# Using Software Maintenance and Evolution on Large Open Source Projects In An Introductory Software Engineering Course

Keith Gallagher, Mark Fioravanti and Alyssa Marcoux
Department of Computer Sciences and Cyber Security
Florida Institute of Technology
Melbourne, Florida, USA 32901
Email: {kgallagher@|mfioravani1994@my.|amrcoux@my.}fit.edu

***Abstract*—Background: Introductory undergraduate software engineering courses are rife with pedagogical problems. There is the matter of presentation. It appears that most of the widely used texts use a breadth-first approach to the central ideas of software engineering: requirements, architecture, design, code, test, etc. The approach is important and outcomes crucial as this is an introductory and is used as a feeder course to advanced individual courses in requirements, architecture, design, etc. When using this rubric, the course project is usually a greenfield development effort: a project is started from scratch and pushed through to delivery, as each major area of the software engineering process is covered in the course, a deliverable is constructed. So as much as we rail against it, a waterfall model is implied, because the project follows the course content delivery.**

**Aim: So, we ask: what happens when the course content follows (is directed by) the project? That is, what happens when the project is delivered first, and as various questions arise about the usual software engineering issues (design, requirements, etc) are addressed via the mechanism of the project?**

**Method: The project presentation method that we use is not greenfield; projects are not developed from scratch. We use mature open-source projects, with long revision histories, many open bugs and many feature enhancement requests. Students select projects of 55 - 3000 KSLOC, made up of 470 to 6300 files, in as many as 34 languages. The course project is then selecting a collection of issues (bugs and enhancements), finding the issue in the code base, and making a verifiable change, (without damage to the existing system).**

**Results: The results are interesting, to say the least. Some project teams write fewer than 100 lines of code for the entire course, after they find out where.**

**Limitations: Usual limitations apply: one semester course; undergraduate students; various technical majors and programming ability, and familiarity with the implementation language.**

**Conclusion: By turning the project of the introductory software engineering course into a software evolution exercise on a sophisticated system, the major topics can still be easily introduced and examined. Contributions to the open source community can be made.**

## I. Introduction

### A. In the Beginning

A collection of American college sophomores (second-year students, in their second term) assemble. Computer Science and Software Engineering students following the catalog plan have had 2 courses in Java and one semester of Algorithms and Data Structures, and they are also learning C++ this term. There are other upper division students from various engineering disciplines; these students may or may not have been in an introductory course in C++.

It's not the usual first day of class, with a course introduction and syllabus. Instead, students' names are placed in a hat, then drawn out to form software engineering project teams. The second day, the teams perform a Java debugging exercise from Barr [1]. On the third day a collection of projects are introduced to the students. These vary in size from 56,000 lines to some 3 million lines, when measured by a simple line count. The repositories and issue tracker databases are introduced. The teams then meet, as a lab exercise, to browse the projects and the issue repositories and think about which one they might like to work on. Before these projects were introduced, the largest piece of code that anyone in the room had worked on was less than 1000 lines (by themselves).

### B. Playing Together and Putting Your Toys Back

So begins the Florida Institute of Technology course CSE 2410, Introduction to Software Engineering, whimsically subtitled the same as this subsection. We present and share an approach to introducing Software Engineering to students with no team project experience via a *major* maintenance and evolution exercise. There is one caveat: this presentation is about *learning* to do software engineering, not teaching it. [3] An *environment for learning* is established. The outcome is up to the student.

### C. Outline and Contribution

This paper outlines the way we use the *project* to guide the course, using Rajlich [2] as the course text. It is this approach that we have been encouraged to share. It includes

- Course administration and grading.
- The available projects.
- A team evaluation approach.
- Laboratory administration.
- Group grading.
- Examination approach.

- A "Digression" section that lists other computing and software engineering topics covered that depart from the text; they may appear to be fun but each has a pedagogical point.
- A summary of project outcomes.
- Student stories.
- A Perspective from the teaching assistant.
- Contrasts with others who have used a similar approach.
- A short reflective perspective.

## II. Basic Course Outline

### A. Administration

Rajlich [2] chapters 1 - 11 (*Introduction* up to and including *Conclusion of Software Change*) form the basic structure of the course. The term is 14 weeks; class has a lecture - laboratory format with 6 hours of contact time. A laboratory is crucial for at least two reasons. First, there is a fixed time in each student's schedule in which the entire team can have an uninterrupted meeting, and *everyone can attend* as there are no timing conflicts. Secondly, it provides the instructors with a time to have individual sit downs with each team to audit the progress and address any technical (or social) problems.

Given the project-based nature of the of the course, it appears sometimes that the content lectures from the text are squeezed around the project concerns. And indeed they are supplementary *to the project*.

### B. Grading

*The projects are graded as a team*. This is, of course, disconcerting and perhaps upsetting to some, especially given the nature of the team formation, and current academic practice. The response is thus: When you get hired by XYZ corporation after graduation, you are not asked if you want to work with your friends; you are placed on a team and told to get the job done. Moreover, in industry, *projects*, not individuals, succeed or fail. These ideas follows Harvey, who contends that "connection with other persons is a requirement for psychological and physical survival. Alternatively any act (requiring that people work alone) leads to breakdown both mental and physical." [3]

Attendance is required at Laboratory Reviews. (See section IV.) Failure to attend *any* scheduled laboratory review results in the grade of **F** *for the course*.

