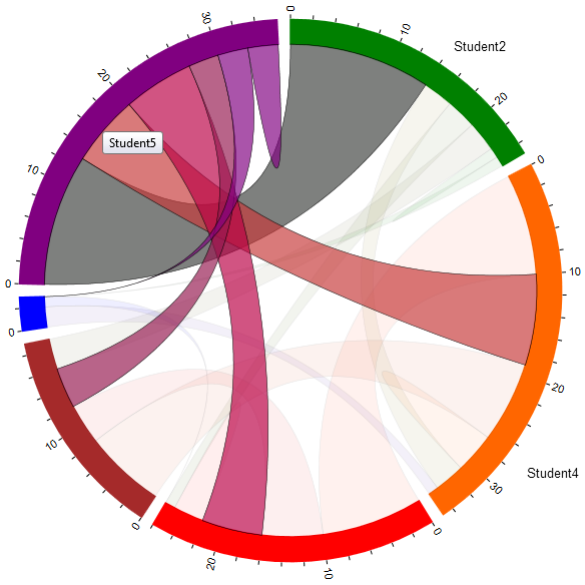

Figure 4.4: Chord diagram that illustrates email communication to/from Student5. Using this diagram we noticed that students in different subteams interacted more through email while students in the same subteams preferred other channels of communication.

2. What was the role of each member in the project?

The visualizations led us to inferences regarding the specific role of each member of the team. We observed that a number of work items and artifacts were associated with pairs of team members. Namely, one pair of students had the majority of Android work items and artifacts (Figure 4.2), another pair had iPhone items associated with their activities and a third pair of students was involved with Java and web development items. From this data, we inferred that students had subdivided into pairs to tackle the different platforms of their project. This was confirmed with the interview and peer reviews.

Student2: We have divided our group into 3 pairs based on the development platforms. We have one group doing iOS, one for Android, and one group working on the web-based administration.

3. How much effort did each member put into the project? How much time did each team member spend working on the project and what work did s/he specifically do?

Figure 4.3 shows the time spent by each student in the project, segmented by activity (i.e., documentation, coding, testing or other, shown as default) and within an activity segmented by work item. The size of each rectangle is relative to the number of hours that the individual worked.

From this visualization we could identify that one student (student4) was putting more time in

the project, with nearly twice as much as the second student. While hours worked does not equal contribution to the project, the discrepancy in the hours worked by one of the students in comparison to the others clearly shows that one of the students had a higher workload. Although on the team interview students agreed that the team had equal contribution to the project, the following snippets extracted from the students' peer reviews show that student4 was indeed seen by the group (and himself) as having worked more than the others.

student2: I don't know how productive people actually were, but if I were asked to guess... student4 *had the most groundwork to cover.*

student5: I wouldn't say that student4 *worked the hardest necessarily, but it took him the most time to wrap his head around the system.*

student4: I was the most productive team member for I had the most to research, learn and did the most of the programming on the iOS side and I was the only one who really actually tracked his progress in Jazz.

4. How much did the team members communicate with each other? What tools did they use for communication?

Regarding the process followed by the team during the course of the project, we noticed that they maintained a fairly consistent communication among them via email, and at a smaller scale, via work-item comments. We also noticed that email communication was higher among members who were not collaborating closely (i.e., members of different subteams). These facts were confirmed and justified by the team members on the interview and peer reviews. Email was the main form of communication among all members and stakeholders because it was the easiest way to broadcast messages, while members of the same subteams preferred to have chat conversations or collaborate face-to-face. Figure 4.4 illustrates this scenario by showing a chord diagram where a student in one of the subteams exchanges a higher amount of emails with two other students, none of them being his subteam partner. This behavior was similar to the other students as well, except for student4, that had a good amount of communication with all other team members (again exhibiting evidence of leadership).

student4: Email was by far much more used than work items.

student5: That was like a lot of person to person.

student2: As a group we used the weekly meetings and the team's email account to communicate. As for communication between me and student1 it was done with skype and calling each others cell phones.

5. How was the progress of the team members throughout the project lifecycle (i.e., did they do everything in the last minute or activities were fairly consistent during the project life span?

Instructors are often interested in knowing how the team is collaborating along the way, in order to identify potential problems early on. In this setting, Figure 4.5 shows the activity history of the work items during the project in the form of a timeline where each entry is an action performed by

one student. Inconsistent/irregular activity patterns might raise concern. Our team exhibited a fairly consistent activity throughout the project; however, the activity-visualization widget made evident for us that considerably more effort was devoted to the project just before deadlines. This was also confirmed by students on the interview and peer reviews. We suspect that our visualizations may also help identifying other smaller deadlines (e.g., the ends of sprints, preparation of a customer demo) but for the team we worked with, since the team was fairly consistent throughout the project, only the course deadlines were apparent.

