Simulating Critical Software Engineering

David Evans Michael Peck
University of Virginia, Department of Computer Science
Charlottesville, VA
[evans, mike]@virginia.edu

ABSTRACT
One goal of many introductory software engineering courses is to simulate realistic software engineering. Unfortunately, many of the practical constraints of typical courses are antithetical to that goal: instead of working in large teams on large projects, dealing with changing requirements and maintaining programs over many years, courses generally involve students working alone or in small teams with short projects than end the first time the program works correctly on some selected input. Of course, it is impossible (and undesirable) to carry out full industrial software development within the context of a typical university course. However, it is possible to simulate some aspects of safety critical software engineering in an introductory software engineering course. This paper describes an approach to teaching introductory software engineering that focuses on using lightweight analysis tools to aid in producing secure, robust and maintainable programs. We describe how assignments were designed that integrate analysis tools with typical software engineering material and reports on results from an experiment measuring students understanding of program invariants.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: *Software/Program Verification: assertion checkers, class invariants, formal methods, programming by contract, reliability*, F.3.1 [**Logics and Meanings of Programs**]: *Specifying and Verifying and Reasoning about Programs - assertions, invariants, logics of programs, mechanical verification, pre- and post-conditions, specification techniques*, K.3.2 [**Computers and Education**]: *Computer and Information Science*.

General Terms
Reliability, Security, Verification.

Keywords
Software engineering, dependability, security, invariants, static analysis.

1. INTRODUCTION
Teaching software engineering is extremely difficult. The root of the problem lies in the impact of scale. Most of the principles central to software engineering are crucial for producing large, robust, long-lived programs, but hardly relevant (and oftentimes counterproductive) for the smaller, non-critical, short-lived programs that can be developed in the scope of an academic course. This means the methods and theories taught in software engineering courses are often regarded by students as abstract, academic concepts which are quickly forgotten after the final exam. Without experiencing their practical impact on realistic programs, students rarely develop a deep understanding or appreciation of important ideas in software engineering.

A common approach to this problem is to increase the size of programs students deal with in classes, often by having students work in groups. This helps, but it is still impractical to do industrial size projects within the scope of a semester course. Students with limited time and other classes cannot be expected to construct an industrial scale program for a course. Even if they could, one experience is often insufficient. Because the project ends when the semester is over, students don't have to try to figure out their code six months later to make changes to it.

It is becoming increasingly important that all students who go on to produce software will know what is involved in producing secure and reliable software. Nearly all software today is critical to some degree. Any program running on a machine connected to the Internet can be exploited by an attacker, even if the function of that program is not itself critical.

This paper reports on our experiences teaching an experimental software engineering course at the University of Virginia. The course is the second computing course for most students, and satisfies a requirement for students in non-computing majors. We taught a small experimental version of the course in Fall 2002 with 27 students, and are currently teaching the course with 68 students. The course uses Liskov's *Program Development in Java* [11] as the textbook and is heavily influenced by MIT's software engineering course. Our primary contribution is to develop an approach that uses lightweight analysis tools to support and enhance the value of the systematic and analytical approach to programming presented in that text and course.

2. UNDERSTANDING CHALLENGES
The first step to producing reliable software is to understand how difficult it is. Many students leave their first programming course believing that their program is correct once it works as expected on one input. After all, in many courses that is enough to get full credit for the assignment. Students are usually not expected to deal with uncooperative users who generate atypical (or even

malicious) inputs. After all, no one other than the student will ever run the program.

In real software engineering projects, of course, this is not the case. We used several strategies to simulate quality requirements of deployed software. The first was evaluating students programs based on subjective criteria such as how easy it would be for someone to maintain their code, as well as objective results from running their program on both public and secret test cases. For some of the early assignments, students would submit their programs to a web form that would automatically execute their code on a set of test cases. Some of the test cases were distributed to the students as part of the assignment, but others were kept secret. When students submit their programs to the web site, they would receive feedback on whether or not their program passes each test case, but would not receive any information about what the test cases their program failed were. This encouraged students to think carefully about their own testing strategy. They were unlikely to produce a correct program without thinking systematically about all possible inputs to the program.

