JRefleX: Towards Supporting Undergraduate Software-Development Team Projects

Eleni Stroulia Warren Blanchet Ying Liu Curtis Schofield Kenny Wong
Zhenchang Xing
Department of Computing Science
University of Alberta
{stroulia,blanchet,yingl,schofiel,kenw,xing}@cs.ualberta.ca

Abstract

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 undegraduate 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 a 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 lifecycle, 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 paper introduces the JREFLEX *environment, which we developed upon the Eclipse framework, to support the reflection on small software teams.*

1. Introduction

The software-engineering research literature abounds with information on how to develop high-quality software, on time and on budget. This knowledge has also made its way to the textbooks used in undergraduate software-engineering courses. However, textbook learning alone is not enough to train competent software professionals; students need to practice and apply their textbook knowledge and to acquire "hands-on" experience with realistic software-development projects.

Recognizing this need, most undergraduate software-engineering programs have included a capstone project course in their curriculum. The intent behind such courses is to provide the students the opportunity to work in a realistic software-development context. Most often, they are organized in small teams, they are given a, possibly incomplete, set of requirements. Then, in the course of an academic term, they have to face the challenges of elucidating the requirements, designing an application to meet them, assigning the various development tasks among them, developing, testing and integrating the various pieces of software and documenting their work, while all along they have to make sure that they stay on schedule.

In this context, instructors have the dual role of the manager and the mentor for all the teams; on one hand, they have to evaluate the team's process and products and, on the other, they have to provide specific and timely feedback on what the team is doing well and what needs to be corrected or improved. Unfortunately, high student-to-instructor ratios make fulfilling these roles a challenge. In our experience, involving roughly 30 small teams in a course of over 120 students, there are often major variations among the team projects and the skills of team members, making the detection of individual problems too subtle. Students may get mired in the complexity of the product or their individual components, and not recognize signs of problems in their overall design or development process early enough to effectively involve the instructor.

This experience was the main motivation behind the JREFLEX project, whose goal is to develop a tool to monitor the collaboration process of software teams and to aid the understanding of how software designs and codes evolve through the project life-cycle. The primary objective of the JREFLEX tool is to infer high-level information about how a team project progresses, including the team's organization style, the impact of each member's contribution or lack thereof, and the evolution of the product's design and code, in order to enable instructors to quickly perceive when and how they need to intervene to support this team. The data to support these needs is gathered unobtrusively as developers work on their code. Visualizations are created and delivered to the instructor so that he can always have an up-to-date view of their progress and make comparisons across teams.

As a secondary objective, we are considering producing team-specific views and making them available to the stu-

dents to monitor themselves and to see how their teams might rank against others in the course. We hypothesize that teams who are aware of their own collaborative process, reflect upon their progress, and make adjustments as needed are more likely to make the right project-management decisions when new challenges arrive.

In the longer term, the JREFLEX environment is to provide an experience repository for the collaborative development processes of a series of projects. Such a repository could be data mined to discover interesting correlations between objectively collected process and product data, subjective developer perceptions of their own work, and their performance as assessed in their project marks.

The JREFLEX environment currently uses CVS (Concurrent Versioning System) to support collaborative software development. The environment accesses the CVS history of a team's software development work on a project. The analysis modules are implemented as Eclipse plugins. As well, JREFLEX provides a Wiki-based user interface to deliver the results of analyses in a web-based format.

The rest of this paper is organized as follows. Section 2 introduces the architecture of JREFLEX. Section 3 discusses our experience with JREFLEX to date and the results of our initial evaluation of the tool. Section 4 places our work in the context of the related literature and Section 5 briefly summarizes the lessons we have learned to date and our plans for future work.

2. The JREFLEX Architecture

The JREFLEX environment consists of five components:

- the development environment (based on Eclipse),
- a repository, in which a set of facts regarding software products is stored,
- the analysis components that process the repository contents to infer high-level information about the progress of the development,
- a browser-accessible wiki server, WikiDev, 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.

2.1. The development environment

With respect to development tools, JREFLEX 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, JREFLEX 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 JREFLEX 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 JREFLEX 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.

2.2. The repository

The repository consists of a CVS, where all development work products are stored, and a database, where work-product meta-data and analysis information about 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 *Quality*.

More specifically, a *CourseTerm* represents a particular group of *Projects* that are being developed for a class project during an academic term. A *Project* represents a particular module or portion of a module within a CVS area, and is associated with the *Team* developing it. In turn, a *Team* is a group of *Developers* who are working together on one or more *Projects*. A *WorkProduct* is part of a *Project*, i.e., a file within the Project's CVS area, that requires constructive effort by a *Developer*. The actual information regarding what a *Developer* has produced is stored as a *Version* of a *WorkProduct*; essentially, a *Version* parallels the notion of a CVS file revision. 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*. An *Activity* describes a particular type of work that *Developers* may do while working on *Projects*, such as for example, planning, design, coding, testing, documentation, etc. A *Quality* describes a particular kind of non-functional requirement that is of interest for a *Project*, which instructors use for product evaluation, such as learnability, usability, and extensibility, for example.

