"If This Is What It's Really Like, Maybe I Better Major in English": Integrating Realism into a Sophomore Software Engineering Course

Robert R. Kessler
Computer Science
University of Utah
Salt Lake City, UT 84112
kessler@cs.utah.edu

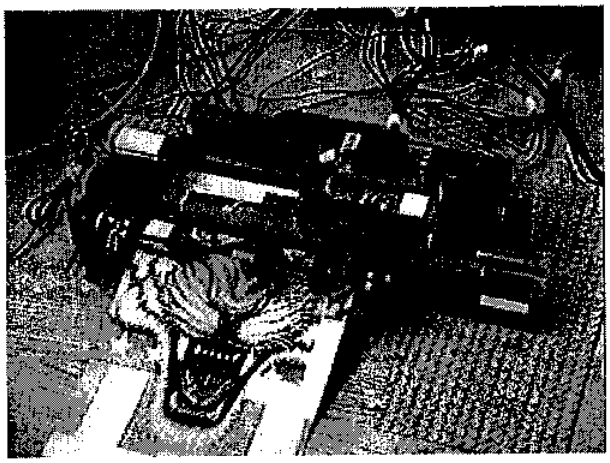
Laurie A. Williams
Computer Science
University of Utah
Salt Lake City, UT 84112
lwilliam@cs.utah.edu

Abstract- *In the Fall 1998, our sophomore Computer Science students took a new course called "Software Practice." At this early stage in their academic career, the course gave them practical, realistic exposure to the process of creating large software systems in teams. Their previous CS1/CS2 experience focused on the development of C++ language proficiency skills on the Unix platform, with short, programming-in-the-small assignments reinforcing the "skill of the week." Much to the students' periodic frustration and anguish, the Software Practice class differed in most every way – having one, programming-in-the-large, full semester project on the Windows NT/Visual C++ platform that often forced them to independently research new programming techniques. Course evaluations indicate that the students found the course, with its stark view into the realism of software development, enlightening, highly valuable and fun.*

History and Introduction

In the Fall Semester of 1998, we presented to sophomore students a new class "Software Practice." The class was intentionally developed and scheduled early in the sophomore year in order to give students earlier exposure to Software Engineering and insight into the realities of software development. The textbook for the class was Code Complete [1]. The text, lectures, and assignments emphasized using good coding practices and style (see www.cs.utah.edu/classes/cs3500 for all of the course materials).

The class grew out of a curriculum change that we started in the fall of 1995 (see www.cs.utah.edu/~cs451/ for the 1997 version of the fall quarter of the class). At that time, we added software engineering concepts such as object-oriented analysis and design, testing, and teambased development to the year-long senior software laboratory class. Previously, there had been very limited exposure to software engineering in the curriculum. In addition, we used a newly constructed lab of PCs running Windows NT instead of our traditional UNIX lab. The domain for the class project was the control of physical devices constructed out of Lego and fischertechnik kits. The students built line following vehicles, a scanner, and eventually a very large, simulated manufacturing system. This diagram shows the Lego scanner and the results of the scan (after 1.5 hours– it was very SLOW).



The class met many of the goals that we had outlined, but seemed to have very polarized reactions from the students. One set of students felt that the class was exactly the kind of material that they needed to learn (in fact, several were able to confidently answer job interview questions based on what they learned in the class). Their main criticism was that they felt that they wished that they had been exposed to the material much earlier, so they could work on applying the skills during their academic careers

0-7803-5643-8/99/$10.00 © 1999 IEEE

instead of being thrown immediately into the working environment with only minimal experience. The other set of students (a much smaller set) disliked the class all together. Their main reason was that they felt that they already knew how to program and thus learning structured methods for requirements analysis, design, and testing was a complete waste of time. They already had developed skills (albeit unstructured) and processes to perform the tasks.

