University of Alberta

TOWARDS UNDERSTANDING COLLABORATIVE SOFTWARE DEVELOPMENT

by

Ying Liu ©

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

Edmonton, Alberta
Fall 2005

To my husband, parents, and sister.

Abstract

The success of team software-development project depends on many factors, such as the technical competence of the developers, the quality of the tools they use and the project-management decisions. To successfully complete a project developers need to have an overall understanding of their project status, to possess sufficient programming experience, to collaborate well with the other team members, and to be able to react promptly to any unforeseen events during the project.

Instructors of project-based software-development courses, and more generally project managers, who are responsible for overseeing collaborative project development are sometimes overwhelmed by the task of monitoring the progress of their teams. Sometimes, problems in a team may go unnoticed until it is too late to be fixed. This issue, i.e., "how to support managers to understand the progress of their teams and to provide timely feedback" is the underlying motivation of the work in this thesis.

CVSChecker, is a tool that analyzes the collaborative software-development process. It examines the history of operations that team members perform in their project repository and the evolution of the software artifacts stored in this repository to discover interesting patterns and events in the (a) collaboration style among the team members, (b) the development contributions of individual team members and (c) the evolution of the software project. It produces reports and visualizations that can help instructors to notice issues in a team's process that should be addressed in order for the team to succeed in their task. CVSChecker was evaluated, with positive results, in two different contexts: (a) academic team projects of student developers and (b) open-source projects.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

A team is a group of people who share a common objective and need to work together in order to achieve it. It is a primary means for developing products in complex situations. Good teamwork is an essential factor for effective team performance [74].

1.1 Motivation

The software-development project success depends on the technical competence of the development team, the quality of its tools and the project-management decisions it makes during the software life-cycle. New or volatile requirements, tight delivery schedules and team-member turnaround present the team with challenges. To effectively deal with such obstacles requires that the developers have an overall understanding of the current status of their project, possess sufficient programming experience, collaborate effectively within their teams, and are able to react promptly.

Although software-engineering research literature abounds with information on how to develop high-quality software on time and on budget, book learning alone is not enough to train competent software professionals. Developers, especially software-engineering students, need to practice and apply the knowledge obtained from books and to acquire "hands-on" experience with realistic software development projects. Like project managers, instructors who teach courses involving collaborative project development are often overwhelmed by the task of monitoring the progress of multiple teams and problems in the team's process may go unnoticed until it is too late to be fixed. They may get mired in the complexity of the product or the individual components.

Supporting instructors to effectively monitor their software-development teams so that they can provide timely feedback was the overall motivation behind the CVSChecker plugin. The goal of this tool is to implement a methodology for monitoring the collaboration process of software to aid the manager's understanding of how the team members work through the project life-cycle.

More specifically, CVSChecker is interested in examining and shedding some lights into several related research questions. With a large variety of group types – from academic student teams, traditional industrial groups to the emerging open source communities – are the natures of teamwork, role performance, and collaboration the same? Can some specific patterns be abstracted and be representatives of a specific group type or a role? Will these patterns be affected by different project developing processes? What strong relationship do these patterns have and how could it consequently affect the whole team performance and the final product quality?

1.2 Research Problem and Methodology

CVSChecker is a component of the JReflex project [75]. JReflex integrates a set of tools, including CVS, Eclipse platform, PostgreSQL database, and uses a browser-accessible Wiki-based user interface as a front end to all the analyses results.

CVSChecker examines data collected from CVS, including CVS history and log as well as metrics on the assets stored in CVS. It analyzes the collected data from multiple perspectives. First, it tries to identify interesting patterns in the roles and contributions of individual developers to the team project. Next, it proceeds to analyze the evolution of the individual project files, stored in the repository. Finally, it comparatively examines the development process of a set of teams. CVSChecker produces as output visualizations of its analyses, and reports summarizing the team behavior and patterns of interest to the instructor monitoring the software-development team.

The process of analyzing a project with CVSChecker involves several steps: data collection, feature extraction, visualization, querying, data analysis, and knowledge extraction. CVSChecker extracts a substantial amount of information by examining the historical data recorded by source-management systems, presents the trends in these data through visualizations and reports, and examines the project-development process from several perspectives, including team collaboration, individual-developer role and source-artifact evolution.

1.3 Thesis contributions

This thesis contributes the following to the state-of-the-art in this research area:

- a repository-analysis method that examines and analyzes, in addition to the evolution of the software artifacts managed in it, the development behavior of the individual developers and the team as a whole;

- a set of distinct development-process patterns characteristic of different roles in the team, relevant in multiple life-cycle processes; and
- a tool that automates the above analysis and pattern extraction.

We have applied CVSChecker plugin on five student teams in our exploratory case study to examine whether CVSChecker works well on teams in a university environment. At the same time, we wanted to understand how students working in teams interact and to find out if there is any correlation among the educational environment, roles, their grades and the nature of their collaboration.

Our second case study involves three teams from the open-source community. The goals of this case study were similar to the goals of our student case study.

In addition, by comparing the results of the two case studies, conducted in different environments, with teams consisting of developers with different levels of experience motivated by different objectives, we wanted to develop some intuition regarding the impact of these factors on the software-development process.

1.4 Outline of the thesis

This chapter presented the motivation, thesis statement, and an overview of our approach. The remaining chapters of the thesis are organized as follows. Chapter 2 covers related work. Chapter 3 introduces the architecture of JReflеX system and the methodology of CVSChecker plugin. Chapter 4 presents an exploratory case study with 5 student teams while Chapter 5 provides another case study on 3 open source projects. Finally, Chapter 6 concludes, highlighting the contribution of our research.

Chapter 2

Related work

This chapter is divided into three parts:

- Related work on the analysis of data from version control systems;
- Related work on team roles and collaboration; and
- Related work on the empirical studies in universities on software development processes.

2.1 Related work on the analysis of data from version control systems

During the lifetime of a software project, configuration management or version control Systems (such as Concurrent Version System – CVS [33], Rational ClearCase [37], and Microsoft Visual Source Safe [26] are essential tools to allow handling of different versions of files in a cooperating team.

The analysis of version control system data began in 1990's; Ball was one of the first researchers to analyze the data from version control systems, and his paper "If your version control system could talk" [2] was treated as the earliest publication in this topic. Because Concurrent Version System (CVS) is popularly used in many universities (such as those student projects in our case studies) and open source communities (such as www.sourceforge.net), in this thesis, we focus on CVS.

2.1.1 CVS

CVS can store a large amount of historical information about the whole development process and allow development teams to work together on the same set of source code files. The main functionalities are listed as followings:

- Keep track of file version.

- Merge changes made on the same files by different developers.
- Retrieve old versions.
- Undo not working modification.

CVS data constitutes a valuable source that describes interesting aspects of a project's evolutionary changes. But CVS also has some weaknesses as follows:

They only store the entire source of the last revision. Other revisions have to be recovered by means of deltas. The changes are only stored at the line level in CVS, using a file difference algorithm. However, new tools, such as Eclipse [30], are capable to display the entire code of any previous version based on CVS information.

CVS does not keep track of which files have been changed together in a single commit operation. Often this information is required for the analysis, e.g., for the determination of a logical coupling. Researchers propose different solutions. A typical solution is to consider several changes as a transaction if the same developer made them at the same time, with the same log message (rationale). Usually commit operations take several seconds or minutes - especially the ones involving many files. There are two different approaches to defining the "same time": using fixed time windows [12] [15] and using sliding time windows [79]. The Sliding time window can recognize transactions that take longer to complete than the duration of the first one.

CVS does not keep track of which revisions resulted from a merge. Michael Fischer et al. [10] proposed a heuristic to detect these revisions. In addition, they introduced an approach for populating a release history database that combines version data with bug tracking data, and other data, such as merge points, missing in version control systems. CVS does not provide enough mechanisms for tracking the evolution of large software systems. Therefore, researchers usually combine CVS data with other project-related information, such as bug reports, mailing lists and so on.

2.1.2 Related work on CVS data

Version control system data are freely available now, for example via SourceForge.net. This kind of data provides lots of information on the evolution of a software project. Some researchers provided alternative interfaces to CVS and did some work in purpose to improve source code navigation. Moreover, such CVS data enable many new analyses, such as program analysis, software evolution analysis, metrics and quantitative analyses, and visualization.

Providing alternative interfaces to CVS

Several researchers provided easy-to-use interfaces for CVS. The two most commonly interfaces to CVS are Bonsai [24] and Irx [16]. They operate by retrieving the revision information of each file, which is then stored in a relational database. Both of them allow connecting to a particular archive in CVS repository via a web-based interface and isolate the users from the complexities of the CVS commands. They allow the users to inspect the history of any given file in the project. However, neither of them attempt to enhance the software trails available in the repository.

ViewCVS [27] is a browser interface for CVS and Subversion [69] version control repositories. It generates HTML templates to present navigable directory, revision, and change log listings. It can display specific versions of files as well as diffs between those versions. Basically, ViewCVS provides the bulk of the report-like functionality one expects out of a version control tool, but in a more user-friendly way than the average textual command-line program output.

TortoiseCVS [35] is another tool with similar functions to ViewCVS. It lets you work with files under CVS directly from Windows Explorer. With TortoiseCVS, users can directly do the CVS commands by right clicking on files and folders within Explorer, such as: check out modules, update, commit and see differences. In addition, users can see the state of a file with overlays on top of the normal icons within Explorer, perform tagging, branching, merging and importing, and go directly to a browser web log (using ViewCVS or CVSWeb [32]) on a particular file.

Xia [72] is a plugin for Eclipse for the visualization of CVS repositories based on the Shrimp Visualization tool [67]. Xia recovers relations available in the logs of a CVS repository and allows the user to navigate them. It uses nodes to represent files, their revisions and developers, and arcs to represent the relationships between them. Xia has two limitations: (1) Xia does not extract the CVS trails; it operates at the revision level instead of at the MR (Modification Request) level. (2) It relies on the Eclipse's API to CVS, which makes it extremely slow in large projects.

Source code navigation

CVSMonitor [19] is a perl CGI application for monitoring activities in CVS repositories in a much more useful and productive way than the previous tools. It is somewhat similar to CVSWeb, but far more useful when one wants to keep an eye on current development, or provide a public view into the source codes. If users use CVSWeb/ViewCVS and want to let the public see the repository, CVSMonitor should be used instead.

Codestriker [21] is an online collaborative code-reviewing tool and open-source web application. Traditional document reviews are supported, as well as reviewing diffs generated by an SCM (Source Code Management) system. It integrates CVS, Subversion, Bugzilla, LXR, ClearCase, Perforce, and Visual SourceSafe. Codestriker aims to minimize paper work, to ensure that issues, comments and decisions are recorded in a database, and to provide a comfortable workspace for performing code inspections. An optional configurable metrics subsystem can record code inspection metrics as part of the process.

Hipikat, a tool developed at UBC [7], supports source code navigation. It aggregates many sources of information such as bugzilla, CVS repository, mailing lists, emails, etc. and provides a searchable query interface. The purpose of Hipikat is to "recommend software artifacts" rather than summarize and visualize them. Thus Hipikat is much like Google for a software project. One interesting feature is that it correlates software trails from different sources, inferring relationships among them.

Hipikat relates two files using the transaction approach discussed previously. It provides a "What's related" button to suggest which files are closely related to the file under consideration. However, Hipikat determines coarse-grained relationships between files only. Besides, "relate" in Hipikat is more than evolutionary coupling: two artifacts may also be related if they refer to the same bug report number, appear in the same email, or log message.

To guide programmers, a number of other tools have exploited textual similarity of log messages or program code (for example: Version Sensitive Editing).

CVSSearch [23] searches for code fragments using CVS comments. Specifically, it takes advantage of the fact that a CVS comment describes the lines of code involved in the commit and that this description will typically hold for many future versions. The CVS comment history aids understanding of what the code does - including its motivation and history. Therefore, CVSSearch offers a better search than just looking at the most recent version of the code can.

Version Sensitive Editor [1] is a tool that puts the change history into the editor where it can be instantly accessed and used to control editing and convey version information. The purpose is to make the change history easily available to benefit the coding process.

Program analysis

Evolutionary coupling includes coarse-grained coupling (between files or classes) and fine-grained coupling (between program entities such as functions, methods, or attributes)

To our knowledge, the first work that leverages the product history to detect coupling within a system and between modules is the paper by Gall, Hajek and Jazayeri [11]. The authors have used their CAESAR system to analyze the coupling within a large telecommunication switching system, and found that the history of 20 releases can indeed reveal couplings within a complex system.

Later, Gall et al. proposed a three-tier software evolution analysis method (QCR) involving three different types of analysis: Quantitative analysis uses version information for the assessment of growth and change behavior. Change sequence analysis identifies common change patterns across all system parts. Finally, Relation analysis compares classes based on CVS release history data and reveals the dependencies within the evolution of particular entities. In [12] the authors focused on the Relation Analysis. They use CVS logs to expose relationships between classes and files that might not be found by other methods, such as call graphs. In [9], Fischer and Gall analyze the modification requests (MRs) and described the different types of logical coupling among the files included in the MR.

Some researchers also analyze different program revisions to detect coupling and interference between modules, such as the MORA/RECS tool of Snelting [66]. NORA/RECS use concept analysis to detect fine-grained coupling between variant configurations.

In contrast, Zimmermann et al. [78] do not analyze release histories of the entire system, but revision histories of the individual product files. This increases the granularity, allows examining fine-grained coupling between individual entities like functions, methods, and attributes. They chose CVS archives as the base for the investigations and implemented a prototype called Reengineering of Software Evolution (ROSE) [78] to analyze the evolution of CVS archives. ROSE adopts the "Right Way" method used in CVS2cl to detect coupled changes and calculates the strength of the coupling. In [79] they elaborate four essential preprocessing tasks necessary for a fine-grained analysis CVS archives. In [80] they apply data mining technique and tried to guide programmers along related software changes.

Van Rysselberghe and Demeyer at University of Antwerp did some work on mining version control systems for Frequently Applied Changes (FACs) [61]. They combined two CVS commands, "cvs log" and "cvs diff", to extract the change information and use clone detection techniques (Kamiya's clone detection tool CCFinder) [42]) to locate identical and similar code fragments.

Analysis of software evolution

The following lists several typical tools that do some software evolution analysis based on CVS repository data and other related historical data.

SoftChange [15] is a tool that extracts and summarizes information from CVS and bug tracking databases. SoftChange retrieves raw data from mailing lists, CVS repositories, bugzilla, documentation files, and ChangeLog files, identifies the code changes by analyzing the deltas and grouping them into modification requests (MRs) and obtains the related measures. In addition, it generates problem reports (PRs). The tool was tested on a typical open source project - Ximian Evolution.

The authors presented a more detailed description of their methodology in [14], which tries to rebuild the test project (Ximian) using "software trails" [14] from several perspectives: software releases, development activities, MRs, contributors, revisions, file types, change logs, source code hot spots, and modules. SoftChange rebuilds MRs based on a sliding window algorithm [13] and classifies them as code MRs, bug MRs, and comment MRs.

Cvsplot [22], formally known as CVSStat, is a Perl script which analyses the history of a CVS-managed project. The script executes on a set of files, analyses their history, and automatically generates graphs that plot lines of code and number of files against time. The tool was created to satisfy management reporting requirements. It is revealing to be able to see the "growth" of a project in terms of pure line counts, and how they correspond to the project's history.

StatCVS [28] is an open source project that generates a static suite of web pages, with charts and tables, which contain metrics about the evolution history of a software project. Although StatCVS was popular with users, lack of scalability, flexibility and interoperability led to the creation of the Bloof system.

Draheim and Pekacki, exploit source code repositories to extract information about project evolution. They proposed a new process-centric perspective that extracts CVS log data into a database and visualizes the software evolution using metrics. The result is the Bloof system [20] based on StatCVS. Bloof system includes a GUI tool, the Bloof Browser –which enables the user to perform data access, analysis and visualization – and a Java API for analyzing CVS data. Data artifacts can be navigated, filtered and grouped. Additionally, Bloof provides a set of compound queries, and visualizes the results and enables the user to export them into a XML document and web server.

Alberto Sillitti et. al. [65] designed CodeMart (CM) – a tool for the acquisition and the analysis of data regarding the software development process from version control systems. The information is stored in a data warehouse system that has five dimension tables: author, branch, module, time, and type of modification. CM includes two subsystems: the data extractor and the data analyzer. Users can query based on four categories: file, module, author, and statistics.

Researchers in Osaka University described Empirical Project Monitor (EPM) in [57]. EPM automatically collects and measures data from three kinds of repositories related to the evolution of a project: CVS repositories, mailing list managers, and issue tracking systems (Bugzilla). EPM provides integrated measurement results graphically, and helps developer/managers keep projects under control in real time. The goal is to develop an environment composed of a variety of tools for supporting measurement based software process improvement.

CVSAnalY [60] is another tool that extracts statistical evolution information out of CVS (and most recently Subversion) repository logs and transforms it to XML or stores it in a SQL database. It has a web interface where the results can be retrieved and analyzed. Luis Lopez-Fernandez et al. proposed [46] social network analysis to CVS data, for characterizing open-source software projects, their evolution over time and their internal structure.

Metrics and quantitative analysis

The above works mainly get information about software evolution and code structure from CVS repositories, and provide this information to users using different ways. It is not enough to view and display CVS data and programs through a convenient interface. Other researchers try to get deeper and higher level knowledge about the source code, process design, and product, to provide better direction to users. Statistics, metrics, data mining, and machine-learning technique have been used for this purpose.

Koch and Schneider [43] studied the evolution of open-source software projects using publicly available data. They proposed a data model and a set of metrics for open source projects. A Perl-script retrieves the necessary data from CVS repository through a web interface. A lot of charts were generated to show the relationships among person, file, discussing list, number of checkin, Added_LOC, deleted_LOC, time, from the perspective of software engineering with quantitative data. Cluster analysis was used. The work of Shirabad and Lethbridge is aimed at supporting software maintenance. In [62] they describe the application of inductive methods to extract relations that indicate which files are relevant to each other in the context of program maintenance (Maintenance Relevance Relation). They also tried to extract models through mining the software maintenance histories using data mining and machine learning techniques. In [63], they use classification techniques to learn relations that can be used to predict whether a change in one source file may require a change in another source file.

Annie Ying developed an approach that uses association rule mining on CVS data [77]. She especially evaluated the usefulness of the results, considering a recommendation most valuable or "surprising" if it could not be determined by traditional program analysis. She found several such recommendations in the Mozilla and Eclipse projects. Her work is on file level, not finer-grained entities.

Michail used data mining technique on the source code of programming libraries to detect reuse patterns in the form of associations [49] or generalized association rules [50]. The latter take inheritance relations into account. Both works lack an evaluation of the quality of the patterns found.

Researchers at University of Toronto also did some work on analyze the CVS data [51]. They conducted case studies on a second year undergraduate computer science course and analyze a set of course assignments. The also described a system for parsing CVS data and storing the results into SQL database. The system can extract various statistical measures of the source code and version histories. Through these measures, they attempted to correlate the code measures and repository histories, but the result is negative.

2.2 Related work on team roles and collaboration

Software engineering aims to support the building of software on time and within budget. Many of them are large-scale systems. In such a situation, no one person can carry all the details in his head. Teamwork becomes a hallmark. A team is a group of people who share a common objective and need to work together in order to achieve it. It is a primary means for providing products in complex situations.

There has been some research on analyzing the nature of teams and the team-members' roles. Dickinson et. al [8] summarized "7 key components of teamwork". Belbin [4] identified "9 team roles" and developed a theory of which combinations of these roles would lead to successful teams, and proposed five principles for building a high-performance team. Pfeiffer et. al [76] designed quizzes, inventories, and personality tests to measure members' social and teamwork aptitude, and to help individuals identify the roles most appropriate for them.

Although there are some ways to help team members identify their appropriate roles, understanding the nature of the team's collaboration including potential problems in it is not obvious. In contrast to the well-studied individual modeling, group modeling is still very immature. To our knowledge, only Mike Winter [74] did some research work on it. He worked on developing a group model based on an academic student team, and claimed that teamwork and social skills are the most essential factors that influence group performance and behavior.

However, all this previous research does not take into account differences in team types and development processes, which is the focus of our research. More specifically, the questions we want answer through my thesis are: are the nature of teamwork, role-specific behavior and performance, and collaboration patterns the same, with a large variety of group types to consider – from academic student teams, to industrial groups to the emerging open source communities? Or can we extract patterns representative of a specific group type or a team-member role? How will these patterns be affected by different project-development processes? How do these patterns relate and affect the whole team performance and the final product quality? The above questions constitute the main research problems that I am addressing in my thesis. In a word, the focus is trying to relate the process and role with a team's collaboration, performance, even the final product quality.

2.3 Related work on the empirical studies in universities on software development processes

As new software development processes become more popular in industry, there is a growing demand to introduce these development practices in post-secondary education. Agile [47] development method values individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a plan. Compare with classic life-cycle development, iteration and flexibility are the two main keys to the agile approach. Extreme Programming (XP) [3] is a mature and quite typical agile method. XP was proposed by Kent Beck [3] and a detailed treatment can be found in book [3]. In our case studies, in order to capture and compare some similar and different patterns among different software development processes, instructors taught XP and RUP [44] in a third-year undergraduate course.

There are also some computer science educators who are sparked by anecdotal evidence from industry extolling the benefits of these practices and are expressing interest in integrate XP into formal educational courses, and also to measure their effect. Some of them have already introduced these methodologies in software engineering undergraduate courses.

According to our knowledge, their works usually can be classified according to the following purposes: (1) introduce new methodologies and processes to students and train them to experience the associated practices; (2) help instructors to teach students and improve the course quality using the new processes; (3) evaluate the real effect of all/some practices of a new methodology in academic settings.

2.3.1 Introducing New Methodologies and Processes to Students

Muller and Tichy present their experiences on XP with 12 (6 pairs) CS graduate students in [53]. The goal is to gather experience with XP in an unbiased fashion. This case study is also a fair evaluation of XP. Most students have teamwork experience without pair programming experience. Project process is composed of three-week training (three small exercises to familiarize the environment, Junit, XP practices, test practices and refactoring) and 8-week project development. Students change to different partners for each exercise and project. 5 Questionnaires were filled. The authors gave not positive observations for some practices and confirmed that coaching is very important.

Schneider and Johnston thought that this is not a straightforward task the corresponding practices may run counter to educational goals or may not be adjusted easily to a learning environment. They defined educational objectives for software engineering courses in [64], evaluated XP practices with regards to these objectives, and listed a few recommendations for the curricula. The authors thought that XP should not suitable for typical educational environments if the instructors do not carefully craft the curricula. It is much more useful to equip students with the capabilities to use and tail the available techniques according to their situations than just teach them to following the steps. Noll [55] provided some observations from initial experience on applying XP to student projects. They thought that XP is an excellent aid in learning, due to its highly iterative nature, allowing students to make mistakes and learn from them. Moreover, they thought that the "test first" practice is difficult to learn.

2.3.2 Helping Instructors to Improve the Course Quality

Because teaching software engineering is difficult since many of the practices are motivated by large projects and large organizations of which the students have little or no experience, some researchers in universities apply XP as a method to help them teach basic software engineering concepts in undergraduate courses.

Hedin, Bendix and Magnusson adopted XP in their PT (Programming in Teams) course to help instructors to teach software engineering concepts. Their goal was not to teach XP, but to use XP as a vehicle for teaching. The detailed experiment designs are listed in [18]. The experience results are positive and there are two important aspects in their setup: "team coaching" and "team-in-a-room". Moreover, they presented many lessons learned from running the PT course in the new format.

Because the development of competent to excellent software practitioners remains a challenge, instructional models were developed to prepare students to become effective practitioners by Williams and Upchurch in [73]. They explore XP practices and provide some guidance in a software engineering educational context, discuss four different strategies in their educational program to improve the number and quality of skilled developers, re-examine and evaluate the practices of XP in an educational context where students are equipped with software engineering skills.

Holcombe et. al introduce XP to undergraduate students for real business project clients. [38]. The detailed experiment design is listed in [68]. The goal is to emphasis two issues for students: how to communicate with a client and capture the real requirements, and how to deliver a real high quality and bug-free product.

Back and Milovanov think that a university setting could be the ideal place to perform practical experiments and test new ideas in software engineering. However, there are still some problems that hinder the research and improvement of these techniques. In [40], they discuss how XP features can be applied to help instructors to minimize and circumvent those problems and difficulties appeared in a university environment. XP was used as the base software process to practice new programming methodologies, such as the Stepwise Feature Introduction (SWFI) in the paper.

2.3.3 Evaluating the Effect of Practices and Methodologies in Academic Settings

The work also falls into two main categories: the adoption of coding practices (pair programming and test first) or the adoption of all XP practices. In coding practices, pair programming is the most popular one that has been evaluated to certain content in introductory/programming courses.

Experience and empirical studies of the programming practices have generally been very positive. Laurie Williams et. al have done much work in this field. They found [70] that pair programming increased many measures of code quality, although at a slight increase in programming time. A late study found that pair programming increased both grades and retention in a first programming course.

Bisant and Lyle [5] investigated the effect of a two-person inspection method on programmer productivity. They used a pretest-posttest design with a control group that constitutes 29 undergraduate students. The students paired in the experiment group and performed a design inspection, a code inspection, or both, for 20 minutes and tried to find errors. The students in the control group developed the programs on their own. Bisant and Lyle reported a significant improvement in the experiment group as a result of using the two-person inspection method. The time saving was greater than the time lost in the pair inspection steps. The result may have more to do with the benefits of inspections than with pairing.

Nosek [56] conducted an experiment to compare the pair programmers and individual programmers. Five pairs and five individuals solved a challenging problem. The evaluation of the posttest questionnaire showed that pairs enjoyed the problem-solving process more and that the pairs were more confident in their solutions. However, on average, a single individual took 41% more time than a pair, in another word, this means that two individuals, working independently, will be 30% more productive than a pair. Therefore, Nosek argue that the loss of productivity is made up by better quality. Astrachan et. al [45] apply XP practices in their curricula and courses at Duke University and the University of Northern Iowa. They introduce some of the ways in which students differ from those real industrial developers; therefore academic environment can not embrace the XP principles thoroughly without any changes. Based on these observations, they design their curricula and methods to help students practice certain XP aspects, use pair programming between lecturer and an entire class to teach programming, and also the "small releases" and "refactoring" practices to teach software design.

Johnson and Caristi trained 11 students as two teams to follow XP practices in developing their course projects. The results were listed in [41]. The XP practices were divided as required, encouraged, and not easily simulated. Both student responses and instructor's observation show that this XP-like process resulted in good team communication and a broader knowledge of the project as a whole.

No matter how different the purposes are, all the above researchers have a common feeling: XP can not be applied as K. Beck claimed without any adaptation in an academic environment. Moreover, none of these researchers paid attention to the comparison among different methodologies.

Chapter 3

Collaboration Analysis

In this chapter, we first discuss the overall architecture of the JReflex, in the context of which CVSChecker was developed, then we describe the internal architecture of the CVSChecker plugin, and finally, we elaborate on its process, step by step.

3.1 CVSChecker in the context of JReflex

The JReflex environment, diagrammatically depicted in Fig. 3.1, consists of the following main parts:

- The development environment (based on Eclipse [30]);
- The repository, in which a set of facts regarding software products is stored;
- The analysis component that processes the repository contents to infer high-level information about the progress of the development;
- A browser-accessible wiki-server, WikiDev [29], that delivers and visualizes the analysis results, and
- A project-assessment component, through which developers and instructors can explicitly provide their own information regarding the project.

3.1.1 The development environment

The development environment – shown at the middle right corner of the diagram of Fig. 3.1 – is based on Eclipse and is tightly integrated with the repository CVS. Its primary purpose is to record the software-development process unobtrusively, as it occurs within the Eclipse environment.

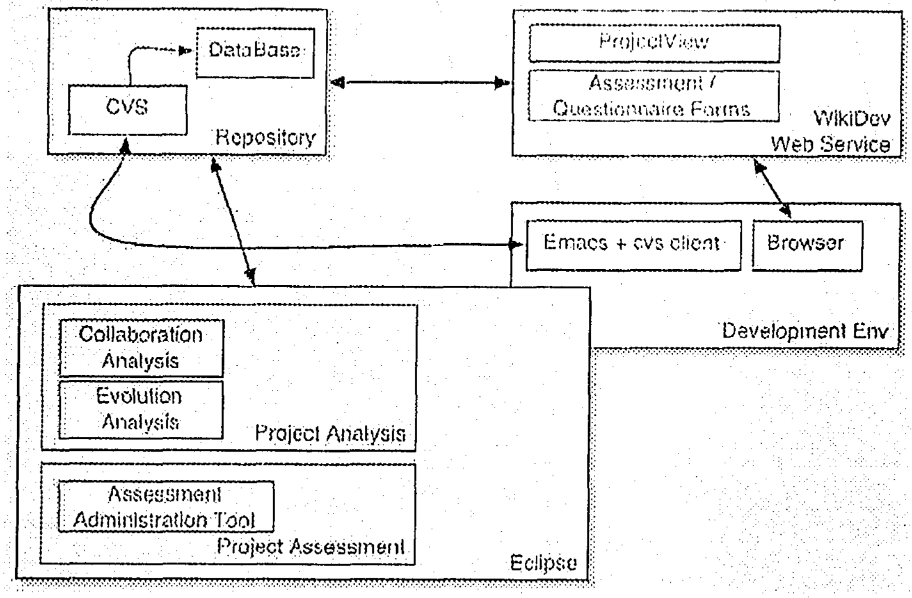


Figure 3.1: The JReflcX System Architecture

With respect to development tools, JReflcX assumes, at the very least, the existence of CVS, as the repository where all software assets are stored. Information about the contents and the operations' history of CVS populates its database of "facts" related to the Projects. In addition to CVS, JReflcX is tightly integrated with Eclipse as the development environment: the analysis components are implemented as Eclipse plugins and the visualizations of the data-analysis results are available as Eclipse views, in addition to being accessible through WikiDev.

The architecture of JReflcX relies on Eclipse as the main development tool, to provide a seamless integration of software construction and analysis activities. From a practical point of view, however, Eclipse is computationally intensive, and in cases where the hardware infrastructure is not sufficiently current - such as the case for most of the students' home computers - its adoption may not be immediate. The JReflcX architecture enables, even teams that do not adopt Eclipse as their development IDE to gain much of its benefits as long as they use a web browser and CVS: although the analysis components are developed as Eclipse plugins, their results are stored in the database and their visualizations are also served by WikiDev.

3.1.2 The repository

The repository was shown in the top left corner of Fig. 3.1. The repository consists of a CVS, where all development work products are stored, and a database, where work-product meta-data, qualitative and quantitative metrics of the software process, and its products are maintained. The database provides the core underlying structure for storing the JRefleX products and results, around the following basic concepts: CourseTerm, Project, Team, Developer, WorkProduct, History, Version, Activity and Assessment.

3.1.3 Collaboration and evolution analysis

The analysis component - shown at the bottom left corner of the diagram of Fig. 3.1 - is responsible for analyzing the collaboration process of the development team, as captured in the history of the repository CVS.

JRefleX has two analysis components. The collaboration-analysis component aims at inferring information regarding how the team members collaborate in the context of their project development by analyzing the CVS repository history of member actions and software changes. The evolution-analysis component, on the other hand, aims at discovering interesting patterns in the evolution of the project design and code, by analyzing the differences between subsequent versions of the project class hierarchies. Both analysis components are implemented as Eclipse plugins. Visualizations of their results are accessible through specialized Eclipse perspectives and through the WikiDev. In this thesis, we only elaborate the collaboration analysis component: CVSChecker plugin.

JRefleX relies to some extent on Eclipse as the main software-development platform. However, even teams that do not adopt Eclipse for development can use it, as long as they use CVS. The implication for such teams is that the only source of data regarding the collaboration process is the CVS history. This data can be obtained, stored and analyzed by the analysis component, and the team can access the results through the repository Wiki server.

3.1.4 The Wiki server

WikiDev, the JRefleX Wiki server, leverages open-source software, phpwiki, as a framework for maintaining and exchanging information about the projects in a free-form, flexible manner. WikiDev is a collection of plugins and modifications to the phpwiki, which extend the original functionality of the WikiWikiWeb concept as pioneered by Ward Cunningham (see http://www.c2.com for more information).

The WikiDev extensions are primarily concerned with group based security and CVS integration. Each team is associated with a specific Wiki. There is also a special Wiki for the instructor team, i.e., the course instructor and the TAs. Each Wiki is accessible only by members of the team associated with this Wiki. Once logged in, team members can view project information, change passwords, or simply collaborate in a WikiWikiWeb fashion by constructing new pages of their own, to maintain and exchange information about their work with their team members. Through the Project View plugin, team members have access to all their projects. Specific work products and their versions can be inspected for each of these projects through special wiki pages, automatically constructed by the WikiDev based on the contents of the CVS repository. This gives users the ability to edit and attach concepts or documentation to their work products, in a manner that enables change and refinement through the versioning capabilities of the Wiki.

3.1.5 The project assessment component

The primary objective of the JReflex tool is to unobtrusively collect and analyze data from the tools that students use in their software development, in order to infer information that can help the instructor and the developers themselves to effectively monitor the development process. Currently, the main source of such input data is CVS with its operation history and its contents. In the longer run, we intend to exploit the upcoming Eclipse instrumentation API to unobtrusively record the fine-grained tool actions of developers working upon their code and documentation.

However informative such information, implicitly inferred from tool-usage data, may be, it is also interesting to compare it with "objective" data, explicitly provided by the developers and the instructor team. The JReflex assessment component addresses exactly the need to enable the collection of such "objective" data.

In the past, students of our project-based software engineering courses were required to answer a set of questions at specific points during their project development. The questionnaire was implemented as a stand-alone web-based application with a specific list of questions. The answers were collected as HTML documents, which made automatic analysis of this data difficult, and limited the kinds of information that could be obtained. For this reason, the JReflex assessment component has been designed to be configurable with respect to the types and amounts of data requested as part of these questionnaires.

Currently, questionnaires are created in an administration tool implemented as a set of Eclipse views. Data, i.e., answers, are collected through a WikiDev plugin. Team members who can log in their team wiki see the questionnaires that require completion, and fill them out. When a filled questionnaire is submitted, the component validates the provided data against the expected question-answer types and stores the data in the repository database. Since the WikiDev is where teams will do most of their collaboration, this is currently the best environment in which to inquire about collaboration. Finally, in addition to enabling self assessment of team members, questionnaires can also be used by the instructor team to evaluate the project deliverables.

In this manner, data regarding the developers' own view of the project progress and instructor-provided "objective project evaluation" data can become part of the database, and can provide an external validation instrument for the inferences of the analysis components.

3.2 The CVSChecker Data Model

In this thesis, we focus on CVSChecker, the collaboration-analysis component. The motivation of CVSChecker plugin is to analyze the nature of individual developer's roles and team collaboration in the context of different software-development processes and constitutions. The final goal of this research is to design a process-mentoring tool that can help managers provide timely and relevant feedback to the teams by recognizing problematic patterns and events.

This section describes the CVSChecker data model, i.e., the schema of the PostgreSQL [34] database where work-product data and analysis information are maintained. CVSChecker plugin has two main data sources: file revision-related information and operation-related information. We collect all these data using two CVS commands: "cvs log" and "cvs history". The details of data collection can be found in next section.

The database provides the core underlying structure for storing the JReflexX products and results, around the following basic concepts:

- CourseTerm: a CourseTerm represents a particular group of Projects that are being developed for a class project during an academic term. In database, there is a corresponding table – "courseterm". It includes two main columns: "course" and "team". This table was designed originally for student teams and recorded the course name and team information.

- Project: a Project represents a particular module or portion of a module within a CVS area, and is associated with the Team developing it. JReflexX database has a same-named table "project" with four main columns: "teamid", "coursetermid", "modulename", and "cvsroot". "teamid" references table "team" and "courseteamid" references table "courseteam". "cvsroot" records the address of the CVS repository of this Project, and "modulename" lists all the subdirectories under the "cvsroot".

- Member: a Member is an individual who joins one or more teams for one or more Projects. Table "member" in database includes two main columns: "userid" and "unixname". "userid" references table "users" where records all the JRefleX users and "unixname" shows their unixnames.

- Team: a Team is a group of Members who are working together on one or more Projects. Projects, Teams, and Members lay the groundwork for a particular piece of a Project, referenced to as a WorkProduct. The column "name" in table "team" records the name of each Team. We get all above information from the project background collection.

- WorkProduct: a WorkProduct is a part of a Project, i.e., a file within the Project's CVS area that requires constructive effort by one or several specific Members. The actual information regarding what a Member has produced is stored as a set of Version of a WorkProduct. JRefleX database has a table named "workproduct". It includes 5 main columns: "projectid", "mimetype", "filename", "modulepath", and "isremoved". "projectid" references table "project", and "mimetype" tells us that what file type this Workproduct belongs to. "filename" and "modulepath" display its location, and "isremoved" lets us know that whether this Workproduct is still exist in CVS. The information was collected from CVS repository.

- Version: a Version parallels the notion of a CVS file revision and contains much the same metadata. Table "version" in database has 9 important columns: "workproductid", "memberid", "revision", "revdate", "linesadded", "linesremoved", "log", "totallines", and "content". Each row in this table is a file version. In time "revdate", developer "memberid" adds "linesadded" lines and removes "linesremoved" lines on "workproductid". "revision" is generated, and the total size of this new revision is "totallines". The rationale of this modification is explained by "log", the new code version is recorded as a large object, and its oid number is recorded in "content". All the data in this table are collected using "cvs log" command and captured from RCS files, see section 3.3.1.

- History: Essentially, The History contains records of all performed CVS operations of all types, during the project life cycle. These operations may have been performed to a specific WorkProduct or to a Project module. Each CVS operation committed by a Member will leave history trail on CVS according to the development conditions. If the operation is committed on a WorkProduct/Module, a History_FileLevel/ History_ModuleLevel record will be added. Table "history_filelevel" in JReflex database includes 5 main columns: "date", "opertype", "revision", "author", and "workproductid". It shows that on "date" "author" does an "opertype" operation on the version "revision" of "workproductid", a corresponding history is recorded in CVS repository. Similarly, table "history_modulelevel" records all the histories related to a module instead of a Workproduct. "history_modulelevel" has "projectid" and "modulepath" columns instead of "workproductid" and "revision". These data come from CVS command "cvs history". See section 3.3.1 for details.

- Activity: An Activity describes a particular type of work that Developers may do while working on Projects. In database, we have a table named "activity" and it has columns "name" and "description". In our first case study, we have the following activity names: planning, design, coding, testing, documentation, etc. "description" column records the detailed explanation for each of them.

- Assessment: Each Member will be asked to fill a set of Assessment forms for each specific Project. Table "assessment" includes four main columns: "userid", "project", "questionnaire", and "filledout". Each row means that Member "userid" filled "questionnaire" on time "filledout", and all these questions are related to "project".

3.3 The Collaboration-analysis Process

The process of analyzing a project with CVSChecker involves several works as following: data collection, feature extraction, data storage, visualization, querying, data analysis and knowledge extraction, and reporting. Fig. 3.2 depicts the detailed architectures.

In general, CVSChecker plugin has following major functions:

- Unobtrusively captures information along the developing process without interfering with the developers' activities;

- Automatically parses the information into the database;

- Provides a simple interface for users to query for multiple aspects of the whole developing process;

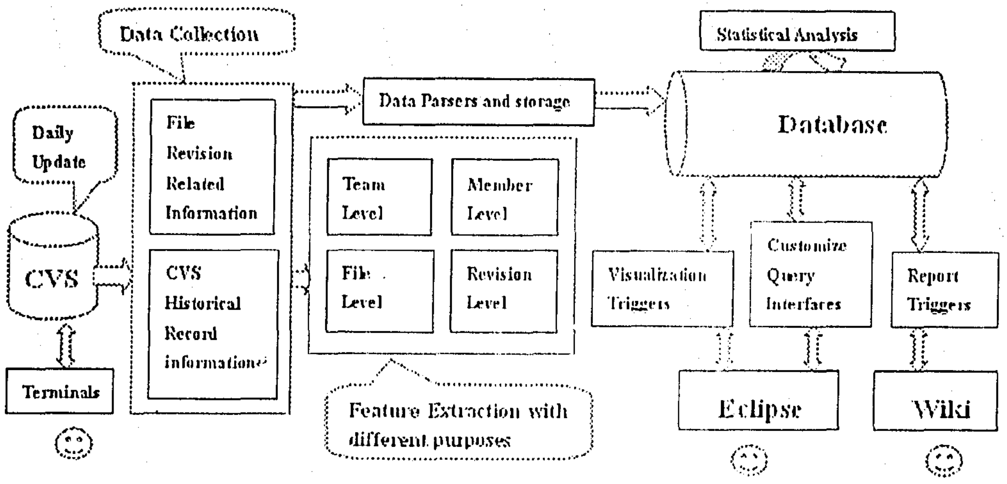


Figure 3.2: The architecture of CVSChecker plugin

- Displays those query results using vivid visualization ways;
- Reveals symptoms of bad design and unbalanced task divisions;
- Assists team leaders or instructors to have an entire understanding of the project;
- Gets heuristic knowledge based on each role, module, team, and so on;
- Summarizes typical performance patterns that are related to different roles or environments.

