Software: Craft, Science, and Engineering

Ruth E. Davis, John R. Allen
Santa Clara University

Abstract
What is Software Engineering — art, trade, craft, science, or profession? Some believe that, at least in its current state, software engineering is an oxymoron. If we believe that there is something worthy of being called software engineering, how do decide what constitutes a proper education for a software engineer? We argue that it is informative to study the history of traditional engineering, when and how it became science-based, and when and how engineering education became science-based. By learning from that example, perhaps we can speed up the adoption of appropriate foundational studies in software engineering programs, allowing us to move more rapidly from a craft to a profession.

Traditional engineering moved from what is called the “shop culture” to the “school culture” over a period of 200 years. Today “shop” engineering is relegated to trade schools, and professional engineering is the province of the colleges and universities. We will trace the mathematical advances that supported this change of perspective, and outline the corresponding shift in the educational environment as well as the sometimes-reactionary resistance to this change. We show that software science has a common mathematical root with traditional science; and that foundational problems in the mathematics of the continuous led to the development of discrete mathematics and the foundations of software science. We then illustrate how these mathematical results blossomed into modern computation, and should give rise to a “school culture” of software engineering and a corresponding professionalism.

Rise of Professionalism in Traditional Engineering
When we look at what now constitutes preparation for a degree in civil, mechanical, or electrical engineering, we take a lot for granted. Imagine for a moment that you are an entering student in a traditional university’s engineering program. What if the first offering you were required to take as a Civil Engineering student was a course in heavy equipment operation? Or as a Mechanical Engineering major, you were shipped off to the auto shop? Or Electrical Engineering began with a course in basic household wiring?

Though this seems patently absurd nowadays, there’s plenty of historical precedent for this “shop culture” approach to these engineering disciplines. But as theory developed and showed its benefits in practical applications, such apprenticeship-driven, hands-on programs were replaced by what was called the “school culture.” We see the school culture in contemporary engineering programs: the first courses are heavily loaded with theory — calculus and physics: the symbolic manipulation of continuous phenomena. The “hands on” courses are relegated to labs and to the
trade schools.

Yet history tells us that as late as the 1920s the educational establishment in the United States questioned the necessity for calculus in the engineering curriculum. History also tells us that Newton and Leibniz set down the foundations of the calculus around 1700. Two hundred years is a long time. What went on in the interim? And what do the developments in traditional science and engineering have to tell us about the science and engineering of software?

There is always a lag between the development of mathematical ideas and their incorporation into the wider body of knowledge. Science, with its goal of extending knowledge, may take up the new ideas more rapidly than engineering, where the goal is to apply existing knowledge.

**Early Engineering Education: Shop versus School**

Not surprisingly there is a strong parallel between the development of engineering and engineering education. Early engineering was a product of accumulated experience, passed down to novitiates through word-of-mouth and strenuous apprenticeship. Apprenticeship not only transfers knowledge and wisdom, it also transfers culture—the professional attitude toward the particular craft and the attitude that a practitioner should cultivate toward customers and quality.

Before symbolic algebra, and particularly before Newton and Leibniz, there was no alternative to a craft-based engineering practice: one could not begin to substitute symbolic analysis for practical experience. But near the end of the 16th century things started to change.

In a concentrated period of time we have Copernicus, Brahe, Kepler, and Galileo challenging the status quo regarding the makeup of the physical world. We moved from simple categorization of nature to the analysis of, and experimentation on, natural phenomena. Thus it was critical that symbolic algebra came upon the scene.

As we went into the 17th century, most of the pieces were coming together—the beginnings of scientific discovery and symbolic algebra. And so the stage was set for Newton and Leibniz; and with their work, modern science and scientific engineering were within grasp.