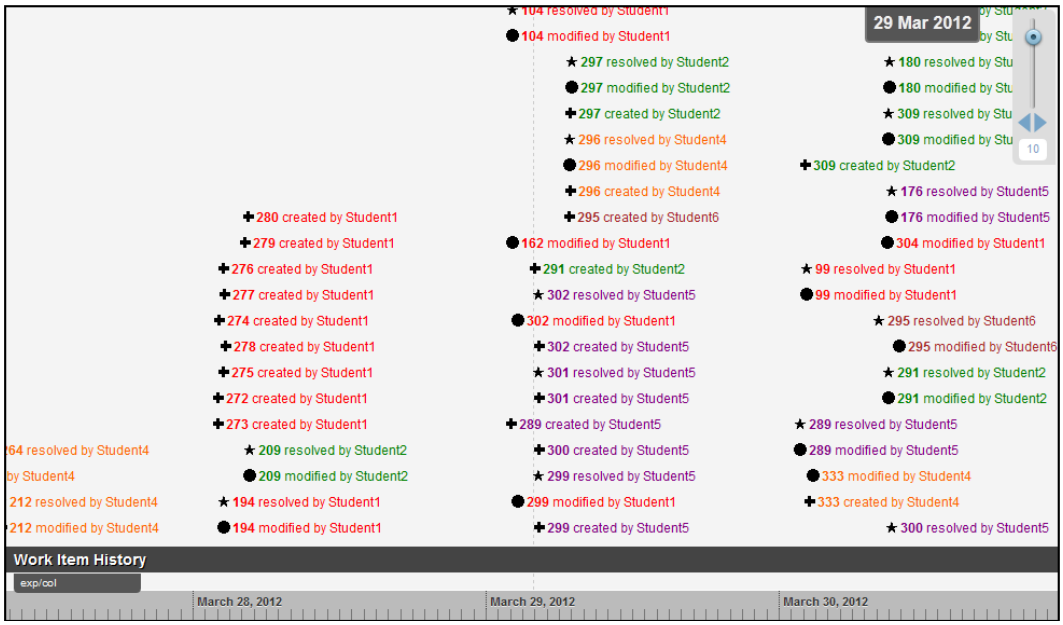

Figure 4.5: This timeline shows the work item activity history. A peak is observed before the final project deadline (March 30th).

Interviewer: So I have another view which is a kind of a timeline. In this view I want to understand if the peaks reflect the deadlines you have for the course. I can see a big concentration of activity at the end of March.

student2: Yeah. The 30th was the last week of code.

student5: 30th was when it was due, yeah.

student6: That reflected the deadline. That's his question. The 30th right there.

Interviewer: And this one.

student4: That's going to be a deadline, original documentation deadline. This definitely reflects how our deadlines work.

Based on our experience, we believe that our visualizations toolkit can indeed lead to valuable insights about the individual contribution and role played by each member of the team, which could help instructors when making decisions about individual performance/marks. Although, the reports of the team members in the group interviews did not directly corroborate the visualization insights, there are many convergent clues in the Jazz data (work-item creation, amount of time worked) as well as in the students' questionnaires, to lend credibility to the "leadership" implications of the visualizations.

Our visualization toolkit was also very useful in enabling inferences on how the team communicated, through formal and informal channels, and how this communication patterns correlated with the distribution of tasks among the team. We found that team members collaborating on specific tasks tended to use formal channels, i.e., work-item and commit comments, and face-to-face communications, while members across specific tasks tended to communicate informally through email to enable a general baseline awareness of their activity across the team. Understanding this communication convention (which may or may not be generally adopted) can be useful for instructors and teaching assistants, for instance, when following how teams are working during the term, in order to guide teams where certain members are not participating as expected.

4.2 A Grounded Theory Study on Software Engineering Students

In parallel to our empirical study, we conducted a grounded theory study to understand the human and social dynamics of the team we were studying. This is important for many reasons. First, better tools can be built that can address things that otherwise would be unknown. For instance, rather than mailing lists, students nowadays use other means for communicating (e.g.: *instant messengers*), which future tools should be able to retrieve data from. Second, it gives us the opportunity to understand why certain things, influenced by human factors, happened. Lastly, it gives us insights on how to improve software engineering courses by understanding how students behave in practice.

In this section we describe in details each step of our grounded theory study, which follows the scheme described in Figure 2.1, we present the theory outline and a set of five guidelines that may help instructors improving the learning experience for students of software engineering courses. We also discuss some points of concern and related studies to our grounded theory.

4.2.1 Data Collection

Our grounded theory study was conducted in parallel with the empirical study mentioned in section 4.1. As such, our data collection consisted of one interview with each of six students from one team of a senior undergraduate software engineering course. We treated each interview with a student as an iteration of data collection. Although the interview format was consistent and structured, the interview sessions often turned into conversations where students would give information beyond the scope of what was originally asked. Nevertheless, every individual interview was recorded on audio and notes were taken by two researchers. Each interview was about 30 minutes long and when transcribed generated about 10 pages of text, which were parsed (sentence by sentence) for analysis.

Furthermore, we also used the questionnaires each student submitted over the term to the instructor, containing their perceptions of the course and of how they and their teammates performed

in the team project. Likewise, each questionnaire was also treated as an iteration of data collection and was similarly parsed.

4.2.2 Data Analysis

The primary goal in the data analysis was to find trends and patterns in the qualitative data from the interviews and questionnaires, and then, following a grounded theory approach, produce a theory or narrative that describes the data. Hence, the term theory in this context means that it explains what has been found in the data and does not imply a universal truth that will hold true in all contexts.

