Teaching Students to Understand Large Programs by Understanding Historical Context

Collin McMillan[1] and Richard Oosterhoff[2]
[1]Department of Computer Science and Engineering
[2]Department of History
University of Notre Dame
Notre Dame, IN, USA
{cmc, rooster1}@nd.edu

ABSTRACT

Program comprehension is one of the most important challenges that new software developers face. Educators have sought to prepare students for this challenge through hands-on software development projects. These projects teach students effective software engineering principles. But, students often struggle to see the value of these principles in class projects, and therefore struggle to recognize them outside the classroom. The inevitable result is that these students have difficulty comprehending large programs after graduation. In this paper, we argue that a remedy to this problem is to teach the history of how software development principles were created. In this collaborative work with the Notre Dame Department of History, we present a course that blends a discussion of this history with a hands-on software project. We present a summary of the history covered in our course, and reflect on our teaching experience.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering

General Terms

Education

Keywords

software engineering education; history

1. INTRODUCTION

A key problem software engineering practitioners face is *program comprehension*: the task of understanding the behavior of large software projects [23, 21, 35]. Novices struggle much more than experts with program comprehension. Two independent studies at Microsoft have found that this process is one of the most important challenges that new college graduates face [9, 5].

One solution to this problem is to give students "hands-on" experience with large programs in a classroom setting [2, 4]. This experience may come in different forms, for example as a multi-semester project with different students each semester [32], or as contributions to open-source software [27]. These large-scale assignments help students to understand real world problems and avoid some of the difficulties they would otherwise encounter after graduation.

What is missing from these hands-on projects is guidance to students on *how* to understand large programs. Techniques for program comprehension are very difficult to impart, even with hands-on projects and close mentoring [22, 32, 8]. Begel *et al.*'s Microsoft study [5] found that new software developers often do not know "when they do not know" how a program works: fledgling programmers struggled to understand large programs, even when an experienced developer demonstrated an appropriate program comprehension technique. Some new programmers even voiced concerns that their "mental model [about the program] was wrong", but did not know how to develop a correct mental model. The result is that, even in a classroom setting, novice programmers learn through trial and error, which leaves novices' understanding "fragmented and piecemeal" [5].

At the core of this problem is that novices do not understand the *rationale* behind the source code that they read, even if they understand what the source code does. Novices struggle to connect the low-level implementation details they read to the high-level software engineering principles they learn in school. Different studies of programmers in cognitive psychology refer to this struggle as a problem with the "assimilation process" [37]: novice programmers have difficulty understanding how the principles they learn affect the source code they read and write.

In this paper, we argue that a remedy to this problem is to teach students the history of how software development principles were created. The crux of our argument is that historical techniques for understanding problems "are considered integral parts of scientific literacy" [17] in disciplines ranging from mathematics [36] to chemistry [38]. But, this history is often overlooked in software engineering [3]. Our view is that students would learn to read and comprehend programs more quickly if they *first* learn the historical context behind the implementation of those programs.

In this collaborative work with the University of Notre Dame's Department of History, we propose a software engineering course that encourages students to enter the historical debates behind prominent software engineering principles, while illustrating those principles in hands-on class projects. We provide a brief history of the principles covered in our course, and reflect on our teaching experience.

ICSE '14 Hyderabad, Andhra Pradesh, India

2. BACKGROUND / RELATED WORK

The most common approach to helping students understand large programs is through hands-on software engineering projects. Rajlich proposes teaching incremental changes to a pre-existing program [32]. The idea is that students will make changes while understanding a small piece of the program at a time. Marmorstein has proposed a similar course in which students contribute to open source software [27]. This course takes advantage of open source software that is designed so novice programmers may make useful contributions through bug fixes and feature enhancements. The advantage of these hands-on projects is that students practice program comprehension alongside other software engineering tasks (e.g., code reviews and version control) [4, 19].

Research in different disciplines has recommended teaching students the rationale behind the principles of those disciplines. The National Council of Teachers of Mathematics calls this a "problem solving" focus to teaching [29]. In science education generally, the creation of physical models has long been taught to improve understanding of the rationale behind scientific principles [7, 17]. Mathieu *et al.* suggest that shared understanding of this rationale, in the form of mental models, is related to effective team processes [28]. Rouse *et al.* point out that the construction of correct mental models is a teaching goal in numerous domains [34], a finding supported in software engineering literature [24]. Nevertheless, despite being proposed in computer science [3], software engineering education often overlooks the rationale and history of software engineering principles.

3. OUR SOLUTION

In this section, we discuss our design for a software engineering course, CSE40232 in the Department of Computer Science and Engineering at the University of Notre Dame.

