Viewpoint | Wei-Lung Wang

Beware the Engineering Metaphor

There are great similarities between the work of the software developer and the work of the mathematician.

LISA HANEY

As software developers, we often fail to learn from our mistakes. Many software projects appear needlessly painful because we didn't "leverage an existing code base," or "draw on proven techniques and best practices." While this is sometimes the result of a failure to instill discipline in the development process, it is also a reflection of the fundamentally malleable nature of our work. No matter how hard we try, it is often difficult to reuse past designs or to adhere to any fixed set of development rules. The result is software quality disproportionately dependent on the abilities of its developers rather than on the quality of instituted processes and building blocks. This reality has often prompted us to refine the engineering approach to building software with a focus on ensuring software quality.

While the engineering approach has its place, it is important we realize the work of building software is inherently *not* an engineering task. Engineering is the work of applying scientific and mathematical principles to practical ends. Software engineering, on the other hand, is the use of engineering practices to craft mathematical rigor. As software engineers, we do not engineer software, we merely adopt engineering practices when building software.

This distinction may appear subtle, but its implications are real. Unlike engineering, the essence of building software is its mathematical undertones and the role of information as its raw material. This difference is manifested in two ways: our inability to create a frame of reference—making it difficult to establish rules by which to carry out work; and our inability to reliably engineer structural quality.

Software Engineering Is Not Engineering

Fundamentally, engineering operates within the framework of the immutable laws of nature. These laws dictate the realm of engineering possibility, and engineers work by designing and constructing within the bounds of these laws. Their permanence and universality allow engineering principles, which signal the boundary of what is safely possible, to be established. For example, engineers who violate known electrical circuit design guidelines are potentially breaching the physical limits imposed by the forces of electricity. These guidelines can be established simply because their applicability is universal. Competent engineers observe well-known limits that cannot be breached, while negligent engineers who fail to observe these limits are potentially negligent.

Software engineering, on the other hand, has no fixed framework in which to operate. At its heart, software is the embodiment of a Turing program. Like the Turing machine that can compute all that is computable, software can be designed to process information in any way it needs to be processed. Because Turing instructions are put together in the context of a set of information requirements, they observe no "natural limits" other than those imposed by those requirements. Unlike the world of engineering, there are no immutable laws to violate. Likewise, software components, being collections of instructions, only function in the context of their information requirements. They can be assembled together only insofar as their information requirements are compatible. In other words, the natural limits of component interaction come solely from the information requirements unique to the project and component design. As software developers, we are in the unique position of not only having to design our systems, but are also called on to design the micro and macro information requirements, and by proxy, the framework of "natural limits" in which we operate.

Requirements, software design, and natural limits are tightly intertwined. This unique position is best illustrated by trying to answer these questions: How do we determine if professional negligence caused a software project to fail? Did the software engineers violate known guidelines? Or, was it simply a series of bad judgement calls?

This absence of a fixed framework of natural limits has two major implications. First, we have difficulty establishing across-the-board design guidelines and principles. The framework of natural limits differs for each project, meaning there are no hard and fast rules that apply across the board. The best we have are guidelines whose effectiveness depends on the developers' interpretation. It is the people and not the rules and processes that are paramount in the development process. The second implication is that we have difficulty building general reusable components. Each project has different information requirements, meaning that components must be developed specifically for it. Only in the most generic cases, such as mathematical operations and system functions where information requirements are standard, can components be reused. Everything else usually ends up being built from the ground up.

Limits of the Engineering Approach

These implications reflect the mathematical nature of software, and are symptomatic of the limits of the engineering approach. As a mechanism for information computation, a piece of bug-free software is essentially a mathematical function in some system of axioms. The information domain, range, and mapping requirements are simply the constraints this function must satisfy.

In this light, the work of building software is a mathematical exercise within two distinct phases: the determination of the set of information requirements (the parameters of the function); and the creation of a mathematically rigorous function that meets these requirements. The work of the software developer is very much the work of a mathematician. In the world of one-man programming, the programmer, like a mathematician, uses his or her creative insights to create mathematically sound programs. Team-based software engineering, on the other hand, is akin to putting many minds together to engineer a single mathematical construct. This is not an approach that has a record of success in the world of mathematics, yet we are by necessity building software in this manner.