Our approach, following qualitative methods in empirical studies, started by documenting the interviewer's observations during and immediately after each interview section, and writing notes (process described as *memoing* in grounded theory) of these observations. These notes, along with the interview transcripts and the answers of the questionnaires, were stored in two ways: (a) on an HTML file, upon which we used a browser plug-in[1] to annotate codes on relevant sentences, and (b) on database tables, where we could later easily submit queries on specific codes to gain a better understanding of the emerging theory. Each step of the process is further described below.

Open Coding

The first step in our analysis was *open coding* [12]. In this step, key points of the transcribed data are assigned a code (i.e., a small phrase or expression that summarizes the key point in one or a few words). Throughout our analysis, we had no predefined guidelines on how the data would be organized and coded; instead, we let our research questions guide the coding process, which is why we coded our entire data using a set of codes relevant to our research questions.

We present some of these emerging codes, as well as properties that categorize all textual data that have that code, below:

- Management: the student is reasoning about process management related activities such as task division and deadlines.
- Face-to-face communication: the student is reasoning about face-to-face types of communication such as group meetings or a conversation with a team mate that is attending a course in common.
- Student-Client Relationship: the student is reasoning about the relationship with the customer, such as a meeting with the customer to talk about requirements.
- Effort assessment: the student is reasoning about the effort put into some part of the project by one of the team members, as when asked to evaluate the work of their peers on the questionnaires.

[1] AnnotateIt - http://www.annotateit.org

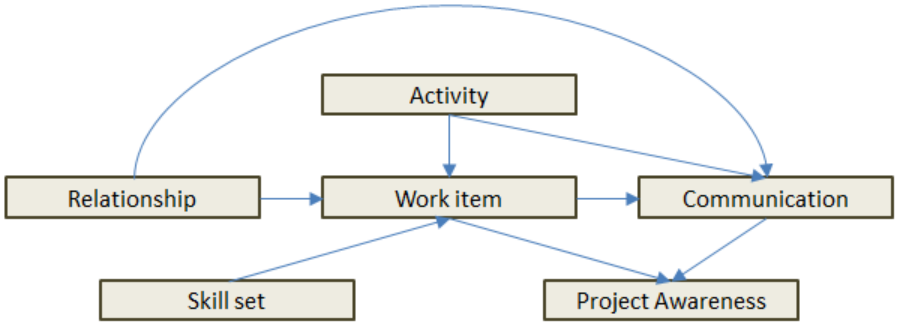

Figure 4.6: Grounded Theory categories and their relationships.

It is important to note that the creation of codes and properties is a creative process and not enforced by the researcher to force a theory but rather iteratively developed from analysis of the data. In other words, the researcher did not try to put the data in pre-defined buckets but rather create recurrent *categories* from the data. The codes arising were continually compared against previous codes, from the same interview and from other interviews and questionnaires, as specified by the grounded theory's *constant comparison* method [14].

After the categories were defined, the next step was to identify the *core category*. The core category is a category that reoccurs often, is related meaningfully to all other categories and is responsible for most variations in relation to the data. This is a crucial step as it serves as the base for the narrative of the grounded theory. By identifying the *Activity* category as the core category, we created an outline of the theory through selective coding, and describe how the different categories relate to the core category.

Selective Coding

After we identified the core category, we went through the data again doing what ground theory calls *Selective Coding*, which consists of tagging the data for categories. The goal of this step is to find relations between different categories and how different categories relate to the core category.

Figure 4.6 shows the relationships among the categories that emerged from selective coding. Although the *Work Item* category accounts for more relationships with other categories, the *Activity* category relates more meaningfully and strongly with the other categories, having great influence on how the other categories work and reoccurring very often in the data analyzed. The next section presents supporting data and examples for the relationships among the categories. The edges in the figure mean that the categories are related, in a way that the category indicated by the beginning of the edge influences the category pointed to. The relationships are described in Section 4.2.3.

Categories

After the open coding and also following grounded theory procedures, we repeated the constant comparison method on the codes to generate a higher level of abstraction called *Category*. The cate-

Table 4.1: Categories that emerged from the data analysis, their respective description and codes.

Category	Description	Codes
Activity	Common activities throughout a software engineering course	Management, Code, Documentation, UI, Design, Test, Learning, Process, Deployment
Communication	Communication channels among team members	Face-to-face, Email, Tickets
Project Awareness	Opinions or facts about the project and its members	Who, When, How, How much
Relationship	Interaction among students	Student-Student, Student-Manager, Student-Client, Student-Team
Skill set	Experience and competence on a particular subject or technology	Skills, Knowledge
Work Item	Process components	Task, Defect, Milestone, Issue, Business Need, Risk

gory level encompasses codes that are similar conceptually. As a result, several categories emerged and represented different aspects of the process followed by the student throughout the course. For instance the *Activity* category contains codes related to code, management, testing, documentation, and other activities common throughout a software engineering course project. Another example is the *Communication* category, which relates to means of communication used by the students, such as email, work item tickets or face-to-face communication (e.g., meetings). The relevant categories, their description and the codes represented by them can be found in table 4.1.