*1) It Gets Worse:* From the syllabus: "If you do all work perfectly on every evaluation instrument, you will have earned a B." and "To earn an A in my class, you must show intellectual initiative." [4] Enterprising students often ask: "How do I show initiative?" The response is: "Please read the supplemental and suggested text by Nelson, *1001 Ways to Take Initiative at Work* [5]" and not much more.

*2) Then It Gets Better:* How the grades are allocated:

- The syllabus will have Grading TBD.
- Marking allocation is not determined by me.
- Marking allocation is divided between projects and other work.
- It is determined by the class as a whole, by a vote.
- The allocation of percentages between project[s] and / or examination[s] is determined later in the term, by vote.
- This gives you [the student] a chance to control your own outcomes, after you have had on opportunity to assess how your effort allocation, learning style and methods of presenting work mesh with my evaluation rubrics and techniques.

*3) And Even Better:* Examinations are designed by the instructors and students in a class. This serves as a "review session." It also takes the guess work out of what topics to study. The exams are sprinkled throughout the term with 3 questions in an hour session. The questions are of two forms. Performance: i.e., compute the supplier slice of this node in this graph. And Reflection: i.e., what would happen to the field of Software Engineering if one of Brooks' five essential difficulties were negated?

## III. The Projects

As reported above, team creation is the first order of business as "This course is the project!" Teams are always four members or less. The second day serves as team-building exercise; the team is given only one or two copies of the errant program, so they have to share! At the end of the exercise, *the team* reports its results to the instructors. Once teams have been formed, they must sit together in lectures.

The class finds the project introductions daunting, to say the least. But before they can catch their breath, the Enterprise GitHub server is introduced. The first laboratory exercise for the students is to get their respective teams set up on the University's GitHub server. (See section IX.) Each team can choose its own (socially acceptable) name. The instructors are given complete access to the group project. The *issue tracking* feature of Enterprise GitHub is central to the operation and success of the project, but it does generate a lot of email.

Figure 1 lists the possible projects and bug database location; figure 2 shows the number of files and the lines of codes. The measures are admittedly coarse, but consider that the students have never seen *anything* like these! For instance, 7Zip has 968 files, Most are C and C++; there are about 100 other kinds: assembler, makes, Windows Support files, &C. At the other end of the spectrum, node.js has some 6400 files in 34 languages, that include dialects, batches, scripts. Pascal, Ada, Lisp and SAS are even sprinkled in! Pidgin was abandoned in the Spring of 2016 as it has fallen into disuse. Atom and VLC were added in the Spring of 2016.

Different groups can select the same project. Different groups can even select the same issue. Different groups working on the same project can communicate and are encouraged to do so [3].

## IV. Laboratory Reviews

### A. Managing Groups and Group Meetings

Meeting with a collection of people who barely know each other, have been thrown together on a project and must depend

| Possible Projects | Project Webpage | Bug Database/Issue Tracker |
|---|---|---|
| Filezilla | https://filezilla-project.org/ | http://trac.filezilla-project.org/ |
| Node.js | http://nodejs.org/ | https://github.com/nodejs/node/issues |
| Wireshark | https://www.wireshark.org/ | https://bugs.wireshark.org/bugzilla/ |
| 7-zip | http://www.7-zip.org/ | http://sourceforge.net/p/sevenzip/bugs/ |
| Notepad++ | https://notepad-plus-plus.org/ | https://github.com/notepad-plus-plus/notepad-plus-plus/issues |
| VLC Media Player | http://www.videolan.org/vlc/ | https://trac.videolan.org/vlc/ |
| Atom | https://atom.io/ | https://github.com/atom/atom/issues |
| Elastic Search | https://github.com/elastic/elasticsearch | https://github.com/elastic/elasticsearch/issues |
| Pidgin | http://pidgin.im/ | https://developer.pidgin.im/wiki/OpenTickets |

Fig. 1. Possible Projects with Issue Locations

| Project | Number of files | Lines |
|---|---|---|
| Filezilla | 508 | 181K |
| Node.js | 6390 | 1589K |
| Wireshark | 3822 | 3026K |
| 7-zip | 968 | 176K |
| Notepad++ | 745 | 305K |
| VLC Media Player | 2357 | 572K |
| Atom | 470 | 56K |
| Elastic Search | 4878 | 488K |
| Pidgin | 1039 | 375K |

Fig. 2. Simple Size Data for Projects

on each other is fraught with difficulties. Mixing in "one from every tribe and nation" further complicates matters.

### *B. Group Evaluation*

The instructors must firmly, yet tactfully, set the tone. We have accidentally stumbled upon *psychological safety*, "a sense of confidence that the team will not embarrass, reject or punish someone for speaking up," [6] quoted in [7]. A question directed at one team member cannot be answered by another. No interrupting, eye-rolling or sighs. The instructors focus on the person speaking and follow-up as needed. Everyone gets a chance to show what they have done. Meetings start and end with: "Are we talking to each other? ... and listening?" and "Is everyone being respected?"

A home grown web service permits the students to evaluate themselves, their teammates and the group as a whole. At each fortnightly audit meeting, students evaluate their own progress and contribution, their individual team members progress and contribution, and the group as a whole. For individual and other team member evaluations we use a form adapted from ReadWriteThink [8].

