Teaching Software Engineering in a Compiler Project Course

WILLIAM G. GRISWOLD
University of California, San Diego

A compiler course with a long-term project is a staple of many undergraduate computer science curricula, and often a cornerstone of a program's applied-engineering component. Software engineering expertise can help a student complete such a course, yet that expertise is often lacking. This problem can be addressed without detracting from the core class materials by integrating a few simple software engineering practices into the course. A domain-specific, risk-driven approach minimizes overhead and reinforces the compiler's material, while treating the project as a "real world" enterprise reinforces key engineering lessons. The method might be called "syntax-directed software engineering," being driven by specification centered around a BNF-style grammar. Engineering lessons are reinforced with general engineering principles and contextualization of the subject matter. The approach can be taught without substantial software engineering background. The domain-specific risk-driven software engineering approach can be applied in other courses such as operating systems by redesigning the practices around its domain.

Categories and Subject Descriptors: K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education; Curriculum*

General Terms: Experimentation, Human Factors

Additional Key Words and Phrases: Compiler course, software engineering

1. INTRODUCTION

With a plethora of exciting topics emerging on the computer science scene–computer security and the internet, to name two–it is becoming increasingly difficult to ensure that undergraduates obtain adequate exposure to every topic that their professors think necessary. Moreover, the software projects that professors would like to assign–and students would like to implement–are becoming increasingly complex, and would be much easier if the students had software engineering expertise. Yet it is just as likely as not that such a project would be assigned before a student has taken a software engineering course.[1]

This is the circumstance at UCSD, where software engineering is a senior elective that often comes too late, if at all, for undergraduates who face a demanding two-quarter compiler project course (CSE 131 A-B) in their junior year. Students in this course find, particularly in the second "B" quarter of the course, that they are ill-prepared to design a complex software system, rigorously test it, work productively in teams, and organize their time for a project deadline that is five weeks away. Although students are well versed in

Author's address: Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA 92093-0114; email: wgg@cs.ucsd.edu

[1] The exact reasons for the uneven penetration of software engineering in the core of undergraduate computer science curricula are a matter of continuing debate and beyond the scope of this paper. It suffices here that it is indeed a problem.

designing, implementing, and testing small programs, few have worked on a system that required the skills, teamwork, and discipline to build a quality software system on a schedule. Even if they had taken the software engineering course and had experience in building large systems, the unique properties of the topic and the short quarter demand a lean software engineering process that is customized to the domain, which the students may be ill-prepared to design for themselves. Consequently, the lessons of compiler algorithms and design can be lost in a mass of software chaos.

To ameliorate these problems, I integrated basic software engineering techniques and principles into UCSD's compiler construction course. The main challenge is that there is plenty of compiler material to teach as it is, which I overcame in a number of ways. First, I introduced software engineering material only when relevant to problems that many students had encountered in the past. Second, I focused on techniques and principles, not methodologies, counting on the intelligence of the students to use the principles to inform their broader activities. Third, the techniques are customized to the domain (i.e., compiler construction) and to the scale of the project. In particular, the approach centers on the project's syntax-directed specification, both giving the techniques added power and reinforcing the students' appreciation for this significant compiler concept. Finally, the engineering and compiler material is situated in the broader context of society and the software industry, motivating the subject matter and rationalizing the organization of the course. If the students never take a software engineering course, they may have an incomplete education, but at least the knowledge that has been conveyed is useful, and hence appreciated. For impressionable students building their first complex system, appreciation imbued with experience and a few principles prepares them for further learning on their next system.

The next two sections of this article describe the original design of CSE 131B and the problems that students encountered in it. Sections 4 through 7 detail how software engineering was introduced into the course and why particular choices were made. Section 8 then discusses the impact that the approach has had on the course and the students, and compares the method to an earlier approach. Before concluding, Section 9 briefly discusses how software engineering might be introduced in other project courses using a similar approach.

2. COMPILER CONSTRUCTION CSE 131B

Compiler construction is taught over two 10-week terms in the standard "phase order" of the compilation pipeline seen in so many compilers texts. The first quarter, 131A, covers front-end issues such as lexical analysis, parsing, and scope-checking of identifiers. The second quarter, 131B, covers static semantic- checking (primarily type checking) and code generation. Grading is divided equally between exams and the compiler construction project.

The project assignment is for a pair of students (a team) to progressively define and construct a compiler.[2] Every few weeks the students are given a project assignment that adds a useful capability to their compiler. Although the final compiler is not complete until

[2] Three students are allowed in a group if there are an odd number of students in the class or someone drops out of the course during the term.

the last week of 131B, at each major phase a useful program is delivered: a lexical checker, a syntax checker, a scoping checker, a type checker, and finally the working compiler with code generation. The most demanding parts of the project come in 131B, in part because the formalisms for type-checking and code generation, albeit powerful, are not as readily automatable.[3]

In order to reinforce concepts from programming languages and computer architecture, a real programming language and target assembly language are used as the source and target of the compiler. Due to time constraints, the students constructed a non-optimizing compiler for a useful subset of a "lean and clean" language. The language in current use is Oberon-2, as it embodies modern language concepts such as object-orientation, yet with few complex features. Even with simplification, the resulting compiler will usually be on the order of 5,000 lines of code, virtually ensuring that the members of a team will have to share the workload in order to finish the project.

In addition to the Oberon-2 language manual, the project assignments provide more precise (and formal) specifications that resolve ambiguities, refine requirements, and clarify which features must be handled. In keeping with standard compiler construction practice, the specifications are usually in the form of a language (automaton) description such as a BNF-style context-free grammar in which each grammar rule is associated with structured English specification of the required behavior.