4.2.3 Supporting Data and Examples

Activity and Work Item Activities define which work items will be created. In the team's data, we observed that activities (i.e., code, documentation, UI) define which work items will be created. Several pieces of work were divided into many work items, related to the same functionality, but associated with different activities. One example of this is the search functionality, which contained work items such as "Search UI", "Search Logic" and "Search Interaction Details". That practice is revealed by students when asked about how they go about creating work items.

Student2:*We've kind of just been creating whatever came up.*
Student4:*Right now, we haven't made we've, kind of, just been making work items as we go.*

Activity and Communication Activities define how people communicate. In our data we observed that communication is related to the type of activity being performed by the team members

and the team in general. For example, when students have to perform activities like high-level design and documentation, they communicate less frequently and online (e.g., email), whereas when working with code (i.e., busiest times), they communicate more often and in person (e.g., meetings).

Student1:*Well, we're supposed to have weekly durations but without code how could you have weekly durations? Every couple of days we would talk about something, crunch time of course meant much more communication.*

Skill set, Relationship and Work Item Skill set and relationships define the tasks of team members. The background and previous experience of students with certain technologies or techniques define how the team is divided. People with the same skill set will usually collaborate more closely because they usually have shared work items (tasks) and challenges. The team studied was divided into three smaller sub-teams, each of which tackled a different platform required in the project (i.e., web, Android and iOS).

Student2:*I have worked with HTML before, but communicating with a server through a web page is entirely new...*
Student4:*Given some people do have done Android developing in the past...*

Along with skill set, the other determinant factor when defining tasks within the team is the relationship of students. In a senior software engineering course, students have often taken other courses in the past. These relationships, when existent, play an important role of how team members collaborate and communicate (examples follow below).

Relationship and Communication Different relationships imply different forms of communication. Students that know each other from previous courses usually communicate in a more direct manner, such as through phone calls, SMS messages and instant messengers, while students that do not have this previous relationship usually use tickets, email and other less direct forms of communication.

Student2:*So we were friends before the class and we communicate a lot, mostly just through phone, text message. Everyone else, it's we use the e-mail discussion board, pretty much.*
Student4:*Well the main person I collaborate with is Student3 cause pair programming. And we mainly just communicate by talk to each other. We're in all the three same classes together. The other guys we've used NHP emailing account to communicate and we set meetings if there's anything important to discuss that we have to talk about.*

Project Awareness, Work Item and Communication Students' awareness is related to work items. Students often know about what other students are working on if they are working on the same or related work items, and know little or nothing about students working on unrelated work

items, mainly because people that work on the same or related work items communicate more with each other. Students of the team knew more about people they worked with in certain work items and little or nothing with people that they communicated less with, even though the team had only six people.

Student1:*Having our group split up into three, I cannot say exactly...*
Student2:*It is hard for me to make judgements amongst the rest of the group members as I mainly worked with Student6...*
Student4:*Can't really say seeing that I didn't monitor the process of the other teams...*

4.2.4 Discussion

In this section, we provide the theory we have developed through the grounded theory method described above. It is important to emphasize that this theory is grounded within the bounds of the data collected. We then discuss details of how this theory was obtained what it represents in practice.

4.2.5 Theory Outline

In this course, senior software engineering students are expected to choose a process to follow during the development of their course project. No specific process is enforced by the course instructor but generally students choose a waterfall process or an agile process such as XP or Scrum. Our grounded theory is that *although students explicitly choose one of the processes for their project, they do not follow the process strictly, but rather adapt the process (and all its practices) to the course activities*. This modified process influences how students communicate, contribute and collaborate as a team. We explain how these things are related below.

As Humphrey [20] stated, this is true for students and for practicing engineers and it seems to be true almost regardless of the engineers' experience and training. One of the most intractable problems in software, he emphasizes, is getting engineers to consistently use effective methods. In his report he lists several reasons of why this happens and during our study we relate more closely to two [20]:

- Once engineers have learned how to develop small programs that work, they have also established some basic personal practices. The more engineers use and reinforce these initial methods, the more ingrained they become and the harder they are to change.
- Since no one generally observes the methods software engineers use, no one will know how they work. Thus engineers do not have to change their working methods if they do not want to.

Students adapt to their natural process as soon as they find themselves under pressure to get things done and because the process is not valued as much as the process outcome, a working piece

of software. In our grounded theory study, however, we found that not only students care more about a working piece of software at the end, but they also care (in a smaller degree) in performing the activity currently required by the instructor, in detriment of whatever process they decided to follow in the first place. To this end, in Section 4.2.8, we offer some guidelines that may help instructors making students follow a process. These guidelines, however, have still to be evaluated to verify whether or not they improve the learning experience of students.

4.2.6 Detailed Observations

Team member's contribution is affected by the type of activity

Some students only dedicate fully to coding, leaving activities such as documentation and design to other students who work on all types of activities. Besides, we found that students within the team are divided into two types. The first type includes the students evaluated by their peers as tops, which are students who dedicate themselves to coding, but are also doing the 'boring' work of documenting and designing everything on behalf of the group. The second type includes students who are rated as 'so-so', and includes those students who, although contribute equally to the coding part, do not feel excited about (or just do not contribute to) the documentation and designing activities.