2.3. Collaboration- and evolution- analysis

JREFLEX has two analysis components. The collaboration-analysis component [16] 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 these analysis components are implemented as Eclipse plugins. Visualizations of their results are accessible through specialized Eclipse perspectives and through the WikiDev.

Collaboration Analysis The primary data source for the collaboration-analysis component is the *History* table of the repository database, i.e., information regarding the developers' operations on the project files and modules. The collaboration-analysis daemon examines CVS on a daily basis and populates the database *History*. Based on this data, it then proceeds to calculate a rich set of derived metrics, which are also stored in the repository database. These metrics refer (a) to the team, as a whole, or (b) to a specific team-member, or (c) to a particular work product or (d) to a particular type of CVS operation.

Examination of these metrics can provide interesting insights to the dynamics of a team's collaboration style. For example, it can reveal team members who do not contribute to the evolution of any work product throughout the project life-cycle, or sudden changes to the profile of a member's development behavior. Furthermore, comparative examination of corresponding metrics of different teams may reveal interesting trends and exceptions in the way most of the teams collaborate.

For the first two terms of developing and using JREFLEX, visualizations of these metrics were available only to instructors and teaching assistants, through an instructor-team wiki on WikiDev. We have in the mean time developed a set of team-accessible Wiki pages, where team-specific metrics can be accessed by the students. Furthermore, we have also developed an Eclipse plugin that can be invoked directly from the development environment to display the same visualizations, shown in Figure 1.

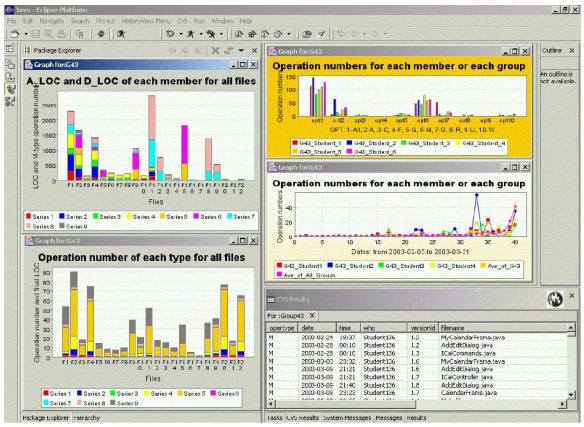

Figure 1. Collaboration-analysis perspective

Evolution Analysis The evolution-analysis component of JREFLEX comprises a suite of methods for analyzing the modifications on the software design from one version to the other, through comparison of class hierarchies. Further analysis of the recovered design-level modifications results in interesting insights regarding the evolution history of the software system under analysis. Comparison of original designs against designs reverse-engineered from code could reveal discrepancies between the designer's intent and the actual implementation. On the other hand, comparison of a sequence of reverse-engineered designs, corresponding to a sequence of software-system versions, could recover the evolution profile of individual application classes, identify transformations brought about by refactoring, and characterize the nature of evolution of the application design.

The main input for the evolution-analysis plugin is a sequence of design models, represented in UML (XMI 1.3), corresponding to a sequence of snapshots of an object-oriented application, generated by regular checkouts from the CVS repository for a project. These UML models are currently reverse engineered from the application code, using a round-trip engineering tool. The core of the plugin relies on recovering the structural design changes from one version to the next. That is, the plugin implements a UML differencing algorithm that can surface structural modifications to the application classes and interfaces, their attributes, their methods, and their specialization-generalization relations in terms of the additions, deletions, moves, and renamings of object-oriented entities. The algorithm produces *change trees* that report the deltas of the compared versions.

Aggregate information is then extracted from a sequence of such change trees. By examining and analyzing the change trees and the aggregate data, we can obtain a quick overview of the whole application evolution history. In particular, we can recover the overall software evolution history at three different levels:

- At the system level, we can identify different evolution phases, such as functionality extensions vs. refactorings, and through the application evolution history.
- At the class level, we can recognize different types of classes according to their evolution profiles, such as continuously modified classes vs. legacy classes.
- At the change-tree level we can identify various change patterns, such as co-evolution and refactorings.

Finally, the collected design-evolution information can be visualized to present different views of the application evolution to the interested developers. The students or instructors using the evolution-analysis plugin start with a set of XMI models of a software system. They can analyze any two versions, or run the plugin incrementally. The plugin reads in the XMI models, parses their class-hierarchy trees, applies the UMLDiff algorithm against these trees and saves the deltas into change trees, and finally extracts and analyzes the aggregate information. The results of the analysis are then visualized in the various views of the plugin organized in an Eclipse perspective, shown in Figure 2.