1) Identify yourself.
2) Describe your responsibilities in the group.
3) Assign yourself for grade: A - F.
4) What percentage of the work did you complete? (Total team percentage must add up to 100.)
5) Did you complete your tasks? If you did not complete your assigned tasks explain why.
6) How would you rate your participation in this group?
   a) I was incredibly involved.
   b) I completed my work but was not otherwise involved.
   c) I completed most of the work assigned to me but other group members contributed more than I did.
   d) I did not contribute.
   e) if you need to defend your answer, do so here.
7) For each other group member:
   a) Name.
   b) Assign this person a grade: A - F.
   c) What percentage of the work did this person complete? (Total team percentage must add up to 100.)
   d) This person was incredibly involved.
   e) This person completed all assigned work but was not otherwise involved.
   f) This person completed most of the work assigned but other group members contributed more.
   g) This person did not contribute.
   h) if you need to defend your answer, do so here.

For group process evaluation we use an adapted form from Christianson [9]. Rate your *entire team* on a 1 to 5 scale with respect to the following:

1) **Goals:** Goals are unclear or poorly understood, resulting in little commitment to them... Goals are clear, understood, and have the full commitment of team members.
2) **Openness:** Members are guarded or cautious in discussions... Members express thoughts, feelings, and ideas freely.
3) **Mutual trust:** Members are suspicious of one anothers motives... Members trust one another and do not fear ridicule or reprisal.
4) **Attitude toward difference:** Members smooth over differences and suppress or avoid conflict... Members feel free to voice differences and work through them.
5) **Support:** Members are reluctant to ask for or give help... Members are comfortable giving and receiving help.
6) **Participation:** Discussion is generally dominated by a few members... All members are involved in discussion.

7) **Flexibility:** The group is locked into established rules and procedures that members find difficult to change...Members readily change procedures in response to new situations.
8) **Use of member resources:** individual's abilities, knowledge and experience is not well utilized...Each members abilities, knowledge, and experience are fully utilized.

Self aggrandizement in self reporting is rare. Students who are not participating are honest with their lack of contribution, ultimately. While we have no data to support this, it seems that since *everyone* is submitting, we have a better approximation of the truth. Moreover, this gives the instructors a chance to privately meet with under performing members, or under performing groups.

Each term the same story repeats. At the first evaluation everyone gets an A, and the work has been equally divided. At the second evaluation, a member who is not doing their part gets a B, and a slightly smaller percentage of the effort, while everyone else has their value increased. Finally the charade is abandoned and a member who is not participating gets the same abysmal marks from all the other members. This person is invited in for a private chat.

We believe that what makes this work is that these responses are submitted online, with submitters known only to the instructors, but not seen by other team members. Originally, it was paper and pencil. The next incarnation was a quiz through the Learning Management System. We now have a (brittle) web service to do this.

Usually once a term, a team "crashes and burns." The team is then disbanded and reformed or students are marked individually. Individual students may have their mark separated from the rest of the team when lack of performance warrants.

### C. Laboratories

Once the team selects a project it must be uploaded onto the server and compiled there. *Then* each student pulls the project onto her local machine and compiles it there. These activities are demonstrated in a laboratory. This is not as simple as it seems. Students need to find, and perhaps install, the correct IDE on their machines. Then get the project into the IDE and get it to compile and run. Oftentimes, and under our encouragement, the students set up *virtual machines* [10] on their laptops. It provides a clean and clear environment for the task.

The third review - audit is to put the team's name in the *About* box of the selected project. The results must be demonstrated on a local machine *without a complete re-compile*. The necessity of this activity is obvious: small changes, to be made later, cannot induce complete recompilations. (It also serves as a gentle introduction to *concepts and concept location*.) This distinction is new to many who have not worked on any large system before this class; they just recompile the whole system, at 0 cost. When a complete recompile for a small change takes 20 minutes, while the instructor is watching, the lesson is learned.

All the while, the teams have been surveying the project's issue repository. The teams select a minimum of five issues, one of which must be an enhancement request and one an error repair. The next review is to *demonstrate* these issues to the instructors.

The next demonstration to the instructors is a *preliminary location of the selected issues in the code.* The demonstration need not be as specific as a line of code (or even be correctly located!). The goal is to get the team digging around in the project.

Finally, the teams show the instructors the *exact location* of the issue.

Audits through out the rest of the term are to check progress, which we know anyway because GitHub is telling us so!

As the term winds down, we remind them that they must present at the final, discussed below.

## V. Examinations

As noted previously, examinations are sprinkled throughout the term. They are straightforward and student designed. The idea behind having students help build the exams is that everyone knows what the topics are.

### A. The "Mid-Term"

The mid-term examination is an evaluation of the *instructor*, submitted via the Learning Management System as follows:

1) Is the textbook easy to read and understand?
2) Are you going to keep it?
3) What are the objectives of this class? Are you accomplishing them? or not? Am I helping you to accomplish them? or not?
4) Has your curiosity been stimulated? how? why?
5) Do you think and solve problems better now than when you started this class?
6) What are the major weaknesses of the course? What is least important thing you have learned?
7) What are the major strengths of the course? What is most important thing you have learned?
8) What are your suggestions to improve the course? This includes things to add and things to remove. What did you think of the class format? time? etc?
9) Would you recommend this course, with this instructor, to another student? why? or not?
10) Would you take or recommend this instructor for another course? why? or not?
11) Jot down some thoughts, coherency optional, about your 'lessons learned' in the this class, both technical and personal.
12) Anything else?