3.1 Key Idea

Our key idea is to teach principles of software development within the framework of historical thought behind those principles. Many principles of software development are not grounded in physical laws; they are the result of decades of debate, opinion, and empirical evaluation. Crucially, these principles reflect the findings of research into how programmers understand software. A principle may be considered "useful" if there is evidence that it helps programmers form a correct mental model of the software. For example, different researchers have found that programmers tend to read only the code relevant to a given task [33], which is why the principle of concern separation is "useful" to programmers. It is also why concern separation is a motivation behind metrics such as coupling and cohesion. Our view is that students should learn this history of, and philosophy behind, software development principles so that the students will understand 1) the reasons to use those principles, and 2) how to comprehend programs written using those principles.

3.2 Details of Educational Plan

Our course design has four components: 1) in-class debates about software development principles, 2) a group project where software is developed from scratch, 3) an individual case study project where each students adds a feature to a complex open source project, and 4) instructor interviews of students about software development. Next, we describe the purpose behind and details of these components.

3.2.1 Debates

We host a series of debates about different software development principles. These debates place one student on each side of a controversial topic. Before each debate, the instructor presents the topic in two lectures. The first of these lectures covers the history of the principle (we provide an outline of this history in Section 4). The next explains the principle in detail, emphasizing examples of programs where the topic is implemented. Two students are then assigned to debate the merits of the principle. Each student must research the prominent opinions and empirical evidence for or against it, including the relevant academic literature. The students are also asked to write a report summarizing their evidence. Two weeks after the instructor lectures on the topic, the students present their arguments in an in-class debate. The students cross-examine each other, must answer questions from the classroom, and respond to counter-arguments.

These debates encourage students to think critically about each of the principles. The goal is for the students to form opinions about these principles, and to understand the opinions of others. At the same time, the students must support their opinions through academic literature. Our aim is that, by forming opinions about these topics, the students will learn to recognize the topics in software that the students read. If the students recognize how the principles are implemented in software, then the students will be prepared to comprehend that software. The students will learn to build a mental model of the software based on the principles of software development.

3.2.2 Large Group Project

One component of the class is a large group project involving every member of the class (13 students). The instructor takes the position as team leader, and demonstrates how the principles learned through the debate process can be applied to a software development project. Accepted practices for version control, requirements elicitation, software design, documentation, and maintenance are covered through demonstration and practice. The instructor dedicates one lecture period per week as an "all hands" meeting, where tasks for the following week are prioritized and assigned. The students collaborate via an issue control system. Throughout development, students are asked to reflect on the effectiveness of different software development topics, such as design patterns and software metrics, so that the students learn to recognize these topics in practice.

3.2.3 Case Study Project

In a case study project, we ask each student to implement a change to an open-source software program. This project is intended to give each student experience in reading and comprehending a large (>100 KLOC) program. Collaboration is possible only at a high-level: students may communicate with each other about how the program works, but may not read source code written by other students. This style of collaboration is intended to help students learn to explain source code; through explanation, the students will clarify their own mental models of the code. Through the lecture and debate series, our intent is for the students to connect concepts from this case study project to concepts discussed in class. The idea is that the students will learn how the software development principles help to form correct mental models about the case study project's behaviors.

3.2.4 Individual Interviews

In lieu of a final exam, the instructor will conduct a one-hour individual interview with each student. The instructor asks questions related to the group and case study projects, and elicits opinions about the concepts discussed during the debate series. For example, "did our implementation of design patterns help solve the problem we had formatting the display for our group project?" The instructor evaluates the student's understanding of both the concept and the project through these questions. The primary goal is for the student to practice communicating ideas about program design, but a secondary benefit is job interview practice.

4. BRIEF HISTORY OF S.E. PRINCIPLES

This section highlights the movements behind the growth of software engineering principles that we debate in our course. Most of the history of software engineering remains to be written [10, 11, 26], and our history is not intended to be comprehensive. Rather, our goal is to equip students to understand software by understanding why earlier programmers adopted certain principles to create that software.

A *software engineering principle* is any principle that programmers follow to write software (e.g., separation of concerns, modularity, abstraction). Programmers found two features especially important in developing these principles. First, accepted principles should lead to correct behavior in programs. Second, the principles should lead to programs in which the behavior can be easily understood. These two factors were famously explained by Edsger Dijkstra in his 1968 letter "Go To Statement Considered Harmful" [15]. While focusing on a specific programming language feature, Dijkstra made the broader point that programs should be written to mirror the way in which human programmers understand software. In this view, software engineering principles should lead to software that can be intuitively read and understood.

Software engineering principles are often controversial because of debate over how programmers read and understand software. Early computer scientists turned to different cognitive models to explain how programs are understood [37], notably with different evidence on whether program comprehension is primarily an inductive or deductive thought process. The Logo programming language, released in the 1960s, was partially inspired by Jean Piaget's studies of inductive learning processes in children [30]. The argument was that, according to Piaget, humans learn by connecting their actions to concrete, real-world outcomes; therefore, programs should operate by executing simple actions on well-defined program units. Logo illustrated this idea for children's learning: an onscreen "turtle" moved based on easy-to-understand actions such as "forward." Studies have shown limits to this inductive learning technique [22], and the ideas behind the technique remain controversial.