The change trees are shown in a tree view in the top-left corner, the *Change Tree* view. This view works similarly to the navigator pane of many IDEs. The different icons represent the different object-oriented entities, the top right adornments show the modifiers of the object, for example, "abstract", "static" etc. The bottom-right adornments represent the different types of changes of particular object, such

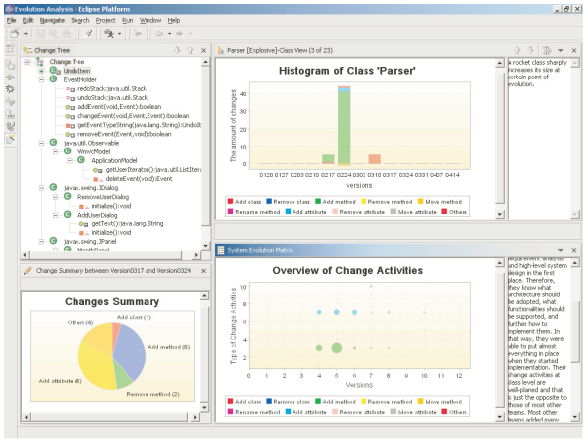

Figure 2. Evolution-analysis perspective

as plus sign for “insert”, minus sign for “delete”, filled triangle for “rename”, empty triangle for “change signature”, arrow with minus sign for “move source”, arrow with plus sign for “move target”. The tree view presents the developers the detailed structural modifications to the class model of software system. The user can expand or collapse tree to see more information. To look into the source codes of a specific element, one can double click on the element to bring out the java-source editor, shown in the top-left corner. To inspect previous or next change trees, one has to click the arrow button to move backward or forward.

The bottom-left corner, the *change summary* view, shows a pie chart that summarizes the amount of different types of changes and their ratios from one version to the next. It complements the matrix and the histogram views with actual number and ratio of changes.

The bottom-right corner stacks three views, the *system evolution* view, the *class evolution* view, and the *refactoring* view. The system-evolution and the class-evolution views can be toggled to show the evolution matrix or the histogram of the software system or individual application classes. Each column in the evolution matrix, shown to the bottom right of the Figure 2, represents a version of the software, while each row represents the different types of changes. The area of the bubble represents the amount of such types of changes. Thus, a bubble of size s at the (x,y) point in the matrix indicates that s number of changes of type y happened between version x-1 and x. The histogram - one is reported in Figure 10 - depicts the change profile of a system or individual classes. It is a color stacking bar chart. The horizontal axis of the histogram represents the versions of the software system, while the vertical axis represents the amount of change. The different colors represent the different types of changes. “Delete”-type changes have negative values and all others positive values. Both the matrix and the histogram provide a good way to visualize the high-level evolution information of a software system, while the detailed information about what happened in a particular version can be obtained from the change-tree and the change-summary views. An editable comment view can be toggled to let the users input any information they may want to note about the evolution of system or individual classes.

Furthermore, the plugin core analyzes the change profiles of individual classes, classifies them into one or more of the evolution types that are shown in the class evolution view’s title, and add the default comments on evolution types as shown in the right-hand side editable comment window. The query mechanism is implemented to allow the users to select and show the classes of evolution types they are interested in. The plugin also identify co-evolving classes of individual classes. The users can select one of them from the drop down menu of the class evolution view to show the evolution information of that class, if any.

The refactoring view shown in Figure 11 side a list of all the identified refactorings that have been made in a particular version. The right-hand side is a tree view that displays the snippet of the change trees corresponding to the selected refactoring.

2.4. 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/cgi/wiki` 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 *ProjectView* 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.

2.5. 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. Cur-

rently, 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, such as PSP/TSP-related information for example [14], 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. Evaluation

JREFLEX was partially[1] deployed during a single-term, third-year undergraduate course in software engineering. At that time, students were required to use CVS and they were also given the first version of WikiDev; at the same time, the analysis components were under development. At the end of the term, the collected data was analyzed by researchers, largely independent with the course delivery. The objective of this case study was to examine, to what extent the analysis components of JREFLEX can provide insightful information to the instructors, so that they can provide timely and relevant feedback to the students. The content of the study and the results of the analysis components are described in the subsequent section.

3.1. Team and Project-deliverable structure

Our case study involved 85 students organized in 23 teams. 51 students (including the team members of 5 teams) gave us permission to use their data; henceforth we will refer to these teams as team *A*, *B*, *C*, *D* and *E*. 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 total duration of the project was 55 days, organized in three 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, were due. At the end of the second cycle, a working horizontal prototype was due, exhibiting the interactive functionalities of the project but not necessarily the underlying support functions. Finally, the whole working project was delivered at the end of the third cycle.