The conclusion drawn from these two observations was that the class should be earlier in their academic career. We believed that it was important to provide them these skills earlier, before they became locked into their own bad habits. Finally, the University of Utah was forced to convert to semesters. We took this as an opportunity to make a substantial curriculum change and created "Software Practice" for sophomores. Previous to this change, the students got very limited exposure to software engineering concepts and experiences until their senior year.

Two main challenges surfaced in the design and implementation of the new class. The first was the skill level of the students, and the second was how one teaches programming style. The first was a problem because our goal was to create a real enough set of exercises to feel that the students really were experiencing the "real world" of software development. This, however, led us to discover unexpected holes in their knowledge. For example, the concepts of virtual functions, dynamic link environments, and even parsers were all foreign to the students. Our solution was to have on-demand mini-tutorials on the subject and then move on to the task at hand.

The second problem, involving teaching programming style, was solved by using two techniques and the excellent book, *Code Complete* [1]. Each week, the students read a set of chapters talking about style issues. In a following class, we discussed in a question/answer format the positives and negatives of each style issue. Then we would take some source code, analyze it and try to improve it to meet the style guidelines that we had just encountered along with those issues that we had covered earlier. Additionally, the ParaSoft CodeWizard tool was used to identify style problems in code. Use of this tool is discussed below.

"Will he ever get over this hangup with Lisp?"

As one would expect in this kind of class, there were a loof programming assignments. We ended up with nine assignments over the course of the semester. Beginning with the second assignment, each built on previous assignments, incrementally developing a very functional Lisp interpreter in C++. The building of the Lisp interpreter via these assignments is summarized in Appendix A.

The semester-long, programming-in-the-large project was one of the first big difference from past classes, where the students had been used to weeklong, programming-in-the-small assignments in CS1 and CS2. They had developed a mindset that if they were too busy to do a quality job on one assignment – no bother – they got a fresh start the next week. Not so with the Lisp assignments – they just kept coming back. When new assignments were discussed for the first time in class, sighs of "not again" could be seen throughout the room. Students complained to teaching assistants, "When will he ever get over this hangup with Lisp? I'm sick of Lisp." But, doubtlessly the students learned a valuable lesson on incremental development and on "suffering" with their own bad code and lack of understanding. Additionally, the students, who had no previous exposure to any functional language, fully understood and appreciated functional programming by the end of the semester.

Assignment #1: "Good-bye Unix, Hello Windows NT"

The first assignment was designed as an immigration away from their past experience with Unix/g++ to using the Windows NT/Visual Studio lab. The goal was to develop a web page that described them. The page had a set of requirements that included a means of physically identifying each student (usually this was accomplished with a picture) and times when they were busy or available. These web pages were used by the teaching staff for name/face recognition and by other students when they were picking partners for group projects. (The University of Utah is generally a commuter school with a student population where most students have a family and work. Therefore, the best mechanism for identifying partners was to find someone who had a similar schedule.) They had to be developed using a WYSIWYG editor on the PC. Some students had personal computers at home with Windows; others had limited experience with personal computers and Windows. Those with limited experience quickly became frustrated and could be found in the lab with desperate "spoonfeed me" expressions on their face. Some resorted to emacs/HTML hacking to get the job done. Eventually, all became quite proficient in their new development environment.

Assignment #2: "Where's the Source Code?"

The building of the LISP interpreter began in earnest with the second assignment. However, the students didn't get to participate in the "building," only the testing. A mythical programmer, J. Hacker, had all the fun coding. The students were given a specification via the homework assignment, a header file, and an executable. No source code! The students did not feel they had control of the assignment without seeing the source code. In industry, objective,

independent test groups that never see source code often do black box testing. So, this experience prepared the students for the future.

Shortly after the students received their assignment and code, they needed to produce a test plan to validate that the code actually performed according to the specification. Their test plan outlined a test case and the expected results when the test case was run. At assignment completion, they needed to hand in a revised test plan with all test cases, the expected results, and the actual results. They also needed to hand in a document enumerating all defects they detected in the code and fully-documented, nicely-structured test code.

