Teaching Software Engineering in a Compiler Project Course*

William G. Griswold
Department of Computer Science and Engineering, 0114
University of California, San Diego
La Jolla, CA 92093-0114
wgg@cs.ucsd.edu

Abstract

A compiler course with a term-long project is a staple of many undergraduate computer science curricula and often a cornerstone 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 material by integrating a few simple software engineering practices into the course. A domain-specific, risk-driven approach minimizes overhead and keeps the compiler material in focus, 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 a 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 approach of domain-specific risk-driven software engineering can be applied in courses such as operating systems by redesigning the practices around the its domain.

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 hope, at least in the desired order. Moreover, the software projects that professors would like to assign—and the 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 compilers 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 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 have taken the software engineering course and have 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 design and algorithms 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 was 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 limited techniques, 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. Finally, the engineering and compilers material is situated in the broader context of the software industry and society, motivating the subject matter and rationalizing the organization of the course. If the students should never take a software engineering course, students may have an incomplete software engineering education, but it is useful and hence appreciated. For impressionable students building their first complex system, appreciation imbued with experience and a few principles prepares their minds for further learning on their next system.

The remainder of this paper describes the original design of CSE 131B (which remains largely intact) and the problems that students encountered, details how software engineering was introduced into the course, explains why these particular choices were made, discusses the impact it has had on

*This paper was written while on sabbatical with the AOP group at Xerox PARC.

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

the course and the students, and compares my method to a previous approach. The paper concludes with a brief discussion about how software engineering might be introduced into 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 the final 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. The most demanding parts of the project come in 131B, in part because the formalisms for type checking and code generation are not as readily automatable. (The approach described in this paper is, in a sense, a method initially inspired by Johnson [6] for applying these formalisms through software engineering.)

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 construct a non-optimizing compiler for a useful subset of a "lean and clean" language is used. The language in current use is Oberon-2, as it embodies modern language concepts such as object orientation with few complex features. Even with simplifications, the resulting compiler will usually be on the order of 5,000 lines of code, virtually ensuring that 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 next, 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 will provide a compiler on request, albeit with no guarantees about its quality (it tends to be very good, but certainly not perfect).

Grading of 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 choice 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 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 the 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 the perspective) in software development. Beyond tools, knowledge about how to design a compiler is in a relatively advanced state, with plenty of guidance available in the form of software architectures (e.g., the pipeline design), symbol table design, type representation, and so forth. The project organization also provides valuable experience with development in a team and phased system development.

Although these lessons and experiences are a vital part of a computer science education, the students are relatively unprepared for them and the course as designed 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, and 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 a unit of modularization (but perhaps a fine unit of code ownership), and 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).

[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.

As a consequence, students often complained that the project was unfairly difficult (despite many quarters of simplification), yet trivialized the project as "just a lot of coding". On the other hand, the class material seemed straightforward to them, at least in comparison. Likewise, many students felt the project or its grading was unfair for a number of reasons. Many a student objected that his or her grade 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 to students who had done well. Others begged to be allowed to fix just one line of code after a phase deadline because their compiler crashed on a vast number of our test cases because of a "minor" oversight. Essentially, the students saw little connection to "compiler construction" and the causes of their troubles, and hence felt their grade did not reflect their knowledge.

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

- Getting integers to work was easy, but we couldn't get it to work for user-defined types.
- We ran the compiler on all your 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 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 deadline, and other factors to guide their design efforts. Such low-level problems later led to failures in communication, schedule, 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 project.
- *Scheduling.* Students were failing to allocate enough time for the project, presumably because of competing demands on their time, ignorance about compiler construction, and lack of experience in building a system that took more than a couple of weeks to build.
- *Teamwork and Communication.* Students were failing 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.

These are classic failure modes for commercial projects, but the underlying causes here are different, with a lack of background and wide-ranging commitments outside the class (e.g., other classes, a job) dominating.

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

4 Weaving Software Engineering into Compilers

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

The classic method for failure-avoidance in engineering is to use a risk-driven process [2, 9]. This approach has the added 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 "ir-

relevant" software engineering material whose application is unclear, thus increasing the chance that lifetime learning in software engineering is encouraged.

