Soft skills for software engineers

Gary Pollice

September 15, 2006

from The Rational Edge: What skills do you value among the individual contributors who make up your software development team? See how a veteran of the software industry, who's now a college professor, links the skills that get measured in an academic setting to successful practice in "the real world."

A couple of years ago, I had a conversation with a colleague who had been on sabbatical the prior year. He spent his time at a local company where he helped them improve their software engineering practices and did research with engineers from the company into new design methods. One of the people at the company had asked him, "How do you grade software engineering assignments?" He was somewhat taken aback by the question and then realized that he didn't have a good answer.

I've thought about how I might answer this difficult question, and I'd like to share my thoughts about this. In many ways, grading software engineering in school compares to performance evaluations for software developers in industry.

What's so hard about grading?

Some of the difficulty we educators have with grading software engineering assignments stems from something I and many others have talked about before. Software engineering is different from other types of engineering.¹ If I enroll in a civil -- or some other established engineering -- curriculum, I learn the basic physical laws that govern the domain in which I operate. I learn the formulas that have been proved by mathematical logic and empirical evidence. The instructor poses a problem, and I solve it using a theoretically and practically sound body of knowledge. There is usually one right answer to the problem.

This is not the case in software engineering. When I grade software engineering assignments, I feel more like a humanities professor grading essays than I do a science and engineering
professor grading tests. I determine grades by several soft criteria that, to my students, seem to border on pure subjectivity. Part of the problem is that I have trouble producing an exact definition of the criteria I use when grading. I suspect this inability is common among my colleagues teaching software engineering.

The difficulty, in my opinion, comes from the lack of definitive, canonical answers to most of the problems we work on and assign to our students. True, some problems can be given that clearly have a right or wrong answer. But most are not so clear cut.

A sample problem

Let's look at an example of a problem that has both exact and fuzzy answers. The first homework problem for my object-oriented analysis and design (OOAD) class this term asks the students to expand a simple software application that consists of a RemoteButton class and a DoggyDoor class. The dog owner pushes a button and the door opens for 5 seconds before closing. The assignment asks them to add the capability for bark recognition so that when the dog barks, the door opens for 5 seconds. Additionally, they need to describe their design decisions and identify potential problems with their ideas for solving them. Finally, they have to supply a class diagram that describes their solution.

A simple solution to the problem is shown in Figure 1. We add a new class that interfaces with the bark recognition hardware that is responsible for triggering the door's open method. This is a perfectly acceptable solution for the problem at hand.

![Figure 1: Simple solution to the doggy door problem](image)

Figure 2 shows a different solution. The student who submitted this defended her solution by referencing the don't repeat yourself (DRY) principle. The button and recognizer both needed to open the door, so she encapsulated that behavior in the abstract DoorActivator class.
Figure 2: Alternate solution to the doggy door problem

Which of these two solutions is correct? Is one definitely better than the other? These are questions that provide fodder for great arguments over a drink. These discussions are worth having because, if they are conducted with respect for the participants, they can lead to deeper understanding of different design principles and techniques. But should these two students receive different grades for their design?

Here's my analysis as I try to assign grades to these solutions. In the first place, both solutions work. The second solution requires that the hardware send the message `openDoor` to the appropriate object. This could be a problem if you have no control over the hardware. You would have to add a method to the BarkRecognizer, for example, that serves as an adapter method. In this case, the first solution is better.

Both solutions are clear and understandable. The second design may be better for certain types of future expansion, where other devices are added that also activate the door. The first solution is a bit simpler and follows the agile practice of only doing what you need to solve the current problem.²

As a teacher, I have to ask: "Should either solution receive less credit than the other?" I chose to give them both full credit.

**Some black and white, along with the gray**

Other solutions are easier to find fault with. Some of my students turned in Unified Modeling Language (UML) diagrams that used the wrong symbols or connectors. This is, to me, especially troubling since the tools we use let them reverse-engineer a class diagram from their code. Even if they choose not to do that, the tool makes it very easy to choose a correct connector. I don't stress UML that much in my class, but I do expect the students to be able to create correct diagrams for the little that we do use.
Another part of the assignment that was easy to grade was the requirement that they provide JUnit tests that cover all of their code. All of the students use the Eclipse platform to develop their code, and we are using the Coverlipse plug-in for gathering code coverage statistics when JUnit tests run. It's easy for graders to run the tests under Coverlipse and determine if there is 100% code coverage or not. The screen shot in Figure 3 shows the coverage data produced by running the tests on one solution. All of the files that did not have 100% coverage were test files, which is fine. The files that belong to the problem solution are all covered 100%. This submission received full credit on this part of the assignment.