Analytic alternatives to craft-based engineering began to trickle out of the mathematical community around the beginning of the 18th century; but early analytic techniques dealt only with very idealized situations, and in many cases were not up to the task of analyzing the rough-and-ready real world. Second, and this brings us to another topic: a major component of the practical engineering world was the apprenticeship program and its ability to teach, by example, a sense of personal responsibility in those practicing the craft. Some parts of apprenticeship can be covered by hands-on courses. But if on-the-job personal experience is replaced with in-the-class mathematical analysis, then who takes care of ethics and responsibility?

These issues in the practice of the craft, boiled over into the training of the practitioners, taking shape as the “shop culture” versus the “school culture”—the practitioners versus the academics. In mid 18th-century France, schools for the training and education of engineers began to sprout.
One fountainhead of mathematical engineering flows from the artillery school at Mézières. Gaspar Monge, Charles Coulomb, and several other excellent mathematically educated engineers came through this school. Monge's work represents some of the first applications of mathematical finesse in engineering. As a teenager, Monge developed descriptive geometry. It is something that appears simple to any engineering undergraduate. And yet when Monge described his result to his superiors in 1768, the technique was immediately classified as a military secret and Monge was not allowed to speak publicly about it until 1794. Though the mathematical content of descriptive geometry is humble, it did show conclusively the power of reasoned engineering versus the practitioner's approach. If something as militarily important as the design of fortifications could fall to such simple mathematical tools, how much more could be done with more sophisticated mathematics?

Geographically, the transformation of engineering from a hands-on practical trade to a mathematically based profession proceeded from France to Germany in short order; by the early 1800s Germany was following France's lead in education. By the mid-1800s, professional engineering education was becoming accepted in England. And in the late 1800s the United States began the slow process of transforming their shop-based hands-on approaches into school-based programs.

Even with the acceptance of professional engineering education in England, there remained the tension between reliance on the experience of the practitioner and on analytical methods. We consider a quick example concerning bridges — the Dee Bridge that, for a very short time, crossed the Dee River in Northwestern England. Robert Stephenson, a well-respected bridge designer, built it in 1846. A great deal of practical experience and analytical design went into the bridge's conception. Unfortunately an undue amount of faith went into it as well. All structural design contains a "safety factor" — a fudge-factor that is included to insure that unforeseen occurrences will not cause catastrophic failure. In cast-iron bridge design of the time, it was typical to use a factor of 4 to 7, meaning that if your analysis suggested that a beam of strength N was sufficient, then you’d specify a beam of strength 4N to 7N. Stephenson was so confident of his design and his calculations that his safety factor was only 1.5. (By way of by comparison, a factor of 1.8 was considered appropriate in 1986. And of course in 1986 engineers knew a great deal more about strength of materials, as well as having better analytic and computational techniques.)

When the Dee Bridge opened to train traffic, vibrations were noticed but not considered dangerous. The perceived danger came from embers that fell on the wooden ties supporting the rails. So Stephenson and company decided to add some rock ballast to the roadbed to dampen the vibration and remove the danger of fires. The ballast did both: the bridge ended up in the river where neither vibrations nor fires would plague it. (They didn't think it necessary to review the calculations with the added weight.) Unfortunately five people died as the first train tried to cross the ballasted bridge.

So what’s the lesson here? Engineering is a human enterprise; it requires theory and practice; and it requires rules and laws. But even more it requires a sense of personal responsibility. Engineers must think about, and design for, failure. Whether you’re building a physical structure or writing
code, you must exercise care and must be guided by design principles that take precedence over expediency. Understand the consequences of your choices and act accordingly.

In the late 1800's engineering had grown from its civil base to encompass developments around steam power. This gave rise to mechanical engineering and a well-entrenched bureaucracy of shop-based engineers controlling the profession and its educational component.

The issue of shop versus school varied depending on the field of engineering. Civil engineering — perhaps because of Monge — took to academics faster than mechanical engineering. In fact the term “shop culture” is derived from the practicality of a machine shop, and the attitude that all engineering education began (and frequently ended) with the machine shop.