On a simple early assignment, students were asked to place a bet on whether or not their program was correct. In this case, correctness meant it satisfied the precise informal specification they provided, and that it behaved as a mythical customer expected for any inputs not covered clearly by their specification. Students could bet up to 20 points (on an assignment worth 100 points) on their program; if their program was considered correct, they received their bet; if not, they lost twice their bet. Eight (of 25) students bet 0, although two of those had correct code. Fourteen students bet something between 2 and 10, ten of whom had code that was considered correct. Three students bet the maximum 20 points, only one of whom had correct code. Although this was a somewhat gimmicky way of getting students to measure confidence in their code, it does teach several important lessons. Having correct code, but no confidence that it is correct, is nearly useless: no one will use a program without a reason to be confident it will work. Having overconfidence in incorrect code is dangerous: if someone relies on it for a critical purpose the impact of programming errors can be disastrous.

3. ANALYSIS TOOLS

Formal methods have often been proposed as a means to producing more reliable and maintainable programs. Edsger Dijkstra [1] and David Gries [9] suggest approaches to teaching programming that closely integrate proof techniques. Despite the elegance and apparent benefits of these approaches, they have met with limited success. Students generally only use proof techniques when they are required to do so on small programs in course assignments. In most students' experience, proving correctness properties about programs is a tedious and academic process, not something they do voluntarily with an expectation that it will improve their programs. Despite the value in doing these exercises, it is very clear that these manual techniques would not scale well to a program of non-trivial size. Students rarely leave these courses thinking about adding explicit representation invariants or formal postconditions to their programs.

The problem with most previous attempts to introduce formal methods into computer science curricula is the gap between formal methods and the students' practical programming experience is too large. Jeannette Wing identified tool support as a critical factor in integrating formal methods into undergraduate computer science curricula [15]. To enable widespread adoption of formal methods, we need both appropriate analysis tools and documentation and course materials that relate them to the abstract concepts and methods we aim to teach. Lightweight analysis tools are a promising way to shrink this gap and make the abstract concepts we hope to teach immediately and directly relevant in a practical way. After several years of research efforts by many groups in academia and industry, lightweight analysis tools are now mature enough to be used effectively in undergraduate education. Whereas it requires considerable effort to use traditional formal methods directed towards program verification on even toy programs, students will be able to quickly and effectively use lightweight analysis tools on real programs.

In our course we used two lightweight analysis tools: ESC/Java and Splint. The Extended Static Checker for Java (ESC/Java) is an analysis tool for Java developed at the Compaq Systems Research Center [10]. ESC/Java produces warnings for expressions that may produce run-time errors such as an invalid down cast, null dereference, or out-of-bounds array fetch. ESC/Java also provides an annotation language for specifying preconditions and postconditions of methods and datatype invariants. ESC/Java generates verification conditions based on an analysis of the source code and its annotations. A theorem prover searches for counterexamples to the verification conditions, and translates contexts that produce counterexamples into warnings. For six of the seven programming assignments in the course, students programmed in Java and used ESC/Java on all but the first assignment.

For one of the assignments (the sixth of seven), students used C and the Splint lightweight analysis tool. Splint [5, 6, 7] is an annotation-assisted lightweight static checking tool for C. Splint's checking is done using simple dataflow analyses so it does not have the deductive reasoning power of ESC/Java, but can scale to work on large programs. Students used Splint to detect and fix information hiding violations, possible null dereferences, and memory leaks in a provided C program. Since many students are likely to end up programming in C or C++, we believed it was important to teach techniques for producing dependable and maintainable programs even when using unsafe languages.

4. INTRODUCING ESC/JAVA

To introduce ESC/Java, we provided students with a tutorial-style explanation of the warnings ESC/Java produces for the simple buggy program shown in Figure 1. Running ESC/Java on AverageLength.java produces two warnings. The first warns about an array index possibly too large for args[0]. This corresponds to the run-time error that would occur if the program is run with no arguments. The second warning concerns an unsatisfied precondition:

```
Warning: Precondition possibly not established (Pre)
        String name = names.getEntry (index);
Associated declaration is "./StringTable.java", line 131, col 10:
    //@requires index < numEntries;
```

Path information (not shown) reveals that the precondition for the StringTable getEntry method is not satisfied on the final interation of the loop. This precondition is specified using an annotation in the StringTable specification that requires the value of the

parameter passed to getEntry is less than the number of entries in the table.

This example illustrates how static analysis can assist program development. In the first case, the analysis detects a problem that might not be detected through testing. In the second case, the analysis provides clearer understanding of a problem that would likely also be revealed through testing. For the second assignment in the course, students are asked to explain and fix problems in the AverageLength program revealed by ESC/Java warnings, and to implement a simple program using the StringTable data abstraction. They check their programs using ESC/Java, and fix any problems it reports.

```
public class AverageLength {
  public static void main (/*@non_null@*/ String args[]) {
    String filename = args[0];
    try {
      FileInputStream infile = new FileInputStream (filename);
      StringTable names = new StringTable (infile);
      int numnames = names.size ();
      int totallength = 0;

      for (int index = 0; index <= numnames; index++) {
        String name = names.getNthLowest (index);
        totallength = totallength + name.length ();
      }

      System.out.println ("The average name length is: "
                          + (double) totallength / numnames);
    } catch (FileNotFoundException e) {
      System.err.println ("Cannot find file: " + filename);
    }
  }
}
```

Figure 1. Buggy AverageLength implementation.