On each due date, each team member had to submit an electronic evaluation form to assess the contribution of all team members, including themselves. Note that, at the time of our study, the data of these self-assessment forms were not integrated in the JREFLEX repository; in the mean time, we have reverse engineered the content of the collected forms and we have populated the assessment aspect of the repository, in order to correlate this data with the information inferred by the analysis components.

3.2. Collaboration Analysis

We used the collaboration-analysis component to extract information regarding the following CVS operations in the database *History*:

- Out: a record type from operation "checkout"; A CVS file is checked out from the CVS repository to a working directory.
- DRel: A directory in CVS is released. It has the same effect as direct working-directory deletion, but DRel avoids the risk of losing changes, which users may have forgotten. Three types of records resulting from the operation "commit":
- Add: A file is added to CVS, and the first revision for this file is created.
- Mod: A file is modified and a new revision appears.
- Rem: A file is removed from the CVS repository. Three types of records resulting from the operation "update":
- Col: More than one user checked out and modified the same file version, so a collision occurred when a second update was attempted and a manual merge is required.
- Mrg: Two versions need to be merged and the merge is successful automatically.

[1] The repository and the WikiDev components were available to students, they did not use Eclipse though.

- Wdel: A working copy of a file was deleted during update, because it had already been removed from the repository.

Based on this data, the subsequent metrics about the team, the developers, the project files and their versions was inferred.

The Team aspect 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. Team *E* performed on the average the smallest number of operations in CVS – half the number of operations of team *B* – but it has many addition, *Add*, and checkout, *Out*, operations and the number of files they developed was bigger than average. Furthermore, they 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: on the average, every file was modified only 5.2 times by the all four team members. Let us now look at some inferences the instructor might draw by examining the operations that the teams performed in their CVS repositories. If a team exhibits "abnormal" numbers of operations of all types – i.e., their CVS usage or file numbers are distinctly different than most other teams – then the instructor may examine their collaboration in more detail to evaluate whether they 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.

As another example, we noticed that teams *A* and *B* had a slightly high number of collisions, *Col*. A collision occurs when more than one team members attempt modify a same file at the same time. A substantially high number of *Col* operations could indicate that the design of the software product and the distribution of tasks among the team members are poor, and the project modularization should be reconsidered. We also noticed that these two teams have high number of collisions 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 should commit new versions back to CVS repository promptly after their modification and should 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 were followed, the average number of modifications, *Mod*, should not be very small. 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, *Col*.

Integrating the above heuristics, we can say that the higher the ratio of successful merges over collisions (*Mrg*/*Col*) 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. The ratio *Mrg*/*Col* of team *C* was the highest, where the same metric for team *A* was the lowest. The problem of team *A* seemed to be the small number of files in which they 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. Figure 3 diagrammatically presents the average workloads for the students of the five teams through the whole project cycle day by day. It is easy to see that all teams show peaks of activity around the same dates in the second and third periods. However, there are some interesting differences too. Team C began earlier than the other teams, team *B* usually worked in a single day then stopped for the next several days, and team *D* followed a much more consistent work profile than the rest. With this figure, the instructor might have noticed that a team has not started development, when most other teams have, and might have given the "delayed" team a prompt reminder.

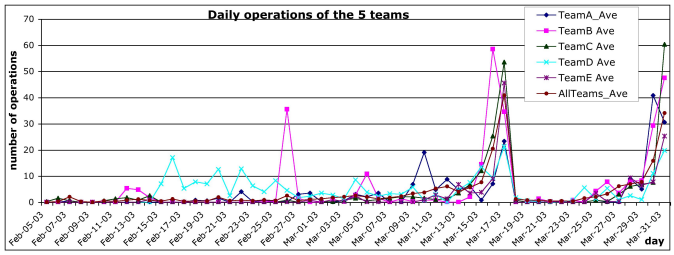

Figure 3. Number of operations by date

The Individual-Developer aspect This type of analysis is intended to support the instructor in assessing the relative contribution of each team member to the project and to notice quickly imbalances in the workload distribution.

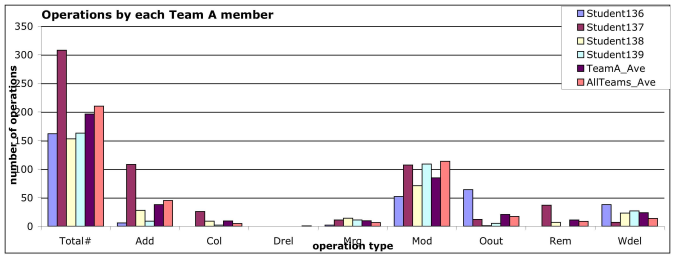

Figure 4. Number of each operation type of Team *A* members

Figure 4 shows the operation distribution over all types of each member in team *A*. It seems to indicate that Student137 did much more work than the other members of team *A*, because he performed many operations in CVS. However, the

number of his modification operations was 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. A plausible inference based on this diagram might have been that Student137 is the team leader who designs the project classes and initially authors the files for other members. Compared to the other members of team *A*, Student139 exhibited a better operation pattern: high number of modifications, few collisions and high ratio of successful merges over collisions – *Mrg*/*Col*.