We have evaluated the effectiveness of CVSChecker with respect to these functionalities with a set of case studies, involving teams in educational environments and the open source community. These case studies will be elaborated in Chapter 4 and 5.

3.3.1 Data collection and fact extraction

As mentioned in related work, we mainly focus on CVS in this thesis because all the students in our case studies use CVS to support their project development and CVS is also adopted by the biggest open-source project community – *www.sourceforge.net*.

CVSChecker examines the development-process trails recorded in the CVS repository of a project to be analyzed. This is an information-rich data source. Not only does it contain a sequence of versions for each software module, but also it records information regarding the usage of each version by each individual developer. A detailed development history (including who performed what operation, when, from where, on which file, why) is maintained by CVS. Based on this information, a lot of valuable information can be inferred.

The history information can be traced by appropriate CVS commands, in many different levels of granularity. CVSChecker has a suite of parsers that extract this information from the source code repository and store it in a relational database that can be easily queried. There are three main data sources that collected by CVSChecker plugin: File revision-related information; CVS historical record information; and Administrative and assessment information.

File Revision-Related Information

A data-extraction command used by CVSChecker is "cvs log". The command can retrieve and display a great amount of meta-information about versions for a file under the Revision Control System (RCS)[71]. The file is ended with extension ",v".

RCS saves all old revisions in a space-efficient way, automatically retrieves multiple revisions according to ranges of revision numbers, symbolic names, dates, authors, and states. A complete history of changes was maintained by RCS. The logging makes it easy to find out what happened to a module, without having to compare source listings or having to track down teammates. Besides, RCS has other functions, such as: to resolve access conflicts by giving alerts, to maintain a tree of revisions, to merge revisions and resolve conflicts, to control releases and configurations, and so on. A parser was created by CVSChecker to get the file revision-related information from all RCS files.

Each RCS file basically includes two parts. See Fig. 3.3.

The first part lists information of each revision of a file. The larger the revision number is, the higher the position is. All the lines above "=====" line in Fig. 3.3 is an example. We collected the following main information:

- File name: the name of the selected file. This information goes to table "workproduct" in database. In Fig. 3.3, it was "./RCS/myscript.sh,v";

- Locks status: The login name of the user who locked the revision (empty if not locked). RCS assumes that users lock a file when they want to use it, and won't allow anyone else to modify that file. Because the lock status of most files are "strict", we do not collect this attribute in our database.

- Total revision number: how many revisions does this file have? In Fig. 3.3, the number was 2; All below attributes together with this one are saved in table "version" in database.

- Selected revisions: the revision number assigned to this revision;
- Description of each revision;
  - Date: The date and time (GMT) the revision is checked in. In our example, the Date of revision 1.2 was "2002/11/05 04:01:13";
  - Author: login name of the user who checked in the revision. He is also the developer who created this revision. In Fig. 3.3, it was "james";
  - Added LOC: how many lines were added for this new revision. The number was 2 in our example;
  - Removed LOC: how many lines were removed from this revision; and
  - Log: a full rationale to generate the selected file revision. In Fig. 3.3, the log of revision 1.2 was "Changed World to $USER to give a more personal feeling".

The second half (lines below "====") uses a space-efficient way to record the real source code of each revision and to enable users to know the real changed code lines: the whole code of the final revision was recorded, for other previous revision; RCS file only lists the basic modification information as followings. All the attributes listed here have corresponding columns in table "version" in database. Based on such concise information, we can regress to the source code of each previous revision as we want.

- Date: The date and time (GMT) the revision was checked in.
- Author: login name of the user who checked in the revision. He is also the developer who created this revision;
- State: The state assigned to the revision;
- LOC added between two consecutive revisions (where and what are these new lines);
- LOC removed between two consecutive revisions (where and how many lines);
- Reason log of this new revision.

Though different files have different contents and sizes, RCS saves them using a fixed format: the latest revision was recorded on the topmost. Each revision section was begun with its revision number. Log of this revision was appended next, with two "@"s as terminuses. A line of "text" begins the code area for each revision, another pair of "@" was adopted. For all non-final revisions, the text area saves the modification information using

```
RCS file: ./RCS/myscript.sh,v
Working file: ./myscript.sh
head: 1.2
branch:
locks: strict
access list:
symbolic names:
keyword substitution: kv
total revisions: 2;     selected revisions: 2
description:
----------------------------
revision 1.2
date: 2002/11/05 04:01:13;  author: james;  state: Exp;  lines: +2 -2
Changed World to {USER} to give a more personal feeling
----------------------------
revision 1.1
date: 2002/11/04 11:57:51;  author: james;  state: Exp;
Initial revision
=============================================================================

1.2
log
@Changed World to ${USER} to give a more personal feeling
@
text
@#include
void main(void)
{
    printf("Hello, world!\n")
}
@

1.1
log
@Initial revision
@
text
d5 1
a5 1
printf("Hello, world!\n");
@
```

Figure 3.3: An example of RCS file

a simply way: "d3 8" means there are 8 lines were deleted from line 3. For all added line messages, the new lines will be shown just below the message line, see Fig. 4. There are two empty lines between two consecutive revision sections. Based on such a strict format rules, we create a parser for CVSChecker plugin to get information we are interested.

All the file revision-related information was parsed into table "version" in our JReflex database.

CVS Operations	Possible Record Types	Descriptions/Conditions
cvs release	F (release)	A directory in CVS is released. Indicates that a module is no longer in use. It has the same effect as direct working-directory deletion, but avoids the risk of losing changes, which users may have forgotten.
cvs checkout	O (checkout)	Checkout sources from the CVS repository to a working directory for editing
cvs export	E (Export)	Export sources from CVS, similar to checkout
cvs rtag	T(rtag)	Added a symbolic tag to the RCS file
cvs commit(Checks the files into cvs)	A (Add)	A file was added to CVS for the first time. The first revision for this file is created.
	M (Modify)	A file is modified and a new revision appears
	R(Remove)	A file was removed from the CVS repository
cvs update(Bring work tree in sync with repository)	C(Collision)	A collision was detected as a result of more than one developer modifying the same code area in the same file revision; A manual merge is required
	G(Successful Merge)	A merge was necessary and it succeeded (this happens when multiple developers change different code areas of the same file revision without causing conflicts)
	P (Patch)	A working file was patched to match the repository
	U(Copy)	A working file was copied from the repository
	W(Delete)	A working copy of a file was deleted during update, because it had already been removed from the repository

Table 3.1: CVS historical record types with the corresponding CVS operations

Operation-related Information

In the related-work chapter, we list the main similar researches that aim at extracting interesting information from the data captured in CVS repositories. No matter on what analysis levels of granularity, from coarse-grained entities (system, module, class, and file) to fine-grained entities (function, method, attribute), these researches mostly start by grouping the CVS change deltas into transactions (or Modification Requests) assumed to represent all related modifications in response to changes in functionality or bug fixes.

An important distinction of our work with CVSChecker is that we examine not only file revision information from the command "cvs log", but also the information from the command "cvs history". The latter command records different CVS operation trails in the repository.

We argue that this information is important because the same operation executed under different conditions will generate different trails. For example, there are five possible consequences for the command "cvs update": C/G/U/P/W. The detailed explanations are listed in Table 3.1.

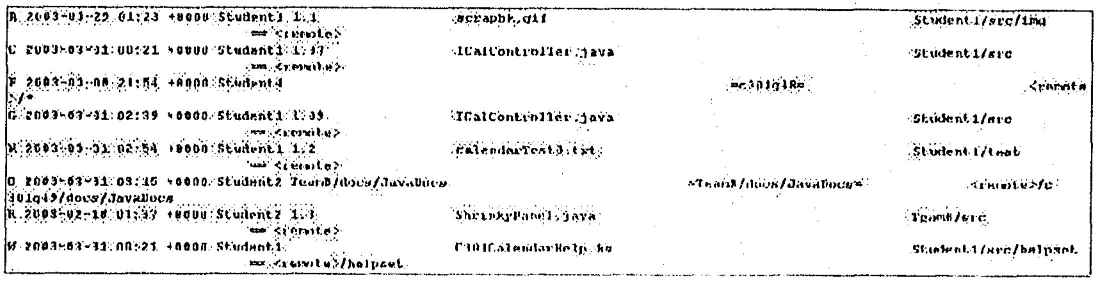
Through collecting and analyzing these CVS operations, a lot of hidden information of project development, especially related to students' collaboration and software design, can be revealed. Based on them, CVSChecker can help users to better understand the development process.

Table 3.1 lists all CVS historical record types with their corresponding CVS operations and the detailed explanations. For each record type, we list the happening condition.

According to the descriptions and conditions of these types, we classify them into the following four classes. CVSChecker plugin has several charts to display these classes from different aspects. Some simplified charts are designed to mainly show the operations in constructive types and red-flag types along the development process.

- Constructive types: such as type A (add), M (modify) and P (patch);
- Red-flag types: such as C (collision) and G (merge);
- Related types: types that relate to each other, such as R (remove) and W (Delete);
- Rare types: these types happen infrequently or do not have much valuable information for our analysis, such as E (export), F (release), T (tag), U (copy).

We use the command "cvs history -ae > History.txt" in the root directory of a project to



```
R 2003-03-29 01:23 +0000 Student1 1.1          scrapbk.gif                                  Student1/src/img
                           == <remote>
C 2003-03-31 00:21 +0000 Student1 1.37         ICalController.java                          Student1/src
                           == <remote>
F 2003-03-08 21:54 +0000 Student4                                      =c301g18=                                <remote
>/*
G 2003-03-31 02:39 +0000 Student1 1.39         ICalController.java                          Student1/src
                           == <remote>
M 2003-03-31 02:54 +0000 Student1 1.2          CalendarTest3.txt                            Student1/test
                           == <remote>
O 2003-03-31 03:16 +0000 Student2 TeamB/docs/JavaDocs              =TeamB/docs/JavaDocs=              <remote>/c
301g49/docs/JavaDocs
R 2003-02-10 01:37 +0000 Student2 1.3          ShrinkyPanel.java                            TeamB/src
                           == <remote>
W 2003-03-31 00:21 +0000 Student1              C301CalendarHelp.hs                          Student1/src/helpset
                           == <remote>/helpset
```

Figure 3.4: A segment of History.txt

collect all CVS operation history records, and the results is saved into a text file-"History.txt" in the same directory automatically. Fig. 3.4 lists an example section of this file.

Using this command, we can easily get the following main attributes:

- Operation record type: what operation record type happened? Table 3.1 lists all possible types;
- Timestamp: when the operation happened?
- Executor of this history operation: who did it?
- Revision number: on which file revision?
- File name: on which file?
- Directory of the file: where was this file?

The CVSChecker plugin parses all these CVS historical record information into tables "history_filelevel" and "history_modulelevel" in the JReflex database.

3.3.2 Derived-Information inferencing

Based on the collected data above , we can extract a rich set of derived metrics, which can be valuable indicators for the individual's performance and the team's collaboration. However, putting all these data together for any analysis will incur the waste of resource and flood the target rules with some unrelated data. Our method advocates different data extraction and filters for different analysis purposes. Different purposes lead to different focuses. If we focus on the performance of an individual, all the data related to him/her should be extracted. If file ownerships were focused, at least "Add", "Modify" and "Remove" typed records are useful. First of all, users should decide their analysis purpose at the beginning.

Although using more attributes may generate more potential results, too much attributes will induce dataset overstuffed and hinder the mining speed. Our method proposes a way to control the data scale without altering the analysis results as following: selecting several basic directly collectable attributes based on the research purposes, and then generating some new concentrated attributes by combining the basic ones.

We organize the parameters into four main categories:

- Parameters specific to a team;
- Parameters specific to an individual developer;
- Parameters specific to a file;
- Parameters specific to a file version.

In each level, the parameters can be divided into two classes: directly collected parameters and derived parameters. Directed collected parameters are those data that exists in the data sources we listed in above section and can be captured easily and directly. Derived parameters can not be directly collected and usually are computed by those directly collected parameters to measure underlying relationships between the team members' work habits, roles, main tasks, the project-design structure, project schedule, and so on. In the following section, we enumerate some examples of these parameters that could shed some light on the above relationships on each level.

The Team Level

In university environment, with the same project requirements, comparisons across different teams are very useful for instructors to monitor the performance of the whole class, and quickly notice unusual trends, lag behind the schedule, and events that might signify problems. Team leaders in industrial environments also can analyze different sub-teams in a large project with the same purposes using these parameters on this level. For each team, we record the following parameters:

Directly Collected Parameters:

- $Member\#$: the total number of members in this team;
- $File\#$: the total number of files with any extension types;
- $Java\_File\#$: the total number of Java files;

- $Revision\#$: the total number of revisions;
- $Java\_Revision\#$: the total number of revision of all Java files;
- $Day\#$: the total days of the whole development process this team participated;
- $BeginDate$: the first day when this team began their development;
- $EndDate$: the last date that recorded in CVS repository of this team;
- $WorkDay\#$: the total number of days that at least one member in this team has CVS operation; Of course, $WorkDay\# \leq Day\#$;
- $Java\_WorkDay\#$: the total number of days that at least one member in this team has CVS operation on Java files, $Java\_WorkDay\# \leq WorkDay\#$;
- $Phase\#$: the total project phase number of this team;
- $PhaseDate\,i$: the ending date of the i^{th} phase, $i \in [1, Phase\#]$;

The following two parameters are only related to those teams in the Open-Source community environments:

- $RegisteredDate$: the date when this team registered in www.sourceforge.net firstly;
- $DevelopmentStatus$: www.sourceforge.com uses seven levels to show the development status of a project: 1 is Planning, 2 is Pre-Alpha, 3 is Alpha, 4 is Beta, 5 is Production/Stable, 6 is Mature, and 7 is Inactive;

Derived Parameters:

- $Ave\_Oper\_Type_i\#$: the average CVS operation distributions by type; $i \in [1, 13]$, see 13 types in Table 1;
- $Ave\_Oper\_Date_i\#$: the average CVS operation distributions by date; $i \in [1, Date\#]$;
- $Ave\_WorkDay\#$: the average days that at least one member in this team has CVS operation.
- $Ave\_WorkDays = (\sum(WorkDay_i))/Member\#, i \in [1, Member\#]$;
- $Java\_Ave\_WorkDay\#$: the average days that at least one member in this team has CVS operation on Java file.

- $Ave\_WorkDays\_Java = (\sum(Java\_WorkDay\#_i))/Member\#, i \in [1, Member\#]$;
- Ave_MR_Size: the average involved file number in a MR,Ave_MR_Size=Revision$\#/MR\#$;
- $Ave\_FileRevision\#$: the average revision number each file includes in this team; $Ave\_FileRevision\# = Revision\#/File\#$;
- $Java\_Ave\_FileRevision\#$: the average revision number each Java file includes in this team;
- $Java\_Ave\_FileRevision\#$ = Java_Revision$\#/Java_File\#$;

Individual-Developer Level

Within a particular team, we want to look into each member's contribution, and suggest adjustments if necessary. We also ask each student developer to complete some questionnaires to tell us their backgrounds and experiences, describe their team construction and task allocation, and assess their own contributions and what they perceive as the contributions of their team mates. For each member, the following main parameters are gathered:

Directly Collected Parameters:

- $Oper\_Type_i\#$: the CVS operation distribution by type; $i \in [1, 13]$. See 13 types in Table 1;
- $Oper\_Date_i\#$: the CVS operation distribution by date; $i \in [1, Date\#]$;
- $AddedFile\#$: the total number of the files that were added by this member;
- $AddedJavaFile\#$: the total number of the Java files that were added by this member;
- $ModifiedFile\#$: the total number of the files that were modified by this member;
- $ModifiedJavaFile\#$: the total number of the Java files that were modified by this member;
- $LastModifiedFile\#$: the total number of the files that were last modified by this member;
- $LastModifiedJavaFile\#$: the total number of the Java files that were last modified by this member;

- $RemovedFile\#$: the total number of the files that were removed by this member;
- $RemovedJavaFile\#$: the total number of the Java files that were removed by this member;
- $AddedLOC\#$: the total number of new added Line of Code of this member;
- $RemovedLOC\#$: the total number of removed Line of Code of this member;
- $TrueAddedLOC\#$: the total number of new true added LOC (deleted all comment or empty lines) of this member;
- $TrueRemovedLOC\#$: the total number of true removed LOC (deleted all comment or empty lines) of this member;
- $WorkDay\#$: the total days that this member has CVS operations;
- $Java\_WorkDay\#$: the total days that this member has CVS operations on Java files;
- $TheFirstDate\_Type_i$: the first date when this member did the i^{th} CVS operation type, $i \in [1, 13]$;
- $TheFirstDate\_JavaFile\_Type_i$: the first date when this member did the specific i^{th} CVS operation type on Java files; $i \in [1, 13]$;
- $TheLastDate\_Type_i$: the last date when this member did the specific CVS operation type i; $i \in [1, 13]$;
- $TheLastDate\_JavaFile\_Type_i$: the last date when this member did the specific CVS operation type i on Java files; $i \in [1, 13]$;
- $MR\#$: the total number of MRs of this member;
- $Revision\#$: the total file revision number this member modified; The following parameters are applied for developers in academic environments:
- $PartScore_i$: the score of this member in the i^{th} project part.
- $FinalScore$: the final project score of this member;
- $PerformanceScore\_SelfEvaluate$: the score this member gave to himself/herself according to his/her performance;

- $PerformanceScore\_PeerEvaluate_i$: the score his/her i^{th} teammate gave to this member according to his/her performance. $i \in (0, Member\# - 1]$;

Derived Parameters:

- $Ave_M Rsize$: the average involved file number in a MR of this member;
- $IdleRatio$: The proportion of this member's idle days to the whole project duration;
- $The\ proportion\ of\ a\ this\ member's\ leading\ idle\ days$: the days between the start of the project and this member's first CVS operation to the entire project duration;
- $The\ proportion\ of\ the\ tailing\ idle\ days$: the days between this member's last CVS operation and the end of the project to the entire project duration, and
- $the\ proportion\ of\ various\ types\ of\ CVS\ operations\ on\ Java\ files\ to\ all\ CVS\ operations.$

The File Level

Above parameters enable the comparative analysis among individuals. To discover potential problems on the project design and the task division, more data about the project files themselves are relevant. For example, three potential problems may be related to files that have a high occurrence of colliding changes, end up being modified by multiple members, or been relocated frequently. For a specific file, the following main parameters are gathered:

Directly Collected Parameters:

- $Revision\#$: the total version number of this file;
- $BornDate$: the date that this file was created;
- $RemoveDate$: the date that this file was removed from CVS repository, if applied;
- $Creator$: the developer who created the first revision of this file;
- $FinalSize$: the LOC of the latest revision of this file;
- $Modifier\#$: the total developer number who modified this file;
- $FinalModifier$: the developer who finally modified this file and created its latest revision;
- $Oper\_Type_i\#$: CVS operation number of the i^{th} type, on this file; $i \in [1, 13]$. See 13 types listed in Table1;

- $Oper\_Date_i\#$: CVS operation number on the i^{th} day, on this file; $i \in [1, Date\#]$;
- $AddedLOC_j\#$: the added LOC number on this file by the j^{th} developer. $j \in [1, Member\#]$;
- $DeletedLOC_j\#$: the deleted LOC number on this file by the j^{th} developer. $j \in [1, Member\#]$;

Derived Parameters:

- $IdleRatio$: The proportion of this file's idle days to the whole project duration;
- *The proportion of a this files leading idle days*: the days between the start of the project and the $BornDate$ of this file to the entire project duration;
- *The proportion of this file tailing idle days*: the days between this file's $RemoveDate$ and the end of the project to the entire project duration;

The Revision Level

Through the analysis based on above parameters, users may notice some interesting or unusual revisions, such as heavy modification, frequent collision, modifier changing, and so on. For each file revision, the following parameters are gathered:

- $Author$: the developer who created this file revision;
- $TimeStamp$: the date and time this file revision was created;
- $AddeLOC\#$: how many lines of code were added in this revision?
- $DeletedLOC\#$: how many lines of code were deleted in this revision?
- $TrueAddeLOC\#$: how many pure lines of code (delete all comment and empty lines) were added in this revision?
- $TrueDeletedLOC\#$: how many pure lines of code (delete all comment and empty lines) were deleted in this revision?
- Log: the change rationale of this new revision;

To summarize, CVSChecker plugin collects the data from those data source we listed and abstract parameters according to the analysis purpose. Moreover, the parameter extraction is a crucial work for the following data storage, visualization, and analysis. With them, team managers and developers can become aware of the performance of an individual developer: his/her work times, work loads, work habits, main work products, performance evaluation and some problems that need to be addressed.

3.3.3 Visualization

A substantial amount of information can be extracted by examining the data directly collected by the CVS repository. Trends in these data can be inferred and presented through diagrams or reports, which leads to meaningful insights regarding the development of the team projects. CVSChecker plugin produces six visualizations based on the collected CVS data especially those extracted parameters for each project:

- Temporal distribution of CVS activity, for each member/team;
- Distribution of CVS operations by type, for each member/team;
- Distribution of CVS operations by type, for each file;
- Added and Deleted Lines of Code (LOC) by each member, on each file;
- Detailed LOC change by date, on a single file;
- File adding and removing by date, for a project;

Temporal Distribution of CVS Activity, for Each Member/Team

The first type of visualization (shown in Fig. 3.5) compares across team members in a team (or different teams in a class or a large project) the number of CVS operations over time. To better analyze the frequency and distribution of operations of interest, we have defined the concept of the interoperation gap (GAP). It refers to the interval between the times of two operations of interest. We choose "day" to be the unit of this measure: it is fairly easy and inexpensive to compute GAP in term of days, although not quite as precise as hour.

It can clearly show the busy (not busy) periods of a member/team and help users to grasp the development trends, have a quick idea about the typical GAPs, and notice some special phases for each member/team along the timelines. The aim of this chart is to compare the various work habits within different members/teams. How fast do they start? How long is their actual development process? How many idle days do they have? Which dates are their busiest times? In this manner, we can identify when the most important period of activity is for the entire project, or for a particular person, or for a file, or for a particular operation

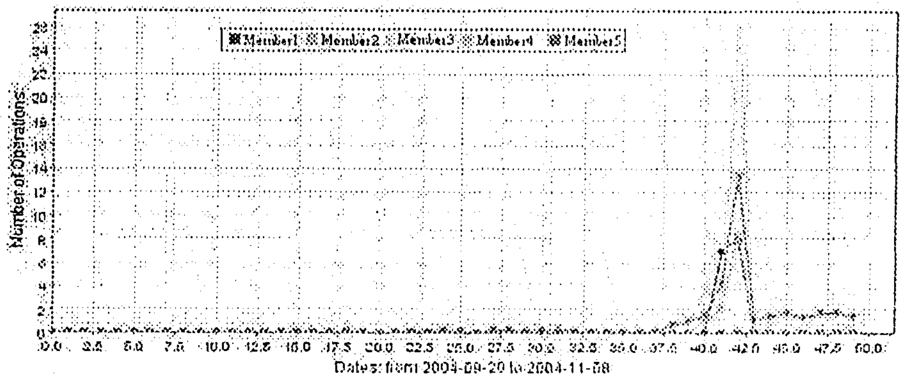


Figure 3.5: Temporal Distribution of CVS Activity, for Each Member/Team

type. The X-axis presents the whole project development process or a specific phase. The Y-axis is the number of operations.

Distribution of CVS Operations by Type, for Each Member/Team

From Fig. 3.5, users can have a rough idea about the total CVS operation amount of each member/team. However, what kind of work does each member/team did? The second type of visualization (Fig. 3.6) compares across members/teams the numbers of CVS operations and shows the contribution of CVS operations over different historical record types. From this chart, we can easily answer the following questions: which member/team did the most CVS operations? Who added the most files? Who removed the most files? Who had the most modification work? And who had the most collision and merges? The X-axis lists all the CVS historical record types with non-zero values. The Y-axis is still the number of operation. Each bar represents a member/team.

We believe that different team roles in the team composition result in different team-collaboration patterns; Based on all these visualization, especially this one, we can get support/oppose information.

Distribution of CVS Operations by Type, for Each File

The third type of visualization (Fig. 3.7) displays what kinds of operations were committed on each files and helps users to detect files with abnormal operations. The heights of the same-color sections indicate the operation distribution of all types. All the deleted files appear below the X-axis. The X-axis lists all files according to the modules. Each bar is a file and each color stack is a CVS historical record type that is listed in Table 3.1;

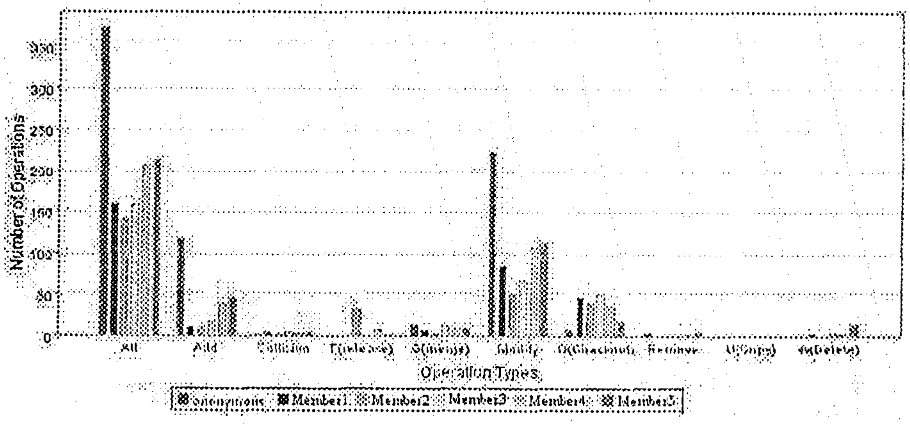


Figure 3.6: Distribution of CVS Operations by Type, for Each Member/Team

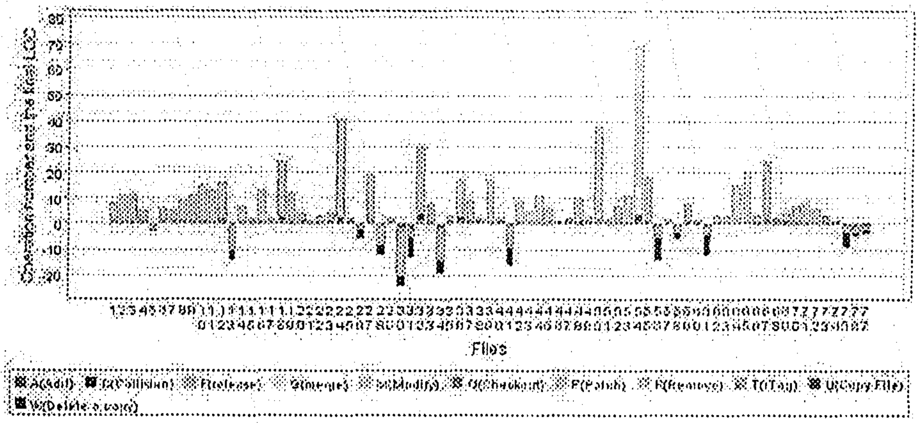


Figure 3.7: Distribution of CVS Operations by Type, for Each File

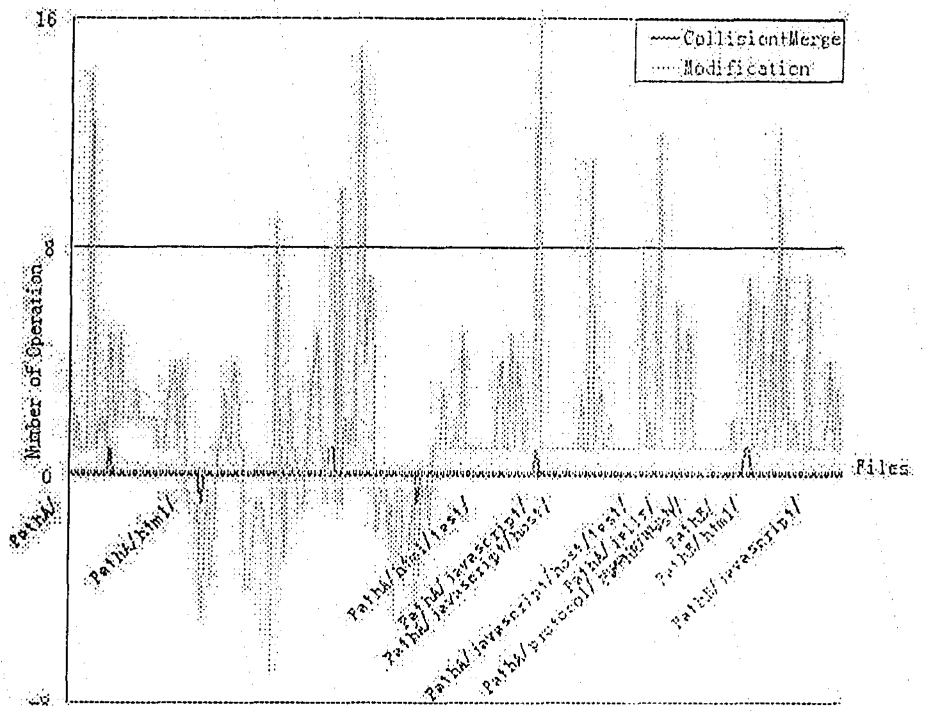


Figure 3.8: Simplified Distribution of CVS Operations by Type, for Each File

The dark and light-gray are two colors that should be paid much attention to. Dark color indicates the number of collisions while the light-gray shows the number of merges happened on this file. When we get this visualization, we can detect these files with inappropriate design easily.

In order to increase the chart readability, we designed some simplified versions for Fig. 3.7: These simplified visualization only display some interesting type classes listed below Table 3.1. Fig. 3.8 is an example that only compares the constructive and red-flag CVS operation types (see Section 3.3.1).

Added and Deleted LOC by Each Member, on Each File

The fourth type of visualization (Fig. 3.9) provides relevant information for each file about who modified it and what the impact of each developer was on that file. This chart can help users to know the modification amount of each member. Moreover, for those dangerous files detected in Fig. 3.7/Fig. 3.8, this chart helps us to know who modified them and who their main developers were.

This chart is also a stack bar chart. The X-axis is same as it in Fig. 3.7. The Y-axis is

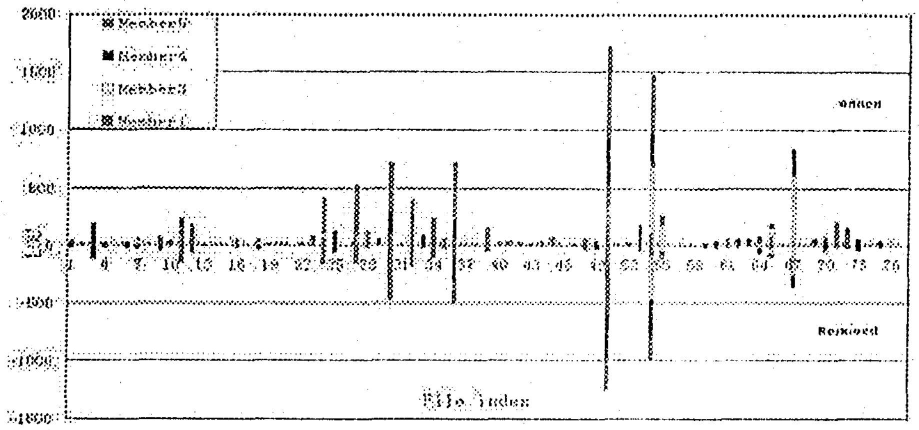


Figure 3.9: Added and Deleted LOC by Each Member, on Each File

the modified LOC number. All the Removed LOCs appear below the X-axis. Each member has the same color for his Added- and Removed-LOCs.

The more sub-bars appear in a single column, the more attention should be put on the corresponding file, since it might be the locus of increased activity, possibly because it is ill-designed and ill-understood.

Detailed LOC Change by Date, on a Single File

When we locate a problem file, detailed information about its versions might be interesting. Fig. 3.10 shows its detailed development history along the time. The X-axis is date. The Y-axis is still the number of modified LOCs, and the removed LOCs appear below the X-axis. This visualization can be understood as the "curriculum vitae" of a specific file.

File Adding and Removing by Date, for a Project

The file adding and removal data contains interesting information. It can help users to better understand the project development process. We design a chart (see Fig. 3.11) to show the adding and removing of all files in a project within a defined time phases.

The X-axis lists all files according to their modules. The Y-axis lists all the dates from the beginning of the project development process. A dark spot represents a new-added file and a light spot shows a removed file from the CVS repository. This visualization can help users to detect some interesting and promising phenomenon, such as a set of same birthday/removing-day files, some short-life files, and so on.

All these visualizations can be produced for the whole project history, or incrementally

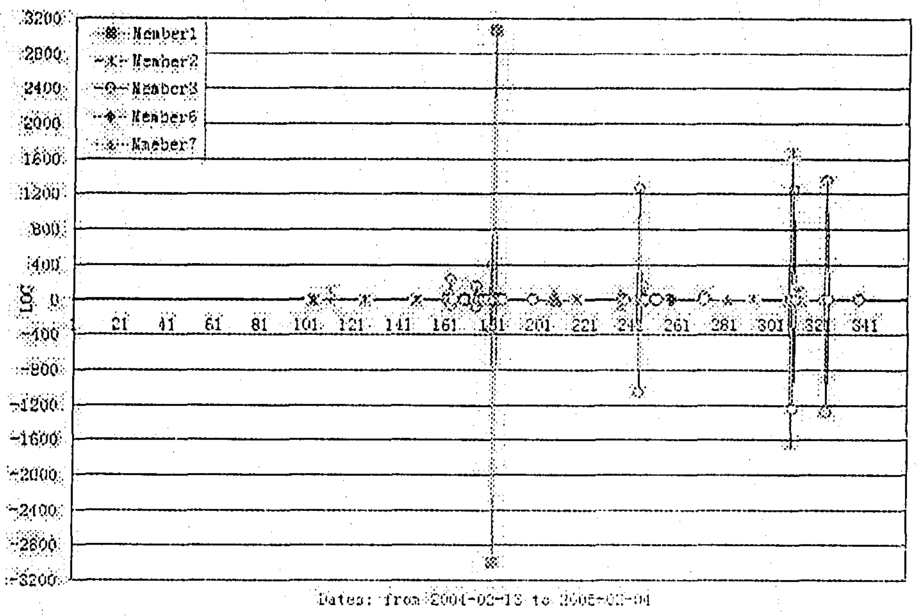


Figure 3.10: Detailed LOC Change by Date, on a Single File

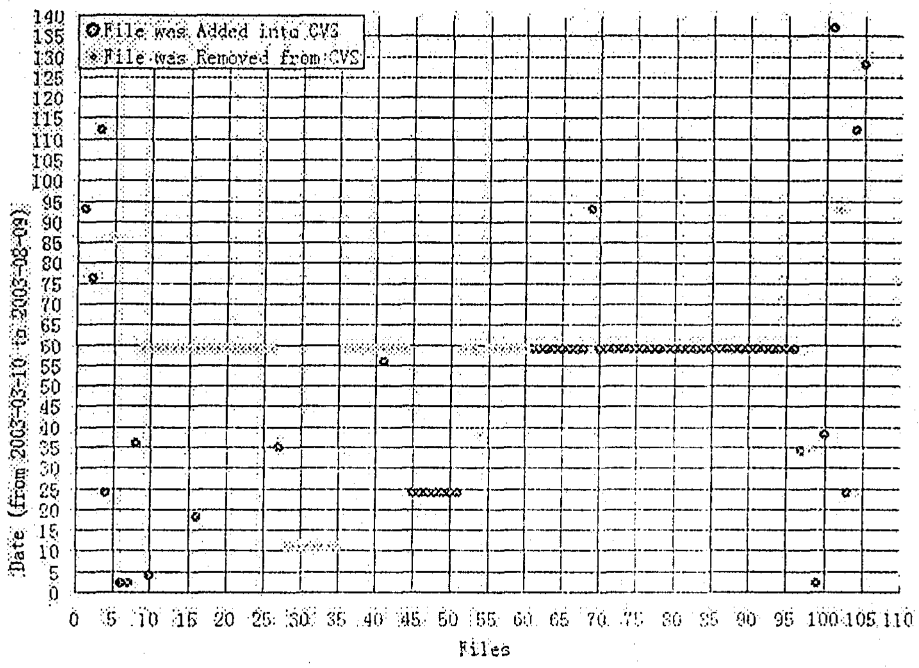


Figure 3.11: File Adding and Removing by Date, for a Project

to provide a sequence of views corresponding to smaller periods (such as between releases, or on a weekly basis.), specific modules, or some noticed developers and files.

CVSChecker has a visualization trigger that enables users to use Eclipse to interactively explore the first four visualizations through a special CVSChecker perspective. The trigger interface in Eclipse contains multiple choice of operations to generate the first four visualizations we introduced above. An up-to-date set of visualizations reflecting the complete project history is maintained in the database. They can be selectively queried to focus on particular CVS operations or on individual team members or specific periods of time to help users instantly grasp the trend and the level of individuals through the comparison among the lines or columns. Developers also can access them through a special-purpose wiki-based collaborative environment, WikiWikiDev [80], to get an up-to-date view of their progress.

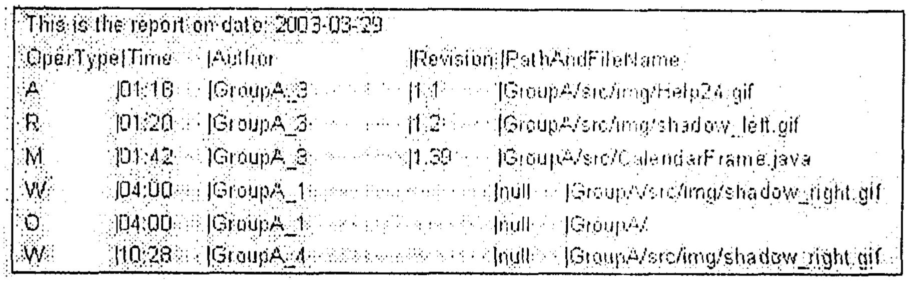
3.3.4 Reporting

Reporting is an alternative means of presenting information, complementary to visualization. In our research, our reports can be classified as two main types: consultation report and summarization report. CVSChecker plugin includes triggers to generate these reports, from multiple levels, such as: whole project, a team, an individual, a specific file, a day and so on. Of course, new triggers can also be added for new reports and visualization as the analysis goes on. All these reports are saved in Database and all the users can view them in Wiki pages.

Consultation reports

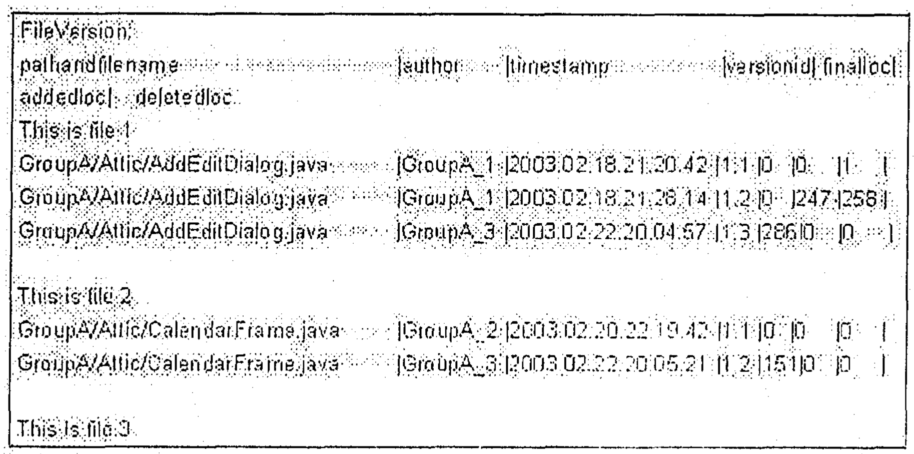
Consultation reports include the detailed data on the basis of which the visualizations are generated. These reports are meant as an auxiliary medium for representing the visualization data, enabling the users interpreting the diagrams to access the details behind any interesting information they may perceive from the diagrams. The query function of CVSChecker can provide a similar assistance. However, because it is embedded in Eclipse platform, the query condition setting and the result layout are still limited. Moreover, it is only available for Eclipse users. The reporting function of CVSChecker is a wonderful supplement, especial for those non-Eclipse users.

