Whither Software Engineering

John R. Allen
18215 Bayview Dr, Los Gatos, Ca
011+(408)554-4358
jallen@engr.scu.edu

ABSTRACT

We address the disparity between the intellectual preparation that is expected in traditional engineering as compared to that accepted in software engineering. Any beginning student of a traditional engineering discipline realizes that their first courses will be steeped in mathematics–calculus and physics in particular. These foundational tools underlie the practical aspects of their future career. At best a software engineering student will begin with a similar program; but such courses are the stuff of software applications, not of the business of software per se. We examine the history of traditional engineering and the corresponding transformation of educational expectations from a shop-culture to a school-culture. From the origins of symbolic algebra in the late 16th century, through calculus and mathematical physics, the basic sciences that support modern engineering were developed. Shadowing this progress, the educational establishment was struggling with how–or whether–to move the new theory into practice. We all know how that struggle turned out. We argue that a similar pattern must occur in software development, not because of some academic whim but because the complexity of software demands that we expect higher standards. The critical problem in modern software is predictability: we need to know what to expect when we run a program or import software from the net. Such expectations are ill-served by current techniques. At best, programs are conjectures, free of justifications and supplied "as-is." In this day of the virus such a cavalier attitude is indefensible. We will outline some mathematical foundations for software and illustrate their application to the interplay between program and specification. Many of these ideas are the result of early 20th century philosophers; ideas that developed into a constructive logic, and from there to a mathematical foundation for programming languages. The process that moved traditional engineering from an experience-based craft to a science-based discipline was a multi-century revolution. Thomas Kuhn's *"The Structure of Scientific Revolutions"* explored a similar process in the advancement science. In the final section we review his arguments for science revolutions; adapt them for traditional engineering; and then show how our proposed revolution in the engineering of software falls within this framework.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification – *assertion checkers, formal methods.* D.4.6 [**Operating Systems**]: Security and Protection – *security kernels.* F.3.1 [**Logics and Meanings of programs**]: Specifying and Verifying and Reasoning about Programs – *logics of programs, pre-and post-conditions, specification techniques*. K.2 [**Computing Milieux**] – *people, theory.*

General Terms

Security, Languages, Theory, Verification.

Keywords

constructive logic, education, history of engineering, school-versus shop-culture, software engineering

1. A History of Traditional Engineering

To understand where Software Engineering should be going it is informative to study the history of traditional engineering, when and how it became science-based, and therefore when and how engineering education became science-based. Traditional engineering migrated 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 "school-engineering" is the province of the universities.

1.1 Rise of Professionalism in Engineering

Imagine for a moment that you are an entering student in a reputable 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 machine shop? Or Electrical Engineering began with a course in basic household wiring?

Though this seems absurd nowadays there's plenty of historical precedent for this "shop culture" approach in these engineering disciplines. But as theory developed and showed its benefits in practical applications such apprenticeship-driven hands-on programs were replaced by what is 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 confined to labs to augment and motivate the theory.

This transformation of engineering education was the last step in a long revolution that began with changes in attitude toward science and mathematics. Aristotle had staked-out a synthetic theory of science predicated on (1) deductive reasoning, (2) beginning from self-evident principles which had been (3) empirically discovered.

But Copernicus, Brahe, Kepler, and Galileo represented a change in attitude; no longer to be satisfied with observation of nature, but to view science as an analytic venture in which to experiment and predict physical behavior. Prediction could be as simple as informal reasoning based on observation; it could involve experimentation with, and measurement on, mechanical models. The most widely recognized tool to support the changed attitude in the 18th century was the calculus of Newton and Leibniz. Acceptance grew through the 19th century with the adoption of school-culture engineering in Europe and England, but then bumped into resistance in the 20th century United States. For as late as the 1920s, the educational establishment in the U.S. questioned the necessity for calculus in the engineering curriculum. Even in post World-War II America, 30% of its engineers had not attended college.

1.2 Mathematical Foundations of Engineering

When we think of engineering mathematics, we usually think of the 17th century calculus of Newton and Leibniz. But their 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 advances. The fundamental breakthrough occurred approximately one hundred years earlier with the invention of symbolic algebra.

It is a common misconception that what we call "algebra" had been around for millennia when the calculus came on the scene. Actually symbolic algebra was a recent arrival. In the late 16th century François Viète in France, followed by Thomas Harriot in England, began to move algebraic ideas into a symbolic notation. In prior ages, algebra had been either (1) rhetorical–as word-problems in a natural language, (2) geometrical–tied to a strict geometric interpretation, or (3) partially symbolized by Diophantus in what's called syncopated algebra.

Of course these categories are not mutually exclusive: in pre-symbolic times the process of solving a rhetorical problem would also be expressed in natural language; the solution might also be constrained by geometric considerations. Syncopated algebra contains the rudiments of symbolic unknowns, and many before Viète extended these ideas. But there's more to Viète's work than just the use of symbolic unknowns. Viète introduced coefficients as symbolic parameters, so rather than solving a specific problem one could now express solutions to a general class of problems.

Even so, the introduction of parameters is only a symptom of a more powerful vision. Prior to Viète those "unknowns" were simply abbreviations for as yet undetermined numbers. With his introduction of parameters he could now think of equations as expressing general relationships, and with that he introduced the first equational algebra. He made the first formulation of general rules for manipulation of equations. This led him to what he called "The science of correct discovery in mathematics" involving a refinement of the ancient technique of analysis.

From the time of Plato, reasoning had proceeded in one of two ways. Analysis is the style of reasoning that assumes the desired conclusion, and then works back to accepted premisses, each step justified by logical principles. Synthesis is the alternative to analysis; it involves reasoning in the style of elementary geometry–from accepted assertions to conclusion using convincing argument. Viète concentrated on the notion of analysis, adding to its original two phases–called Zetetics (the problem formulation), and Poristics (the transformation by convincing argumentation)–a third phase he called Exegetics. Exegetics is the science of interpretation–the interpretation of the symbolic result as an answer to the original problem. Viète's three-part innovation takes on a very familiar form:

$$\begin{tikzcd}
\text{problem} \arrow[r, "\text{Zetetics}"'] \arrow[d, dashed] & \text{(symbolic algebra)} \arrow[d, "\text{Poristics (equational transformation)}"'] \\
\text{solution} & \text{(symbolically isolated variable)} \arrow[l, "\text{Exegetics}"']
\end{tikzcd}$$

As originally envisioned by Viète, Zetetics is the transformation of a problem into an algebraic equation; Poristics, the transformation of the equation into equivalent (but simpler) form resulting in a value for the unknown. And Exegetics is the interpretation of the numeric solution as an answer to the original problem. But the key element in all cases is symbolic algebra; first, to formalize the problem; second, to transform the formal statement using algebraic rules; and third to interpret the result using the ontological content of the problem domain.