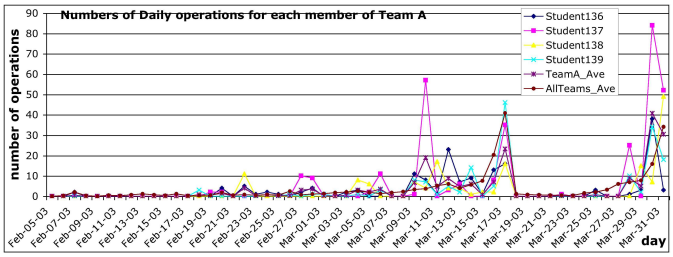

Figure 5. Number of team *A*'s daily operations

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. Figure 5 clearly shows the busy (not busy) periods of the team and enables us to have a quick idea about the typical GAPs of each member along the time-line. In this manner, we can identify when is the most important period of activity for the entire project, or for a particular person, or for a file, or for a particular operation type.

From Figure 5 we notice large any-type-operation gaps between the beginning of the project to February 17 and from March 18 to March 24. We also notice that the average team activity is fairly consistent for all the teams: most operations occur in phase2 and at the end of phase3: around the two major deliverables of the project. All four members of team *A* appear to have similar operation frequency and distribution except for student137, who was much more active around Mar10 and Mar30. This diagram provides counter-evidence for our earlier hypothesis regarding the leadership role of student137, which indicates that more accurate analysis results come from multilevel data. Student137 did not start earlier than the other team members, 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 contributed to the team.

Figure 6 shows the work pattern of teams *C* and *D*. It seems that all members of team *C* only worked just before the deadlines. On the other hand, team *D* is much better; all its members started earlier than average and worked consistently almost every day.

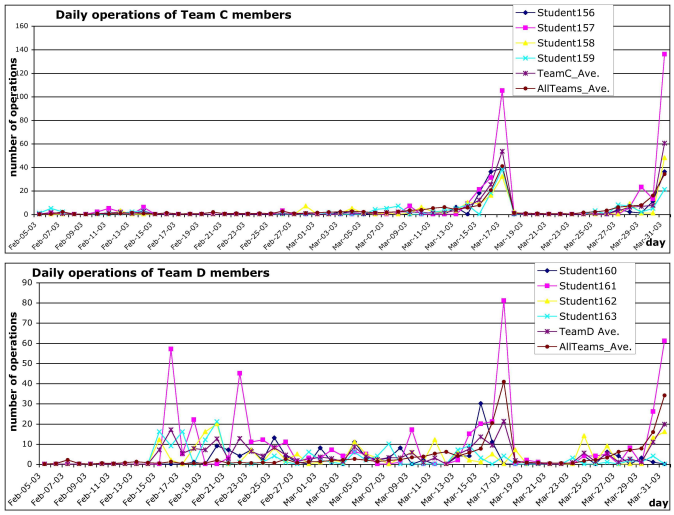

Figure 6. Number of daily operations of teams *C* and *D*

The File aspect Let us now examine how the project workload was distributed across the files. The JREFLEX analysis component produces two diagrams for each team to show the file-related information: Figure 7 shows the number of the different types of operations performed on each file of the team *A* project. From the height of columns, we can have a quick idea about which files suffered many operations; the height of each colored section corresponds to the number of operations of a particular type.

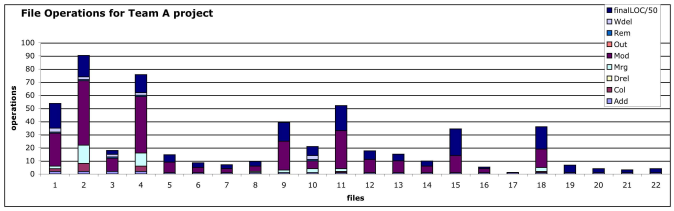

Figure 7. Numbers of team *A*'s operations on each file

Besides providing an overview of the number of modifications, *Mod*, Figure 8 enables a deeper view into this information, presenting the numbers of LOC added to and deleted from each file by each team member. The more subbars 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.

