From Craft to Science: Searching for First Principles of Software Development

by **Koni Buhrer**
Software Engineering Specialist
Rational Software

Developing large software systems is notoriously difficult and unpredictable. Software projects often are canceled, finish late and over budget, or yield low-quality results -- setting software engineering apart from established engineering disciplines. While puzzling at first glance, the shortcomings of software "engineering" can easily be explained by the fact that software development is a craft and not an engineering discipline. To become an engineering discipline, software development must undergo a paradigm shift away from trial and error toward a set of first principles.

*In this first article of a two-part series for **The Rational Edge**, I will provide support for this theory and propose a first principle of software development, along with a corresponding set of "axiomatic" software requirements. In my next article, I will outline a set of design rules -- the "universal design pattern" -- featuring four types of design elements that, in combination, can describe an entire software system in a single view.*

Crafting vs. Engineering

When I look at the fabulous Gothic cathedrals in Europe, I often wonder: Who were the architects that designed and constructed them? How did they acquire the knowledge to create structures that were not only beautiful, but also sturdy enough to withstand the forces of nature for centuries? How were they able to construct these buildings without computers, hydraulic tools, or modern construction materials?

Clearly, the medieval architects had extraordinary knowledge of building design and construction -- knowledge for which they were highly respected and honored as masters of their trade. And masters they were; but they do not deserve to be called engineers.

Why not? The reason is simple. While the medieval architects had profound knowledge of how to construct tall buildings, they knew nothing about the laws of physics. They knew that the shape of an arched ceiling had to be conical, but they had never heard of differential equations. The knowledge of the medieval architects was based exclusively and entirely on experience from trial and error and did not have any scientific foundation. In a nutshell: The medieval architects knew *how* to do things, but they had no idea *why* things had to be done that way[1]. Building design and construction was a craft, and the medieval masters of architecture were craftsmen and not engineers.

The difference between the medieval master of architecture and the modern construction engineer is that the engineer understands the *reasons* for the architectural rules. He can deduce the rules from the laws of physics and therefore prove the soundness of a design before doing any construction work, even if he has no applicable experience. The medieval architect, by contrast, had to accumulate his knowledge slowly and painfully through trial and error. Eventually he would become a master and pass on this knowledge to his apprentices. Some of the apprentices would in time become masters themselves and pass on their knowledge to their apprentices. And so the medieval craft of architecture slowly matured over time -- all the while remaining a craft.

Building design and construction abruptly advanced from a craft to an engineering discipline when it was based upon the scientific foundation provided by the laws of physics. Maturity alone does not turn a craft into an engineering discipline; engineering requires a paradigm shift. Crafting is based on trial and error, while engineering is based on a set of first principles (often the applicable laws of physics) from which all knowledge can be inferred. Trial and error is the hallmark of crafting; first principles are the hallmark of engineering.

What Is Software Development Today?

With this distinction in mind, let's ask ourselves: Is software development today a craft, an engineering discipline, or something in between? Many software developers would probably assert that software development is not yet an established engineering discipline, but it is well on its way to becoming one.

I think that is a delusion. In my view, software development is pure craft. It can't be an engineering discipline -- and in fact does not embody any aspect of engineering -- because it has no first principles. All knowledge on how to develop software is based exclusively on trial and error. Yes, we have made some progress since the dawn of software development. Masters of software development like Grady Booch, Jim Rumbaugh, and Walker Royce, for example, have invented successful methods and rules, often called *best practices*, to guide the software developer. Yet, like the

medieval architects, software developers have learned and validated those best practices through trial and error. What software development lacks at this time is a set of first principles on which its rules and methods could be based[2].

The lack of first principles[3] of software development has serious consequences, often captured by the phrase "the software crisis." Software projects are often canceled before completion, and the projects that *are* completed are often over budget, late, and yield low-quality results. As others have observed, the key factor for the success of a software project is a good architecture. And the inability to routinely create a good architecture is exactly what distinguishes software development from established engineering disciplines. Why this embarrassing discrepancy? Well, unlike the software developer, the engineer uses a set of first principles to prove the adequacy of a design before any construction work is done. The software developer, on the other hand, must rely on testing to assess the quality of an architecture. Or, to put it plainly, the software developer has to find a good architecture by trial and error.

And that explains why the two most widely seen problems in unsuccessful software projects are "late scrap and rework" and "late design breakage." The software developer creates an architectural design early during software development, but has no means to immediately assess its quality. The software developer has no first principles at hand to prove the adequacy of a design. Testing the software eventually reveals all architectural defects, but only in a late development phase when fixing them is costly and disruptive.

What Is Different about Software Engineering?

The inability of the software developer community to turn software development into a successful engineering discipline has caused quite a bit of embarrassment. The low success rate of software projects clearly sets software development apart from established engineering disciplines. Many reasons have been proposed to explain why software *engineering* should be different from other engineering disciplines. Let's look at three well-known examples.

