Experiences with a Software Maintenance Project Course

James H. Andrews, *Member, IEEE,* and Hanan L. Lutfiyya, *Member, IEEE*

***Abstract*—A report is made on an experience of teaching a senior-year course on software maintenance, centered around a maintenance project. For the course, students organized themselves into groups and worked on adaptive and perfective maintenance of selected real-world software products. The projects involved such issues as code understanding, requirements engineering, and maintenance design, and dealt with both open-source and proprietary software. The main triumphs and pitfalls of the course are recounted, and recommendations are made on project selection and general course conduct.**

***Index Terms*—Software maintenance, software project courses.**

I. Introduction

SOFTWARE maintenance refers to the modifications that are made to software after its initial release [1], [2]. Given the costs of new system development, many organizations are reluctant to completely replace older software. Organizations therefore tend to modify existing software in order to repair design faults, adapt to new hardware, or address evolving user requirements. In concert with the need for software maintenance comes the need to deal with legacy systems (systems that have been in use for years). There is increasing demand from industry for professionals who understand the key issues and problems of modifying existing software.

The past decade has seen an increased focus on university-based education in software engineering (SE). There are now a number of courses in introductory SE, software architecture, and human–computer interfaces. However, there are relatively few courses in software maintenance.

In September 1998, the authors' department offered for the first time a senior-level SE course on software maintenance, as part of a six-course SE specialization program. The course was focused on software maintenance projects chosen by the instructors, students, and client organizations. This paper reports on experiences the authors had in offering this course.

II. the Computer Science Program at University of Western Ontario (UWO)

This section situates the course being described within the overall computer science program at the University of Western Ontario (UWO).

Manuscript received August 16, 2000; revised September 5, 2000.

The authors are with the Department of Computer Science, University of Western Ontario, London, ON, Canada N6A 5B7 (e-mail: andrews; hanan@csd.uwo.ca).

Publisher Item Identifier S 0018-9359(00)10235-3.

The Computer Science Department at UWO offers a B.Sc. in computer science with a software engineering specialization. Both an honors and a general degree are offered. This program requires the students to take the following SE courses:

1) A second-year introductory course in SE. The students are introduced to the fundamentals of design, testing, requirement understanding and project management. The course requires that students develop (in teams of four to six people) software that is formally tested by the course instructors.
2) A third-year course in intermediate SE. The focus of this course is on advanced design using object-oriented design and analysis approaches. Again, the students are required to develop (in teams of four to six) software that is formally tested by the course instructors.
3) Three from the list of following courses: Human–Computer Interaction, Testing and Specification, Project Management and Advanced Software Design and Architecture.
4) The software maintenance course (that is examined in this paper), formally called Software Maintenance and Configuration Management.

Thus, the students complete two team projects before their fourth year and have experience in having others formally test their software. However, upon entering fourth year they have yet to modify existing software except to find bugs in other project members' code. Since much of software development deals with existing software, it was felt that it was important to give students the necessary experience.

III. the Software Maintenance Course

This section describes the structure of the course as planned.

The course was defined as lasting from the beginning of the first term of the school year until the end of the second term. (The two terms at UWO are September–December and January–April.) The course nominally consisted of two hours per week of lectures. In fact, the course was planned and executed such that lectures were given in the lecture hours only for the first few weeks, and then the lecture hours were used for presentations and small group discussions.

From the earliest stages in curriculum planning, the intention was to make this a course focused on projects executed by teams of students. The choice of projects was motivated by considering the different classes of maintenance tasks that exist. Software maintenance can be divided into three categories: *corrective*, *adaptive*, and *perfective* [1]. Corrective maintenance deals with the removal of residual errors present in the product when it is delivered. Adaptive maintenance involves modifying the software to changes in the environment, e.g., a new release of the hardware or operating system. Perfective maintenance involves changing the software to improve it according to new requirements.