Student2:*I would rate Student3's performance as so-so. Very hard to get in touch with, I'm sure he did a lot of work but it is frustrating not being able to communicate with someone. He barely contributed to the documentation, but regularly attended meetings.*

Student2:*I would rate Student4's performance as tops. Contributed heavily to documentation and regularly attended meetings.*

Student4:*Individually, some members did do more work, not just in terms of coding...*

Team's collaboration is affected by the type of activity

We found that since team members tend to collaborate together only with other members that work on related activities, and even though they know what other people are doing, they find hard to assess how much the peers who they are not working with have actually contributed to the project. That shows that communication among team members (rather than the team as a whole as discussed previously) is also related to the activities of the project, through a set of related work items.

Student4:*I always felt like I had no idea what Student1 was doing most of the time and if he was actually contributing.*

The actual process is affected by the type of activity

From the grounded data we observe that the process used by the team is much related to the current focus of the course. In other words, the team as a whole will collaborate, communicate and

contribute heterogeneously throughout the term according to the current activity of the course (documentation, implementation, tests).

4.2.7 Threats to Validity and Grounded Theory Evaluation

Grounded Theory is a specific method. We do not claim this theory to be universally applicable but rather accurately represent the context investigated, which is dictated by our research agenda. When analyzing the contribution of qualitative research, researchers often use terms such as quality and credibility, rather than the traditional validity, which is more appropriate for quantitative research. The quality of a qualitative research refers to its usefulness and thoughtfulness, while the credibility refers to its reliability.

In this sense, we follow Coleman and O'Connor's interpretation of Corbin and Strauss' list of criteria to evaluate the quality and credibility of grounded theory research [3, 13]:

- Fit: The theory must fit with the substantive research area and correspond to the data.
- Understanding: the theory makes sense to practitioners in the research area.
- Generality: The theory must be sufficiently abstract to be a general guide without losing its relevance
- Control: The theory acts as a general guide and enables the person to fully understand the situation.

In regarding to *fit*, in this research, we were careful in ensuring the codes and categories were being generated from the data. Furthermore, we also made sure the generated categories fit the general area of software engineering. To satisfy the *understanding* criteria, we tried to present the theory in a clear and understandable way, and, at the same time, care about presenting the findings in a sufficient details. We concluded our study in what is called the "substantive" level (context-specific) rather than in a general level, due to time constraints. In this sense, the *generality* requirement is left to be confirmed by future studies. In this work we developed a theory that applies to one student software team but is left to be verified for other contexts (e.g.: student teams of other less senior software engineering courses). In regarding to *control*, our theory was able to fully describe the situation studied, enabling the researcher not fully understand the situation.

Our grounded theory approach to understanding how students collaborate, contribute and communicate in the context of a software process, is a unique approach and contribute to the field in two ways; First, it contributes with a qualitative study to the field, from a grounded theory perspective. Second, it raises a question of pedagogical value, which is how to make sure students follow a specific process in a senior software engineering course, which we discuss below.

Furthermore, although data derived from interviews is known to be prone to bias, in grounded theory bias is not a threat, but rather a requisite to refining the questions for subsequent interviews until saturation is reached, and developing a theory.

4.2.8 Pedagogical Guidelines

We found that grounded theory is a useful method for understanding the social aspects of software engineering. We were able to formulate a theory based on qualitative data of students' perceptions collected through interviews and questionnaires. As our theory suggests students adapt their process to the activity being carried out at one specific moment during the course. For instance, usually in beginning of the term, they need to produce a set of documents (e.g.: project plan, requirements document, etc) and find no need to meet in person, collaborating usually remotely to perform that activity. This leads to some of the students contributing more than others and usually everything being done at the last minute. In this section, based both on our study and previous experiences, we offer a few guidelines that instructors may find useful to make students make the most out of software engineering courses. These guidelines may help students contribute equally to the team and the team as a whole follow a software engineering process.

1. Emphasize the importance of the process. Lately, senior software engineering course have been adopting real clients who need a software solution to a real problem they have. However, how requirements are gathered, meetings are held and, in general, the conformity of the team to a software process is not evaluated. Having a working solution to present to the client is important, but having students learning to follow a process (i.e., effective planning, quality management, etc) should be the ultimate goal, even if adaptations have to occur due to the nature of students having different schedules and other unavoidable things. Today, they must learn these practices on their own. This perpetuates the common student ethic of ignoring planning, design, and quality in a mad rush to start coding [20].

2. Do not undervalue activities like documentation and testing. Code is not the only important output of software projects and should not be treated so. More often than not the working solution at the end of the course is worth much more grade points than things like documentation. This usually leads to students only caring about code and treating the other artifacts as low priority. How should students that only contribute to one activity of the project, even if considerably, be evaluated?

3. Ask students to specifically define their roles upfront. This can be useful for minimizing having students doing things for others, as it is often the case. Even if students decide that everybody will do every activity, having that specified helps answering the question raised above. We argue that even though students might have contributed more to a working solution, if they contributed nothing to documentation, they should be penalized. Software engineering is not programming.

4. Make peer evaluations mandatory. In a real setting, software engineers are often required to peer review their co-workers, either on work products (e.g.: code reviews) or on performance

(e.g.: 360-degree reviews). In student courses, however, such evaluations are usually optional, even though they are confidential. When students refrain from answering such questionnaires they are not only not giving their opinion about things but is avoiding the work of reflecting about the process and team work, important activities that should be encouraged in any team setting.

