Teaching software development vs. software engineering

Gary Pollice

December 15, 2005

from The Rational Edge: Why do so many software development projects either fail or result in poor-quality products? It may have something to do with the way we educate software professionals, this author contends. Drawing a distinction between software engineering and software development, he outlines the practical approach he uses to prepare college students for their first industry job.

Since coming to Worcester Polytechnic Institute in 2003, I've taught the undergraduate software engineering course six times, counting the current offering. My students now know that, sooner or later, I will include the following item on a test:

Describe the differences between software engineering and other engineering disciplines.

I expect them to understand that software engineering has fundamental characteristics that make it especially challenging to teach and study.

There is no single correct answer to my question. In fact, a good answer will identify several differences. Here are some possibilities.

- **Software is soft.** It is intangible. You cannot touch it. It is malleable and designed to be changed. This is quite different from the physical devices that mechanical engineers design and construct, for example.
- **There are few, if any, laws of software that can be universally applied.** Chemical engineers and electronic engineers design their creations by following the fundamental laws of physics and chemistry. If there are laws of software, we have not discovered them yet. Although computer hardware designers can use well-established formulae to compute the amount of...
heat produced by the chips they design, software engineers have not arrived at a consensus about how to measure their products’ properties, such as program size.

- Software is not mass produced. Commodities such as automobiles are engineered for mass production. Although they can accommodate customized accessories, their basic designs are used repeatedly. Not so with software. Although you may copy and release a particular program such as an operating system to millions of customers, you still need to build only one actual program. The manufacturing process involves making a copy, not building another identical product.
- The specifications for software continuously change, even late in the development cycle. If you are building a bridge that is half completed, the customer is not going to say, “Gee, I think the bridge would be more useful at the next bend in the river rather than over here.” Unfortunately, such requirements changes occur continuously with software.

Once my students understand that software engineering is different from other types of engineering, they might begin to wonder whether the term engineering actually applies to building software-intensive systems. I constantly wrestle with this question. As I noted in this column last December,¹ our discipline has matured and advanced in the last half century. Yet something is amiss. Much, if not most, of the software we create is still produced, not precisely engineered. And we are still putting up with too many overruns and failed projects, and with poor-quality software.

What is the solution? There are no easy answers. I suspect that applying rigorous engineering principles to most software projects would not improve results. It might even slow down product delivery. Instead, I think we need to rethink the notion of software engineering and how we prepare people to enter the field.

**Defining software engineering -- again!**

You can find many definitions of software engineering. The very first may be the following, from a 1969 conference sponsored by the North Atlantic Treaty Organisation (NATO):

> Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.²

Today, the accepted definition is this one from the Institute of Electrical and Electronics Engineers (IEEE), issued in 1993:

> Software Engineering: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).³

Let’s consider the first definition. Many development projects produce reliable software that works efficiently. They also produce the software economically -- if they didn't, we would not have so many profitable software companies. But do these companies apply sound engineering principles to the software’s construction?
I'm not sure there is universal agreement about what we mean by sound engineering principles. The second definition attempts to provide some clarification about this. But does it reflect, any better than the first definition, the way software systems are actually implemented?

Most engineers apply rigorous problem analysis and build models of proposed solutions before they commit to manufacturing them. Electronics engineers build their models with breadboard circuits, using the laws of physics to determine expected values for various properties and then precision instruments to measure actual values. Chemical engineers prototype chemical manufacturing processes by building pilot operations. Aeronautical engineers take their models into wind tunnels to measure performance factors. Civil engineers build mathematical models of bridges and other structures to calculate how much weight and stress they can bear before committing to an expensive construction phase.

Also, in most engineering disciplines, engineers use common components for their constructions, which is more cost-effective than designing and implementing custom components. They can determine specifications for a needed component, go to a catalog, and place the order. They can predict, with reasonable accuracy, the cost of building and manufacturing their creations. However, this is not true for all engineering disciplines, including software, in which cost overruns abound.

Why don't software engineers engage in these sound engineering practices? I think part of the problem is how we educate people to work in this field.

**SE instruction lacks useful, practical information**

I recently took a look at five typical undergraduate software engineering (SE) textbooks to see what subjects they addressed and in what proportions. The results of my informal, back-of-the-envelope calculations were rather disheartening. All of the books cover process, requirements, design, testing, and project management. Most have a section on analysis, although at least one bundles analysis with requirements management and design. However, most of these sections are surveys that barely scratch the surface on any specific process or technique.

For example, although all of them mention use cases as a requirements and analysis tool, only 43 pages out of a collective 3,500 pages in the five books actually discuss use cases. That's an average of six pages per textbook. Now, if those six pages were about writing good use cases, they might be useful, but they're not. Instead, they describe use-case diagrams and their components. Only one book (Maciaszek) devotes twenty-three pages to use cases and employs them in case studies. On the whole, students simply cannot learn much about using use cases for requirements based on the information in these books.

None of the books devotes much space to discussing how to tell if you have the right requirements, either. Some have short sections on validation and verification, but those simply point either to IEEE standards or other formal methods. This is largely a waste of space. How many times do software engineers actually take time out to prove a program is correct? Almost never. Such formal methods may help us sharpen our logic, but we almost never apply them in the real world.
Most software engineering texts spend some amount of copy on project planning and estimating, including tools like the Constructive Cost Model (COCOMO).\(^5\) I don't know about you, but in my more than thirty-five years of industry experience, I have never worked on a project that was estimated using COCOMO. This doesn't mean that it is not a good tool, but it's certainly not one of the most important things software engineers need to do their jobs well.