In the development of the course, corrective maintenance was not considered to merit a great deal of attention. First, corrective maintenance is considered to be a relatively small proportion of all maintenance done [2]. Second, the students have experience in corrective maintenance. In other courses, they have been asked to find bugs in code that they had not written. In addition, many have experience in finding bugs in other people's code from their two previous projects. Hence, the focus was on finding projects that provided students with experience in adaptive and perfective maintenance.

In the first month of the course, it was decided which projects would be attempted, and which students would be working on which projects (the projects and groups are described below). During the course of the year, each project group was expected to hand in the following items.

- An initial project plan in mid-October. By this point, it was felt that each group should have an understanding of the various technologies and tasks concerned with their project; this assignment was intended to get them to plan, at a high level, what they were going to do over the course of the year.
- A progress report in early November. This was a one-page report describing the work done to start the project; for example, the tasks performed to get the software installed and compiled, and the initial attempts at understanding the code or architecture.
- A progress report in late November. This report was a 5–10 page paper summarizing the progress in understanding the program, establishing version control/checkout procedures, and giving a more detailed description of the maintenance objectives and the approach the group was going to use in meeting them.
- A progress report in mid-January. There were three parts to this progress report: 1) A description of the overall architecture of the system; 2) A description of each maintenance objective in detail; and 3) A revised project plan describing in detail how they were going to execute the project during the remainder of the year.
- A final report in early April. This was a description of the achievements of the group over the course of the year, including whether and how each maintenance objective was attained.

In addition, the following presentations were expected:

- In January, an overall description of the project, a description of the work so far, and the maintenance plan.
- Small inter–group seminars in February and March, where each student described aspects of their project to the instructor and to students outside of their project group.
- A final presentation presented by each group. Each member of the group was expected to contribute to the final presentation.

IV. Group Organization and Project Selection

This section describes how the groups were formed, the projects which the groups selected and the clients sponsoring the projects, and some problems that arose with project cancellations.

A. Initial Organization of Groups

At the start of the course, it was suggested to the students that they organize themselves into groups of no more than six, with each group choosing a project (for some of the projects, a group size was suggested). The students were trusted to do this without interference from the instructors. This was done because the students had had at least two previous courses in which they had worked in groups, because they generally knew each other well, and because many of them already had experience in industry which helped their organizational abilities.

To a certain extent, this trust was rewarded. However, by the end of the group organization period, there were still seven students who had not gotten into groups. Six of these students were placed in a group on their own (group 5); the seventh was placed in an existing group, a move which initially caused some friction with students who felt that a new group member was being imposed on them.

Another problem was that some students (as they confided) would have been more comfortable with a smaller group, but because their group had not reached the maximum size of six, they felt obliged to take new members when asked by other students. In this sense, paradoxically, it might have been easier for the students if the instructors had had more control over the group formation process.

B. Clients, Projects and Groups

Client A was a software development department within a large company. Client A was interested in exploring the idea of restructuring a system in order to move some of its services from a central computer to regional computers. Student group 1 took on this task. Group 1 consisted of students who had just spent a 16-month internship at Client A's company, and so were well-suited to the project.

Client A's project was not classic maintenance as is generally understood. That is, it did not involve the understanding of legacy code and the relatively minor modification of it; rather, it involved writing a new version of the software system based on parts of the legacy system. This involved both adaptive maintenance (incorporating new hardware, operating systems and programming platforms) and perfective maintenance (restructuring the system for efficiency reasons).

Client B was a software development department within a service organization. Client B proposed a number of projects, and student groups 2 and 3 took up two of their proposals. These projects were also not maintenance in the classical sense, but also involved writing new software which interfaced with legacy systems or databases. These projects therefore also involved both adaptive and perfective maintenance.

Finally, several projects suggested by the GNU Project were proposed to the students. GNU [3] is a worldwide, cooperative software development project which aims to produce freeware versions of all standard Unix utilities; many GNU programs are used in Linux systems. Groups 4 and 5 took up two of the proposed projects. These projects were classic maintenance, as they involved the understanding and modification of codebases of significant size. The emphasis was on adding features and improving performance. These are examples of perfective maintenance.