The project is progressive not only in that each phase builds on the ones prior, but the students must build on their own previous work. The only exception is when students move on from 131A to 131B, at which point we provide a compiler on request, albeit with no guarantees about its quality (it tends to be very good, but certainly not perfect).

Grading the project at each phase of delivery is designed to be straightforward: the project is run on a wide-ranging set of test cases. The grade assigned is roughly the number of test cases that succeed. Success is determined by correct behavior in two dimensions: correct behavior and user-friendliness. The latter is largely determined by the helpfulness of error messages; for example, "type error" is a bad message because it does not report the line number or the kind of error detected. Partial credit may be assigned if, for example, an error message is reported at the appropriate point in the parse but the message is not helpful. No credit is given for coding style or other "non-functional" properties of the software. Although this grading system was originally dictated by a lack of resources, as discussed later, it turns out that this policy embodies a number of key engineering lessons, if handled properly. The grading test cases are released immediately so that students can perform their own "grading" and undertake any necessary repairs for the next phase or make an informed decision to drop the course.

3. CRITIQUE

The compiler milieu and basic project organization provide an excellent basis for learning a number of key engineering lessons. For example, use of the parser generator yacc teaches the engineering value of tools (or components, depending on the perspective) in software development. Beyond tools, knowledge about how to design a compiler is in a relatively advanced state, with ample guidance available in the form of software architectures (e.g.,

[3] This article is in part inspired by Johnson's observations about the vital relationship between compiler theory and practice [Johnson 1978].

the pipeline design), symbol table design, type representation, and so forth. The project organization also provides valuable experience for the student with phased system development in a team.

Although these lessons and experiences are a vital part of a computer science education, the students were relatively unprepared for them and the course did not address this lack: little guidance was given about how to work in a team or how to define a large system (incrementally or otherwise). In particular, classroom time was largely dedicated to the algorithms, data structures, and theory germane to compiler construction; textbooks adopt a similar focus. The standard compiler architecture as presented in most texts provides little guidance (or even misguidance) on these matters. An inexperienced programmer might suppose that the four major phases of compilation will be linked in data-transforming chains mediated through a symbol table. Thus, a first cut at a division of labor might be to define a system of five components and to have each member of the team in charge of one or more phases. Of course, such components are too large as units for modularization (but perhaps fine units for code ownership). Moreover, the division of labor is unsuited to timely phased delivery, since only one programmer would be coding for any particular phase (and not all phases are equally difficult).

As a consequence, students often complained that the project was unfairly difficult (despite many quarters of simplification), and yet they trivialized the project as "just a lot of coding." On the other hand, the class material seemed straightforward to them, at least comparatively. Likewise, for a number of reasons, many students felt the project or its grading was unfair. Many students objected that their grades depended on another student's performance. Others complained that they had to continue building on their previous flawed code (or drop out), since it put them at a disadvantage with students who had done well. Others begged to be allowed to fix just one line of code after a phase deadline because their compiler had crashed on a vast number of test cases due to a "minor" oversight. Essentially, the students saw little connection between "compiler construction" and the causes of their troubles, and hence felt their grades did not reflect their knowledge.

For students who did poorly in the course, these complaints were often complemented by revelatory excuses:

- Getting integers to work was easy, but we couldn't get the compiler to work for user-defined types.
- We ran the compiler on all your practice test cases, so we figured it was OK.
- Our compiler core dumps, and we couldn't figure out why.
- I had to wait for my partner to finish before I could start.
- My partner didn't finish his part.
- The lab's computers were too overloaded, so we couldn't finish in time.
- We worked on it all week but we couldn't finish in time.
- My partner and I tried to integrate our code the day before the deadline, but we couldn't get it to work in time.
- I did the first project, so it was my partner's turn to do the second project.
- I thought my partner was working on the project, but I found out last night that my partner dropped the course.

These reasons are largely unrelated to compiler design *per se,* or to the students' understanding of it. Indeed, each reason falls into one or more classic causes of software project failure:

- *Software design.* Students were incapable of examining the software requirements, team strengths, project deadlines, and other factors to guide their design efforts. Such low-level problems later led to failures in communication, scheduling, etc.
- *Testing.* Students did not have a clear idea of what it meant for their systems to be reliable, did not know if their software was reliable, and did not know how to test their software in a systematic fashion.
- *Use of tools.* The students did not know how to use debuggers, compilers, configuration management tools, etc., to manage the software and their projects.
- *Scheduling.* Students failed to allocate enough time for the project, presumably due to competing demands on their time, unfamiliarity with compiler construction, and lack of experience in building a system that takes more than a couple of weeks to build.
- *Teamwork and communication.* Students failed to work effectively with their partners, presumably because they were unaccustomed to team projects of any length. Responsibilities outside the class were likely a contributing factor as well.

Although t hese problems are classic failure modes for commercial projects, the underlying causes here are different, such as a lack of background and wide-ranging commitments outside the class (e.g., other classes, a job).

4. WEAVING SOFTWARE ENGINEERING INTO COMPILERS

To eliminate the students' failure modes and address their feelings that the course was unfair, I refocused the course theme around compiler construction as an exemplar of complex system construction. Since mature domain knowledge is a vital prerequisite to successful system construction, traditional compiler material would not be replaced, only augmented and enriched, by more general engineering knowledge.

Given that course time is limited, and operating on the principle that formal education is only a springboard for a lifetime of learning, I chose to address these specific failures with a narrow, domain-specific approach to software engineering that could be seamlessly integrated into the basic compiler material.