### B. The Final

The final exam, which is a team presentation, is handed out with the syllabus. The final project presentation includes, at least, a memory stick with Individuals and Team Name and

- Source
- Executable
- List of Fixes
  - For each fix:
  - Estimated Impact Set
  - Actual Impact Set
  - Source Files Changed
  - Estimated and Actual Times to fix
  - Before/After Screenshots
  - Tests
    - To Reproduce
    - Validate the fix
- Comments: interesting things that happened
- Unmaintainable Code [11] (see VI-D.)
- Summaries of team meetings: dates, attendance, problems, actions...
- Management issues: technical and organizational
- Any presentations given throughout term
- The final presentation
  - a thorough demonstration of one issue from above
  - a summary of the rest

## VI. Digressions

### A. The Professor

I (KG) spend a day describing my interest in software maintenance and evolution. I start with Weinberg's charming and inspiring (to me) "Kill That Code!" [12] and introduce my own interests. I am always on the lookout for good students; pointing them to my work is their first quest.

Later in the term, I present the results of the "Mid-Term" from above. It makes for a fun day. When contradictory answers are given by different folks for the same question, ("Stop with the storytelling!" "I love your storytelling!") and interesting discussion ensues. It also gives a chance for reset, to go back to the beginning, see what the course is about, and get a seasoned view on things that may have been confusing at the beginning.

The midterm indicates that I either "make it big" or "fail miserably" with most reporting accomplishing the objectives, a few absolutely hating the class, and almost none falling in the middle. [3]

### B. How Well Do You Play with Others?

A lecture on team structures, from democratic to hierarchical is presented. The old idea of "egoless programming" is introduced. Technical roles, work styles and communication styles are discussed. *Programming teams* with collective goals, collective egos, internal walk-throughs and frequent code reading are stressed.

Are you Rational? Intuitive? Introvert? Extrovert? Which combination? The point that cannot be over stressed is that this information is for *self knowledge* not judgment of others knowledge. When a person has some idea of how they themselves behave under stress and can recognize it, everyone wins. No one who has the courage to find out about themselves, and share it with others, cannot have that information used against them. "You rational extroverts are so bossy!!!" is a no-no.

### C. Help!!

Once or twice a semester, sometimes more often, student sends an email or requests a private meeting. These contacts are: "I have absolutely no idea what to do in this course," or "Everyone in my group is so much smarter than me that I am a drag on the team's progress."

Both concerns are addressed in the same manner. A lecture is devoted to the issues and concerns raised by those students. The people who have made private contact get to see that they are not alone, and, as in the usual case, about half the class will *publicly admit* having those same concerns. Everyone is relieved. Working a large, undocumented system does have it stressors!

### D. Unmaintainable Code

Early in the term, a day is spent on Green's "How To Write Unmaintainable Code" [11]. Students find the digression disturbingly charming, especially when asked: "None of you have ever done this, right?" However, the teams are required to report on Unmaintainable code discovered throughout the term. It's all fun and games until one is reading, changing and recompiling it.

### E. Compilation and Linking

Link editing and object file management are covered in more depth than in Chapter 3. MonkeySort [13] is used as an example of separate file compilation. (More on MonkeySort follows.) The program is easily read. It has `extern` variables. A simple makefile is used to display dependencies. Use of the (*nix) link editing tools (*ar* and *ld*) are demonstrated: how to build object libraries; how to edit object libraries; and how link to one's own object library. Static and dynamic linking are demonstrated. The point of this exercise is to show students that programs and systems are much, much more than the actual source. Even students who use one of the popular IDE's, with its left side class management panel and *build*, button are surprised at the underlying mechanisms.

Evidently students still struggle with object libraries: finding the correct ones from the Internet and then getting them to link with their chosen projects. (See section IX.) This is also the a first experience of editing "non-readable" files; *Emacs* in hex-mode is an eye-opener!

### F. MonkeySort [13]

It's abstract:

> Monkeysort is a pedagogical program that turns the usual concern of efficiency upside down by attempting to be as dumb as possible, yet still correct. In this program, whose inner workings are accessible to all students, MonkeySort exhibits significant ideas that

are central to computer science: partial correctness, generate-and-test solutions to NP-hard problems, Stirlings approximation of n!, and subtle applications of the use of permutations. It also demonstrates central software engineering ideas: integer overflow; the use of coverage tools; CPU monitoring tools; and timing analysis approaches more sophisticated than mere statement counting. An appropriately tailored classroom discussion of MonkeySort can be used as fodder for graduate student homeworks, or to illustrate to the non-scientist exactly what it is that Computer Scientists do.

Big ideas abound in MonkeySort. It is *NP*. It is *partially correct*. It is a simple `while` statement. Students who have never read or thought deeply about `while`'s are often surprised to learn that the negation of the entry condition is true when the loop exits! The entry condition in this case is `!(checksort( a, n))`, so whatever `checksort` does, it is true when the loop exits. The issue of note is that the loop may never exit!

The demonstration computer has eight processors. Running the CPU-intensive MonkeySort many times in the background constitutes a Denial of Service attack, which is easily shown by examining the *system monitor*. *Profiling tools* give an empirical, rather than analytical, demonstration of sorting times. It is easy, if somewhat unpredictable, to have MonkeySort to achieve an *integer overflow*, which makes counting data useless. Finally, students see for the first time the astounding Stirling approximation for $n!$ that inexplicably involves $\pi$, $e$ and the square root!

