Teaching Evolution of Open-Source Projects in Software Engineering Courses

Joseph Buchta, Maksym Petrenko, Denys Poshyvanyk, Václav Rajlich
Department of Computer Science
Wayne State University
Detroit, Michigan USA 48202
{JBuchta, max, denys, Rajlich}@wayne.edu

Abstract

In the traditional software engineering courses, the students develop small programs from scratch. This does not correspond to industry practice where programmers spend most of their time evolving medium to large systems. In order to narrow this gap, we developed a course where students practice software evolution through the implementation of change requests on medium-sized open-source software systems. The results of the course show that this type of software engineering course gives students a more realistic experience than traditional software engineering courses. In the survey at the end of the course, the students expressed a higher level of satisfaction with both rating the course and assessing how much they learned. Additionally, the resources required by such a course are not excessive.

1. Introduction

Traditionally, software engineering course projects have used the waterfall model, while more recently enterprising instructors have implemented projects utilizing software evolution and Extreme Programming (XP) [1]. These projects are either small in scale and expect the student to follow predefined steps already undertaken by experienced software engineers [2, 3], or use medium sized programs in laboratory sessions closely supervised by experienced programmers [4-6]. These courses often require a commitment of substantial resources.

Our traditional course Advanced Software Engineering (CSC6110) taught at the Department of Computer Science at Wayne State University is a graduate level course. This course is presented annually to anywhere from 10 to 20 senior undergraduates and first year graduate students.

In the past, the course project was a traditional project implemented and completed in three phases by teams of 3-4 students. Each phase consisted of certain activities designed to guide the student through the waterfall development model. The resulting "toy" program usually only contained few classes, the most rudimentary features, and a low quality of code.

This traditional approach has outlived its usefulness for several reasons. First, the typical projects given to the students did not match the realities experienced by industry programmers, who deal with much larger systems that contain many features and the code quality expectations are much higher. Also, in industry, seventy or more percent of time is devoted to the evolution or maintenance of large programs [7, 8] rather than new development.

Secondly, the arrangement of teams has promoted abuse of the system in which few motivated students carry out the work load on behalf of the less motivated students who contribute little or nothing to the success of the project. This abuse is often invisible to the instructor because of the false solidarity among the students.

This paper describes our solution to these problems by using the process of software evolution in open-source projects. Software evolution is the process of adding additional functionality to an existing software system as a result of changing business or customer needs [3]. The use of open-source projects guarantees that the students will have an experience with a software system of realistic size and complexity. It is our assertion that a project of this nature prepares a student better for their future software engineering career.

Section 2 presents a discussion regarding previous approaches to teaching software engineering courses. Section 3 provides an overview of software evolution, while section 4 presents the framework for the course project. Section 5 contains a discussion of the required resources and both student and manager experience. Lastly, section 6 presents conclusions and future goals.

2. Related Work

While software engineering instructors generally agree with the inclusion of a team project in software engineering education, the implementation of such a project is still a matter of debate. Sommerville [8] pointed out that the critical goals of software engineering projects are to make students practice "programming in the large", to let them practice concepts learned in the classroom, and to learn how to use software tools. Gnatz et al. [9] offers non-technical goals: understanding the customer's domain and requirements, working in a team, organizing the division of work, and coping with time and pressure.

Postema, Miller, and Dick [2, 3] propose to focus the projects on the four main software maintenance activities: corrective, adaptive, perfective, and preventative. The students work in groups of two on each of the activities using a medium sized, open-source project. At each step, the students are given directions; the instructors then individually verify the students' work.

In the first stage students find errors in the program. In the next stage they fix the errors and review the changes of other groups. The project concludes with the implementation of a graphic user interface as a front end to the software. Hence the beginning stages focus largely on bug correction and the project concludes with introducing new functionality to the system.

The authors provide an introduction to software maintenance through a well structured program, where the solutions to each stage have previously been implemented. The usage of Computer Aided Software Engineering (CASE) tools and/or Concurrent Versioning Systems (CVS) is mentioned as a large expense that cannot be undertaken.

Gnatz et al. [9] presented a software engineering project, where students interacted with a real customer on a fictional project. The course consisted of three teams of four members developing a new project via a waterfall-like approach: requirements specification, design, development of the first increment including testing and delivery, and development of the second increment also including testing and delivery.