One of the key concepts in compiler design is syntax-directed translation (from one language into another), which has a rule-based flavor: as the compiler reads each fragment of the program, it recognizes its syntax (if-statement, addition expression, etc.) and performs a translation of the fragment according to the prescribed semantics of that syntactic category. In fact, programming languages are often specified as a set of syntactic constructs (i.e., a grammar) with a specification for each construct. The specification for a piece of syntax typically consists of two parts: a set of additional integrity constraints (e.g., required types of operands) and a specified behavior. Such a specification can serve as the driver for a compiler-specific software engineering approach in order to avoid project failure.

The classic method for failure-avoidance in engineering is to use a risk-driven process

[Petroski 1994], as embodied in the Spiral model of software development [Boehm 1988]. The Spiral model approach has the advantage that the general idea–iteratively identify your biggest risks and work to resolve them–can be specialized to any software project, not just a compiler. With this approach, students have a better chance of taking their experience from the compiler course and applying it to other projects in the future. Moreover, with a streamlined process, students are less likely to be frustrated by being taught "irrelevant" software engineering material whose application is unclear, thus encouraging a lifetime interest in software engineering.

Each team's measure of success is operationalized as their grade. In terms that the students can readily understand, then, their primary risks are correctness and schedule: If the compiler is buggy or turned in late, the team's grade will suffer. All other risks ultimately transmute into correctness or schedule risks. Meeting both correctness and schedule requirements is best met by incrementalizing the software development process: precious time is spent on new features only if earlier ones (which the new features depend on) are finished and known to work.

A software process designed around a syntax-directed specification can be incrementalized to reduce risk in a number of dimensions: design, implementation, testing, and delivery of the product can be staged in terms of the language's syntax. Moreover, the process can be further staged according to the specification of static qualities (syntactic correctness itself and the integrity constraints) and dynamic qualities (executable behavior).

This risk-driven approach is introduced into the course in three ways, described in the next three sections: sensitization and contextualization, techniques, and principles.

5. SENSITIZATION AND CONTEXTUALIZATION

Many students entering CSE 131B lack appreciation of the problems they are about to encounter–and in ten weeks it will be too late. Consequently, the course introduction presents the particular challenges the students will face in their project, conveyed both as stories about prior projects and the specific failure modes listed in Section 3.

Many students seem immune to these warnings, perhaps due to over- confidence or handholding in previous courses. To lay out the consequences of failure in graphic terms, the students are told that: their project will be graded solely on how well it performs on our rigorous test cases, late projects will be severely penalized (5 minute grace period, then 1% per minute late), and all members of the team receive the same grade. This provides both an unambiguous and realistic context for the application of software engineering.

Sensitization alone has proven inadequate. Many students feel that hard work should be rewarded, or exceptions allowed. Many are used to being graded on programming style as well as the reliability of their software. Consequently, the introduction–complemented by digressions during the course–also discusses the larger context of software in society. The students are reminded that a software product's success is measured by the benefits it brings to its users (i.e., usefulness), and consequently to the company that developed the software (e.g., profits from sales). An unreliable or late product will be displaced in the marketplace by other products.[4]

[4] Although programming style is a superficial quality to grade in an upper-division course, software design would be appropriate to grade. When we attempted to grade design, teaching assistants repeatedly cited both the

Of course, engineering is not only about profits, but also about social responsibility. I remind the students that software is everywhere, including life-critical systems such as nuclear power plants (one operates only 30 miles from San Diego), airplanes, medical equipment, and banking.[5] Consequently, quality software is not just a classroom topic, but a responsibility that the students will carry with them daily as software developers.

With this context, I explain that the rules established for the class are no different than what is expected of them outside the classroom, and that this is essential to providing a classroom context suitable for learning software engineering. This applies equally to the design of a compiler, an example of a software system that must be useful to be successful.

6. TECHNIQUES

For a 10-week project involving two team members, lightweight methods are the most appropriate. With scheduling the overriding concern, any unnecessary effort is to be avoided. For example, although communication is a risk factor for busy students, an emphasis on documentation is unwarranted given the small size of the project. With just two people in a team, communication risks are better overcome by clear separation of responsibilities and low barriers to communication (e.g., e-mail, instant messaging, meeting after class, and intuitive software interfaces).

Specification. Due to time constraints, it is not feasible to teach students how to write specifications, especially since they likely have no experience doing so. The focus here is on the first step, learning to *use* a specification.

The students are given a syntax-based specification intended to drive the entire software process (see Figure 1). Although based on the actual grammar, it is more succinct, ignoring precedence issues and the like, since these were resolved earlier. The domain-specific, semiformal document is novel to the students, compared to the low-level interface specifications that they typically see in lower-division courses. Since this is not an interface specification, there are a number of potential ambiguities (a classic specification issue) that the students must appreciate and overcome. Moreover, despite years of refinement by two professors, students still uncover errors in the specification. (This frustrating fact further supports our decision to shield students from writing the specification.) The students are also given some "non-functional" requirements to work out, such as "error messages should be informative and include an accurate line number," with only examples to clarify the requirement.

Design. A modern software engineering course will likely teach some form of object-oriented design (OOD). I have found that although OOD results in effective design, its pure form is too abstract for most college juniors and seniors. With a design space that is initially constrained only by the specification, students may become overwhelmed. Since it is impractical to teach general OOD as part of the course, and it is not appropriate to dictate the compiler's design to the students, a simplified form of OOD appropriate for the domain is taught.