The real break with the hands-on craftsman and apprentice tradition came with the beginnings of electrical engineering. With electricity one no longer could depend on immediate sensory information. Since measurement was indirect, mathematics became essential to assure effective and safe application. The laying of the Trans-Atlantic cable around 1860 put the shop-versus-school approach to engineering in stark contrast. The first models of the telegraph (around 1835) were seat-of-the-pants affairs: try something and see what it does. Marconi was of this opinion, while Michael Faraday and William Thomson supplied many of the theoretical underpinnings of telegraphy.

As the practitioners formulated their plans to lay a cable from Dover to Calais, Thomson and Faraday were able to predict that the performance of the proposed cable would be unacceptably poor. But Cyrus Field, the entrepreneur in charge, would have none of this. His engineer, Dr. Wildman Whitehouse described Thomson’s work as “a fiction of the schools” and proceeded to lay 2350 miles of the cable at the bottom of the Atlantic Ocean. After several attempts, the cable was completed and service was attempted. But the cable soon failed; Dr. Wildman — in his infinite practical wisdom — had continued to increase the power as the signal weakened and the cable failed. This outcome was something that Thomson’s “school fiction” could have told him.

The cable was the source of an even more striking example of the power of analytical techniques. In many places, the cable was two miles below the ocean's surface, and, since it was susceptible to breakage, the issue of repair was non-trivial. How do you find a break in a cable that is on the ocean floor? Obviously, one could not hope to locate breaks by visual inspection. The answer is an easy application of mathematical analysis, given a cable of uniform construction and knowing the resistance per unit length of cable.

This replacement of physical measurement and observation with symbolic manipulation and prediction is a hallmark of later physical science and scientific engineering. The case for analytically trained engineers was made, and it was made on a practical basis not on some theoretician’s whim.

Correspondingly, the education of new engineers required less and less hands-on training and more and more mathematics. England and Europe seemed to be catching on, but there was substantial resistance from traditionalists in the US. The most telling reflection of this is the fact
that in 1920 there was still a debate whether or not engineering students should be required to learn calculus!

**Engineering Mathematics**

When we think about engineering mathematics we usually think of the calculus as the starting point. But the calculus is the frosting, not the cake in modern science. Without a symbolic language for general mathematical ideas, the originators of the calculus would have been hard-pressed to make their advance. The fundamental breakthrough occurred approximately one hundred years earlier with the invention of symbolic algebra.

In prior ages, algebra had been either tied to a natural language, rhetorical format, or to a strict geometric interpretation, or had been partially symbolized in what’s called syncopated algebra. In the late 16th century François Viète in France, and Thomas Harriot in England, began moving algebraic ideas into a symbolic notation. With the development of symbolic algebra, Newton in England and Leibniz in Germany had the language they needed to symbolize the calculus. Algebra was still tied to numbers, but that too was soon to change.

On the basis of symbolic algebra, modern mathematics followed two courses: applications in the mathematics of the continuous, and applications in the mathematics of the discrete. The particular domain of interest in discrete mathematics was the theory of equations: given a polynomial, \( f \), for what values of \( x \) does \( f(x) = 0 \) hold? Progress here went two ways: (1) higher order polynomials — quadratic, cubic, and quartic equations soon fell, but quintics were problematic, and (2) the values acceptable for \( x \), the domain for \( f \), they changed from naturals, to rationals, to reals, to “imaginary” numbers.

As this process continued, symbolic algebra became more abstract — more “imaginary.” It became more a study of structure and less about particulars, with perhaps Galois’ investigation of the quintics being the turning point. Rather than just examine the structure of the domain of equations, he also had to examine the structure of the language that manipulated that domain. In a similar vein, Hamilton freed the language of algebraic operations from their arithmetical properties with his work on quaternions. Boole continued that abstraction, in his search for a symbolic language for human reasoning.

By the mid-19th century, the mathematics of the continuous was in full bloom, and discrete mathematics had freed itself from numerical domains and operators. Symbolic algebra was becoming a symbolic language. Mathematicians now felt free to explore the general manipulation of discrete symbol systems. This culminated in the work of Frege.