Every engineering discipline involves elements of art and science. This is true, but it is the scientific element -- not the artistic element -- that distinguishes engineering from craft. And the scientific element is exactly what software development is lacking, because it has no first principles on which any science could be based. Moreover, the artistic element of engineering is negotiable. A bridge designed by a civil engineer may be ugly, but it will always be safe. Software development, however, is all art (or craft), and the quality of a software system depends solely on the artistic capabilities of a software developer[4].

Other engineering disciplines have a long history and lots of past

experience, but software engineering is still in its fledgling state.
Many engineering disciplines do indeed have a long history, because they emerged from an ancient craft. But the transition from a craft to an engineering discipline is not a matter of maturity; it is always the result of a paradigm shift away from trial and error toward a set of first principles. Most modern engineering disciplines underwent this paradigm shift quite abruptly when they became based upon the laws of physics. Software development, however -- because it lacks any scientific foundation -- is still a craft. Although it will mature over time, maturity alone will not turn it into an engineering discipline.

Software projects are so unpredictable because software has unlimited flexibility.
This is nonsense. The truth is that many software developers neither know how to create good software architectures nor know how to spot the bad ones. The lack of first principles in software development makes it impossible to objectively distinguish between good and bad architectures. And as a result, software architectures are often bad. Yet the great flexibility of software -- and it is true that software is very, very flexible -- allows a project to progress a long way toward implementing even the worst architecture. Eventually, however, architectural defects become major obstacles that cause extensive scrap and rework, time and budget overruns, and generally poor software quality. So, it is not the great flexibility of the software that makes software projects hard to manage; it is the inability of software developers to routinely and predictably create good architectures.

Software Development Is Inherently a Design Activity

How did we get into this mess? Why does the spirit of trial and error so deeply permeate software development? Why haven't we explored the fundamental laws of software development and discovered its first principles yet?

To answer these questions, we need to understand that software development is inherently a *design* activity with no aspect of *construction* or *manufacturing*. You may find this claim hard to swallow, but it can easily be justified. We have a good understanding of what design is, where it ends, and where construction or manufacturing starts. Consider the following two arguments:

1. The boundary between design

Puzzled by Software Development Issues? There's a Simple Explanation!

The fact that software development is inherently a design activity helps to explain a number of otherwise puzzling issues. Here are some examples:

Q. Why is it much harder to create a detailed work breakdown structure for software implementation than for skyscraper construction?

A. Because software implementation is the equivalent of skyscraper design (i.e., the creation of the

and construction is always clearly marked by an artifact: the blueprint. Design encompasses all the activities needed to create the blueprint; construction encompasses all the activities needed to create products from the blueprint. In a perfect world, the blueprint would specify the product to be created in every detail -- which of course is rarely the case. Still, the purpose of a blueprint is to describe the product to be constructed as precisely as possible. Does the architectural design of a software system describe the software product "as precisely as possible"? No way. The architectural design is intended to describe the essentials, but certainly not all the details of a software system. So the architectural design is clearly not a blueprint. Only the high-level language code describes all the details of a software system, and thus qualifies as the software's blueprint. And because all activities leading up to the blueprint are design, all software development must be design.

2. The effort (time, money, resources) needed to create a commercial product can always be divided into design effort and manufacturing effort. What is the difference? The design effort is common to all copies of the product and has to be expended only once. The manufacturing effort has to be expended every time a copy of the product is created. A software product is typically the binary executable of a program delivered on a CD-ROM. Clearly, the effort to create the source code of a program -- including architectural design, detailed

blueprints for a skyscraper) and not skyscraper construction. Design work is naturally less amenable to planning than construction work -- which is true for skyscrapers as well as for software -- because the scope and complexity of the end product are discovered only in the course of the design work.

Q. Why can we implement two software modules in parallel, whereas the floors of a building have to be constructed sequentially?

A. Because the equivalent of implementing a software module is designing -- and not constructing -- a building floor. And two architects can very well design two building floors in parallel, as long as they observe certain interface constraints -- just as two software developers can implement two modules in parallel. The compiler, on the other hand -- which truly constructs the software product -- must compile the modules in the right sequence, just as building floors have to be constructed in the right sequence.

Q. Why can't we achieve economies of scale in software development?

A. The diseconomy of scale we see in software development is inherent to its design nature. We know from other industries that economies of scale apply only to manufacturing processes but not to design tasks. For good reason: Design not only has to deal with each of the design elements, but also with all interactions between the design elements.

design, and high-level language code -- has to be expended only once, no matter how many copies of the software are produced. Consequently, the effort to create the source code of a program is entirely a design effort, and all software development is design.

Software developers do not construct software; they *design* software. The final result of that design effort -- the high-level language code[5] -- is the blueprint of the software. And it is the compiler/linker that mechanically *constructs* the software product -- a binary executable -- from the high-level language code. The architectural design of a software system most closely corresponds to the cardboard models or design sketches used in some engineering disciplines.