Focusing on team *A*, from Figure 7 the instructor might infer that students spent more operations on files 2, 4, 1, 11, 9, 18, and 15 (in that order). Their numbers of modifications, *Mod*, differ but their sizes are comparable (see FinalLOC). However, it is not always the case that a file that is modified many times is also modified substantially. Consider for ex-

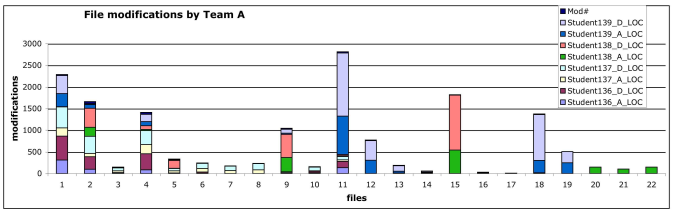

Figure 8. Mod_LOC and Mod numbers of each file in Team *A*

ample files 2 and 11: file 2 has had many more modifications than file 11, however, the eventual sizes are almost the same, and Figure 8 tells us that the total number of modified lines of file 2 is much less than that of file 11.

Combining the information from the above figures one can see that the files with the highest density of modifications, i.e., high ratio of modified lines per total lines of code, such as files 11, 2 and 4, were touched by multiple team members. Furthermore, and not surprisingly, most files that have been modified by a single team member have less number of collisions, larger ratio of total LOC per modification number and smaller ratio of modifications per number of modified LOC. These three pieces of evidence seem to imply that when a team member is the "owner" of a file, i.e., he is its 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 high frequency and large numbers of modified lines may be 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. So if a team has a large number of collisions, the instructor might suggest to students to inspect the multiply modified files and see whether they can be re-designed or whether their maintenance should be assigned to a single person.

3.3. Design-Evolution analysis

Evolution analysis was performed on weekly snapshots of the teams' projects from their CVS repositories, from January 20th, 2003 through April 14th, 2003, resulting in 13 versions for each project.

The evolution matrices - those of teams *B* and *C* are shown in Figure 9 - revealed some interesting insights regarding the evolution style of their development. Based on evolution-phase analysis, we discovered that teams *A* and *E* defined a few classes in the first place, and proceeded to develop them one step at a time. Their change activities involved continuous small modifications. Another characteristic of these two teams was that major changes were made in the middle of

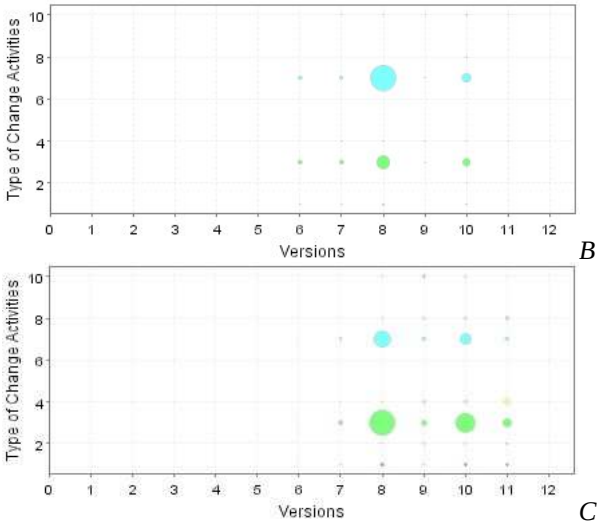

Figure 9. Evolution matrices for teams *B*, *C*

their project development, mainly between weeks 7 and 10. They did not try to implement their project at the last minute, like teams *B* and *C*.

The evolution processes of teams *B* and *C* contained two versions with aggressive growth spurts. Their projects started with a few classes, and did not change a lot until week 7. However, there was a sharp increase in the size of their projects at week 8, which is followed by small changes until week 10, in which another growth spurt is observed. These occasional large modifications coincide with the deadlines for project part 2 and part 3. This means that most features and/or functionalities of their projects were implemented just before the deadline - a bad but not untypical practice.

Teams *B* and *C* exhibited similar evolutionary development styles. But, since team *C* adopted the MVC model as the application architecture, their work is more organized than that of team *B*, and their project quality as evaluated by the course TA was better. This result validates the intuition that good architecture enables software quality. Team *B* combined the application and user-interface objects together. As a result, as the various views evolved to meet usability requirements, the model design was also affected. Furthermore, team *B* created a giant class "Entry", shown in Figure 10, to contain all possible back-end objects of the application and the application logic associated with them, like how to recognize conflicts for example. Note that this observation is consistent with the of its high number of collisions, observed during collaboration analysis.

Team *D* exhibited a very interesting evolution style. The most changes were made within the first two consecutive versions when they started the development of their project, in weeks 4 and 5. They may have developed a very good requirement analysis and high-level system design in the first place. Therefore, they seem to know what architecture should be adopted, what functionalities should be sup-

ported, and further how to implement them. In that way, they were able to put almost everything in place when they started implementation. Actually, they obtained the best mark for the first deliverable which is essentially a requirements-and-design document, Their change activities at the class level are well-planed and that is just the opposite to those of most other teams. Most other teams added many new classes when the project deadline was approaching in week 10. Team *D* just added a few things, but the most remarkable thing for team *D* at week 10 is that they moved some methods among classes, which means that, at the end, they were trying to improve the quality of the system structure, when most other teams were still struggling to meet their requirements.