Each team's measure of success is operationalized as their grade. In terms that the students can readily understand, then, their primary risks are schedule and correctness: If the compiler is turned in late or is buggy, the team's grade will suffer. All other risks ultimately transmute into schedule or correctness risks. Meeting both schedule and correctness 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, development, 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 specification of static qualities (syntactic correctness itself and the integrity constraints) and dynamic qualities (the executable behavior).

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

5 Sensitization and Contextualization

Many students entering CSE 131B lack appreciation for 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 displacement in the marketplace by other products.

Of course, engineering is not only about profit, but social responsibility. I remind the students that software is everywhere, including life-critical systems such as nuclear powerplants (one operates only 30 miles from San Diego), airplanes, medical equipment, and banking.[3] Consequently, quality software is not an just a classroom topic, but a responsibility that they 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, as well as the design of a compiler, an instance 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 schedule being the overriding concern, any effort detracting from meeting the schedule is to be avoided. For example, although communication is a risk-factor for the busy students, an emphasis on documentation is unwarranted given the small size of the project. Communication risks are better overcome by clear separation of responsibilities and low barriers to communication (e.g., e-mail, meeting after class, and intuitive software interfaces).

Specification. Because of time-constraints, it is not feasible to teach students how to write specifications, especially since they are likely to have little experience, unlike with programming. The focus here is on the first step, learning to *use* a specification.

The students are given a syntax-based specification that is 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, semi-formal document is novel to the students compared to low-level interface specifications that they typically see in lower-division courses. Not being an interface specification, there are a number of potential ambiguities (a classic issue with specifications) that the students must appreciate and overcome. Moreover, despite years of refinement by two professors, students still uncover errors in it. (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 designs, it is too abstract for most college juniors and seniors. With a design space that is initially constrained only by the specification,

[3] Examples with detail are hard to come by, but a couple suffice [7, 5].

```
Expr -> Expr1 AddMulOp Expr2
AddMulOp -> - | + | *
   if type of Expr1 and Expr2 is numeric then
      type of Expr is smallest numeric type
            including the types of both operands
   else error

Expr -> Expr1 / Expr2
   if type of Expr2 or Expr2 are both numeric then
        type of Expr is REAL
   else error

Expr -> Expr1 MulOp Expr2
MulOp -> DIV | MOD
   if types of Expr1 and Expr2 are INTEGER then
        type of Expr is INTEGER
   else error
```

(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)

Figure 1: An excerpt from (a) the compiler specification for the type checking project and (b) the typing rules from the language specification.

students can become overwhelmed. Although the lectures and discussion sections spend ample time on design issues, we cannot speak to every detail. Moreover, our goal is to teach design, not dictate a design.

To help students make the transition from specifications to design, the students are encouraged to "code to the specification" in a syntax-directed fashion. In particular, the students are advised *first* to 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 they 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 the symbol table entries (and then type objects) are driven top-down from the specification, rather than considered in the abstract.[4]

[4]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.

```
  ...
  if (procobj->compatible(Expr1, Expr2))
    return procobj->resultType(Expr1, Expr2);
  else // error
    ...

bool ArithOp::compatible(STE *Expr1, STE *Expr2) {
  return (Expr1->isNumeric() && Expr2->isNumeric());
  }

Type *ArithOp::resultType(STE *Expr1, STE *Expr2) {
  return Expr1->type(); // extend for coercions
  }
```

Figure 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.

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. A class's design is determined bottom-up from the ways that its clients use it. The design is also, paradoxically, determined top-down from the specification, leading to the second benefit: coding to the interface means that the compiler's code structure and terminology will mirror the specification structure and terminology as much as 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 testing and debugging (what code is broken when a test does not work). This also minimizes the number of concepts the students have to keep track of, and yields a reasonable object-oriented design.

Although this approach is not the latest-and-greatest in software design methodologies, it 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 abandoning the approach and the benefits it offers. Examples come later in the course that show the need to design for change, not just intuitiveness.

Implementation. Most students, given a design, are reasonably good at implementing it. However, students have a tendency to write their entire compiler and then try to compile it. This approach maybe worked for them on smaller programs, but is a common cause of failure on the compiler project. Consequently, students are advised that their compiler should be "always runnable". As a consequence, their systems (or portions thereof) 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, simple implementations that quickly implement a subset of the functionality are possible. For example, the method body for `resultType` in Figure 2 works only for extact type matches, but can be extended later to handle coercion without disturbing the design. Consequently, compilation and testing can happen earlier in the design process.