Symbolic algebra thus offered a new medium for prediction: symbolic reasoning, based on the transformation of symbolic representations. If we can effectively symbolize essential characteristics of a phenomenon and capture informal manipulations as symbolic transformations, then we can replace much experimentation and observation with formal manipulations. We only need to verify that a symbolic result is consistent with the natural phenomenon, even though the path to the result may pass through steps that violate Aristotle's requirement of empirical evidence. Our simple commutative diagram of Viète's work is a microcosm of what transpires now in any modern scientific analysis. So as we entered the 17th century, the pieces were coming together–the beginnings of scientific discovery and symbolic algebra. The stage was set for the calculus and with that, modern science and scientific engineering was within sight.

1.3 Engineering: From Craft to Discipline

The curiosity of Galileo and Kepler converged with the algebra of Viète giving rise to the calculus of Newton and Leibniz. With that there was a mathematical tool that allowed one to symbolize physical phenomena, apply symbolic manipulations, and arrive at testable conclusions. Mathematical physics was born. At first these investigations were applied to large-scale cosmic phenomena; but that was soon to change.

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 of engineering. A major component of pragmatic engineering 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 experience is replaced with classroom 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 engineers, 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 flowed from the artillery school at Mézières. Gaspard Monge, and several other 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 current engineering undergraduate. Yet when Monge described his result in 1768 the technique was immediately classified as a military secret and Monge was not allowed to discuss it publicly 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.

Later, Fourier developed more sophisticated tools for mathematical physics and engineering, moving from the cosmic and continuous to the worldly and discontinuous. His work in 1807 on heat opened up new practical areas while setting the stage for a major confrontation on the foundations of the calculus.

Geographically, the transformation of engineering from a hands-on practical trade to a mathematically-based profession migrated from France to Germany; by the early 19th century Germany was following France's lead in education. By mid-century, professional engineering education was becoming accepted in England. And at the end of the century, the United States finally began the slow process of transforming their shop-based approach into school-based programs.

During the 19th century the engineering field itself had expanded. By the late 1800s 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. 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) in the machine shop.

The real break with the hands-on approach came with electrical engineering. With electricity, one no longer could depend on immediate physical information. Since measurement was indirect, mathematics became essential to assure effective and safe application. Even so, much of early electrical engineering was pragmatically determined. For example, in 1883 Edison was still designing his electrical systems by building scale models and experimenting with various gauge wire to find an acceptable design. His assistant, Frank Sprague, had studied electrical engineering at the U.S. Naval Academy and was able to utilize the "school approach" to produce the optimal solution that was much superior to Edison's *ad hoc* experiments.

But the development of the Trans-Atlantic cable around 1860 put the shop-versus-school conflict in stark contrast. The first models of the telegraph (around 1835) were seat-of-the-pants affairs. Marconi proceeded this way, while Michael Faraday and William Thomson applied mathematical physics to the understanding of telegraphy.

As the practitioners formulated their plans to lay a cable between Ireland and Newfoundland, Thomson and Faraday predicted that the performance of the cable would be unacceptably poor. But Cyrus Field, the entrepreneur in charge, would have none of this. His engineer, a medical doctor Wildman Whitehouse, described Thomson's work as "a fiction of the schools" and proceeded to lay 2350 miles of cable on the bottom of the Atlantic Ocean. After several attempts, the cable was completed and service was attempted. But the signal weakened as Thomson had predicted. To compensate, Whitehouse increased the power. The cable soon failed–also as Thomson's "school fiction" could have predicted.

The cable was the source of an even more striking example of the power of mathematical techniques. In many places the cable was over two miles below the ocean's surface, and since it was susceptible to internal failure the issue of repair was non-trivial. One could not hope to locate breaks by visual inspection; but given a cable of uniform construction, Thomson knew there was a simple computation that would locate a failure.

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

1.4 Summary

So though traditional engineering appears a staid and sedate discipline today, it has a background at least as rambunctious and ill-behaved as what we currently see in Software Engineering. Today's ill-mannered, low-quality software inundates us with viruses and bugs. Perhaps the shop culture of software construction has run out of gas. Perhaps we can no longer afford to ignore a mathematical approach to software engineering. But then we have to address several questions: what is a viable candidate for a mathematical foundation, and can we identify a critical application that cannot be addressed effectively without that foundation? In the following sections we will outline a mathematical basis and will argue that the issue of security is as definitive for Software Engineering as the Trans-Atlantic cable was for Electrical Engineering. Can we draw some parallels between the experience of traditional engineering and what we should expect to see in Software Engineering's future? And finally what is a "school culture" for Software Engineering education?

2. WHAT *IS* SOFTWARE ENGINEERING?

The phrase "software engineering" was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering.

NATO Conference Report on Software Engineering, 1968

Unfortunately the report is short on both theoretical foundations and practical discipline. It has plenty of discussion of "manufacture" and "craft," but not of science or theory. The report, like the current field, deals with software *management*, not engineering *per se*. The 1969 NATO Conference attempted to address these omissions, but in the process also exposed the gulf between the shop practitioners and the school theoreticians. If anything the gap has become more pronounced in the intervening 40 years.

So how can we remedy this situation? The pattern we illustrated for traditional engineering shows modern engineering flowing from symbolic sciences, which in turn flows from symbolic mathematics. We will demonstrate a comparable path here, outlining the theoretical foundations of software engineering after visiting its underlying mathematics.

2.1 Mathematical Foundations for Software Engineering

The limits of my language mean the limits of my world.
Ludwig Wittgenstein, 1922

As with traditional engineering mathematics, symbolic algebra is at the foundation of software engineering. One difference is the domains to which the algebra is applied: continuous in one case; discrete in the other. But the more intriguing distinction involves the issue of language itself.

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. Initially, symbolic algebra was still tied to numbers, but that was soon to change. As experience with symbolic algebra matured, algebra became more abstract, becoming more a study of structure and relations and less about particulars, with perhaps Galois' investigation of the quintics being the turning point. Rather than just examining the structure of the domain of equations he also had to examine the structure of the language that manipulated that domain. More directly, William Rowan Hamilton freed the language of algebraic operations from their arithmetical constraints with his work on quaternions. Boole continued this trend with his symbolic language in which to express propositional reasoning.

But within 19th century continuous mathematics, language was also becoming an issue. The rather cavalier algebraic manipulations used by engineers and mathematical physicists, when treating infinitesimals or infinite sums were under assault. The calculus as conceived by Newton and Leibniz was a pragmatic business; it was useful because it worked. It worked for physics and it worked for engineering. In some cases the results were well-deserved, in others they were valid results based on invalid steps. But as long as the final answer meshed well with measurable predictions, the process was accepted–even though some of those results might have been the product of faulty logic.