The course required five supervisors and an industry partner. Even with all this personnel, the course ran behind schedule and the authors did not have a chance to teach software maintenance issues.

Hazzan and Dubinsky [5] report on a course project that followed all twelve core practices of XP that include planning, small releases, system metaphor, simple design, continuous testing, refactoring, pair programming, collective code ownership, continuous integration, 40 hour work week, on-site customer, and coding standards. The students in the course participated in studio exercises in which all XP practices can be tightly controlled and reviewed by a designated tutor or instructor.

In the student evaluations, the authors assumed that other students would not allow "parasites" on the team. Whenever this assumption proved false, the project manager or instructor acted temporarily as the partner for the students in question and assessed their capabilities in this way.

Hedin et al. [10] used pair programming to teach a second year undergraduate course to 107 students in a closed lab setting. Students were graded pass/fail depending on their perceived participation.

Some authors raised concerns over the grading schemas used in software engineering projects. Hayes et al. [4] recognized that students often hide in groups and will attempt to circumvent the grading scheme. Solutions include mandating that students post daily internet work logs, meet frequently with course instructors, or evaluate each other during the course of the project. Each of these seeks to mitigate the risk of freeloading students, but often fail when it comes to the collusion of student teams.

Each of these authors presented a partial solution to the challenge of bringing a realistic software development environment to an academic environment, but the full attainment of this goal still requires further effort.

3. Software Evolution

Software evolution consists of the processes that change existing software in order to meet changing customer requirements [11]. One such process is incremental change (IC) [12]. While there may be several versions of the IC process, we have decided to base our course on the IC process proposed by Rajlich and Gosavi [12] with some adaptations. In their paper, IC activities are classified into four main groups: initiation, design, implementation, and testing.

IC initiation begins when a customer or other project stakeholder requests new functionality to be incorporated into an existing system. This can be the result of a bug or a problem report, which requires certain functionality to be corrected. It can also occur when the customer requires a completely new functionality resulting from changing business or technological needs. In either case, the change request contains relevant information, such as the functionality required and/or program logs and traces. In open-source software, change requests are often listed in a "wish list" [13-15]. The change request is received by the software team and is assigned a priority.

IC design is the response by the software engineering team to a change request. It includes concept extraction, location, and impact analysis. Each of these activities is undertaken before the actual change is introduced into the system.

Concepts appear as nouns, verbs, or short clauses in the change request; often, there are several concepts embodied within a single change request. Concepts also appear within the source code [16]. Concept extraction extracts the most important concept related to the change. Concept location is the process of locating the implementations of this concept in the code [11, 16, 17].

There are several different methodologies available for concept location, including string pattern matching, information retrieval techniques [18], and static dependency search [19]; a more complete explanation of these techniques can be found in [17]. Still, other methodologies include dynamic techniques [20-22], formal concept analysis [23], use of CVS repository comments [24], or combined techniques [25].

In our course, the students were required to use the static dependency search technique [17, 19]. This approach calls for the programmer to traverse the program dependencies in a manner similar to that of depth-first search, but the search is conducted by the programmer instead of a computer.

In doing so, the programmer is required to follow the static program dependencies contained within the source code and make decisions that guide the search. For example in Java, class A depends on class B if class A refers to class B as a data member, local variable, argument, or data cast; if class A inherits from class B; or if class A implements the interface of class B. Class A is called a *dependent class* and class B is called a *supporting class.*

The static dependency search usually begins at the top class containing the main () or init () function. If the concept is not implemented there, the programmer must determine which of supporting classes leads to it. If none of the supporting classes lead to the desired concept, then a wrong turn must have been taken, the search backtracks to the previous class, and a new dependency is chosen. This proceeds until the concept is located within the program's classes.

Impact analysis determines a change's extent by determining the strategy for the change and the set of classes IC affects [26, 27]. The outcome of the analysis is a list of the relevant classes involved in the change.

The next activities of IC implement the desired change. These activities are pre-factoring, actualization, incorporation, change propagation, and post-factoring.

Refactoring restructures the architecture of a program without changing its functionality [28]. Pre-factoring is an opportunistic refactoring that takes place before the change proper. Its purpose is to localize the code affected by IC before the actual change is implemented.

Actualization produces the code that implements the new functionality required by the IC. In the case of a small change, the new code may be a part of an existing class. A larger change may require one or more new classes to be designed and implemented.