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 far too little and haphazardly.

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, they should write a test case for each integrity constraint and behavior in the rule's specification, plus important combinations (See Figure 3). The students are advised that they should also do stress testing; that is, write big programs, weird programs, etc., to catch implementation errors. Finally, the students are reminded that testing finds bugs, it does not prove correctness, and hence testing requires pessimism about what testing has actually achieved.

approach provides three benefits. First, all the 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 between 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. Because of the project's short duration and focused functionality, the simplist scheduling methods are adequate. However, the short duration with strict deadlines dictates scheduling that tolerates feature omission in favor of meeting schedule. Consequently, the schedule is managed around milestones at which a key subset of functionality has been both implemented and tested. If a team falls behind, they can at least turn in a gradable subset of the system. The organization of the specification, design, and testing around the language grammar makes the project's status visible, hence 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 deadlines, it can be customized to their needs.

Consistent with this incremental approach, the simplist, basic features are scheduled in the first two milestones, and more advanced (and often dependent) features are scheduled in the last three. This not only gives the students something on which to build in the latter milestones, but also helps the students build some experience, confidence, and grade points early on. In fact, the students are told that a higher percentage of the grade is assigned to the basic features and advanced features, making it clear to the students that a subset of functionality is valuable, in particular basic functionality is more valuable than the advanced functionality.

```
(* RULE:
 *  Expr -> Expr1 AddMulOp Expr2
 *  AddMulOp -> - | + | *
 *    if type of Expr1 and Expr2 is numeric then
 *       type of Expr is smallest numeric type
 *          including the types of both operands
 *    else 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 (* assigns 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.
```

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

Team Management. Because of the small team size, *ad hoc* management techniques could almost suffice. However, students have historically shown bad judgment about divi-

sion of responsibilities, frequency of meetings, and the like. 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 themselves. For similar reasons, teams are encouraged to make important decisions by consensus rather than dictate.

The students are advised during sensitization that communication is a key risk because of the divergent responsibilities that they face beyond this one class. They are advised to work together on the project several times a week at a common location, minimizing communication lag and the problems that come with it. The students are also told that since the project is too large to be completed by one person, each member must contribute equally if the project is to succeed. Furthermore, because each team member possesses special skills, the teams full potential can only be realized by determining responsibilities according to those abilities. Students are then told of two complementary ways that labor can be divided effectively.

First, it is suggested that they each take responsibility for a portion of the grammar rules (i.e., the specification). The concreteness of this approach makes responsibilities clear: if that part of the language is not processed properly by the compiler, then the "owner" of those features is responsible. This division also has the benefit that each team member can concretely see the fruits of his or her labor. A downside of this approach 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 the "look over the shoulder" approach [1]. The labor time apparently lost is more than made up by discovering issues and catching errors early, as well as achieving crucial "buy in" by the entire team, preventing sore feelings and finger-pointing later.

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

Tools. Many students have not thought about configuration management, test harnesses, or system instrumentation, so the class TA prepares a set of online notes regarding the use of tools for their projects. The students are also advised to implement their own instrumentation facilities for dumping the symbol table and tracking key system events. Time is set aside in the first discussion section to review these tools and their importance.

7 Principles

Since the students have been given only one example of the simplist techniques, they are best backed 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 these principles in the small in earlier courses; reinforcement at this scale brings these principles to life in the realm of engineering. Their use 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 these risks, the project is much more likely to succeed [2, 9]. The students are periodically reminded that their overriding risks are schedule (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 duties can be parallelized by cleanly dividing activities, new software problems can be solved by 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.

Conceptual Integrity: Problems or entities that are similar should employ similar solutions, problems that are different should employ different solutions. Or from a more metaphysical, large-project perspective: a system should appear to be the product of a single coherent mind [3].

Young engineers are prone to two kinds of mistakes when encountering a new design problem: devising the "perfect" special-case solution or reusing an ill-fitting solution. 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. It can result in a system that is a mass of incompatible concepts, requiring extra intellectual effort to grasp an unfamiliar part of the system, and causing conflicts when new code is introduced that must work with more than one of these solutions. Reusing ill-fitting solutions costs much time in getting the old solution to work (students all-too-often dis-

cover this first-hand by trying to reuse a "dictionary" class from their data structures class for their symbol table), and can create the false appearance that an entity is less (or much 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 it solves. 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), but is stated separately because of its special importance to compiler construction. In particular, students are repeatedly shown how the nested structure of the programming language is reflected in a stack-based bottom-up parser, the symbol table, all the way to the assembly language. Without such intellectual control, implementing and debugging compilers would be virtually impossible. The students are also discover towards the end of the course 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, because all things being equal, simpler is better.