The first signs of discord came not from the mathematicians or mathematical physicists but from philosophers. One of the most influential attacks on the *status quo* came from Bishop George Berkeley. The content of his argument isn't relevant here; what is relevant is that his attack struck home. The purists' response took the high road: the calculus should be justified in the style of Euclid's geometry. That involved axioms and rules of inference for synthetic development to balance its analytic application. This would address the definitional issues of limit, differential, and integral, giving rigor to the informal notions of infinitesimal, fluxion, and quadrature.

But this foundational enterprise was also pragmatically driven. Adventurous individuals were beginning to apply the calculus to domains that were not as well behaved as the planetary motion that drove early applications. In particular, Fourier had developed an extremely powerful tool–the "Fourier series." No one could doubt the efficacy of his methods; yet many could–and did–question the means by which he derived them, and raised concerns about their range of applicability. The new element here involved the manipulation of infinite series. Yet the derivation of some of Fourier's most valuable results utilized invalid rules–like the interchange of summation and integration. So from pragmatic concerns, people had to know the range of applicability of Fourier's results.

Between the direct approach by Boole and Hamilton and the indirect as typified by Weirstrauss' epsilon-delta definition, the issue of symbolic language was being brought to a place of importance comparable to that of the application. Symbolic algebra was becoming a general 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*–meaning a symbolic notation for concepts–became what we now call Predicate Calculus. What is left of symbolic algebra is the notion of a finite system with rules for manipulating symbols. Numbers have just become particular instances of symbols. But as correspondence between Frege and David Hilbert reveals, such axiomatizations were still guided by empirical knowledge. The point was to eliminate intuition and ontology from the process of deduction, while distilling informal knowledge into axioms particular to the specific discipline.

As with many previous practical enterprises there were philosophical objections to the program proposed by the likes of Hilbert and Frege. The Dutch mathematician and philosopher, L. E. J. Brouwer, crystallized the discomfort felt by him and his French contemporaries like Poincare and Borel. This lead to the philosophical thread called Intuitionism. In practical terms, Intuitionism involves the distinction between discovery and justification: between the creative process of formulating and validating a conjecture versus the potentially more well-traveled road that will convince others of its correctness. This may help explain how an analysis of Intuitionism developed into a constructive logic, which became a foundation for Software Engineering.

Around the turn of the 20th century, 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 manipulation that mimicked the informal. Hilbert used such techniques to investigate areas as diverse as geometry, psychophysics, and insurance mathematics.

There's a long tradition for "model-based reasoning," stretching back to antiquity. The innovation of the post-16th century period was that symbolic models could replace mechanical models, and symbol manipulation could replace physical manipulation of those models. Hilbert's program effectively mimicked Viète's paradigm to analyze the very tools of mathematics by using formal systems themselves. Hilbert called this *metamathematics*.

In more detail, let Γ |~ P express the assertion that P is a conclusion that can be drawn informally from the set of hypotheses, Γ. If P=>Q represents the manipulation of P into Q, then we expect that if Γ |~ P holds then Γ |~ Q does also. This kind of invariance (here of |~) with respect to transformation (here =>) will play a critical role in the foundations of Software Engineering.

Following the Italian mathematician Peano, Hilbert realized that a specific Γ–call it Γ_{nat}– could be isolated to characterize sufficiently the properties of natural numbers, and a formal language could be constructed such that formal version, $\boldsymbol{\Gamma}_{\mathbf{nat}}$, of Γ_{nat} could be constructed. In such formalization, symbols called numerals are the formal counterparts of natural numbers.

Furthermore we assume we can express the reasoning techniques represented by |~ as a collection of inference rules –call them |- – that take sequences of formal symbols into sequences of formal symbols. Then for any assertion, P, about natural numbers, we replace the question "does Γ_{nat} |~ P?" with the question "does $\mathbf{\Gamma_{nat}}$ |- **P** ?" Furthermore Hilbert conjectured that a mechanical procedure could answer "does $\mathbf{\Gamma_{nat}}$ |- **P**?" and therefore answer "does Γ |~ P?" The idea being that we can replace questions about numbers with questions about numerals; and results derived about numerals would carry weight when viewed as answers about numbers. Expressed *a la* Viète, we have:

$$\begin{tikzcd} ?\ \Gamma_{nat} \mid\sim P? \arrow[r, "\text{Zetetics}"] \arrow[d] & ?\ \mathbf{\Gamma_{nat}} \mid- \mathbf{P}\ ? \arrow[d, "\text{Poristics}"] \\ \text{No / Yes} & \bot / \top \arrow[l, "\text{Exegetics}"] \end{tikzcd}$$

But then Kurt Gödel, in a *tour de force* of language analysis, showed that there must be situations in which Γ_{nat} |~ P for some P, and yet it cannot be the case that $\mathbf{\Gamma_{nat}}$ **|- P**. This is the content Gödel's First Incompleteness Theorem. Though the result was important for Hilbert's plan to formalize mathematics, more important to Software Engineering is an intermediate result of Gödel: his *Representability Theorem*.

To understand the Representability result we have to look deeper into the issue of translation. As mentioned, it appeared to be straightforward to translate Γ_{nat} and P to $\mathbf{\Gamma_{nat}}$ and **P**, making $\mathbf{\Gamma_{nat}}$ and **P** statements in a formal language that talked about numerals. But Gödel saw a way to represent $\mathbf{\Gamma_{nat}}$ and **P** as natural numbers. This encoding, now known as Gödel numbering, is a case of seriously "non-abstract" data structures. Formulae and sets of formulae can be encoded as Gödel numbers and meta-relationships among formulae, such as |-, can then be encoded as relationships among Gödel numbers.

But Gödel numbers can themselves be represented as numerals, and metamathematical questions–like the provability of formulae–can be represented as formulae in the language of $\mathbf{\Gamma_{nat}}$.

The first part of this symbol manipulation is an intricate, but reasonably simple, programming task. In fact several people have referred to Gödel as a "programmer" and "possibly the world's first hacker." But this misses the point. He not only had to describe the encoding, but also had to show that his encoding was correct–that, so to speak, his program met his specifications. It is more accurate to call Gödel the first Computer Scientist.

Behold part of Gödel's *Representability Theorem*: For any relation R in the class of relations over natural numbers called "effectively constructible" (EC) relations, there is a corresponding formal **R** such that for natural numbers, a_i and numerals $\mathbf{a_i}$:

if R(a_1, ... , a_n) holds informally (i.e., Γ_{nat} |~ R(a_1, ... , a_n)) then its formal counterpart is deducible (i.e., $\mathbf{\Gamma_{nat}}$ **|- R($\mathbf{a_1}$, ... , $\mathbf{a_n}$)**)

So the specification involves the informal (that Γ_{nat} |~P); and the implementation involves the formal (that $\mathbf{\Gamma_{nat}}$ |- **P**). And the Representability Theorem asserts that the implementation satisfies the specification. This was established in 1931, long before hardware computation.