The debate on programmer understanding persisted in the debate on object-oriented programming in part because, from the 1970s on, programmers were less and less constrained by hardware. Smalltalk, release in 1972 [25], epitomized the idea of connecting well-defined actions and program units. All values in Smalltalk are "objects." Programmers define these objects and the actions they perform. The structure of the program is modeled as the interaction between these different objects. Historically speaking, what is significant about this object-oriented design is that it has become a methodology for creating programs distinct from procedural techniques, with separate theories to describe object-oriented program behavior [1]. In other words, object-oriented designs represent a different strategy for understanding software, requiring different mental models [14]. Indeed, the arguments for object-oriented designs center on the benefits of objects in design recovery [6], while arguments against these designs claim that objects obscure the procedural control flow of the program [31].

Software engineering principles became more established, and by the 1990s researchers had invented metrics to measure how well a program adhered to these principles. These metrics came to be known as "software quality" metrics [20]. Software quality metrics such as cohesion, coupling, and cyclomatic complexity reward code that is modular or functions that are short and simple [20]. In general, the assumption behind these metrics is that programs that score well on these metrics are easier to understand, and therefore will contain fewer bugs. But this assumption is controversial, partially because since the early 1990s metrics are often associated exclusively with object-oriented designs [12], and partially because empirical evidence does not strongly support or refute metrics [18]. At the same time, design patterns in object-oriented software were proposed as strategies for understanding and reusing solutions to common programming problems [16]. Design patterns rely on software engineering principles being widely-accepted, and are therefore controversial for the same reasons as quality metrics.

Debate over program understanding also affects software development processes. The Agile development process was proposed in 2001 as a process that more-closely matched programmer comprehension of software than previous plan-driven processes [13]. In the view of Agile developers, programmers understand software largely by communicating with other programmers [33], and therefore the development process should reflect this communication. As a result, Agile methods remain a subject of intense debate.

5. CLASSROOM EXPERIENCES

In this section, we reflect on our classroom experiences and present recommendations for implementing our course design. At the time of writing, the course has completed eight of 14 weeks. The students have largely responded well to the historical perspectives discussed via the debates. However, we have also found that the students were unaccustomed to debating concepts "in a science class", as one student mentioned, and hesitated to present opinions. Our strategy for encouraging the students has been to present arguments from academic literature supporting *and* refuting different topics. We have also found that the students respond much more strongly when the instructor assigns each student a side to debate for a topic, rather than allowing students to choose. The students were not required to defend a personal opinion, but instead the opinion of established researchers or practitioners.

Grading was another area of confusion for some students. As the large group project is semester-long and involves all students, the instructor assigns a project grade at the end of the class based on each student's individual performance. There is a risk that this grade will seem arbitrary or unexpected. Our approach to limiting this risk has been to provide biweekly status reports to each student. These reports describe areas of strength for each student and specific feedback on how to improve. Each report also includes

a non-binding letter grade representing what grade the student would receive, if the semester were complete now.

We have had to accept a flexible schedule in the course. For the large project, the class is implementing a multiplayer 3D game. The students must coordinate different programming challenges, such as networking and GL graphics, while practicing concepts discussed in class. For example, the students would practice writing design patterns to solve a problem in the project. Meanwhile some students may slow the entire group if he or she does not complete an expected section. As a result, the project schedule is not entirely predictable. We may debate design patterns on schedule, but practice implementing much later. We have found that the biweekly reports are an invaluable tool for communicating the instructor's expectations to the students, and to reduce anxiety of scheduling conflicts.

Our perception has been that the students have become more able to recognize the value of the software engineering concepts taught in our course. One observation has been that the students have asked increasingly insightful questions during job interviews. Seven of the 13 students have, on their own initiative, met with the instructor to discuss upcoming and past interviews. The students reported asking key questions such as why the company chose an Agile development process, or how the company evaluates source code quality. The students report feeling comfortable conversing with interviewers about advanced software engineering topics. Some students directly credited the historical context we provided with this increased confidence and understanding.

Our view is that the historical context will ultimately benefit the students in understanding large programs. We have three strategies to evaluate our view. In the short term, we will conduct individual interviews with students and grade each student based on their ability to answer questions about how a hypothetical large program *should* be designed (see Section 3.2.4). Also, we will grade case study projects in which students must understand and alter a large program. If our view is correct, in the case study projects we would expect to see the students achieve higher grades with less reported effort compared to classes where our historical perspective is not explained. In the long term, we will conduct workplace surveys after our students enter industry, to compare the difficulties they report to existing studies of new programmers (e.g., [9, 5]).

