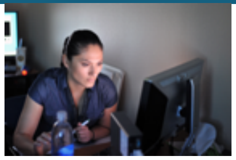
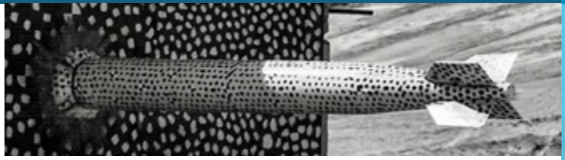
# Software Engineering as Craft

*Paul Wolfenbarger*

craftsmanship in software matters as much as algorithm design or optimization and has as great a reward

## Backsliding

As part of an effort to improve our software testing, my group at work recently started watching some software engineering videos. What struck me most forcibly about them was not the content itself, since I have read and used or rejected almost all of these methods in my career. The real surprise was that somehow I had ceased to utilize them in a full and robust manner. I can think of several reasons why this occurred: understaffing, short deadlines, and significant technical debt, for starters. Yet these are exactly the techniques that are best suited to help resolve those issues in the long term, so how did they get bypassed?

## Finding your way back

Simple beats cool, fast, slick, and innovative, basically anything except simpler. Yes, I work in high-performance computing (HPC) where every clock cycle can and usually does matter; but, yes, simpler IS better than faster.

- *We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. -- Donald Knuth.*
- Of course, when it counts, we want the important code to be in that 3% and to run fast. The other 97% will be read more by humans, used to connect to other software and determine how reusable your software really is, and it will carry more information and receive less attention.
- This less-critical code needs to be as robust as possible upfront precisely because it will get less attention every day.
- When you write any code, take the time to make it small, make it legible, test it well, and make it compile cleanly. You will thank yourself later.

## Know your History

*I never look back darling -- it distracts from the now. -- Edna 'E' Mode in the Incredibles.*

*No Capes! -- Edna 'E' Mode in the Incredibles.*

- Everyone who works in a field learns something that can help everyone else. Do I need to understand the methods a Roman engineer would have used to design an ancient aqueduct to create a modern dam overflow?
- I might stop with just what the equations are and how I apply them
- I might want to understand how they were discovered
- If I have an interest, I can go figure out how men without those equations designed structures that still bring water reliably over large distances.
- Whatever level of interest you have, you will learn something helpful to your current effort and improve your work.

## Think About Interfaces

Interfaces are so ubiquitous in software we often don't even realize we have written them or violated the boundaries they are meant to represent. Each source file, each header, every library, data structure, function, and class, each decision about what headers will be installed and which will not represents a boundary that we set up for one reason or another. What those interfaces are and how they are (and can be) used will determine how your code can be understood by others and how they will use it. Allowing them to happen by chance is simply not acceptable.

*Any fool can write code that a computer can understand. Good programmers write code that humans can understand. -- Martin Fowler, 2008*

While Fowler was not actually referring only to interface design, his comment applies well.

## Think About Interfaces

In my own case, we are dealing with several large legacy codes that use multiple research libraries and several commercial libraries. The dependency graph is the proverbial "Big Ball of Mud." That means our compile times are slow, extraordinary methods are in place to constrain configure times, and linking is not parallelized because the applications are simply too large. The compiler understands all of this, but most of the developers are lost without reading prior examples. We do see that not all of our dependencies contribute to this issue. In fact two require only run-time linking to change versions in most cases. The difference is entirely in the volatility of the API and ABI, which have been well designed to constrain the interfaces, whereas most of our in-house work and partner work is more free-form or completely unstructured.

Interfacing with the compiler or interpreter is the easy part; the humans are harder. When you convolve both of those with complex build systems it can seem hopeless. It's not. Simplify every interface.

In short, make sure you can sort it out easily years down the line when you can't even remember writing this thing. The machine will have far fewer problems anyway.

## Science or Craft

Craftsmanship in a project can be more important than the engineering, architecture, or science involved. We use those terms to describe what we do, but they are ill-fitting. Perhaps you prefer artisanship or professionalism over craftsmanship, but attention to detail and care about what you are doing will make more of a difference than all the theory in the world.

My educational background is in structural engineering, and I have done many masonry buildings here in the Southwest. I realized quickly that the difference between having a good versus an excellent mason was equally as important as having a good engineer. Having a good mason often made a mediocre design feasible whereas a poor mason could make a good design have a very short lifespan.

In the same way that masonry craftsmanship makes a huge difference in a construction project, software ``craftsmanship" makes a critical difference as well. Good computer science and engineering knowledge are important, but we need to realize that proper software assembly and design are crucial. This is the point that I forget the easiest. I see a lot of my colleagues forget it as well, and I have to think that's because it's not built into our reward structures (if it's built into yours, let me know).

## Test Everything

In 1999 I was working on a rule-based embedded AI system that did some interesting stuff. Needless to say, the interactions between the various pieces was the actual algorithm, not any of the routines you could point at. I found bugs and unintended interactions left and right, and I started keeping a set of inputs that would allow me to be sure that issues I fixed did not come back. I had started a regression suite. Later I started to add unit tests and library level tests, and I have gradually joined the vast majority of code monkeys on the planet in testing my code thoroughly. Or somewhat at least -- we all know that most software is lucky to hit 75% or 80% of line coverage and a lot less of the use cases or scenarios are actually covered. Yet if that is so, how do we know it works? The answers range from bad *"the daily use is the best test"* to the dangerous *"it did what I wanted when I wrote it"* to the oblivious.

The stint watching SE videos lately has my team trending toward test-driven development (TDD). It did not change the team policy of "new code is 100% covered, and modifications to old code are new code." It has helped us live up to that policy, and that I cannot fault.

## Test Everything

- So, why do we test? What do we test?
  - We test to make sure the code does what we think it does,
  - We should test everything if we can.
- How do we know the code won’t be asked to do more than we expect? (It will.)
- What is everything?
  - Every modification to the code should add modifications to the tests.
  - Don’t let Joe down the hall use your code for something odd if he’s not going to add the tests.
  - Tests and code must match, or an error shows up.

## Do it all Again

Each of us will have our own method for keeping these principles working. Some will insist that the most modern IDE must be used, some that hand coding the build system is the best defense, others that generated documentation will reveal all. In the end it is attention to the work that really matters, and each of us finds a way to that on our own.