### G. Testing

Testing is introduced via the famous triangle problem of Myers [14] and a faux "quiz". The simple problem statement (A program accepts three integers as input. These are taken to be sides of a triangle. The output of the program is the type of triangle determined by the three sides: Equilateral, Isosceles or Scalene. Write a set of tests for this program.) is placed the projector screen and students generate tests. Six issues arise: that *expected output* is crucial to any test; that *viewing the source* is not necessary to testing; that there is a *maximum integer*., and when it is used as input the output is *unpredictable*; that the integers *may not form the sides of a triangle*; that the input *need not be integers*; and that there may be *more or less than three* input values. Students who usually test their programs with a few simple inputs are surprised that we can come up with about 100 tests. Moreover, the students realize that they will need a program to test this program; scripting is introduced.

We have about a half dozen implementations, in various languages, of the triangle program. Exploratory testing [15] is applied here to again demonstrate that the source is not needed for a test. Students use the information gleaned from the Myers' "quiz" to thoroughly exercise the program. The same test can behave differently depending upon the implementation language, especially in languages that support arbitrary integers.

Coverage tools are introduced in this digression. A particularly "branchy" and state-variable filled version of the triangle program from Jorgensen [16] demonstrates how difficult covering a "simple program" can be.

### H. Opening Exercises

Occasionally, class is opened with a puzzle from Feuer's puzzle book [17]: `a = a++ + ++a`? or a particularly bizarre snippet from the obfuscated C contest [18]. The obfuscated programs afford another opportunity to *edit executables*. Careful reading and outside-the-box thinking are re-enforced with some infuriating puzzles [19].

### I. Programs as State Machines, Debugging and "Proofs"

A program has a collection of live variables, and those variables have values. This collection of variables and values is called the *state of the computation*. An elementary version of a Hoare triple is introduced [20]. Assertions are noted (from earlier courses).

Programs are redescribed not as a collection of *statements*, but a collection of *states*; the statements are artifacts that change the states. This "state of computation model" is illustrated in a visual debugger. When a statement changes the state it is easily seen in the watch window. And " How do we know that our program is wrong?" It's in the wrong *state*, of course!

Then without thinking of which *statement* should be used, the question is: What is the *state* that the program should be in? The statement writes itself.

The "math haters" get uneasy when they find out that a program is really a mathematical proof. The input state is the hypothesis, the output state is the theorem, and the program (all those statements) is the proof!

### J. Fun with Max

A collection of *C* and *C++* programs illustrate again that computers cannot be trusted to compute, especially with numbers. The harmonic series, a divergent sum, actually has a value! `x = x + 1` leaves `x` unchanged for certain values, such as `FLT_MAX`, and `x` *decreases* when `1` is added to INT_MAX. Machine epsilon is calculated; the `nextafter()` function is examined.

### K. Reading, Thinking and Writing

A day is devoted to "The Task of the Referee," by Smith [21]. While ostensibly about refereeing articles, it also give a clear set of instructions on *writing* them (and other academic papers) too.

A day is spent on Perry's tale of the "Abominable Mr. Metzger" as reported in "Examsmanship and the Liberal Arts: An Epistemological Inquiry" [22].

## VII. Outcomes

Most groups complete the issues, and get a B. Occasional teams and Individual students show initiative. Not all teams submit changes back to the project repository.

> ... students either "make it big" or "fail miserably" In a teaching environment, most people fall into the middle (Some say that their performances form a bell-shaped curve. I have concluded that well-shaped distributions of performance in academia are artifacts of an environment in which teaching is stressed. In a learning environment, performance is generally bimodally skewed, with most persons performing very well, a few performing very badly, and almost none falling in the middle.) Harvey [3]

This course is about the *process* and not the *product*, so reporting on the size and number of changes misses the point. It is a major accomplishment when the student finds the *one line* that needs an additional parameter, makes the change and makes it work.

The actual data collected on project and submitted in the final is secondary to the task of actually collecting it. Students are typically over prepared at the final.

Figure 3 gives some historical data for the system selected most often, 7Zip. It has been selected 10 times over the years. There are not 50 distinct changes because some groups implement the same change and other groups find the same change available in the next term. The other projects have similar results.

Not all resubmit to the repository. It is not required. The students who do submit to the repository get the unfortunate experience that too many have when submitting to open source: their properly submitted changes are "flamed" yet found in the next release. Only two groups intended to keep contributing; only one individual did.

| Different Teams | Different Changes | Biggest Change & Impact | Smallest Change & Impact | Submitted to Project |
|---|---|---|---|---|
| 10 | 35 | 40 loc<br>6 files | 1 loc<br>1 file | 4 of 10 |

Fig. 3. Some 7Zip Data

## VIII. Student Stories

The end of term evaluations are not available [3]. Here are some quotes from "the midterm" and student posts. They are subject to observer and confirmation bias. [Edited for grammar and spelling.]