Gottlob Frege’s *Begriffsschrift*, a symbolic notation for concepts, became what we now call Predicate Calculus. What is left of symbolic algebra is the notion of a finite symbol system with rules for manipulating symbol strings.

Around the same time that Frege was working on a general language in which to express concepts, problems were arising in the application of the calculus. This portion of the story is well known: applications of the calculus to new domains gave unexpected results; proof techniques...
that seemed sound gave spurious proofs when applied in new domains. Continuous mathematics was having what has been called a foundational crisis. More precise definitions of the notions of limit followed, but questions about the scope and effectiveness of the methods remained. What were those methods?

It is typical in mathematics and science to reason from hypotheses to conclusion using informal, but precise and convincing, reasoning. But the foundational problems seemed to be saying that these techniques were not as trustworthy as had been expected. Around 1900 David Hilbert proposed applying formal symbolic manipulation techniques to these informal tools that people had been taking for granted, replacing informal reasoning with counterparts in formal manipulations that mimicked the informal.

In more detail, let the notation \( \Gamma \vdash \neg P \) represent the assertion that \( P \) is a conclusion that can be drawn informally from the set of hypotheses, \( \Gamma \). This represents the state of mathematical reasoning of Hilbert’s time. Now assume that we can express \( \Gamma \) and \( P \) in a formal language. We will notate the formal versions of \( \Gamma \) and \( P \) as \( \hat{\Gamma} \) and \( \hat{P} \). Furthermore we assume we can express the reasoning techniques represented by \( \vdash \) as a collection of re-write rules taking strings of formal symbols into strings of formal symbols. We denote this formal rewriting process by \( \vdash \), which is an abbreviation for \( \hat{\vdash} \).

Hilbert hypothesized that a \( \Gamma \) could be created that would contain a sufficient set of facts about the natural numbers, and a formal language could be constructed such that \( \hat{\Gamma} \) faithfully encrypted those facts. Moreover, for any assertion, \( P \), concerning the natural numbers, one could replace the question of “does \( \Gamma \vdash \neg P ? \)” with the question “does \( \hat{\Gamma} \vdash \hat{P} \)?” Furthermore he surmised that there should be a mechanical procedure that could answer yes or no to the question “does \( \hat{\Gamma} \vdash \hat{P} \)?” and therefore give us answers to “does \( \Gamma \vdash \neg P \)?”

Once Hilbert had staked out the problems, answers began to appear. A version of Frege’s calculus appeared to be an appropriate symbolic language. Dedekind and Peano isolated what appeared to be a sufficient set of properties of the naturals, and that set appeared to translate well into Frege’s language. Subsets of the arithmetic of natural numbers were shown to fall within the net that Hilbert cast. But then Kurt Godel appeared. In a stunning paper he showed that there must be situations where, letting \( \Gamma_{nat} \) represent the informal axioms of natural numbers, we may have \( \Gamma_{nat} \vdash \neg P \) and yet, using the accepted symbol systems for translation it cannot be the case that \( \hat{\Gamma}_{nat} \vdash \hat{P} \). This is usually called Godel’s First Incompleteness Theorem. Though the result was important for Hilbert’s plan for formalizing mathematics, more important to us is an intermediate result of Godel: his Representability Theorem.

To understand the Representability result we have to look deeper into the issue of translation. It’s a straightforward problem to translate \( \Gamma_{nat} \) and \( P \) to \( \hat{\Gamma}_{nat} \) and \( \hat{P} \). This makes \( \hat{\Gamma}_{nat} \) and \( \hat{P} \) statements in a formal language that talks about numbers. Godel saw a way to translate \( \hat{\Gamma}_{nat} \) and \( \hat{P} \), essentially representing \( \Gamma_{nat} \) and \( P \) as numbers themselves. This encoding, now known as Godel numbering, is a case of “non-abstract” data structures. Formulae and sets of formulae can be encoded as Godel numbers and relationships among formulae, such as \( \vdash \), can then be encoded as relationships among numbers.