5. Make the use of collaborative software development tools mandatory. In these tools students can record whatever they have done throughout the course. Furthermore, as shown in this work, the artifacts stored by such tools can generate valuable insights for the team members and instructors, which can ultimately help improving the learning and teaching experiences.

4.3 Summary

This chapter discusses an empirical study in which we used the toolkit presented in Chapter 3 to make inferences about the relative productivity and contribution of individual members in a software team. The findings of our study provide evidence that, indeed, an instructor inspecting the work of a student software team through the lens of our toolkit could draw conclusions with respect to which students assumed team-leadership roles and how the team may be further organized in smaller tightly collaborating subgroups. These conclusions may not be directly confirmed by the students when they are explicitly requested to reflect on their process in the presence of their team members but they are aligned with the personal student reflections.

Furthermore, in order to improve the learning experience of software engineering students, we studied how a team of senior software engineering students behaves within their course project. We presented a grounded theory and identify that the course activities imposed by instructors throughout the course duration plays an important role on the actual process followed by the students. The result of this study was a set of pedagogical guidelines that may help instructors making students make the most out of software engineering courses.

Chapter 5

Conclusion

Software-development environments offer an increasing number of features, integrating tools that used to be separate before and developers increasingly rely on a single IDE to perform all their tasks of the software development cycle. In the process, these integrated environments record large amounts of data about the project artifacts and their evolution and the activities of the software team members. Recognizing the value of the collected data as a means of analyzing and better managing the software process, the software-engineering community is developing a variety of visualizations through which developers and managers may gain insights through this large data. In this work we have leveraged the data stored in these collaborative software development IDEs to extract higher-order information to help instructors to understand and assess the contribution of each individual student of a software engineering project.

We have built a framework, presented in Chapter 3, that provides a set of analysis services, available through a REST interface, that produce information about how different individuals contribute to a software project. Our framework can be further extended to provide additional analysis capabilities, based on the implementation of our Data Provision API. Our framework also provides a variety of visualizations through which users can interactively explore the data from the perspective of the people involved in the project, the products they develop and their process. These visualizations provide an overview of the status of the project, contributions of team members in terms of time and work, and communication channels used by the team, while still providing a way to drill down into the data and trace back to the actual artifacts.

In Chapter4, we extracted data of a senior undergraduate software engineering team that used the Jazz Collaborative Platform and we used our toolkit to answer questions of interest to instructors about the relative productivity and contribution of individual members. The findings of our study provide evidence that, indeed, an instructor inspecting the work of a student software team through the lens of our toolkit could draw conclusions with respect to which students assumed team leadership roles and how the team may be further organized in smaller tightly collaborating subgroups. These conclusions may not be directly confirmed by the students when they are explicitly requested to reflect on their process in the presence of their team members but they are aligned with

the personal student reflections.

Finally, also shown in Chapter4, we engaged in a grounded theory study, which is still relatively unfamiliar to software-engineering researchers, to understand the human and social aspects of the team of students we were working with. We analyzed the qualitative data from interviews and questionnaires following grounded-theory procedures and saw a theory emerge through the data. Although students explicitly choose one of the processes for their project, they do not follow the process strictly, but rather adapt the process (and all its practices) to the course activities going on at the moment. The practice of not following a process is a common phenomenon, not only in an academic setting but also in industry, as related research has shown. In our grounded theory research, however, we found that students drive their process primarily according to the deadlines and activities set by the instructor. To this end, we offered a set of five guidelines that may help instructors addressing this issue and improving the learning experience of students.

5.1 Contributions

This thesis makes the following contributions.

- We developed a toolkit for analyzing and visualizing data collected by collaborative IDEs. The modular architecture of our framework makes it easy for it to be integrated with other IDEs, assuming they support exporting of similar data and our two APIS, the Data Provision API and Analysis Service API, are implemented. Our toolkit helps instructors answer questions about the role and performance of students through the use of visualizations. One of the key features of our toolkit is the possibility for the user to drill down the data back to the actual artifacts for further investigation. This is particularly useful for instructors, that can trace back to the artifacts from a browser, for instance, which is much more easily accessible and intuitive.

- We also contribute to the discipline with a grounded theory study of how a team of software engineering students behave on a course project. Historically, there has been limited research on the human and social aspects of Software Engineering. However, if we are to understand how software engineers use tools, follow processes or collaborate within a project, such studies need to become commonplace. While we do not claim our results are ground breaking, we contribute by providing stepping stones to more such studies be conducted in a technical field.

5.2 Future Work

In the future we plan on improving the set of visualizations to provide a wider range of visualizations to instructors. For instance, by using code-quality metrics, we will be able to provide not only

visualizations about the amount of work a student has done but the quality of the work. Although additional visualizations will help even further instructors on assessing the contribution of a student, the final balance between quantity and quality will still be left for instructors to decide.