To help students make the transition from specifications to design, students are encouraged to "code to the specification" in a syntax-directed fashion. In particular, the

---

subjectiveness and difficulty of grading design. Indeed, Parnas claims that a design is best assessed through simulation [Parnas and Weiss 1985]; an arduous activity for the purpose of grading.

[5] Detailed examples are hard to come by, but a couple suffices: [Gibbs 1994; Leveson and Turner 1993].

students are advised to *first* write code that reads like the specification, without much thought for the classes or algorithms that will be required to make the code runnable. Then

```
Phase I (40% -- 1 1/2 weeks)

Declarations, statements and expressions consisting of variables
and literals of basic type (integer/boolean/real).

Check #1: Detect a type conflict in an expression -- that is, expressions
of the form

        X OP Y

where the types of either X or Y are incompatible with OP....

In Phase I, the operands are restricted to simple variables
of basic type (integer, real, boolean) and literals (this
will be extended in later phases).
```

(a)

EXPRESSION COMPATIBILITY

Operator	1st Operand	2nd Operand	Result Type
+ - *	numeric	numeric	smallest numeric type including both operands
/	numeric	numeric	REAL
DIV MOD	INTEGER	INTEGER	INTEGER

(b)

Fig. 1. An excerpt from (a) the project 1 document, and (b) the typing rules from the language specification.

the students are to write the class definition(s) and the methods (function members) used in the specification-like code. For example, Figure 2 contains code derived from the compiler and language specification fragments shown in Figure 1. First, the code fragment at the top of the figure is written, using the compiler specification's structure and the language specification's terminology. Then, the two methods below are written. From these two methods, it is apparent that the symbol table entries need type tests such as `isNumeric`. Thus, the design of symbol objects (called symbol table entries or STEs) and type objects are driven top-down from the specification, rather than considered in the abstract. [6]

This approach has two orthogonal benefits. First, it frees students from the blank page design problem of trying to figure out what classes to introduce, what their interfaces should be, and how they should be implemented. The design for a class is determined bottom-up from the ways in which its clients use it. The design is also, somewhat

[6] Note that this is not a violation of the dictum to *specification writers* that the *contents* of a specification should not intrude on the realm of design and implementation. I am advising *programmers* that the design should mirror the *structure* of the specification.

paradoxically, determined top-down from the specification, leading to the second benefit: coding to the specification means that the compiler's code structure and terminology will mirror the specification structure and terminology as much as is reasonable. Structural congruence provides traceability between the specification and the implementation, enabling easy assessment of project status (which features are done and which are not) and

```
...
if (OpSTE->isCompatible(expr1, expr2))
  return OpSTE->resultType(expr1, expr2);
else  // error
...
bool ArithOp::isCompatible(STE *expr1, STE *expr2) {
  return (expr1->isNumeric() && expr2->isNumeric());
}
Type *ArithOp::resultType(STE *expr1, STE *expr2) {
  return expr1->type();  // extend for coercions
}
```

Fig. 2. Design example from the class lecture notes that demonstrates how to design from the specification. The code fragment at the top is written first, and then the bodies are written later. Note the use of structure and terminology from the specification to guide the design. The code bodies are only enough to implement exact type match, demonstrating incremental implementation.

testing and debugging (what code is broken when a test does not work). This also minimizes the number of concepts the students have to remember and yields a reasonable object-oriented design.[7]

This pseudo-OOD approach, although not the latest and greatest software design methodology, is simple, concrete, and yields intuitive code. The students are advised that the approach does not work in every case, but that they should have a good reason for deviating from the implied design. Later in the course, examples arise that show the need to design for change, not just intuitiveness.

Congruence with elements of the domain that do not explicitly appear in the specification, such as assembly language, also influences design.

This design method is not taught as a separate idea or in a separate unit of the course. Most of type checking and code generation are taught with this method, explicitly linking compiler concepts and compiler design through the bridging concept of syntax-directed translation. An intended consequence is that it will help students turn abstract ideas into concrete action.

Implementation. Most students, given a design, are reasonably good at implementing it. However, students have a tendency to write their entire compiler and then to try to compile it. This approach may have worked for them on smaller programs, but it is a common cause for failure on the compiler project. Consequently, the students are advised that their

[7] A more advanced version of the arithmetic operator design outlined in Figure 2 creates classes of operator *types* that provide the type-checking operations. Then, each operator object points to an operator type object, and the operator object implements the same type check operations through delegation to the type object. This alternative design can be introduced incrementally by showing how the delegation to the type object can be implemented by small changes to the existing type-checking methods.

compiler should be "always runnable." Hence, their systems are "always testable," "always usable by your team," and "always deliverable," both minimizing delays on other efforts and ensuring that completed features can be graded even if a milestone is missed.

The design approach helps incrementalize implementation. Given reasonable abstractions that match the specification, a simple implementation that quickly implements an important subset of the functionality is possible. For example, the body for the `result-Type` method in Figure 2 works for exact type matches only, but can be written quickly, permitting the testing of certain features and other parts of the compiler early in development. The method can later be extended to handle coercion at incremental cost because it does not disturb the overall design.

Testing. In a typical software engineering course, testing might encompass the varied kinds of testing and test-suite design, including issues such as adequacy, minimality, and coverage. Although past experience suggests that this time would be well-spent, the primary problem is that the students simply test haphazardly and not enough.

Since any technique would remedy the problem and demonstrate the value of planned testing, it is most natural to employ a specification-based "black box" method in the form of syntax-directed testing. The students are advised to derive their tests directly from the specification in a syntax-directed fashion. In particular, for each grammar rule, students should write a test case for each integrity constraint and behavior in the rule's specification, plus important combinations of them (see Figure 3). The students are also advised to do stress testing; that is, write big test cases, weird test cases, etc., to catch implementation errors. Finally, the students are reminded that testing finds bugs, but does not prove correctness, and hence testing requires pessimism about what is actually known about the tested compiler.

