Using Open Software for Software Engineering Class Projects

Mark M. Meysenburg
Information Science and Technology Department
Doane College
mmeysenburg@doane.edu

Abstract

In this paper, we describe a successful software engineering class project, in which a large open-source program was employed. The use of the open-source program allowed students to experience the realities involved with understanding, maintaining, and extending software considerably larger than they are normally exposed to in college courses. In this paper we describe the way in which we chose our open-source program, the activities students completed during the project, and the educational benefits achieved by students.

In the past, the one-semester-only nature of the software engineering course meant that students were often unable to experience first-hand many of the key principles of software engineering. Put simply, students could not develop a system from scratch large enough to require many of the practices taught in the course. Problems such as these led us to use open-source software for our software engineering course project.

We chose the open-source software based on five criteria: source language, run-time environment, operating system, size, and available documentation. During the project, students had to complete several activities. First, they had to successfully build the game used for the project. Then, they had to produce some initial documentation for the system in the form of a structure chart. Next, they had to find and correct several logic errors, inserted into the system by the professor, based on written bug reports. Finally, they had to extend the game in ways of their own design and present their improved games to the class.

Discussions with students during the class made it clear that the students gained much greater respect for the value of quality documentation and sound design principles. The students were able to gain some experience with a large (200+ source files), “real world” software system. Students also gained insight into the work of maintenance programmers, positions where many new software engineers will begin their careers. In these ways and others, the use of open-source software for the class project was very beneficial to our students.

Introduction

In this paper we describe how we were able to successfully incorporate a large, open-source software game into the software engineering course at Doane College. In the sections below, we describe the factors motivating us to add such a project to our course, the criteria used to select our open-source software system, the activities software engineering students participated in during the project, and the educational benefits afforded to the students through the project. Finally, we conclude with a section describing lessons learned, and how we will modify the open-source project for future software engineering courses.

Motivating factors

At Doane College, our software engineering course is a one-semester, 300-level class. The class is offered every other year, with an average enrollment of approximately 20 students. The only pre-requisites for the class are two introductory C++ programming courses. The students in the course are typically a mix of Computer Science students, who usually have slightly more programming experience, and Information Systems students, who often have no programming beyond the introductory C++ courses. The fact that the course is only one semester long introduces certain problems in conveying some of the most important software engineering concepts.

In particular, the one-semester-only nature of the software engineering course meant that, in the past, students were often unable to experience first-hand many of the key principles of software engineering. During the course, we would attempt to build a small system from start to finish. But, put simply, students could not, during the course of one semester, develop a system from scratch large enough to require many of the practices taught in the course.

For example, the critical importance of documentation was hard to demonstrate in such a project, as the systems developed were relatively small and the time span was short enough that all the details could be grasped and retained. Even with a small system *not* constructed by the students, the size of the system would allow students to examine and understand the software without adequate documentation.

Fault isolation is likely to be easier in a small, start-to-finish project than it would be on production-size systems. While ease of fault detection would be a good illustration of the benefits of sound design (OOP, modularity, etc.), it would not adequately prepare students for the difficulties with fault isolation they will likely encounter in their careers.

The particular difficulties encountered in software maintenance are also hard or impossible to illustrate with a small system developed from start to finish. This is particularly true in a one-semester course; students barely have enough time to analyze, design, code and test a system. There simply isn’t time in the course to actually practice software maintenance activities.

The difficulties listed above motivated us to use open-source software for our software engineering course project. In addition, we realize that most entry-level software engineers will be working on maintenance projects, rather than developing new systems, so it makes sense to expose students to software maintenance activities.

Choosing the open-source software

Once we decided to use an open-source system for our class, we needed to find a suitable system. We chose the system for our open-source software project based on five criteria: source language, run-time environment, operating system, size, and available documentation. We searched for our open-source software at one of the largest online open-source repositories, sourceforge.net (http://sourceforge.net).

Since our programming courses are taught in C++, we wanted our project software to be written in C++ (or, less preferably, C). Thankfully, many open-source systems met this requirement, although we found relatively few pure C++, object-oriented systems. Expanding our language requirements to include other languages, such as Java, would have allowed us to choose from more object-oriented systems, but we chose to stay with C / C++, so that the software engineering course could focus on software engineering principles, and not on new language implementation details.