The notions of computation and algorithms predated Gödel of course. But like Viète did for algebra Gödel brought new life and perspective to an old subject. Within a few years two new computational threads appeared: Alan Turing's machines, that epitomized what was to become the stored program machine; and Alonzo Church's λ-calculi, that held the kernel notions for functional programming languages.

2.2 From Intuitionism to Constructive Logic

However, not withstanding this rejection of classical logic as an instrument to discover mathematical truths, intuitionistic mathematics has its general introspective theory of mathematical assertions, a theory which with some right may be called intuitionistic mathematical logic ... L. E. J. Brouwer, 1954

To move the notion of a program from the land of unjustifed conjecture to the safe harbor of defensible assertion, we need to understand the idea of "convincing argument." This idea is well-known from mathematics but the usual logic that sustains mathematical conviction is not sufficient support for a science of software. Mathematics is usually satisfied with what's called *classical logic*, which in the simplest case is a logic of two truth-values: true and false. For example this simplifies the notion of convincing argument by allowing proof by *reductio ad absurdum*: to show that a statement, P, holds assume P is false and demonstrate a contradiction. Therefore P must hold since the opposite of false is true.

Even though classical logic is quite powerful we need a more subtle logic for Software Science and Engineering. Rather than just the simple value, true, as evidence for the truth of an assertion we need a notion of evidence that describes how to arrive at–or construct–that truth-value. This distinction is at the core of our discomfort we feel when told that (12*6)/3 = 24 versus being told that 24=24; there's more information of the left than the right and our logic of software needs to address this discomfort. The expression on the left–like Viete's parameters–makes explicit the origin of the right-hand number. However classical logic does possess a property that is very important for our treatment of logical software: *compositionality*, meaning that the value of a complex assertion is a combination of the values of the component assertions. Of course our logic of evidence should be a compositional logic. In hindsight it is a small step from this logic to a justifiable programming language where the evidence becomes the program; the assertion becomes its justification.

But as with most "small steps" the path was difficult. This logic of evidence, or constructive logic, ironically springs from the philosophical work of L. E. J. Brouwer and his concern for the very non-logical processes by which mathematicians create. He objected some of the very techniques that characterize mathematical proof; notably *reductio ad absurdum* (RAA)

The controversy is most clearly seen in the treatments of an existential assertion and its simpler cousin, the disjunction. If we wish to assert something of the form ∃x.P(x), classical logic will allow us to establish its validity without explicitly exhibiting an element that satisfies P. A typical argument will grant existence by RAA: assume that there is no element satisfying P and derive a contradiction. Or the truth of a disjunction PvQ can be determined by a combination of the truth-values of its components courtesy of compositionality; if either component holds then the disjunction holds. But classical reasoning will allow us to establish a disjunction indirectly by assuming that neither P nor Q hold and deriving a contradiction. For example the classical interpretation says the form $P \vee \neg P$ is true regardless of what P represents, and even if we cannot determine which disjunct holds.

But such classical methods of reasoning are not consistent with the requirements of software. It is not sufficient simply to assert that a program satisfying a given specification *must* exist. To carry any practical conviction we must construct such a program. Most simply put: the logic of programming is the logic of constructions.

In a constructive setting, evidence for an existential assertion $\exists x.P(x)$ is a pair: an individual called a *witness* plus evidence that P holds for the witness. Evidence for a disjunction is also a pair: evidence for one of the disjuncts, plus an indication of which disjunct is supported by that evidence. In a constructive logic, the evidence for any complex assertion is captured as a composition of the evidence associated with each component of that assertion. So in constructive logic, evidence shows the journey, while classical logic only shows the destination–the final value.

Since we assert that this constructive logic is the proper logic for software, we illustrate the notion of evidence for a couple more logical connectives. What's convincing evidence for an implication $P \supset Q$? Classical logic relies on a rather unconvincing truth table. However from our intuition about implication we expect that for $P \supset Q$ to hold there should be a causal relationship between P and Q. We can express that expectation by saying there's a function that will transform evidence for P into evidence for Q. And of course we expect the function is constructive or "effectively calculable" like those introduced by Gödel. For example, evidence for $P \supset P$ is the trivial identity function which indicates that we can transform evidence for P trivially into evidence for P. Finally, given evidence for P–call it E_P–and evidence for Q–call it E_Q–then evidence for $P \wedge Q$ is the ordered pair $< E_P, E_Q>$.

These few instances exemplify the character of a constructive logic. The approach generalizes to cover propositional negation, 2nd-order propositional, and 1st-order predicate calculi. Each of these extensions will have immediate application in the logical description of software.

In slightly different setting, this constructive description is known as the Brouwer-Heyting-Kolmogorov (BHK) interpretation. Heyting, a student of Brouwer, formalized portions of Brouwer's philosophy of mathematics as a logic as did (independently) the Russian mathematician Kolmogorov.

Of course a more explicit view of construction appeared in 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/von Neumann machine is not a helpful model for Software Engineering.

2.3 The Origins of Software Science

It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last. The development of this relationship demands a concern for both applications and for mathematical elegance.

John McCarthy, 1963

The foundations for the science of software are found in the philosophical and logical work of the early 20th century just as the underpinnings for traditional engineering are found in the philosophical and mathematical work of the 17th century. In both cases much had to be done to turn the theory into practice. In both cases the first applications were pragmatic affairs based on genius and insight but often containing logical flaws. The first such software breakthrough was Lisp. In the late 1950s John McCarthy utilized ideas from Church's lambda calculus as a basis for a language with which to manipulate symbolic expressions. With Meta-expression Lisp, McCarthy gave us the first high-level language with "non-flat" data–called Symblic Expressions or S-expressions–replacing numbers and arrays of numbers.

In retrospect we can see Lisp and its "dotted pairs" is a partial realization of the BHK interpretation. The BHK interpretation of $\supset$ becomes the ubiquitous λ-expression; the evidential interpretation of $\wedge$ becomes the equally common trio of car, cdr, and cons. And $\vee$ gives us a variation on McCarthy's conditional expressions:

case obj **of** [<**tag**$_L$ x> $\Longrightarrow$ f(x) | <**tag**$_R$ y> $\Longrightarrow$ g(y)].

Similarly we can map the logic of induction onto the recursive data structure of lists and its corresponding programming construct, **label**. McCarthy also mapped the Meta-expression language of Lisp onto his non-flat S-expressions just as Gödel mapped the language of number theory onto flat Gödel numbers. The practical importance of McCarthy's mapping is well known: Lisp's "McCarthy numbers" are the syntax of the Lisp programming language; and the rest, as they say, is the cdr. In later years some of Lisp's *ad hoc* features have been constrained. The result is an elegant Lisp-like language called Scheme, a Meta-expression subset of which we call BlackBoard Scheme (BBS).