In general, the textbooks stress practices that apply only to large projects. Methods for seven-to-ten-developer projects are given short shrift; yet these methods could help new engineers succeed with their first projects. The textbooks also tend to distort the importance of certain topics. "Documentation is the lifeblood of software engineering," claims one author. If someone told me in a job interview that, "I'm really into software engineering -- my documentation skills are excellent," I doubt if I would hire him. However, if an applicant said, "I understand iterative development and have worked on delivering software that matches the requirements, which were specified by use cases," I'd be thrilled to have her on my team.

The survey-style approach of these textbooks is their greatest fault. When we teach software engineering as a collection of generalities, students learn almost nothing about how to build real software. Even if they end up working on a large, multimillion-lines-of-code system that uses planning tools like COCOMO, or in a Capability Maturity Model (CMM) Level 5 organization, the broad-brush approach will be next to useless. Students learn best when they learn something they can apply.

So what's the best alternative? Read on.

**Software development: applied software engineering**

I've taught an introductory software engineering course for a couple of years, and I think I've hit upon an approach that works. Instead of trying to instruct students in theory and broad surveys of various techniques, I teach them what I think of as software development, the applied side of software engineering. During week one, I get students started on developing their term project, in teams. I throw them into the chaos of trying to figure out what their customer wants them to build, how to do it, what process to use, and how to organize the team. At the end of the seven weeks, these teams look back with amazement about how much they really accomplished and learned. Along the way, I fill gaps in some of the broader topic areas, always keeping in mind that they need information that may really help them.

My course in software development consists of practices students can use for the project they've been given. I make them aware of how to select the right process elements for the project. I teach them how to use a couple of techniques and tools. I try to ensure that they are proficient with these few elements and tools, and to give them the background and confidence they need to expand their capabilities when they need to. During a seven-week term, my software engineering students actively do the following:

- Specify and manage requirements. We usually capture requirements in user stories and use cases. They can see that user stories work well for short projects like their class effort, but
they also see how use cases can tie things together and make system testing more effective. In addition, they learn that requirements change, and they must respond to the changes.

- Develop software iteratively and incrementally. Teams must have working software to demo for clients eleven days after the first class. After that, at the end of each one-week iteration, the teams must demonstrate additional, completely functional features, based upon the customer's priorities.
- Unit test their software, using tools such as JUnit. Unit tests are part of every programming assignment and part of the project.
- Plan their work. For small projects, the extreme programming (XP) practice of the planning game\(^6\) works well. Each week, the team asks the customer to select a certain number of work units, based upon work completed in the prior iteration.
- Document their work. Students are expected to provide enough documentation for future students to understand the project. They learn just enough Unified Modeling Language (UML) to make reasonable class or sequence diagrams that communicate their system's structure. They must also provide user documentation.
- Identify and mitigate risks. Each team develops a risk list and designs mitigation strategies for each item on the list. Although the teams should update these lists continuously, they seldom do -- in part because their time frame is so short.
- Work as a member of a team. My current class has forty-three students who received their first assignment on day one and had two days to form teams (three with about thirteen members each) and set up their project on the school's SourceForge system. Each team is assigned a mentor -- either me or a teaching assistant -- who advises them about issues such as team dynamics and other problems that are tough for inexperienced workers to solve. I expect the teams to develop their own work style, interact effectively, and develop ways of resolving conflicts for the betterment of the team. In extreme cases, the team can recommend to me\(^7\) that a team member be fired. If they have a valid case, and the team member is unable to provide a credible defense, then he is fired from the team. He must then either find another team that will take him on or drop the course.

This project gives students a lot to absorb and become proficient in over the seven-week period. Yet they do surprisingly well as they proceed through a form-storm-perform process. Often, they choose teammates based upon previous experiences or friendships, only to discover that a friend's work style doesn't mesh with theirs. That's when they enter the "storm" period; they're upset with other team members but don't know what to do about it. When one or two students come into my office to talk about how to deal with problem team members, I ask questions to guide them through the problem and let them come up with their own solution. Rarely have I ever had to "fire" a student.

This term, one team did a lot of work during the week, but did not get their work ready for demonstration. Taking on the role of Russell, the customer, I chided them and asked what they thought I should pay them for producing nothing. Then, I suggested that the team grab a corner in a lab some evening and hold a programming party. They could all work together and help each other actually produce something. It worked. Before the iteration ended, they asked the customer for more work to do. In another instance, a team implemented some nice features, but
the customer had not requested any of them. A few words got them on track, and they've done great work since.

So, are my software engineering students learning software engineering? I'm not sure I could achieve consensus on an answer. Will my students be able to work as part of a software development team when they graduate? I am sure about this. The students who learn the lessons I teach can be productive members of a software development team. They will know a little bit about process and how to make it work for them. They will know how to write software that matches the requirements and test that software. They will know how to deal with the constant change that's part of the real world.

I recently heard a talk by the CEO of a company that develops systems for debugging silicon circuits. "The real world is messy," he said. "Mostly what it takes to get a product to market is the ability to organize your work, collaborate, and persevere." My students will be prepared to deal with the mess. For now, I'll leave to others the loftier goal of making software engineering more like other types of engineering. I get my thrills when one of my former students sends me an email to say she's thriving in her job and applying what I taught her.

Notes

1 See "How Far Have We Come?" The Rational Edge, Dec. 2004.


5 COCOMO is the COnprehensive COst MOdel developed by Barry Boehm, currently at the University of Southern California.

6 See http://www.xprogramming.com/xpmag/whatisxp.htm#planning

7 In my role as professor. I also play different persona on the projects, including Russell, the customer.