The information in the consultation report includes detailed data by multiple levels, such as: an operation type, an individual, a role, a day, a module, a specific file, a revision, and so on. It is treated as the auxiliary knowledge for visualization, and provides a consulting base



```
This is the report on date: 2003-03-29
OperType|Time   |Author                |Revision|PathAndFileName
A       |01:18  |GroupA_3              |1.1     |GroupA/src/img/Help24.gif
R       |01:20  |GroupA_3              |1.2     |GroupA/src/img/shadow_left.gif
M       |01:42  |GroupA_3              |1.39    |GroupA/src/CalendarFrame.java
W       |04:00  |GroupA_1                       |null   |GroupA/src/img/shadow_right.gif
O       |04:00  |GroupA_1                       |null   |GroupA/
W       |10:28  |GroupA_4                       |null   |GroupA/src/img/shadow_right.gif
```

Figure 3.12: A segment of DailyOperation report



```
FileVersion:
pathandfilename                      |author     |timestamp              |versionid|finalloc|
addedloc|   deletedloc
This is file 1
GroupA/Attic/AddEditDialog.java      |GroupA_1 |2003.02.18.21.20.42 |1.1 |0   |0   |1   |
GroupA/Attic/AddEditDialog.java      |GroupA_1 |2003.02.18.21.28.14 |1.2 |0   |247 |258 |
GroupA/Attic/AddEditDialog.java      |GroupA_3 |2003.02.22.20.04.57 |1.3 |286 |0   |0   |

This is file 2
GroupA/Attic/CalendarFrame.java      |GroupA_2 |2003.02.20.22.19.42 |1.1 |0   |0   |0   |
GroupA/Attic/CalendarFrame.java      |GroupA_3 |2003.02.22.20.05.21 |1.2 |151 |0   |0   |

This is file 3
```

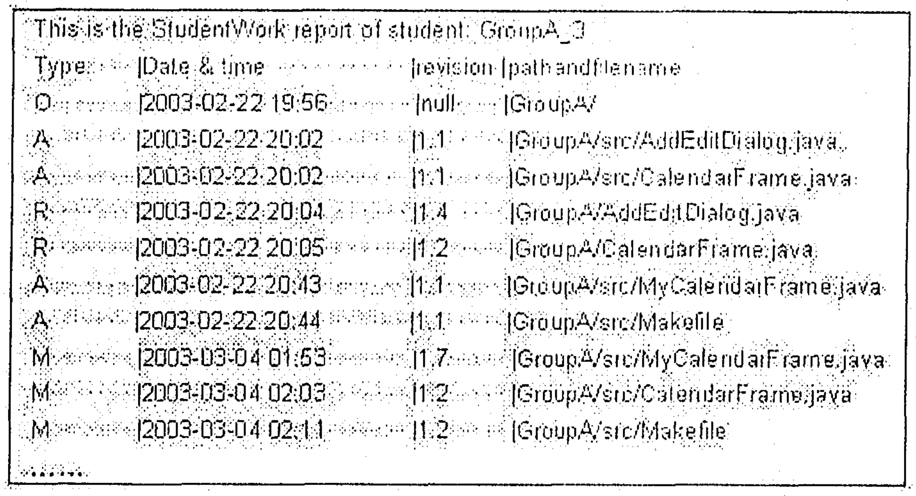
Figure 3.13: A segment of FileVersion report

for those potential patterns or problems noticed from diagrams. We introduce three typical consultation reports: DailyOperation report, FileVersion report, and StudentWork report.

In a team, each date has a DailyOperation report. It lists all the CVS historical operation records in repository committed by all the members in this team, happened on that specific date. All the records are listed chronologically. Fig. 3.12 is a snapshot of the DailyOperation report.

Each team has a FileVersion report. It is the corresponding report for the Visualization 5 (see section 3.1.3) of all the files in a team. It can also be understood as a "curriculum vitae" of files. The detailed modifications on each file are displayed from revision 1.1 to the latest one. Fig. 3.13 is an example.

Each member in a team has a StudentWork report. It displays all the CVS operations of each member since the project development beginning, chronologically. Fig. 3.14 is a snapshot.



```
This is the StudentWork report of student: GroupA_3
Type:    |Date & time        |revision |pathandfilename
D        |2003-02-22 19:56   |null     |GroupA/
A        |2003-02-22 20:02   |1.1      |GroupA/src/AddEditDialog.java
A        |2003-02-22 20:02   |1.1      |GroupA/src/CalendarFrame.java
R        |2003-02-22 20:04   |1.4      |GroupA/AddEditDialog.java
R        |2003-02-22 20:05   |1.2      |GroupA/CalendarFrame.java
A        |2003-02-22 20:43   |1.1      |GroupA/src/MyCalendarFrame.java
A        |2003-02-22 20:44   |1.1      |GroupA/src/Makefile
M        |2003-03-04 01:53   |1.7      |GroupA/src/MyCalendarFrame.java
M        |2003-03-04 02:03   |1.2      |GroupA/src/CalendarFrame.java
M        |2003-03-04 02:11   |1.2      |GroupA/src/Makefile
......
```

Figure 3.14: A segment of StudentWork report

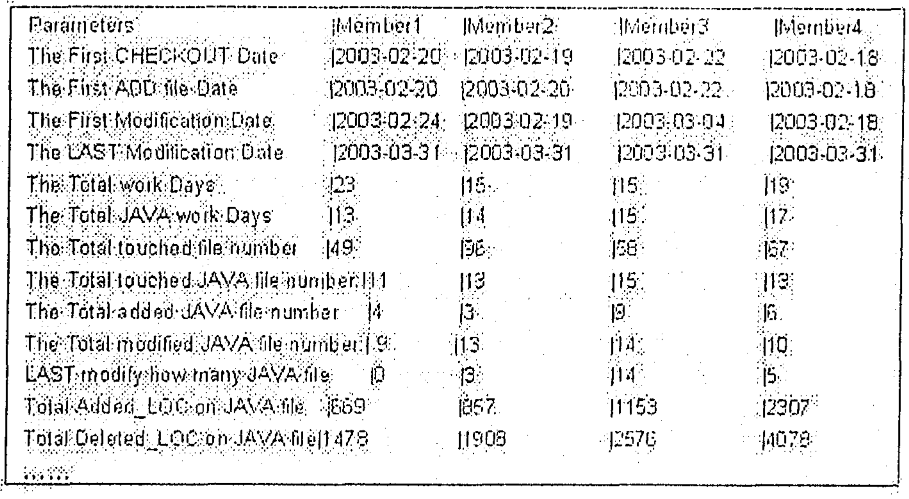
Summarization reports

Summarization reports contain selected interesting data and the results of the statistics and data mining analysis phase. The summarization report constitutes a simple overview of the teamwork. It usually picks the interesting attributes out, itemizes them in a meaningful and comparable order (see Fig. 3.15). Descriptive annotation is also added for some important attribute values. Moreover, we also list the heuristic analysis results from the analysis in Section 3.1.4, especially those red-flag patterns.

This report can be viewed as a decent lively team working description instead of a simple summary. It assists users to learn a team's collaboration as well as each member's detailed performance, with a condensed textual description.

3.4 Further analysis on the CVSChecker data

Version Control System accumulates numerous dispersed contextual information along the developing process. No matter what perspectives each analysis focuses on, there are essential phases in common: collecting data, filtering out useful ones, storing and visual presenting them. However, if we want to understand a process thoroughly, they are still far from enough. Heuristic analysis should be applied. CVSChecker includes two different mechanisms for interpreting the collected data: data mining analysis, and heuristics-base analysis.



```
Parameters                          |Member1      |Member2      |Member3      |Member4
The First CHECKOUT Date             |2003-02-20   |2003-02-19   |2003-02-22   |2003-02-18
The First ADD file Date             |2003-02-20   |2003-02-20   |2003-02-22   |2003-02-18
The First Modification Date         |2003-02-24   |2003-02-19   |2003-03-04   |2003-02-18
The LAST Modification Date          |2003-03-31   |2003-03-31   |2003-03-31   |2003-03-31
The Total work Days                 |23           |15           |15           |19
The Total JAVA work Days            |13           |14           |15           |17
The Total touched file number       |49           |96           |58           |67
The Total touched JAVA file number  |11           |13           |15           |13
The Total added JAVA file number    |4            |3            |9            |6
The Total modified JAVA file number |9            |13           |14           |10
LAST modify how many JAVA file      |0            |3            |14           |5
Total Added_LOC on JAVA file        |669          |857          |1153         |2307
Total Deleted_LOC on JAVA file      |1478         |1908         |2576         |4078
......
```

Figure 3.15: A segment of Summarization reports

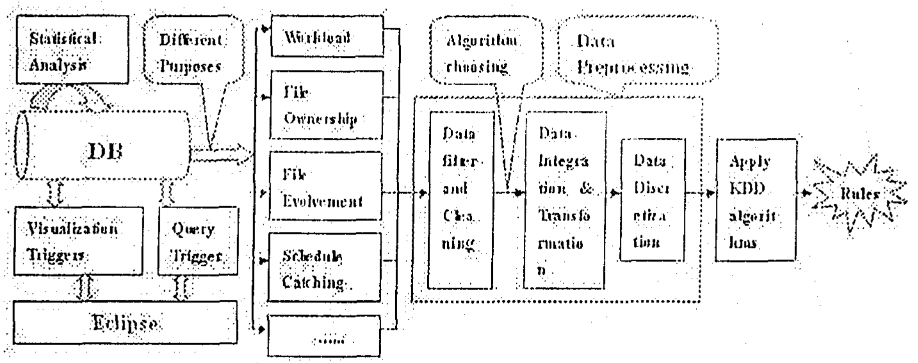


Figure 3.16: Work flows of Bottom-up Hypothesis Generation and Knowledge Extraction

3.4.1 Data mining

In this step, CVSChecker plugin supports users to extract the high-level knowledge hided along the project development process from the original data. Fig. 3.16 shows the main work flows of this mechanism.

Although CVSChecker plugin parses all the related data into the database and provides several visualizations to help users to understand them, directly accessing filtered data from the database is still very difficult. Statistical Analysis commits basic but important statistical works, and enables users have a quick concept about their performance and collaboration. It plays the role as an agent for users to observe concise and easily understandable statistic results of some main attributes. Users will decide whether they have to track down to certain codes or not.

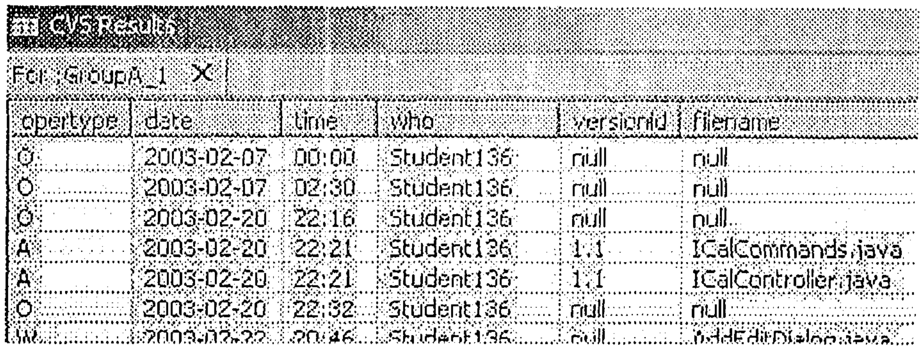
Before the knowledge extraction, users should have specific demands on learning schemes. Different knowledge in different expressions exists for a settled analysis purpose, such as Association Rules, Sequence Patterns, Classification Rules, Clusters, and so on. If users want to analyze a developer's operation patterns, "associate rules" can find some concomitant operation sets of this developer, while "sequence pattern" captures some frequent operation orders. Each learning scheme has its corresponding algorithms. In our data mining analysis and knowledge extraction, we want to get some knowledge about individual workload, file ownership, file evolution, and schedule catching. Apriori [17], an association-rule mining algorithm for discovering interesting patterns, is adopted to know how team members use and modify their software assets.

Most heuristic algorithms have special data format requirements. Extended phases were designed in CVSChecker plugin for applying data-mining techniques on these process- and performance-related data. All these phases are belonging to data pre-processing, which impacts the quality of results directly. The current data in the database is not good enough to be used directly for these analyses, more preprocessing works are needed to change data into non-redundant, discrete, information- centralized datasets in standard formats for heuristic analysis.

Although users already collected useful data and extract attributes according to the analysis purpose, excessive continuous values with minor differences are inappropriate for some algorithms. Thus, changing them into discrete data is a feasible way to eliminate the gaps and keep all values stay at their original levels. It is also redundant analysis if an attribute is highly correlated with another. Data integration [6] can detect and deal with it. Data transformation [17] is another way to change data into appropriate forms without losing its value. We applied Min-Max Normalization [17] for all large-range attributes with known minimum and maximum values, and scaled them within a specified range using a linear transformation.

3.4.2 User-driven data exploration

If users notice an abnormal or interesting result from visualization, statistical analysis, or mined knowledge, they can examine the database through a simple query interface provided by CVSChecker plugin. Query conditions can be easily set, and results will be displayed in a neat format. Once CVSChecker plugin is installed, and the perspective is properly set, users can start exploring the utilities that the CVSChecker provides by clicking the menu


CVS Results

For ;GroupA_1

opertype	date	time	who	versionid	filename
O	2003-02-07	00:00	Student136	null	null
O	2003-02-07	02:30	Student136	null	null
O	2003-02-20	22:16	Student136	null	null
A	2003-02-20	22:21	Student136	1.1	ICalCommands.java
A	2003-02-20	22:21	Student136	1.1	ICalController.java
O	2003-02-20	22:32	Student136	null	null
W	2003-02-22	20:46	Student136	null	AddEditDialog.java

Figure 3.17: Result view of CVSChecker query function in Eclipse

bar "Execute CVS".

If the valid username and password have already been filled, a window will be popped out. Otherwise, there will be no any response, and users should check the login information. There are four combo boxes in this window: "Operation Types", "Date From", "Data to", and "Operator". "Operation Types" lists all CVS historical operation types listed in Table 3.1.

"Date From" and "To" help users to set the period which they want to look into.

And "Operator" enables users to pick out the objects that the users are interested in their works. It can be a member or a team, even the whole class for a project in university environments.

This window interface to help users make queries about their historical works. By clicking "Finish", "CVS Results" view (see Fig. 3.17) will show the query results according to query condition settings.

Of course, user can use the query function first as they want, if the restrictions are too loose and its large-size results are hard to read, users can generate an instant view from the graph point of view on these query results to have a more vivid understanding. In addition to diagrams, CVSChecker plugin also includes triggers for reports, from multiple levels. Such as: project, team, individual, package, specific file, and so on. We will discuss it in section 3.3.4.

3.4.3 Heuristics-based analysis

Visualizations, querying, statistical analysis and KDD analysis uncover correlations in the collected data that may or may not correspond to interesting development behaviors; we would have to assess these correlations in terms of how useful they are as indicators of ef-

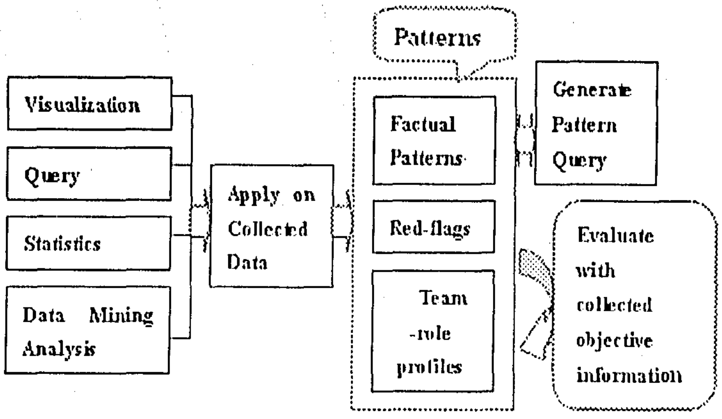


Figure 3.18: Top-Down Hypothesis-Driven Analysis in CVSChecker

fective or problematic performance. At the same time, in the academic setting we formulate our own intuitions as educators about what types of teams succeed (and what types of teams fail) as heuristic patterns in terms of the collected data and we apply them so that we can evaluate their empirical validity. With this approach, we plan to collect a suite of patterns that can be used as "sensors" of when to intervene in a team and how. Fig. 3.18 depicts the main works of the heuristics-driven analysis based on above parts of CVSChecker plugin. This analysis can be understood as training, evaluating, testing, diagnosing and predicting. It can be divided into the following four steps.

- Step 1: We randomly select some team projects from different environments, then apply visualization, querying, statistical analysis and the knowledge extracted in above section on the selected data.
- Step 2: For heuristics-driven analysis, we observe and summarize some typical patterns on the selected data, related to the individual operation, team collaboration, file evolution, module design, and so on. We categorized the patterns into three types: factual patterns, red flags, and team-role profiles. A factual pattern expresses some characteristic of the development history of no obvious negative or positive implication. A red flag captures a problematic situation whose persistence may warrant a preventive action. They should ideally be detected early and avoided. A team-role profile focuses on the characteristics of typical team roles. We summarize some of these interesting patterns from the team projects in different environments, and compare with each other.

- Step 3: Then, we evaluate these patterns with the collected project/team objective information, such as project history, questionnaire fillings, etc.

- Step 4: Of course, what we are interested in is not only the past performance, but also the likely future performance on new team projects. Afterward, we have been working on developing a set of heuristics for understanding the nature of the collaboration among the members of the development team and their roles, and have developed a set of queries that correspond to our intuitions about relevant (both desirable and undesirable) behaviors of teams and individuals at a high level.

In the long run, our intent is to provide context-specific guidance to team managers and developers, based on the actual patterns of behaviors that the team members exhibit as individuals and as a whole.

The results of the analysis component are also stored in the database and are served by the Wiki server in the context of various reports. Some reports are tailored towards the student developers, and they present information specific to the team and comparisons against aggregate data from the other teams. Other reports are tailored to the instructor and they present detailed comparative data across multiple teams.

Chapter 4

An Exploratory Case Study on Five Undergraduate Student Teams

In most engineering disciplines, it is assumed that the education of their professionals involves an apprenticeship component, in addition to their formal training. This is why most undergraduate software-engineering programs involve a capstone project course, where students, in preparation for becoming software professionals, work in teams to design, develop and document a substantial software system. Our experience with such a course has been that the success of such a team project depends, on one hand, on the technical competency of the students, the quality of the tools they use, and the project-management decisions they make during the project life-cycle, and, on the other, the specific and timely feedback of the instructor is invaluable. However, instructors of such courses are, more often than not, overwhelmed with the task of closely monitoring the progress of multiple teams, and problems in a team's process and product may go unnoticed until it is too late to be addressed. This exploratory case study is an evaluation of whether CVSChecker might help instructors to monitor student teams and detect problematic collaboration and performance earlier.

4.1 Objectives

New or volatile requirements, tight delivery schedules and developer turnaround are common challenges facing almost every software-developing team today. To effectively deal with such obstacles requires that the developers have an overall understanding of the current status of their project, possess sufficient programming experience, collaborate effectively within their teams, and are able to react promptly. Such capabilities are difficult to teach and to acquire in the context of a university software-engineering course. Instructors are eager to equip their students with the "tools of the trade" but closely monitoring a large number of software teams in the classroom and acting as a "mentor manager" to all the team members is too difficult a task. The number of teams and developers is usually large and there usually are substantial variations among the team projects, which makes the detection of individual problems too subtle. On the other hand, even with a solid technical background, students may still get overwhelmed by the complexity of the software-construction process and may fail to recognize signs of problems in their own project early enough so that they can involve the instructor.

As we elaborate in Chapter 3, CVSChecker is a method that can track the progress of students developing a term project, using the historical information stored in their CVS repository. This information is analyzed and presented to the instructor in a variety of forms. In this chapter, we discuss a set of analyses that support monitoring student teams and their progress, based on the collected information. These analyses infer a multiple-perspective trail of the project development, and a set of corresponding visualizations presents various statistics, charts, and reports on this trail. Based on the information produced, instructors can track the evolution of a team's work against other teams or compare the performance of members within a team. Furthermore, instructors can inspect the revisions of an individual file. All the methodologies of CVSChecker are followed in the case studies.

We conducted this case study based on undergraduate students' project data because they are the representatives of novices, and university course plans a significant role in the training of future developers. The main goal of this case study was:

- To examine whether CVSChecker plugin works well on teams in a university environment.
- To enable instructors or TAs to have an overview of the whole development process of each team, and detect some teams or students with abnormal phenomena.
- To provide visualization, query, and heuristic analysis on the student historical data, within the whole development process or a specific phase.
- To get some initial validation of whether the data visualizations we developed are informative to instructor.
- To get insight on specific "interesting trends and events" in the project life-cycle that can be discovered through the CVSChecker analyses.
- To reveal correlation of these trends and events with the eventual outcome of the team's work, i.e., the quality of their project, and the correlation of the team roles

with the development habits. If indeed such correlations exist, then discovery of similar trends in the profiles of future teams might be used as evidence in predicting the likelihood of the team's success or failure to deliver a good project; such evidence could support the instructor in monitoring the teams and in discovering quickly potential problems.

To sum up, the goal of this analysis is to understand how students interact and to find out if there is any correlation among the educational environment, roles, their grades and the nature of their collaboration. Understanding these factors will enable instructors to detect potential problems early in the course of the students' projects, so they can concentrate their help on those teams who need it the most.

We believe that if the information is suitably presented and highlighted, it could be useful to students to self-evaluate their own progress and quickly notice symptoms. In addition, teams in educational environment have their specific patterns on individual performance and team collaboration.

4.2 Settings

This case study was conducted on a third-year software engineering course in software engineering. In the context of this course, students work on a project in teams of four and coordinate their software changes using CVS. The project is common across all teams, with three delivery dates spanning a two-month development period. Although various deliverables are required, including unit test cases, UML diagrams, and a user manual, we initially focus our analysis on changes to the source code and the CVS operation records over time.

Students in this course have a substantial background of program development in-the-small, and are knowledgeable in programming with Java in the object-oriented design style. However, for most of them, the course project is their first experience in collaborative software development.

The project work is organized in three or four cycles, each culminating in a deliverable. At the end of the first cycle, a low-fidelity paper prototype and the object-oriented design of the project, represented in a UML class diagram, are due. At the end of the second or the third cycle, a working horizontal prototype is due, exhibiting the interactive functionalities of the project but not necessarily the underlying support functions. Finally, the whole working project is due at the end of the last cycle.

Phase	Period	Length	Comments
Phase1	Feb.07 – Feb.21	15 days	Deadline for Assignment 3
Phase2	Feb.22 – Mar.17	24 days	Deadline for Project part2
Phase3	Mar.18 – Mar.31	14 days	Deadline for Project part3

Table 4.1: 3 Project-development phases

On each due date, a snapshot of each team's CVS repository is extracted and each team member submits an electronic evaluation form to assess the contribution of all team members, including themselves.

Putnam et. al. [58] claims that small size is the key to a successful project, and Belbin [4] has a "9 team roles" theory. It is coincidental that the team size in our course project is also small. Each student has very busy timetable and has to attend several courses in a single term. It is very difficult to regular much common meeting and working time among team members if the team size is large.

Our case study involved 85 students (*Student#*) organized in 23 teams (*Team#*). Most teams have four members, with some exceptions (three or five) because of some drop outs and recombination. 51 students (including the team members of 5 whole teams) gave us permission to use their data. Because one of our analysis emphases is the team collaboration, we focus on these five whole teams. And refer them as team A, B, C, D and E in the following sections.

Students usually can not spend too much time on the course project because of the curriculum design. The total duration of the project was 53 days, which was divided into 3 main phases, as shown in Table 4.1.

As for the processes, we want to investigate whether different processes exhibit their own particular patterns. "Academic case-study" projects in comparable size, length and environment without explicit process types can be adopted as the shared compare benchmarks for other projects.

4.3 Basic Results

This section introduces selected visualizations and heuristic statistics generated in our case study on five student teams (labeled A to E). Diagrams are presented at various levels: by team, by individual, or by file. Such diagrams intuitively show trends, enabling the users to gain a high-level impression of team and individual performance.

CVS Operation	Team A	Team B	Team C	Team D	Team E	All Class
Total	196.5	288.5	208.25	242.0	132.0	213.35
$A\#(Add)$	37.75	46.75	39.5	66.0	44.75	46.95
$C\#(Collision)$	9.25	7.25	3.75	3.5	2.0	5.15
$F\#(Release)$	0.0	0.0	8.5	0.0	0.0	1.7
$G\#(Merge)$	9.5	20.5	10.0	4.75	2.25	9.4
$M\#(Modify)$	84.75	161.0	107.25	149.0	56.0	111.6
$O\#(Checkout)$	20.0	5.25	36.0	12.0	19.5	18.55
$R\#(Remove)$	11.0	10.75	1.25	3.25	2.0	5.65
$W\#(Delete)$	23.75	37.0	2.0	3.5	5.5	14.35
$M\#/C\#$	9.162	22.2	28.6	42.75	28	21.67
Total Files	18	27	42	67	51	49.7
Modified Files	17	27	32	41	43	23.3
Versions Per File	19.9	23.85	13.4	14.5	5.2	19.5

Table 4.2: The numbers of CVS operations of five student teams

4.3.1 The Team Aspect

Three visualizations help us to do the analysis from the team aspect: the average temporal distribution of CVS activity, the average distribution of CVS operation by type, and temporal distribution of Modification Request (MR) of the selected teams. The aim of these team-level diagrams is to compare the various work habits of the student teams. How fast did they start? How long were their actual development processes? How many idle days did they have? How many files did they work on at a time? What proportion of files were Java files? What was the distribution of their CVS operations?

Let us now look at the information that can be inferred by examining all the types of operations that teams performed in their CVS repositories. By examining the bar chart as Fig. 4.1 and checking the data reported in Table 3, the instructor can gain some insights into the ways the various teams use CVS.

As can be seen from this chart and table, team E exhibits "abnormal" small numbers of operations of all types: the members in this team performed - on the average - the smallest number of operations in CVS (less than half the number of operations of team B), but it has many addition and checkout (A and C) operations and the number of files they developed is bigger than average. Furthermore, they seem to have used CVS much more like a storage area for finished products than as a working repository: both in absolute numbers and on the average. They modified their CVS files much less frequently than other teams: averagely every file was modified only 5.2 times by the all four team members.

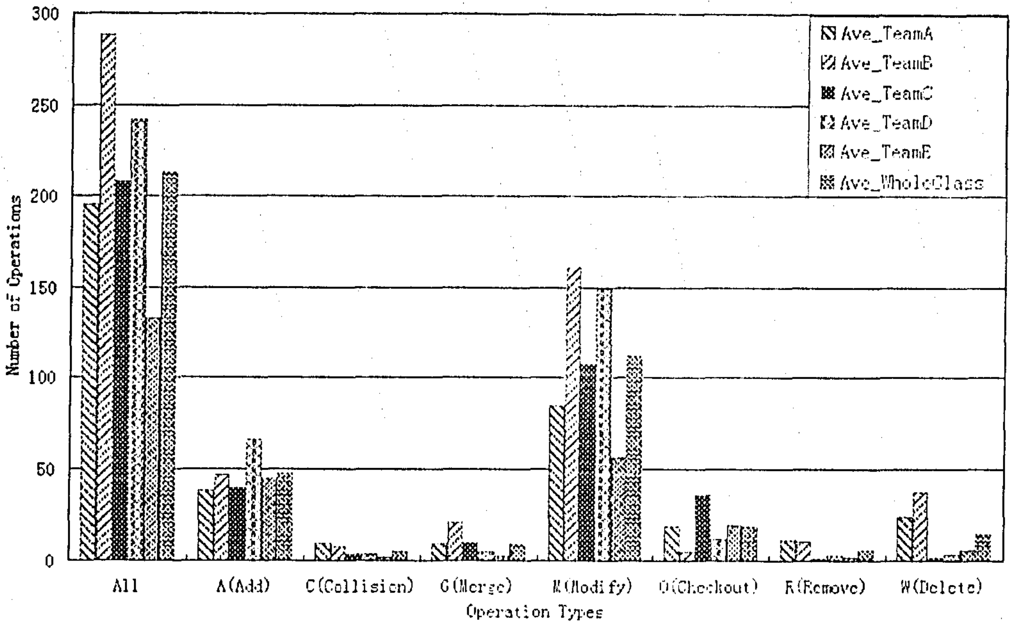


Figure 4.1: Distribution of CVS Operations by Type, for Five Teams

Such "abnormal" phenomena should not emerge suddenly at the last minute. Instructors may examine CVSChecker visualizations regularly, recognize some symptoms at an earlier stage, and evaluate whether the developers are facing any problems or not. For example, sparse usage of CVS might be due to the fact that the team is simply storing and exchanging files outside the CVS. Alternatively, it may be due to the fact that the team is not working enough on the project. Based on the cause analyses, instructors may decide what kinds of action they should execute. As another example, we notice that TeamA and TeamB have a slightly high number of collisions (C) and merge (G) operations. Collisions and merges occur when more than one team members attempt modify a same file at the same time. A substantially high number of C and G operations could indicate that the design of the software product and the distribution of tasks among team members are poor, and the project modularization should be re-considered. We also notice that these two teams have high number of collisions and merges over a relatively small number of files.

In principle, enabling team members to always have the latest version of each file is a good collaboration habit. The instructor, in fact, recommended that students commit new versions back to CVS repository promptly after their modification and do not modify a file heavily without saving it in CVS so that the other team members can have up-to-date local copies; if these instructions are followed, the average number of modifications, $M\#$, should

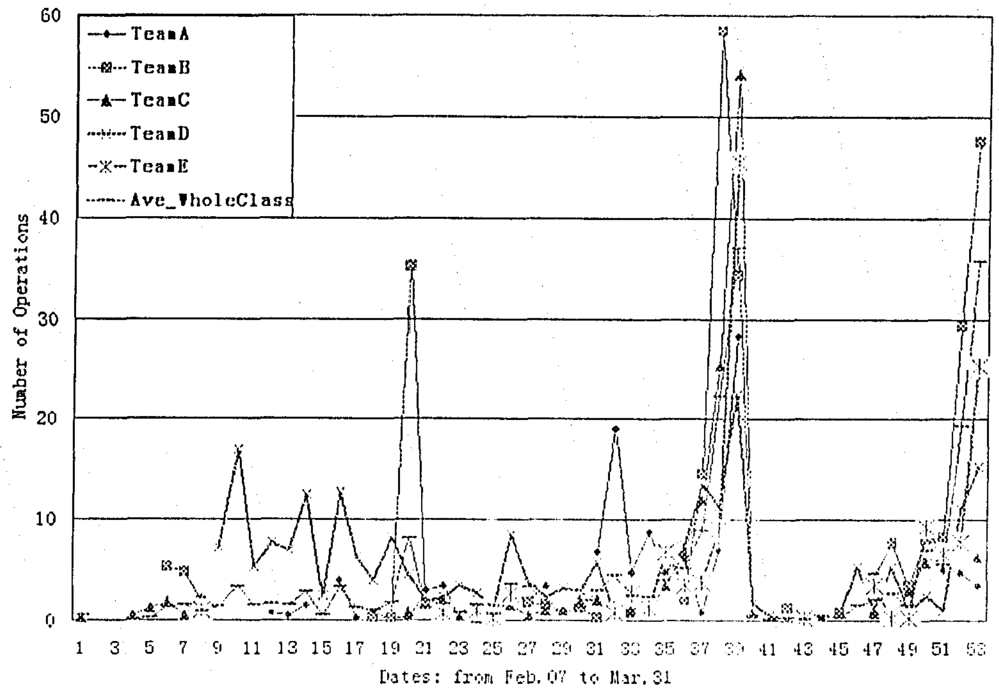


Figure 4.2: Temporal Distribution of CVS Activity, for Each 5 Teams

not be small. However, the $M\#$ of TeamE is much smaller than those of other teams (almost is one third value of TeamB). If the design of the application is not sufficiently detailed and only high-level classes with substantially complex functionalities have been designed, then it becomes more likely that more than one member will have to touch the same file at the same time thus resulting in a higher number of collisions.

Integrating the above heuristics, we can say that the higher the ratio of successful merges over collisions ($M\#/C\#$) the more effective the team collaboration is, since either their design or their inter-personal communication enables them not to step on each other's work products. From Table 3, we can see that the ratio $M\#/C\#$ of TeamD is the highest, where the same metric for TeamA is the lowest. The problem of TeamA seems to be the small number of files in which they have divided their work - i.e., the small number of classes they have identified in their project design; if they had further decomposed their classes into several, simpler and more independent parts, they might have obtained a much better task assignment, module design and file-sharing habits.

Fig. 4.2 diagrammatically presents the average workloads for the students of the five teams through the whole process day by day.

It is easy to see that all teams show peaks of activity around the same dates in the

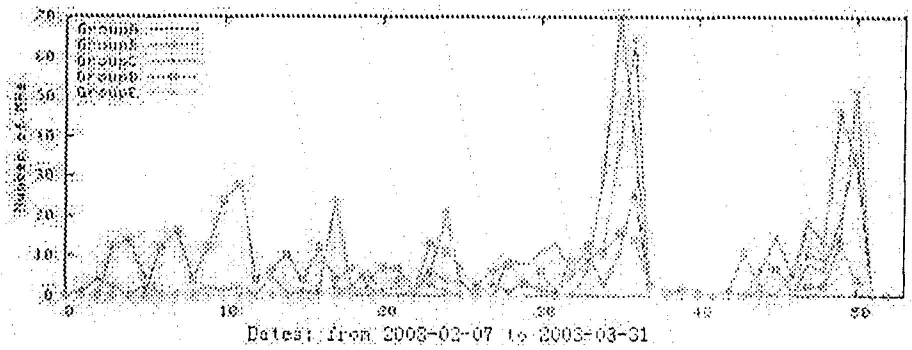


Figure 4.3: Temporal Distribution of MR Activity, for Each 5 Teams

second and third periods (the due day of the project part 1 is the day 15). However, there are some interesting differences too. TeamC began earlier than the other teams, TeamB usually worked in a single day then stop for next several days, and TeamD exhibits a much more consistent work profile than the rest. With this figure, instructors may notice that a team has not started development, when most other teams have and may give them a prompt reminder.

Fig. 4.3 shows the average Modification Request (MR) numbers of these 5 teams. The MR curve trend of each team is consistent with that in Fig. 4.2.

Considering all these charts, we observed that TeamB has the most CVS total operations, modification operations, with pretty high collision and merge numbers. TeamD has more regular workload habits than the other teams, and has a medium number of MRs. TeamB, C, and E have sharp peaks around each delivery deadline, preceded by long idle periods. TeamB has the most MRs at the second deadline while TeamC has the smallest number of MRs. The trends and relative positions of the curves in MR chart are coincident with that of curves in Fig. 4.2. This means that all the teams have similar average MR sizes;

4.3.2 The Individual-Developer aspect

In above section, we examined the aggregate behavior of teams. In this section, we focus on analyzing the five teams from individual behavior aspect. The analysis may support the instructor in assessing the relative contribution of each team member to the project and to notice quickly imbalances in the workload distribution.

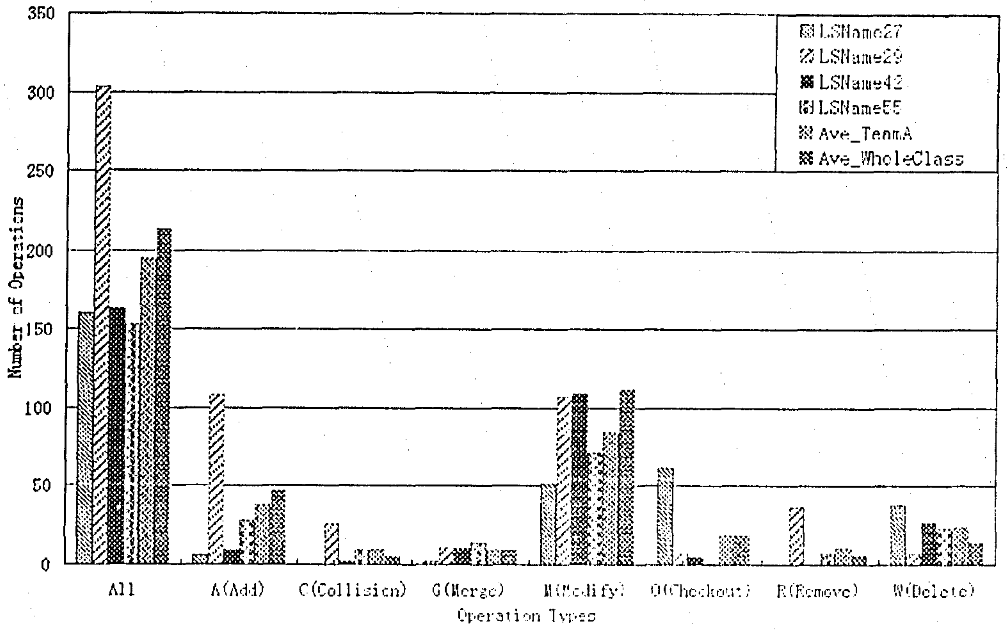


Figure 4.4: Distribution of CVS Operations by Type, for Members in TeamA

TeamA

A simple, yet potentially telling, metric of the nature of the collaboration among the members of a team is the number of their CVS operations according to their type. Fig. 4.4 shows the operation distribution over all types of each member in TeamA while Fig. 4.5 shows these operations throughout the project-development process.

Fig. 4.4 seems to indicate that:

- Student LSName29 did much more work than the other members of TeamA, because he performed many operations in CVS (note: We use the male pronoun to refer to all students, irrespective of the gender of the actual student discussed). However, the number of his modification operations is not correspondingly high. A large part of the operations he performed were the addition and removal of files, and he was also responsible for many collisions.

- From this chart we might infer that LSName29 is the team leader who designs the project classes and initially authors the files for other members. Compared to the other members of TeamA,

- LSName42 exhibits a better (more material) operation pattern: high number of modi-

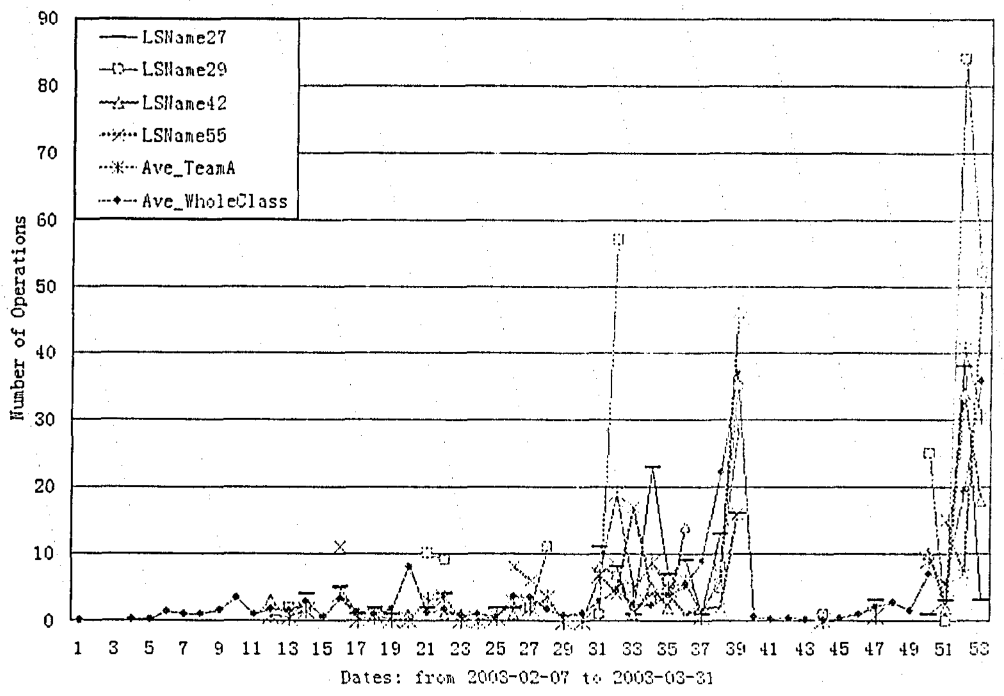


Figure 4.5: Temporal Distribution of CVS Activity, for Each Member in TeamA

fications, few collisions and high ratio of successful merges over collisions $-M\#/C\#$. Seems his work is independent and he might be the developer who focuses on a component.

- LSName27 had almost the least CVS operation records. He did very few modification, but lots checkouts. Based on these symptoms, we can say that this member did not have proper CVS usage.

We will support or reject these guesses with the other visualizations.