![Coverage statistics on JUnit tests using Coverlipse](image)

**Figure 3: Coverage statistics on JUnit tests using Coverlipse**

But these JUnit tests still must be examined carefully to ensure that they are indeed good tests. One of the tests most frequently missed was designed to ensure that the system behaved properly in cases like this:

- The dog barks and the door opens (for 5 seconds).
- Then, 3 seconds later, the dog barks again and the door should remain open for 5 seconds from this bark.

The total door-open time should be 8 seconds. Some students didn't handle the situation at all. Some had the door open for 10 seconds. Both of these were wrong. I worded the requirements specifically to indicate the behavior I wanted.

The remainder of the assignment presented me with other difficult grading problems. Students had to think about possible problems and how to solve them. I look for sound, creative opinions on what they write. Again, there is no right answer. Should they think about cases where the neighbor's dog barks and the door opens to let a skunk in from the backyard? What about the door closing as the dog is entering or exiting? What happens if there's a power failure? All of these are things they might consider. If they don't come up with the same set of complications that I have in my own imagination, are they wrong? Who says I'm right?

Clearly, to me at least, there are many good answers and probably no best answer. I and my teaching assistants who grade these assignments must use judgment that we have gained from experience to fairly grade the students.
I want to point out one area that I grade the students on, something that many of them dislike. If they have spelling or egregious grammar errors, they get points taken off. They must learn to communicate well, and I make this clear from the very first class.

**On to software engineering**

I've just described an example from my current OOAD course, which is the second in our software engineering series. The first is a software engineering course that most of our computer science majors take in their second or third year. So, in the software engineering course, the students haven't learned most of the design principles that I expect my OOAD students to know. Grading is more difficult and often more subjective.

A typical software engineering course for my students focuses on giving them the skills to work effectively on a team, do iterative development, and deliver working software. Many of these objectives and the outcomes I expect are quite soft.

One of the first assignments I give the students in software engineering is to write a simple program, but to work in pairs (i.e., try pair-programming). As part of the assignment, they have to write a short paper describing their experience with the practice, whether they think it helps them develop better software or not, and defend their position. The work is not nearly as intellectually challenging as what they will face in the OOAD class, but it lays the groundwork for team collaboration, which is a major learning goal.

How would you grade the assignment? Clearly, if the program doesn't work, there's a problem. But is the problem with the student's lack of technical knowledge or a problem with pair-programming? How much weight should I place on the code when the goal is to have them try a new practice and reflect upon it? I choose to place most of the emphasis for the grade on the report of their experiences with pair-programming. I look for clarity in their presentation and well-thought-out arguments and conclusions. There is no right answer as for whether pair-programming is good or bad, since it depends upon the individuals and the pair; but faulty logic or simply a bad argument is easier to spot, just as good arguments and sound reasoning are.

The software engineering class has fewer assignments than the OOAD class. That's because the course-long project in the software engineering class is much bigger than any project they've encountered thus far in their academic careers. Teams usually consist of ten to fifteen students, so they spend a lot of time working on their collaboration skills. This collaboration is very hard for the type of student who is used to courses in the hard sciences.

The project counts for about one-third of their grade. My teaching assistants and I get together to discuss the teams and their interactions. Acting as mentors for one or more teams during the term, we try to coach them on developing collaborative skills, and we observe the individual students' interactions with their teammates. These observations are a significant part of the input we use to determine their grades.

Interestingly enough, the best technical project is often not the one produced by the team that gets the best grade. Typically, team dynamics are poor on the teams with better technical solutions,
usually because one student takes over the team and just does things his or her way. And this self-appointed leader isn't the only one to blame; it's also the team's fault for giving up and letting the "hero" have his or her way. We see heroic effort by one person, but a broken, defeated team whose members have lost some of their passion for developing software -- not unlike many teams that I've encountered in industry.

Most of you have probably been on projects like these. Whether or not you were the "hero," my guess is that when you reflect on the experience, it isn't a pleasant memory. You either came to dislike the hero or you felt the negative feelings your teammates had for you. Often in my software career, I encountered really smart, competent software developers responsible for short-term gains in the company's product line, but long-term losses in the quality of working life. I think there were times when I probably fell into this category and I'm now embarrassed by these memories. I'd like to think it was a case or two of youthful exuberance. I hope that as I've aged and matured, I've become more aware of the softer skills and have a broader view of what makes someone a good teammate.

Employers who hire our graduates tell me that one of the big differences between the students who come from WPI and those from other schools is that our graduates are able to work on a team project when they start the job. They're not afraid to take on hard technical tasks, but they know how important other, less technical factors are to the success of the project.