Second, we want to make the visualizations available for students while they develop their projects, in order to study how the analytics services we have developed may or may not increase team productivity through increasing team awareness. In this scenario, we envision new features such as the use of annotations. Team members could annotate visualizations with their perceptions, thus creating constructive discussions around particular points of interest. These annotations could benefit the students themselves, by increasing their awareness about the project, the instructors, by understanding even more team collaboration, and us researchers, who can understand how participants are using the visualizations to make inferences.

Third, we would like to expand our experiments to an industry setting, to see how our visualizations behave with larger teams and consequently, larger data sets. An industry setting also creates the opportunity to experiment with completely distributed teams, and see how our visualizations can raise team awareness in this context.

On the grounded-theory side, our field is just beginning to perform such studies. We need to keep learning from the social sciences and from similar studies on the software engineering field, if we are to build tools that will be widely adopted and processes that will be followed and to understand the complex social side of software engineering.

Bibliography

[1] Steve Adolph, Wendy Hall, and Philippe Kruchten. Using grounded theory to study the experience of software development. *Empirical Softw. Engg.*, 16(4):487–513, August 2011.

[2] Ken Bauer, Marios Fokaefs, Brendan Tansey, and Eleni Stroulia. Wikidev 2.0: discovering clusters of related team artifacts. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '09, pages 174–187, New York, NY, USA, 2009. ACM.

[3] Gerry Coleman and Rory O'Connor. Using grounded theory to understand software process improvement: A study of irish software product companies. *Inf. Softw. Technol.*, 49(6):654–667, June 2007.

[4] Carlton A. Crabtree, Carolyn B. Seaman, and Anthony F. Norcio. Exploring language in software process elicitation: A grounded theory approach. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 324–335, Washington, DC, USA, 2009. IEEE Computer Society.

[5] Barthélémy Dagenais, Harold Ossher, Rachel K. E. Bellamy, Martin P. Robillard, and Jacqueline P. de Vries. Moving into a new software project landscape. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 275–284, New York, NY, USA, 2010. ACM.

[6] Daniela Damian and Teresa Mallardo. The role of asynchronous discussions in increasing the effectiveness of remote synchronous requirements negotiations; presented at. In *International Conference on Software Engineering*, page 917920. Press, 2006.

[7] R.T. Fielding. *Software Architectural Styles for Network-based Applications*. PhD thesis, University of California, Irvine, CA, January 2000.

[8] M. Fokaefs, K. Bauer, and E. Stroulia. Wikidev 2.0: Web-based software team collaboration. In *Wikis for Software Engineering, 2009. WIKIS4SE '09. ICSE Workshop on*, pages 67 –77, may 2009.

[9] M. Fokaefs, B. Tansey, V. Ganev, K. Bauer, and E. Stroulia. Wikidev 2.0: Facilitating software development teams. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 276 –277, march 2010.

[10] Marios Fokaefs, Diego Serrano, Brendan Tansey, and Eleni Stroulia. 2d and 3d visualizations in wikidev2.0. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–5, Washington, DC, USA, 2010. IEEE Computer Society.

[11] Thomas Fritz and Gail C. Murphy. Using information fragments to answer the questions developers ask. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 175–184, New York, NY, USA, 2010. ACM.

[12] B. Glaser and A.: Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine, 1967.

[13] B. Glaser and A Strauss. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory.* Sage Publications, 2007.

[14] Barney Glaser. *Theoretical Sensitivity: Advances in the methodology of Grounded Theory.* Sociology Press, 1978.

[15] Barney Glaser. *Doing Grounded Theory - Issues and Discussions.* Sociology Press, 1998.

[16] Matthew Hale, Noah Jorgenson, and Rose Gamble. Predicting individual performance in student project teams. In *Proceedings of the 2011 24th IEEE-CS Conference on Software Engineering Education and Training*, CSEET '11, pages 11–20, Washington, DC, USA, 2011. IEEE Computer Society.

[17] M. Hasan, E. Stroulia, D. Barbosa, and M. Alalfi. Analyzing natural-language artifacts of the software process. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1 –5, sept. 2010.

[18] James D. Herbsleb, David L. Atkins, David G. Boyer, Mark Handel, and Thomas A. Finholt. Introducing instant messaging and chat in the workplace. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Changing our world, changing ourselves*, CHI '02, pages 171–178, New York, NY, USA, 2002. ACM.

[19] Rashina Hoda, James Noble, and Stuart Marshall. Organizing self-organizing teams. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 285–294, New York, NY, USA, 2010. ACM.

[20] Watts S. Humphrey. Why don't they practice what we preach? Technical report, Software Engineering Institute - Carnegie Mellon University, 2007.

[21] David Krackhardt and Kathleen M. Carley. A pcans model of structure in organizations. *Proceedings of the 1998 International Symposium on Command and Control Research and Technology*, pages 113–119, june 1998.

[22] Ora Lassila and Ralph R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. W3c recommendation, W3C, February 1999.

[23] Thomas D. LaToza and Brad A. Myers. Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, pages 8:1–8:6, New York, NY, USA, 2010. ACM.

[24] Karen L. Reid and Gregory V. Wilson. Drproject: a software project management portal to meet educational needs. In *Proceedings of the 38th SIGCSE technical symposium on Computer science education*, SIGCSE '07, pages 317–321, New York, NY, USA, 2007. ACM.