From Fig. 4.5 we notice the followings:

- There are large any-type-operation periods between Day 28 to Day 39 and from Day 48 to Day 53.
- The average team activity is similar to the average of all the teams and follows a similar pattern in time: most operations occur in phase 2 and at the end of phase 3: around the two major deliverables of the project.
- All four members of TeamA appear to have similar operation frequency and distribution except for LSName29, who was much more active around Day 31 ($Mar.10^{th}$)

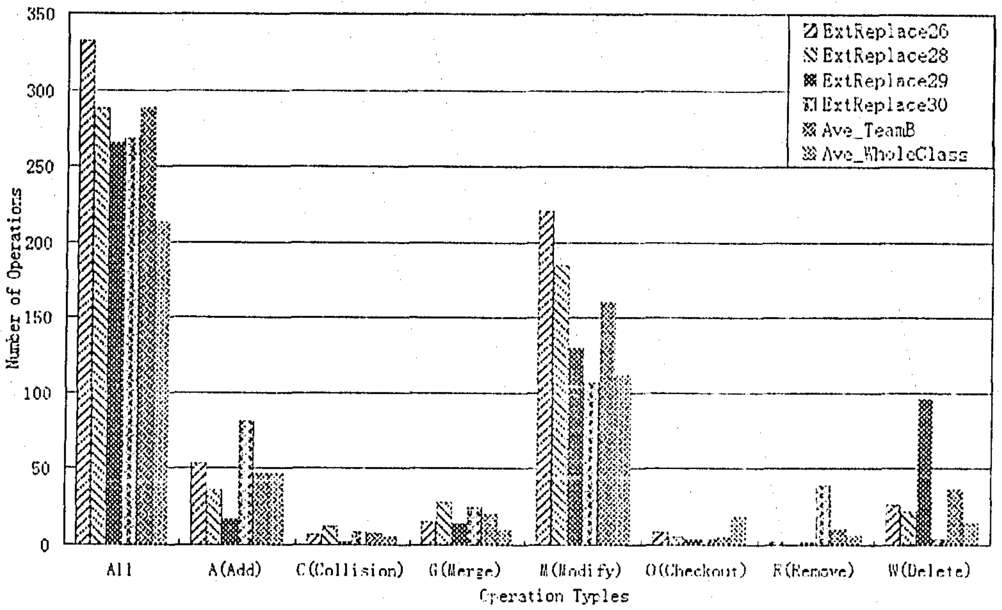


Figure 4.6: Distribution of CVS Operations by Type, for Members in TeamB

and Day 52 ($Mar.30^{th}$).

- This diagram provides counter-evidence for our earlier belief regarding the leadership role of LSName29, which proves that more accurate analysis results come from multilevel data. LSName29 did not start earlier than the other team members although he had several higher spikes comparing with those of his teammates, so he is not likely to be the designer/leader. At this point, we can simply assess his operation profile as "problematic": in spite of his big number of operations, it is not clear how he contributes to the team.

TeamB

Fig. 4.6 and Fig. 4.7 are two individual-developer aspect visualizations for TeamB. From Fig. 4.6, we notice the following facts:

- All the members in TeamB did more CVS operations above the average level of the whole class;
- No many differences among the operation amounts of members;
- Member ExtReplace26 had the most total CVS operations and M-typed operations. However, he was not the main file adder, and his numbers of collisions and merges

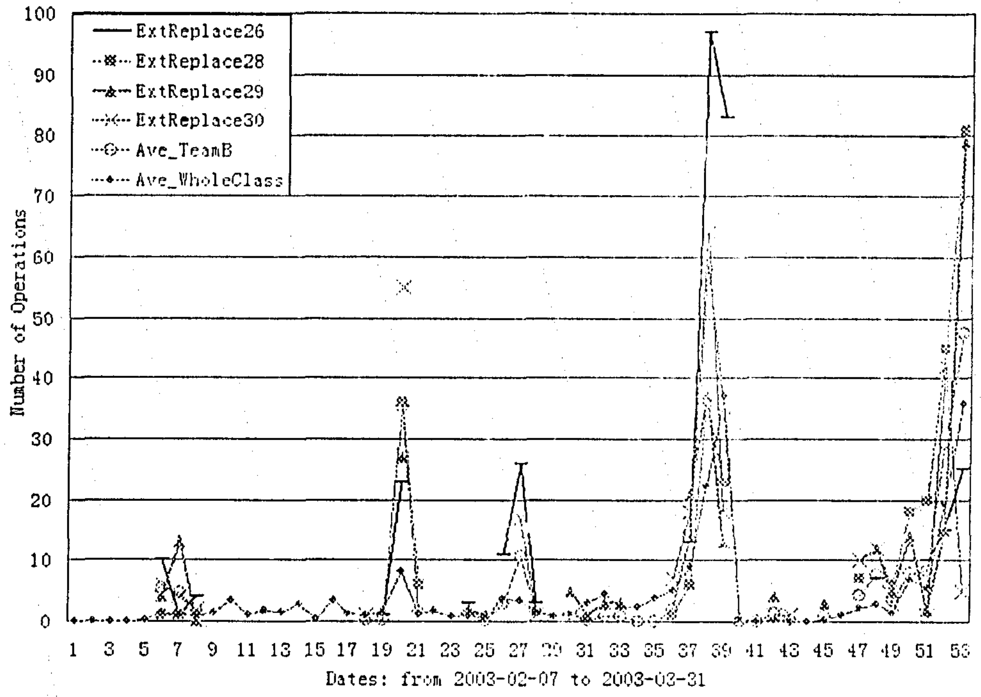


Figure 4.7: Temporal Distribution of CVS Activity, for Each Member in TeamB

are relatively small.

- Member ExtReplace30 almost had the least total operations in TeamB. He did the least modify operations, as much as 1/2 of the amount of ExtReplace26. However, he added the most files to the CVS repository, and almost all the file removing works were committed by him;
- Member ExtReplace28 had plenty of operations with a little bit high collisions and merges;
- Member ExtReplace29 had few operations with the least file adding, collision, and merge.

From Fig. 4.7, we noticed that:

- All the members had the similar curves and spikes; All of them started at the same dates and worked hard at the same small periods; In another word, TeamB has a uniform and regular work trend, and the typical work-at-the-last-minute habit;
- ExtReplace26 and ExtReplace30 had active operations at the earlier stages while ExtReplace28 and ExtReplace29 did more operations at the final stage.

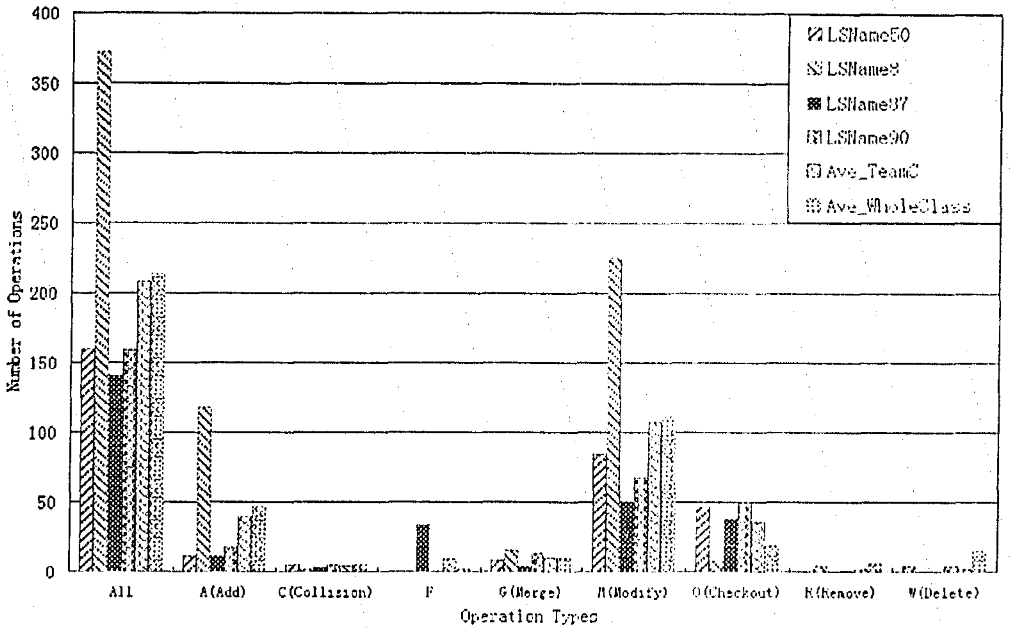


Figure 4.8: Distribution of CVS Operations by Type, for Members in TeamC

TeamC

Fig. 4.8 and Fig. 4.9 are two visualizations of TeamC in the same level:

- The average level of TeamC was similar to the average level of the whole class;
- Member LSname8 did more than double operations comparing with his teammates. Most modification operations were committed by him, and almost all the files were added into CVS repository by him. He did not involved too many collisions and had the least checkouts;
- All the other three members had similar CVS operation distribution;
- This team seldom removed files from CVS repository.

- All the members in TeamC are also typical work-at-the-last-minute developers. Not only those three members with few operations, but also the core developer: LSName8;
- They only had two spikes near the due days at the project part2 and part 3. In the first part, they did not leave many records in CVS;

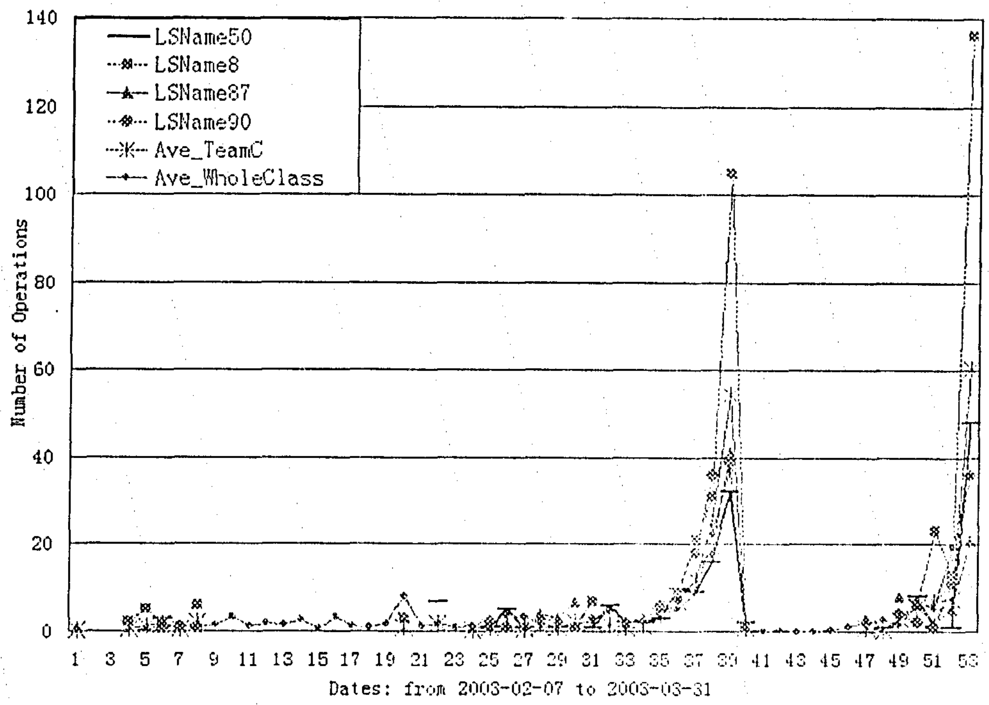


Figure 4.9: Temporal Distribution of CVS Activity, for Each Member in TeamC

TeamD

From Fig. 4.10, we can notice that:

- The average CVS operations of TeamD were larger than the average of the entire class level;
- In this team, there was also a core member: LSName58. He had the most CVS operations, added more than half files into CVS repository, and did almost double amount of the modifications;
- Member ExtReplace27 had the least operation records in TeamD, but removed relatively a large number of files and checked out frequently;

Fig. 4.11 shows that the work habit of TeamD is much better than the previous three teams:

- All the members started earlier than the other groups and worked consistently, almost every day.

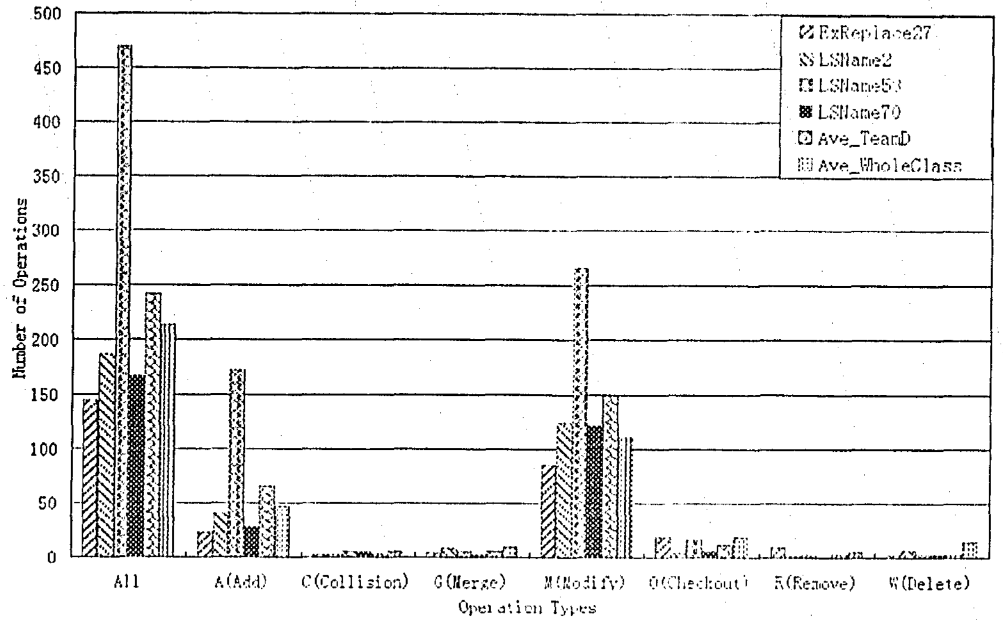


Figure 4.10: Distribution of CVS Operations by Type, for Members in TeamD

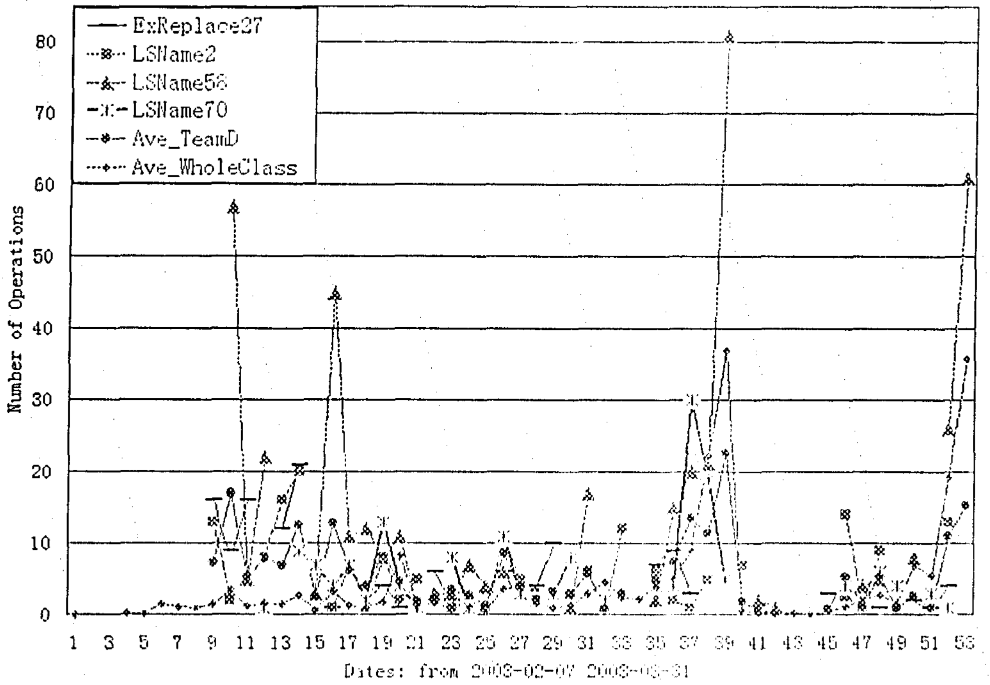


Figure 4.11: Temporal Distribution of CVS Activity, for Each Member in TeamD

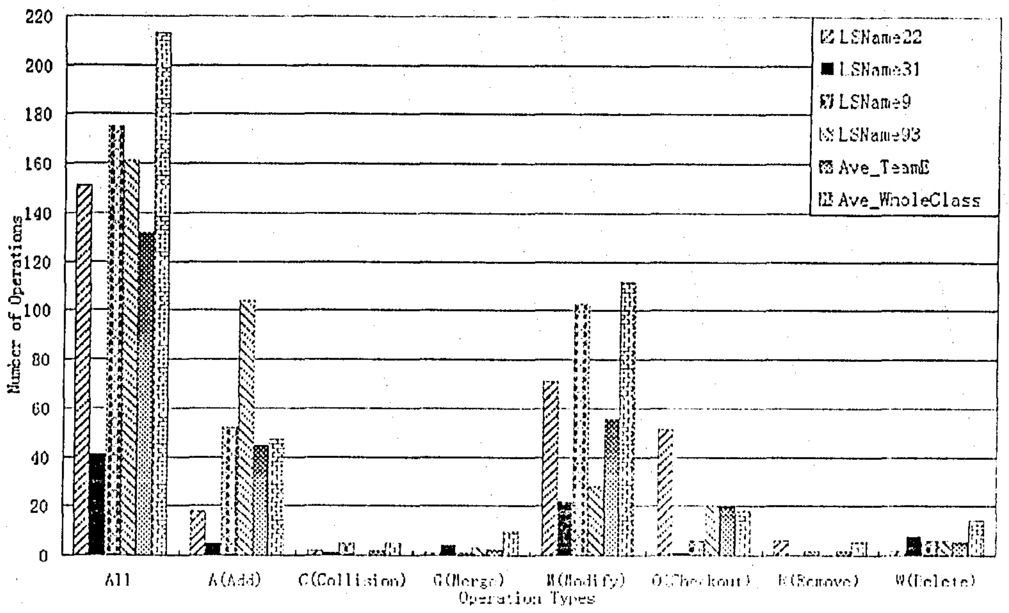


Figure 4.12: Distribution of CVS Operations by Type, for Members in TeamE

- Member LSName58 always had much more operations around the due days although he had persistent works since the very beginning. We conjecture that he acted as the team leader, and always did some mop-ups at the final moments before the deadlines;

TeamE

- The average CVS operation records of TeamE is much smaller than the class average level;
- TeamE added similar number of file into CVS repository as most teams, and did nearly the same amount of checkouts. However, their modifications (*Mod#*) are much less; The reasons can be two possibilities: (a) members in TeamE had normal developments but only treated CVS as a storage tool more than a sharable working platform, (b) members in TeamE did not have enough works;
- Almost half files were added into CVS by member LSName93, but he only did very few modifications on these files;
- Member LSName9 had the most CVS operations with the most modifications in his team;
- Member LSName22 did lots of checkouts and did the most removing operations;

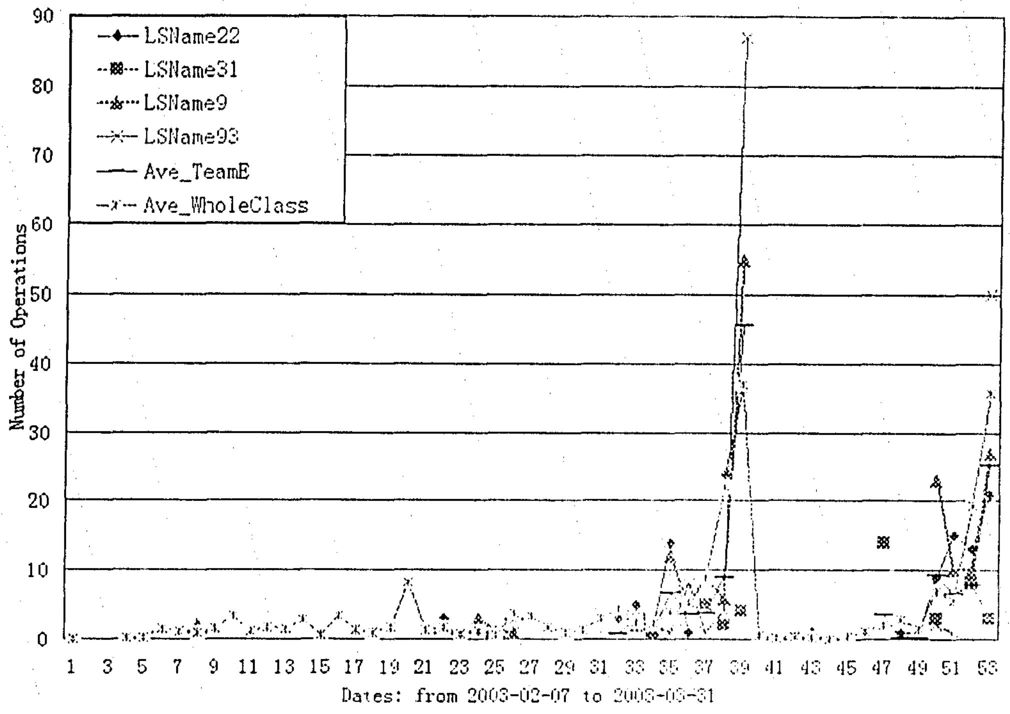


Figure 4.13: Temporal Distribution of CVS Activity, for Each Member in TeamE

- Member LSName31 had very few CVS tracks; Over the almost two-month project development process, he only modified around 20 file versions.
- The total CVS operation number of member LSName93 was not small. Most of the operations were adding new files: he added more than 100 files into CVS repository while only modified less than 30 file revisions;
- Compared with other teams, TeamE had a pretty idle work curve: they almost had flat lines before day 33.
- Although some of them had spikes before the second and third deadlines, the spikes are still very short and small;
- Member LSname9 and LSName22 started a little bit earlier than the other two members;

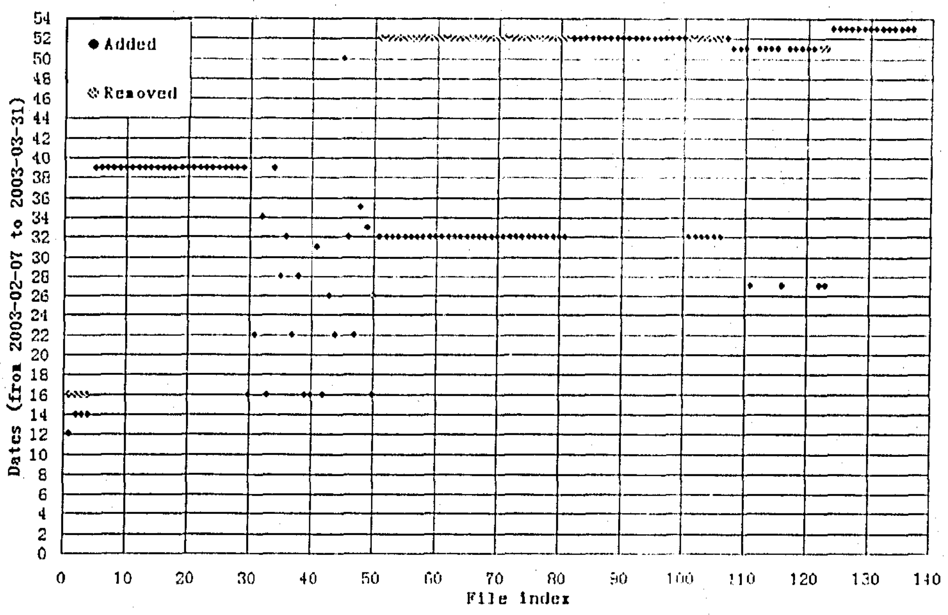


Figure 4.14: File Adding and Removing by Date, for TeamA

4.3.3 The File aspect

Let us now examine how the project workload was distributed across the files. CVSChecker produces three visualizations for each team to show the file-related information.

Fig. 4.14 shows the file additions and removals of TeamA. The X-axis lists all the files in CVS repository.

- There were 137 files in CVS repository at the end of the project; Only three file (file 1, 2, 3, 4) was added before the due day of project part1. They are four Java classes and all of them were added to the team root directory.
- On day 16, the next day of the deadline of the project part1, All the four above classes were deleted and re-located to the subdirectory "src/" (file 30, 33, 39, 40);
- Another new class (file50) and "makefile" (file42) were also added to this place on that day;
- Some other main classes and configuration files (from file30 to file50)were added into "src/" directory in succession before the project part2 deadline (day 39);
- On day 32, 31 files were added to CVS (file51 to file81). All these files were online manual files required by project, and all of them existed on directory "src/helpset".

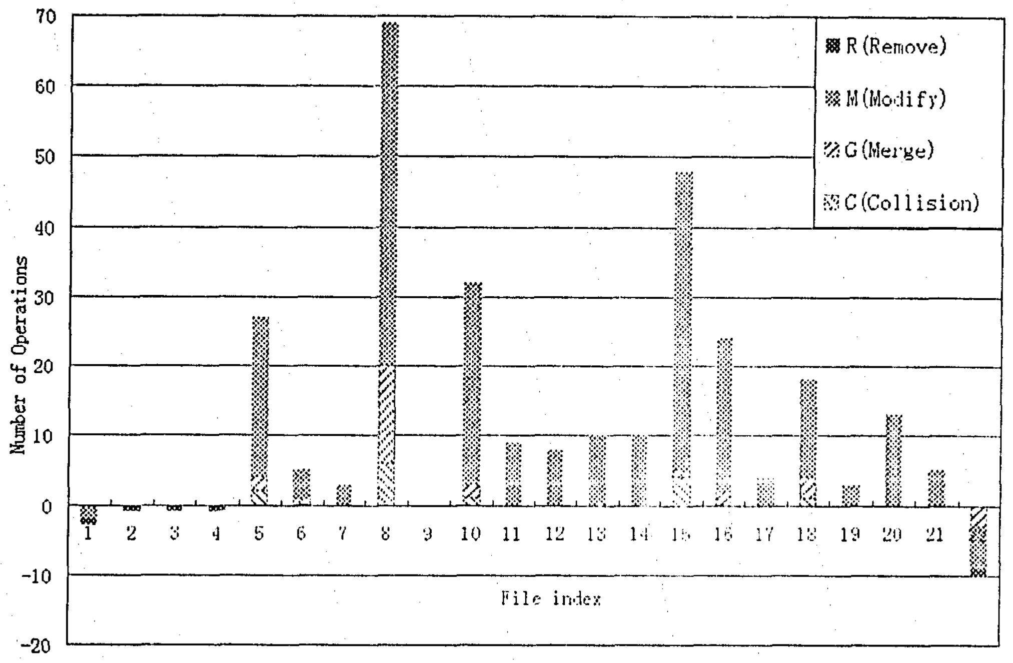


Figure 4.15: Distribution of CVS Operations by Type, for Each Java Class in TeamA

As we noticed, these files were removed again on day 52: all the html files were relocated (file82 to file90, and file107), and all the "html " files were deleted.

- Six files (file101 to 106) were also added to module "src/helpset/JavaHelpSearch/" on day32, and removed on day 52;
- All the files between file108 and file123 were ".gif" files. They were saved in directory "src/img";
- On day 53, 14 files (file124 to file137) were added to "test/" directory. All of them are ".txt" files and they were created to test the project quality before the final deadline arrived.

The following analyses are focusing on the Java files of TeamA (between file1 to file4, and file30 to file50 in Fig. 4.14).

Fig. 4.15 shows different types of operations performed on each Java class while Figure 38 shows the detailed modification works of each member on them;

- TeamA added 22 Java classes to the repository;

- Files 1, 2, 3, 4, and 22 (they are file1, 2, 3, 4, and 50 in Fig. 38 correspondingly) have already been removed at the date this visualization was generated;

- From the height of columns, we can have a quick idea about which files suffered many operations: Files 8, 15, 10, 5, 16, 18, 22, and so on (in that order).

- They are also those files that suffered many M (Modify) operations. Moreover, almost all the collisions and merges happened on these files.

- Their numbers of modifications, Mod#, differ but their sizes can be comparable. Consider for example file 10 and 17: file 10 has had many more modifications than file 17; however, the eventual sizes are almost the same;

With this chart, members of TeamA and instructors can easily detect those files that have problems in the design and prone to incur collaboration problems.

In addition to provide an overview of the number of modifications, Mod#, Fig. 4.16 enables a deeper view into this information, presenting the numbers of LOC added to and deleted from each file by each team member.

Combining the information from the above figures we notice that the files with the highest density of modifications, i.e., high ratio of modified lines per total lines of code, such as files 8, 15, 5, 16 and 18 were touched by multiple team members.

It is not always the case that a file modified many times is also modified substantially. For example: the total number of modified lines of file 17 is much less than that of file 10 although their numbers of modification operations – Mod# – are almost same;

Member LSName27 only modified 4 files: File 7, File 8, File 15, and File 16, and these files were also modified by other members. Moreover, his modified LOC number was small;

- Both members LSName29 and LSName55 modified almost half of the files. However, most files were also modified by other members, with high collision and merge numbers; LSName29 was the author of those 4 early-added-and-removed classes (file1, 2, 3, 4); He was the only members who did some real coding works on project part1;

- Member LSName42 modified 5 files: File5, 10, 11, 13, 18. Although File5 and File18 also modified by other members, LSName42 was the core developers of them. Moreover, LSName42 did not change any line on those two most important files:

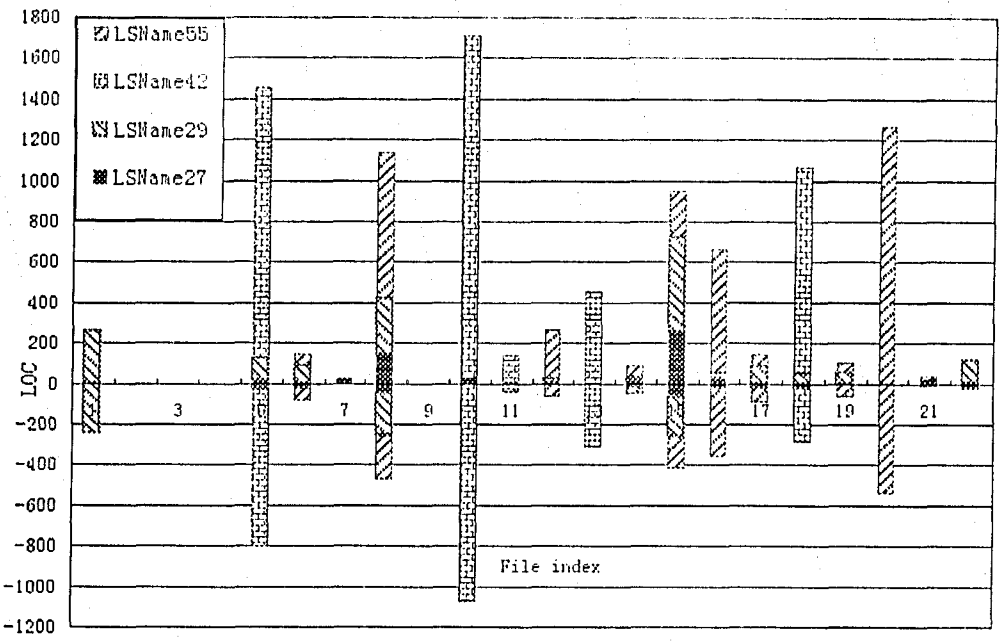


Figure 4.16: Added and Deleted LOC by Each Member, on Each Java Class in TeamA

File8 and File15. In a word, LSName42 spent his work on several classes and he was the only or main developer on them. We can say that LSName42 focused on some independent components;

Combining the information of Fig. 4.15 and Fig. 4.16, we can notice that almost all the files with high collisions and merges were modified by multiple developers, such as File8, File15, File5, File18, and so on. We can say that the design of these files have to be improved.

TeamB

- There were totally178 files in the CVS repository belonging to TeamB;
- Before the deadline of the project part 1, some files have been added to CVS repository since day 6, such as file2, file6, from file123 to file 142. All of them were not Java classes and were removed at the same day or at day 20;
- On day 20, 5 gif files and 1 Java class were added to CVS root directory (file1, 4, 5, 7, 8 and file 3). Later in the same day, these files were deleted from the original

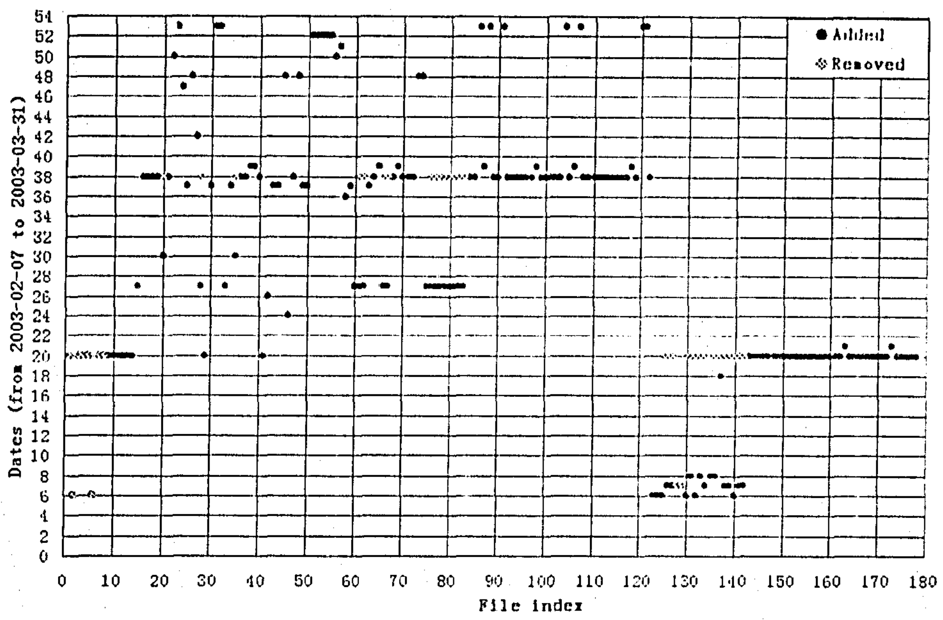


Figure 4.17: File Adding and Removing by Date, for TeamB

locations and removed to a new directory "LFullName108/". Six new files appeared (see file 9 to 14);

- Also on day 20, those files added on project part1 were removed from their original directory "docs/" to "docs/part1/". (see the removing of files: file 126 to 142, and new added files: file143 to 157);
- Some new Java classes and other project files (most of them are gif files) were also added on day 20;
- In project part 2, more Java classes were added, together with some other files;
- A group of files were added on day 27 and removed on day 38 (just before the due day of project part 2). All of them are gif files (file 28, 61, 62, 66, 67, 76 to 83);
- Two renamings happened on day 38 (renamed file 20 to file 21, and file 35 to file 36). Both of them were created on day 30 originally;
- Files between 84 and 121 were JavaDOC files. Almost all of them were added into CVS on day 38 while some of them were added on day 39 and 53.

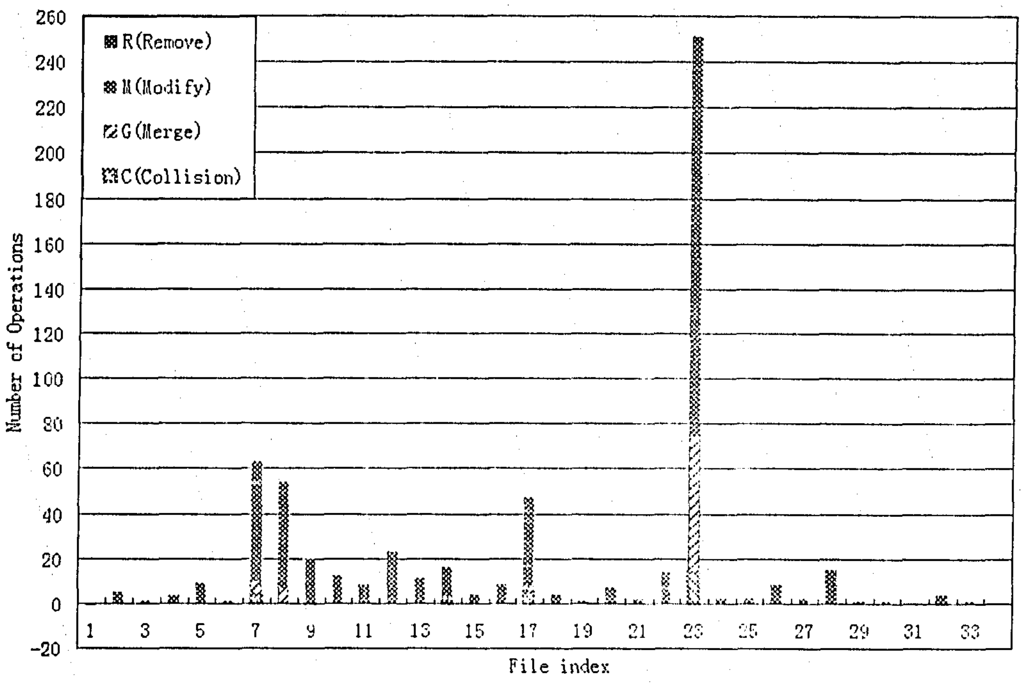


Figure 4.18: Distribution of CVS Operations by Type, for Each Java Class in TeamB

- TeamB had 34 Java classes in CVS repository. All of them scattered before file 75 or after file158 in Figure 4.17;
- Since they created their project modules, they only removed one file from CVS: (file 1 in Figure4.18 and file 3 in Figure4.17);
- Almost half of the modifications were committed on File23;
- In addition to File 23, Files 7, 8, 17, 12, 9, 14 etc. were other relatively important files. They were modified many times and had collisions and merges;
- There were also some small classes with very few modifications;

Based on above observations, we can say that the structure design of TeamB was not well-proportioned. This was an important problem to be improved. Fig. 4.1 gives us supportive facts: TeamB has the largest collision and merge numbers.