C. Project Cancellations

In general, the projects proceeded as expected after the initial group integration. However, there was one major problem: the cancellation of first one, and then another project assigned to Group 3.

Group 3's first project, as proposed by Client B, involved a large software system that Client B intended to install. However, after Group 3 had done approximately six weeks of project understanding work, Client B wrote in e-mail that they could not install the software and therefore were "cancelling" the project. This caused some alarm. However, in an emergency meeting shortly after, the instructors, Client B, and Group 3 agreed on an alternative project involving the setup of a standalone computer.

Unfortunately, in mid-January, Client B announced that they could not set up the standalone computer either. It was then agreed in e-mail that Group 3 and Client B would have an amicable separation, and a search began for a small maintenance project they could do instead in order to salvage the rest of the course. Group 3 settled on a small project involving Mozilla, Netscape's open-source browser software. This became their project for the remaining 10 weeks of the course.

To give Client B their due, they were an extremely busy organization in the midst of a major change in their software base, and so often could not devote enough time to everything they wanted to do. Group 2 also did a project for Client B; that project experienced no cancellations or withdrawal of facilities.

V. The Execution of the Projects

This section describes the main events which happened during the course of the year, and the tools, techniques and schedules used to execute the projects. The course lasted for 26 weeks of class time, not including the December/January exam and holiday period; the events of the year will therefore be referred to as happening in "week x/26," where x is some number between 1–26.

A. Project Planning

The groups were required to produce an initial project plan, including information on the tasks to be done by the various group members, in the first report (in week 6/26). They were also required to report on progress in all other reports except the last. To some extent, the project plan was constrained by the set milestones (see Section III), but the groups were free to identify tasks specific to their projects and allocate group resources to them.

The groups generally separated the tasks into tasks to be done by all the group members together, and tasks to be done by individuals. Typical tasks done by all the group included initial code understanding, maintenance design, and the production of the final report. Typical tasks done by individuals included installation, implementation of specific features, and production of specific sections of the various reports.

Some of the students had experience with the project-planning software Microsoft Project, and so were delegated the task of producing the project plan for their group in graphical format. These plans were found to complement well the written description of the project plan for the groups that produced them, although not all groups could be expected to produce them.

The project plans (including the Microsoft Project versions) were updated in the January progress report (week 15/26) to show the progress on the milestones so far. As expected, the students had to make modifications to their original project plans. The main reason was usually the result of a better understanding of the software, which led them to revise their estimates on the time needed for adding or evaluating new features.

B. Installation and Program Understanding

The initial tasks for all of the groups were to install any necessary software, to understand either the existing code (for the "classic maintenance" GNU projects) or the legacy systems (for the other projects), and to understand and solidify the maintenance requirements. The groups' progress in this was reported on briefly in the November reports (weeks 9/26 and 11/26), and more extensively in the January report (week 15/26).

The groups attempting to understand existing code found a code understanding tool called SourceNavigator [4] to be helpful. SourceNavigator produces some dataflow analysis and cross-referencing, and allowed the students to browse the source code in a structured way. UWO CS had installed a free beta version of the executable of this tool, which had been obtained from Cygnus Software (now part of Red Hat, Inc.). One of the groups also found it helpful to simply step through the running executable using the GNU debugger tool, `gdb` [3]. Naturally, the students also used the usual Unix utilities like editors and the regular expression finder `grep` to aid their understanding.

The groups were also expected to install a revision control system to track revisions being made to the system. Most of the groups reported having done this by the late November (week 11/26) progress report.

The students were taught about the free Unix revision control systems SCCS, RCS, and CVS. None of the groups opted to use CVS, despite its popularity in real-world projects; the most common reason cited for this reluctance was that they did not trust CVS's system of allowing multiple users to check out code and then weaving together checked-in code in a consistent way. In the end this point did not seem to matter, because it is not clear that any of the groups actually used the revision control system for systematic configuration management.