Application. Principles are effectively conveyed through examples. One situation that applies several principles arises in the handling of inherited attributes (program properties that are passed from earlier in the parse to later in the parse), 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 implementing inherited attributes in the context of the `exit` statement (like C's `break`). Students typically suggest using a global variable or a counter (to count loop nesting depth). After some discussion, I guide students to consider a solution that uses a stack (noted to be 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, failures with inherited attributes have been isolated to a single piece of code. The structural congruence of this solution is attractive because it makes the solution easy to understand in terms of the problem it is solving, and it 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 are struggling less with peculiarities of their teammates' code.

8 Discussion

Sensitization and contextualization places 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 the specific lessons of the course in a context that goes well beyond the confines of compiler construction. 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 essential class or project structure, introducing this approach was non-trivial in two ways. First, it requires a careful customization of software engineering practices to fit the domain. Second, this material must be integrated seamlessly into the class. Customization took quite some time, requiring two rewrites of my notes, although this is in part because I didn't 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 the students. Although many examples are borrowed from the text, working out the software engineering implications requires additional work. On the other hand, the notes are very popular, as the text is viewed by most students to be unnecessarily complex. Readings are still assigned from the text, although most students seems to use the book as a reference only, preferring to read my notes.

Evaluation

Quantitative assessment of trends in this context is fraught with difficulty. This approach was introduced gradually over seven years in response to difficulties as soon as they became apparent. In that time, the computer science student body changed significantly. Growing interest in computer science has dramatically increased class sizes, and many of these students are now in computer science because it is a good career move rather than an intellectual interest. The majority of these students now hold demanding programming jobs, whereas before students were likelier to take out loans or work on campus. Efforts to stop cheating increased dramatically in recent years, providing another dimension of unpredictability. Together, these have fundamentally changed the teaching landscape. Year-to-year variability in the course itself such as the particular teaching assistants adds another level of unpredictability.

Anecdotal evidence suggests that some students have been dramatically affected. Some students will always take to one topic while disliking another. My focus on the two aspects

of the project—compilers and engineering—has widened the audience that is interested in the course, 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 speaking to me outside of class are positive about the engineering material, while written comments in course evaluations tend to be more balanced by invective. Several times students have commented that the "compiler wrote itself" due to the recommendations about how to approach the design. Other students have expressed pride over the quality of their test suites and their (well-placed) confidence about the quality of their compiler. Except for the most successful students, there seems to be wide agreement that the project is still too difficult, despite the fact that the trend has been to simplify the project over time. I believe that this attitude in large part can be attributed to the fact that 131B is generally the first such course that students encounter, representing both a change in expectations as the students move into the upper division and a steep learning curve.

Job recruiters, however, have a more uniform perspective: performance in the course is a clear indicator of a student's ability to succeed at their company. They frequently ask for more courses to introduce engineering-oriented projects. No other course in our curriculum enjoys such wide respect among recruiters.

Another way to appreciate the changes to the course is how I can now handle questions. Before the changes to the course, a question on design was often answered by the comment "there are many ways to design a..." followed by a couple of approaches and a recommended solution. To many students this looked like a magical process to which they did not have access. Now when a student asks a question, I can fall back on techniques I had taught in a previous lecture, having the student walk through the prescribed process, with myself only providing guidance. The consequences are twofold. First, the student can solve the problem with only a little help, giving her the experience and confidence to solve the next problem by herself. Second, the approach repeats the course material, minimizing the introduction of (apparently) new concepts and reinforcing learning.

Related Approaches

In the one paper found in the literature on software engineering and compilers, Liu describes the design of a compiler course with engineering sensibilities [8]. 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 the time constraints and the fact that a proper software engineering course tends to focus more on higher-level issues anyway. Otherwise, Liu's approach is fundamentally 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.

For one, the waterfall model is generally not applicable to small projects and the overhead in producing the documents would distract too much from a core function of the class, learning 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's function gives students the leeway to do a poor job of constructing their compiler.