To ensure some level of software rigor, we usually have a single chief architect oversee macro-level conceptual integrity. Unfortunately, the single-mind unity that allows for mathematical rigor is lost at the micro level, where software development is by necessity a team effort. It is rare for all team members' code to have the micro-level rigor needed to work together seamlessly. While this can theoretically be addressed by having the chief architect issue rigid black-box specifications to code to, the reality is software requirements change so often that such specifications would be unwieldy. The many changes that occur on a daily basis, both major and minor, means developers play a crucial role in crafting overall rigor. They need to understand the scope of their work as part of the bigger picture in order to build components that weave into the program's ever-changing information tapestry.

This collaborative engineering approach toward crafting mathematical rigor is perhaps unique to our industry. Because mathematical rigor is difficult to engineer as a team effort, we find it difficult to reliably engineer software quality. Software testing is simply the attempt to reconcile the imperfect results of the engineering approach with the mathematical rigor required for robust software.

Tangible Mathematics—The Essence of Software

If software engineering is an exercise in mathematics, then why does mathematics appear so unimportant to the practitioner? And why has the engineering approach worked as well as it has? The key reason is software is a tangible form of mathematics that lends itself to being engineered. At its core, a program is an abstract sequence of instructions that performs some computation. But when the program is realized on a computer, it becomes an information tool with its own use-feedback cycle. It changes from an ethereal entity to a tangible tool whose actions can be observed. Instead of mathematically proving the results of a program, we can simply run it on some set of inputs and observe its behavior. This tangibility is both software's strength and Achilles heel.

To our benefit, the tangible use-feedback cycle allows the nonmathematically inclined to build software. Software developers, unlike mathematicians, do not need to manipulate abstract concepts to construct functional code. They leverage the use-feedback cycle by running their code and correcting any bugs. The code is gradually brought up to the required specification over repeated compile-execute cycles. This allows us to adopt a build-and-test approach common in the engineering disciplines, and makes it possible for us to engineer complex software systems within practical resource limits.

Building software is a complex and exciting task and it is important we resist the tendency to view it in a single dimension.

Unfortunately, this tangibility is a double-edged sword. While it makes software construction accessible, it also provides a shortcut to the more difficult task of proving the mathematical rigor necessary for bug-free code. By allowing us to engineer software, it deceptively obscures the mathematical nature of software. While it is unlikely we can abandon the engineering approach in favor of mathematical formality, we should always keep in mind that mathematical rigor is the basis for sound software.

Pitfalls of Engineering Software

The bottom line is software engineering is not engineering, and the engineering approach to building software has its limits. This does not mean we should revert to the days of the hacker mentality "programmer-artist," for the scale and complexity of software today requires the disciplined engineering approach. But it is important our familiarity with the engineering metaphor be tempered with the knowledge that building software is inherently not engineering work. We need to avoid the temptation to overengineer the software development process. Rigid development processes and guidelines will strangle the fluid creativity needed to craft mathematical rigor. Likewise, time can be wasted on excessive efforts at componentization and building libraries of reusable code.

The engineering terminology we communicate to customers can also lead to unrealistic expectations. Terms like "components" and "foundation platforms" often create expectations that align themselves more with mechanical and electrical engineering work. This results in such refrains such as: "Since the foundation is already coded, why can't we just bolt on this extra feature?" and "This new workflow is only a variation of the old workflow. Why can't we just reuse the components you wrote earlier?"

Call to Action

As a profession, we need to evaluate the limits of the engineering metaphor and adjust our approach to building software accordingly. We would do well to communicate this to our stakeholders and customers, so our work is not misunderstood. Building software is a complex and exciting task that is a unique confluence of engineering, mathematics, and artistic insight, and it is important we resist the tendency to view it in a single dimension. Only then will we best be able to move our discipline forward. C

Wei-Lung Wang (weilung@alumni.nus.edu.sg) is a consultant at Adroit Innovations in Singapore.