Incorporation is the activity of interconnecting the new code with the old system. This can be achieved by declaring new instances of the new classes within the old code.

After the new code has been introduced into the software system, the system may or may not become inconsistent. In order to return the system back to a consistent state, change propagation changes all components in a system that have become inconsistent. This activity is similar to impact analysis except that actual code changes take place [29].

Post-factoring refactors the new and old code in order to make the system clear and understandable to future software engineers.

Testing includes both unit testing and function testing. The aim is to ensure that all of the new functionality is correct and old functionality remains intact after a change has taken place (through regression testing).

Lastly, a build of the updated software is completed.

4. Software Evolution Course Projects

We choose to base our course on software evolution and IC for several reasons: IC is intuitive and easy to teach; we explained the theory of IC in approximately 10 hours of lecture time at the beginning of the semester. Change requests can be decomposed into several parts and distributed among the team members for later integration, giving the participants a cooperative experience. Last but not least, large software systems can be used since the students do not need to understand the entire code in order to implement a change [17].

In order to emulate an industry experience, students coordinated their efforts through a Concurrent Versioning System (CVS) [30, 31] maintained by the instructors. This encouraged the students to do their own work, but also to cooperate with others in order to clarify and structure their changes appropriately.

The CVS system used for the course was on a Solaris UNIX server maintained by the department. The initial setup consisted of creating student accounts

Table 1. JCDSee Change Requests.

Implement a right-click Context menu feature for picture objects (e.g. for thumbnails, preview picture and full-screen picture)
Implement the ability to launch plug-ins from within the application. Provide a consistent framework for doing so.
Augment the Context menu to allow the Context menu to display the full image.
Allow the user to zoom in on a picture within the Context menu.
Automatically refresh the Context menu screen.
Allow the user to view a slideshow of their photos. Allow the user to select it from the View menu.
Allow the user to view web pages from a viewer located within the application.
Add sorting capability to the View menu. Some sorting options include date, size, image type, and file name.
Integrate an open-source "Album Central" project into the JCDSee application.
Add an "Add to the favorites" option to the Context menu of directory objects (currently the program only has "browse", "slide show", "delete", "view" entries in it).
Implement blur, rotate, sharpen, and resize plug-ins using the previously developed plug-in framework.
JCDSee can call plug-ins only from preview mode. Modify JCDSee so we can call plug-in functions from the full-screen Context menu also.
Modify the sort functionality so that it can sort images in title mode also. Add check boxes ahead of sorting options so the user can see what the current sorting mode is (check box should be displayed only ahead of current sorting option).
MS Windows has files with .URL extensions. These types of files store links to web-pages. Allow the user to view pictures of these web pages the same way as it was done for the local directories.
Implement the ability to auto-size in full-screen mode so that the user will see the picture in the size matching the screen size when it is opened in full-screen mode.

for the CVS login and setting the proper permissions. After this, several CVS clients were evaluated: Eclipse CVS, WinCVS, and TortoiseCVS. TortoiseCVS was chosen because it was easy to use and it is open-source so the students or the instructors would not have to absorb the cost of an expensive CVS license [30]. The students were able to install TortoiseCVS on their computers and work at home after having only a short instruction. The students were given logins and passwords for the CVS system at the start of the course.

The open-source projects were chosen by the instructors and used together with change requests found on the corresponding projects' websites. These change requests often required further clarification as they would in an industry setting.

We selected several open-source software systems written in either C++ or Java. During the software selection, the domain of the software was given considerable attention as we felt that the software domain should be intuitive and easy to understand in order to alleviate the student from having to learn too many domain-specific concepts. This ensured that each student was able to understand the required changes in a timely manner. Other important aspects included complexity and the number of available change requests. In the end, the final list of projects included JAdvisor [13], JCDSee [14], and WinMerge [15]. The sizes of the projects ranged from 14 to 69 classes.

JCDSee 0.6 uses Java and the Swing framework and organizes photos in various formats such as JPEG, BMP, and GIF. It also converts various image formats. It consists of 14 classes with 3506 lines of code. Table 1 shows the change requests that were implemented by the students, including both those taken from the JCDSee website and brand new ones, formulated by project managers.

JAdvisor 0.4.6 consists of 34 classes with 6145 lines of code also written in Java and the Swing framework. It provides the ability for a student to plan a schedule of courses downloaded from a university website, maintain a course history, and plan future course selections. Table 2 contains the change requests that were implemented by the students.