Two, individual grading of any sort on the project encourages students to not work cooperatively. Defining rotating roles for the students does ensure they get equal exposure (and grading) for the different roles, but ties a team's hands in choosing the work model that works best for them. If the project flounders, the students are positioned to blame the professor's work assignments and can (fairly) claim that their compiler grade is suffering (rather than improving) because of the software engineering material. Also, with students juggling numerous responsibilities beyond this class, two team members is enough of a communications challenge.

Finally, project regrades are an intriguing idea at increasing fairness and learning (and many commercial products get a second chance), but the tendency of students to code to the released test cases in the regrade phase would require the graders to write a new set of test cases, an expensive prospect if the first set was any good. Perhaps employing three-student groups and limiting the number of recoverable points on a regrade could mitigate these issues.

Extreme Programming (XP) is an industrial software engineering method for rapid development in small teams [1]. Some of the techniques in XP are applicable to the undergraduate software project context, such as the use of paired programming to avoid low-level design and coding mistakes (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 in rapid development, so the initial coding helps define the feature). Our recommendation to use the domain-specific specifica-

tion 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.

9 Application to Other Subjects

Generically, this approach can be characterized as domain-formalism-directed software engineering, complemented with customized risk-reduction practices gleaned from experience in building systems in the domain. 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 [4]. 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 interruptable execution with races, and was shared by many people. Dijkstra prioritized the requirements of the operating system he would like to provide, 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 replace them with desirable ones.

Three properties of his approach are notable. First, each major requirement was achieved in a separate layer; hence, the solution is structured around the problem and is driven directly from the system requirements and the theory of operating systems. Two, to the extent allowed by the interdependence of features, the most dangerous (risky) properties of the machine were abstracted away first; this risk-driven process increased the probable success of other features because they did not have to cope with this undesirable machine property. Third, the system was incrementally designed, implemented, and tested, and was runnable upon the completion of the first system layer. Today, of course, a microkernel design might be favored over a layered design, but the principles would be the same: organize components around the requirements (a paging component would provide 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 runnable.

10 Conclusion

Teaching a computer science topics course with a significant project is complicated by the fact that students require expertise in both the topic area and software engineering. Many such courses are taken before a software engineering course. Without guidance on software engineering practices, students may fail on the project, hurting their grade and limiting their learning of the topic material.

Introducing software engineering material into such a course can help, but class time is limited. A domain-specific risk-driven approach leads to practices that are natural to the topic and provide more benefits than overhead. I have detailed this approach for a course in compiler construction, a classic project-driven topics course. For compilers, I employ a "syntax-directed" approach that involves designing lightweight software techniques driven organized through the grammar-based specification of the compiler. Moreover, compiler construction provides numerous opportunities for reinforcing engineering lessons such as the importance of conceptual integrity in design, especially if the course is placed in an engineering context early in the course.

Producing such a course is costly. (My notes are available from `www.cs.ucsd.edu/users/wgg/CSE131B`.) A compilers text that was organized around the data abstractions of a compiler, rather than the compiler phases, algorithms, and data structures, would be a significant aid.

As sketched in the previous section, this approach can be applied to other mature to other mature topics, reinforcing the lessons of (software) engineering wherever possible in a curriculum.

Acknowledgments. I thank the numerous TA's and bright students who have helped me develop this course, and Brian Russ who has shared teaching duties on 131B. I especially thank Darren Atkinson, whose dual expertise in compilers and engineering proved invaluable. I also thank Kevin Sullivan and Michael Ernst for sharing their teaching experiences with me.

REFERENCES

[1] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA, 1999.

[2] B. W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, May 1988.

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

[4] E. W. Dijkstra. The structure of the "THE"-multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[5] W. W. Gibbs. Software's chronic crisis. *Scientific American*, 271(3):72–81, September 1994.

[6] S. C. Johnson. A portable compiler: Theory and practice. In *Proceedings of the 5th Symposium on Principles of Programming Languages*, pages 97–104, January 1978.

[7] N. G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[8] H. Liu. Software engineering practice in an undergraduate compiler course. *IEEE Transactions on Education*, 36(1):104–107, February 1993.

[9] H. Petroski. *Design Paradigms: Case Histories of Error and Judgment in Engineering*. Cambridge University Press, Cambridge, England, 1994.