From Fig. 4.19 we can notice that:

- Most of the above listed high-operation files were modified by multiple members;

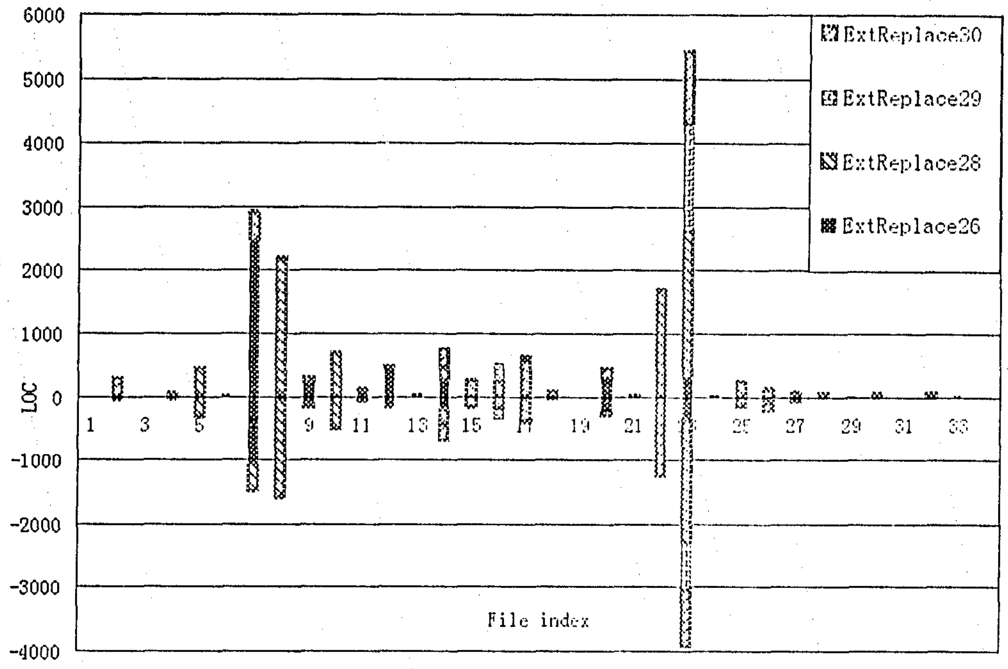


Figure 4.19: Added and Deleted LOC by Each Member, on Each Java Class in TeamB

- Some files were modified heavily (added more then 2000 lines and removed more than 1000 lines, the largest file was changed almost 10000 lines) while some others were almost no touched;
- There were also cases that a file modified many times was not modified substantially (File17), and a file seldom modified was changed substantially (File6, 22);
- Member ExtReplace26 modified 7 files. Most of them were cooperative work products shared with other members;
- Most single-author files in TeamB were maintained by ExtReplace28. Besides, he also joined the developments on some other files with his teammates. Figure4.6 lets us know that ExtReplace28 had some collisions and merges. Thus, we can say that all his collisions and merges came from these cooperative file: File7, 14, 16, 17, 23, 26.
- ExtReplace29 and ExtReplace30 did few modifications, and they always joined the files with other members instead of having own independent files;

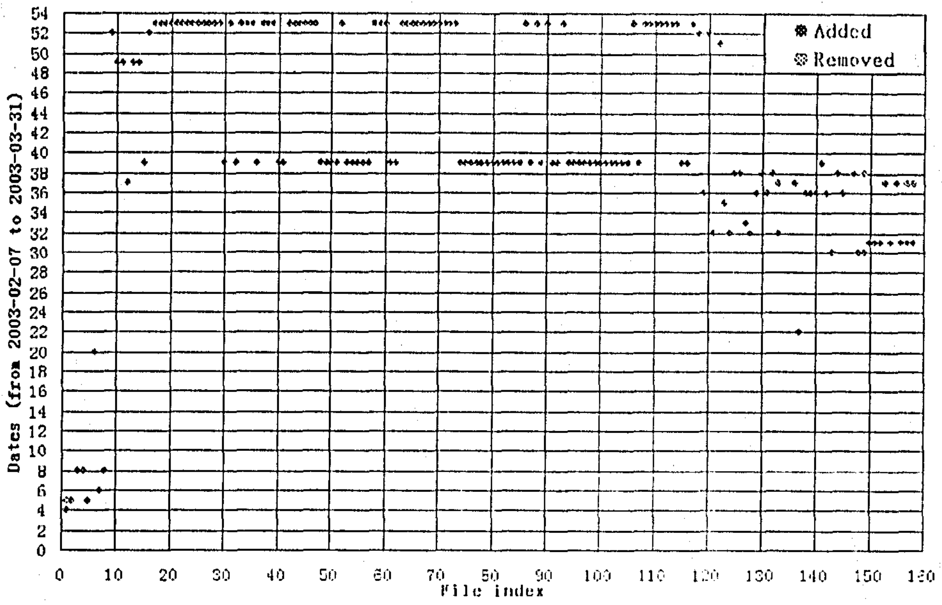


Figure 4.20: File Adding and Removing by Date, for TeamC

TeamC

- TeamC had 158 files totally;
- There were only 7 files that were added into CVS in project part1 (File 1, 2, 3, 4, 5, 7, 8), and in these files only File 3 was a Java class;
- Most image files required by the project were added into CVS on day 53 while some sporadic ones were added on day 37, 39 and 49. All these image files were listed in Figure 4.20, from file 9 to 29 (except three Java classes: file15, 16, and 17);
- All the JavaDOC files (file 30 to 117) have the same situation as those image files.
- All the files with index larger than 117 were Java classes, except a manual doc file (file 146) and a help html (file 138).
- TeamC only had one renaming work: they changed the name of file 133 from "Todo.java" to "ToDo.java" (file 130).
- Only 4 files were removed, and 3 of them were Java classes;

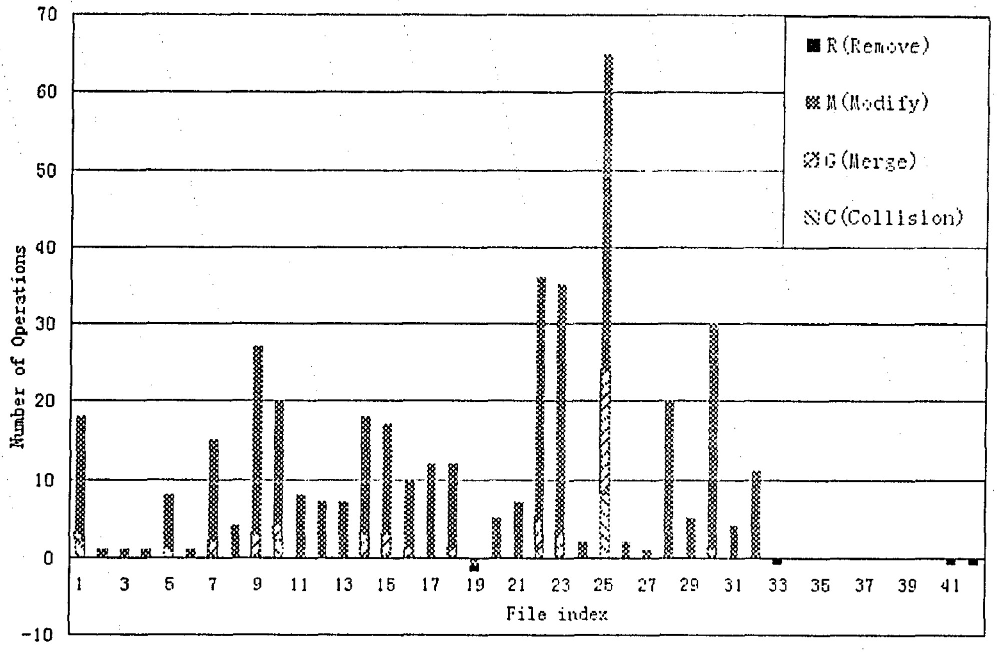


Figure 4.21: Distribution of CVS Operations by Type, for Each Java Class in TeamC

- TeamC had 42 Java classes, and 4 of them were removed from CVS. Actually, one of the removing was the result of a renaming;
- The CVS operations spread on these Java classes much more balanced (located on file15, 16, 17, and 118 to 158 in Figure44), comparing with that in TeamB;
- The following files had many CVS operations with collisions and merges: file 25, 22, 23, 8, 10, 1, 14, 15, 7, 18, 16 and so on;
- Some files had not been modified since they were added into CVS, such as files from 34 to 40. All these files include three removed ones constituted a module "source/wmvc/";

From Fig. 4.22, we have the following observations:

The modification works on most files do not have so many differences as that in TeamB;

Again, those files with high collisions and merges were modified by multiple members;

Some files with a large number of modifications were only maintained by a single member, such as file 28 and file 17;

LSName50 modified 13 files, but none of them was only maintained by him;

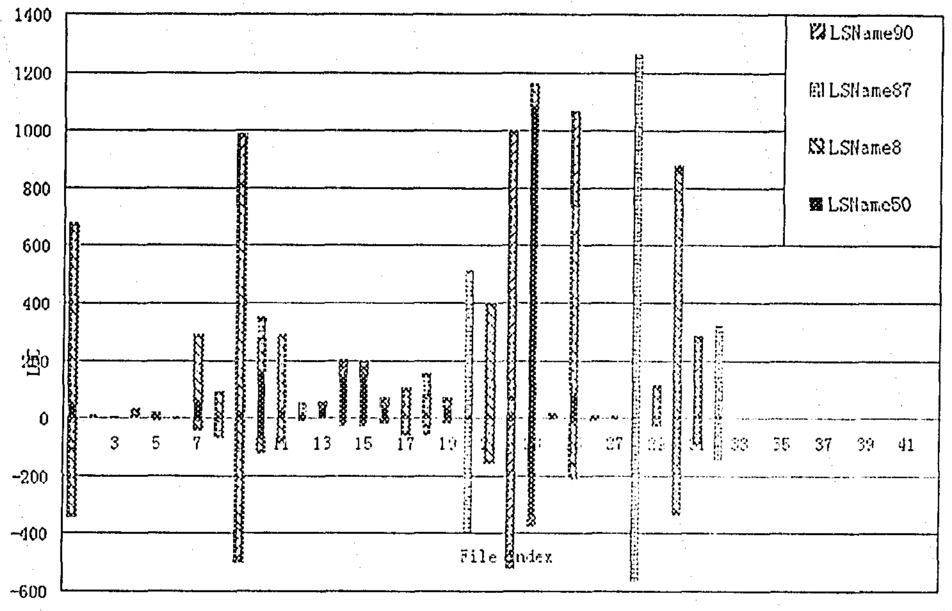


Figure 4.22: Added and Deleted LOC by Each Member, on Each Java Class in TeamC

LSName8 modified 19 files. He contributed a tremendous amount of work on most of the 19 files. Moreover, he was the only developer for some of them;

LSName87 was a very independent developer. Almost all of his modification works were committed on three classes (File20, File28 and File32), no other members touched them;

LSname90 modified 8 files. Two of them were only developed by LSname90.

TeamD

- The number of files in TeamD was exceptionally large: 263 files in total;
- TeamD started the project development earlier than all other teams and have a large proportion of working days;
- Although TeamD has a comparatively better habit, the project had an abnormally high number of Java classes. After inspecting the file report, two reasons were found:
  - All the members moved their individual assignment work into the common project directory (files from 1 to 58), and
  - Member LSName58 dumped another 37 Java files (files from 62 to 98) into a directory named "demo/newLayout" on the due day of project part 2 just for

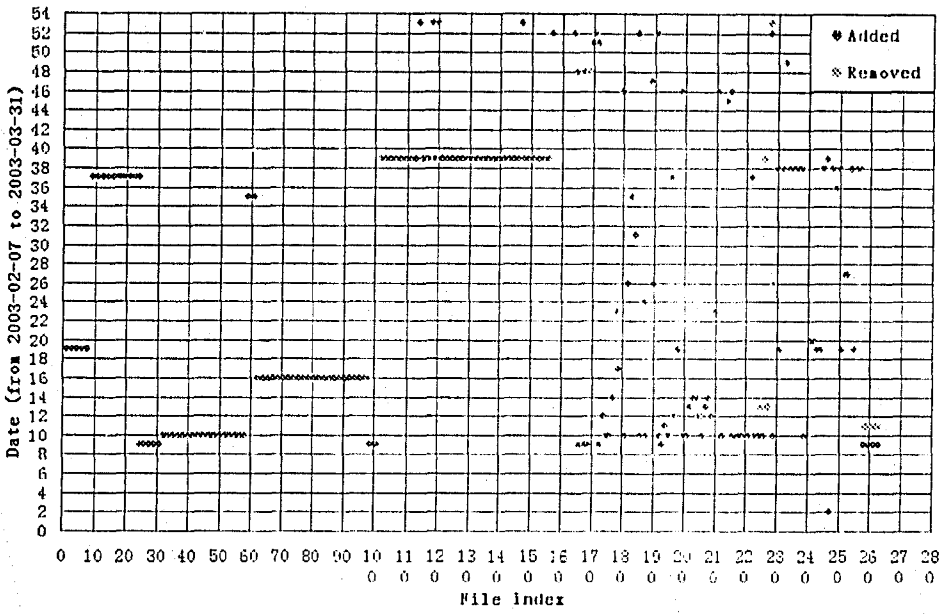


Figure 4.23: File Adding and Removing by Date, for TeamD

the demo, and never touched them any more. All these files were actually the copies of those Java files finished until Day 39.

- Files 102 to 156 were JavaDOC files. They were added into CVS on Day39 or Day53. A group of UML-related files (files 258 to 263) were added into module "UML-diagrams/" very early (on Day 9) while another group of the same-named files were also added in a different module -"docs/UML/", on the same day. Later, the first group was removed on Day 11;
- Files 157 to 172 were also files that related with UML diagrams. They were added into directory "docs/UML/" on the following days: days 9, 48, 51 or 52;
- The removing of file 169 was the result of a renaming on Day 48: its name was renamed from "iCal-View.dia" to "iCal-Views.dia" (file 167);
- Most Java classes were located between file 173 to 229. Some of them were added pretty early (in project part 1);
- All the image files were located between File 230 to 257 in Figure 4.23. Almost all of them were added in project part2, especially just before the deadline.

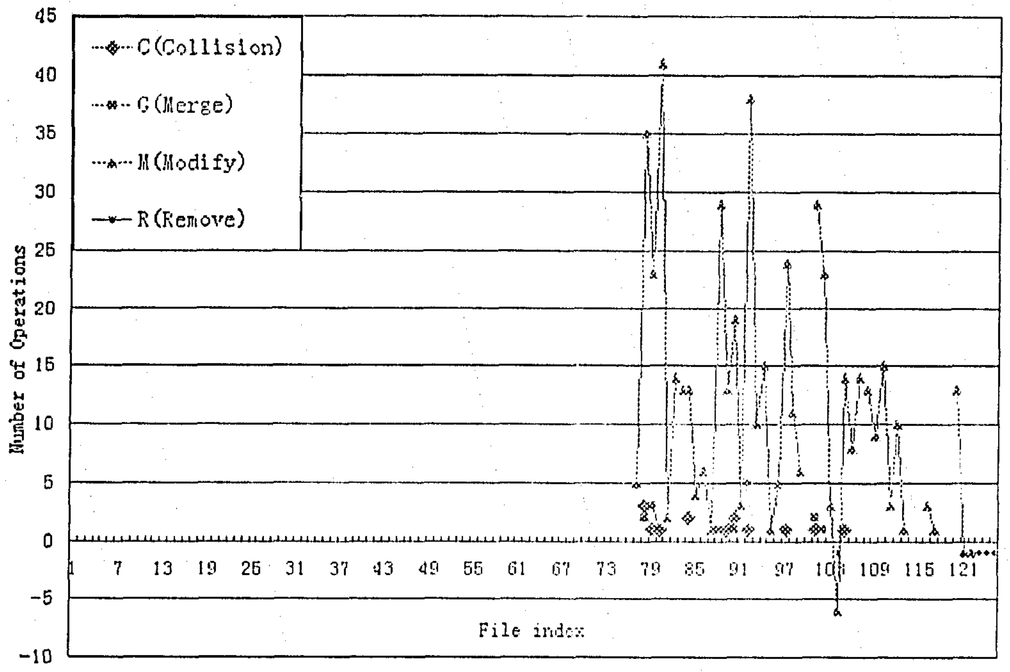


Figure 4.24: Distribution of CVS Operations by Type, for Each Java Class in TeamD

Fig. 4.24 shows us the following facts:

- TeamD added 126 classes to its CVS repository;
- More than half of them were only added without any further modification. We have already stated the reasons above;
- 38 classes were modified in the project development process, and 6 of them were removed from CVS repository;
- More than 10 files had collisions and merges: File78, 79, 80, 84, 88, 89, 90, 92, 97, 101, 102, 104, 105 etc. This shows us that a team with a better work schedule can also have collaboration and design problems.
- Most high-collision-and-merge files were modified by multiple members, such as File78, 79, 88, 90, 92, 101, 102, 104;
- ExtReplace27 did not have his own classes. All the modifications he committed were on files that co-developed with other members;

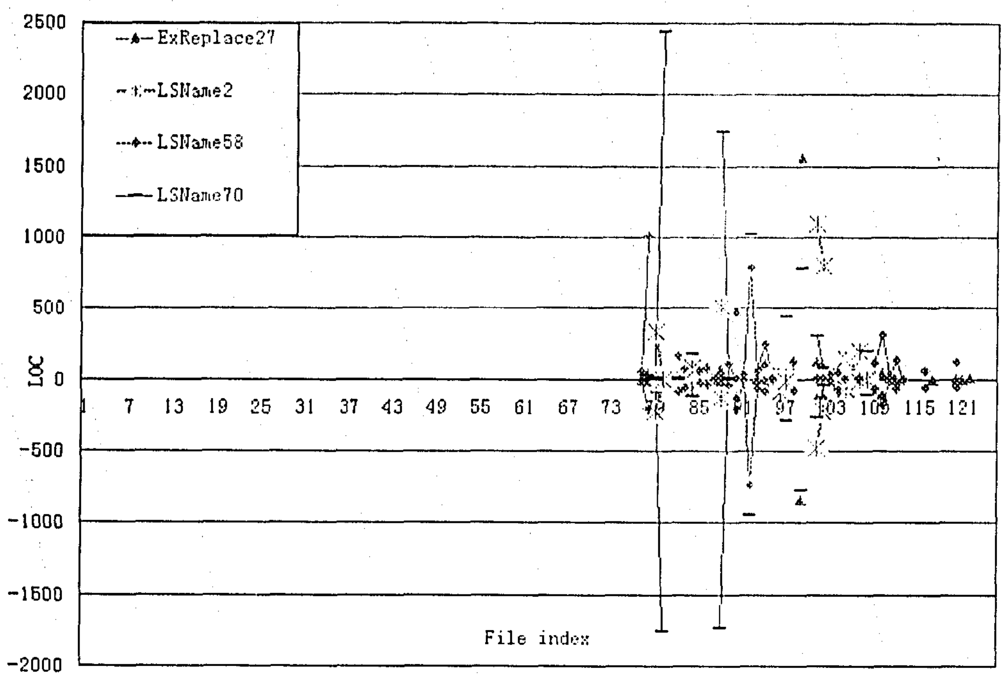


Figure 4.25: Added and Deleted LOC by Each Member, on Each Java Class in TeamD

- LSName2 had 6 shared files and 3 independent files;
- LSName58 had many independent files with several shared files; Figure 4.10 tells us that he was the member who had the most CVS operations in TeamD. He also added the most files and had the most modification works;
- LSName70 changed a great number of lines on several shared files.

TeamE

- TeamE created 178 files totally; The project size is normal compared with most other teams;
- All the members did not add any file into CVS before day 22;
- Most files were added at or just before the deadline of project part 2: Day 39. Same things happened at the project part 3;
- With the inspection of the java information, we found that TeamE did not have reasonable design of the project structure: all the Java classes and most other files were dumped under the root directory near the due days, and no any package concepts;

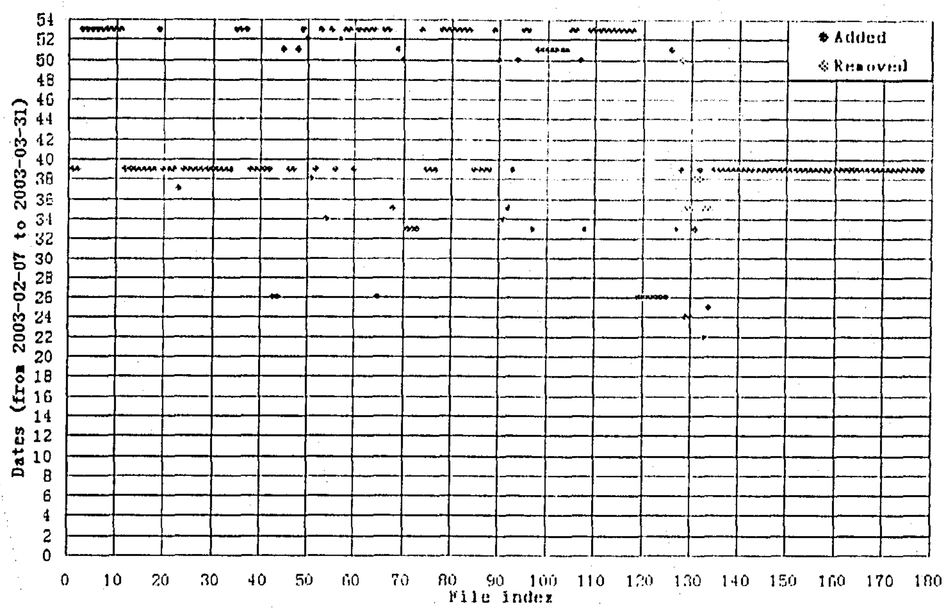


Figure 4.26: File Adding and Removing by Date, for TeamE

- All the JavaDOC files (files 135 to 178) were put in a directory named "html/" on Day 39 (due day of project part 2), instead of the popular module name such as "docs/Javadoc/";

- 7 files were removed (files 128 to 134). They were some temporary existed files for testing.

Fig. 4.27 and Fig. 4.28 show us the following observations:

- TeamE had 51 Java classes in CVS repository totally, and 3 of them were removed;

- The numbers of CVS operations on the files are smaller than the first four teams;

- The following files had collisions and merges: File1, 7, 8, 14, 22, 32, 33, 36, 37, 38, 48; Also, they had many modifications;

- There are some files that were never been modified after they were added into CVS.

- Almost all the high-collision-and-merge files were modified by multiple members;

- Member LSName22 modified 9 classes. Only two of them (file36 and 38) were developed and maintained by himself;

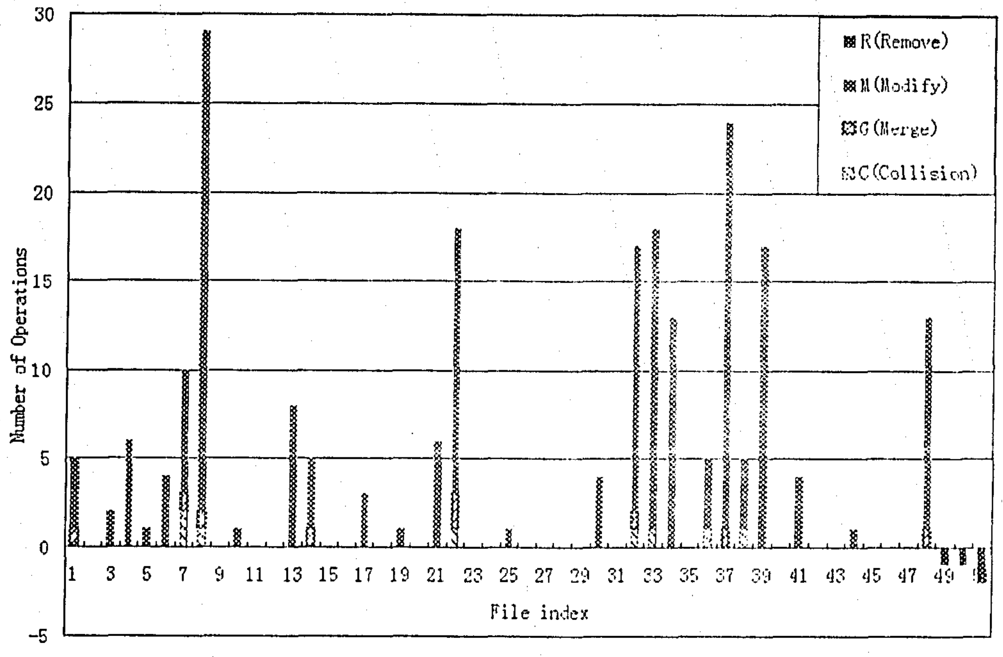


Figure 4.27: Distribution of CVS Operations by Type, for Each Java Class in TeamE

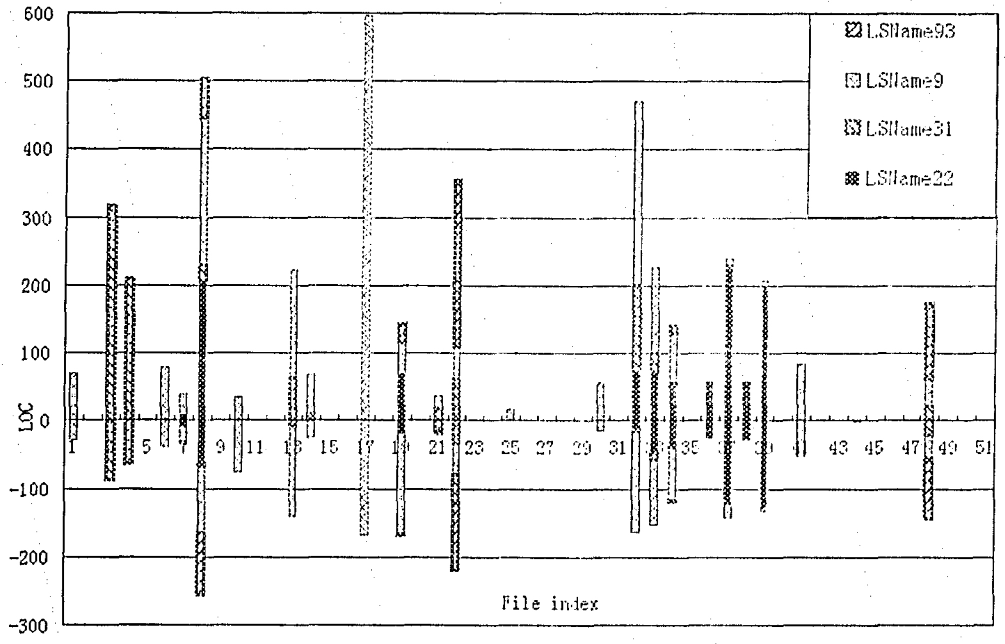


Figure 4.28: Added and Deleted LOC by Each Member, on Each Java Class in TeamE

- Although member LSName31 had very few CVS operations (see Figure 36), his modifications spread on 10 classes, and he had 2 independent classes with heavily changed LOCs;
- Member LSName9 had the most CVS operations. He touched almost all the classes (17 classes) and also had his own classes;
- Although most files were added by member LSName93, he only contributed his modification on 5 shared classes.

4.3.4 The File-Version aspect

In above section, we detected several files that may have design problems: multiple members modified them together with high collision and merge numbers. In this part, we display the detailed LOC changes by date of each of them.

In TeamA, three visualizations in the File level (figure 4.14, 4.15, and 4.16) helped us to detect that File8 and File15 were two major files that may had design problems: three members joined the developments, and some collisions and merges happened on them.

Fig. 4.29 displays the detailed LOC changes by date of File8 with whole name as "TeamA/src/CalendarFrame.java" and Figure52 show the same information of File15, whose whole name was "TeamA/src/ICalController.java".

- File "TeamA/src/CalendarFrame.java" had 50 revisions totally;
- This file was added into CVS repository on Day 16 by LSName55.
- LSName55 also finished the first several revisions on Day 26;
- Most co-developments happened before project part deadlines, such as on Day34, 38, 39, 52, 53. Of course, they were the days that collisions and merges happened on this file;
- Member LSName42 did not touch this file;

- File"TeamA/src/ICalController.java" had 44 revisions and was also added by LSName55 on Day 16;
- LSName29 finished several first revisions on Day21, 22, and 28;
- Member LSName42 seldom worked on it;

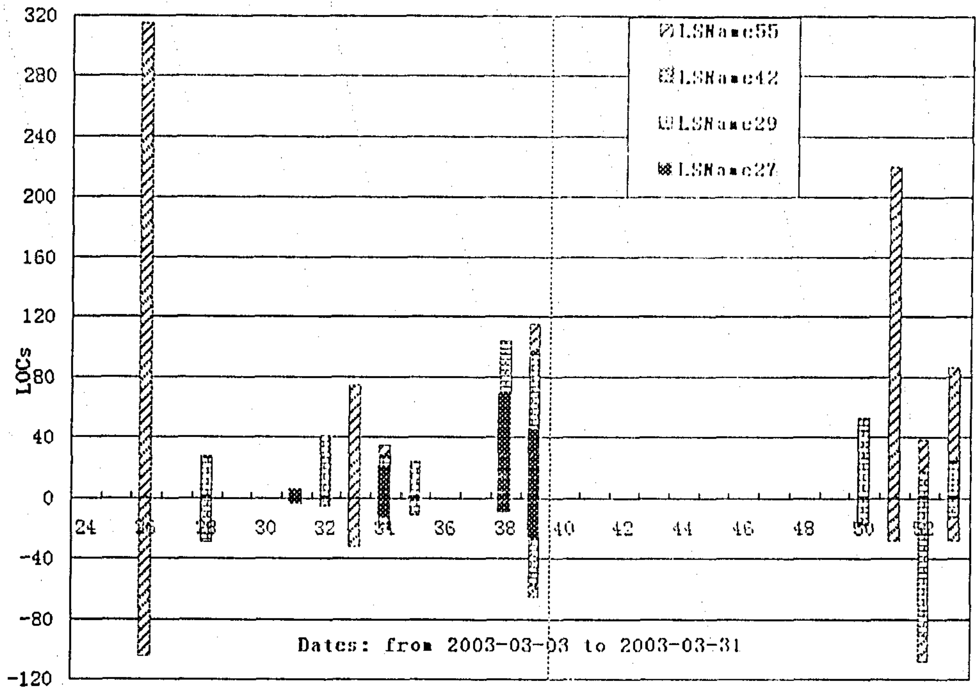


Figure 4.29: Detailed LOC Change by Date, on File "TeamA/src/CalendarFrame.java"

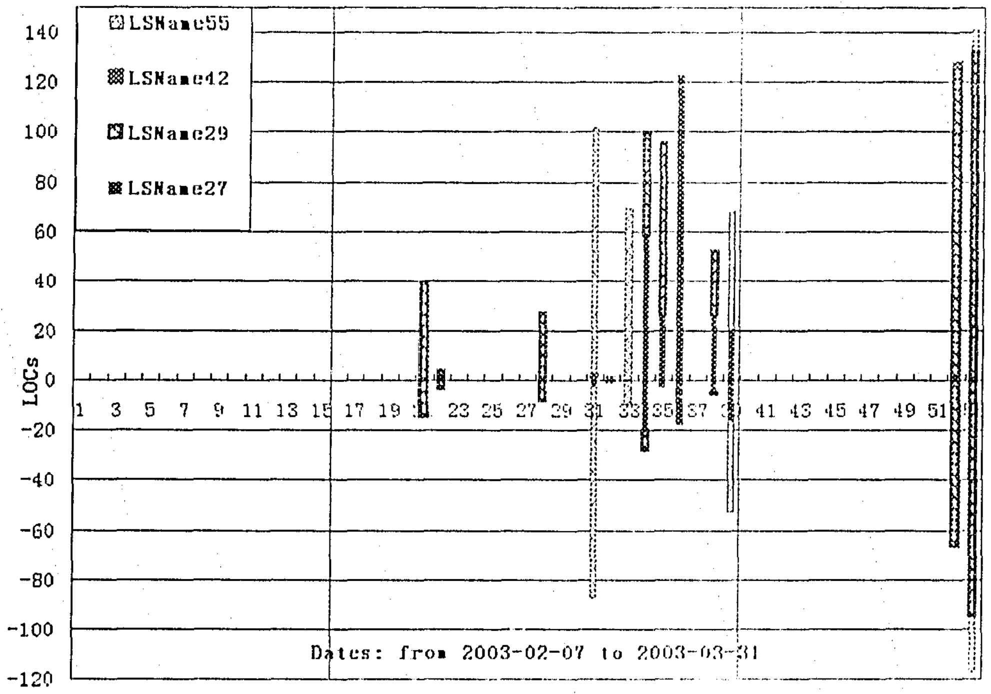


Figure 4.30: Detailed LOC Change by Date, on File "TeamA/src/ICalController.java"

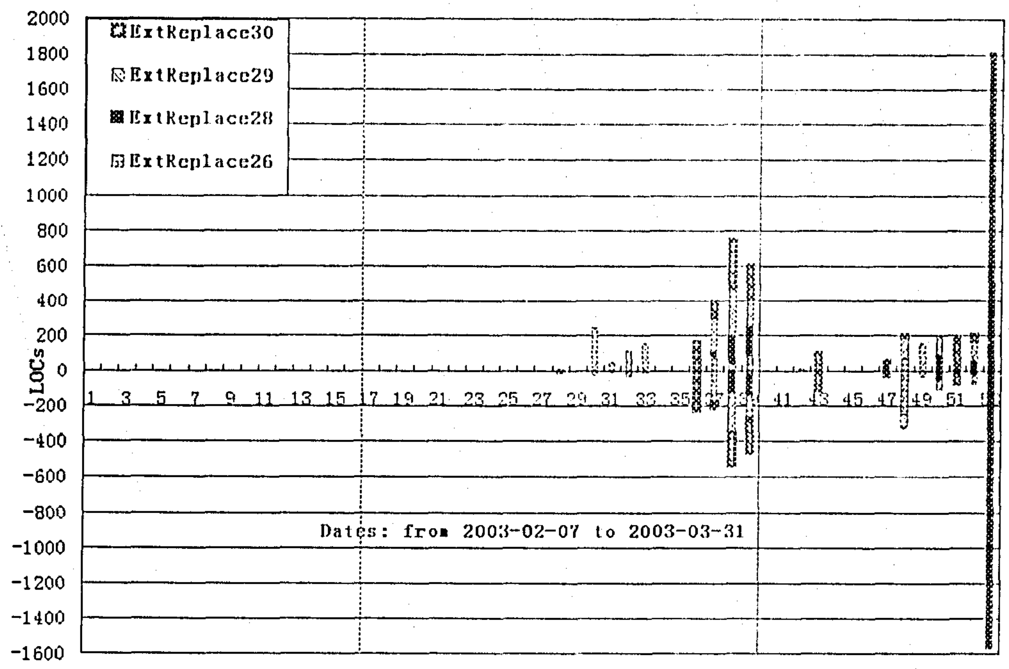


Figure 4.31: Detailed LOC Change by Date, on File "TeamB/code/RCal.java"

- Member LSName29 was the core developer for this file on project part 3;
- Collisions and merges occurred on days 31, 34, 35, 38, 39, 52 and 54.

In TeamB, File 23 was the main class in the project that experienced the most CVS operations and modified LOCs (see Fig. 4.18 and Fig. 4.19). Its whole name is "TeamBcodeRCal.java";

- File23 had 177 revision in total; This number is larger than most other classes;
- File23 was created by ExtReplace30 on Day 27, then ExtTeplace29 took over it and finished 8 revisions on 4 days;
- On the last three days of project part 2, at least three members worked on it, which was a dangerous sign for the class design; Same things happened near the end of the project part3 deadline;
- On the final day of the whole project, ExtReplace28 was the core developer on this file.

Fig. 4.21 and Fig. 4.22 tell us that File25 was modified the most frequently in TeamC. Although its substantial modifications were not the highest, it had the most collisions and

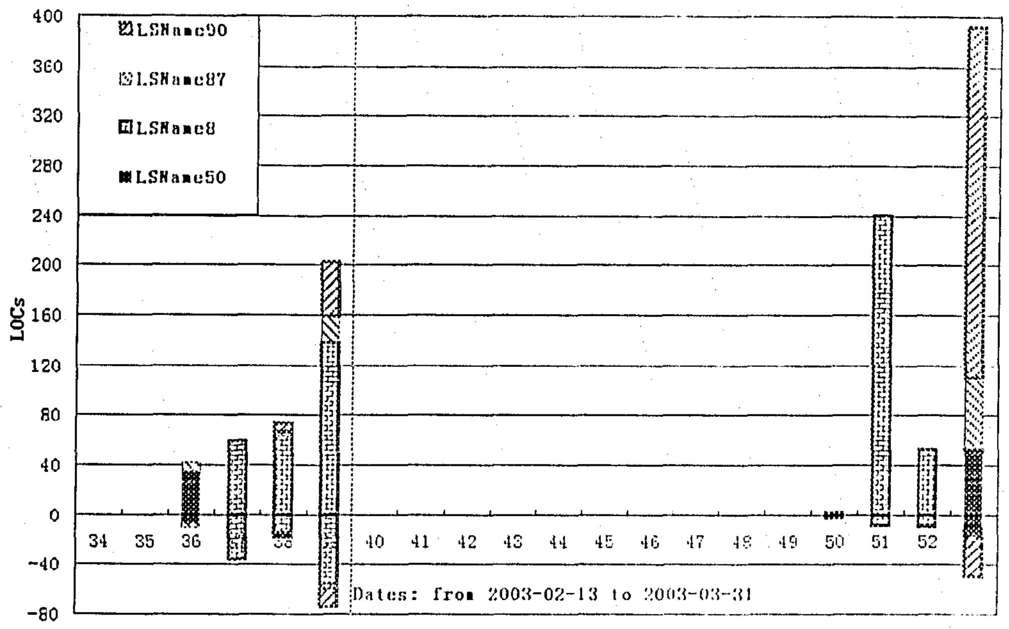


Figure 4.32: Detailed LOC Change by Date, on File "TeamC/source/views/iGorApp.java"

merges, and all the four members in TeamC touched it; Fig. 4.32 shows us a more detailed history of File25 - "TeamC/source/views/iGorApp.java".

- File "TeamC/source/views/iGorApp.java" had 44 revisions;
- Member LSName50 added it to CVS on Day36 and created the original four revisions with member LSName87 on the same day;
- In both main project parts, LSName8 always worked on it earlier. As the deadlines approached, other members joined the development in succession;

Fig. 4.33 shows File 8 ("TeamE/CalendarModel.java") in TeamE.

- File8 had 28 revisions in total; It was added by LSName9 on Day26;
- On project part 2, LSName9 was the main developers on this file at early days.
- At the due day, LSName22 and LSName93 also contributed some modification on it;
- Different from other teams, TeamE had fewer modifications on project part 3; Member LSName31 only modified this file on Day 47.

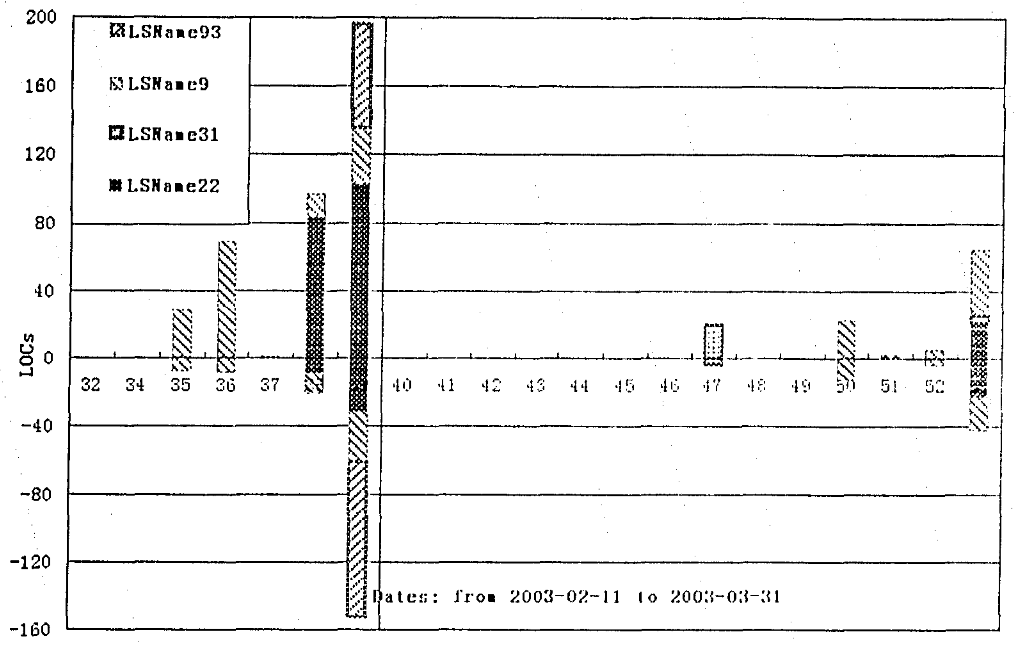


Figure 4.33: Detailed LOC Change by Date, on File "TeamE/CalendarModel.java"

4.4 Heuristic Generation and Knowledge Extraction

In this case study, we analyzed the data of 20 students in five teams. We selected all the directly collectable attributes and generated derived attributes listed in Chapter 3.3.2 from 4 different levels. We transformed all these parameters as ARFF [31] format. Apriori algorithm of Weka [36] system is adopted to discover interesting patterns about how team members use and modify their software assets and to extract the high-level knowledge hided along the project development process from the original data.

Some results were mined from our dataset. The following are rules discovered with minimum support 0.15 and minimum confidence 0.9:

MemberOverTeam_AllTypeOper_JavaFile=1, MemberOverClass_M=1 $\Rightarrow$ *MemberOverOwnTeam_M=1*

This rule tells us that if a student's M-type operation number ($Mod\#$) on Java file is at the low level over his team, and his total M-type operation number on all extension files also takes the low level of the whole class, his total M-type operation over his own team must be low.

$RatioInClass_{FileNumber_M} = 3 =>$

$RatioInClass_{FileNumber_{LastModify}} = 3,$

$RatioInClass_{fileNumberLastModify}javaFile = 3$

This rule shows that if the number of files modified by a student can reach the medium level in the whole class, he has high possibility to be the last modifier of medium amount files comparing with the whole class, so do Java files.

$StudentOverClass_{workingDays}M = 1 => StudentOverClass_{workingDays}M_{java}File = 1, StudentOverClass_{beginDate}A_{java}File = 5, StudentOverClass_{beginDate_{All}TpeOper}{}_{java}File = 5$

This rule implies us: if a student only spent few days on modifying files (include Java files) comparing with other members in the same class, he usually begins his A-type operation on Java files pretty late. Besides, he begins any CVS operations on Java file very late.

All of these mined associate rules are not surprising. We did not discover much more informative or intriguing rules, due to the limited size of the dataset. Currently, the dataset collected from student projects are still too small to mined promising and novel patterns. However, even with these existing results, we still got some supportive information: the performance and development of an individual are basically consistent in different aspects. If the workload of a member on Java files is not large, his total workload on any-extension files should not be large. If a member only worked on several days, his total workload should not be large also. In such a busy and tight undergraduate study period, with all the requirements of a largish project, it is very difficult to get a good score with final gusty developments. That is why we think that a good work habit is important and CVSChecker visualization is useful.

4.5 Heuristic-Driven Analysis

Based on the visualizations, querying, statistical analysis and bottom-up analysis, we had observations corresponding to individual performance and team collaboration in the five student teams from different levels. To validate, polish, and better uncover these correlations in future, we executed heuristic-driven analysis strictly following the steps listed in section 3.2.5.2.

4.5.1 Summarizing patterns

We focus on these five teams because they gave us permission to analyze their data, instead of selecting with some special requirements. We observe and summarize the following facts on the 5 teams, related to the individual operation, team collaboration, file evolution, module design, and so on.

CVS Operation-related Summaries

Usually, each team has its own work trends. Although members might have some diversities on the early stages, all of them had similar spike times and idle periods as the project progressed;

In a team, there is usually a member whose number of CVS operations is much larger then his teammates. In general, such developers can be divided into two groups: (1) Added most files into the team CVS repository and did medium amount modification operations, or (2) committed the largest number of modification operations and had medium amount file adding operations. Most likely, they are team leaders. They act the core roles in their team and had strong impact on the collaboration and final product quality. Instructors should pay attention to their performance and give instant direction if necessary; Team leaders usually started their work earlier than their teammates, and had greater number of CVS operations around the due days (they did some mop-ups for their projects before the deadlines).