C. First Project Presentation

By the time of the midyear project report (week 15/26, mid-January), the groups had completed installation of software and initial understanding of code and legacy systems, had expanded on the maintenance requirements in more depth, and had drawn up a more detailed plan for the maintenance tasks. They presented this work in class, with each group making an approximately 25-min presentation on their project.

In general, the work of all the groups was satisfactory at this point, although the maintenance plans and designs were less detailed than had been hoped. This was probably the result of the well-known tendency to jump from high-level design to code modification, and to skimp on the design phase that should come in-between.

D. Small Group Meetings

Small group meetings were held in weeks 19–22/26. Each meeting included the two instructors and three students from three different groups. Each student was required to give a five-minute summary of their project and a five-minute description of their role in their project, followed for each student by questions from the instructors and a general discussion period among all those present.

These small group meetings were useful for the following reasons. First, the instructors were able to get a better understanding of each individual's contribution. Second, some of the students were able to get valuable advice about particular problems they were having, from the instructors and/or the other students. For example, if a student was having problems with evaluating the performance of a specific piece of code, another student would suggest tools that they had found useful in their own work. Third, in the case of one group, a major flaw was detected in the design of features that were to be added to the software. This enabled the group to revise their design. In the case of a second group, the instructors detected that they had not considered all possible scenarios in evaluating their design. The group did a further evaluation (although there were relatively few changes to the design).

E. Final Presentation and Report

In early April, each group was required to present their final presentation and report. Each member was expected to contribute in the final presentation. Here, they summarized again the goals and plans for the project, and they reported on what they had accomplished and how their accomplishments matched up with the original goals and plans.

In general, the work of all groups was satisfactory, although for two groups, the instructors felt that they should have been able to accomplish more; their marks in the course naturally reflected this.

VI. Project Results

This section describes the overall results of the various projects.

Group 1 achieved most of its maintenance objectives and returned a project which Client A found useful. Although the software was not as complete and integrated as had been hoped, it was also intended by Client A more as a prototype; they therefore expected to change the code they received anyway.

Group 2 also achieved most of its maintenance objectives, again returning a project not quite complete and integrated, but which Client B found very useful as the basis for further development. For both Groups 1 and 2, although the projects were only partly completed, the documentation that was required to be submitted increased the value of the code to the development organizations.

Group 3 achieved some of its maintenance objectives. Unfortunately, after planning and partially implementing one of the major features they planned to add, as suggested by the Mozilla website, they realized it had already been added several months before. The website had not been updated to reflect the change, and because of the size of the codebase, Group 3 had not noticed the code for the new feature. Nevertheless, Group 3 made enough of a start on adding the duplicate feature and another feature that their work was judged to be good considering the time frame they had in which to do it.

Group 4 became fragmented early on, since it had several subsidiary objectives and decided to break up to work on them. Unfortunately, it never really became integrated again. Two of the group members achieved good success in their task, and submitted code which is in the process of being submitted back to the GNU project.

The rest of the group members had more or less success in achieving their goals, but never integrated the new version of the software with that produced by the other two. Although all groups were encouraged to use revision control systems to ensure they produced an integrated product, this was not monitored closely and it seems clear that this was not always done.

Group 5 worked steadily throughout the year, with the exception of one member who did virtually nothing and failed the course. However, the initial design of the modifications to meet their major maintenance objective turned out to be flawed; this fact emerged only in the small-group meetings of February–March. Unfortunately, they were never able to adequately address the flaw, and ended up with code which did not adequately achieve this objective. Nevertheless, their other objectives were more or less achieved, and the incomplete code, along with the documentation, was submitted to the GNU Project, with the encouragement of the main GNU contact for this maintenance project.

One problem with Group 5 was that the project had been selected by majority vote. Two of the students on the losing end of the vote reported that their major problem was a lack of interest and motivation in the project. This may well have been a consequence of the fact that the group had been formed from members not yet in other groups.