This syntax-directed approach provides three benefits. First, all of the usual benefits of specification-based testing accrue: a concrete user-oriented perspective on expected behavior and a clear concept of adequate coverage of the test cases (e.g., code-based coverage does not guarantee that all behaviors have been tested). Second, the traceability among the specification, design, and test cases helps in debugging, since erroneous behaviors can be traced straightforwardly to the code. Finally, traceability provides a concrete measure of progress: a grammar rule doesn't "work" until it passes its test cases.

Scheduling. The project's short duration and focused objectives permit simple scheduling methods. However, the short duration with strict deadlines dictates scheduling that permits feature omission in favor of meeting the schedule. Consequently, the schedule is managed around milestones at which a key subset of functionality has been both implemented and tested (See Figure 1(a)). If a team falls behind, it can at least turn in a gradeable subset of the system. The organization of the specification, design, and testing around the language grammar makes the project's status visible, easing scheduling based on subsets of grammar rules. This schedule, like the specification, is provided to the students, but except for the two graded project milestones (type-checking and code generation compiler phases), it can be customized to their needs.

Consistent with this incremental approach, the simple, basic features are scheduled in the early milestones of each major phase, and more advanced (and often dependent) features are scheduled later. This not only gives the students something on which to build in the later milestones, but also helps them gain some experience, confidence, and grade points early on. In fact, a higher percentage of the grade is assigned to the basic features,

making it clear to the students that a subset of functionality is valuable, in particular basic functionality is more valuable than advanced functionality.

Team management. Due to the small team size, *ad hoc* management techniques could almost suffice. However, students have historically shown bad judgment about division of responsibilities, frequency of meetings, and the like.

```
(* RULE:
 * Expr     -> Expr1 AddMulOp Expr2
 * AddMulOp -> '-' | '+' | '*'
 *
 * "if  the type of Expr1 is numeric
 *  and the type of Expr2 is numeric
 * then the type of Expr is the smallest numeric
 *      type including the types of both operands,
 * otherwise error"
 *
 * The goal is to test ''all possible''
 * combinations of the application of this
 * rule and the associated check. If all the
 * tests pass as expected, you should have
 * high confidence in the result. There are 3
 * operators appearing here (-, +, *), and 3
 * types, INTEGER, REAL, and ''error''. There
 * are also 2 operands (left and right) that
 * permits try all pairs of types. One might
 * also count ''all'' kinds of erroneous
 * types: BOOLEAN, etc. You might even have
 * an ERROR type. Unfortunately, there are
 * an infinite number of user defined types,
 * so we can't do that. You might try one or
 * two user-defined types, or verify by visual
 * inspection of your code that user-defined
 * types are handled here the same as BOOLEAN.
 * ...
 *)
VAR x, y : INTEGER;
    r, s : REAL;
    b, c : BOOLEAN;
BEGIN (* assignments here for syntax correctness *)
  x := x + y; (* expr'n is type INTEGER *)
  r := r + s; (* expr'n is type REAL *)
  r := x + r; (* expr'n is type REAL *)
  r := r + x; (* expr'n is type REAL *)
  r := x + b; (* num. expected, got BOOLEAN *)
  r := b + x; (* num. expected, got BOOLEAN *)
  r := r + b; (* num. expected, got BOOLEAN *)
  r := b + r; (* num. expected, got BOOLEAN *)
  (* this might report two errors *)
  x := b + c (* num. expected, got BOOLEAN *)
END.
```

Fig. 3. Example test case given to students that demonstrates syntax-directed testing.

Yet it is counterproductive for the instructor to dictate a management structure, both

because each team has its special needs and abilities, and because students will be inclined to blame management problems on the instructor rather than take responsibility themselves. For similar reasons, teams are encouraged to make important decisions by consensus rather than dictate.

During sensitization, students are advised that communication is a key risk, due to the divergent responsibilities they face beyond this one class. Students are advised to work together at a common location several times a week to minimize communication problems. Students are also told that since the project is too large to be completed by one person, each member has to contribute equally if the project is to succeed. Furthermore, because each team member possesses special skills, the team's full potential is best realized by determining responsibilities according to abilities. Students are then told of two complementary ways that labor could be divided effectively.

First, it is suggested that they each take responsibility for a portion of the grammar rules (i.e., a portion of the specification). The concreteness of this approach makes responsibilities clear: if a part of the language is not processed properly by the compiler, then the "owner" of that part is responsible**.** This division also has the benefit that each team member can concretely see the fruits of his or her labor. A downside is that responsibility for crucial utilities such as the symbol table are unclear. However, I have seen too many projects fail with the excuse "I wrote all the grammar code, but my partner couldn't get the symbol table to work." Consequently, I recommend that such crucial portions of the compiler be designed (and implemented) as a team using paired programming [Beck 1999]. The labor time that is lost is more than compensated for by discovering problems and catching errors early. Pairing also helps achieve crucial buy-in by the entire team, avoiding sore feelings and finger-pointing later.

Second, labor may be divided by task specialization. In a typical division, one student designs and implements the code, the other writes and performs the tests. This is especially effective for teams that possess a clearly superior designer/coder. Division of responsibility is likewise clear: the designer/coder builds the system and performs basic runs, and the tester ensures that the compiler is ready for delivery on the current milestone. Moreover, this approach reduces clashes on design issues and eases code management.