What is the content of this symbol manipulation? First, it is a reasonably complex programming task. In fact several people have referred to Godel as a “programmer” and even as “possibly the world’s first hacker.” But this misses the point. Godel not only had to write the program, but also had to show that his program was correct—that his program met his specifications. And what are the specifications? Well here is the relationship he established:

**The Representability Theorem**

For any relation $R$ in a certain class there is a corresponding $\llbracket R \rrbracket$ such that if $R(a_1, \ldots, a_n)$ holds informally $(\Gamma_{\text{nat}} \vdash R(a_1, \ldots, a_n))$ then $\llbracket \Gamma_{\text{nat}} \rrbracket \vdash \llbracket R \rrbracket (\llbracket a_1 \rrbracket, \ldots, \llbracket a_n \rrbracket)$

So the specification involves the informal, $\Gamma_{\text{nat}} \vdash P$; and the implementation involves the formal, $\llbracket \Gamma_{\text{nat}} \rrbracket \vdash \llbracket P \rrbracket$. And the Representability Theorem asserted that the implementation met the specification. This was 1931, long before hardware computation.

Likewise the foundations for modern programming languages were also set long before hardware arrived. Contemporaneous with Godel, the American logician Alonzo Church was also working on Hilbert’s dream. Church was attempting to invent a complete language for mathematics. His attempted solution turned out to be inconsistent (it had to be because of Godel’s result) but he was able to salvage a portion of his work—the portion that dealt with what he called “effectively calculable functions.”

Church’s notion of effectively calculable function also answered the third part of Hilbert’s quest—is there a mechanical procedure for answering the question “does $\Delta \vdash P$?” He was able to parlay Godel’s Incompleteness result into a negative answer: in general one cannot construct such an algorithm if $\llbracket \Gamma_{\text{nat}} \rrbracket$ is a consistent system. And since $\llbracket \Gamma_{\text{nat}} \rrbracket$ is supposed to mirror the informal mathematics of natural numbers, he could further assert that such a decision procedure cannot exist if mathematics is consistent.

Church’s formalism was named the lambda calculus and in later years became the foundations of functional programming. Those “later years” began to happen in the late 1950s when John McCarthy utilized the lambda calculus as a basis for Lisp. McCarthy mapped the meta-expression language of Lisp onto S-expressions, the data of Lisp, just as Godel mapped the language of formal number theory onto numbers. McCarthy was also instrumental in articulating the need for proving properties of programs, both in the sense that a program must meet its specification, and the question of equivalence of implementations.

In parallel with these developments was Alan Turing’s work relating the notion of effective computability to mechanical processes. Turing’s work is more widely known and is the basis for the familiar Von Neumann architecture; however, a Turing machine is not a helpful model from a software point of view. For that we look to England in the early 1960’s.

Contemporary with McCarthy are the English computer scientists Peter Landin and Christopher Strachey. Landin’s abstract machine, SECD, supplied sufficient detail that the practitioners could use it as an implementation model, while the theoretician had reason to believe that one could

---

prove properties of those implementations.

Strachey, like McCarthy, raised the flag for mathematical foundations of computation; his emphasis was on the semantics of programming languages. Can we create a symbolic language that will allow us to mathematically model the characteristics of a realistic programming language that includes such notions as state-change and mutable data structures? Landin was able to demonstrate that Church’s lambda-calculus was sufficient to model functional programming languages, and thus complete the trio of programming language, mathematical semantics, and realistic machine, thereby offering the hope of proving properties of programs as well as proving properties of implementations.

The work of these individuals and others in the 1960s represents the beginnings of a mathematical foundation for Software Engineering.

