# Software Engineering I: Teaching Challenges

*Paul C. Grabow, Computer Science Department,*
*Baylor University, Waco, TX 76798, USA, Paul_Grabow@baylor.edu*

**Abstract**

The term software engineering can be traced to the late 1960's in response to large-scale, software development problems. Since then it has evolved as a discipline, both within industry and the academy. There have been distinct educational successes: "Standard practice" has matured (and found its way into more textbooks), the ACM and IEEE Computer Society have published curriculum guidelines, computer science programs commonly offer at least one software engineering course, and software engineering degrees (undergraduate or graduate) are more common. However, software engineering still presents a challenge. The term itself has no consensus definition; software applications and development environments are significantly varied; and development practices are often unique to their problem domain. This paper describes these challenges and some possible responses, with some thoughts related to an existing Software Engineering I course.

## Introduction

Here we assume that we are dealing with a single, introductory Software Engineering I course (that is not necessarily part of a software engineering degree). Since the term software engineering has multiple meanings, we will also assume the IEEE definition:

> "The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software." [1]

Also, we assume that an undergraduate degree in software engineering is defined by the Accreditation Board for Engineering and Technology (ABET) [17], which has accredited only twenty-one programs (See [20] that describes the University of Rochester's program.) An accredited software engineering degree should not be confused with what the U.S. Department of Education's calls a "degree in computer software engineering" [3], of which 106 are listed on their website (including some two-year degrees and some for-profit organizations).

## Challenges

### 1. Terminology

The term "software engineering" has no real consensus definition. Software companies routinely refer to their programmers as software engineers; job listings often associate "software engineer" with entry-level programming jobs with minimal degree requirements; and, as mentioned above, the U.S. Department of Education assumes a definition that conflicts with that of the IEEE and ABET-accredited programs. Consequently, this causes confused expectations among students, academic administrators, practitioners, employers, and software engineering instructors.

### 2. Software Variety

Software applications and development environments are significantly varied. Consequently, a single software engineering course cannot address the full variety of application domains nor the variety of development environments, much less the variety that will exist in the future.

### 3. Development Practices

Process models, tools, frameworks, and languages are often unique to a particular problem domain and they may fall in or out of favor. So, it is not possible to choose the "most likely environment" that students will encounter after they graduate, much less the environment of the future. Furthermore, it is not reasonable for the instructor (or the academic department) to frequently change process models, tools, frameworks, or languages.

## Responses to Challenges

### 1.Terminology

You cannot change the definitions used by the U.S. government (the Labor Department and the Education Department, in particular). Nor can you alter the language in company job notices. But you can affect the definition within your department, your school, and among your students.

It is advisable to adopt a single, standard definition of software engineering, e.g., the IEEE definition mentioned above, within your sphere of influence. This does not imply that other definitions are not possible, but it acknowledges that a single definition can avoid unnecessary confusion and that the definition has satisfied a recognized standards organization.

A discipline involves standard terminology plus a collection of "how to's". Therefore, it is important to emphasize a consistent and coherent set of software engineering terminology, especially within the course. Students, in particular, tend to discount the significance of definitions, but software engineering relies heavily on clear communication, which in turn requires consistent terminology.

Choose a text that uses terminology consistently, has a coherent software engineering philosophy, and pays attention to the history of the discipline. A discipline is based on principles and practices that develop over time, along with its philosophy.

### 2. Software Variety

Emphasize a problem domain that interests you and can serve as the basis for several projects over time. This allows you to become more of an expert and it provides administrative stability from semester to semester. If you change problem domains too frequently, it creates extra overhead and forces you to spend time on what is new rather than what is foundational. This does not mean that you do the same project each semester, but it does mean that one project may build on another (which is a good experience for the students) and that you are "creating useful capability" within a problem domain.

Also, try projects that are not confined to information technology, because an expanding body of software involves hardware devices and control systems – what could be called "engineering applications". You may have to do some of the coding yourself, depending on the application, but that would provide a more real-world flavor since students would not implement everything from scratch. Such projects can be more interesting, your students will expand their job prospects, and it can be done with fairly inexpensive hardware. [1]

[1] See Sparcfun [8] and similar vendors.

The infrastructure that you choose (e.g., programming language, operating system, framework, IDE, source code control system, and hardware platform) will be determined, to a large extent, by what precedes your course in your curriculum and what is feasible within your department. Fortunately, open-source software is readily available. Though, what to choose and when it should change will still be an issue. The infrastructure can change over time, but resist changing too much at once and avoid making major changes each year. Whatever you do should be both feasible and sustainable.

### 3. Development Practices

It is important to be aware of current development practices, which means reading, trying out new things, and periodically altering the course. But there must be a balance between change and stability. Change increases overhead and too much change can cause chaos. Yet, too much stability causes stagnation.

Students should understand the reasons for process models, the options available, and the criteria for choosing for a given situation. There is a spectrum of choices [11].

A process model should be chosen for the group project that is appropriate and tailored for those circumstances. Students should come to realize that no one process model is always appropriate and that any process model should usually be tailored. For example, Scrum will likely not be the same from one setting to another.

