Teaching Software Engineering Using Lightweight Analysis
David Evans
June 2001

Summary

In teaching software engineering, it is a major challenge to integrate methodology and theory into the practice of software development. The goal of this project is to explore a new pedagogical approach to teaching software engineering centered on the close integration of analysis tools with instruction in programming methodologies. Instead of learning methodologies as abstract ideas, students will directly benefit from applying analysis tools that embody methodologies to large, realistic programs.

We propose a new approach to teaching software engineering that exploits lightweight analysis tools. Our approach will allow for realistic experience with industrial scale programs, and enable direct application of theory and methodology to practical programming. We propose to develop a pilot course that will use lightweight analysis tools to teach software engineering. Materials and ideas from that course will then be integrated into our core software engineering courses and made available and adapted for use at other schools.

We will focus on the use of lightweight analysis tools that offer clear and immediate benefits with minimal initial costs. These tools include LCLint [EGHT94, Evans96, LE01], a lightweight static analysis tool that exploits annotations added to the program source code; the Extended Static Checker for Java (ESC/Java) [Detlefs98, Leino01a], a static analysis tool that incorporates an automatic theorem prover; and Daikon [Ernst00, ECGN01], a tool for automatically determining likely program invariants. Examples of topics where these tools can be used for pedagogical benefit include information hiding, invariants, memory management and security.

Project Description

1 Goals and Objectives

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) -critical, short-lived programs that can be developed in the scope of an academic course for the smaller, non. Hence, students often regard the methods and theories taught in software engineering courses as abstract, academic concepts. 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. However, 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 insufficient. Because the project ends when the semester is over, students do not have the experience of needing to modify their code six months later.

Edsger Dijkstra [Dijkstra76] and David Gries [Gries81] suggest approaches to teaching programming that closely integrate proof techniques. Despite the elegance and apparent benefits of this approach, it has 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. Many curricula now offer courses in formal methods, but these are typically taught as advanced undergraduate or graduate courses separate from the core curriculum and not taken early enough to be used throughout a curriculum.

The problem with previous attempts to introduce formal methods into computer science curricula is the large gap between formal methods and the students' practical programming experience. Lightweight analysis tools are a promising way to shrink this gap and make abstract concepts 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 on even toy programs, students can quickly and effectively use lightweight analysis tools on real programs.

By using and understanding analysis techniques and tools, students will encounter concrete embodiments of previously abstract ideas. By using analysis tools to automatically or semi-automatically identify relevant program fragments, students will be able to work with and modify industrial size programs. Students will learn how to use lightweight analysis and associated methodologies to quickly isolate relevant parts of large programs so they can modify and reason about industrial scale programs without needing to understand the entire program.

This project proposes to develop educational materials for using analysis tools in both specialized courses, and more importantly, for integrating analysis tools into core software engineering courses. We will concentrate on lightweight analysis tools since they are readily accessible and provide clear and early benefits for limited effort. Jeannette Wing identified tool support as a critical factor in integrating formal methods into undergraduate computer science curricula [Wing00]. 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. Educational materials produced will include tutorials and documentation on using analysis tools, a sequence of exercises that involve applying analysis tools to realistic programs to explore and reinforce concepts and principles, and lecture notes and slides using analysis tools to teach software engineering and introducing analysis techniques.

2 Detailed Project Plan

We plan a pilot course that will be taught as an elective for students who have completed at least one programming course. This course would be small enough that there would be an opportunity to closely monitor students' progress and observe how they use the analysis tools.

Based on the pilot course, we will develop experience and course materials for use in mainstream, required software engineering courses (e.g., UVA CS 201, MIT 6.170, and similar courses at other schools). It is important that analysis tools are introduced in the standard curriculum, and then used in other courses, not a stand-alone course that is considered a special topic. We also expect that modules developed for the pilot course would be adapted into other courses. For example, some of the preliminary assignments on information hiding might be appropriate for an introductory (CS 101) course. A more extensive security module would be appropriate for a security elective. Modules on invariants will be appropriate for theory courses. We believe the approach of developing and refining course materials in a specialized, focused course will give us the best opportunity to introduce analysis into a curriculum with minimal disruption.

The remainder of this section describes the analysis tools we intend to use, and provides some examples illustrating how they can be used effectively in teaching.

