Teaching Statement

Scott D. Fleming

As a teacher of software engineering (SE), I am passionate about preparing graduates to begin practicing software engineering professionally. The expected outcomes for SE students go far beyond the ability to program software [1]. Graduates must master the key SE principles, such as modularity and abstraction, as well as the heuristic methods and best practices, such as software design patterns, which support those principles. Moreover, they must acquire the practical skills required for success in a real-world development environment. Such skills include the ability to work effectively in teams, to perform software configuration management, to manage large-scale code bases that depend on third-party libraries, to cope with conflicting and changing requirements, and to communicate effectively with stakeholders.

In my two years as a full-time computer-science instructor and three years as a teaching assistant, I have identified several problems in SE education. Firstly, students are often able to memorize SE methods; however, they have difficulty applying those methods. Secondly, they fail to acquire many practical skills, such as proficiency with version-control and build systems. Lastly, they have difficulty comprehending and retaining material presented in the classroom. I intend to address these challenges by emphasizing experiential learning, by designing course projects that simulate realistic development environments, and by supporting classroom instruction with pedagogical theories, such as active learning.

Applying Experiential Learning to Software Engineering Education

SE education emphasizes the teaching of principles, best practices, and heuristic methods for building large software systems. Students consistently demonstrate the ability to memorize and regurgitate these abstract concepts; however, they have difficulty applying the concepts to concrete problems. At Michigan State University, we use a novel experiential-learning approach [2] that has proven effective for teaching students how and when to apply SE principles and methods in practice. Following the approach, we introduce students to concepts via concrete motivating examples and present more abstract views of the concepts over time. For instance, we typically address a problem in lecture, then motivate it in a course project, and finally introduce a design pattern, strategy, or method that generalizes the solution.

The novelty of the approach is that we present the material in the reverse of the order commonly used. Typically, SE courses are organized around the waterfall model of the software lifecycle. They cover each phase of the lifecycle in sequence—for example, analysis, followed by design, followed by implementation. However, the purpose of many SE principles and methods, which are applied early in the development process, is to avert problems in later phases. Organizing material around the waterfall model, students lack the experiences necessary to motivate the SE principles and methods that they are expected to apply. Our approach covers the SE body of knowledge by starting with implementation and working backwards toward analysis.†

† We refer to the approach as bottom up because it works from lower levels of abstraction (e.g., implementation) to higher levels (e.g., analysis).
With our approach, students demonstrate improved ability to apply SE principles and methods to new situations. By first giving students experiences that epitomize abstract concepts, they are better able to understand the applicability of the concepts. By working backwards through the development process, we are able to motivate methods whose benefits are only realized during later phases of the process. Given our success with the approach, I intend to continue refining it by developing new materials, such as sample programs and libraries, and by incorporating innovative teaching techniques, such as collaborative learning.

Covering Practical Skills

Preparation for real-world software development environments requires learning practical skills, such as configuration management and working with third-party libraries. Covering many of these practical skills in the context of a college course is difficult. SE principles and methods are conceptually rich and require extensive coverage in lecture. However, the practical skills tend to be rote in nature and warrant little or no coverage. Mastering such skills requires thorough practice. So then, the problem is how to integrate such practice into a course without expending valuable class time.

I address this problem by infusing elements of realism into course projects. Such elements provide students with practice in practical aspects of software development. In essence, my goal is to produce realistic simulations of real-world development environments. My teaching and research dovetails with respect to this goal. In my empirical research, I aim to produce realistic simulations for the purpose of studying developers in a laboratory setting (e.g., [3]). The lessons learned and artifacts produced by developing simulations in one context (e.g., research) naturally transfer to the other (e.g., teaching).

For example, I designed a course project for a junior-level software-design course at Michigan State University. The project involves extending the functionality of a multithreaded GUI browser application. I incorporated several elements of realism into the project. Firstly, the browser is small (1695 SLOC) but has a realistic model-view-controller architecture [4]. Secondly, students must program against three third-party libraries, FLTK\(^2\), ACE\(^3\), and a library of reusable abstractions specifically created for the course. Lastly, students are required to use a version-control system, CVS\(^4\), to check out the source code and commit incremental changes. These elements require minimal class time to explain and provide students with valuable practice for the real world.

Applying Pedagogical Theories in the Classroom

I endeavor to provide top-notch instruction to students. To this end, I continually look to the literature on education for new classroom strategies and techniques. For example, two pedagogical theories that I incorporate in my teaching are active learning and individual learning styles.

Active learning emphasizes actively engaging students in course material [5]. Studies consistently show that passively listening to traditional didactic lectures is not conducive to student learning [5]. Students’ ability to assimilate knowledge drops off sharply after roughly 20 minutes of continuous lecturing. To implement active learning, I break up my lectures into 20-minute

---

\(^2\)http://www.fltk.org  
\(^3\)http://www.cs.wustl.edu/~schmidt/ACE.html  
\(^4\)http://www.nongnu.org/cvs/
mini-lectures, interspersed with short (e.g., 5–10 minutes) activities. The types of activities include reading, writing, problem solving, and discussion. For example, I frequently use Think-Pair-Share activities [6]. During one mini-lecture, I introduce a problem, such as how to make a buffer thread-safe. Next, I ask students to think silently about a solution. Then, I ask them to pair up with a partner to compare and discuss their solutions. Finally, I randomly select a student to describe her solution or to summarize her discussion.\(^5\)

I am also sensitive to individual differences in the learning styles that students favor. For example, some students are visual learners, which means they prefer to work with information in the form of pictures and text enhanced with color and layout [7]. Other students are aural learners, which means they prefer to process information verbally, for example, by listening or engaging in conversation. Although no student is restricted to just one learning style, students may have strong preferences toward some styles and struggle with others. Given such individual differences, I try to present information in as wide a variety of styles as possible. For example, in my lectures, I augment my aural presentations with visual aids, such as UML diagrams, graphs, and animations. Moreover, in-class activities provide an excellent opportunity to support a variety of learning styles, such as those that favor conversation or reading/writing. Being sensitive to variations in learning style helps ensure a learning environment that is fair for all learners and improves student performance in the aggregate.

**Engaging Students in Research**

I firmly believe in the value of engaging students in research. Students benefit from such involvement by being exposed to new methods and technologies, by learning about how research is conducted in general, and by gaining firsthand insight into a potential career path in research. The research community benefits by promoting the value of SE research to future SE professionals, by increasing access to qualified study participants, and by recruiting new researchers.

I engage students in my research by inviting them to participate in studies (e.g., [3]). Students make excellent subjects of study. They represent an important subpopulation of software engineers, and recruiting students for pilot studies is convenient and cost effective. In addition to compensating students fairly for their participation in studies, they benefit by being introduced to new SE concepts and technologies, or by getting additional practice with familiar concepts. Furthermore, students receive a debriefing in which they learn about the design and findings of the study. Students can also serve as assistants who help conduct research studies. The NSF Research Experiences for Undergraduates (REU) program provides supplemental funding for on-going NSF grants to pay such assistants.

**Course Preferences**

I look forward to teaching graduate-level courses in software engineering, formal methods, empirical methods, and distributed systems, as well as seminar courses on special topics, such as concurrent programming models and empirical software engineering. I am qualified to teach most undergraduate computer-science courses and have a preference toward software engineering, empirical methods, distributed systems, and compilers.

\(^5\) Random selection ensures that students are individually accountable for participating.
References