We have been using iterative development methods since the late 1990's, when we used *UML and Patterns* [19], even before the text began to mention agile. Lately we have used Scrum-specific references and have incorporated some Scrum concepts (e.g., user stories, product owner, sprint) within the project. However, the course schedule (i.e., MWF) does not easily allow the daily standup; the team members are not really "experts"; the team does not have as much autonomy as they would in a company setting; and my role of instructor is not always compatible with my role as Scrum product owner.

There are several misunderstandings concerning agile that are hard to displace. Students often think that agile allows the development team free reign and removes the need for any documentation or a plan, which is a misreading of the Agile Manifesto [2]. Also, organizations or individuals may treat agile (e.g., Scrum) as a magic box that will produce "better software, faster", or the customer may not realize that agile demands more of them, in many ways, compared to a "command and control" setting. See [11] for an insightful discussion of agile methods.

*The Essence of Software Engineering* [18] attempts to move above "competing" methods to describe basic elements of any development methodology, i.e., the SEMAT kernel. Their work appears promising.

## Software Engineering I

Trying to accomplish "everything" within a 3-hour course is not possible, so, you must choose topics wisely, focusing on what is "foundational". Trying to cover everything in a typical software engineering text will not allow enough time for a project. Also, it is important for students to realize the difficulties of software development, via a small group project.

### Structure

We have been using a general software engineering text [12] to provide terminology and structure, making sure that the terminology is "standard" (as much as possible) and that the text is not biased with respect to a particular development method. We then supplement with shorter references (e.g., "Why Software Fails" [15], "Scrum: A

Breathtakingly Brief and Agile Introduction" [21]). For example, Braude and Bernstein [12] uses terminology that is consistent with Fairley [16], one of the texts used in the Software Project Management course. [2]

A Canvas [4] quiz is due at the start of each chapter, which highlights important terms and concepts. Questions are mostly True/False and fill-in-the-blank to reduce grading time. Each quiz can be taken multiple times prior to the due date (when it is recorded), and students can revisit any quiz to study for an exam. Repetition is helpful.

Two in-class, closed-book exams are worth 40%, Canvas open-book quizzes (and written assignment) are 15%, the group project is 20% and the comprehensive final is 25%. The final is a good time to review concepts, especially those from earlier in the semester. There is a temptation to forgo the final when there is a project, but the project cannot capture everything that is in the course and it is important for each student to demonstrate that they understand the "software engineering mindset".

Class time during the last four weeks of the semester are spent in the lab working on a group project, with four or five people per group.

## The Project

The four-week project typically has four iterations (e.g., Scrum sprints). During the Spring 2015 semester the four sprints took 5, 10, 9, and 7 days, respectively. Sprint 1 lays the groundwork and should be short. Sprints 2 and 3 represent the bulk of the implementation and sprint 4 contains system testing and polishing. Students receive feedback from both the "instructor" and the "product owner" after each sprint, which can be confusing since I play both roles.

The project does not try to replicate a "real" project. Rather, it is a "guided" experience, not a simulated work environment. Students are not ready for a "real project" in an introductory course. However, they are ready for "experiences" that convey the difficulties of working in a group and to be exposed to sound design and process methods.

A guided project means that the architecture, the artifacts, the basic design and the code structure are, to a large extent, dictated. For example, the structure of the domain classes and their respective sets are standardized, the format of the test output is specified, and the schedule is pre-determined. The reason: The project is short and there are many concepts to illustrate. Also, our capstone course is based on a "real" project, suggested and supervised by a third party.

The centerpiece of the project is a domain model (i.e., a UML class diagram lacking methods) and a high-level architecture, both specified by the instructor. The domain model gradually evolves into a design class diagram (consistent with Larman [19]). Students formulate user stories from an initial list of functional requirements, are given an IBM Rhapsody project [3] that contains the UML domain model, and they then create use cases and related sequence diagrams for a small number of user stories.

This approach allows for some agile (i.e., Scrum) concepts such as sprints, a product backlog, sprint backlogs, and self-organizing teams. But there are no daily stand-ups (due to the nature of a three-hour, MWF course), I cannot really play the role of product owner for multiple project groups (last semester there were nine), and the product backlog does not change as much as it could in a real setting. However, there is enough "reality" for students to realize that communication is important and that configuration management is necessary when code is shared.

We have successfully used Trello [10] to manage product and sprint backlogs. I create a Trello "board" with a specific structure for each group and then invite students to join their respective group. This allows me to

[2] R. Fairley is a senior member of the IEEE who has chaired several software standards committees.

[3] IBM Rhapsody [7] is free to universities.

manage the product backlog (which I control), each group can manage their sprint tasks, and I can see how things are progressing during the sprint as tasks are moved from "Doing" to "Done".

A peer review occurs after each sprint, which encourages students to improve their evaluation by the end of the project. Each student receives a score at the end of a sprint that is the product of their peer review (as a percentage) and their group score. The peer review asks the following questions, each of which uses a 5-point scale:

- Do they do what they are asked to do?
- Do they actively seek to help?
- Are they congenial?
- Do they communicate well?
- Do they produce high-quality work?
- Do they show up when expected?