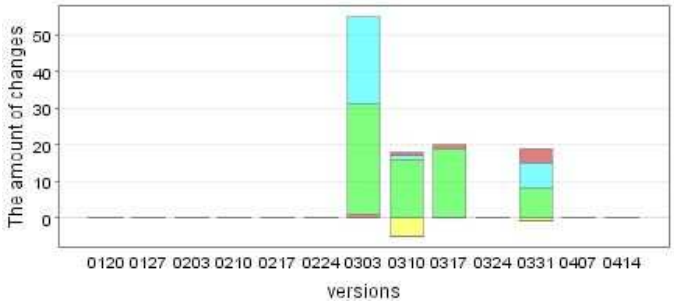

Figure 10. Histogram of a "giant" class

In these five projects, we were able to find instances of all the class-evolution types but die-hard and legacy, such as the one reported in Figure 10. We believe that the reason is the nature of the undergraduate term projects. They are relatively small and must be completed within about 3 months. The structure of system is simple, and thus it does not need such maintenance activities that bring about die-hard and legacy classes. On the other hand, due to time constraints students aim at completing a working system and are usually unwilling to perform such maintenance activities. However, we found evidence of refactoring. For example, as shown in Figure 11 taken on week 11, team *E* created a utility class named "DateWorker" and date-related functionality was moved from the pre-existing "Appointment" class to the new "DateWorker" class. This is an example of the "class extraction" refactoring.

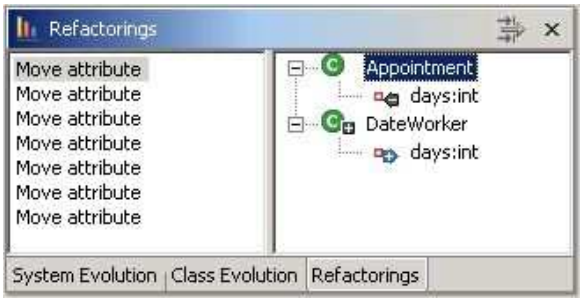

Figure 11. A snippet of the change tree for a "move field" refactoring

3.4. WikiDev Usage

During this case study 2 of the 5 teams utilized WikiDev in an interesting manner. Team *B* used the WikiDev plugins most extensively and team *E* used the Wiki features primarily with only a small focus on the WikiDev plugins. The "ideal" usage scenario that we had hoped for would be the union of these two usage patterns. Team *B* constructed a TaskLog in their wiki and used the ability to assign a task on a file to a particular member, in order to better coordinate their development of their project. They also used the main page as a reference to the most interesting pieces of code in their project. Team *E* created their own wiki pages to keep track of various portions of the design, coding, and documentation of their system. They also assigned tasks to the team members, maintained group notes regarding their project, and they also created a page to describe their understanding of an application framework that they utilized during the development of their project. Finally, they also constructed a page to maintain contact information for the team members.

To encourage significant use of WikiDev, similar to the behavior of team *E*, the current WikiDev version has added the ability to annotate WorkProduct pages (these pages were originally locked). As well, in the future, Bugzilla integration is intended to replace the TaskLog functionality of the first WikiDev version to extend the small feature set provide by the TaskLog.

4. Related Research

Ever since Osterweil pointed out the importance of software process [18], a substantial body of work was devoted to developing Process-Centered Software Engineering Environments (PSEEs) to support developers in their tasks. However, most of them focus on mature developers who are already experienced with process models, and focus on enabling them to visualize their work products and to communicate with one another [13, 3]. At the same time, there has also been a lot of work on process measurement [17, 2], without however, going as far as to provide feedback about how to better manage and control the process in response to these measurements.

Our research with JREFLEX aims at integrating results in software process research within a pedagogical framework: our objective is to support instructors in their effort to mentor novices in their apprenticeship for becoming the future expert developers. To that end, JREFLEX enables implicit and unobtrusive, as well as explicit, data collection in order to compile a precise snapshot of the project progress status. The novelty of its collaboration-analysis capability however lies in the fact that, in addition to data collection, it also employs KDD methods [12] in an effort top discover interesting correlations, patterns and trends that may enable the instructor to gain further intuitions into the students process.

On the other hand, JREFLEX's evolution-analysis component is designed to produce semantically rich reports on the

project's evolution. There already exists a substantial body of literature on the subject of "software-evolution understanding". A vast majority focuses on analyzing, not the system design, but its code metrics. Eick et al. [7] analyze the change history of the code to derive "Code-Decay Indices"; fault potential and change effort is predicted as a function of these indices through regression analysis. The same team also developed metrics visualization tools [8]. Gall et al. [11] use information in the release history of a system to uncover logical coupling among modules. Their method aims mainly at understanding module co-evolution. However, unfortunately, such documentation is not always readily available; even worse, even when such documentation exists, it is seldom kept in sync with the code modifications, and therefore it is an unreliable source of system changes. Demeyer et al. [5] define four heuristics based on code-size and inheritance metrics to hypothesize generic classes of refactoring activity; unfortunately, no concrete refactorings, i.e., the ones described in [10], are identified. Lanza [15] describes how to use a simple two-dimensional graph to convey the implicit information of software metrics of object-oriented entities.