Tools. The benefits of tools for automating tedious activities, especially when linked to domain formalisms, are made obvious to the students through the use of a parser generator like yacc. However, many students have not thought about configuration management, test harnesses, or system instrumentation, so the class teaching assistant must prepare a set of online notes regarding the use of tools for projects. The students are also advised to implement their own instrumentation facilities for dumping the symbol table and tracking key system events. In the first discussion section, time is set aside to review these tools and their importance. The importance of an automated test harness for cost-effective regression testing after making (even apparently minor) changes is emphasized when discussing incremental development.

7. PRINCIPLES

Since the students are given only one example of the simplest techniques, they are best supported by overarching principles that can both guide the application of the techniques and inform the solution to problems that fall outside the scope of the techniques.

People seem to learn best through repetition and experience, so these principles bear

reviewing in the context of thorny problems that arise in the project. The following time-tested software engineering principles appear in CSE 131B as overarching themes for the course. Of course, students have seen many of the principles in-the-small in earlier courses; reinforcement at this scale brings these principles to life in the realm of engineering. My method of teaching these principles in a lecture is demonstrated through an example at the end of this section.

Failure avoidance by risk reduction. Software projects succeed by failing less than their competitors. By maintaining a focus on the risk factors to the project and introducing practices and technology to minimize them, the project is much more likely to succeed [Boehm 1988; Petroski 1994]. The students are periodically reminded that the overriding risks are scheduling (strict project deadlines), communication among busy team members, and technology (e.g., unfamiliarity with assembly language, network failures).

Divide and conquer. Hard problems are best solved by breaking them into intellectually manageable pieces. Risks can be reduced by isolating them: intellectual complexity can be reduced by breaking problems into manageable subproblems, project responsibilities can be parallelized by cleanly dividing activities, and new software problems can be solved with old solutions if packaged in good designs. Related principles are the *modularity principle*, which guides the separation of implementations from their uses, and the *incrementality principle*, which advises taking small steps because large ones too often prove unmanageable. The syntax-directed software engineering approach is exemplary of the divide-and-conquer method.

Conceptual integrity. Problems or entities that are similar should employ similar solutions; and problems that are different should employ different solutions. From a metaphysical, large-project perspective, this can be stated as follows: a system should appear to be the product of a single coherent mind [Brooks 1975].

When encountering a new design problem, young engineers are prone to two kinds of mistakes: devising the "perfect" special-case solution or reusing an ill-fitting one. Both can result in a fragile system that is difficult to understand. Many special solutions can cost precious time in implementing the new technique when an old one could have been used. A plethora of solution types may require extra intellectual effort in order to understand an unfamiliar part of the system, as well as cause conflicts when new code is introduced that must work with more than one solution. Reusing ill-fitting solutions costs much time in getting the old solution to work (all too often, students discover this first-hand by trying to reuse a "dictionary" class for their symbol table from a data structures class), and can create the false impression that an entity is less (or more) than it really is (e.g., the dictionary implies stability, whereas a symbol table is constantly elaborated).

Structural congruence. A solution should reflect the structure of the problem that it solves [Jackson 1975; 1976]. This is a corollary of the conceptual integrity principle (and is also a tenet of object-oriented design, which uses the problem domain as a starting point in design). For example, students are repeatedly shown how the nested structure of the programming language is reflected in a stack-based bottom-up parser, the symbol table, and all the way to the assembly language output. Without such intellectual control, implementing and debugging a compiler would be virtually impossible. Towards the end of the course, the students also discover that structural congruence has its limits; a concise, efficient, or flexible design may combine or generalize structures in a way that compromises congruence. Yet it is a valuable starting point in design, since all things being

equal, simpler is better.

Application. Principles are effectively conveyed through examples. One situation that exemplifies several principles arises in the handling of inherited attributes (program properties that, conceptually, are passed from higher in the parse tree to the current construct), which are not handled automatically by the parser because it operates in a bottom-up rather than a top-down fashion. I ask students to suggest solutions to implement inherited attributes in the context of the exit statement (like C's break). Students typically suggest a global variable or a counter (to count loop-nesting depth). After some discussion, I guide the students to consider a solution that uses a stack (a generalization of the counter). The stack reflects the nesting structure that must be tracked, and is not only capable of tracking the level of nesting (to check if the exit appears in a loop at all), but also the goto label that must be generated in the assembler output in a later project phase.

I then observe that a generic stack implementation would be suitable to solving the many inherited attribute problems that lie ahead, and that in fact their symbol table is one such (giant) inherited attribute. At this point a review of the lessons of this exercise is possible: Splitting off and solving the inherited attribute problem once and for all (e.g., divide and conquer) means that it will not have to be solved again, saving considerable labor. Likewise, bugs or design problems in the handling of inherited attributes are isolated to a single piece of code. The structural congruence of this solution is attractive since it makes the solution easy to understand in terms of the problem it is solving, and is more likely to be resilient to change because it is an accurate model of the domain. The conceptual integrity (consistency) achieved by reusing this solution over and over again is attractive because team members would then struggle less with the peculiarities of their teammates' code.

8. DISCUSSION

Sensitization and contextualization place compiler construction in the engineering context, not only broadening the apparently narrow lessons of the course but also rationalizing the course structure. Syntax-directed software engineering techniques seamlessly integrate engineering practice into compiler design, maximizing their benefits, while at the same time minimizing the introduction of new concepts. Engineering principles serve to reinforce and generalize the specific lessons of the course. Collectively, then, these three elements serve to reinforce each other as well as the relevance and richness of the compiler construction material itself.