It has long been my personal view that the separation of practical and theoretical work is artificial and injurious. Much of the practical work done in computing, both in software and in hardware design, is unsound and clumsy because the people who do it have not any clear understanding of the fundamental design principles of their work. Most of the abstract mathematical and theoretical work is sterile because it has no point of contact with real computing. Christopher Strachey, 1972

Contemporary with McCarthy are the English computer scientists Peter Landin and Christopher Strachey. Landin's abstract SECD machine supplied sufficient detail that a practitioner could use it as an implementation model while the theoretician had reason to believe that properties of those implementations might be proveable. Landin also demonstrated that Church's lambda-calculus could model a realistic functional programming language–one he called ISWIM. Strachey did fundamental–as well as practical–work on the semantics of programming languages.

Not to be outdone, Floyd, Hoare, and Dijkstra had brought mathematical logic to some of the machine-oriented constructs of imperative languages. They related computational constructs to logical properties in a way that reflects the state-oriented behavior of imperative languages. These individuals and others, working from the computational side in the early 1960s, established the beginnings of a logical foundation for Software Engineering.

But in late 1960s the logicians got back into the game they'd started more that 30 years before. One of the first major indications of the future collaboration came in late 1969 when

Dana Scott–in conjunction with Strachey–announced the first mathematical model for the type-free λ-calculus. From that followed the Scott-Strachey view of denotational semantics for programming languages. Earlier in 1969 Scott circulated an influential paper on a typed computational formalism. This paper begat Milner's LCF (Logic for Computable Functions), which in turn begat the typed programming language ML. Prior to that, in late 1968 Scott published on the semantics of Intuitionistic logic. Following a lead of Kreisel from 1962 Scott expanded on the early work of Heyting and Kolmogorov. Scott's results along with the underground work of William Howard, led to what's known as the Curry-Howard Isomorphism.

Specifically the isomorphism tells us that if the term is well-typed then its type is a theorem of a constructive logical calculus. The correspondence makes itself most apparent when we view the logical inference rules compared side-by-side with the analogous rules for well-typed expressions. For example behold the formal version of the logical rule, Modus Ponens, and its corresponding rule for well-typed function application:

Logic Rule	**Computation:type Rule**
$\dfrac{\Gamma \mid\!- P \supset Q \qquad \Gamma \mid\!- P}{\Gamma \mid\!- Q}$	$\dfrac{\Gamma \mid\!- f{:}\alpha \rightarrow \beta \qquad \Gamma \mid\!- a{:}\alpha}{\Gamma \mid\!- f\,(a){:}\beta}$

where "**:**" is read as "has type" and the logical connective ⊃ maps to the type relationship, →; and the propositions P and Q become types α and β.

This gives rise to a treatment of constructive propositional calculus interpreted as type-structure for an explicitly typed ML-like functional language, one we will call BlackBoard ML (BBML). BlackBoard ML is a typed version of BlackBoard Scheme.

As in BBS we have computation**:**type-rules that map the conjunction ∧ to the Cartesian product ⊗ and map the disjunction ∨ to the disjoint union ⊕. For example:

Logic Rule	**Computation:type Rule**
$\dfrac{\Gamma \mid\!- P \qquad \Gamma \mid\!- Q}{\Gamma \mid\!- P \wedge Q}$	$\dfrac{\Gamma \mid\!- a{:}\alpha \qquad \Gamma \mid\!- b{:}\beta}{\Gamma \mid\!- \text{cons}\,(a, b){:}\alpha \otimes \beta}$
$\dfrac{\Gamma \mid\!- P \wedge Q}{\Gamma \mid\!- Q}$	$\dfrac{\Gamma \mid\!- a{:}\alpha \otimes \beta}{\Gamma \mid\!- \text{cdr}\,(a){:}\beta}$

$$\dfrac{\Gamma \mid\!- P \vee Q \quad \Gamma, P \mid\!- R \quad \Gamma, Q \mid\!- R}{\Gamma \mid\!- R}$$

$$\dfrac{\Gamma \mid\!- a{:}\alpha \oplus \beta \quad \Gamma, x{:}\alpha \mid\!- b{:}\,\delta \quad \Gamma, y{:}\beta \mid\!- c{:}\delta}{\Gamma \mid\!- \textbf{case}\ a\ \textbf{of}\ \textbf{[}\langle \textbf{tag}_L\ x{:}\alpha\rangle \Longrightarrow\ b \mid \langle \textbf{tag}_R\ \ y{:}\beta\rangle \Longrightarrow\ c\textbf{]}{:}\delta}$$

As we mentioned in the prior section this interplay between logic and type carries over to more complex programming constructs: a discussion of negation explicates exceptions conditions; 2nd-order propositional calculi rationalize the notions of parametric polymorphism and abstract data types, while 1st-order predicate calculi handle dependent types.

The final theoretical piece we need for our security applications is again courtesy of some early 20th century work in logic. The relationship, called *Subject Reduction*, was first noticed by Haskell Curry in the 1920s. In programming vernacular, this says "if a term Y with type ξ, evaluates to Z, then Z must also have type ξ." Note the similarity to one we mentioned earlier: If Γ |~ P and P => Q, then Γ |~Q.

We began this discussion by highlighting the interplay between an assertion and its supporting evidence; that morphed into the notion of an assertion coupled with a construction, or program. We now take the final step and view an assertion as a type and look at the program-assertion pair as a program-type pair. The idea of a "typed programming language" dates back at least to FORTRAN. But what's new here is that the Curry-Howard Isomorphism allows us to view the type information as logical information within a coherent logical system.

Now (finally) we can begin to reap the harvest practical benefits from a logic of software. For example if a programming language supports Subject Reduction then we are assured that the statically determined type will be maintained throughout the dynamics of computation. It tells us that the type of the result of a computation will be the same as the type of the original expression.

A simple type system allows us to specify simple properties like "this argument must be a **boolean**" or "this list may only contain natural numbers." Using the underlying logic we can prove things like "this program produces a value of type **integer**," or "the type of a reversed list of naturals is a list of naturals." In a polymorphic setting we can define more generic types like the variable-type αlist that can be specialized for various flavors of α. In both of these cases the type is independent of the computational process. But we can be even more demanding, refining list-objects on the basis of their size: **natlist (3)** indicates the type of three-element lists of natural numbers.

Type systems in which a type may depend on a computational term are called *dependent type systems*. Such type systems are more complex; but we can now say more interesting things like: "the length of a reversed list is the same as that of the original list" or "the length of the result of appending two lists is the sum of the lengths of those two lists.