Some members had very few CVS operations. There are two possible reasons: incorrect CVS usage, or deficient contribution; The visualizations of CVSChecker can help instructors and members to know the real reason and help them do some corresponding adjustments;

Some other members do not have many file adding operations, but their numbers of modifications and total CVS operations are in medium-level. Moreover, they have few collisions and merges. These members in all probability take charge of several independent classes or components;

Teams with better work habits can still have unbalanced workload allocation and problematic project design; However, they usually have smaller possibility to have deficient development;

Teams with a great number of CVS operation records in repository and better work habits usually finish their projects in a better quality. TeamB and TeamD were two teams that got better assessments (project scores) from TAs and instructors and their average CVS operation amounts were higher than the average class level. In additions, they had comparatively more regular and even work schedules, especially in TeamD. This observation provided supportive information to our thinking: distinct team-collaboration patterns affect team performance and the product quality.

File-level Summaries

It is not always the case that a file modified many times is also modified substantially.

Most files that have been modified by a member have less number of collisions and merges, larger ratio of total LOC per modification number and smaller ratio of modifications per number of modified LOC.

When a team member is the "owner" of a file, i.e., he is the file's only modifier, then he tends to concentrate on their work mostly outside CVS; updates of the file in CVS are less frequent and represent more substantial changes;

Both abnormally high frequency of modification operations and large numbers of modified lines may be the evidence of an unstable file (i.e., a file that is either poorly developed or a highly coupled file that is affected by changes in many other files). Analysis of the modification operations correlations might indicate the latter, or records in a bug database might support the former hypothesis. In any case, this phenomenon may trigger the instructor to examine the file in question further and advise the students accordingly. Therefore, if a team has a large number of collisions, the instructor might suggest students to inspect the multiply modified files and see whether they can be re-designed or whether the maintenance can be assigned to a single person.

Most teams have some files that have not been modified since they were added into CVS repository;

The most possible reason of the abnormal large project size is the module copying;

Most file removals happened only because of the renaming or relocating;

Hardly Java classes were added in the project design phase (project part 1);

Some testing files usually have been added just before each deadlines, and some of them will be deleted soon;

Almost all the added batch files were not Java classes. They could be the test files, image files, JavaDOC files, and so on.

Most Java classes were added into CVS repository in procession after the project part 2;

Minor refactorings (e.g.: renaming) usually happened around deadlines;

Most creators of those important files finish the first several revisions before other members joined in;

4.5.2 Evaluating the summarized patterns

We evaluate above summaries with the collected project/team objective information, such as project history, questionnaire fillings, etc. In our case study, each member was required to finish a set of questionnaires. Their answers are the best subjective information to consult with.

4.5.3 Developing heuristics and queries based on these validated patterns

Since we are not only interested in the past performance, but also the likely future performance on new team projects, repeating the observations on CVSChecker visualization and data analysis manually is not a wise way when we have the data of a new team. Thus, to achieve the above goals, we have been working on developing a set of heuristics and have developed a set of queries that correspond to our intuitions about relevant (both desirable and undesirable) behaviors of teams and individuals at a high level. Our current queries include as following:

- Team leader query;
- Independent developer query;
- File management query;
- Problematic file query;

We applied these queries on the data of the 5 team again, detected the following results, and got confirmation from students' questionnaires:

- Detected team leaders of TeamA (LSName29), TeamC (LSName8), and TeamD (LSName58).
- Members LSName42 (TeamA), ExtReplace26 (TeamB), and LSName87 (TeamC) were members who focused on several independent components;
- Members LSName29 (TeamA), ExtReplace30 (TeamB), and LSName8 (TeamC) were file managers. They added most new files into their project CVS repository and committed most removing;
- 48 classes were picked out by the queries as the problematic files of all teams in the class. All those dangerous files discussed in section 4.3 were gathered;

4.5.4 Applying the heuristics and queries

We extended and applied these heuristics and queries, generated new patterns, re-evaluated the results on these new datasets, and did some adjustments on the heuristics if it is needed. We used the data of those students who gave us permission and queried the following results: 6 team leader candidates were selected automatically and 5 of them were obviously proven with the questionnaire results; 5 students were detected as members who mainly focused on components and a set of dangerous files of them were picked out. Those role-related results were confirmed by CVS reports and questionnaire results.

4.6 Patterns

In this section, we select and list some important patterns summarized based on the analysis of CVSChecker. All these patterns are applicable to small-size teams in educational environments. We will compare them with the patterns summarized from other environments, such as open source communities in next chapter.

We categorize the patterns into three types: factual patterns, red flags, and team-role profiles. A factual pattern expresses some characteristic of the development history of no obvious negative or positive implication. A red flag captures a problematic situation whose persistence may warrant a preventive action. They should be detected early and avoided. A team-role profile includes those patterns that are related to some specific team roles.

4.6.1 Factual patterns

- Late file additions. Student teams usually do not added files (especially the core development files, such as Java classes) into CVS repository until their whole design phase finished.

- Aggregative files. Most test files or image files or configuration files always were added into CVS at the same day/phase. Usually, the date is near a deadline, and sometimes the group of test files would be removed again after the deadline.

- Adding Java class in succession. Most Java classes were not added in batches. Their additions usually scatter from end of the design phase to the final deadline.

- Renaming and relocation existed. In this exploratory case study, each team had several renaming or relocation cases. If these things happened frequently, it is also an embodiment of the insufficient planning and design before they started works.

4.6.2 Red flags

- Underuse of CVS. Most members used CVS very little in the early phases, and they exhibited an irregular workload curve - long idle times interleaved with sudden peaks before deliverable deadlines. This pattern is problematic because we found that it often is either a symptom of under-contribution or a source of future collisions.

- Multi-way collisions. Collisions usually involved more than two members. This pattern may be indicative of high coupling, poor modularization, or poor allocation of the labor.

- Watch for merges. Most files with collisions had earlier successful merges. This pattern seems to suggest that when successful merges of divergent file revisions are noticed, the team should consider re-design their responsibilities around the affected files to avoid future collisions.

- Miscellaneous. Several other less pervasive problematic patterns were also identified, including excessively large files, frequent collisions/merges, and repeated alternating file additions and removals.

4.6.3 Team-role profiles

- Leaders vs. component developers. The two most common roles in these case studies were team leader (a core contributor who is de facto in charge of the overall project and steers the development effort for a given period) and component developer (an exclusive contributor to a specific file or module for a given period).

- Leaders are architects. Leaders tended to add a lot of new files in the beginning of the project. Consequently, they had the most influence over the architecture and evolution of the system and the division of labor.

- Component developers work on existing artifacts. Unlike leaders, component developers tended to add few files or no files at all.

- Leaders contribute heavily. Leaders usually also performed a large number of CVS operations, modifications in particular, that exceeded by far the number of operations performed by their teammates.

- Leaders contribute steadily. Leader had a better working habit. They started contributing early in the project and had relatively even work curves.

- Component developers have limited focus. Not surprisingly, most of the CVS operations of component developers were modifications to a small set of files, with relatively few collisions with their teammates.

All these patterns will be compared with those patterns extracted from our next case study in the next chapter (teams in open-source communities).

Chapter 5

Three teams in open-source community

The open-source process model is emerging as a new lightweight paradigm and increasingly popular paradigm for software development. It has already produced several successful products. This process is fundamentally different from more traditional analysis- and design-driven processes, which raises a set of interesting research questions: what activities are carried out in open-source projects and by whom? Are there typical or exceptional patterns? In this chapter, we report a case study conducted using CVSChecker to examine three small open-source project teams. We discuss the insights that the CVSChecker analysis produced regarding these teams and compare them with the results from previous case study with senior student teams depicted in Chapter 4.

5.1 Objectives

Our first case study with CVSChecker examined the development process of senior undergraduate student teams and identified several patterns. Some of these patterns can be thought of as indicative of good teamwork and others as symptoms of problematic performance. However, that case study was conducted in a controlled environment, in the sense that the student teams followed a process largely orchestrated by the instructor. Software teams vary greatly - from small student teams in an academic environment, to teams of various sizes in the software industry, to the expanding open-source communities. More recently, our interest has expanded to the open-source context.

The influential "Cathedral and Bazaar" paper [59] discusses the open-source development process as an almost silver-bullet solution: "the open source movement consists of ideal cooperative people, where conflicts are few and can be resolved within a community."

In this case study, we try to gain some insights on how this model works in practice. To that end, we apply the CVSChecker tool on several typical open-source projects towards a better understanding of the nature of teamwork and collaboration in such projects. As an initial step, we are interested in the similarities and differences between this style of development and the more controlled styles observed in controlled academic settings.

The main goal of this case study was to analyze the teams in open-source environment with the following questions:

1. Can CVSChecker also be applied to Open-Source Projects (OSPs) to reveal developer collaboration and file evolution patterns?

2. Can one easily and intuitively understand the development trajectory of an OSP only with the help of CVSChecker?

3. Can CVSChecker detect healthy and problematic patterns in OSPs?

4. How similar (or different) are role-specific behaviors and team-collaboration patterns in academic and open-source environments, and

5. What are the characteristic differences, among different project-development processes (e.g. inexperienced student teams in academic environment following a design-driven process and teams in self-regulating open-source communities)?

To sum up, the goal of this case study was to examine whether the CVSChecker tool is useful for OSPs and whether its functionality is sufficient to reveal interesting information in the behavior of teams following this type of process. In addition, this case study is designed to investigate whether those patterns identified in the initial case study are applicable, and identify new patterns that are possibly unique to OSPs.

We believe that if the information is suitably presented and highlighted, CVSChecker tool can help developers (especially newcomers) in OSPs to better understand the project development process and the code evolution. Moreover, teams in open source communities have their specific patterns together with some other patterns similar with those in educational environments.

5.2 Settings

5.2.1 Steps

Based on above scopes and goals, this case study was executed with the following steps:

Step 1: project selection. There are abundant OSPs in open-source communities, how to select the objects in this case study? We give a farther explanation in 5.3.2.

Step 2: Inferring development milestones. Open source projects usually last for a long period, such as more than 2 years. Their code sizes are also very large: hundreds of files were created, developed, and maintained. We use CVSChecker visualizations to get a quick and rough idea about the whole process; We believe that CVSChecker visualizations can help us to identify important project milestones, specific developers, or suspicious files with underlying design or collaboration problems.

Step 3: Focusing on the each phase or initial development phase. To further understand the work of these developers or files, we divide the whole process into several small phases according to those milestones, and zoom in some specific ones. Most OSPs usually have an initial release followed by long maintenance periods with several new releases. The implication is that we had to figure out when the initial development phase ended and when the maintenance phase began. This information can usually be retrieved by CVSChecker and proved from the supplementary project records, but it is not always accurately recorded. Fortunately, locating the various milestones, whether or not they coincide with explicit releases or documented in project records, based on CVS data is an important function of CVSChecker.

Step 4: Zooming in specific phases and apply the standard CVSChecker methodology. Once we focus on a specific phase, we analyze it with CVSChecker from different aspects elaborated in Chapter 3.

Step 5: Summarizing observations and extracting patterns. We recapitulate the observations we had in each phase, and extract patterns based on them. The patterns should include two groups: specific to OSP, and common patterns as those in student teams.

5.2.2 Project Selection

Open-source software is developed according to the "bazaar" model of distributed software development, as characterized by Eric Raymond [59], where the source code is allowed to be studied, modified and redistributed. It enjoys considerable patronage as the chosen development model for a number of well-known and widely-adopted projects including the GNU/Linux kernel, Apache and Mozilla [52]. Beyond these long-term, large-scale projects, the open-source process model is also adopted by thousands of smaller, more short-term projects. These projects, created and managed by several volunteers with limited experience, are comparable to the student projects that have been analyzed. Therefore we

Project	Number of Developers	Register Date	Our Checkout Date	Development Status
OSP_A	6	2002-07-10	2004-12-24	4-Beta
OSP_B	6	2002-02-08	2005-02-03	5-Production/Stable
OSP_C	8	2003-12-03	2005-02-04	4-Beta

Table 5.1: 3 open source project teams

wanted to investigate to what extent the team behavior was similar or different.

Because we try to analyze the impact of processes on team collaboration and member performance in OSPs, the project selection has a vital influence on the results.

We identified several OSPs comparable to the student projects we had studied. Putnam et al. [58] claim that small size is the key to a successful project. We have been following this adage in organizing the student teams, and for our OSP case study, we looked for several similarly small open-source projects with no more than nine members (according to Belbin's "9 team roles" theory).

Duration is also a good metric for project scale. Most OSPs usually have an initial release followed by long maintenance periods with several new releases. The student case studies lasted for approximately two months, with design and coding as the two main activities. Students usually can not spend too much time on the course project because of the curriculum design. The project deadline could be considered as equivalent to the first product release date with stable, complete end-to-and functionality. It would have been impractical to constraint the OSP length to be similar to the student projects' length. One-year and more than two year, are two project length yardsticks for our project selection.

In www.sourceforge.net, the projects development status is divided into the following 7 levels 1: Planning, status 2: Pre-Alpha, status 3: Alpha, status 4: Beta, status 5: Production/Stable, status 6: Mature, and status 7: Inactive. An OSP can span several levels at the same time. To avoid too young or two idle projects, we selected projects from level 4 to level 6.

We also decided to constraint ourselves to Java-based projects to be consistent with the student projects.

Based on above considerations, we selected three OSPs randomly which we will refer to as OSP_A, OSP_B, and OSP_C in the rest of the paper, from www.sourceforge.net. Table 4 lists the basic information of them:

5.3 Basic Results

This section introduces those selected visualizations and observation summaries generated in our case study on three OSP teams (labeled OSP_A to OSP_C). Diagrams are presented at various levels: by team, by individual, and by file. Such diagrams intuitively show trends enable us to gain a high-level impression of team and individual performance.

5.3.1 OSP_A

OSP_A is a command-line Java application that generates HTML reports from CVS repository. There were six members involved in this project totally. Coincidentally, it appears that all the members were volunteering university students. There were two sub-modules under the root node, which we refer to as Module1 and Module2. All the six team members contributed to the development of Module1 while Member1 was the only developer for Module2. Because we focus on team collaboration and Module2 only includes images or html files instead of program files, we avoid it in following content.

Fig. 5.1 shows the distribution of CVS operation types for each member while Fig. 5.2 shows the temporal distribution of CVS operations for each member. To enhance the chart readability, we deleted the columns of showing the total CVS operation number of each member.

From Fig. 5.1, we have following observations:

- The total number of CVS operations of Member1 was far greater than that of his teammates. Similarly, the numbers of his addition and modification operations were larger than those of his teammates; Similar to those main developers in student teams, Member1 also answered for most file removals;
- Member2 almost had no CVS trail at all (only few modifications);
- Almost all the operation records of Member3, Member4 and Member5 were modification besides very few file additions, removals, and local deletions;
- Member5 was the major member who involved in collisions and merges, although his number of modification operation was not large;
- Most operations of Member6 were patching;
- Anonymous developers left only three operation record types: O (checkout), P (patch), and W (removal of local file copy) because they do not have right to modify the files

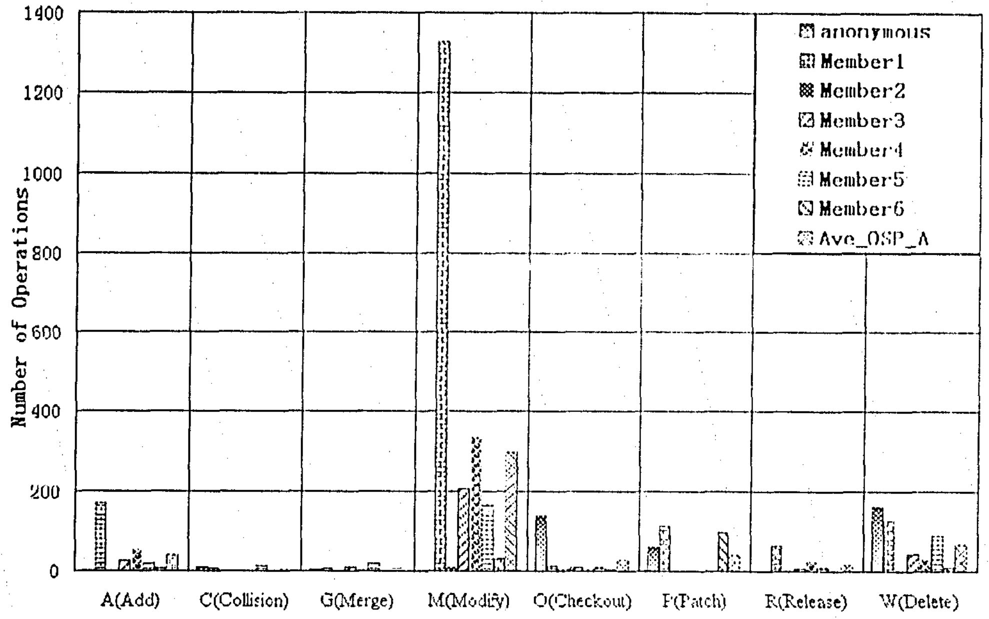


Figure 5.1: The distribution of CVS operation types for each member in OSP_A over whole process

in CVS repository;

- Except anonymous developers, the real members did very few checkouts;
- Collisions and merges seldom occurred in OSP_A, comparing with student teams.

From Fig. 5.2, we can have several interesting observations:

- There are several spikes, such as near days 40, 50, 260, 280, 520, 600, 830, etc. These dates should be examined more closely;
- Member1 was very active throughout, especially after day 260;
- Member2 only did 6 modifications on 2003-03-17 (Day 250) and 2 modifications on 2003-08-11 (Day 397);
- The operations of Member3 scattered on those dates with spikes;
- Member4 was active before day 260, but almost did not do any work later;
- Most operations of Member5 congregated before day 350;
- Member6 almost did not have any traces before day 260;

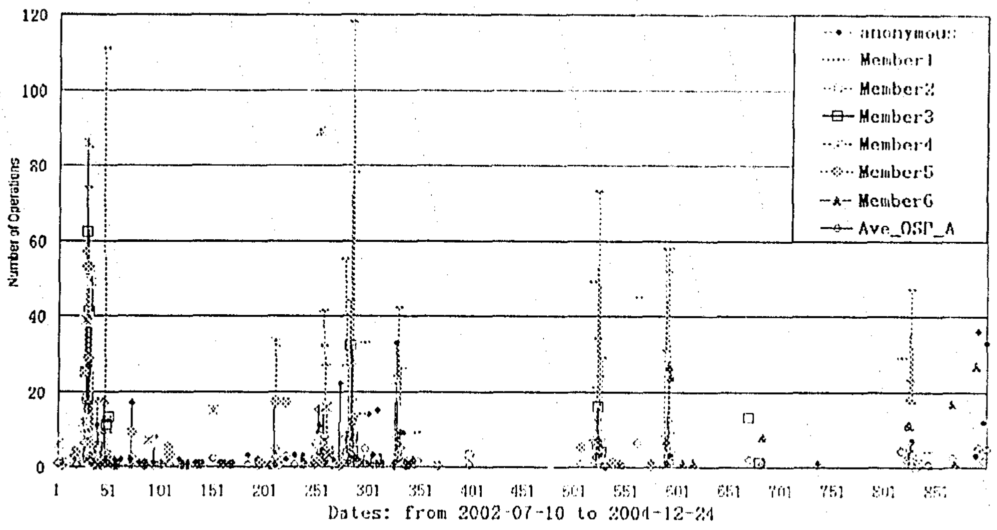


Figure 5.2: The temporal distribution of CVS operations for each member in OSP_A over whole process

- Anonymous developers had a long idle phase after day 350, and resumed near the end of the process;

It seems that Member1 and Member4 could be the two core developers and Day 260, Day330, and Day825 could be the most important milestones.

According to the project history, Day260 (2003-03-26) was the release date of version v0.1.3 and Day826 (2004-10-13) was the release date of v0.2.2.

Fig. 5.3 displays us the file additions and removals within the whole process. We labeled these two milestones with lines.

- All the file additions and removals assembled on four phases: from day 1 to day 100, from day 200 to day 400, from day 500 to day 600, and after day 800. All of those special dates we detected from Fig.5.3 scattered in the four phases, instead of those idle periods. Different from those charts of the student teams in Chapter4, OSP_A has started their development since very beginning. Many Java classes were added into CVS at the early stage. This phenomenon did not happen in student teams because many OSP developers began their projects usually a little bit ahead of the setting up of the project CVS repository in www.sourceforge.net, and moved the rudiments into CVS later;

- Similar to student developers, the OSP developers had some batch processing near

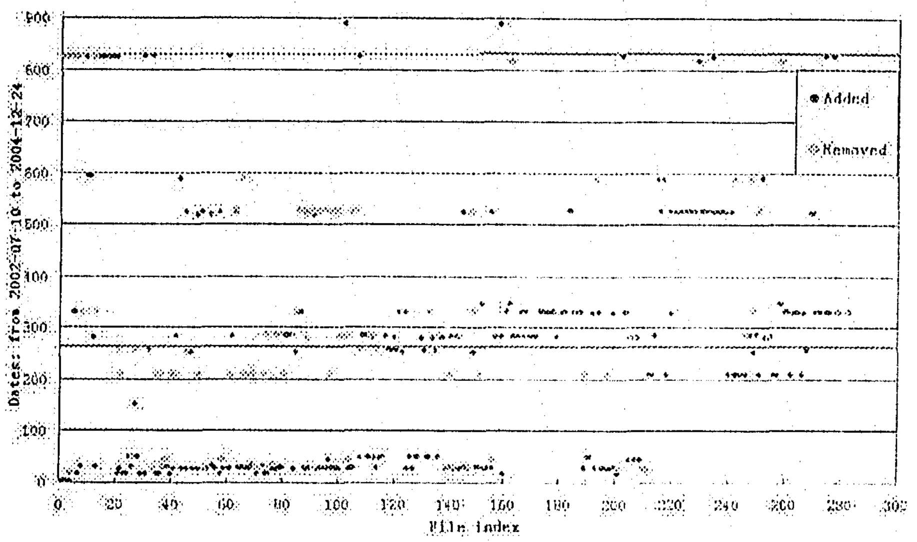


Figure 5.3: File additions and removals in OSP_A over whole process

the deadlines;

- Most Java classes were located between Files 20 and 210, almost all of them were added into CVS in the first two phases;
- The files between file210 and 275 were added later than day200. File index enables us know that all of them are test files existed in "tests-src/net/sf/OSP-A/".
- Only one file was added between day100 to 200. It was a configuration file named as "src/net/sf/OSP_A/logging-silent.properties".
- Relocation also can be detected. A typical example is that the main page file "index.html" and four log-related image files (File278 to 282) were added to module "htdocs/" at the very beginning, then moved to the root directory on Day330; Another example is that several configuration files (file1 to 12) were deleted or removed from the root directory to "etc/"on Day825.

Since we want to focus on the initial development phase, we first separate the history into two main phases by this release date of v0.1.3: Phase 1 (from 2002-07-11 to 2003-03-26) and Phase 2 (from 2003-03-26 to 2004-12-24). Zooming in Phase 1, we made the following observations according to Fig. 5.4 and Fig. 5.5:

- Member1 still had the most CVS operations in this phase. More specifically, there were two busy periods for him. The first was from day 25 to day 50. Remarkably

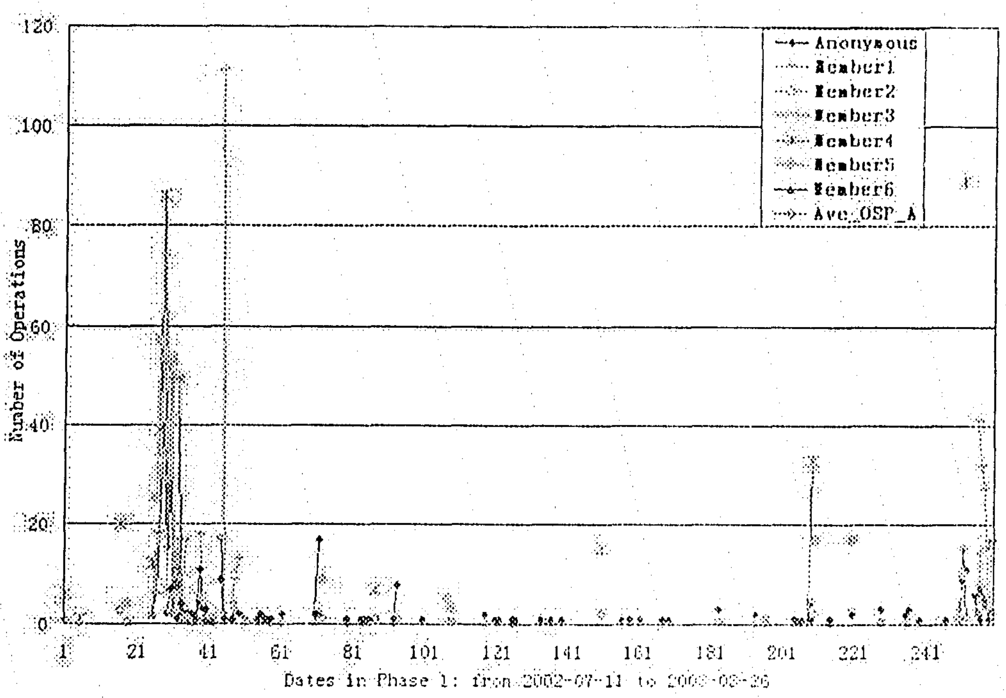


Figure 5.4: The temporal distribution of CVS operations for each member in OSP_A in Phase 1

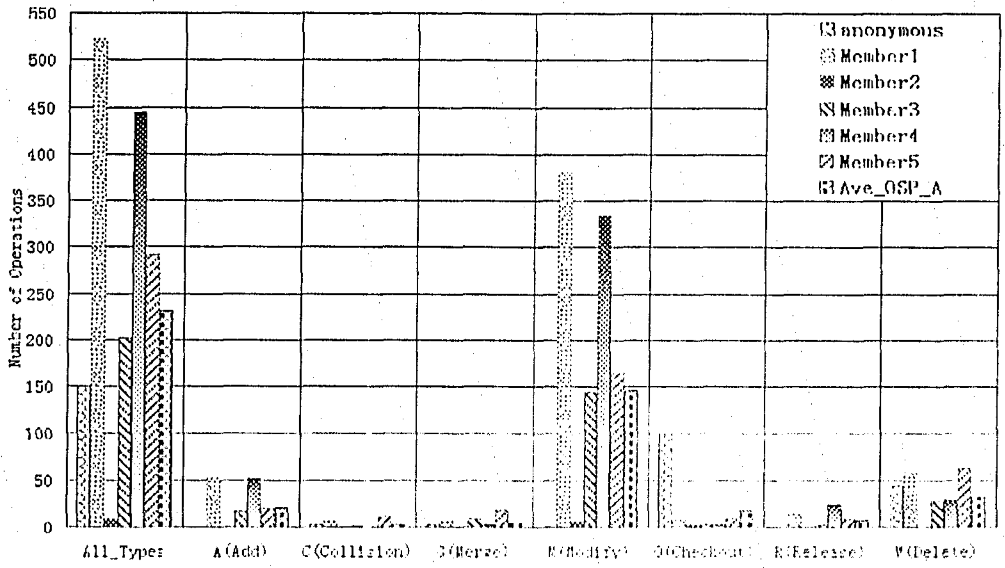


Figure 5.5: The distribution of CVS operation types for each member in OSP_A in Phase 1

around date 48, there was a significant peak. The second peak covers the days just before the release of v0.1.3 (Day260).

- Member4 had an almost equally large number of operations. There were also two active periods for Member4: one was around day 30 and the other was around the days approaching the release of v0.1.3.
- There was a long, relatively idle period from around day 75 to day 200. Only Member4, Member5 and anonymous developers had a few sporadic actions during this period.
- During Phase 1, no P (Patch) operations were performed, which is not surprising since this is the initial release of the system and outside contributors did not have the opportunity to participate the project yet.
- Most merge and collision operations were caused by Member5, and almost all of them happened in Phase 1. This may indicate that the responsibilities of this developer are not clear since he appears to be interfering with the development of other members.

The blown-up CVSChecker charts for Phase 1 indicates that spikes just before day 50 could coincide with another project milestone. We consult the project records and figure out that day 46 (August 25 2002) was the delivery date of v0.1.2.b. In order to see the details before this release, we zoomed in on a smaller period. The result is shown in Fig. 5.6.

From the distribution of operations in this sub-period (not shown), we realize that the contributions of each member was not remarkably different than they were in the enclosing period, Phase 1. However, Fig. 5.6 quickly revealed that the days between day 25 and 30 constituted another peak period in development activity. Moreover, two core developers (Member1 and Member4) had an overlap around this period, could it be that the former was handing over the project leadership to the latter?

Records showed that on day 32 (August 11 2002), a new version, v0.1.1.a, was released. Because this date coincided with the only peak before this release date, we identify the period from 2002-07-11 to 2002-08-11 as the initial development phase for the comparison with the student projects in previous section. Afterwards, it is most likely that the maintenance and updates started. Therefore, we think that the team collaboration and individual performance patterns of this new period can be compared to the student case studies, which did not involve maintenance and updates. Fig. 5.7, Fig. 5.8, Fig. 5.9, and Fig. 5.10 drill

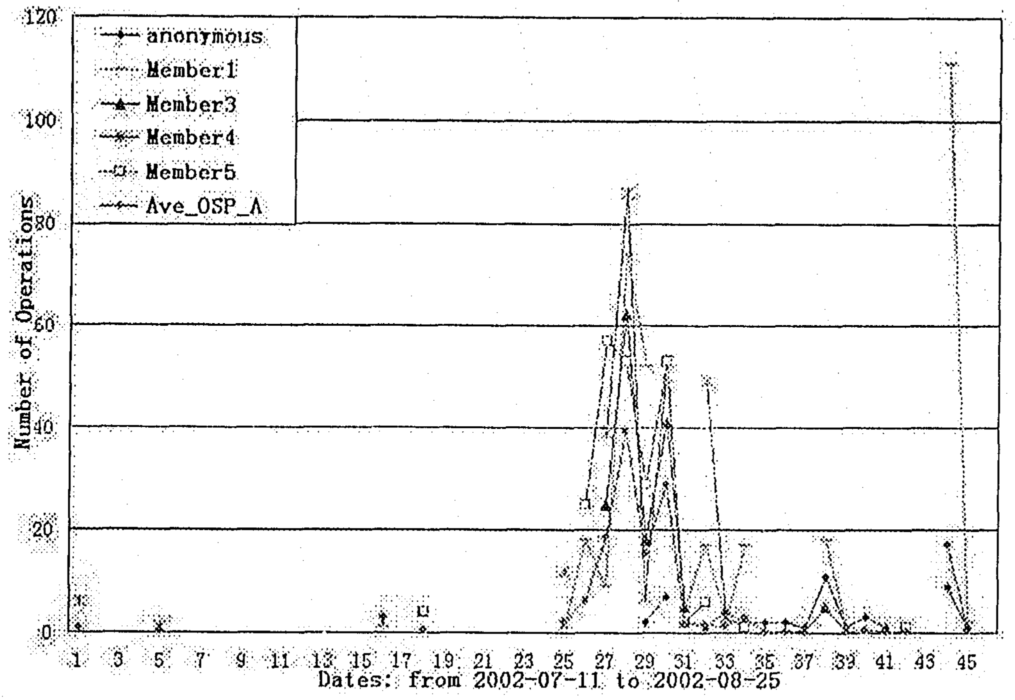


Figure 5.6: The temporal distribution of CVS operations for each member in OSP_A before the release of v0.1.2b

down again to illustrate what really happened in the initial development phase. Fig. 5.7 shows the operation type distribution for each member, Fig. 5.8 displays the file additions and removals in this phase while Fig. 5.9 and Fig. 5.10 show detailed Java file views in this period.

Comparing the initial development phase with the visualizations of later phases, Fig. 5.5 and Fig. 5.7 confirmed our hypothesis of a handover of leadership. It appears that Member4 was the core developer in the initial development phase, but did not manage the project after v.0.1.2.a was released (Day 32). After this release, Member1 took over the lead role. This result revealed by the CVSChecker was confirmed by project records.

All these visualizations displayed the following information: 91 files were added in this small phase. 77 of them were Java classes. They were added mainly in three time slices: Day2, Day17, and between Day27 to Day31. 11 Java classes were modified by at least three members. The numbers of collisions and merges on these files were higher than that of other files. Except Member1 and Member4, nobody else independently took charge of individual files.

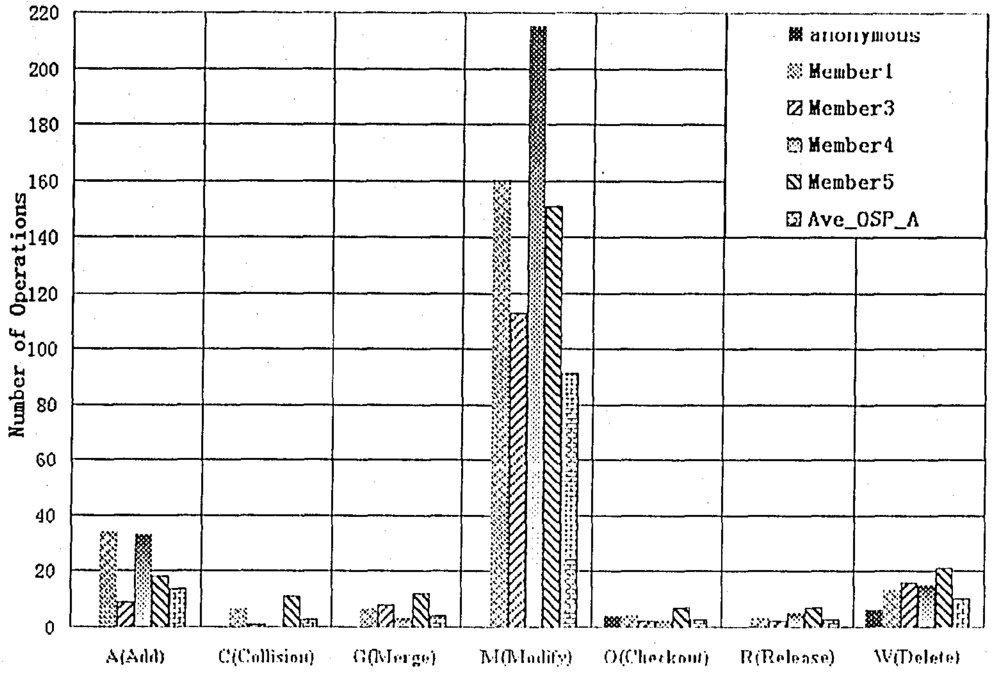


Figure 5.7: The distribution of CVS operation types for each member in OSP_A from 2002-7-10 to 2002-08-11

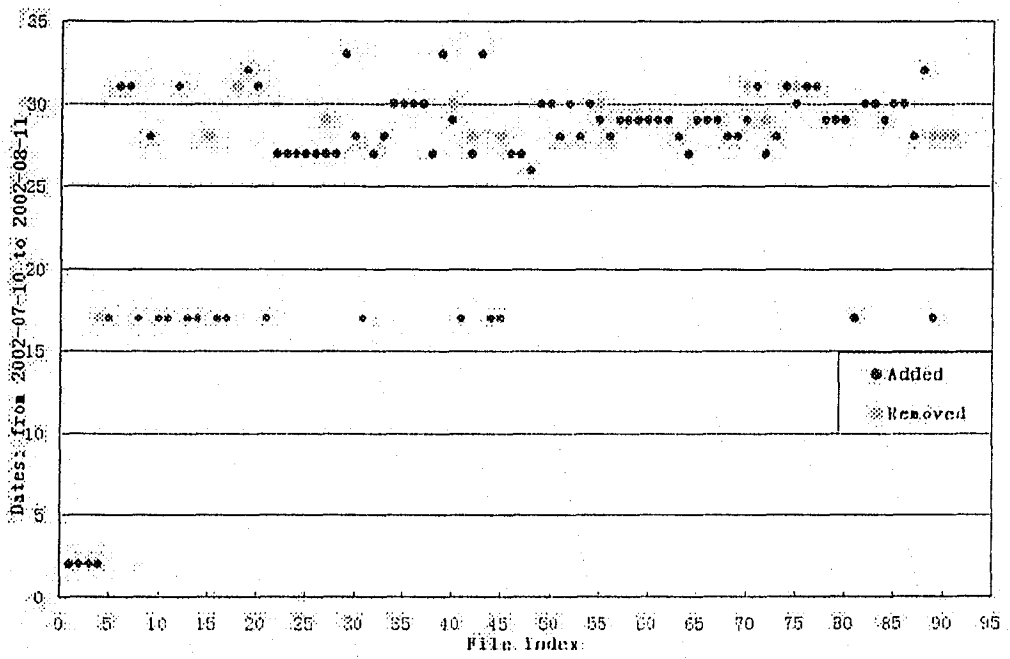


Figure 5.8: The file additions and removals in OSP_A from 2002-7-10 to 2002-08-11

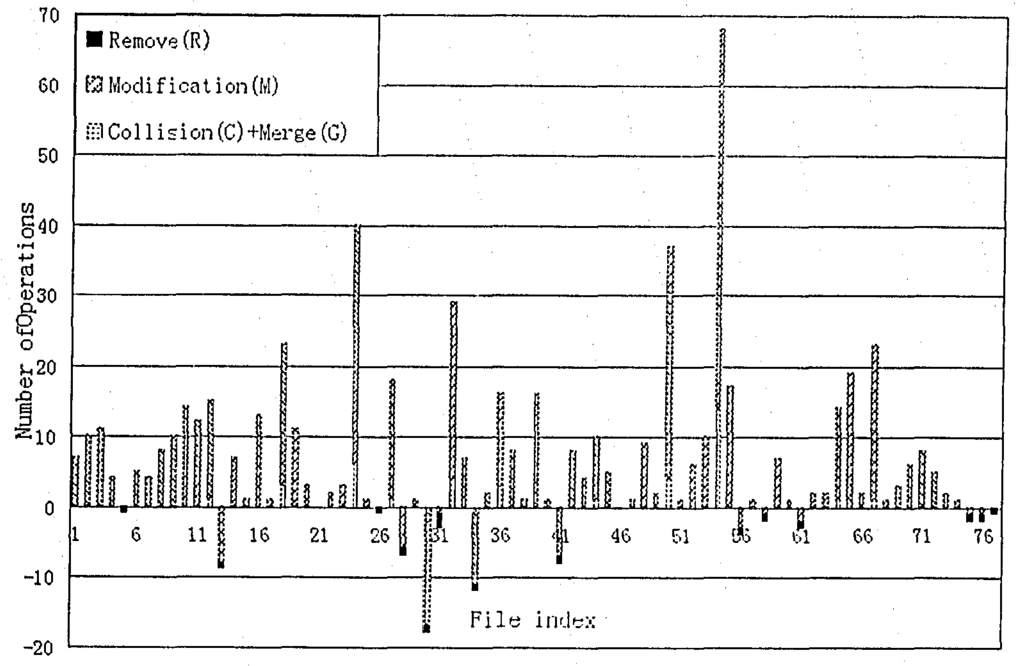


Figure 5.9: Distribution of operations by type in OSP_A, on each file from 2002-7-10 to 2002-08-11

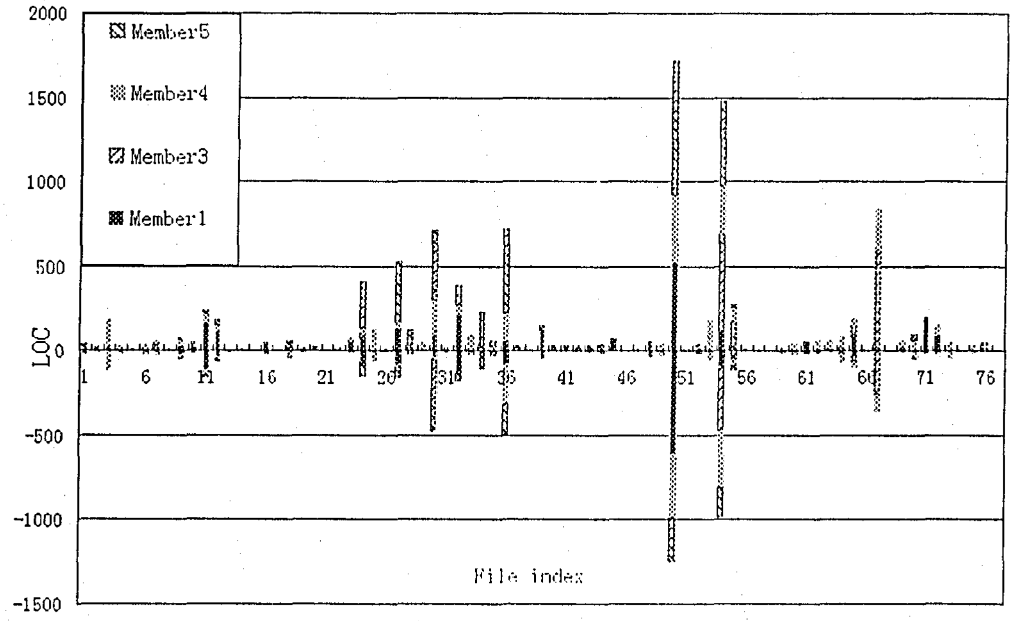


Figure 5.10: Added and Deleted LOC of each member in OSP_A, on each file from 2002-7-10 to 2002-08-11

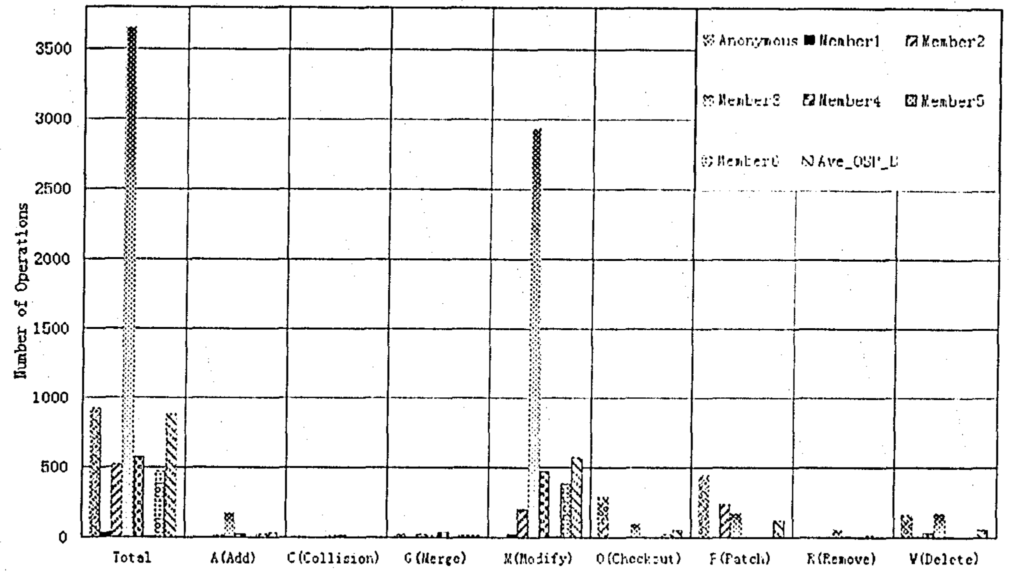


Figure 5.11: The distribution of CVS operation types for each member in OSP_B over whole process

5.3.2 OSP_B

OSP_B is a unit testing framework written in Java. There are also six developers. Fig. 5.11 shows the distribution of CVS operation types for each member in OSP_B while Fig. 5.12 shows the temporal distribution of CVS operations for each member over the whole process (from 2002-02-08 to 2005-02-03).