By the time our students take the software engineering course, most of them have had no experience with GUI or event-driven programming. Therefore, we wanted the project software to run in a console-based (non-GUI) environment. Of course, this further restricted the list of possible systems, particularly since we also wanted a system our students could identify with. For example, the Savant Web Server [1] would meet the language and non-GUI requirements, but would not be easy for the students to interact with.

Given our computing resources, we needed to limit the open-source software options to systems that would run under Windows 98. Due to the Linux heritage of many open-source systems, this somewhat restricted our choices. We found that many of the systems that would run under Win32 required “Unix-like” compilers and libraries, instead of the Microsoft Visual C++ IDE our students were used to.

In addition, the software we chose had to be large enough to simulate “real world” software. In order to overcome the difficulties enumerated in the previous section, the software had to be much, much larger than any systems students had encountered to date.

Finally, we wanted software with very inadequate documentation, in order to further drive home the importance of documentation to our students. Via such a “worst case” introduction to large software systems, students would be able to gain a personal appreciation for all kinds of software documentation.

Based on these criteria, we chose the SLASH'EM adventure game [2], a descendant of the popular role-playing game Rogue. The game is written in C, will compile under several different compilers and architectures (Win32, DOS, Unix, MacOS, and others), and runs in console mode. The game can also be compiled to run with VGA graphics tiles. The distribution source we used came as a 4 MB compressed tar file. Once the distribution was uncompressed, there were 271 C source files and 135 header files required to build the game. Other than inaccurate read-me files regarding installation (see below), the only real documentation for our distribution of SLASH'EM was sporadic source code comments.

Project activities

During the project, students had to complete several activities. These activities were spread throughout the semester, with due dates that were established in the syllabus on the first day of class. In this way, our project thread ran through everything we did in the class, rather than being back-loaded toward the end of the semester. Students completed the project in teams of three or four, with the knowledge that each team was in direct competition with the other teams for project scores.

For the first phase of the project, students had to successfully build the SLASH'EM game. This was no small task, as the distribution of SLASH'EM would not build without making several undocumented changes to the system makefiles. Students had to decide what changes to make, based on several "clues" in the documentation, and on compiler error and warning messages. In addition, we needed to use the freeware djgpp C / C++ compiler [3] to build the game, and this was the first exposure for many of our students to a command-line driven compiler (as opposed to the Visual C++ IDE). The first indication that the project was on the right track was the shock of our students when they discovered the ten-to-fifteen-minute compile times for SLASH'EM.

For the next phase of the project, the students had to produce some initial documentation for the system in the form of a function structure chart. This forced the students to get into the source code and gain at least an initial understanding of where the various operations of the game were located. Along with the structure chart, the students had to provide a sentence or two describing what they thought each function in their structure chart did.

In the next phase of the project, the students had to find and correct three logic errors, inserted into the system by the professor. The students were given written bug reports describing the symptoms of the logic errors, but no other information to help them find the errors. All three bugs were intended to mimic errors that could easily be made by programmers in the "real world." The changes made to introduce the bugs, and the symptoms of the bugs, are summarized below.

The first logic error caused the player's character to frequently die when they kicked an object in the game. At times during the game, players may need to kick objects (to open a

locked door, for example), and kicking a hard object is supposed to do damage to the player. However, this error caused the damage inflicted on the kicking player to often be fatal. This error was inserted into the system by changing a single function call of the file dokick.c (line 966), from

```
losehp(rnd(ACURR(A_CON) > 15 ? 3 : 5), kickstr(buf),
            KILLED_BY);
```

to

```
losehp(rnd(ACURR(A_CON) > 15 ? 3 : 51), kickstr(buf),
            KILLED_BY);
```

The change from ‘5’ to ‘51’ meant that the player was quite likely to be killed by the damage sustained during a kick.

The second logic error caused every object the player picked up to be counted as gold, regardless of what the object really was. In normal game play, players would find weapons, scrolls, potions, and other types of objects to use; this bug turned everything to gold as soon as it was picked up. This bug was introduced by changing line 330 of the invent.c file from

```
if (obj->oclass == GOLD_CLASS)
```

to

```
if (obj->oclass = GOLD_CLASS)
```

The change from a comparison to an assignment caused every object picked up to become gold, and of course caused no compile-time errors or warnings.

The third logic error caused one of the VGA graphics components of the game to update improperly. At the bottom of the SLASH’EM screen, there is a status bar reflecting the player’s remaining hit points. This bug prevented the status bar from showing the player’s reduced hit points, under certain circumstances. The bug was introduced by changing line 1791 of the hack.c file from

```
flags.botl = 1; /* Update status bar */
```

to

```
flags.botl = 0; /* Update status bar */
```

Once again, a simple typographic error in the source code was able to produce noticeable flaws in the game, without causing any compile-time error or warning messages.