With more complex type systems, we are factoring more static information out of the dynamics of computation. And since we expect that the more complex language supports Subject Reduction, then this static information is preserved across evaluation. Most intriguing for security, dependent type systems allow us to state assertions like "this program doesn't violate array bounds." Given the devastating effects of viruses, security failures like "array bounds violation" have the potential to be a "forcing event" similar to what the failures of the transatlantic cable were to traditional engineering. Perhaps our epidemic of security concerns is the opening we need to shift software from a shop- to a school-culture.

3. FROM FOUNDATIONS TO SOFTWARE ENGINEERING

...the United States is not a culture, but merely an economy, which is the last refuge of an exhausted philosophy of education.
Neil Postman, 1993

We've seen that developments in mathematics and changing attitudes toward science fueled the development of theory-based traditional engineering. Symbolic algebra coupled with the realization that many physical behaviors are predictable, led to the development of scientific engineering. Now we use design-rules based on mathematical physics and not wishful thinking and conjecture to assure predictable behavior in physical structures. We need a similar advancement in design rules for logical structures, thereby moving software construction into the realm of software engineering. Rather than make conjectures about program behavior we need languages that allow us to predict behavior prior to execution. We believe that sufficient theoretical and pragmatic results exist to present them as a basis for software engineering, particularly in that most critical area of software security.

3.1 A Security Problem: Buffer Overflow

It is almost daily news that some anti-social individual has exploited what's called "buffer overflow" to install a virus. Buffer overflow occurs because an "array-bound violation" has not been detected. This may occur because of the following failures: the programmer wasn't sufficiently observant when coding the problem; or the source language wasn't sufficiently expressive to recognize a violation; or finally in the name of efficiency, the compiled code did not included a run-time array-bound test. If it were only the programmer who suffered, justice would be served. But in these days of networks, imported programs, and mobile agency, then any recipient of the infected code is a potential victim. The usual solution is to run the code and hope for the best. A more enlightened view is to "sandbox" the code, monitoring its execution for potentially damaging behavior. Of course, such monitoring adversely impacts performance. In the future, we should expect to *prove* that a properly constructed program can't contain safety violations like buffer overflow. Such a program could be run unmonitored with confidence.

3.2 A Security Solution: Proof-Carrying Code

The issues that drove security concerns in the early days of operating systems are still with us. In the old days, security involved the interface between a user and an operating system. Hardware and software protected the system from interference by supplying well-defined protocols for system-level requests. In theory such avenues could be used in these days of host/parasite- and agent-interaction; but the large amount of interaction that these behaviors welcome would adversely impact efficiency. A more efficient approach would be to let the parasite introduce code into the realm of the host; but this could be fatal if such code violates the security of the host. To guard against this we can have the host monitor the execution of the parasitic code and verify that no untoward actions occur. Of course this has a runtime penalty. Another possibility is to have the host specify the language in which the parasite must present code. This allows the host to embed its security requirements in the constructs–either as coding conventions or as a type system. That way the host has some assurance that parasite's code is acceptable. A better way to address safety is for the host to make its security policy explicit and require parasites prove their code satisfies that policy. To accomplish this the parasite supplies the code that the host then augments with security requirements; the parasite must then generate a proof that the code meets those requirements. The resulting bundle–called *Proof-Carrying Code* (PCC)–is shipped to the host. The host is equipped with a proof-checker that can verify that the asserted proof is correct, and that the proof corresponds to the parasite's code. If so the host can accept the package and execute the code with confidence and without run-time checks. Of course that confidence is based on the host's competence in articulating an effective security policy. This is a less ambitious goal than program correctness. Here we simply expect that the program "does no harm." But in these days of "wild west" software, that would be a substantial improvement.

The PCC approach allocates responsibility between the parasite and host. The host must be able to articulate a policy; that is as it should be. The parasite must be able to supply a convincing demonstration that its code meets that policy. The host must stand ready to check the proof. And since proof-generation is typically much more difficult than proof-checking, the parasite's job is more difficult than the host's. That too is as it should be. After all it is the parasite that wishes to intrude on the host's domain. Practically speaking the "trusted computing base" (TCB) that the host must supply–the proof-checker–is substantially less complex than the software required by the parasite to generate the proof.

3.3 Languages for Predictable Software

Given a precondition and a postcondition we consider pairs of a program and a proof that the program satisfies the postcondition given the precondition. We call such a pair together with the pre- and postcondition a deliverable, since it is what a software house should ideally deliver to its client ...[that is] *a program plus a proof in a box with the specification printed on the lid.*

Rod Burstall and James McKinna, 1991

The PCC paradigm is a special instance of our more general desire: the notion that programming constructs should contain two pieces of information–one computational and one justificational. Though this idea is contained within the general realms of intuitionistic and constructive logic, it was explicitly addressed by Rod Burstall in the late 1980s. Though this and many of these theoretical ideas can be extended to languages for concurrency, for this discussion, we restrict attention to sequential languages, dividing them into two categories: imperative, and functional. Imperative languages involve constructs to prescribe state-change. Therefore assertions in an imperative PCC language will describe the effects of such change. A language that requires Floyd/Hoare assertions as an integral part of each programing construct could meet such a desire. Thus a program in a PCC imperative language will be a combination of state-change instructions and pre- and post-conditions that embody security-related assertions. The client process here will supply imperative code along with "verification conditions" extracted from the code, coupled with a proof that the verifications conditions hold.

Functional languages, on the other hand, are based on the notion of function application. In a PCC world the types of the parameters to functions are the pre-conditions and the type of the result is the post-condition. The functional language must be designed such that the types are inextricably tied to code. So a program component consists of two parts: a computational expression and a specification expression. We also expect that type-checking is decidable and a practical algorithm for this exists. For ease of programming and clarity of expression a type-inference system is also desirable. Finally, we insist that Subject Reduction holds so that static specifications will be predictors of dynamic behavior.

Of course we need to express realistic security concerns as type constraints in functional languages. And if so, that we have an algorithm to establish that a program is "well-typed" and that the code is consistent with the embedded type assertions. It is well known that the general problem of proof-discovery for complex statements is undecidable. That means there is no algorithm to determine if a proof exists for an arbitrary statement. Now the issue becomes (1) do we have formalisms in which can we express realistic security-policies, (2) can those policies be framed as assertions about the static type-structure of programs, and (3) are there practical algorithms for testing the properties arising from the types associated with programs.

There is actually a third possibility for a language: rather than being a side-effect-free functional language where application and combination are only the structuring devices, function-*based* languages like ML include assignment and side-effects along with abstraction, application, and combination. In such languages, an extension of the Curry-Howard concept, allowing state information to be percolated across inference, holds promise.

Once the basic approach is assimilated, it is natural to explore its application to the more modern areas of concurrency and agency. These are topics of current research interest.