### Avoid

Here are some things that I try to avoid during the project.

- Allowing a third party to define the project (and dictate technology), because there is not enough time for the extra overhead in an introductory course.
- Evaluating a project by only looking at the output. Good design is usually "under the hood".
- Allowing a superhuman team member to single-handedly carry the project over the finish line.
- Using a source control system, such as Subversion [9] or Git [6], without an established configuration control procedure.
- Allowing test output that can be easily faked, e.g., "Test successful".
- Allowing each group to choose how much to do for each iteration (which would be difficult to track with multiple groups).
- Allowing each group to choose their own project problem (which would create too much administrative overhead).

### Concepts to Emphasize

Concepts emphasized throughout the course include the following.

- Software development is a human enterprise that requires cooperation and communication.
- Change must be intentionally managed throughout the software life cycle, not simply during development.
- Decisions should be recorded, not simply conveyed.

- Configuration management is a process, not simply a tool.
- Coupling and cohesion influence quality.
- Purpose and strategy are different.
- Verification compares a work product with a standard and validation determines whether the work product has value.
- There is no "silver bullet" for the "Werewolf" [14] or the "tar pit" [13] that is software development .
- Any development model should be tailored to your circumstances.
- If someone inherits your project you should have provided enough information to support it.

**Recommendations**

- Use hardware devices to allow for interesting projects with minimal cost. For example, lately we have used RFID readers with a patient tracking system (within a doctor's office), a timing system for marathon runners, and a room access system.
- Coordinate early with your IT support staff. Last-minute requests are unreasonable, both for them and for you.
- Employ a three-layered architecture, i.e., presentation layer, logic layer, and data layer, with a facade object containing all logic-layer capabilities for the presentation layer.
- Use Subversion or Git to manage the software configuration during the project from within the Eclipse [5] IDE.
- Limit the number of students per project group to 4 or 5. In a real project more would be reasonable, but communication overhead becomes a significant burden when the group goes beyond five in a class setting.
- If there are multiple project groups, it is advisable to have each group work on the same problem. Multiple problems with multiple groups create far too much overhead.

## Comments

The Computer Science department at Baylor University has a software engineering "track" (not a software engineering program), which is a restricted form of the BSCS degree. In addition to the BSCS requirements (which include SWE I and SWE II), the track requires Software Project Management, Software Quality Assurance, Engineering Economics, and two semesters of calculus-based physics. SWE I students have already taken two semesters of C++ programming, data structures, algorithms, and software systems (i.e., machine language). In addition, a one-credit Java course is a co-requisite for SWE I.

## References

[1] Standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, page 67, December 1990.

[2] The agile manifesto. `http://agilemanifesto.org/`, 2015. Accessed: 2015-05-25.

[3] Bachelor of software engineering. `http://en.wikipedia.org/wiki/Bachelor_of_Software_Engineering`, 2015. Accessed: 2015-05-21.

[4] Canvas. `http://www.canvaslms.com/`, 2015. Accessed: 2015-06-22.

[5] Eclipse. `http://www.eclipse.org/`, 2015. Accessed: 2015-06-22.

[6] Git. `https://git-scm.com/`, 2015. Accessed: 2015-06-22.

[7] Rhapsody. `http://www.ibm.com/developerworks/downloads/r/rhapsodydeveloper/`, 2015. Accessed: 2015-06-22.

[8] Sparcfun electronics. `https://www.sparkfun.com/`, 2015. Accessed: 2015-05-21.

[9] Subversion. `https://subversion.apache.org/`, 2015. Accessed: 2015-06-22.

[10] Trello. `https://trello.com/`, 2015. Accessed: 2015-05-25.

[11] B. Boehm. Get ready for agile methods, with care. *Computer*, 35(1):64–69, Jan. 2002.

[12] E. J. Braude and M. E. Bernstein. *Software Engineering: Modern Approaches*. Wiley, 2nd edition, 2011.

[13] F. P. Brooks, Jr. *The Mythical Man-month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[14] J. Brooks, F.P. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.

[15] R. N. Charette. Why software fails. *IEEE Spectr.*, 42(9):42–49, Sept. 2005.

[16] R. E. Fairley. *Managing and Leading Software Projects*. Wiley-IEEE Computer Society Pr, annotated edition edition, 2009.

[17] S. E. Insider. Abet accredited software engineering programs. `http://www.softwareengineerinsider.com/abet/abet-software-engineering-programs.html`, 2015. Accessed: 2015-05-21.

[18] I. Jacobson, P.-W. Ng, P. E. McMahon, I. Spence, and S. Lidman. *The Essence of Software Engineering: Applying the SEMAT Kernel*. Addison-Wesley, 2013.

[19] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall, NJ, USA, 2004.

[20] M. J. Lutz, J. F. Naveda, and J. R. Vallino. Undergraduate software engineering. *Commun. ACM*, 57(8), Aug. 2014.

[21] C. Sims and H. L. Johnson. *Scrum: A Breathtakingly Brief and Agile Introduction*. Dymax, Menlo Park, CA, USA, 2012.