WinMerge 2.0.2 is a Win32 tool for displaying and comparing various document versions. It also provides visual tools for side-by-side line differencing and merging. It consists of 69 classes with 62990 lines of code written entirely in C++. Table 3 contains the change requests that were implemented by the students.

Each project was assigned to a separate team of 4-6 students. Each team had a project manager, a Ph.D. graduate student assistant, who was responsible for the change requests and supervision of the team members. Project managers also acted as formal customers when

Table 2. JAdvisor Change Requests.

Implement a wizard to generate a schedule of courses using the courses selected by the user. Automatically add the selected schedule to the planner and schedule portions of the user interface.
Use the school adapter framework to implement a new school adapter for Wayne State University.
Allow for partial course overlap when scheduling courses. Signify the overlap using a special color of your choice and notify the user.
Define an XML schedule schema. Implement a feature which outputs the schedule in the schema and also loads the schema.
Implement a feature which is able to read in the HTML created by the application and recreate the schedule. Populate the scheduler with the data.
Implement a feature which allows the user to block off time and assign it a category. Possible categories include lunch time, study time, etc.
Implement a feature which detects when a course has been previously taken and notify the user of the duplicate course. Allow the user to override such a selection.
Implement the functionality to save the planner information. Allow the user to track course completion and grades as well. Calculate GPA to date based on grades recorded by the user.
Allow the user to search the courses available by partial course numbering, credits, time offered, building, or teacher.
Automatically add a course which has been scheduled on the scheduler tab to the planner tab. Assume the course is to be added to the current semester being planned.

change requests needed clarification and provided resolution in case of a team members' dispute or an unexpected CVS problem. This was done via weekly meetings or emails between the students and project managers. It also allowed each project manager to be familiar with each student's capabilities as the students progressed through their changes.

Each team member worked independently on specific change requests and interacted with other team members via a CVS repository. Therefore, the students had to communicate with each other to ensure that their changes did not conflict. Any disputes were resolved by the project managers.

The course project was conducted in three main phases. At the end of each phase the project manager performed a build of the current CVS and saved the project as a release build. This ensured that errors did not persist from phase to phase and helped to recover the last working version of the project if critical errors were introduced into the code.

The first phase was the simplest so that the students were able to experiment with the CVS system and learn the course format. Typically, these changes required minimal communication among them and affected only a small number of the classes. This allowed the students to explore the architecture of the system and to experiment with the methodology.

After the first stage, the students understood the process of IC and the course expectations. Thus, the second and third phases introduced more complex change requests that required the students to communicate as their change requests often overlapped. The changes in these two phases were selected to ensure that all the stages of IC were

Table 3. WinMerge Change Requests.

Use the Microsoft framework to implement the "Tip of the Day" functionality.
Implement a persistent cache, which is a cache of merges that persists between executions of the application. These cached merges can be used to automatically repeat a previous merge, given the same two files that participated in the original merge as input
Modify WinMerge so that it highlights XML code. If either of the two files presented to WinMerge contain XML code, the XML should be highlighted.
Extend the application to allow for three-way document comparison and merging.
Provide the user with line statistics (the number of lines that are modified, deleted or added) as a menu choice option.
Add features to the Find dialog box to store the last searched text. Also add a drop-down menu which maintains a history of previous queries.
Allow the user to swap panels. When the user selects Swap, the documents should switch positions on the screen.

Table 4. Outline of Report Submitted by Students after Completing a Change.

Course Report Format
Change Request: Explain the change that was requested by your project leader.
Concept Location: Explain the method that you used in concept location. Describe the process of concept location including all classes that you visited. Give reasons to justify your decisions. Give the reasons why you believe the class(es) that you located represent your concept.
Impact Analysis: List the classes that may be affected by the change. Describe why these classes should be considered in your analysis. Remember that this is an analysis and may change after the actual implementation of the change request.
Pre-factoring: Explain whether or not the concepts described above need to be pre-factored. Use your analysis to support your choices.
Description of Implementation: Explain where and why you made a change in the code. Did you have to post-factor anything after implementing the change? Explain your decisions.
Modified Source Code: Attach your modified code including only the classes which were modified. Highlight the code that was changed or added using the computer. The report should have a professional look.

included. The students were given three weeks to complete each phase and the change requests were estimated to take 25 to 40 hours.