4. WHAT *IS* SOFTWARE EDUCATION?

[A guild's] *technique for protecting the secrecy is by keeping the secret knowledge unformulated; therefore, the apprentice has to join a master for seven meagre years, during which he can absorb the craft by osmosis, so to speak. The university is at the other end of the spectrum: it is the professor's task to bring the relevant insights and abilities into the public domain by explicit formulation.* E. W. Dijkstra, 1996

4.1 The Software Shop-Culture

... but I fear–as far as I can tell–that most undergraduate degrees in computer science these days are basically Java vocational training. Alan Kay, 2004

The saga of software-as-craft is well known. The post World War II years saw the rise of the von Neumann machine and the fixation on numerical calculation. Of course this was just a computational version of the Viète diagram–the translation of a problem into a numerical setting followed by computational, rather than symbolic, transformation and finally a re-interpretation of the numeric result as an answer to the original problem. This picture improved only marginally with the beginnings of higher-level numerical languages that included primitive data structures like numeric arrays and strings. Initial attempts at computational symbol manipulation via list-processing were also primitive. Assembly languages like IPLV only hinted at the potential then hidden within the notion of symbolic calculation. These early days were all about the applications of the new computational tools and their implementations; elegance of expression came in a poor second.

Education and training fared no better. Programming was–and still is–taught as a craft, with all of its attendant drawbacks, but now without a recognizable guild or apprenticeship component that guided the shop-driven programs of the past. That might be acceptable when one looks upon programming as simply a tool for some other discipline but it's unacceptable for those who wish to understand the phenomenon of computation. Our courses teach programming; and usually not just any programming. They teach the latest industry standard language as a job-skill, rather than using an idealized language that illustrates concepts. At best, students may be exposed to proper technique and style, but seldom will they be exposed to the responsibility of programming.

The issues of specification and design also get short shrift. Instead of the logic of software we see courses on the *management* of software. These no doubt are useful tools to control complexity, but they do not address the underlying problem: how to develop software to a specification such that its behavior is predictable. Management tools also reflect a different set of goals–the control of production; we should have learned the drawbacks of *that* approach from engineering's sorry experience with Frederick Winslow Taylor's time-and-motion studies.

We prepare a program of education for automotive engineers to enable them to develop better engines, transmissions, ... for the cars of the future; instead, all of our students want to be taxi drivers. Tony Hoare, 2005

We must distinguish between the education required to design buildings or create software systems from the training needed to construct buildings or to program applications. We teach traditional engineers the mathematics they need to analyze problems and specify solutions. We require traditional engineering students to spend two years studying the mathematics of the continuous–calculus and differential equations. These are the ideas that describe physical reality and support traditional engineering's tools. We must do the same for software engineers. Yet many schools require only a single course in discrete mathematics for software engineering students, ignoring the fact that the problem is not a mathematical one; it's a logical one. It is logic that provides the ideas behind the tools of software engineering. It is logical consistency that provides the reality that constrains software engineering.

The fundamental issue that Software Engineering must confront is whether it is to continue as simply a supplier of employees for the software industry or whether it is to become a free-standing intellectual endeavor. In the latter case, we have to identify content that is independent of transient technology. The elegant interplay between logic and computation as exemplified by the Curry-Howard Isomorphism offers great promise to supply that content.

We've seen that traditional engineering made a successful transition to a science-based profession, and that science itself preceded engineering in making a transition from an experience-based craft to a discipline founded on mathematics; but there is a compelling reason to believe that these are but special instances of more general pattern of progress.

5. THE STRUCTURE OF SCIENTIFIC REVOLUTIONS

Lifelong resistance, particularly from those whose productive careers have committed them to an older tradition of normal science, is not a violation of scientific standards but an index to the nature of scientific research itself. T. S. Kuhn, 1962

In 1962, Thomas Kuhn's "The Structure of Scientific Revolutions" was published. It was not particularly well received since it questioned the current thinking that scientific progress proceeded in an orderly fashion, incrementally refining existing knowledge in an inexorable march toward the truth about natural phenomena. Kuhn argued that the story is more complex; that science moves in what we might now call "punctuated equilibrium," with periods of relative stability followed by a spurt of rapid revolution and re-thinking brought about by the appearance of crises in the current orthodoxy. That orthodoxy, called "normal science," is captured in a "disciplinary matrix." That matrix describes the parameters that define the accepted beliefs, the rules for examining evidence, and techniques that are employed to further the scope of the discipline. But when evidence that does not fit the practitioners' expectations becomes too hard to ignore a crisis results and some individuals look for alternative explanations. The investigation of those anomalies of normal science leads to an unsettled state, opening a phase called "revolutionary science," that would be resolved by a consensus into a new normal science.

While in a period of normality, progress is measured by the solution to unanswered questions posed within the vocabulary of the current disciplinary matrix. Such questions are called "puzzles," something like a crossword puzzle, inferring that one expects to find a solution using the available rules. Questions can yield solutions; can remain unanswered, and thereby pose further research questions; or can require answers that cast doubt on validity of the current matrix. This last category of questions is called "anomalies," and if too many anomalies are found, the accuracy of the current matrix is called into question. Such a situation leads to a "crisis" and a search for a new disciplinary matrix begins.

The canonical example of this process is the development of the heliocentric view of the solar system as an alternative to the geocentric view espoused by Aristotle. There were increasingly sophisticated viewpoints discussed by Ptolemy, by Copernicus, by Galileo, by Brahe, and by Kepler, each of whom supplied a disciplinary matrix–a *Weltanschauung*–that was in conflict with some substantive portion of their predecessor. It is usually the case to terminate this early development with Newton, but one can argue that he unified the work of Kepler and his predecessors, while bringing mathematical elegance to the field. This was no small feat; it was a revolution in itself. That unification through symbolic mathematics turned a revolutionary science into the normal science of the next centuries. The educational institutions, the texts, the research agendas, all became purveyors of the new Newtonian science. And as we've seen, little-by-little mathematical physics became the foundation for a revolution in engineering.

5.1 The Structure of *Engineering* Revolutions

Every failure is an incontrovertible counterexample to the implicit or explicit hypothesis that a design would succeed, and as such each failure contains potentially more valuable information than all the successes that theretofore confirmed that (false) hypothesis. Henry Petroski, 2001

Of course this pattern of orthodoxy, doubt, revision, and revolution is not unique to science. As Kuhn argued, this pattern appears in science as the search for explanation and predication of natural phenomena, but it also appears in religion–the search for meaning in life; in art–the search for aesthetics and beauty; in politics–the search for justice; in academia–the search for tenure. But how does this pattern apply to engineering? One can argue that modern traditional engineering is a direct descendant of applied science as derived from Newtonian ideas. But there's a more direct relationship: both science and engineering progress through failure. In science, nature doesn't fail, theories fail; while in engineering, theories can fail and constructs can fail.