An important learning experience of thisassignment was due to the intentionally poorlywritten specifications. The students asked questions about what the specs meant, what happens in error situations, what happens when a certain set of arguments is given, etc. By the time we evaluated the assignment, they had learned the benefit of well-written specifications. Having been treated to poor specifications, they realized that they would not want to force bad specifications on any other programmer.

Assignment #3: "How Do We Know When We Are Done?"

J. Hacker did it again – added more buggy garbage collection code to the Lisp interpreter! Much to their relief, the students were actually able to see the source code this time. The students had a specification, purposely not thorough, via the homework assignment. Their job was to identify defects in the code by running their test cases from Assignment #2 as regression test cases, coding and running new test cases to validate the added function, and using the Visual C++ debugger. Again, the students were irritated by the "open-endedness" of this assignment. "How do we know when we are done?" "How many defects are in there, anyway?" Some very interesting questions . . . that we never *answered.*

In their previous experience with the g++ compiler, gdb was the only debugger available. Most students found gdb difficult and resorted to debugging by placing print statements in their code. This assignment, however, could not be done without using and learning the Visual C++ debugger. The students had todocument all the defects they found, complete with the breakpoint, the call stack at the breakpoint, the values of the arguments and variables at the breakpoint, and the necessary changes to repair the error. They also had to hand in their complete set oftest cases and revised code with all defects fixed. This assignment broke the students of their "debugging with print statements" habit, setting the stage for more efficient debugging on future assignments. It also taught them that in the real world, one does not always debug your own code.

Assignment #4: "Debugging in Style"

J. Hacker's code style was very poor. There were not enough comments, the style was inconsistent and would certainly not conform to any coding standards. Coding with good style and adhering to standards was emphasized in lecture material. Now, the students needed to fix J. Hacker's style and documentation problems, while ensuring all regression test cases continue to work.

The students needed to add appropriate comments and function and program headers. The more challenging aspect of the assignment was getting the code to conform to the coding standards enforced by ParaSoft Corporation's CodeWizard tool. The CodeWizard for C++ tool performs static, source code analysis to enforce C++ coding standards to produce cleaner, more robust code. The standards enforced by the tool are based the "items for effective C++ programming" described in the popular books *Effective C++* [2] and *More Effective C++* [3], Meyers-Klaus items and Universal Coding Standard items. From this assignment forward, any code the students wrote had to conform to the CodeWizard standards. We found the ParaSoft tools very useful; the students were thrilled because ParaSoft gave all 160 of them copies for their home use. Although the student learning in this assignment was exactly what we planned, a big problem was grading the assignments. J. Hacker's code contained so many coding style problems that the teaching assistants were forced to hand check each and every line of each program. This was VERY time consuming.

Assignment #5: "It's Too Slow," complains the customer

In the fifth part of the assignment, J. Hacker implemented a symbol table for the Lisp interpreter. But alas, "the customer" complained that it takes way too long to create new symbols and to find existing symbols. "How much faster do we need to make it?" inquired the students. "Oh, I don't know. It needs to be a lot faster," replied the customer. Another open-ended assignment! The students were challenged to be among the top 15% fastest performers to obtain an extra 15 point bonus.

The students were taught to use the Visual C++ profiler to identify specifically which parts of the program were time hogs. It turned out that J. Hacker implemented the symbol table as a static array that used a linear search. The students, who had not taken a Data Structures class yet, had to research solutions for making it faster. Most did determine that a hash table would be best. They needed to improve the data structures and implementation of several functions. Then, they had to turn in a faster working version that ran all previous regression tests and new tests related to the symbol table functions, and that adhered to coding standards enforced by CodeWizard. Everyone seemed to enjoy this

assignment because they felt that it was an interesting challenge to first isolate the offending part of the program and then try to make that part perform faster.

Assignment #6: "Finally, Our Own Code"

Finally, a full two months into the class, the students finally got to design and implement their own code. They developed the specified functionality for printing Lisp Items. There already was, of course, an established public interface that they were required to match. They also needed to ensure all previous regression test cases ran, develop new test cases related to printing and again adhere to coding standards.