Each phase was graded by the project manager. At the end of each phase, the students had individual meetings with their managers in which they demonstrated the new functionality. The students were also individually interviewed as to their interaction with their team members and any problems with the CVS. After the initial interview, the students were required to turn in a report detailing the process they followed and the documentation for the change. Table 4 contains the questions that were addressed by the students in the project report.

The grade of each student was primarily based on individual effort, group interaction and communication, and the extent to which they followed the IC process. The individual effort comprised seventy percent of the grade and the team component was thirty percent of the grade. Table 5 lists the rubric used to grade the student submissions.

In assessing the individual grade, the project managers considered the students' explanation of the IC process. This included the process of successfully locating the required concept, identifying the impact set, performing pre-factoring and post-factoring, and implementing the change in a clear and concise manner, which is consistent with the code of the system. When discussing each activity, the students were expected to provide justification for their actions and architectural decisions.

The team portion of the grade consisted of the evaluation of the interactions among the team members. This required successfully communicating with other team members when change requests conflicted, ensuring that all files were merged appropriately within the CVS, and successfully performing a release build, which included the union

Table 5. Grading Scheme Used By Project Managers to Grade Student Submissions.

Sample Grading Scheme	Points
CVS Used and Functionality Checked-In: Was the new functionality successfully added to the CVS? Was it properly integrated with the other students change requests?	10
Met with Group Members Regarding Conflicting Change: Met with other group members when a change required coordination of more than one group member?	10
Interaction with Project Manager: Met with the project manager when asked. Asked questions pertinent to the change request.	10
Followed Correct Format of the Report: See Table 4 for explanation of format of the report.	10
Content of Paper: See Table 4 for explanation of paper content.	10
Incremental Change Time Table: Time required for each change.	5
Demonstrated Complete Functionality: Demonstrated functionality to project manager.	15
Correctness of Implemented Solution: Was all the functionality present and functioning correctly?	20
Organization of Solution and Comments: Were classes refactored when needed? Were interfaces defined appropriately?	10

Table 6. Student Survey Results Before and After Introduction of the Software Evolution into Course Projects

	How would you rate the course?	How much did you learn in this course?
Fall 2002 Mean	3.4	3.4
Fall 2002 Median	3	3
Fall 2004 Mean	3.7	3.8
Fall 2004 Median	4	4

of the new functionality added by all team members. Through the individual interviews and analysis of CVS data log files, the project managers were able to determine those team members who were not doing their part. This also allowed them to accurately assign grades accordingly. This ensured that no team member was able to hide in the team and that no team member was unduly punished if the CVS became corrupted.

The projects were also organized to deter cheating. As it was mentioned earlier, all of the change requests were either taken form the corresponding project sites or formed by the project manager. This made cheating almost impossible, when combined with the CVS, since the implementations of the change were not available and the students had to produce their own implementation.

5. Course Assessment

The course had one project manager per team (three for the whole course) in order to ensure that students were given adequate attention and to spread the pre-course planning and course workload. Most of the time was spent in determining appropriate course projects, setting up the CVS, and structuring change requests. This process took place a month before the course was scheduled to begin and was complete before the course projects were assigned. We estimate the shared setup cost to be 60 hours of planning and preparation time.

After the course infrastructure was in place, each project team required roughly six or seven hours of manager's time in an average semester week and around ten to twelve hours in the weeks where build releases were due.

The new version of the course projects was first offered in Fall 2004, while the last course with the traditional projects was offered in Fall 2002. At the end of course, a course assessment survey is traditionally conducted. The survey is comprised of several questions relating to the course project and the course content. Students are asked to rate the course on a five point scale. Question one asks the students to rate the course as a whole with five being "Excellent" and one being "Very Poor". Question two asks the students to rate their learning in the course with five being "a Great Deal" and one being "Nothing at All". Table 6 shows the large increase in student satisfaction and student learning in the course.