Although there was little change to the basic class or project structure, introducing this approach was non-trivial in two ways. First, it required a careful customization of software engineering practices to fit the domain. Second, this material had to be integrated seamlessly into the class. Customization took quite some time, requiring two rewrites of my notes, although this is in part because I did not fully appreciate the problems that students were facing on the project. Integrating the material led me to write a complete set of coherent notes for the students. The classical organization of most compiler texts almost precludes following one closely, as they emphasize algorithms and data structures rather than the design issues that most trouble students. Although many examples were borrowed from the text, working out the software engineering implications required additional work. On the other hand, the notes are very popular, as most students view the text as unnecessarily complex. Readings are still assigned from the text, although most students

seem to use the book as a reference only, preferring to read my notes.

8.1 Evaluation

In this context, quantitative assessment of trends is fraught with difficulty. This approach was introduced gradually over seven years, in response to difficulties as soon as they became apparent. Over that time, the computer science student body has changed significantly. Growing interest in computer science has dramatically increased class size, and many students are now in computer science because it is a good career move rather than an intellectual interest. Many computer science students now hold demanding programming jobs, whereas before they were likelier to take out loans or work on campus. Efforts to stop cheating have increased dramatically in recent years, adding a dimension of unpredictability. Together, these factors have fundamentally changed the learning landscape. Year-to-year changes in the course itself, such as the individuality of the teaching assistants, have added another level of unpredictability. Finally, our grading methods have improved, as the graders now write test cases using the same methods taught in class.

Anecdotal evidence suggests that students have been affected dramatically. Some students will always take to one topic while disliking another. Thus the focus on the two aspects of the project – compilers and engineering – has widened the audience, although students with extreme biases are prone to complain about the intrusion of the material that does not appeal to them. However, it is difficult to judge the balance of opinion, since the majority of students who spoke to me outside of class were positive about the engineering material, while written comments in course evaluations tended toward invective. Many students commented that the "compiler wrote itself" due to recommendations on how to approach the design. Other students expressed pride over the quality of their test suites and their (well-placed) confidence in the quality of their compilers. Except for the most successful students, there seems to be widespread agreement that the project is still too difficult, despite the trend, over time, to simplify the project. I believe that in large part this attitude can be attributed to the fact that 131B generally has been the first such course that students encounter, representing a change in expectations as the students move into the upper division.

Job recruiters, however, have a more uniform perspective: student performance in the course is a clear predictor of success. Recruiters frequently ask for more courses that incorporate engineering-oriented projects. Students who are proud of their compilers use them in job interviews as evidence of their programming ability. No other course in our curriculum enjoys such wide respect among recruiters.

Another way to evaluate the changes to the course is to appreciate the new ways I can respond to student questions. Before the changes to the course, a question about design was often answered with the phrase "There are many ways to design a…," followed by a couple of approaches and a recommended solution. To many students this looked like a mystical process to which they did not have access. Now when a student asks a question, I can fall back on techniques taught in a previous lecture, walking the students through the prescribed process, with the student providing the answers at each step. The consequences are two-fold. First, students can solve the problem with only a little help, giving them the experience and confidence to solve the next problem by themselves. Second, the approach

reprises the course material, reinforcing learning and minimizing the introduction of (apparently) new concepts.

Although the techniques I've introduced have been streamlined to fit the particulars of the course, some compiler material had to be sacrificed initially (e.g., interpretation and garbage collection, usually covered at the end of the course). This material was restored when computer projection was introduced into my classroom, which increased the efficiency of my lecturing. At a small risk in student comprehension, another way to restore lost material would be to teach a portion of type checking and code generation the traditional way.

8.2 Related Approaches

Extreme programming (XP) is a software engineering method for rapid development in small teams [Beck 1999]. Some of the techniques in XP are applicable to the undergraduate software context, such as use of paired programming, to expose design and coding issues (it also aids learning), and writing unit test cases along with the unit. The method also advises that initial coding of a feature should focus on functionality rather than design, only refactoring once the feature has been fleshed out (the assumption is that the requirements are underspecified, so the initial coding helps define the feature). Our approach using a domain-specific specification as a way of structuring the design is similar, in that little up-front effort is invested in design; an awkward result leads to redesign, if necessary.

In the one paper found in the literature on software engineering and compilers, Liu [1993] describes the design of a compiler course with engineering sensibilities. A progressive project for a real language is used, and the focus is on the "lower" levels of the software engineering process (design, test, etc.), due to time constraints and the fact that, in any case, a proper software engineering course tends to focus more on higher-level issues. Otherwise, Liu's approach is qualitatively different. First, Liu requires use of the waterfall model of software development, with each phase producing standardized documents. Second, Liu assigns three students to a group (allowing two in special circumstances) and gives them defined (but rotating) roles: design, implement, and test. Third, project grading is substantially different. Only 30% of the project grade is based on the correct performance of the compiler, whereas the other 70% is based on the documentation. Components of the project can be regraded if corrected and turned in again. Moreover, 30% of the documentation grade is assigned according to an individual student's performance (presumably this is possible because of the defined roles).

In the context of the UCSD computer science curriculum, this approach is untenable for three reasons, thus motivating an alternative approach.

First, the waterfall model is generally not applicable to small projects, and the overhead to produce the documents would distract too much from a core goal of the class, i.e., learning to construct compilers. Also, I have found that students are all too willing to forgo learning how to construct a compiler if they can (a) trick their partner into writing it, or (b) earn their grade by doing some other kind of work. The focus on grading documentation rather than the compiler function gives students leeway to do a poor job of constructing compilers.