- That first day was very interesting...
- This is not an easy world and I am pretty sure he knows that, that is why he pushes us the way he does.
- I had no idea of what I was getting myself into...
- I believed I was ready for this class and ready to pass with ease. [Failed student.]
- This class is here to show us and teach us how to read another person's code and understand it.
- The hardest obstacle to overcome before progress could be made was compiling the open source program.
- It would have been useful to go over basic conventions in the open source community like build scripts, and go over a sample dependency problem resolution.
- Treat it more like a real-life job. Have people who feel confident or have experience volunteer to be their team's "lead"and give them the ability to assign tasks and deadlines to others and themselves. They should also be the ones to schedule team meetings and ensure the project is on track. The course is great to introduce people to coding in the open source world, but I felt the administrative side of things was lacking a bit and the projects might be able to flow more smoothly with a system like that in place.
- CSE 2410 was the first major software project I worked on. I believe that for "newer" programmers 2410 is a course we are very nervous about. We feel like we have something to prove.
- I thought the course was going to be much easier than it ended up being...
- I can say I contributed to the Open Source community in my job interviews.
- I would like to see more time spent on how to build large projects, maybe covering maven/gradle to better understand how to use them as well as to understand how they work. I think a little bit more time could be spent on git, and dealing with things such as how to resolve merge conflicts.
- This course gave me everything I needed to know to succeed in my summer internship.
- Find open source projects that aren't stale, find open source projects using new technologies especially ones that aren't necessarily even well documented, and where the answers to bugs require both research and originality. Projects like I2P, hypervisors, rabbitmq, etc.; those projects are what truly stretch people.
- I was pretty excited to finally get my hands dirty with my first open source project.
- What helped the most getting started was the idea of having a safe-net in case things went wrong: the Git repository. I hadn't used Git too much prior to this class and I imagine it's the same with a lot of my classmates. I wasn't worried about trying to poke around looking for a bug without breaking things because I knew that I could tinker with any part of the program I wanted to without consequences. If things went south, I could always roll back to the last master branch version and try again. In that sense, my goal was to break things.
- It was the first coding experience I had where I'd be reading/understanding less than 1% of the codebase - a scary thought when every program I've worked with previously had been my own.

It's clear that we need to spend more time on building code bases and managing repositories. The presumption has been this is just another tool, another environment, another installation that students have to perform, They are doing this all the time on their rooted phones, gameboxes and laptops. Evidently, that experience is insufficient.

## IX. The Graduate Student Assistant (MF) Says:

From the perspective of the GSA, there are a number of issues that the assistant must be prepared to support throughout the semester. The four major areas of concern are: issues with version control, resolving compiler dependencies, transitioning to an understanding of event driven programming and documenting progress. Most students participating in the class are likely second or third year computer science or computer engineering students. The projects they have previously worked on as part of their coursework have not been sufficiently complex or required working in larger teams to warrant the use of version control.

The first problem that students will encounter with version control is understanding and setting up their team repositories. Students tend to require a few days to setup the repository and perform their first commit with the code from their selected project.

As the semester progresses two different issues tend to surface; performing rollbacks to a previous commit and merging branches. Some teams have the problem in which one student fixes an issue and commits a change, but as all of the students are working from the master branch, the fix ripples through the code and causes other team members builds to suddenly stop working. Performing a reset on the branch is not difficult with a version control system such as git, but students are often not familiar enough with the terminology to know what to search for when attempting to reset to a previous commit.

The other issue is encountered with teams which have a student who is previously familiar with version control and has created multiple branches so each team member can work on their own issue in their own separate branch. These teams know enough to get themselves into situations with the version control system that are not easily resolved, and may require assistance to merge all of the branches back into the master branch at the end of the semester.

The next major issue after teams have successfully established their repositories is often encountered as students attempt to create their first build. Most of the projects available for students to choose from are written in C/C++ while most students have only had a single semester previous with C/C++. Students are faced with issues in getting complex projects to compile and some of the build processes online are inaccurate due to updated dependencies or compiler options. For example, Filezilla has a number of specific dependencies which must be resolved during the build process and some projects such as 7-zip require that the build scripts be edited to account for newer compiler settings and options. Although most students are working from a Microsoft Windows host, some teams discover that it is easier to make use of a virtualization product such as VMware Workstation to build their projects on a Ubuntu guest.

The third problem is transitioning to a new programming paradigm of *event driven programming*. Up to this point most projects that the student has completed as part of their coursework have been simple toy programs which accomplish a single task. Of the selected projects, almost all of them make use of a GUI framework such as Qt, GTK or the Microsoft Windows API. In addition to understanding how to program a GUI the students must deal with Event-Driven Programming. When tracing an issue, students are confronted with an event handler and are often confused as the event handlers do not seem to be directly tied to the programs entry point.

Finally there is the issue of documentation, students often fall into two different categories with regards to documentation at the beginning of the semester; they do not document their progress at all or they record too much detail. Students can often be convinced to documenting their progress when they understand that the issue tracker is a place which can retain notes on discoveries during the process of correcting the issue. The teams which are most successful during the semester often have made use of the issue tracker to record their progress and the wiki to record information that is generally useful to the rest of the team (such as build notes).

## X. Related Work and Comparison

This theme reported herein is not new. Many others have used open source projects and used software maintenance in software engineering classes. This section summarizes, compares and *draws distinctions* between those offerings and ours.

As early as 1992, Pierce [23] noted in this conference, following Cornelius, et al. [24], that students may learn more in a maintenance exercise in a project course than they might in a greenfield exercise. These were necessarily *small projects* with respect to those now found on the Internet, but nonetheless pointed the way that students could learn more by changing existing code than by writing from scratch. Van Deursen and Letherbridge [25] convened a panel at this conference in 2002, to how and where software maintenance and evolution *could and should* be in incorporated into a software engineering curriculum. Rajlich and Gosavi [26] further noted that impact analysis and incremental change were fundamental to a *medium sized* software engineering exercise.