5. INVARIANTS

One of the goals of our course was to encourage students to think precisely about properties of their programs. By considering and precisely documenting procedural interfaces, students will produce more reliable and maintainable software. Documenting representation invariants will encourage students to think more carefully about their datatype implementations and make those implementations more modular and maintainable.

Without using analysis tools, it is difficult to teach students to develop good invariants and to convince them that it is a worthwhile activity other than to satisfy a particular course requirement. The problem is the effort required to precisely document an invariant is usually considerably more than the apparent benefit. Since the assignments students are working on are small, both in size and time, it is unlikely they will need to understand someone else's code or their own code after they have forgotten it. Further, if invariants are used only to aid informal reasoning, their value is apparent only to the extent that students find it useful to reason informally about the correctness of their code.

Using analysis tools can dramatically increase the benefits of documenting program invariants. The difficulty in providing invariants increases somewhat since they need to be expressed using a formal annotation language. However, once invariants are expressed formally the benefits of providing them increase dramatically. Analysis tools can check that the specified invariants are maintained, and use those invariants to check implementations. For teaching purposes, it is especially useful that warnings produced often lead students to realize than an invariant they are depending on for correctness is not documented.

For example, consider implementing a StringSet datatype in Java using a java.util.Vector representation (based on the IntSet example in [11]). One invariant is that all the elements of the representation Vector are non-null Strings. Suppose we implemented a naïve choose method using:

```
public String choose () { return (String) els.firstElement(); }
```

ESC/Java reports two warnings for the choose implementation. The first indicates a possible run-time error for the type cast. To know the String cast will not produce a run-time exception, we need to know that all elements of the els Vector are of type String. This invariant of our representation can be documented using an annotation: //@invariant els.elementType == \type(String). This invariant clearly documents the requirement that all elements of the representation vector must be strings. ESC/Java will check that any elements added to the vector are of type string.

The other warning reports that the precondition for firstElement is not satisfied. It requires that the vector contains at least one element. One way to address this would be to add a precondition to the choose method that requires the set contain at least one element: //@requires numElements > 0. The numElements variable is a specification variable that maintains the number of elements in the set. We relate this to the representation by adding an invariant: //@invariant numElements == els.elementCount.

Note that we have fixed the problem without changing the code. By adding the precondition, we have documented a restriction on calling contexts. Since the precondition is specified formally, ESC/Java is able to check that it is satisfied by all call sites.

For the third assignment, students implement a data abstraction according to a specification we provide. In the current version of the course, the data abstraction is a table that associates a count with a string and provides methods for getting strings based on the rank of their associated count in the table. This datatype is used by an application that uses the Google API [8] to find words most commonly associated with a particular search term. Students are expected to document their representation invariants both informally and as ESC/Java annotations in their code. The checking provided by ESC/Java nearly always leads students to identify missing invariants upon which the correctness of their code relies.

6. EXPERIMENT

We conducted an experiment at the end of the course to better assess student difficulties with annotating invariants. The goals of the experiment were to determine how well students understood annotating programs, and whether the problems were primarily syntactic or conceptual. Our experiment was based on an experiment conducted by Jeremy Nimmer and Michael Ernst to evaluate the effectiveness of annotation inference [13].

Students downloaded annotated implementations of two classes taken from a data structures textbook [14]: StackAr (119 total lines, including comments and annotations), a stack of objects data abstraction implemented using an array; and, BitSet (113 total lines), a datatype that represents a set of integers with values

between 0 and *capacity*-1 using an array of booleans. Students were also provided with sample client programs for each datatype. The programs were annotated with some correct and some incorrect annotations. We instructed students to run ESC/Java on each datatype implementation and client, and eliminate the reported warnings by only changing the annotations.

Students were asked to participate in the experiment after the end of the course and were not graded on their participation or performance in the experiment. Seventeen students (out of 27 in the course) attempted the experiment. Two did not follow the directions closely enough to be included in our analysis (they changed the code instead of the annotations), leaving 15 valid subjects. There was some correlation between students overall performance in the course and their willingness to do the experiment. The average final course rank of students who did the experiment was 12.3, for those who did not, 16.1. Seven of the top ten students in the course did the experiment, compared to five of the bottom ten students. This is too small a sample to make strong quantitative results, but sufficient to consider qualitatively.