There has also been some work on "software design understanding". Emden et al. [9] present a tool for detecting and visualizing code smells based on the analysis of extracted facts of program structure. Egyed [6] has investigated rule, constraint based transformation and comparison approach for consistency checking between UML diagrams when developers add new information to system model or modify existing ones.

UMLDiff is a special case of a tree-matching algorithm. The general tree-to-tree correction problem has been studied extensively [4], and has been applied to show differences between XML data [1]. The major difference between these general algorithms and UMLDiff is, UMLDiff takes into account the structural syntactic information contained in the class model of application, and it can identify the "move" of object-oriented entities, which enable us to identify perfective changes that cannot be identified from documentation like revision archives.

5. Conclusion

In this paper, we described JREFLEX, a tool for monitoring and analyzing the collaboration process of novice software teams and the design of the produced software system. JREFLEX's analysis is intended to support instructors of capstone-project software-engineering courses to better understand the progress of their students, in order to enable them to provide timely and relevant feedback.

Our work on JREFLEX is still in progress, and much research remains to properly test its effectiveness. Nevertheless, in our first case study, we have collected some promising experiences on how such an environment could be deployed. JREFLEX extract a rich amount of data and its visualizations enabled even people not directly involved in the course to quickly obtain a high-level picture of how the students worked. In the future, we plan to work towards better heuristics for analyzing the data and more intuitive visualizations of the inferences.

This work was supported by CSER, the Consortium for Software Engineering Research, and an IBM Eclipse Innovation Grant.

References

[1] Mosell EDM Ltd.: `http://www.deltaxml.com`.
[2] StatCvs: `http://statcvs.sourceforge.net/`.
[3] V. Ambriola, R. Conradi, A. Fuggetta, Assessing Process-centered Software Engineering Environments, ACM Transactions on Software Engineering and Methodology, 6(3):283 328, 1997.
[4] D. Barnard, G. Clarke and N. Duncan, Tree-to-tree Correction for Document Trees, Technical Report 95- 375, Queen's University, January 1995.
[5] S. Demeyer, S. Ducasse, and O. Nierstrasz, Finding refactorings via change metrics, ACM SIGPLAN notices, 2000, 35(10):166-177
[6] A. Egyed, Scalable Consistency Checking between Diagrams - The VIEWINTEGRA Approach, Proceedings of the 16th IEEE International Diego, USA, 2001, pp. 387.
[7] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, Does Code Decay? Assessing the Evidence from Change Management Data, IEEE Transactions on Software Engineering, 2001, 27(1):1-12.
[8] S. G. Eick, T.L. Graves, A.F. Karr, A. Mockus, and P. Schuster, Visualizing Software Changes, Software Engineering, 2002, 28(4):396-412.
[9] E. V. Emden and L. Moonen, Java Quality Assurance by Detecting Code Smells, Proceedings of 9th Working Conference on Reverse Engineering, Oct, 2002.
[10] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.
[11] H. Gall, K. Hajek and M. Jazayeri, Detection of Logical Coupling Based on Product Release History, Proceedings of the International Conference on Software Maintenance, Bethesda, Washington DC, November 1998.
[12] J. Han, M. Kamber, Data Mining: Concepts and Techniques, Morgan Kaufmann, 2000.
[13] J. D. Herbdlrb, A. Herbdlrb, T. A. Finholt, R. E. Grinter, An Empirical study of Global software development: distance and speed, 23rd Int. Conference on Software Engineering (ICSE 2001), Toronto, Canada, May12-19, 2001.
[14] W. Humphrey, PSP/TSP, `http://www.sei.cmu.edu/tsp/watts-bio.html`.
[15] M. Lanza, The Evolution Matrix: Recovering Software Evolution using Software Visualization Techniques, Proceedings of International Workshop on Principles of Software Evolution, 2001.
[16] Y. Liu, E. Stroulia. A Lightweight Project-Management Environment for Small Novice Teams. ACSE 2003: 3rd Int. Workshop on Adoption-Centric Software Engineering in the 25th Int. Conference on Software Engineering, Portland, Oregon, USA, May 9, 2003.
[17] C. Lott, Technology trends survey: Measurement Support in Software Engineering Environments. Int. Journal of Software Engineering and Knowledge Engineering, 4(3), Sep. 1994.
[18] L. Osterweil, Software Processes are Software too. Proceedings of the 9th Int. Conference on Software Engineering, pp. 2-13, Monterey, CA, Mar. 1987.