VII. Lessons Learned

This section reports on some of the things learned while giving the course. It is broken up into lessons learned about the nature of maintenance tasks, a comparison of the commercial-client and open-source maintenance projects used, thoughts about what could have been done differently, and things that were done well.

A. The Nature of Maintenance

One of the things the instructors had to accept early on was that the maintenance projects suggested by industrial partners were not classic code modification tasks, but rather concerned the development of new code to fit into existing business processes. This seems to be primarily a result of IT being concerned in the short term with the shift to network computing and client-server applications.

B. Commercial-Client Versus Open-Source Projects

Which kind of project was better, the ones where the students interacted with a commercial client or the ones where the students maintained open-source software? The two authors have somewhat different opinions on this issue. This section simply discusses the pros and cons of the two kinds of projects, and what can be done to emphasize the good points and mitigate the bad points.

In commercial-client projects, the students had contact with a project of immediate commercial value to their clients. They could also interact directly with these clients, who could in turn give them valuable references for future employment opportunities. These projects tended to be more cutting-edge, since they involved the conversion to newer technologies such as the client-server architecture. Hence the students were more enthusiastic about these projects.

However, the commercial-client projects brought with them considerable risk. There was always the potential that a project be cancelled, or drift away from the goals of the course. Yet, for instance, no leverage existed to demand anything from Client B when the project was cancelled, since no guarantees could be given that finished software would be delivered. It may have been useful to draw up a semi-formal agreement among clients, students and instructors on what the obligations were on all sides. This would not have been legally binding, but would at least have afforded some basis for a semi-formal complaint if things had gone awry.

The open-source projects followed a more predictable course. They were based on publically available code and widely used platforms, and therefore could be carried out on student machines without depending on other organizations or equipment. The requirements for the projects were relatively simple and clearly defined, and the students were able, through their work, to contribute to open-source projects such as GNU and Linux.

However, many students were unenthusiastic about the GNU projects. This may have been simply the nature of the projects: unexciting Unix utilities implemented in C Versus GUI-based, object-oriented client-server applications implemented in Java. The students may have shown more interest if the overall impact of GNU, Linux, and open-source code on the software industry had been explored more deeply with them. There was definite interest in the Mozilla project, so choosing projects from such open-source applications as Mozilla and Apache seems a good choice; however, care must be taken here to select projects with a long time frame, that can be completed before these fast-moving codebases have changed so much that the students' work is irrelevant.

C. Things That Could Have Been Done Differently

As discussed above, it would have been useful for the instructors to assume more control of the group formation process, in order to take some of the organizational responsibilities off the students' shoulders. As an alternative to what was done, students could submit information about the projects they are most interested in and information about whom they want to work with. The instructors could then form the groups based on this information.

Although some of the methods, techniques, and tools used when maintaining evolving software were introduced, and the students were encouraged to use them, there were mixed results. For example, the students were familiar with configuration management systems, and they were highly encouraged to use such systems in their projects. However, it is not clear that any of them did so successfully. On the other hand, as mentioned above, two of the groups found S-Navigator [4] to be very useful. Part of the problem may be that the free configuration management tools that are available are somewhat behind the commercial tools, while S-Navigator is not. In the future, several examples will be walked through of how the free configuration management tools can be useful to their projects, and the use of one of the configuration management tools will be made a required part of the project.

Thirdly, concepts related to software architecture and design patterns should have been introduced earlier. The department has a separate course on software architecture and design patterns; hence, there was relatively little formal discussion on this. However, the architecture course took place from January to April. The groups indicated that knowing this material earlier would have better helped them formalize their architectures. Two of the groups made many last minute additions to their software based on the design patterns they learned in the course. They indicated that the design patterns gave them insight into how to make sure that the addition of their features would be maintainable in the future. In future years, this course may be offered from September to December.

D. Things That Were Done Right

Finally, some words about what was successful in the course.

The students were asked to do real-world maintenance tasks that potentially had an impact beyond the bounds of the course. This was crucial for helping the students maintain interest in the projects, and for giving the students who had not had commercial development experience a feel for what software development was like in the "real world."