Based on their username, students would receive a different version of the zip file containing the experiment code. Half the students recieved StackAr first, and the other half BitSet. There were also two variations on each implementation with different provided annotations. Students randomly received one version of each data type implementation. The annotations were generated using Daikon, a tool that determines likely invariants automatically by analyzing program executions on test data [3, 4]. Daikon examines values in test executions and infers useful invariants based on patterns and relationships detected in all executions. Invariants reported by Daikon are true for all executions in the test data, but not necessarily true of all possible program executions. Daikon has been used in conjunction with ESC/Java to automatically add ESC/Java annotations to Java programs [12]. Although students in our experiment did not run Daikon themselves, we believe it is realistic to believe they could do so in future courses. This would have the added benefit of encouraging and evaluating test suites; the annotations produced by Daikon are directly related to the quality of the test suite. Daikon will produce invariants that are unsound (they hold for all executions in the test suite, but not for all possible executions of the program), and will fail to produce necessary invariants. The test programs we used contained both incorrect invariants and missing invariants that are necessary for ESC/Java checking.

Students were told to spend no more than 30 minutes on each datatype. For the experiment, students used a modified ESC/Java script that would record the time, inputs and results each time they executed ESC/Java. They submitted their code after completing the task, either because they succeeded in eliminating all warnings or they ran out of time.

Table 1 summarizes student performance on the experiment. Over the entire experiment, only one student mistakenly removed correct provided annotations. Removing incorrect annotations was understandably more difficult. Most students were able to remove incorrect annotations that corresponded directly to warnings produced when ESC/Java is executed on the code and test client. For example, version 1 of the BitSet datatype contained a single incorrect annotation that imposed a false precondition on the insert method: requires bits[el] == false. Daikon derives this invariant, since the inadequate test suite used to produce this version never added an element to the set that was already a member of the set. Running ESC/Java on the provided test client produces a warning that the precondition is not satisfied for a call to insert. Four out of seven students were able to correctly interpret this warning and remove this precondition. On the other hand, when the incorrect invariants are revealed only indirectly by the warnings ESC/Java produces (such as in StackAr version 1), only a few students were able to remove the corresponding incorrect invariants.

Adding annotations requires students to realize what invariant should be documented, and also express it using ESC/Java's annotation language. BitSet version 2 and StackAr version 1 require complex annotations involving a forall quantifier and a logical implication. Since the students had not encountered such complex annotations in any of the course assignments, we were not surprised that no students were able to correctly produce these annotations. About half the students were able to correctly add necessary simpler annotations. For example, the constructor took an int parameter and needed a requires capacity >= 0 annotation. Eleven of fifteen subjects were able to correctly add the necessary annotation.

		Correct annotations preserved	Incorrect annotations removed	Necessary annotations added
BitSet Version 1	Ideal Solution	16.00	1.00	9.00
	Best Student Solution	16.00	1.00	9.00
	Average Student	16.00	0.57	4.29
	Standard Deviation	*0.00*	*0.53*	*3.59*
BitSet Version 2	Ideal Solution	21.00	4.00	2.00
	Best Student Solution	21.00	2.00	2.00
	Average Student	21.00	0.88	1.13
	Standard Deviation	*0.00*	*0.99*	*0.99*
StackAr Version 1	Ideal Solution	20.00	13.00	2.00
	Best Student Solution	20.00	11.00	1.00
	Average Student	19.71	3.71	1.29
	Standard Deviation	*0.76*	*4.99*	*0.49*
StackAr Version 2	Ideal Solution	18.00	2.00	4.00
	Best Student Solution	18.00	2.00	1.00
	Average Student	18.00	1.25	0.75
	Standard Deviation	*0.00*	*0.89*	*0.46*

Table 1. Summary of Experimental Results. For each task variation, we show the number of provided annotations that were preserved correctly, the number of provided incorrect annotations that were removed, and the number of necessary annotations added.

7. CHALLENGES

There are a number of impediments to successfully introducing analysis tools into introductory software engineering courses.

Since the amount of material typically covered in these courses is already excessive, it is necessary to exclude some other material to make room. We believe this tradeoff is justified. A primary goal of introductory software engineering courses should be to teach students to produce reliable and trustworthy programs and to be able to reason informally about their correctness.