2.1 Analysis Tools

We concentrate on lightweight analysis tools that provide substantial and immediate benefits relative to the effort required to use them. The range of tools we select provides a sampling of the design space of analysis tools where efficiency, effort required, soundness and completeness are often conflicting goals.

LCLint

LCLint [EGHT94, Evans96, Evans00] is an annotation-assisted lightweight static checking tool for C developed through a joint research project by the University of Virginia, MIT and Compaq SRC. LCLint is designed to be as efficient and easy-to-use as a compiler. If minimal effort is invested adding annotations to programs, LCLint can perform stronger checks than can be done by any compiler or standard lint[1]. LCLint checking ensures that there is a clear and commensurate payoff for any effort spent adding annotations. Some of the problems that can be detected by LCLint include: violations of information hiding; inconsistent modifications of caller-visible state; inconsistent uses of global variables; memory management errors including uses of dead storage and memory leaks; and buffer overflow vulnerabilities. LCLint checking is done using simple dataflow analyses. This means the checking is as fast as a compiler, and LCLint can easily be introduced into standard development cycles.

LCLint has been in active use for more than six years, and is used by thousands of programmers in industry, especially in the open source development community [Orcero00, PG00]. LCLint has been used in several courses including introductory programming courses at RMIT University in Australia, the University of London, and Universidade Estadual Paulista in Brazil. These courses used LCLint to assist students in programming assignments, but did not explore ways to use LCLint to teach software engineering concepts directly.

ESC/Java

The Extended Static Checker for Java (ESC/Java) is an analysis tool for Java developed at the Compaq Systems Research Center [Leino01a]. ESC/Java fits in the design space for static

[1] Lint was a static analysis tool developed in the late seventies as a tool [Johnson78] for warning programmers about inconsistencies in source code. Although standard lints include mechanisms for using source code comments to suppress errors, they do not provide mechanisms to describe programmer assumptions.

checkers somewhere between LCLint and a program verifier. It incorporates an automatic theorem prover, hence it requires more time and effort to use than LCLint, but can check complex and precise properties about programs that are well beyond LCLint's capabilities. 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. ESC/Java can check for conditions that would lead to run-time errors in Java such as null dereferences or out-of-bounds array fetches; synchronization errors (race conditions and deadlocks), and violations of annotations added to programs.

The developers of ESC/Java have expressed strong interest in supporting its use in education, and several universities (including the University of Toronto, Stevens Institute of Technology, and Imperial College) are currently considering developing courses that use it [Leino01b].

Daikon

LCLint and ESC/Java require programmers to invent the invariants and express them as annotations. Daikon determines likely invariants automatically by analyzing program executions on test data [Ernst00, ECGN01]. 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 [NE01]. When a static checker confirms an annotation generated by Daikon, it increases our confidence that the annotation is correct. When a static checker fails to verify an invariant detected by Daikon it often reveals interesting properties about the static checker, test data or program. Daikon was use in a limited way in MIT's Software Engineering course in Spring 2001 [Ernst01].

2.2 Pedagogical Uses of Analysis

This section uses some examples to illustrate how analysis tools can be used to enhance the teaching of software engineering. These examples are not intended to be comprehensive, but should provide some understanding of how analysis tools might be used to teach concepts in software engineering and a flavor of what the course materials and course would be like.

Information Hiding

Information hiding, often in the form of data abstraction, is one of the most important concepts in software engineering (and in design more generally). Students typically encounter forms of information hiding in courses and are told that data abstraction is a "good thing", but language issues (e.g., classes) generally obscure what it really means and why it is useful and important. Many students complete Computer Science degrees without developing a good understanding of this, and it is reflected in the poor quality of the systems they design and the programs they write.

Although C does not provide support for data abstraction, programmers can use an LCLint annotation to declare a type as abstract. By naming convention, certain code is considered part of the implementation of the abstract type and is permitted to access the representation of that type. For all other code, LCLint will report errors for any dependencies on the type representation.

This means the type is checked by name and it may not be used as an operand to primitive operations that depend on its representation. LCLint also detects representation exposure, occasions where storage that is part of the representation of an abstract type may be modified externally. Since languages that claim to support data abstraction well do not prevent representation exposure, the problems and causes of representation exposure are typically not understood well by students in traditional curricula. By using LCLint to provide data abstraction, students can gain a deeper understanding of what it is and its importance in designing maintainable and robust programs. Information hiding is better understood by seeing what must be done to introduce it into a programming language without it, then by learning complex language constructs that integrate data abstraction with modules, type hierarchies, and syntax.