Assignment #7: "Let's Integrate at 11:30 Tonight"

Assignment 7 was, for most, the first time the students ever worked with a partner to produce a software product. As a pair, the students had to write the "read" and "evaluate" modules to complete the read/eval/print loop. We did not specify exactly how to assign the tasks between the students, but generally, one student of the pair designed and coded the read module; the other student designed and coded the evaluate module. The students also had to test their work, and were encouraged to swap modules for testing prior to integration. Working with a partner and integrating code turned out to be much harder than they anticipated. More students were late than on any other assignment. One group begged for mercy the morning after it was due, "We figured as long as we got together to integrate at 11:30 last night, we'd be able to turn it in by midnight." The lessons on the difficulty of integrating proved valuable for the end of the semester project, assignment 9.

Assignment #8: "A Rose By Any Other Name . . ."

In assignment 8, the students were exposed to the Rational Rose tool. First, they used to tool to re-engineer the design of their interpreter. They then added to this design by using the tool to define a new class with attributes and methods. Rose then was used to generate skeleton code for these methods. Although this was somewhat of a diversion from the task of creating the Lisp system, it proved to be a good exposure to these types of tools. The students were very impressed.

Assignment #9: "The Finale"

For the final project of the semester, the students worked in groups of four to six students to extend their Lisp system. They were given the choice of building a GUI or to building the capability to dynamically define functions. However, by the end of the project, they needed to integrate their code with that of another group. Therefore, their working product they turned in for a grade had both a GUI and the ability to dynamically define functions.

The group structure and procedures were defined early. Each team had a password-protected web page, which was used to communicate with the teaching staff and among team members. Members of the team chose roles, such as meeting facilitator, team scribe, team leader, lead requirements gatherer, lead designer, lead tester, integration specialist, etc. Each of these roles had assigned duties that had to be done in addition to any coding they were assigned. Requirements and design were submitted midway through the assignment. After that time, all changes had to be approved. Official code inspections by the group were held and documented.

After the long, hard semester, the students were tremendously proud to demonstrate their working Lisp interpreters. Many who had struggled to understand functional programming and Lisp had an "Ah-ha" experience at this point. Several groups went way beyond the assignment requirements, obtained Lisp manuals, and, essentially, built a *fully*-functional Lisp interpreter with a GUI.

Summary: "It was a great class to have taken, but not to be taking."

Course evaluation metrics placed the class among the top 15% of undergraduate classes offered by the college of engineering, quite a feat for the first offering of a large undergraduate class. (The highest rated classes are traditionally the smaller classes; this class had 160 students.) At the end of the semester, the students resoundingly agreed "It was a great class to have taken, but not to be taking." They had successfully made the transition from programmers to software engineers by expanding their views on what it really takes to produce a large software product. They had mastered a new development environment. They had learned a functional programming language. Even the student who really did say, "If this is what it's really like, maybe I better major in English" decided to stick with Computer Science.

Bibliography

1. McConnell, S., *Code Complete*. 1993, Redmond, Washington: Microsoft Press.
2. Meyers, S., *Effective C++*. Professional Computing Series, ed. B.W. Kerninghan. 1998, Reading, Massachusetts: Addison-Wesley.

3. Meyers, S., *More Effective C++*. Professional Computing Series. 1996: Addison-Wesley.

Appendix A: The Building of the Lisp Interpreter

Assignment #	Addition to Lisp Interpreter (additional learning experience)
9	**GUI, dynamically defined functions** (Team-Based Development)
8	(Rational Rose Tool)
7	**Read and Eval modules** (Integration)
6	**Printing**
5	**Symbol Table** (Profiling, Performance, Data Structures/Hash Table)
4	(Adherence to Coding Standards, CodeWizard Tool)
3	**Garbage Collection** (Debugging via Visual C++ Debugger)
2	**Storage Manager** (Black Box Testing, Perils of Poorly-Written Specifications)