[25] Fatma Cemile Serce, Robert Brazile, Kathleen Swigger, George Dafoulas, Ferda Nur Alpaslan, and Victor Lopez. Interaction patterns among global software development learning teams. In *Proceedings of the 2009 International Symposium on Collaborative Technologies and Systems*, CTS '09, pages 123–130, Washington, DC, USA, 2009. IEEE Computer Society.

[26] Christoph Treude, Patrick Gorman, Lars Grammel, and Margaret-Anne D. Storey. Workitemexplorer: Visualizing software development tasks using an interactive exploration environment. In *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[27] Timo Wolf, Adrian Schroter, Daniela Damian, and Thanh Nguyen. Predicting build failures using social network analysis on developer communication. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.

Appendix A

Data Description

A.1 Source Control Data

revision - Revision that identifies this commit.

date - Date of the source code commit.

author - Author of the source code commit.

comment - Comment about the commit.

changeSet - Change set containing the list of files that were committed.

A.2 Work Item Data

projectArea - Code that identifies the project the work item belongs to.

identifier - Code that uniquely identifies a work item.

priority - Priority of the work item (e.g.: high, normal, etc).

severity - Severity of the work item (e.g.: critical, normal, etc).

state - State of the work item (e.g: open, closed, etc).

resolution - Date of resolution of a work item.

title - Title of the work item.

description - Description of the work item.

created - Date of creation of the work item.

creator - Team member that created the work item.

modified - Date of modification of the work item.

modifiedBy - Team member that modified the work item.

ownedBy - Team member that currently owns the work item.

type - Type of the work item (e.g.: task, defect, etc)

resolvedBy - Team member that resolved the work item.

resolved - Date of resolution of the work item.

comments - Team member comments on this work item.

changeSet - (optional) Change set (source control data) that is linked to this work item.

Appendix B

Interview Questions

Team member questions

1. What role do you play in the team? How did you define that role?
2. What is the role of the other team members in your opinion?
3. Is there a leader (in management/ in development) in the team? Who? Why?
4. Are your work items related to your role? Who created your work items? What do they cover (coding/design/docs/learning)?
5. Who do you collaborate more closely with? How do you communicate? How do you communicate with the other members?
6. How much do you contribute to the team in term of time and effort? Do you have too many work items? Too few? Just about right? How important are they?
7. Did you notice any co-changing/co-evolving classes in your project? Are they changed by the same member or multiple members?

Team questions

1. How is the team organized? How were the roles divided?
2. What is the process followed by the team? How do you keep track of it?
3. What is the biggest risk of the project? What measures are taken to overcome it?
4. How much overlap you have in terms of work items / source code? How do you communicate to manage it?

Process questions

1. What would help your team to collaborate/communicate better?
2. What views/charts would increase your team productivity/awareness?

Appendix C

Questionnaires

C.1 Questionnaire 1 - Middle of term

Interaction with the Client

1. How many meetings did you have?
2. How many did you attend?
3. All in all, do you have a positive feeling about your interaction?
4. How computer-savvy are your clients?
5. What is the biggest challenge in your interaction with your clients?

About the project

1. What is interesting to you about your project?
2. Do you think that your team has sufficient technical knowledge to take this on?
3. What do you need to learn to get the job done?
4. What is the most important technical challenge for your project?
5. What is the most important non-technical challenge for your project?

Work Breakdown Structure

1. How is your team organized?
2. How have you decided to split the work?
3. Do you have a dedicated QA person?
4. How much code do you think you will develop vs. how much will you reuse?

Team Communication

1. How have you decided to communicate within your team?

Asprirations

1. What do you (personally) hope to gain from the experience?

Comments on Your Team Members

For each member of the team (starting with yourself), discuss:

1. What (you think) his/her role is?
2. How has he/she been contributing until now?
3. How you would rate his/her performance? (a) tops; (b) so-so; (c) thumbs-down?

C.2 Questionnaire 2 - End of term

Interaction with the Client

1. How regular were your meetings?
2. How frequently did your team attend the meetings as a whole?
3. Were there team members who miss meetings regularly?
4. How many did you miss?
5. Do you feel that you and your clients were at the same page throughout the project?
6. How confident are you that your clients will be satisfied with your project?
7. What was the biggest challenge in your interaction with your clients?

About the project

1. Are you happy (have you regretted) that you picked your project?
2. Are you happy with your accomplishments?
3. What is the major discrepancy between what you planned to deliver and what you did deliver?
4. What was the most important technical challenge for your project?
5. What was the most important non-technical challenge for your project?

Work Breakdown Structure

1. How balanced were the various tasks among your team members?
2. Who was the most productive team member?
3. Who was the least productive team member?

Team Communication

1. What channels did you use to communicate within your team?
2. How frequently did you communicate? Think about a typical day, and tell me what ”communication activities“ it might have included.

Self Reflection

1. What did you (personally) gain from the experience?

Comments on Your Team Members

For each member of the team (starting with yourself), discuss:

1. What (you think) his/her role was?
2. What has he/she contributed to the project?
3. How you would rate his/her performance? (a) tops; (b) so-so; (c) thumbs-down?
4. What is one positive comment that you would like to make about this person's contribution to the project?
5. Should all team members get the same mark for the project?