**Software Education: Craft, Science, and Engineering**

The approach to teaching programming at most American universities is depressingly craft-based. The knobs and gears (syntax) of a specific tool (programming language) are presented and the students are drilled in the use of the tool in labs. Sometimes higher-level tools are used, program development environments that assist in the management of syntax and program structure. Tools are useful; they make the accomplishment of a task easier, whether that task is constructing a building or a software system, but they don't eliminate the need for careful analysis and specification. Far too little effort is expended on learning the techniques that support existing tools and preparing the students to build their own tools.

We must distinguish education in the foundations required to design a building or software system from training in a construction trade. We teach traditional engineers the mathematics they need to analyze problems and specify solutions. We require traditional engineering students to spend 1 1/2 to 2 years studying the mathematics of the continuous, calculus and differential equations. These are the ideas that support traditional engineering’s tools. We must do the same for software engineers. Yet many schools require only one course in discrete mathematics for software engineering students. And few courses provide an in-depth investigation of the ideas that support programming languages — the tools of software engineering.

Software engineering, and to a large extent software science, is in a similar condition to that of traditional engineering and education in the early 20th century. Now we are faced with transatlantic cable-like software problems. We are inundated with viruses and security issues, to say nothing about low-quality software. The shop culture of software engineering has run out of gas.

Is there an alternative? Or one might rather ask: is there the seed of an alternative. Much like someone in 1750 might see calculus as a seed, we suggest that there is, and it is based on the work of Godel and Church and others. Applied thirty years later by McCarthy, Landin, Strachey, and others, it is being brought to practicality in forward-thinking universities and corporations. There is an alternative to the shop-based production of software. There is a path to professionalism in software engineering. And there are alternatives to shop-based education and
training in the universities. Over the past twenty years, we have developed such a course at Santa Clara for the graduate degree in Software Engineering.

The course is titled “Truth, Deduction, and Computation.” It deals with these three ways of interpreting formal languages — their proof theory, their model theory, and their mechanical evaluation. Three basic classes of language come into play: the propositional languages, their predicate enhancements, and the language of computable functions. We examine the relationships among the languages and among the means of interpreting them.

But this is not a course in mathematical logic; neither is it a course for computer scientists. It’s designed for software engineers. So implementation techniques and potential applications are as important as the theoretical interrelationships among the languages and their interpretations. Therefore, the students see interpreters for propositional evaluation, applications of predicate languages to logics of knowledge and action, and abstract machines that realistically portray the run-time structures that underlie modern language design. And of course they also see proof techniques applied in situations that mirror what is expected in the verification of a claim that programs are equivalent or that a program meets its specification.

Our approach is not the only way to incorporate these ideas into the computing curriculum, and a single course will never suffice. Many others have recognized the importance of theory to support the analysis and design required to engineer software. For example, researchers such as Gries, Manna, Waldinger, Parnas, Barwise, Etchemendy, Hoare, Dijkstra, Constable, Chandy, Broy, and Pratt, have expounded on the importance of theory in software engineering education.

Is there resistance to this approach? Development of a theory-grounded "school culture" has always been resisted by the craft-based "shop culture." Remember the US engineering educational system in 1920. It took two hundred years for engineering education in the U.S. to embrace calculus. Perhaps we will be quicker to embrace foundations for software engineering.

Bibliography


McCarthy, John. “A Basis for a Mathematical Theory of Computation.”
http://www-formal.Stanford.EDU/jmc/


Biographical Information

RUTH E. DAVIS holds Bachelors and Masters degrees in Mathematics, and a Ph.D. degree in Information Sciences. She received the ACM Doctoral Forum award for outstanding Ph.D. thesis in Computer Science in 1979. She is currently Professor of Computer Engineering at Santa Clara University, with research focused on formal methods in software engineering, and increasing the diversity of the engineering workforce.

JOHN ALLEN holds Bachelors and Masters degrees in Mathematics. He did graduate work at the University of Chicago and Stanford, and research in Computer Science at Stanford’s Artificial Intelligence Lab. He taught at the University of California at Los Angeles and at California State University at San Jose. He teaches occasionally at Santa Clara University. His interests are in the mathematical foundations of programming languages.