- Within the whole process, the number of CVS operations committed by Member3 was far exceed the number of other members, not only the all-type total number, but also file additions and modifications;
- Member 1 and Member 5 almost had no contributions recorded by CVS;
- Except anonymous developers, the real members did very few checkouts; Also, they removed very few files from CVS;

It is obvious to divide the whole process into several small phases based on those typical spikes in our visualization. Project history proved that new releases happened on almost all the spike day. We divided the whole process into 7 phases according some of them:

- In almost all the phases, Member3 dominated the development. However, the disparity has been getting smaller since Phase5;

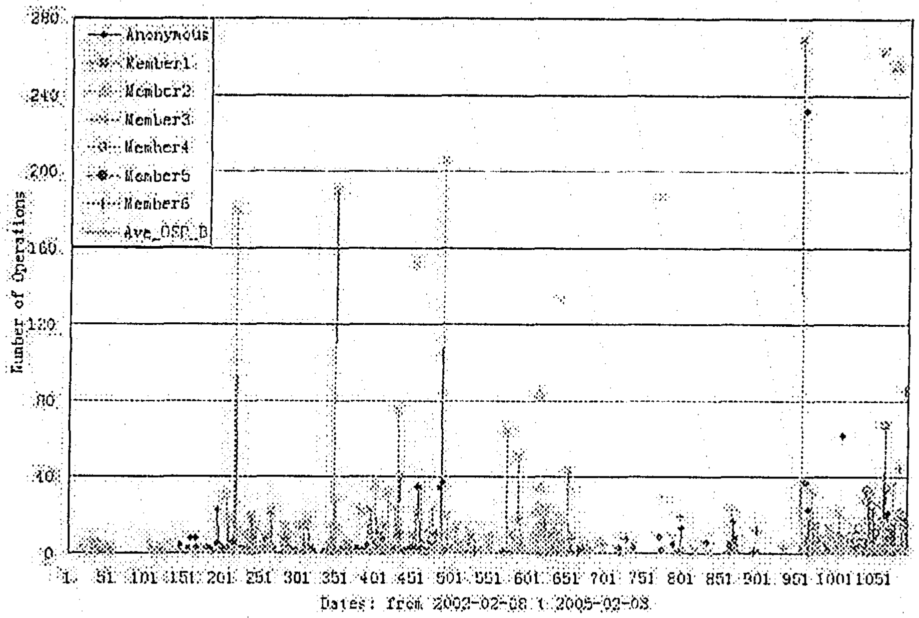


Figure 5.12: The temporal distribution of CVS operations for each member in OSP_B over whole process

	Start Date	End Date	New Release	Day number
Phase1	2002-02-08	2003-03-09	v1.2	375
Phase2	2003-03-10	2003-08-09	v1.2.3	527
Phase3	2003-08-10	2003-11-06	v.1.3 Perl	617
Phase4	2003-11-07	2004-09-26	v 1.3 Pre2	942
Phase5	2004-09-27	2004-11-12	v 1.3	989
Phase6	2004-11-13	2005-01-09	v 1.4	1050
Phase7	2005-01-10	2005-02-03	Our checkout	1073

Table 5.2: 7 phases of OSP_B

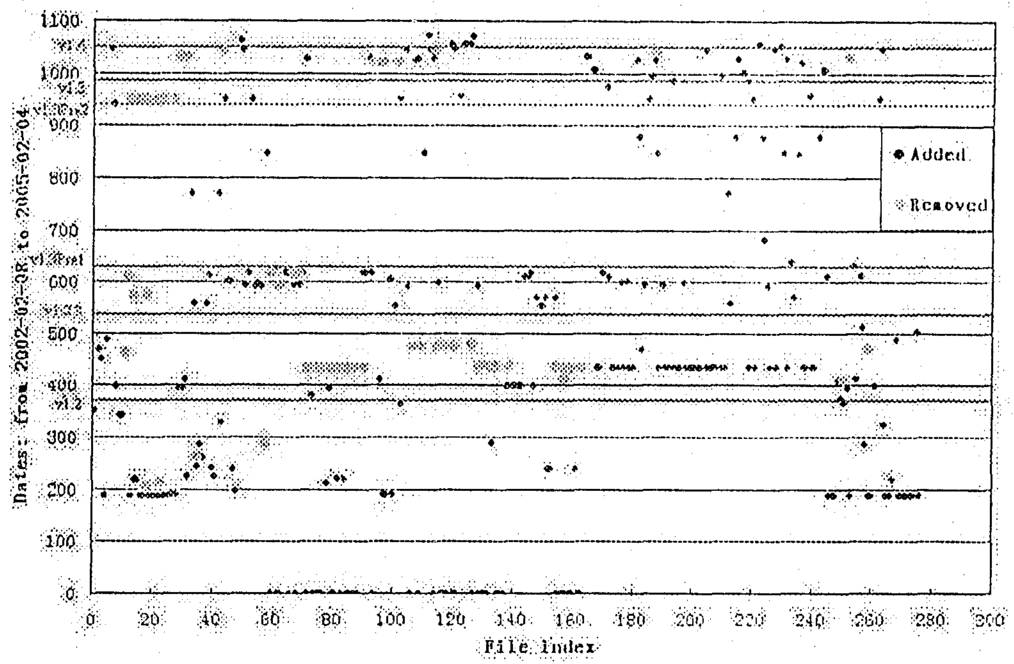


Figure 5.13: File additions and removals in OSP_B over whole process

- Anonymous developers always had the homochronous spikes as those of core developers in team, especially on those important release dates of new revisions;
- Member2 and Member6 had suddenly high CVS operations between the birthday of v1.4 and the day when we checked out;
- Member2 also had a short high-operation period at the late part of Phase3. He definitely did some special works; we should track deeper to figure it out later.
- The operations of Member4 mainly occurred after Phase5.

In Fig. 5.13, I indicated six minor milestones of project development process using lines.

- No file additions or removals happened until Day186 (2002-09-02). Using the query function of CVSChecker, we knew that in this idle period, only some members and anonymous developers checked out;
- There were still some sparse phases, e.g.: between Day1 to Day186 and Day700 to Day800. All these phases are not near the release days;

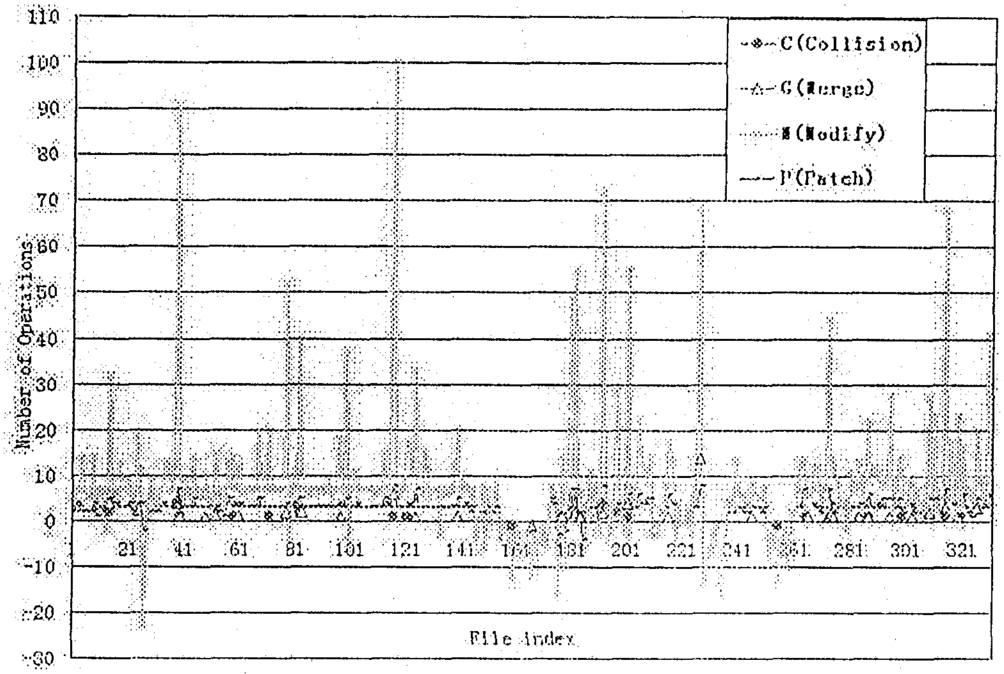


Figure 5.14: Distribution of operations by type in OSP_B over all files

- Testing files or image files always were added or removed batch by batch, and the removals usually happened after the new release days.

From Fig. 5.14 we noticed that:

- Almost all the Java classes shown above the X-axis experienced patches while most removed classes did not. After checking the file index, we found out that the three major removed groups below the X-axis were temporary testing files. Combining with Fig. 5.13, we can notice that these removals happened in Phase2 (between the release dates of v1.2 and v1.2.3).

- All the other sporadic removed classes were outcomes of renaming or relocation. Even the open source community is claimed as an ideal cooperative paradigm, we still find that there are some classes experienced collisions and merges.

- Member3 was not only exceed his teammates on the number of CVS operations, he also substantially changed a great deal of the LOCs on Java classes, and took charge many small classes independently.

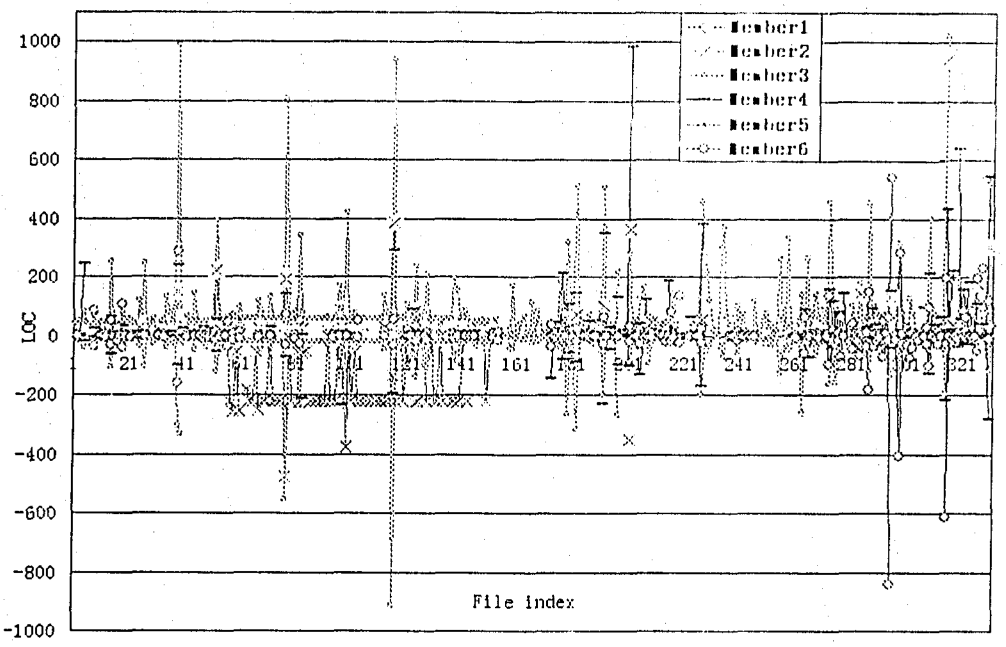


Figure 5.15: Added and Deleted LOC of each member in OSP_B, on each file over the whole process

- Those classes with collisions and merges shown in Fig. 5.14 usually were modified by more than two members.
- An interesting phenomenon was shown in Fig. 5.15: Member2 deleted the same number of LOCs from a large group of classes. This should not be a coincidence, and there definitely were relationship among those removed code pieces.

Now we zoomed in a smaller phase to have a more detailed examination. To save the space, we only present those promising visualizations. Fig. 5.16 displays the CVS operation distribution on each type of each member in Phase1: in addition to the anonymous developers, only Member3 did real contributions in Phase1. No collision, file removal, patch at all.

In Phase2, Member3 still was the only developers besides the very few modifications executed by Member2. Several groups of test classes were removed by Member3. Till now, there were still very few collisions and merges.

In Phase3, although Member3 continued his domination, Member2 began his substantial contributions, especially in the last 35 days (see Fig. 5.17); Member6 also had few modifications. However, as more developers joined in, collisions and merges emerged also

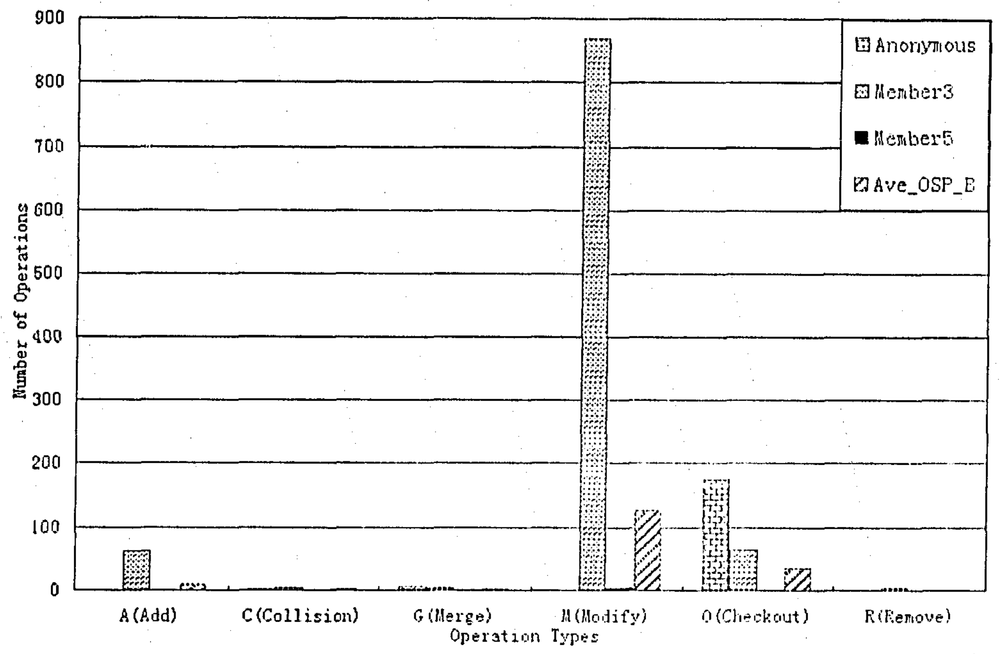


Figure 5.16: The distribution of CVS operation types for each member in OSP_B in Phase1 (from 2002-02-08 to 2003-03-09)

as shown in Fig. 5.18.

Fig. 5.19 shows the LOC modifications in Phase3. Although the number of CVS operations of Member2 in this phase was less than that of Member3, Member2 did some very special modifications on some classes: he deleted same LOC sizes from dozens of classes. We checked the code and figured out that he did a very important refactoring: on day613, Member2 "removed a lot of redundant code from a lot of classes ." and "factored out to a common base class".

Fig. 5.20 shows the works in Phase4 while Fig. 5.21 shows the works in the remainder phases (Phases 5, 6 and 7).

- Members began the patch operations, especially those anonymous developers;
- Member3 still remained on the leading level;
- Member6 contributed a lot in this phase;
- Member2 had some operations after the refactoring works in Phase3.

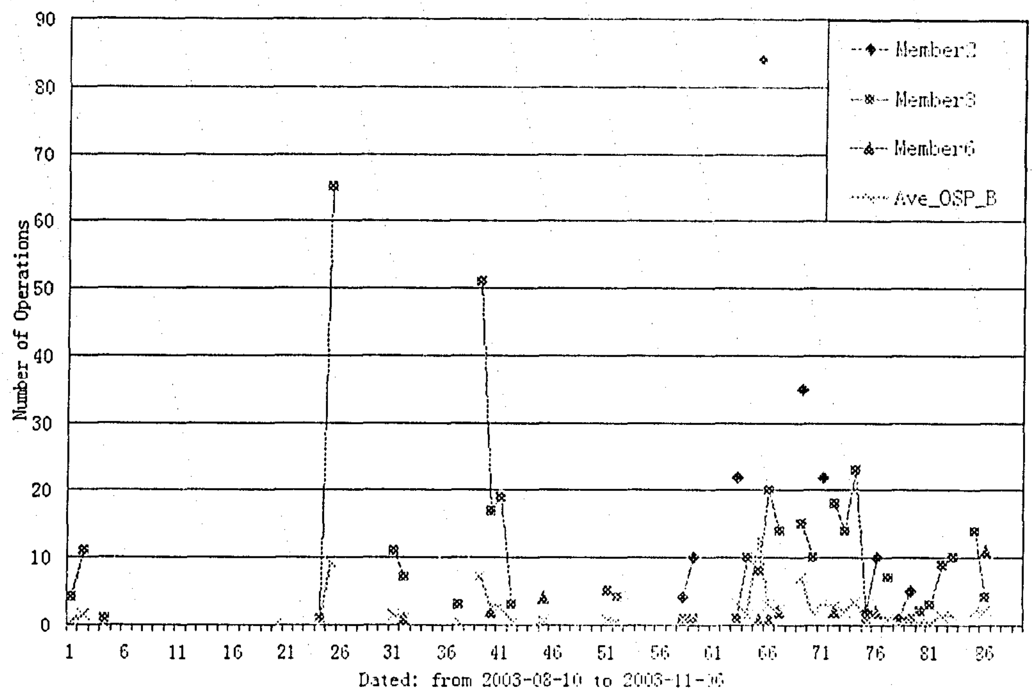


Figure 5.17: The temporal distribution of CVS operations for each member in OSP_B in Phase3 (from 2003-08-10 to 2003-11-06)

- Member4 exceeded Member3 on the total number of CVS operations in this period. Although almost all his operations were modifications, he also incurred collisions and merges;
- Member3, Member6 and Member1 continued their contributions;
- Member2 only did patches as those anonymous developers;

Combing all above visualizations and observations, we can summarize that:

- OSP_B also had idle gaps and spikes. Although they might not have some strict deadlines as those students in university, developers in OSPs still had the similar work manners that contribute remarkably near the release dates;
- Member3 should act as the leader/manager role in the team, and he developed and maintained this project for a long time before other members joined in;
- Members did not have to keep the workload balanced as we advocated in student teams. The operations of anonymous developers always happened around the releases, and almost all the operations were checkouts and patches;

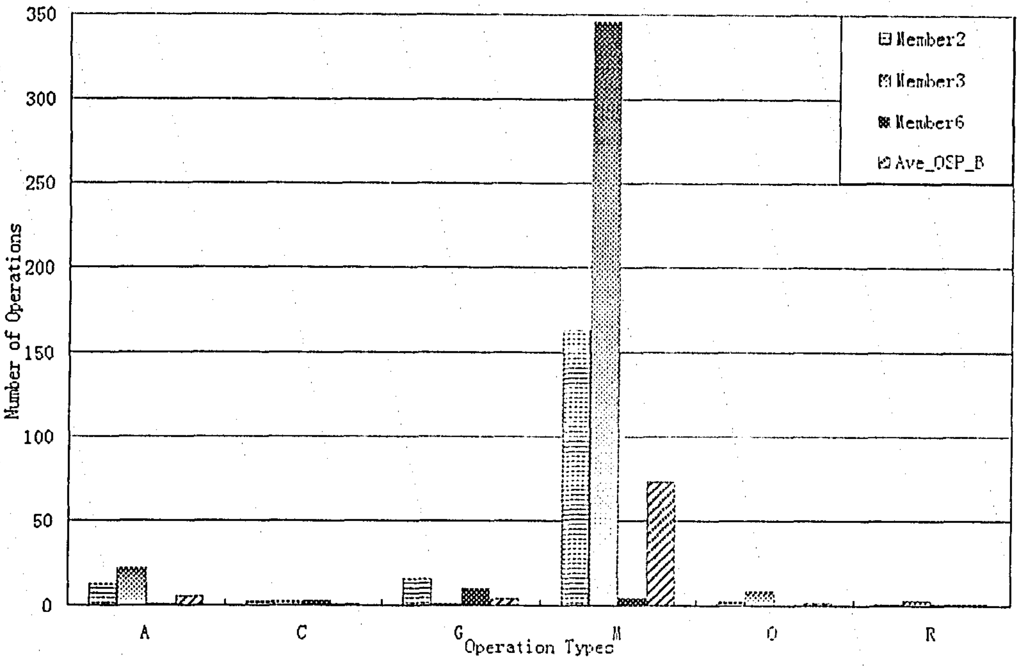


Figure 5.18: The distribution of CVS operation types for each member in OSP_B in Phase3 (from 2003-08-10 to 2003-11-06)

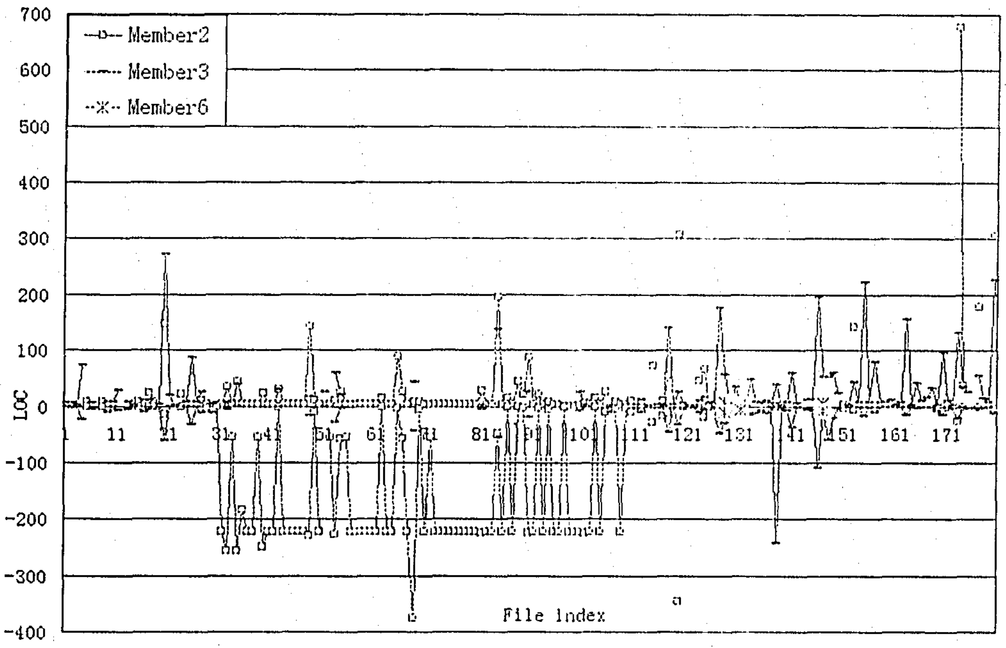


Figure 5.19: Added and Deleted LOC of each member in OSP_B, on each file in Phase3 (from 2003-08-10 to 2003-11-06)

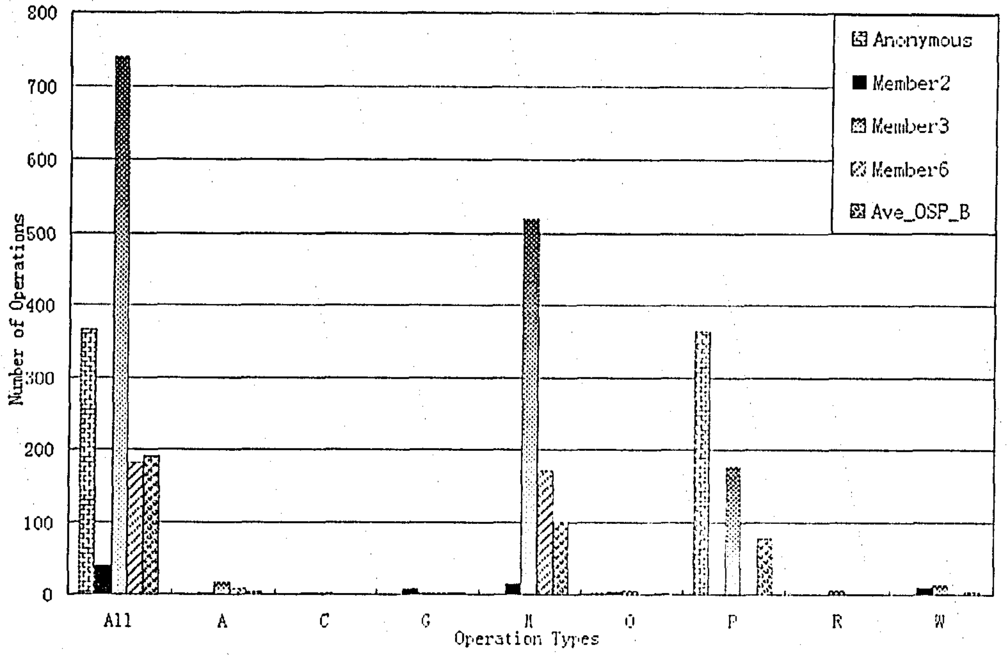


Figure 5.20: The distribution of CVS operation types for each member in OSP_B in Phase4 (from 2003-11-07 to 2004-09-26)

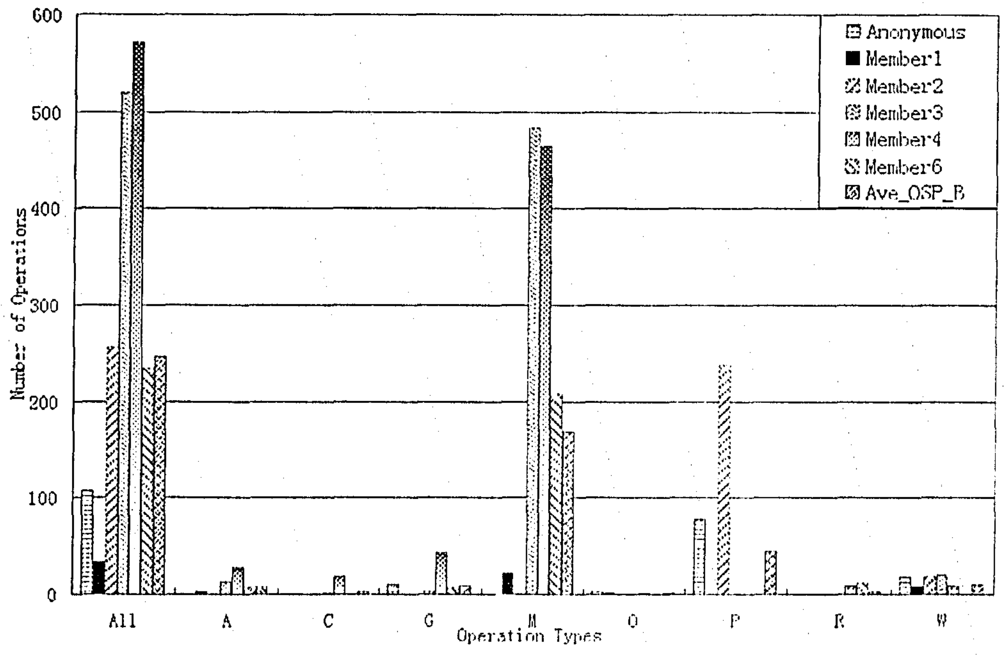


Figure 5.21: The distribution of CVS operation types for each member in OSP_B after Phase4 (from 2004-09-27 to 2005-02-04)

- Developers seldom checkout after their local working directories were set up. However, almost all the members in student teams committed "cvs checkout" command constantly. It still had design and collaboration problems: each time when new members began their contribution, new collisions and merges always arose by them more or less.

The project development process can be predigested as such a short story: the rudiment of OSP_B was designed and finished before being moved to *www.sourceforge.net*. Member3 was the initial developer and contributed on it incessantly with one-up works. As the project growing up, test files were introduced temporarily and removed shortly after. As new members joined the development, structure design problems emerged and were detected. Some members tried refactoring works. Since the new revisions were releasing, the number of patches increased.

We got validations from the official website for our analysis results (such as "OSP_B was originally written by Member3 and released under an apache style license. Since then, it has received many contributions from other developers "), modification logs (such as labeled as refactoring), and so on.

5.3.3 OSP_C

OSP_C is a static analysis tool that examines class or JAR files looking for bugs. Fig. 5.22 and Fig. 5.23 give us a quick idea about the CVS historical operations along the whole process. Although this project registered in www.sourceforge.net on 2003-12-03, the CVS repository did not have any historical records until 2004-02-13. Therefore, our visualizations start from 2004-02-13. Compared with OSP_A and OSP_B, OSP_C is still pretty young.

There were several important days, such as around days180, 200,255, and so on. Project history told us that Day180 was the release date of revision0.8.4. We divided the whole process into two phases: Phase1 started from 2004-02-13 to 2004-08-10 (from day1 to day180), and Phase2 started from 2004-08-11 to 2005-02-04 (from day181 to day 358). Member2 started early and did most operations while some other developers, such as Members 6 and 7, began their contributions later.

Combining Fig. 5.22 and Fig. 5.23, we noticed that:

- Almost all those operations done by Members6 and Member7 in the later phase were patches;

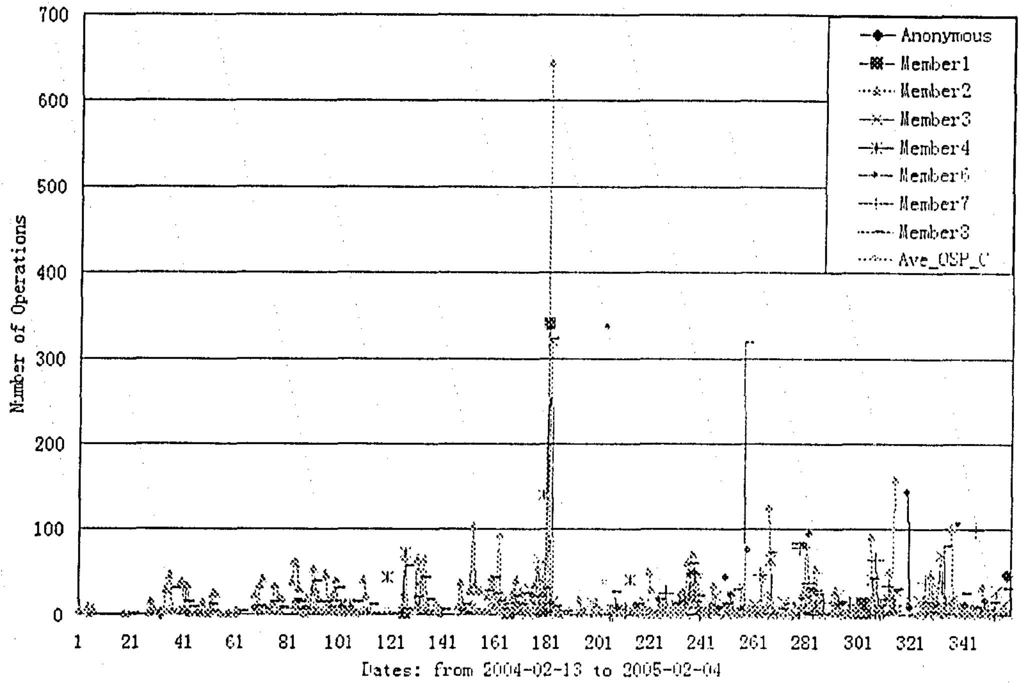


Figure 5.22: The temporal distribution of CVS operations for each member in OSP_C over whole process

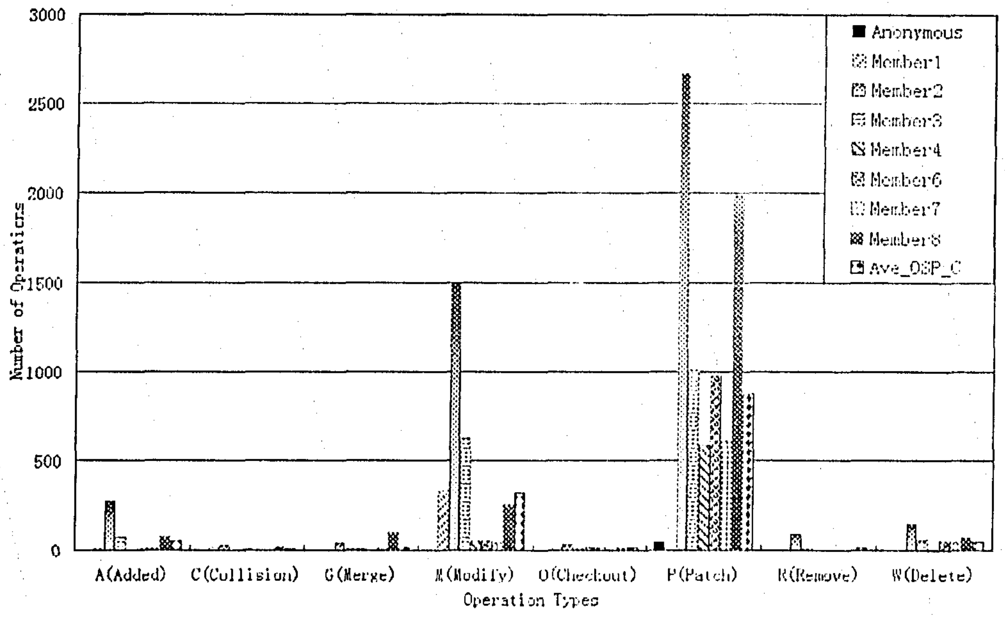


Figure 5.23: The distribution of CVS operation types for each member in OSP_C over whole process

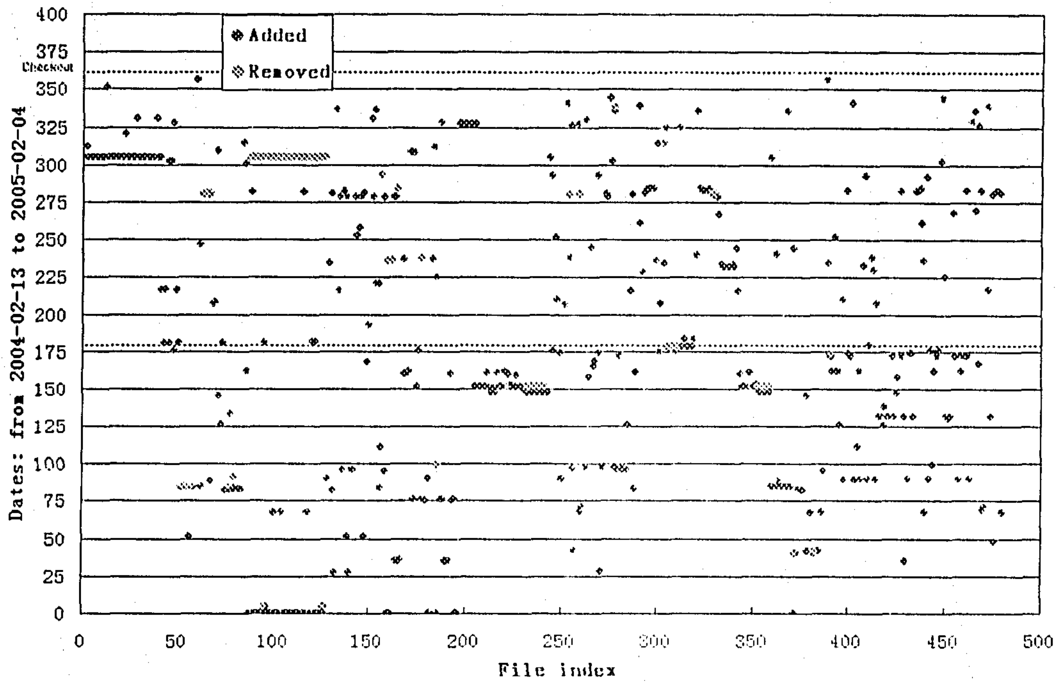


Figure 5.24: File additions and removals in OSP_C over whole process

- Anonymous developers did not have many operations. The possible reason is that this project is still not mature enough (All the released revision numbers were started with "0.");
- Although Member1 had very few operations, all of them were modifications;
- Almost all the operations of Member4, 6, and 7 were patching;
- Member5 did not leave any records in CVS repository. Later, we figured out that Member5 was another username of Member8. The only difference between these two usernames is that Member8 included the first letter of his given name. The developers did not leave any records in CVS with the username "Member5".
- All the members seldom removed files from CVS repository.
- OSP_C hardly had collisions and merges.

Fig. 5.24 displays the file additions and removals in the whole process. I labeled the release date of revision 0.8.4 and our checkout date using lines.

- Most significant file relocations or renames can be detected easily from this chart:

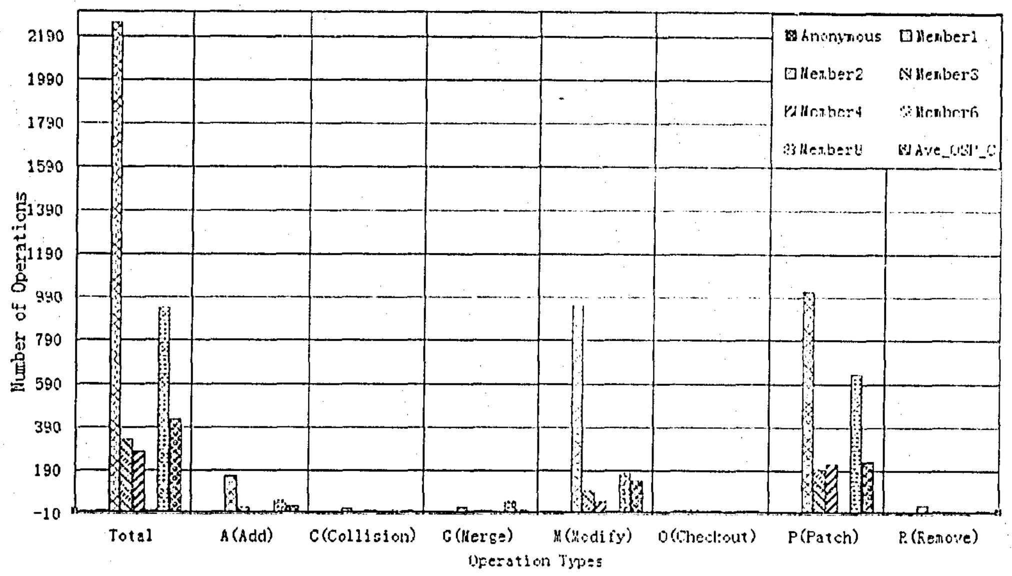


Figure 5.25: The distribution of CVS operation types for each member in OSP_C in Phase 1 (from 2004-02-13 to 2004-08-10)

- Example1: 40 files (from Filed 87 to 127) were removed from module "src/eclipsePlugin/de.tobject.OSP_C/.classpath/" to "src/de/tobject/OSP_C/" (from File1 to 40) on Day305.
- Example 2: 11 Java classes (File232 to 242) were added on Day147. All the original names were started with "X", and they were removed 5 days later. New files were inserted into file index from File206 to 226. Same thing happened on File353 to 358 at the same day.
- Example 3: 6 configuration files (File305 to 310) were added on Day 177, and then removed from "src/java/edu/UNIVNAME/cs/OSP_C/gui/" to "src/java/edu/UNIVNAME/cs/OSP_C/gui/bundle/" on the next day - Day178.