Second, individual grading of any sort on the project encourages students not to work cooperatively. Defining rotating roles for the students ensures they get equal exposure (and

grading) for the different roles, but ties a team's hands in choosing the work model that's best for them. If the project flounders, the students will most likely blame the instructor's work assignments and can (rightly) claim that their compiler grade is suffering (rather than improving) due to the software engineering material. Also, since students already juggle numerous responsibilities beyond this class, it is enough of a communications challenge for two (much less three) team members to stay in touch.

Finally, project regrading is an intriguing idea for increasing fairness and learning (and many commercial products get a second chance, too), but the students' tendency to code to the released test cases in the regrade phase would require graders to write a new set of test cases (an expensive prospect if the first set was any good). Perhaps limiting the number of recoverable points on a regrade could mitigate these difficulties.

9. APPLICATION TO OTHER SUBJECTS

This approach can be characterized generically as domain-formalism-directed software engineering, complemented by customized risk-reduction practices gleaned from experience in building systems. For example, in the area of operating systems, Dijkstra characterized the problem as providing each user with an efficient, tractable abstraction of a complex computer [Dijkstra 1968]. In particular, the abstraction presented to the user should be a sequential machine with an arbitrary amount of memory, even though the underlying machine had limited resources, parallel and interruptible execution with race conditions, and was shared by many people. Dijkstra prioritized the requirements of his operating system and then used step-wise refinement and layered design to incrementally abstract away the machine's most undesirable features first (starting with interrupts, which destroy our most basic notions of sequential execution) and then replace them with desirable ones.

Three properties of his approach are notable. First, each major requirement is met in a separate layer; hence, the solution is structured around the problem and is directly driven by the system requirements and operating systems theory. Second, to the extent allowed by feature interdependence, the most dangerous (risky) properties of the machine are abstracted away first; this risk-driven process increases the probable success of other features, since they do not have to cope with the dangerous properties. Third, the system is designed, implemented, and tested incrementally, and ready to run once the first system layer is completed. Today, of course, a microkernel design might be favored over a layered design, but the principles are still the same: organize components around the requirements (a paging component provides an infinite-memory abstraction); implement (only) the most critical features in the microkernel; and incrementally develop the system from the inside out, starting with the microkernel, so that the system is always able to run.

10. CONCLUSION

Teaching a computer science course with a significant software engineering project is complicated by the fact that students require expertise in both the topic of the course and in software engineering. Many computer science courses are taken before the one in software engineering. However, without guidance in software engineering practice, students may fail the project, hurting their grade and limiting their knowledge of the topic.

Introducing software engineering material into a course may help, but class time is

limited. A domain-specific risk-driven approach leads to practices that are natural for the topic area and provide more benefits than overhead. I have detailed this approach for a course in compiler construction, a classic project-driven course. For compilers, I employ a "syntax-directed" approach that involves designing lightweight software techniques organized through the grammar-based specification of the compiler. This approach not only eases construction of the compiler, but also reinforces the syntax-directed concept itself. Moreover, compiler construction provides numerous opportunities for reinforcing engineering lessons, such as the one on the importance of conceptual integrity in design, especially if the course is placed in an engineering context early on.

Producing such a course is costly. A text on compilers organized around the compiler's data abstractions rather than compiler phases, algorithms, and data structures, would be a significant aid. The notes for the class discussed here are available at the following site: http://www.cs.ucsd.edu/users/wgg/CSE131B.

As sketched in the previous section, this domain-oriented approach can be applied to other mature topic areas, serving to reinforce the lessons of (software) engineering and the concepts of the domain itself wherever possible in a curriculum.

ACKNOWLEDGMENTS

This paper was written while on sabbatical with the Aspect J group at Xerox PARC. I thank the numerous teaching assistants and students who have helped me develop this course, and also Brian Russ who shared teaching duties for 131B. I especially thank Darren Atkinson, whose dual expertise in compilers and software engineering proved invaluable. I also thank Kevin Sullivan and Michael Ernst for sharing their teaching experiences with me.

REFERENCES

BECK, K. 1999. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA.

BOEHM, B. W. 1988. A spiral model of software development and enhancement. *Computer 21,* 5 (May), 61–72.

BROOKS, F. P. 1975. *The Mythical Man Month: Essays on Software Engineering*. Addison-Wesley, Reading, MA.

DIJKSTRA, E. W. 1968. The structure of the "THE"-multiprogramming system. *Commun. ACM 11,* 5 (May), 341–346.

GIBBS, W. W. 1994. Software's chronic crisis. *Sci. Am. 271,* 3 (Sept.), 72–81.

JACKSON, M. A. 1975. *Principles of Program Design*. Academic Press.

JACKSON, M. A. 1976. Constructive methods of program design. In *Proceedings of the 1st Conference of European Cooperation in Informatics*. Springer-Verlag, 236–262.

JOHNSON, S. C. 1978. A portable compiler: Theory and practice. In *Proceedings of the 5th Symposium on Principles of Programming Languages*. 97–104.

LEVESON, N. G. AND TURNER, C. S. 1993. An investigation of the Therac-25 accidents. *IEEE Computer 26,* 7, 18–41.

LIU, H. 1993. Software engineering practice in an undergraduate compiler course. *IEEE Trans. Edu. 36,* 1 (Feb.), 104–107.

PARNAS, D. L. AND WEISS, D. M. 1985. Active design reviews: Principles and practices. In *Proceedings of the 8th International Conference on Software Engineering*. 132–136.

PETROSKI, H. 1994. *Design Paradigms: Case Histories of Error and Judgment in Engineering*. Cambridge University Press, Cambridge, UK.

Received June 2002; revised December 2002; accepted December 2002.