As the Internet and open source communities blossomed Carrington and Kim [27] used open source, albeit in a course on *design and testing*. Buchta, et al. [28], [29] demonstrated that *small to medium sized* open source projects could be used as a class project, with a modicum of support tools. Dionisio, et al. [30] attempt to *restructure the entire undergraduate computing curriculum* around open source, while Xing [31] also used open source in a *graduate course*. Pedroni, et al. [32] studied *motivation and inclination* of graduate students *to continue to work* on open source projects. Nandigam, et al.,[33] used *source code exploration* to examine open source.

Meneely, et al., [34] attempt to assuage the difficulties of using open source by *creating a repository*, ROSE, (A Repository of Education-friendly Open-source Projects) for all aspects of software engineering education.

Rajlich presented a tutorial at this conference [35]. In addition to addressing concept location, change, etc., introductions to *agile programming and the personal software process* were presented. Marmorstein [36] describes a course similar to ours, but has more structure and an *insistence on interaction with the open source community. Students were not required to work in teams.* Dorman and Rajlich [37] report on a *single programmer* working on an open source project. McCartney, et al., [38] have students *reverse engineer* an open source project, then make a change. Teel, et al., [39] investigate open-source *tool usage* in a *two term project*, while Stroulia, et al., [40] focus on *capstone project*. Gokhale, et al., [41] *compare* design - centric and maintenance - centric approaches in the *same software engineering class, and note the difference between the two cohorts*. Rajlich [42] outlines a team-based project course, as first course in software engineering. *The students are graded individually.*

## XI. Perspectives and Conclusion

In 1979, Mills, et al., [43] advocated *reading* programs *before writing* programs. Their approach is wholeheartedly accepted and practiced here. Students can *read* and *comprehend* programs that are much more sophisticated than they could ever write at this stage of their studies.

The course is about *learning* software maintenance, not teaching it, hence the awkward title. The goal is to establish an environment for learning, not an environment for teaching.

Students report that it succeeds. That's enough.

> I tend to agree with Carl Rogers' (1961) "Personal Thoughts on Teaching and Learning." In essence, he contends that anything of value can't be taught, but that much of value can be learned… Harvey[3]

## Acknowledgments

The authors would like to David Binkley, Vaclav Rajlch, The Program Chairs, William Nyffenegger, Luke Wiskowski, Giordano Benitez, Taylor McRae, Pablo Canseco and Borja "Pablo 2.0""Canseco for ideas, suggestions, comments and experience reports. And all previous students in CSE 2410, infuriating and inspiring as they are!

## References

[1] A. Barr, *Find the Bug*. Addison Wesley, 2004.

[2] V. Rajlich, *Software Engineering: The Current Practice*. Chapman Hall CRC Press, 2011.

[3] J. B. Harvey, *How Come Every Time I Get Stabbed in Back, My Fingerprints are on the Knife?* Josey Bass, 1999, ch. 4: Learning to Not*Teach, pp. 71– 84.

[4] K. Gallagher, "Standard stuff." [Online]. Available: http://cs.fit.edu/~kgallagher/Courses/Standard\%20Stuff/Grading.html

[5] B. Nelson, *1001 Ways to Take Initiative at Work*. Workman Press, 1999.

[6] A. Edmondson, "Psychological safety and learning behavior in work teams." *Administrative Science Quarterly*, vol. 44, no. 2, pp. 350 – 383, 1999. [Online]. Available: http://search.ebscohost.com/login.aspx?direct=true&db=bth&AN=2003235&site=ehost-live

[7] C. Duhigg, "What google learned from its quest to build the perfect team," *The New York Times Magazine*, Feb 2016.

[8] ReadWriteThink, "Group assessment." [Online]. Available: readwritethink.org

[9] R. Christianson, "Group process evaluation form." [Online]. Available: http://www.coopzone.coop

[10] "vmware." [Online]. Available: http://www.vmware.com

[11] R. Green, "How to write unmaintainable code." [Online]. Available: http://mindprod.com/jgloss/unmain.html

[12] G. M.Weinberg, "Kill that code!" *Infosystems*, Aug 1983.

[13] K. Gallagher, "MonkeySort," *The Journal of Computing Sciences in Colleges*, vol. 15, no. 3, pp. 70 – 81, February 2005.

[14] G. Myers, *The Art of Software Testing*. Wiley, 1979.

[15] C. Kaner, "A tutorial in exploratory tesing," in *Quality Engineered Software and Testing Conference*. QAI Global, September 2008.

[16] P. C. Jorgensen, *Software Testing: A Craftsman's Approach, 4th ed.* CRC Press, 2014.

[17] A. R. Feuer, *The C Puzzle Book*. Prentice Hall, 1982.

[18] "The international obfuscated c code contest." [Online]. Available: http://www.ioccc.org/

[19] P. Sloane and D. McHale, Eds., *Infuriating Lateral Thinking Puzzles*. Puzzlewright Press (Sterling), 2010.

[20] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969. [Online]. Available: http://doi.acm.org/10.1145/363235.363259

[21] A. J. Smith, "The task of the referee," *Computer*, vol. 23, no. 4, pp. 65–71, April 1990.

[22] A. M. Eastman, Ed., *The Norton reader; an anthology of expository prose*. Norton, 1969, ch. Examsmanship and the Liberal Arts: An Epistemological Inquiry, by William Perry, Jr.