An exercise would require students to modify a large program organized loosely around data types but without definitive abstractions it in a way that requires changing a data representation. By using LCLint, students would identify relevant data abstractions and change client code that depends on its representation. This provides a realistic industrial experience that would apply a methodology in a clear and concrete manner.

Invariants

Invariants are properties that are always true at particular program points. Common types of invariants include preconditions (properties that are true at function entry points), postconditions (properties that are true at function exit points), representation invariants (properties that are true about data representation at access boundaries), and loop invariants (properties that are true at loop heads). Invariants have long been considered a useful technique for creating better programs and for reasoning about and understanding programs [Gries81, LG86]. Program bugs often result from programmers implicitly assuming invariants that are not valid (for example, that a certain function always returns a non-NULL value).

Although students may be required to include explicit invariants in their programs for some assignments in a software engineering class, students almost never document invariants in their programs voluntarily. Students view expressing invariants as a tedious and theoretical endeavor, far removed from the realities of practical programming. The notable exception to this is the use of run-time assertions (as embodied in C by the `assert` macro). Good programmers quickly learn that these can be a great aid in reducing their debugging time and producing reliable and maintainable programs, and use them liberally. We believe that the reason for this is that the run-time assertions provide a clear and obvious benefit that more than outweighs the effort and cost required to add them. The benefits of unchecked invariants are much less clear, and it is hard for students to see the value in adding these to their programs.

By using analysis tools, students can obtain clear and immediate benefits from precisely describing program invariants. Initial exercises would lead students through using annotations to precisely describe possible program invariants and using analysis tools to detect invalid invariants. Function preconditions and postconditions can be expressed in LCLint using annotations on parameters and return values as well as specialized requires and ensures clauses; ESC/Java supports general requires and ensures clauses. An exercise would start with an annotated program, and require students to make a change to the program that involves strengthening a function precondition. By using the analysis tools, students would identify code fragments that do not satisfy the new precondition. Other exercises might involve weakening the postcondition of a library function and determining what other code needs to change as a result.

Representation invariants provide implicit preconditions and postconditions on member operations. Thinking about representation invariants is an important part of designing good datatype implementations [Liskov00]. By expressing these invariants precisely and using analysis tools to detect possible violations of these invariants, students will gain a better understanding of how representation invariants work, what kinds of explicit invariants are useful, and how they can improve datatype implementations. All attempts I am aware of to require students to include representation invariants in their code have failed to instill the practice of explicitly documenting invariants after the class completes.[2] Students perceive the payoff for the somewhat tedious task of determining and documenting an invariant is too low and do not appreciated the value that comes from being able to use a precisely expressed invariant to automatically detect bugs in code. A possible exercise would ask students to improve the performance of a datatype implementation in ways that involve changing its representation. By precisely expressing the invariant and using static analysis tools to detect possible inconsistencies, students would be able to more efficiently and reliably identify necessary changes to the code.

Further experience with invariants would be gained by using Daikon to dynamically determine likely invariants. The invariants produced by Daikon reveal interesting properties of the program and its test suite. If the test suite is insufficient, Daikon may infer an invariant that is true over the test executions but not true over other possible program executions. In testing libraries, surprising invariants produced by Daikon may reveal weaknesses in the test suite (for example, it never attempts to pop an empty stack). Students would use Daikon in conjunction with static analysis tools to check invariants produced by Daikon. This approach has been used with ESC/Java with promising results [Nimmer01], and will be attempted with LCLint soon [Ernst01].

Security Vulnerabilities

Bad software is by far the most common technical cause of security vulnerabilities [MV01]. The preponderance of security problems stem from programming errors, not from flaws in algorithms or protocols. Nevertheless, security is often not part of an undergraduate curriculum, and rarely discussed in software engineering classes.