These results were also reiterated by the students during the course project survey. One part-time student, who also is a practicing software engineer for a large company, commented that, "*The IC methodology is helpful in analyzing and implementing change requests.*" Another student echoed this by saying, "*This being our third phase, we habitually followed this approach as soon as we were handed the change request. We realized that this approach saves a lot of time and is beneficial.*"

Another pair of students responded by saying, "*Before taking this course, we did not have these concepts* (methodological). *Now we realize that they greatly improve the clarity of code and make it easier for later modification. We were more dependent on intuition and experiences to change code before. We now can apply these new techniques to modify code systematically and efficiently.*"

The students particularly liked the new ability to update large and unknown software. Positive comments from two practicing programmers included, "*The incremental change approach is a well defined method that helped us implement our changes in a neat fashion. This method was very useful to us when we were asked to implement change requests. Generally when people are asked to perform changes to large projects they would be lost or find difficulty in starting. We think this* (referring to IC) *is a methodological and standard way to implement changes.*"

Project managers were also positive in their experiences with the course, as they gained experience in managing a team of several students.

6. Conclusions and Future Work

The approach of applying the process of IC to an open-source project allows students to work on realistic programs and thus gives them a valuable project experience. This approach places the emphasis on individual contribution within the contexts of the team, as is often the case in the industry settings.

The course, outlined in this paper, fulfilled expectations when addressing the problems of teaching software engineering: realistic size programs, industry-like setting, and individual accountability. Medium sized software systems were evolved using realistic change requests. This noticeably increased the motivation of our students and their understanding of the software engineering process. This is reflected in the student assessments of the course.

Although additional resources are required in terms of course administration to teach such a course, the resources are not excessive and are well spent when evaluating the benefits to both the students and the instructors.

Based on this experience, tools such as JRipples [32] and IRiSS [33] are being implemented to facilitate the IC process and guide student users through it. Moreover, the techniques used for concept location, impact analysis, and change propagation are currently being improved and expanded. These techniques will be incorporated into future software engineering courses. The course also would be greatly enhanced by a textbook that is based on this or a similar approach to teaching software engineering.

Acknowledgments

This work was supported in part by grants from the National Science Foundation (CCF-0438970), the National Institute for Health (NHGRI 1R01HG003491), and by 2005 IBM Faculty Award. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, NIH, or IBM.

References

[1] Beck, K., *Extreme Programming Explained*. Reading, MA: Addison Wesley Longman Inc, 2000.

[2] Dick, M., Postema, M., and Miller, J., "Improving student performance in software engineering practice", in Proceedings of International Conference on Software Engineering Education and Training (CSEE&T'01), Charlotte, NC USA, 2001, pp. 143-152.

[3] Postema, M., Miller, J., and Dick, M., "Including Practical Software Evolution in Software Engineering Education", in Proceedings of *14th Conference on Software Engineering Education and Training (CSEE&T'01)*, Charlotte, NC USA, 2001, pp. 127-135.

[4] Hayes, J. H., Lethbridge, T. C., and Port, D., "Evaluating Individual Contribution Toward Group Software Engineering Projects", in Proceedings of 25th International Conference on Software Engineering (ICSE'03), 2003, pp. 622 - 627.

[5] Hazzan, O. and Dubinsky, Y., "Teaching a Software Development Methodology: the Case of Extreme Programming", in Proceedings of 16th Conference on Software Engineering Education and Training (CSEE&T'03), 2003, pp. 176 - 184.

[6] Xu, S. and Rajlich, V., "Pair Programming in Graduate Software Engineering Course Projects", in Proceedings of 35th ASEE/IEEE Frontiers in Education Conference (FIE'05), 2005, pp. 7-12.

[7] Schach, S., *Object Oriented and Classical Software Engineering*, Fifth ed: McGraw-Hill, 2002.

[8] Somerville, I., *Software Engineering*, Sixth ed: Addison-Wesley, 2001.

[9] Gnatz, M., Kof, L., Prilmeier, F., and Seifert, T., "A Practical Approach of Teaching Software Engineering", in Proceedings of 16th Conference on Software Engineering Education and Training (CSEE&T'03), 2003, pp. 140-147.

[10] Hedin, G., Bendix, L., and Magnusson, B., "Introducing Software Engineering by means of Extreme Programming", in Proceedings of 25th International Conference on Software Engineering (ICSE'03), 2003, pp. 586 - 593.

[11] Rajlich, V. and Bennett, K., "A Staged Model for the Software Lifecycle", *Computer*, vol. 33, no. 7, July 2000, pp. 66-71.

[12] Rajlich, V. and Gosavi, P., "Incremental Change in Object-Oriented Programming", in *IEEE Software*, 2004, pp. 2-9.