[23] K. R. Pierce, "The benefits of maintenance exercises in project-based courses in software engineering," in *, Conference on Software Maintenance, 1992. Proceedings*, Nov. 1992, pp. 324–325.

[24] B. J. Cornelius, M. Munro, and D. J. Robson, "An approach to software maintenance," *Software Engineering Journal*, pp. 233–236, July 1989.

[25] A. v. Deursen and T. C. Lethbridge, "How should software evolution and maintenance be taught?" in *International Conference on Software Maintenance, 2002. Proceedings*, 2002, pp. 248–250.

[26] V. Rajlich and P. Gosavi, "Incremental change in object-oriented programming," *IEEE Software*, vol. 21, no. 4, pp. 62–69, Jul. 2004.

[27] D. Carrington and S. K. Kim, "Teaching software design with open source software," in *Frontiers in Education, 2003. FIE 2003 33rd Annual*, vol. 3, Nov. 2003, pp. S1C–9–14 vol.3.

[28] J. Buchta, M. Petrenko, D. Poshyvanyk, and V. Rajlich, "Teaching Evolution of Open-Source Projects in Software Engineering Courses," in *22nd IEEE International Conference on Software Maintenance, 2006. ICSM '06*, Sep. 2006, pp. 136–144.

[29] M. Petrenko, D. Poshyvanyk, V. Rajlich, and J. Buchta, "Teaching Software Evolution in Open Source," *Computer*, vol. 40, no. 11, pp. 25–31, Nov. 2007.

[30] J. D. N. Dionisio, C. L. Dickson, S. E. August, P. M. Dorin, and R. Toal, "An Open Source Software Culture in the Undergraduate Computer Science Curriculum," *SIGCSE Bull.*, vol. 39, no. 2, pp. 70–74, Jun. 2007. [Online]. Available: http://doi.acm.org/10.1145/1272848.1272888

[31] G. Xing, "Teaching Software Engineering Using Open Source Software," in *Proceedings of the 48th Annual Southeast Regional Conference*, ser. ACM SE '10. New York, NY, USA: ACM, 2010, pp. 57:1–57:3. [Online]. Available: http://doi.acm.org/10.1145/1900008.1900085

[32] M. Pedroni, T. Bay, M. Oriol, and A. Pedroni, "Open Source Projects in Programming Courses," in *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '07. New York, NY, USA: ACM, 2007, pp. 454–458. [Online]. Available: http://doi.acm.org/10.1145/1227310.1227465

[33] J. Nandigam, V. N. Gudivada, and A. Hamou-Lhadj, "Learning software engineering principles using open source software," in *Frontiers in Education Conference, 2008. FIE 2008. 38th Annual*, Oct. 2008, pp. S3H–18–S3H–23.

[34] A. Meneely, L. Williams, and E. F. Gehringer, "ROSE: A Repository of Education-friendly Open-source Projects," in *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science*

*Education*, ser. ITiCSE '08. New York, NY, USA: ACM, 2008, pp. 7–11. [Online]. Available: http://doi.acm.org/10.1145/1384271.1384276

[35] V. Rajlich, "Teaching undergraduate software engineering," in *2010 IEEE International Conference on Software Maintenance (ICSM)*, Sep. 2010, pp. 1–2.

[36] R. Marmorstein, "Open Source Contribution As an Effective Software Engineering Class Project," in *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '11. New York, NY, USA: ACM, 2011, pp. 268–272. [Online]. Available: http://doi.acm.org/10.1145/1999747.1999823

[37] C. Dorman and V. Rajlich, "Software Change in the Solo Iterative Process: An Experience Report," in *Agile Conference (AGILE), 2012*, Aug. 2012, pp. 21–30.

[38] R. McCartney, S. S. Gokhale, and T. M. Smith, "Evaluating an Early Software Engineering Course with Projects and Tools from Open Source Software," in *Proceedings of the Ninth Annual International Conference on International Computing Education Research*, ser. ICER '12. New York, NY, USA: ACM, 2012, pp. 5–10. [Online]. Available: http://doi.acm.org/10.1145/2361276.2361279

[39] S. Teel, D. Schweitzer, and S. Fulton, "Teaching undergraduate software engineering using open source development tools," *Issues in Informing Science & Information Technology*, vol. 9, pp. 63+, 2012, 63.

[40] E. Stroulia, K. Bauer, M. Craig, K. Reid, and G. Wilson, "Teaching Distributed Software Engineering with UCOSP: The Undergraduate Capstone Open-source Project," in *Proceedings of the 2011 Community Building Workshop on Collaborative Teaching of Globally Distributed Software Development*, ser. CTGDSD '11. New York, NY, USA: ACM, 2011, pp. 20–25. [Online]. Available: http://doi.acm.org/10.1145/1984665.1984670

[41] S. Gokhale, R. McCartney, and T. Smith, "Teaching Software Engineering from a Maintenance-centric View," *J. Comput. Sci. Coll.*, vol. 28, no. 6, pp. 42–49, Jun. 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2460156.2460166

[42] V. Rajlich, "Teaching developer skills in the first software engineering course," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 1109–1116.

[43] H. D. Mills, R. C. Linger, and B. I. Witt, *Structured Programming: theory and practice*. Addison-Wesley, 1979, ch. **5**: Reading Structured Programs. ch **7**: Writing Structured Programs.