Analysis tools can be used to great benefit in teaching students to write secure code. One example is detecting vulnerabilities to buffer overflow attacks. Buffer overflows account for approximately half of all security vulnerabilities [CWPBW00, WFBA00]. LCLint has been used to statically detect likely buffer overflow vulnerabilities in security critical programs [LE01]. By understanding how LCLint detects likely vulnerabilities, and by using LCLint to check programs, students would gain a good understanding of this problem. Checking depends on annotating programs to document design decisions about buffer sizes, which may involve dependencies on

---

[2] My experience is mostly from TA'ing the Software Engineering (6.170 Spring 1993 and Spring 1995) and Compilers (6.035 Fall 1993 and Fall 1995) courses at MIT. In the Software Engineering course, students were taught about representation invariants and abstraction functions, and required to include explicit invariants in their code. By the end of the course, most of the students were capable of producing reasonable representation invariants. In the Compilers course, typically taken the following semester by the better students from the Software Engineering course, students worked in teams on a large semester long project where representations invariants would have been valuable but were not requires. Nevertheless, it was extremely rare for students to include them, and when they did they were invariably either invalid or too weak to be useful. Similar experiences have been reported in discussions with faculty from other schools who have tried similar approaches.

other values. Checking depends on using heuristics to match and analyze known programming idioms. Students will be able to use analysis tools to better understand these programming idioms, and why they make programs easier to analyze and understand.

Extensible Checking

All the checking described so far involves using checks already provided by analysis tools (although annotations are used to influence and describe the specific checks that are done). There is also considerable opportunity to have students define new checking rules. This should not only lead to a deeper understanding of how analysis tools work, but also give students and opportunity to embody new methods and application-specific constraints in checking rules.

LCLint provides support for extensible checking using a meta-state description. Users can define abstract state associated with particular program entities, and define rules for transforming and checking that state. A simple example would be to check proper use of files by adding state associated with file objects that indicates the state of the file (open, closed), and introducing annotations that describe assumptions about that state. LCLint's extensible checking has been used by an undergraduate to develop rules to assist porting applications between Unix and Win32 [Barker01]. A course exercise could involve developing checking rules to enforce application-specific constraints and using extensible checking as an integrated aspect of software design.

In addition to using the built-in extension mechanisms, changing analysis tools by modifying their source code will be instructive and worthwhile. An example would be extending Daikon to support a new invariant. Students do this by defining a new Java class and adding one line to Daikon to incorporate the new invariant. Students could then extend LCLint to support the new invariant. By having students invent and implement new checks, they would develop a deeper understanding of how analysis tools work. In addition, an assignment that asked students to invent new checking that followed from a design principle would lead to greater understanding of the principle as well.

3 Experience and Capability of the Principal Investigators

David Evans

As the primary developer of LCLint, David Evans has considerable experience with lightweight analysis tools. In the seven years LCLint has been publicly available, he has interacted via email with thousands of students and professionals at various levels who have used LCLint in different ways, and gained a good understanding of the issues that confuse and enlighten users of analysis tools. In addition, he supervises several undergraduates and graduate students work on LCLint-related projects. David Evans joined the University of Virginia in November 1999, and has taught courses in Security and Programming Languages. He won a University Teaching Fellowship in 2001.

4 Evaluation Plan

The fundamental evaluation question is can lightweight analysis tools be used to improve teaching of software engineering. Software engineering is not yet at the point as a discipline where there are clear and objective metrics for measuring a student's ability. Some of the subjective criteria we propose to use to evaluate the success of the pilot course include:

Do students gain a better understanding of abstract concepts by using analysis tools?

Do students become better software engineers because of their experience with analysis tools?

Are students able to efficiently manage and manipulate larger programs than with previous techniques?

Do students develop original checking rules and use them effectively?

In addition to analyzing student work on assignments, we will use anonymous attitudinal surveys to assess the effectiveness of our approach. Student surveys would be conducted at the beginning of the course, in the middle of the course, and at the end of the course to assess the effectiveness of different analysis approaches and measure student perceptions.

Analysis can also be done on the student assignments. We will ask students to record the amount of time they spend on each assignment; this will provide some insight into which tools and techniques are used productively in different situations. For some of the assignments, students will use an instrumented version of LCLint that records information about how it is executed and the code it analyzes. For example, we could extract information about the process students' use to add annotations to a program by examining the information produced by the instrumented LCLint.