We found it very difficult to prevent students from focusing only on getting the code to appear to work, even when it is only presented as one question out of ten on a problem set. Students are naturally driven by creating programs, which is generally a good thing. Most students were not sufficiently convinced of the benefits of using analysis tools, however, to use them as part of that process. Instead, they would struggle to get the code working, and then run ESC/Java on the code and start thinking about invariants and adding annotations to eliminate the warnings. Further, students who were not able to get the code working often did not even attempt to run ESC/Java on their code. This is contrary to our goals, since we want to encourage students to consider invariants while developing their programs.

For this year's course, we have solved those problems by changing the way students use ESC/Java. Instead of running the Java compiler and ESC/Java from the command line, we have developed a plug-in that runs ESC/Java within Eclipse, an open source integrated development environment [2]. The plug-in runs ESC/Java as part of the compilation process, and integrates ESC/Java warnings with compiler warnings. It has the further benefit, that students can jump directly to the location of a warning by clicking on the message.

A more serious challenge concerns the limitations of lightweight analysis tools. Both ESC/Java and Splint are unsound and incomplete. This means they produce both false positives (generate spurious warnings) and false negatives (miss legitimate problems). Experienced developers and tool users can often identify false positives quickly and understand the limitations of the analysis that lead to them. Introductory programmers are justifiably reluctant to discount a warning as a false positive. As a result, they can waste many hours trying to solve a non-problem. After they are told that the warning is incorrect, this experience can lead them to incorrectly assume other warnings are also false positives. There is no complete solution to this problem, although as the quality of available tools improves it will be mitigated.

8. CONCLUSION

Software engineering courses should strive to instill in students the attitude necessary to produce dependable and maintainable programs, and intellectual and pragmatic tools that assist that goal. It is becoming increasingly important that all students who go on to develop software have an appreciation and understanding of what it takes to produce reliable programs.

Our goal of simulating large-scale safety critical software development in introductory software engineering courses is aided by the use of lightweight analysis tools. Programmers who develop the habit of thinking precisely about invariants when they design and develop their code will produce better programs, and analysis tools are effective in encouraging students to do that. Although there are many challenges associated with using these tools in introductory courses, they can provide substantial benefits.

9. ACKNOWLEDGMENTS

The authors thank Mike Ernst and Jeremy Nimmer for providing code and assistance for the experiment, the National Science Foundation for supporting this work through NSF CCLI 0127301, Sol Chea, Serge Egelman, Tiffany Nichols, and the CS201J students.

10. REFERENCES

[1] Edsger Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[2] Object Technology International. *Eclipse Platform Technical Overview*. Feb 2003. http://www.eclipse.org/whitepapers/eclipse-overview.pdf

[3] Michael Ernst. *Dynamically Discovering Likely Program Invariants.* PhD Thesis, U. Washington. August 2000.

[4] Michael Ernst, J. Cockrell, William Griswold and David Notkin. *Dynamically discovering likely program invariants to support program evolution.* IEEE Transactions on Software Engineering, Feb 2001.

[5] David Evans, John Guttag, Jim Horning and Yang Meng Tan. *LCLint: A Tool for Using Specifications to Check Code*. SIGSOFT Foundations of Software Engineering. Dec 1994.

[6] David Evans. *Static Detection of Dynamic Memory Errors*. SIGPLAN Conference on Programming Language Design and Implementation. May 1996.

[7] David Evans and David Larochelle. *Improving Security Using Extensible Lightweight Static Analysis.* IEEE Software, Jan/Feb 2002.

[8] Google Web APIs. http://www.google.com/apis/

[9] David Gries. *The Science of Programming*. Springer-Verlag, 1981.

[10] K. Rustan M. Leino. *Extended Static Checking: a Ten-Year Perspective.* Schloss Dagstuhl Tenth-Anniversary Conference. Springer LNCS volune 2000, 2001.

[11] Barbara Liskov with John Guttag. *Program Development in Java: Abstract, Specification, and Object-Oriented Design.* Addison Wesley, 2001.

[12] Jeremy W. Nimmer and Michael D. Ernst. *Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java*. First Workshop on Runtime Verification, July, 2001.

[13] Jeremy W. Nimmer and Michael D. Ernst. *Invariant inference for static checking: An empirical evaluation*. SIGSOFT Foundations of Software Engineering, Nov 2002.

[14] Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java.* Addison Wesley Longman, 1999.

[15] Jeanette Wing. *Weaving Formal Methods into the Undergraduate Computer Science Curriculum*. 8th International Conference on Algebraic Methodology and Software Technology, May 2000.