6. CONCLUSION

We have argued that software engineering curricula should include the history of software engineering principles in order to promote student understanding of large software programs. We have pointed to a need in industry for improved education of program comprehension, and have presented the design of a class that targets this need by helping students understand the rationale behind software design through historical context. We presented an overview of this history to guide other educators, and reflected on our classroom experiences with our course.

7. REFERENCES

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, Secaucus, NJ, USA, 1996.

[2] B. Adelson and E. Soloway. The role of domain expenence in software design. *IEEE TSE*, 11(11):1351–1360, Nov. 1985.

[3] A. Akera and W. Aspray. *Using History to Teach Computer Science and Related Disciplines*. CRA, 2004.

[4] E. Allen, R. Cartwright, and C. Reis. Production programming in the classroom. In *Proc. of SIGCSE*, pages 89–93, 2003.

[5] A. Begel and B. Simon. Struggles of new college graduates in their first software development job. In *SIGCSE'08*.

[6] T. J. Biggerstaff. Design recovery for maintenance and reuse. *Computer*, 22(7):36–49, July 1989.

[7] M. Black. *Models and Metaphors: Studies in Language and Philosophy*. Cornell University Press, 1962.

[8] E. Brannock and N. Napier. Real-world testing: using foss for software development courses. In *SIGITE'12*, pages 87–88.

[9] E. Brechner. Things they would not teach me of in college: what microsoft developers learn later. In *OOPSLA'03*.

[10] M. Campbell-Kelly. *From Airline Reservations to Sonic the Hedgehog (History of Computing Series): A History of the Software Industry*. MIT Press, Cambridge, MA, USA, 2003.

[11] M. Campbell-Kelly, W. Aspray, N. Ensmenger, and J. R. Yost. *Computer: A History of the Information Machine*. 2013.

[12] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE TSE*, 20(6):476–493, June 1994.

[13] A. Cockburn. *Agile software development*. 2002.

[14] U. Dekel and J. D. Herbsleb. Notation and representation in collaborative object-oriented design: an observational study. *SIGPLAN Not.*, 42(10):261–280, Oct. 2007.

[15] E. W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, Mar. 1968.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, Boston, MA, USA, 1995.

[17] J. D. Gobert and B. C. Buckley. Introduction to model-based teaching and learning in science education. *IJSE*, 2000.

[18] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE TSE*, 31(10):897–910, Oct. 2005.

[19] A. Hunt and D. Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley, 1999.

[20] S. H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley, 2002.

[21] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE TSE*, 32(12):971–987, Dec. 2006.

[22] D. M. Kurland and R. D. Pea. Children's mental models of recursive logo programs. *Edu. Comp. Res.*, 1(2):235, 1985.

[23] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *ICSE'06*.

[24] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. *J. Syst. Softw.*, 7(4), 1987.

[25] C. Liu. *Smalltalk, Objects, and Design*. iUniverse, 2000.

[26] M. S. Mahoney. What makes the history of software hard. *Annals of the History of Computing, IEEE*, 30(3):8–18, 2008.

[27] R. Marmorstein. Open source contribution as an effective software engineering class project. In *ITiCSE'11*.

[28] J. E. Mathieu, T. S. Heffner, G. F. Goodwin, E. Salas, and J. A. Cannon-Bowers. The influence of shared mental models on team process and performance. *Journal of App. Psy.*, 2000.

[29] NCTM. *Agenda for Action*. NCTM, 1980.

[30] S. Papert. *Mindstorms: children, computers, and powerful ideas*. Basic Books, Inc., New York, NY, USA, 1980.

[31] C. Ponder and B. Bush. Polymorphism considered harmful. *SIGSOFT Softw. Eng. Notes*, 19(2):35–37, Apr. 1994.

[32] V. Rajlich. Teaching developer skills in the first software engineering course. In *Proc. of ICSE*, pages 1109–1116, 2013.

[33] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How do professional developers comprehend software? In *ICSE'12*.

[34] W. Rouse and N. Morris. On looking into the black box: prospects and limits in the search for mental models. *Psychological Bulletin*, 100(3):349–363, 1986.

[35] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *CASCON'07*.

[36] C. Tzanakis, A. Arcavi, C. Sa, M. Isoda, C.-K. Lit, M. Niss, J. Carvalho, M. Rodriguez, and M.-K. Siu. Integrating history of mathematics in the classroom: an analytic survey. *History in Mathematics Education*, 6:201–240, 2002.

[37] A. von Mayrhauser and A. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[38] V. M. Williamson and M. R. Abraham. The effects of computer animation on the particulate mental models of college chemistry students. *Journal of Res. in Sci. Tea.*, 32(5), 1995.