Constructs can fail for at least two reasons: the initial design is faulty, or the actualization of that design is flawed. In the first case, a revolution might be in order; but in the second case, the failure may not impact the underlying science-based assumptions. And in modern engineering, assumptions tend to be science-based. We have standards, building-codes, and design-rules, all of which are grounded on mathematical physics. Of course this was not always the case; witness the repeated failures of the Transatlantic Cable.

The story of the Transatlantic Cable exemplifies what might be called "the structure of *engineering* revolutions." It's an instance pattern of "normal" engineering–experience-based engineering–being challenged by an alternative based on symbolic mathematical physics. This revolution was brought on by "crises" not well-treated by the current orthodoxy. The failures of the cable are but one example. Engineering revolutions are not driven by an academic exercise–an inadequate understanding of nature, but by a real-world threat to the goal of engineering**:** the construction of safe and reliable artifacts.

The pattern of change in science and engineering is similar. The *status quo* resists and revolutionaries insist. The crises give birth to a new "disciplinary matrix," and the emerging paradigm works its way through the educational system, where (hopefully) the revolutionaries hold sway, and have sufficient distance from current practice to effect the necessary changes. But of course this may not happen if the academy is too closely connected with its industrial counterparts. Industry, being conservative, has a tendency to perpetuate the known rather than explore the unknown. When coupled with a conservative academy, the results are not good–witness the reluctance in segments of the American engineering education establishment to require calculus. Of course there are examples where the cooperation between academia and industry was mutually reinforcing; witness MIT and the rise of cybernetics; and Stanford and its silicon spin-offs.

5.2 A Revolution in *Software* Engineering

If we're gonna die, Benny, let's die playin' our own thing.

. Gene Krupa, 1935

Just as traditional engineering has gone through Kuhnian revolutions similar in style to those experienced in science, it should not be unexpected that software engineering should follow suit, and move from its current pre-Copernican state to something more analogous to post-Newtonian sophistication.

Though traditional engineering's revolution took several centuries to complete, we cannot expect to be so patient in building a community to overthrow the current thinking in software engineering. The issues surrounding the failures in security are not isolated incidents like a plane crash or a bridge collapse. Software failures may not be as pictorial as crashes and collapses, but their potential damage is far more wide-spread.

Just to be clear, when we're talking about software and engineering together, we're *not* talking about the use of software to enhance some aspect of a traditional engineering discipline. Rather we're advocating the use of modern techniques to improve reliability in the construction of software. We expect that engineering disciplines have a well-defined mathematical basis whose foundations rest on mathematical physics–the formalization of the current normal science of specific natural phenomenon. We expect that the engineering practice be guided by standards, building codes, and design rules, all of which derive their credibility from those foundations. We should expect no less from well-engineered software.

We've said that software is not nature-based, it's logic-based. It reflects a formalization of one or more forms of constructive logic. Such is the science of software. So what is its engineering, since here we are unable to test our symbolism against the hard reality of nature?

The answer is the good news and the bad. Rather than testing our theories against the–only partially understood–real world, we can describe the symbolic reality that our creation purports to implement. That's why it's the good news and the bad: not only *can* we, we *must.* We must specify what exactly our software is claiming to compute. Even more, we must be willing to defend that claim, supplying a convincing proof that the software meets that specification.

As with Godel's proofs of Representability, those proofs need not be formal, just convincing. We should expect that beginning software curricula give equal time to programming, specification, and proof. We should make it clear to students that a program is at best an unverified conjecture, and without a specification and a convincing proof, it will remain so. Wishful thinking is not engineering.

So just as traditional engineering curricula begin with calculus, someday we should expect that software engineering curricula begin with courses in constructive logic–this being the result of a revolution, supplying a new disciplinary matrix that captures software creation as a proof-based engineering discipline.

6. REFERENCES

[1] Adler, K. *Engineering the Revolution,* Princeton University Press, NJ, 1997.

[2] Bell, E. *Men of Mathematics*. Simon and Schuster, NY, 1937.

[3] Boyer, C. (revised by Merzbach, U.). *A History of Mathematics* Wiley, NY, 1991.

[4] Burstall, R, and McKinna, J. *Deliverables: a categorical approach to program development in type theory*. ECS-LFCS-92-242, University of Edinburgh, 1992.

[5] Calvert, M. *The Mechanical Engineer in America, 1830-1910: Professional Cultures in Conflict.* Johns Hopkins Press, MD, 1967.

[6] Corry, L. *Modern Algebra and the Rise of Mathematical Structures*. Birkhauser, MA, 1996.

[7] Grabiner, J. *The Origins of Cauchy's Rigorous Calculus*. MIT Press, MA, 1981.

[8] Kuhn, T. *The Structure of Scientific Revolutions*, 3rd edition, Chicago University Press, Il, 1996.

[9] Layton, E. Jr. *The Revolt of the Engineers*. Press of Case Western University, Cleveland, OH, 1971.

[10] Lundgren, P. Engineering Education in Europe and the USA, 1750-1930: The Rise to Dominance of School Culture and the Engineering Professions. *Annals of Science, 47*(1990), 33-75.

[11] Mahoney, M. The Beginnings of Algebraic Thought in the Seventeenth Century. *Descartes: Philosophy, Mathematics and Physics*, S. Gaukroger (ed.), Sussex: The Harvester Press, Totowa, NJ, 1980.

[12] Martin-Löf, P. On the Meaning of the Logical Constants and the Justifications of the Logical Laws. www.hf.uio.no/filosofi/njpl/vol1no1/meaning/

[13] McCarthy, J. A Basis for a Mathematical Theory of Computation, http://www-formal.Stanford.edu/jmc/

[14] Mumford, L. *Technics and Civilization,* Harcourt, Brace &Co., NY, 1934,

[15] Necula, G. Proof-Carrying Code, *POPL97*, ACM, 1997.

[16] Petroski, H. *To Engineer Is Human: The Role of Failure in Successful Design.* St. Martins Press, NY, 1985.

[17] Postman, N. *Technopoly: The Surrender of Culture to Technology*. Vintage Books, NY, 1993.

[18] Scott, D. Constructive Validity. *Lecture Notes in Mathematics, Vol 125,* Springer-Verlag, NY, 1968.

[19] Seldin, J. Curry's Anticipation of the Types Used in Programming Languages*, Dept of Math &C.S.*, University of Lethbridge, Lethbrigde, Ca.

[20] Strachey, C. Towards a Formal Semantics, *Formal Language Description Languages for Computer Programming*. T. Steel (ed.), North Holland, 1966.

[21] [19] Viète, F. *The Analytic Art*, trans. by Richard Witmer, Kent State University Press, 1983