**What about the "real world"?**

For most people who work outside of academia, the question of "grading" software developers in an organization is frequently relevant. If you're a manager, you want to reward good values and behaviors, but which ones? If you're a developer, do you have the ability to affect your team? If so, do you affect the team positively or negatively -- and from whose viewpoint?

I submit that managers should think about evaluating their developers along the lines that we academics do in grading our students. I believe that many of the qualities we value in our students are the same that will help business organizations thrive.

It seems obvious to me that there must be a **bottom-line** component to any evaluation. But there are tactical bottom-line victories that can lead to strategic defeats. Part of being a good manager is the ability to develop talent in your team that can lead to strategic, long-term success. You have to balance the short-term gains against the long-term competitive viability of the organization.

**Developing non-technical skills**

Quantifiable technical ability is a primary attribute that gets a person hired as a software developer. They bring needed skills to the organization and project. For many years, this was the only thing that mattered. Developers would go to a company, do great technical work, then find that they needed to move on -- usually because other members of their project team really didn't want to work with them on another project. But there were lots of companies who needed their technical expertise and were willing to pay for it. Job hoppers were the norm during those years. In the 1980s, we rarely encountered anyone who actually reached their five-year anniversary with a company.
For the sake of full disclosure, I will admit that the '80s was for me, too, a time of frequent job changes. Part of it was working for companies that were in trouble, but mostly I was still trying to find my own path in software development. I wanted to learn, I wanted to do great work, but no one had ever mentored me on the non-technical skills that I now realize are so important. The choice was to be a journeyman developer or become a manager, and I've never wanted to be a manager -- at least not a manager who went for an MBA and had ambitions to be a captain of industry. Then I read Gerald Weinberg's book, *Becoming a Technical Leader*, which got me thinking differently about teams and teamwork and how I could better fit in and have a positive influence.

But back to the point: If technical ability is the primary characteristic that gets someone the job, what keeps them on the job? For the strategic goals of the company and the longevity of the employee, the relationships among employee, company, and the people in the organization are important.

Remember my grading analogy? I tell my students about the soft skills, and along with my teaching assistants, I help them learn something about them. It doesn't work for everyone. Some of my students are "fired" from their teams because they are unable to work on a team. These students don't pass the course. Others find talents they never thought they had and learn that delivering a product through collaboration can be an extremely rewarding experience.

**Valuing "soft" skills**

Do you evaluate your developers on their soft skills? How much emphasis do you put on them? For that matter, do you make it clear what you value? This last question is the key and the one I want to leave you with.

I find that if I don't tell students exactly how they're going to be graded, some of them are surprised when they don't do well. They may think they've done commendable work, and technically speaking, they have often. But I want them to be part of a team, and if they're alienating their teammates, they're failing the lesson. If you take one thing away from this month's column, I hope it's the importance of clearly expressing what you value, whether it's for those who report to you or even for your peers.

Companies, especially large ones, have huge human resource departments that provide corporate guidelines on how to perform performance evaluations. They provide training for managers, and much of it is really good. But they often miss the mark when it comes to defining the value of the soft skills.

When annual reviews came around, back in my industry days, I was always frustrated, sometimes depressed. My wife couldn't understand it. I almost always got great reviews and a nice raise. My problem was that I was working as hard as I could, giving the company my best. I kept thinking "what more are they going to want from me now?" When companies asked me what **stretch goals** I was setting for the next year, I would picture myself as a rubber band, stretching more and more each year until one year, I would suddenly break. I'm sure this wasn't the intention, but that's how it felt. By contrast, simple communication of what the goals really meant, not to mention getting
my buy-in, would have gone a long way to helping me understand how to better fit in with the corporate culture and contribute.

Not all employee-company matches will work. There are many times when the goals, values, and needs just don't match. That's okay. Like software projects that are doomed to fail, it's better to find out sooner than later. But some of the failure can be avoided with a clear description of how you "grade" your software engineers. By clearly expressing what you expect from them, and by mentoring them along the way, you will find that you'll have more matches than mismatches. And taking a little time to improve your grading skills can yield employees that will want to work with you for a long time.

Notes

1 I'm using the term software engineering broadly here. It includes design and analysis courses as well as courses that are titled "software engineering."

2 This is called You Aren't Going to Need It (YAGNI). The agile folks have another principle, which is to do the simplest thing that can possibly work (for the problem at hand).

3 Coverlipse is an open-source plug-in available at http://coverlipse.sourceforge.net/index.php