In the last phase of the project, the students had to extend the game in ways of their own design, and present their improved games to the class. The size of the SLASH'EM game meant that the students had many different opportunities for improving the game or adding new features. The students were told that they were competing with the other teams during the project, so that the team with the "coolest" new features would receive the most points for this portion of the project.

Educational outcomes

This project was very valuable for our students, as it allowed them to experience working with a "real-world" system, rather than with a small toy built during the course of the semester.

The students gained experience in building large, complicated software systems. The large number of source files and the complicated build procedures were new experiences for the students, who previously had only compiled programs consisting of a few source files. In addition to gaining experience in the complexities of compiling large systems, the large number of source files helped illustrate the need for a source management and versioning system. Several groups, in the process of adding special features to their games, introduced fatal logic errors. They had not kept backups of their previous versions of the game, and so were forced to start over from the initial "clean" source files. These instances provided good instructional opportunities, to emphasize the importance of source management and versioning.

The students gained experience with the difficulties of producing documentation for a large system that lacks any real documentation. The students were forced to dig into the source code in order to produce the structure chart and function descriptions. Through this activity, they were able to get some idea of the design and operation of the game. More importantly, however, they started to realize how much they did not know about the system, and how much more they could have known if adequate, up-to-date documentation had been available. For example, during our class discussions on data dictionaries, the students realized how much of a benefit a data dictionary would have been to them during the project. This real-life illustration of the necessity of adequate documentation was perhaps the most important learning outcome of this project.

The students gained experience as maintenance programmers when they had to locate and fix the bugs planted in the SLASH'EM game. Through their knowledge of the system's architecture (gained in the documentation activity) and through the descriptions of the bug symptoms, the students had to locate, diagnose, and fix faults in the system similar to faults that could easily appear in real-world software.

Finally, students gained more maintenance experience through the improvements they added to the game. Each group added different features, ranging from modification of the rules (characters allowed to eat items other than food), through modification of the interface (new VGA tiles and the ability to start a new game without returning to the

DOS prompt). Some groups were more successful than others during this phase. In particular, groups that put off the addition of new features until close to the deadline, or who tried to do too much during the allotted time, were unable to get much accomplished.

All in all, this project was very valuable for our students, in that it illustrated in a hands-on manner many of the concepts in software engineering that we were formerly only able to lecture about. It's one thing to discuss the importance of documentation, versioning, source management, etc., in class, but quite another to discover their importance on your own through an actual project.

Lessons learned

We plan to make a similar open-source project a permanent part of our software engineering course. We have also added a second, follow-on software engineering course that will allow students to practice a full cycle of software development, after they have been introduced to the required concepts in the first course.

Based on the way we introduced bugs into the SLASH'EM game, we recognize that industrious students could have located the bugs by downloading "clean" copies of the source code from the internet and then using a file compare tool to look for differences between the clean and buggy code. For the next project, we will prevent this problem by using a search and replace utility to add extra whitespace around parentheses, braces, and other elements in the code. This should not change the compilation or operation of the system, but will make most of the lines in the buggy code textually different from the original code.

In the next project, we will introduce more bugs into the system, so that different bugs can be assigned to different teams. In this way, each team will be forced to find and solve their own bugs. Once all the bugs have been found and resolved, the teams will be required to document their fixes and distribute them to the other teams. This will help improve the communications skills of our students.

In addition, in the next project, we plan to require the teams to implement at least one improvement to the system based upon "customer" requests, in addition to the improvements the teams implement on their own initiative. In this way, we will be able to move through an entire iteration of the maintenance life cycle, from analysis to design to implementation. This will give students more structure and guidance in this phase of the project, and will also allow them to experience a complete, small example of the software life cycle.

Our experience with this project has convinced us that the use of carefully selected open-source software systems can be an integral part of an undergraduate software engineering course. Open-source software projects allow students to have a much more meaningful software engineering class than would be possible with other types of projects.

References

1. Savant Web Server home page, http://savant.sourceforge.net .
2. SLASH'EM home page, http://slashem.sourceforge.net .
3. djgpp home page, http://www.delorie.com/djgpp .

Acknowledgements

Thanks are owed to the Doane Information Science and Technology Advisory Committee, who first suggested using open-source software in the software engineering course.