[13] Rawls, C., "JAdvisor", Date Accessed: 8/18/2004, http://jadvisor.sourceforge.net, 2002.

[14] Tri, T. T. H., "JCDSee", http://jcdsee.sourceforge.net, 2001.

[15] "WinMerge", http://winmerge.sourceforge.net, 2004.

[16] Rajlich, V. and Wilde, N., "The Role of Concepts in Program Comprehension", in Proceedings of IEEE International Workshop on Program Comprehension (IWPC'02), 2002, pp. 271-278.

[17] Marcus, A., Rajlich, V., Buchta, J., Petrenko, M., and Sergeyev, A., "Static Techniques for Concept Location in Object-Oriented Code", in Proceedings of 13th IEEE International Workshop on Program Comprehension (IWPC'05), 2005, pp. 33-42.

[18] Marcus, A., Sergeyev, A., Rajlich, V., and Maletic, J., "An Information Retrieval Approach to Concpet Location in Source Code", in Proceedings of 11th IEEE Working Conference on Reverse Engineering (WCRE2004), Delft, The Netherlands, November 9-12 2004, pp. 214-223.

[19] Chen, K. and Rajlich, V., "Case Study of Feature Location Using Dependece Graph", in Proceedings of 8th IEEE International Workshop on Program Comprehension (IWPC'00), Limerick, Ireland, June 2000 2000, pp. 241-249.

[20] Antoniol, G. and Gueheneuc, Y., "Feature Identification: A Novel Approach and a Case Study", in Proceedings of 21st IEEE International Conference on

Software Maintenance (ICSM'05), Budapest, Hungary, September 25 2005, pp. 357-366.

[21] Wilde, N., Buckellew, M., Page, H., Rajlich, V., and Pounds, L., "A Comparison of Methods for Locating Features in Legacy Software", *Journal of Systems and Software*, vol. 65, no. 2, February 15 2003, pp. 105-114.

[22] Wilde, N. and Scully, M., "Software Reconnaissance: Mapping Program Features to Code", *Software Maintenance: Research and Practice*, vol. 7, 1995, pp. 49-62.

[23] Eisenbarth, T., Koschke, R., and Simon, D., "Locating Features in Source Code", *IEEE Transactions on Software Engineering*, vol. 29, no. 3, March 2003, pp. 210 - 224.

[24] Chen, A., Chou, E., Wong, J., Yao, A. Y., Zhang, Q., Zhang, S., and Michail, A., "CVSSearch: searching through source code using CVS comments", in Proceedings of IEEE International Conference on Software Maintenance (ICSM'01), Nov. 2001, pp. 364-373.

[25] Poshyvanyk, D., Gueheneuc, Y., Marcus, A., Antoniol, G., and Rajlich, V., "Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification", in Proceedings of 14th IEEE International Conference on Program Comprehension (ICPC'06), 2006, pp. 137-148.

[26] Bohner, S. and Arnold, R., "Software Change Impact Analysis," in *An Introduction to Software Change Impact Analysis*. Los Alamitos, CA: IEEE Computer Society Press, 1996, pp. 1-28.

[27] Briand, L., Labiche, Y., and Sullivan, L., "Impact Analysis and Change Management of UML Models", in Proceedings of International Conference on Software Maintenance (ICSM'03), Amsterdam, The Netherlands, September 22 - 26, 2003 2003, pp. 256-265.

[28] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D., *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison Wesley, 1999.

[29] Rajlich, V., "A Model for Change Propagation Based on Graph Rewriting", in Proceedings of IEEE International Conference on Software Maintenance (ICSM'97), 1997, pp. 84-91.

[30]"TortoiseCVS", http://tortoiseCVS.sourceforge.net, 2004

[31] "WinCVS", http://www.wincvs.org, 2004.

[32] Buckner, J., Buchta, J., Petrenko, M., and Rajlich, V., "JRipples: A Tool for Program Comprehension during Incremental Change", in Proceedings of 13th IEEE International Workshop on Program Comprehension (IWPC'05), May 15-16 2005, pp. 149-152.

[33] Poshyvanyk, D., Marcus, A., Dong, Y., and Sergeyev, A., "IRiSS - A Source Code Exploration Tool", in Industrial and Tool Proceedings of 21st IEEE International Conference on Software Maintenance (ICSM'05), Budapest, Hungary, September 25-30 2005, pp. 69-72.