Perhaps the most definitive measure of success will come from whether or not students continue to use analysis tools on their own after completing the class. If the course is successful, students will understand and appreciate software engineering concepts that are enhanced by the use of analysis tools. Since nearly all students who take our pilot course will go on to take further courses in our curriculum that involve substantial programming assignment, we will have the opportunity to observe and measure differences between how students who took the pilot course and those who did not develop software. In particular, it will be possible to measure how much students who took the pilot course continue to use analysis tools voluntarily in later courses.

Another measure of success will be how many other courses adopt analysis tools. We will evaluate our success based on whether or not we are successful in convincing designers of other courses to exploit analysis tools and to make use of the materials we develop.

5 Dissemination of Results

The main audience is designers and instructors of software engineering courses. Our results should be applicable to a wide variety of courses ranging from industrial-oriented introductory software engineering courses, to more theoretical and methodology-focused software engineering courses, to special topics courses in formal methods. The materials we produce, including software, assignments, notes and course designs, will all be made available on the Internet for unrestricted use. We will seek to present our results at computer science education conferences (SIGCSE) and organize a tutorial on using analysis tools to teach software engineering at both education and software engineering conferences. After the grant concludes, the materials will continue be maintained and supported by the Lightweight Static Analysis research group. We have a good track record of supporting LCLint use in teaching and industry, and believe continued efforts to support use of lightweight analysis tools in education will contribute to both our educational and research goals.

References

[CWPBW00] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie and Jonathan Walpole. *Buffer overflows: Attacks and defenses for the vulnerability of the decade.* DARPA Information Survivability Conference and Exposition. January 2000.

[Detlefs98] D.L. Detlefs, K.R.M. Leino, G. Nelson, J.B. Saxe. *Extended Static Checking.* Compaq SRC Research Report 159, 1998.

[Dijkstra76] Edsger Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. *Dynamically discovering likely program invariants to support program evolution.* IEEE Transactions on Software Engineering, February 2001. (Previous version in International Conference on Software Engineering, May 1999.)

[EGHT94] David Evans, John Guttag, Jim Horning and Yang Meng Tan. *LCLint: A Tool for Using Specifications to Check Code*. SIGSOFT Symposium on the Foundations of Software Engineering. December 1994.

[Ernst00] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants.* PhD Thesis, University of Washington. August 2000.

[Ernst01] Michael D. Ernst. Personal Communication, April 2001.

[Evans96] David Evans. *Static Detection of Dynamic Memory Errors*. SIGPLAN Conference on Programming Language Design and Implementation. May 1996.

[Evans00] David Evans. *LCLint User's Guide*. Version 2.5. May 2000. http://lclint.cs.virginia.edu/.

[Gries81] David Gries. *The Science of Programming.* Springer-Verlag, 1981.

[Johnson78] S. C. Johnson. *Lint, a C Program Checker*. Unix Programmer's Manual, AT&T Bell Laboratories, 1978.

[LE01] David Larochelle and David Evans. *Statically Detecting Likely Buffer Overflow Vulnerabilities.* To appear in 10th USENIX Security Symposium, August 2001.

[Leino01a] K. Rustan M. Leino. *Extended Static Checking: a Ten-Year Perspective.* To appear in the Proceedings of the Schloss Dagstuhl Tenth-Anniversary Conference. Springer LNCS volune 2000, 2001.

[Leino01b] K. Rustan M. Leino. Personal Communication, May 2001.

[LG86] Barbara Liskov and John Guttag. *Abstractionand Specification in Program Development.* MIT Press, 1986.

[Liskov00] Barbara Liskov (with John Guttag). Program Development in Java: Abstraction, Specification, and Object-Oriented Design. Addison-Wesley, 2000.

[MV01] Gary McGraw and John Viega. *Building Secure Software.* To appear, Addison-Wesley, 2001.

[NE01] Jeremy W. Nimmer and Michael D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. Submitted for publication, 2001.

[Orcero00] David Santo Orcero. *The Code Analyzer LCLint*. Linux Journal. May 2000.

[PG00] Pramode C E and Gopakumar C E. *Static checking of C programs with LCLint*. Linux Gazette Issue 51. March 2000.

[WFBA00] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. *A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities.* Network and Distributed System Security Symposium. February 2000.

[Wing00] Jeanette Wing. *Weaving Formal Methods into the Undergraduate Computer Science Curriculum.* Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology (AMAST), May 2000.