- The file additions and removals of OSP_C were more evenly distributed along the process than that in OSP_A and OSP_B.

Fig. 5.25 and Fig. 5.26 display the works in Phase1 (from 2004-02-13 to 2004-08-10) and Fig. 5.27 and Fig. 5.28 show the works in Phase2 respectively:

- All the operations of Member1, 6, 7 and anonymous developers only happened in Phase2. Actually, almost all the operations of Member1 were executed on a single

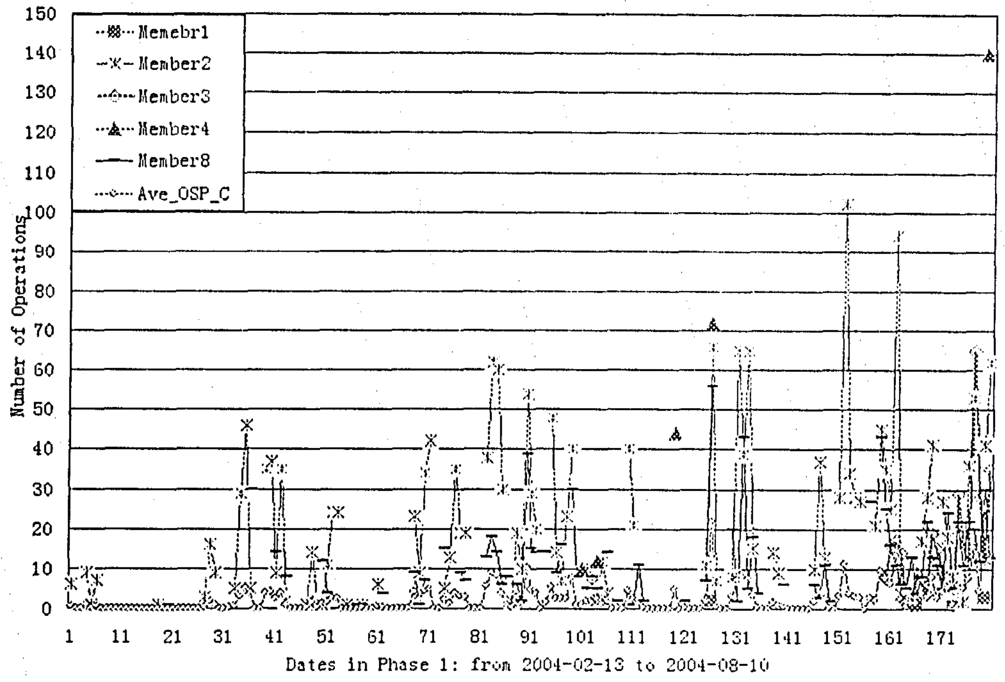


Figure 5.26: The temporal distribution of CVS operations for each member in OSP_C in Phase1 (from 2004-02-13 to 2004-08-10))

day - 2004-08-11 (the day after the release date of revision0.8.4), and all these operations are file modifications. We will figure out what happened on that day with the file level visualizations of CVSChecker later;

- Member1, 3, 4, 8 also had lots of CVS operations on or just after 2004-08-11. With the help of the query function of CVSChecker, we knew that Member1 modified three class revisions, added three new classes, and had 639 patches on 2004-08-11. On the same day, member4 made 328 patches and Member8 made 321 patches, and Member3 made 317 patches on 2004-08-12;

- Member3 dominated the development in both phases;

- Member8, 3 and 6 also had many CVS operations in Phase2. However, only Member3 had substantial modifications on Java classes while the other two did many patches.

Fig. 5.29 and Fig. 5.30 are simplified visualizations to show the CVS operation distributions and the number of modified LOC committed by each member on Java classes on Phase2:

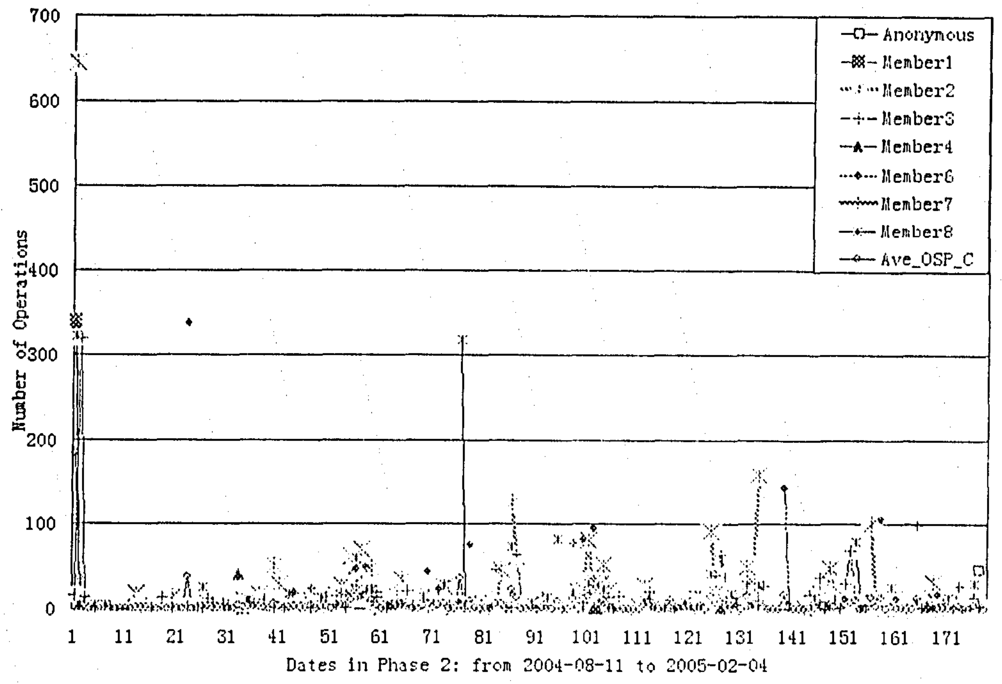


Figure 5.27: The temporal distribution of CVS operations for each member in OSP_C in Phase 2 (from 2004-08-11 to 2005-02-04)

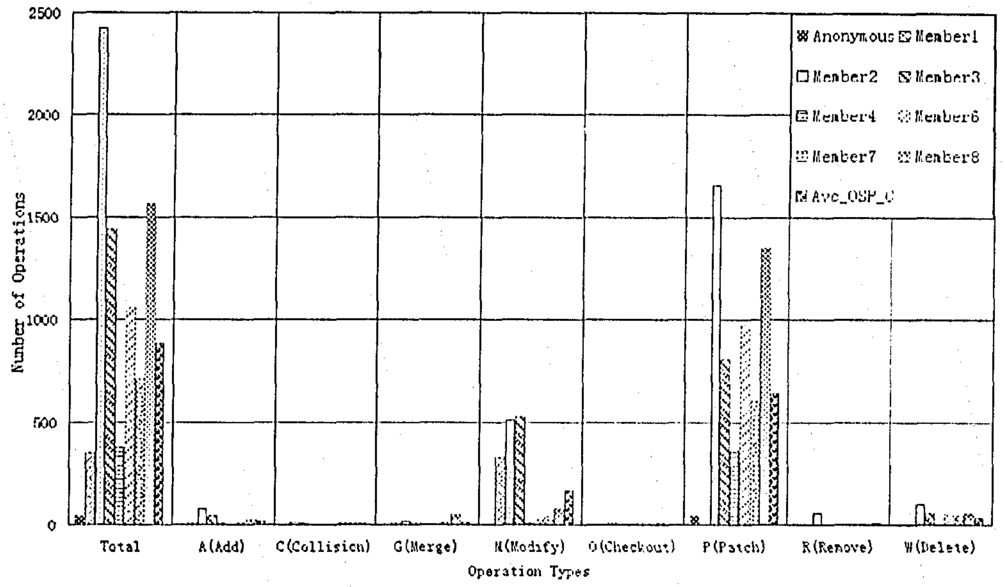


Figure 5.28: The distribution of CVS operation types for each member in OSP_C in Phase 2 (from 2004-08-11 to 2005-02-04)

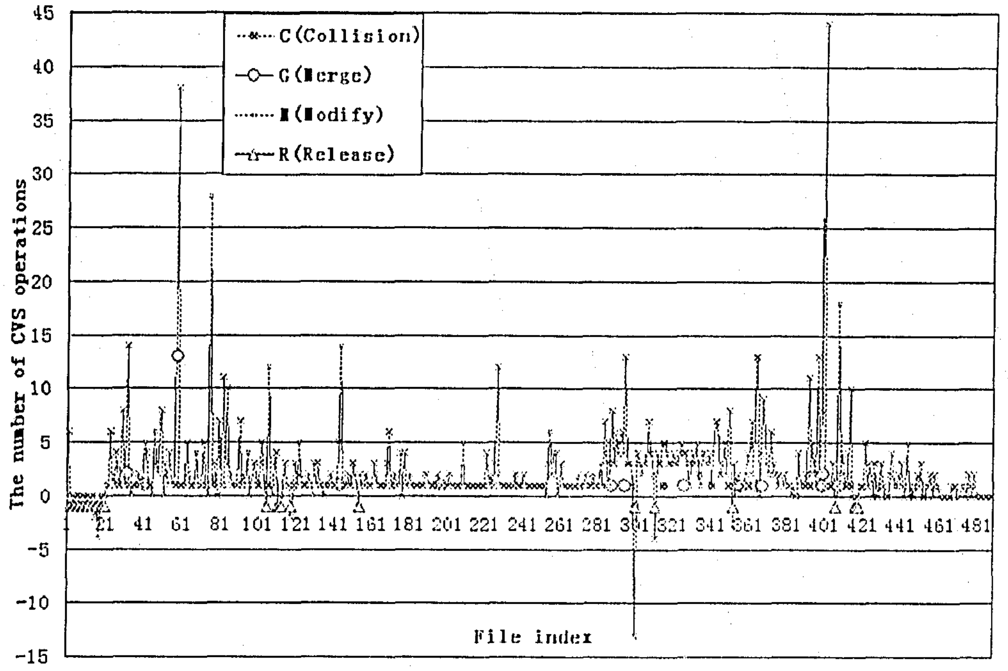


Figure 5.29: Distribution of operations by type in OSP_C, on each file (from 2004-08-11 to 2005-02-04)

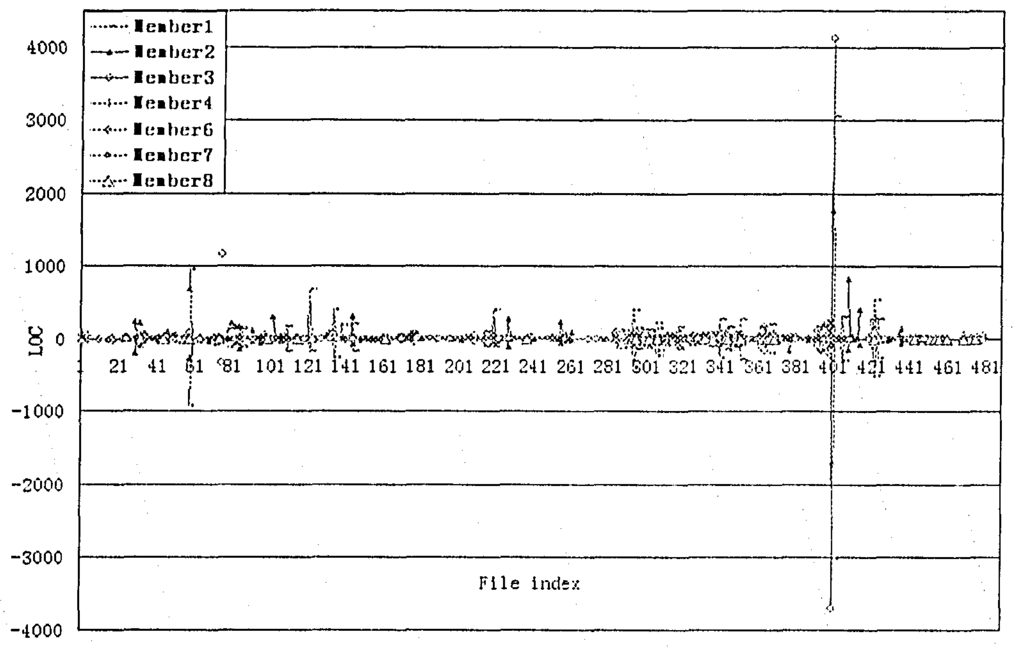


Figure 5.30: Added and Deleted LOC of each member in OSP_C, on each file (from 2004-08-11 to 2005-02-04

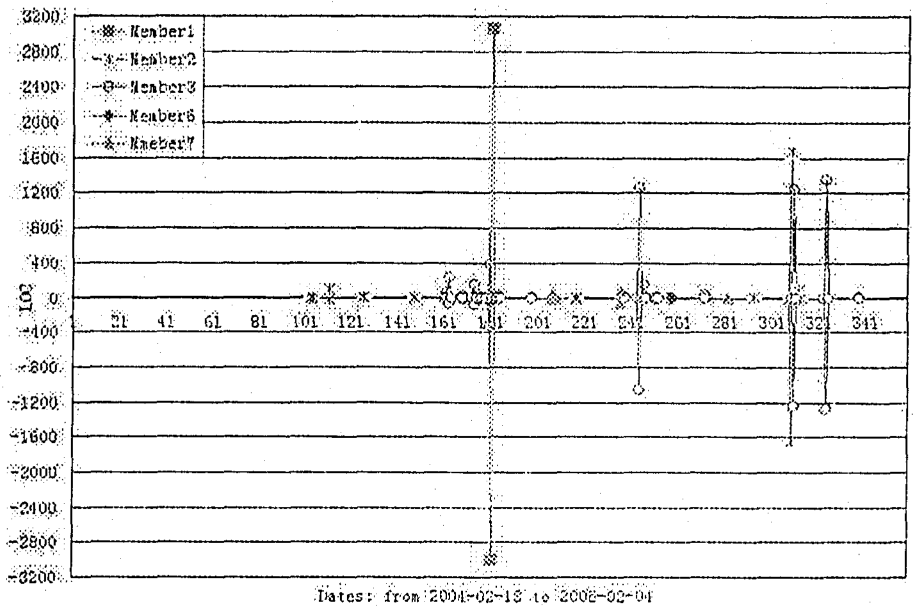


Figure 5.31: Detailed LOC Change by Date, on "../../cs/OSP_C/OSP_CFrame.java" (File401)

- Although there were almost 500 classes, some of them had only 1 or 2 collisions and merges;
- File61 and 401 are two typical files modified by more than one developer. Although Member1 only modified files on 2004-08-11 and his number of modifications was much smaller than that of Member2, he modified almost same LOCs as Member3 did. We use the query function of CVSChecker and found out that Member1 modified 316 classes at that time and the reason was "Massive whitespace checkin; reformat code with IDEA, substitute all spaces with tabs, conform to 4-space Sun coding conventions.". It can be understood as refactoring after new release;

Fig. 5.30 helps us to know that File401 experienced heavily modification by Member1. Because Member1 only worked on this day, File401 must be a part of that large MR happened on 2004-08-11 mentioned above. What is this file? What kinds of operations it experienced? Fig. 5.31 displays the detailed history of this Java class: "src/java/edu/UNIVNAME/cs/OSP_C/OSP_CFrame.j (File401).

- This file was created by Member2 and he also finished the first dozens of revisions;
- Member3 took over the development around Day160 (2004-07-23);

- In the following period, there were four obvious busy dates. Thousands of LOCs were changed in these days and in some of them, more than one member modified it. If we want to know who involved the collisions and merges in this file, these visualizations can help us to figure it out;

- On 2004-08-11, Member1 made a huge modification on it because he "Massive whitespace checkin; reformat code with IDEA, substitute all spaces with tabs, conform to 4-space Sun coding conventions.";

- Member3, Member7, and Member2 continued the developments with trivial modifications;

- With CVSChecker queries and reports, we figured out that around Day250 (2004-10-13), Member3 added a new menu function to the project. Around Day310 (2004-12-17), Member2 "Reformatted using tabs for indentation." Member3 did similar works on 2004-12-17 and 2005-01-01. All these dates are just after the new releases. Their operations can be understood as refactorings.

Combining all above visualizations and analyses, we can summarize that OSP_C is still on its immature stage without the formal release of revision v1.0. The basic functionalities have already been developed mainly by Member2, Member3 and Member1, while almost all the members keep patching. Refactoring-like works happened after new releases.

5.4 Patterns

In this section, we select and list some important patterns of OSPs summarized based on the analysis of CVSChecker (We did not analyze the data using KDD technique in this case study because the data accumulation is still not big enough). We also compare them with the patterns summarized from educational environments to get common or specific patterns. The patterns are still categorized into three types: factual patterns and red flags and team-role profiles.

5.4.1 Factual Patterns

- *Early file additions.* Different from those student teams in our first exploratory case study, members in OSP teams usually added files (especially the core development files, such as Java classes) into CVS repository since the early stages. This pattern is a reflection of the typical history of an OSP project: some interested members work on an issue until they achieve some presentable results before they move the works to a publicly available place.

- *Aggregative files.* Similar to student teams, most test files or image files or configuration files in OSP teams always were added into CVS in the same day/period. Usually, the day is near a deadline. Some batches of test files were also removed after deadlines.

- *Adding Java classes in succession.* Similar to student teams, most Java classes in OSPs were not added in batches. The additions usually scattered the whole process recorded in CVS until the most recent.

- *Renaming and relocation exist.* In OSPs, teams also had renaming or relocation cases, but they were less frequent than that in student teams.

- *Idle and busy phases exist.* Although OSP teams do not have strict deadlines as students had in courses, CVSChecker visualizations still revealed idle and busy phases in each OSP development process. Almost all the spikes happened in release dates, and all the members had jagged trends as long as their workloads were not too small.

- *Fewer file removals.* There are three main reasons for file removals: removing from CVS, renaming or relocation. All these instances in OSPs were much fewer comparing with student teams.

- *Fewer checkouts.* Except anonymous developers, the members in OSP teams seldom checked out once their local working directories were set up. Student developers were different: each of them in our exploratory case study had checkouts more or less.

- An important difference between the academic case studies and the OSP case study is that contribution was mandatory in the former while it was voluntary in the latter. The absence of certain patterns in the OSP case study could perhaps be explained by this difference.

5.4.2 Red Flags

- *Multi-way collisions.* Similar to student case studies, several members in OSPs were involved in collisions and some files were modified by multiple members. Comparing with student teams, the numbers of collision and merge in OSP teams were much smaller. But they are still patterns that may be indicative of high coupling, poor modularization, or poor division of labor.

- *Watch for merges.* We did not observe this pattern in the open-source case study. The possible reason should be that the project had a small code base already before it was moved to www.sourceforge.net. However, the initial development phase was still responsible for the files that were overall subjected to the highest number of collisions and merges. These files were not removed later, and continued to cause collisions in the later phases.

- *Underuse of CVS.* CVSChecker can help up to detect members with very few CVS operations in OSPs. Although this pattern is problematic because we found that it often is either a symptom of under-contribution or a source of future collisions, we can not treat it as a red flag because there is no workload balance rule in OSPs: all the members join OSP teams voluntarily and work according to their interests, capabilities, timelines, etc.

- *Miscellaneous.* Several other less pervasive problematic patterns were also identified, including excessively large files, repeated alternating file additions and removals, and so on.

5.4.3 Team-role profiles

- *More team roles in OSPs.* There were more common roles in OSPs. Not only team leader (a core contributor who is de facto in charge of the overall project and steers the development effort for a given period), component developer (an exclusive contributor to a specific file or module for a given period), but also patching developer (members seldom made other CVS operations except patching), inactive developer (members drop in and out of the project in different phases), anonymous developers, and so on.

- *Multiple Leaders.* In a student team, there usually is only one team leader for a project. However, there always are more than one leader in OSPs. The main reasons are the longer development process and the larger project scale. Usually, a leader proposed an idea and took charge it until some releasable versions were finished. As the project getting larger and more mature, another member took over the leading and maintenance roles .

- *Leaders contribute heavily.* Leaders performed a large number of CVS operations (especially additions and modifications) that exceeded the number of their teammates in their respective leading phases. This pattern was also pervasive in the academic case studies.

- Leaders are architects. Similar to those leaders in student projects, the leaders in OSPs added a lot of new files and therefore had the largest impact on the overall structure and evolution of the project.

- *Leaders contribute steadily.* They started contributions early in the project. However, different from the even lines of student leaders, OSP leaders usually had steeper spikes around milestones, with idle phases. Perhaps this pattern is unique to the course projects in an academic setting and is not typical of OSPs. However, Michlmayr [?] argues that steady contribution is a factor in an OSP's success. Perhaps the absence of this pattern constitutes an early warning sign.

- *Component developers.* In OSP teams, component developers are not easy to be detected. Most leaders were also component developers because most files were developed by them and very few other members exclusively owned specific files.

- *Component developers have limited focus.* Not surprisingly, most of the CVS operations of component developers were modifications to a small set of files, with relatively few collisions with their teammates.

- *Component developers work on existing artifacts.* Even there were a few non-leader component developers, they tended to add few files or no files at all. Ususally, team leaders prepared these files already.

- *Anonymous developers.* Anonymous developers had very few operations until the project was publicly released. Most operations of them were patching.

- *Patching developers.* Although there were some other peoples registered as the team members in OSPs, they did not join the early developments until the later maintenance phases.

Chapter 6

Conclusions and Future Work

In the context of this thesis, we developed CVSChecker, a tool that analyzes the collaborative software-development process of software teams. CVSChecker uses as its primary input data captured by the repository in which the team stores its software assets: it extracts facts regarding the history of operations that team members perform in their project repository and the evolution of various software-project metrics. Next, it proceeds to analyze these facts to infer more complex types of information about the (a) style of the team-members' collaboration, (b) the development contributions of individual team members and (c) the evolution of the software project. CVSChecker presents its analyses results in two ways: first, it generates a set of related graphs, visualizing the team's activities and the project progress; second, it produces a set of reports summarizing the analysis results to the team and the instructor.

The work of this thesis aims a very important general research question: Are there distinct patterns, trends and events in the collaborative software-development process of teams that one can recognize in the trail of the development activities captured by the software repository in which the team stores their assets? And once recognized, what do these patterns say on the "health" of the software project and the team's progress and how can they inform the project manager's decisions?

This research question is receiving a lot of attention recently and a substantial number of research projects world-wide evolve around it. CVSChecker contributes to this area the following.

- A method for analyzing the development behavior of the individual developers and the team as a whole. To our knowledge, the majority of the work on analyzing CVS repositories focuses on understanding the software maintained in the repository. There has been very little work focusing on systematically examining and understanding the team dynamics based on their operations' trail in CVS. CVSChecker implements a methodology for analyzing both at the same time: as a result, the results of both types of analyses can cross-fertilize and potentially produce more informative intuitions for the instructors.

- A tool that automates the above analysis in support of software-engineering instructors. Effective project management and student-team mentoring is of critical importance in software-engineering education. Recently, we have noticed an increase in educators' reflections on their teaching methods. However, with few exceptions, the analysis of educators' experiences is conducted in an ad-hoc manner. Moreover, most tools aimed at supporting software-engineering instruction focus on the mechanics of materials delivery and marking. CVSChecker is novel in that it aims at analyzing the students' development process in order to help the instructor guide this process. Furthermore, it does this in a systematic way that enables the comparative analysis of multiple case studies.

- A set of distinct development-process patterns and red flags. Through our experience using CVSChecker in the context of a third-year software-engineering course for over two years and examining the student-teams' process with it, we have formulated a set of interesting patterns, characteristic of different roles in the team, relevant in multiple life-cycle processes. CVSChecker can recognize these patterns and report them to the instructor, who can draw informed inferences about how healthy the team dynamics are and how well the project is progressing.

We have evaluated CVSChecker with two case studies: the first examined collaborative software development of student developers while the second focused on open-source projects of similar complexity and length. The results of these case studies indicate that CVSChecker is, indeed, able to discern interesting information about both case studies which implies that it could be a useful tool to instructors as well as project managers in general.

Moveover, CVSChecker also extracted patterns for each case study, and generated some pattern queries based on the evaluated ones. Through comparing the results from educational environment and open-source community, we had the following conclusions:

Student teams usually start their programming late, all the Java files are created in succession since the first CVS operation to the last due day. Batch processing happens on supportive files, such as test files, image files, configuration files, and so on. They usually are added into CVS repository in groups ahead of the deadlines, and most of them are removed soon. Because most student developers are not so experienced and professional, file renaming, relocation, and removing exist along the process. Some developers have low CVS usage, comparing with their teammates. Collisions happen on some files, and most of them are touched by at least three members. Fortunately, merges usually happen ahead of the collisions on these problematic files, instructors can give early suggestions with the help of CVSChecker visualization. In student teams, there are two typical roles: team leader and component developer. Team leaders usually are those students who start work earlier, create many files at the early phase, and have much more consistent contribution along the whole process. Moreover, they also have heavily contribution near the deadlines. Team leaders usually dominate the whole process. Component developers are those students who add few new Java classes to CVS, and their contributions focus on a small group of files. Moreover, they are the only developers for most of these files. Usually, component developers involve in few collisions. All these patterns were extracted and listed in section 4.6 (page 82 to 84) with detailed explanations.

Although open-source projects usually last for a longer process, Java files are added into CVS repository since the very beginning. Some of them have already been developed a little bit. In such a new environment, the teams still have batch processing in supportive files and the sequential addition of Java classes. Although open-source development process is advocated as an almost silver-bullet solution, files with high collisions and merges are still exist, together with some file renaming and relocation. Same as student teams, merges still happen ahead of collisions. Because most developers have more experience than those student novices, fewer files are removed from CVS and most members only check out once at the beginning (commend "cvs update" is enough to bring local working directory in sync with repository later). Because people join in open-source projects voluntarily according to their interests, no workload balance has to be kept, some members may have heavily contribution all time while some others only have a checkout and several minor modifications. The work trends of them are still very uneven: small busy phases and spikes scatter among idle periods. In open-source projects, some new roles appear. Multiple team leaders exist in many teams, and some of them are only dominate one or several phases instead of the whole process. There are many anonymous developers who have no right to modify the codes in CVS. Component developers still exist as in student teams. Some team members can be treated as maintainers or patching developers, both of them do not attend the early-phase development. Maintainers usually begin work after several formal releases and only have some minor contribution, such as reformat the code, refactoring, and so on. Patching developers only do some patches instead of other CVS operations. Section 5.4 (from page 113 to 115) elaborated all these patterns related to open-source projects.

There are several dimensions along which CVSChecker could be extended and improved.

- More data could be integrated in the CVSChecker data model, such as code measurements, development profiling information captured by the IDE and PSM-style metrics [48]. A richer set of facts would enable more analyses, at the cost, however, or the unobtrusiveness of the tool.

- The visualization component could be extended and improved. Design more informative visualizations to convey information to instructors and possibly with developers is a short-term future goal.

- More case studies are necessary. Our long-term objective is to systematically conduct case studies, experimenting with different processes, such as XP and RUP for example, different project size, and complexity, and different team-members' competencies. Then we can comparative analyze the collected data to discover which patterns are characteristic of success or failure in the context of which process.

- Finally, as the number of case studies increases, more in-depth data comparisons become possible. We expect that data-mining methods [17] will become viable for extracting significant correlations between successful software development and typical behavior patterns of teams and team members.

Bibliography

[1] D. Atkins. Version Sensitive Editing: Change History as a Programming Tool. Proc. of the 8th Conference on Software Configuration Management, Brussels, July 1998.

[2] T. Ball, J. M. Kim, A. A. Porter, and H. P. Siy. If your version control system could talk... Proc. of ICSE Workshop on Process Modeling and Empirical Studies of Software Engineering, Boston, Massachusetts, USA, 1997.

[3] K. Beck, Extreme Programming Explained: Embrace Change, Addison Wesley, 1999.

[4] M. Belbin. Management Teams – Why they succeed or fail, John Wiley and Sons. New York, 1981.

[5] D. Bisant and J. Lyle. A two-person inspection method to improve programming productivity. IEEE Transactions on Software Engineering, 15(10):1294-1304, Oct.1989.

[6] W. W. Cohen, J. Richman. Learning to Match and Cluster Large High-Dimensional Data Sets For Data IntegrationIn Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), Edmonton, Alberta, Canada 2002.

[7] D.Cubranic and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In Proc. 25th International Conference on Software Engineering (ICSE), pages 408 - 418, Portland, Oregon, May 2003.

[8] T. L. Dickinson, M. Robert. A Conceptual Framework for Teamwork Measurement. In Team Performance Assessment and Measurement: Theory, Methods and Applications. Michael T. Brannick, Eduardo Salas, Carolyn Prince, 1995.

[9] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. Proc. of 10th Working Conference on Reverse Engineering (WCRE 2003), Victoria, British Columbia, Canada, Nov. 2003.

[10] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. Proc. of International Conference on Software Maintenance (ICSM 2003), Amsterdam, Netherlands, Sept. 2003.

[11] H. Gall, K. Hajek and M. Jazayeri. Detection of logical coupling based on product release history. Proc. of International Conference on Software Maintenance (ISCM'98), Washington D.C., USA, Nov. 1998.

[12] H. Gall, M. Jazayeri, and J. Krajewski. CVS Release History Data for Detecting Logical CouplingsProc. of International Workshop on Principles of Software Evolution (IWPSE 2003), pp. 13 -23, Helsinki, Finland, Sept. 2003.

[13] D. German. An empirical study of file-grained software modification. Proc. of the 20th IEEE International Conference on Software Maintenance (ICSM'04) pp. 316-325, Chicago Illinois, USA, Sept. 2004.

[14] D. German. Using software trails to rebuild the evolution of software. Proc. of the International Workshop on Evolution of Large-scale Industrial Software Application (ELISA), Amsterdam, The Netherlands, 2003.

[15] D. German and A. Mockus. Automating the measurement of open source projects. In Proceedings of ICSE '03 Workshop on Open Source Software Engineering, Portland, Oregon, USA, May 2003.

[16] A. G. Gleditsch and P. K. Gjermshus. lrx Cross-Referencing Linux.

[17] J. Han and M. Kamber, Data Mining: Concepts and Techniques, The Morgan Kaufmann Series in Data Management Systems, Jim Gray, Series Editor Morgan Kaufmann Publishers, August 2000.

[18] G. Hedin, L. Bendix, B. Magnusson, Introducing software engineering by means of Extreme Programming, Proceedings of the 25th International Conference on Software Engineering, SESSION: Papers on software engineering education and training: extreme programming, Pages: 586 - 593, Portland, Oregon, 2003.

[19] http://ali.as/devel/cvsmonitor/

[20] http://bloof.sourceforge.net/

[21] http://codestriker.sourceforge.net

[22] http://cvsplot.sourceforge.net/

[23] http://cvssearch.sourceforge.net/

[24] http://cvs.gnome.org/bonsai/cvsqueryform.cgi

[25] http://lxr.sourceforge.net/, Visited Feb. 2004

[26] http://msdn.microsoft.com/vstudio/previous/ssafe/

[27] http://sourceforge.net/projects/viewcvs/

[28] http://statcvs.sourceforge.net/

[29] http://wiki.org/wiki.cgi?WhatIsWiki

[30] http://www.eclipse.org/

[31] http://www.cs.waikato.ac.nz/ ml/weka/arff.html

[32] http://www.freebsd.org/projects/cvsweb.html

[33] http://www.gnu.org/software/cvs/manual/

[34] http://www.postgresql.org/

[35] http://www.tortoisecvs.org/

[36] http://www.weka.net.nz/

[37] http://www-306.ibm.com/software/awdtools/clearcase/

[38] M. Holcombe, M. Gheorghe, F. Macias: Teaching XP for Real: some initial observations and plans, Second International Conference on eXtreme Programming and Flexible Processes in Software Engineering XP, Sardinia, Italy, 2001.

[39] JMetric:http://www.it.swin.edu.au/projects/jmetric/products/jmetric

[40] B. Ralph-Johan, M. Luka, and P. Ivan and P. Viorel. XP as a Framework for Practical Software Engineering Experiments, Proceedings of the Agile Processes in Software Engineering XP2002, Alghero, Sardinia, Italy 2002.

[41] D. Johnson, H. Caristi, J. , Extreme programming and the software design course, in "Extreme Programming Perspectives", Addison-Wesley, chapter 24, pp. 273 - 285, 2003.

[42] T. Kamiya, S. Kusumoto, and K.Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. IEEE Trans. Software Engineering, 28(7): 654-670, July 2002.

[43] S. Koch and G. Schneider, "Results from Software engineering research into Open source development projects using public data," Wirtschaftuniversitat Wien, Austria, Working Paper 22, 2000. http://citeseer.csail.mit.edu/koch00result.html;

[44] P. Kruchten, The Rational Unified Process: An Introduction, Addison-Wesley, 2000.

[45] O. Astrachan, R. Duvall, E. Wallingford. (2003), Bringing extreme programming to the classroom, in "Extreme Programming Perspectives", Addison-Wesley, chapter 21, pp. 237–250.

[46] L. Lopez-Fernandez, G. Robles, M. Jesus, G. Barahona, Applying Social Network Analysis to the Information in CVS Repository, international Workshop on Mining Software Repositories (MSR), 25th May, 2004 Edinburgh, Scotland, UK;

[47] R. C. Martin. Agile Software Development Principles, Patterns, and Practices, Prentice Hall, October 2002.

[48] J. McGarry, D. Card, C. Jones, B. Layman, E. Clark, J. Dean, F. Hall. "Practical Software Measurement - Objective Information for Decision Makers", Addison-Wesley Oct. 2001.

[49] A. Michail. Data mining library reuse patterns in user-selected applications, pp.24-33, Automated Software Engineering. 14th IEEE International Conference, Cocoa Beach, FL, USA ,1999.

[50] A. Michail. Data mining library reuse patterns using generalized association rules, nternational Conference on Software Engineering, Proceedings of the 22nd international conference on Software engineering, Limerick, Ireland, 2000.

[51] K. B. Mierle, K. Laven, T. Sam, Roweis, G. V. Wilson. CVS Data Extraction and Analysis: A Case Study, http://www.cs.toronto.edu/ roweis/papers/cvsanalysis.pdf. [100] M. Michlmayr, "Managing volunteer Activity in Free Software Projects," in Proceedings of the FREENIX Track: 2004 USENET Annual Technical Conference, Boston, MA, June-July 2004.

[52] A. Mockus, R. Fielding, J. Herbsleb, "Two Case Studies Of Open Source Software Development: Apache And Mozilla," ACM Transactions on Software Engineering and Methodology, volume 11, number 3, 2002, pp. 309-346.

[53] M. M. Muller, W. F. Tichy. Case study: Extreme Programming in a University Environment, Proceedings of the 23rd International Conference on Software Engineering, Toronto, Ontario, Canada PP537 - 544, 2001.

[54] A.Mockus, R.T. Fielding, and J.D.Herbsleb. Two case studies of open source software development: Apache and Mozilla. ACM Transactions on Software Engineering and Methodology, 11(3): 309-346, 2002.

[55] J.Noll, Some Observations of Extreme Programming for student Projects, position paper at the workshop on Empirical Evaluation of Agile Processes, Chicago, Illinois, August 7, 2002.

[56] T. Nosek, The case for collaborative programming. Communications of the ACM, Volumn 4, Issue 3, pp. 105-108, ACM Press, Mar. 1998.

[57] M. Ohira, R. Yokomori, M. Sakai et al.. Empirical Project Monitor: A Tool for Mining Multiple Project Data, Proc. of International Workshop on Mining Software Repositories (MSR2004), pp.42-46. May/25/2004, Edinburgh, Scotland, UK.

[58] L. Putnam, W. Myers, Five Core Metrics: The Intelligence behind Successful Software Management, Dorset House, 2003.

[59] E. Raymond. The Cathedral and the Bazaar, Musings on Linux and Open Source by an Accidental Revolutionary, O'Reilly UK 2001.

[60] G. Robles, S. Koch and J. M. Gonzlez-Barahona. Remote analysis and measurement of libre software systems by means of the CVSAnalY tool, Proceedings of the 2nd Remote Analysis of Software Systems (RAMSS) Workshop held at the 26th International Conference on Software Engineering. Edinburgh. May 2004.

[61] F. V. Rysselberghe and S. Demeyer. Mining Version Control Systems for FACs (Frequently Applied Changes), Proc. of International Workshop on Mining Software Repositories (MSR '04), Edinburgh, Scotland, UK, May 2004.

[62] J. Sayyad, C. Lethbridge, Supporting software Maintenance by Mining software update records, Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01), PP. 22, 2001;

[63] J. Sayyad Shirabad, C. Timothy. Lethbridge, S. Matwin, Mining the Maintenance History of a Legacy Software System, 19th IEEE International Conference on Software Maintenance (ICSM'03) pp. 95.

[64] J.-Guy Schneider, L. Johnston. eXtreme Programming at universities: an educational perspective. Proceedings of the 25th International Conference on Software Engineering, SESSION: Papers on software engineering education and training: extreme programming Pages: 594 - 599, Portland, Oregon, 2003.

[65] A. Sillitti, G. Succi, T. Vernazza. Analysis of Source Code Repositories, http://www.unibz.it/web4archiv/objects/pdf/cs_library/2/analysis_of_source_code_repositories.pdf.

[66] G. Snelting. Reengineering of configurations based on mathematical concept analysis. ACM Transactions on Software Engineering and Methodology (TOSEM), 5(2): 146-189, 1996;

[67] M. A. Storey, C. Best, and J. Michaud. SHriMP Views: An Interactive and Customizable Environment for Software Exploration. In Proc. of International Workshop on Program Comprehension, May 2001.

[68] A. Stratton, M. Holcombe, P. Croll. Improving the quality of software engineering courses through university based industrial projects. In Projects in the Computing Curriculum, (eds.) 1998, 47-69.

[69] Subversion: A compelling replacement for CVS. http://subversion.tigris.org/

[70] L. Williams, R. Kessler. Pair Programming Illuminated, Addison-Wesley, 2002.

[71] W.Tichy. Design, Implementation, and Evaluation of a Revision Control System, Proceedings: 6th International Conference on Software Engineering pp. 58-67. IEEE Computer Society Press, 1982.

[72] X. Wu. Visualization of version control information. Master's thesis, University of Victoria, 2003.

[73] L. Williams and Richard Upchurch, Session Eetreme programming for software engineering eduction? Proceedings of 31st ASEE/IEEE Frontiers in Education Conference, Oct 2001, Reno, Nevada, USA.

[74] M. Winter, Developing a group model for student software engineering teams. Master thesis, Univ. of Saskatchewan, 2004.

[75] K. Wong, W. Blanchet, Y. Liu, C. Schofield. E. Stroulia, Z. Xing, JRefleX: Towards supporting small student software teams, Proc. Of Eclipse Technology exchange workshop, pp.56-60, OOPSLA 2003, Oct. 27 2003, Anaheim CA, USA.

[76] J. Pfeiffer. N William, Instrumentation in Human Relations Training: A Guide to 92 Instruments with Wide Application to the Behavioral Sciences Second Edition. University Associates, La Jolla, California. USA, 1976.

[77] A.T.T.Ying. Predicting source code changes by mining revision history. Master's thesis, University of British Columbia, Canada, Oct. 2003.

[78] T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). Proc. of International Workshop on Principles of Software Evolution (IWPSE 2003), pp. 73-83, Helsinki, Finland, Sept. 2003.

[79] T. Zimmermann, P. Weibgerber. Preprocessing CVS data for file-grained Analysis. Proc. of 1st International Workshop on Mining Software Repositories (MSR), Edinburgh, UK, May 2004.

[80] T. Zimmermann, P. Weibgerber, S. Diehl, A. Zeller. Mining Version Histories to Guide Software Changes, Proc. of 26th International Conference on Software Engineering (ICSE), Edinburgh, UK, May 2004.