The students were required to submit extensive documentation about what they had learned about the code they were working with, how they approached the project, and what the results were. Since many maintenance tasks are poorly documented in the real world, these documents turned out to be a valuable contribution in their own right. In one of the GNU projects, there was more interest expressed in the document than in the code produced, since the document described the organization of the code.

Interaction among the students was encouraged by requiring them to present the results of their projects to each other periodically, and by requiring them to attend the small-group meetings, at which no other members from their group were present. This allowed them to feel more cohesive as a class, and to exchange ideas amongst themselves, rather than just with the instructors and the members of their project group. The small-group meetings also gave the chance to identify problems with personnel or project management, which paid off in some instances.

Two of the projects highly encouraged students to think about how their additions could be made maintainable. In one case, the client requirements for the additional features changed on an almost daily basis. The students found this frustrating, but it forced them to think about designing their additions so it would be easy to change the software in the future. Their realization of how important design is to maintenance was exciting. Although this was emphasized in other courses, this is something not truly appreciated until experienced. More projects that emphasize this aspect would be very desirable.

VIII. Related Work

In developing this course, a number of programs in Canada and the United States were examined by going to their web sites. No courses which took a similar approach for teaching software maintenance were found. Most computer science programs offer no more than two software engineering courses. The closest courses to software maintenance were courses at the Universities of Alberta, Victoria, and Waterloo which contained assignments focusing on determining the architecture of a large piece of source code.

The course described here differs as follows. First, there is more variability in the type of projects allowed. For example, some of the projects allow students to look at making a sequential piece of software distributed. This does not necessarily require an in-depth source code analysis, but it is an important problem which requires the students to learn how to integrate pieces of existing legacy code. Second, the projects selected require the students to test their understanding of the legacy software by requiring them to enhance the software.

IX. Conclusion

The following are suggested reasons why there are very few courses in software maintenance.

Attitude:	Software maintenance is often not considered a discipline by many academics.
Texts:	There are very few texts that focus on software maintenance.
Course Development:	A course in software maintenance requires projects. There is a perception that finding projects appropriate for undergraduate students is difficult.

However, the course described in this article gave students valuable experience in the qualitatively different task of software maintenance. Key to this was the selection of and focus on real-world projects. As this was the first time the course was offered, some difficulties had to be worked through. However, the authors hope to learn from these difficulties, and have written this paper in the hope that others can learn from them as well.

References

[1] A. A. Takang and P. A. Grubb, *Software Maintenance: Concepts and Practice*. London, U.K.: Int. Thomson Comput. Press, 1996.
[2] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*. Englewood Cliffs, NJ: Prentice-Hall, 1991.
[3] GNU Project, "Gnu project web pages,", www.gnu.org, 2000.
[4] Red Hat Inc., "SourceNavigator web pages,", sources.redhat.com/sourcenav, or reachable by searching www.redhat.com, 2000.

James H. Andrews (M'98) received the B.Sc. and M.Sc. degrees from the University of British Columbia, Vancouver, Canada, in 1982 and 1986, respectively, and the Ph.D. degree from the University of Edinburgh, U.K., in 1991, all in computer science.

He was with Bell-Northern Research, Ottawa, ON, Canada, from 1982 to 1984, with Simon Fraser University, Vancouver, BC, Canada, from 1991 to 1995, and with the University of British Columbia on the FormalWare project from 1996 to 1997. He has been with the Computer Science Department, University of Western Ontario, London, ON, since 1997, where he is an Assistant Professor. His research interests include software testing, semantics of programming languages, and formal specification.

Hanan L. Lutfiyya (M'92) received the B.Sc. degree from Yarmouk University, Jordan, the M.Sc. degree from the University of Iowa, Ames, and the Ph.D. degree in 1992 from the University of Missouri, Rolla, all in computer science.

She has been with the Computer Science Department, the University of Western Ontario, London, ON, Canada, since 1992, where she is an Associate Professor. Her research interests include distributed systems management and software engineering.