Now, to understand why software developers have not sought and found first principles yet, let's imagine a world where creating a skyscraper would require no more than a detailed blueprint. With the blueprint, an architect could, at the push of a button, have the skyscraper constructed -- instantly, and at virtually no cost. The architect would then test the skyscraper and compare it to the specifications. Should it collapse or not pass the test, the architect could have it demolished and the rubble removed -- instantly, and again at no cost. Would this architect spend much time on formally verifying that the skyscraper design complies with the laws of physics? Or even try to explore and understand those laws? Hardly. He could probably get results more quickly by constructing, testing, and demolishing the skyscraper a couple of times, while fixing the blueprint each time around. In a world where construction and demolition are free,

The number of interactions between design elements rises with the number of elements in a non-linear fashion. This is true even with extensive reuse and commercial off-the-shelf components. Software development, which is pure design, will therefore never achieve true economies of scale.

Q. Can we profit from economies of scale in software *manufacturing*?

A. No, because software products are manufactured by the compiler, the linker, and other operating system tools at a largely negligible cost. The sales price of a software product thus only has to cover its design cost, but practically no manufacturing cost. In other industries it is the manufacturing cost of each copy of the product that substantially decreases if more copies are produced. This does not apply to software products.

Q. Why does software have such a low level of reuse?

A. In most industries, reusing existing design elements has a double benefit. It speeds up system design and decreases manufacturing cost. Electronics components, for example, are so popular because they are cheap -- a result of mass production. Using off-the-shelf electronics components thus lowers the traditionally high manufacturing cost of electronics systems. Unfortunately that doesn't apply to software, because software is manufactured at a largely negligible cost already.

trial and error is the approach of choice, and basic research is for suckers.

Well, software development happens in exactly such a world. The software developer creates a blueprint of the software in the form of a high-level language program. He then lets the compiler and the linker construct the software product in the blink of an eye, at virtually no cost. Yes, creating the blueprint requires considerable effort, but construction by the compiler and the linker is practically free. And the software developer certainly doesn't worry about demolition and removal of debris -- at least not until he runs out of disk space. No wonder that the trial and error spirit is deeply entrenched in the software development process, and that the software developer community has not bothered to explore the first principles of software development.

In fact, trial and error has brought us a long way. But with the increasing complexity of modern software systems we are approaching a hard limit. Beyond a certain level of complexity it becomes impossible to create quality architectures by trial and error[6].

Proposal: A First Principle of Software Development

Without first principles, software development will remain a craft forever, and the software crisis (cancelled projects, or results that are over-budget, late, and of poor quality) will persist. So, let's go out and find some first principles -- but where? The laws of physics don't apply to software. Does that mean first principles of software development do not exist? Or have we just not looked carefully enough to find them?

Let's consider how the presence of a first principle -- say, the law of gravity, for instance -- affects the architecture of a system. From an abstract viewpoint, a first principle does nothing more than induce non-negotiable requirements, which the architect has to deal with. I will call those requirements *axiomatic* requirements. For example, in construction a simple axiomatic requirement would be, "The building shall withstand the force of gravity." What makes axiomatic requirements so special is that they apply to each and every system that ever has been and ever will be constructed. Axiomatic requirements are in fact so obvious and common that they are usually implicit. But from the axiomatic requirements, established engineering disciplines have derived a set of architectural rules -- rules all designs must comply with to be worthy of consideration. Designs that violate these rules are obviously at odds with the first principles and are probably faulty.

If it is too hard to find the first principles of software development, it is easy enough to find some axiomatic requirements -- requirements that apply to each and every software system. Here is the list I'd like to propose:

1. The software must obtain input data from one or more external (hardware) interfaces.

2. The software must deliver output data to one or more external (hardware) interfaces.

3. The software must maintain internal data to be used and updated on every execution cycle.

4. The software must transform the input data into the output data (possibly using the internal data).

5. The software must perform the data transformation as quickly as possible.

That sounds brain-dead, doesn't it? But believe me: Very few software systems have an architecture that *obviously* satisfies the axiomatic requirements above. Many software systems may implicitly satisfy them, but not obviously![7]

The axiomatic requirements above seem to be related to the hardware environment of a software system to some degree. I would therefore postulate that the underlying first principle of software development is: "Software runs on and interacts with hardware -- hardware that has only finite speed."

That the hardware environment should play such a dominant role in software architecture is a little surprising. Many software design approaches seem to ignore the hardware environment in the logical, structural, and dynamic views of the architecture. Hardware is often treated as a deployment issue with little significance for architectural structure and behavior. I, however, believe that the hardware environment should be a primary driving force for the architectural design of a software system.

Next Step: Defining a Universal Design Pattern

In my next article for *The Rational Edge*, slated for the January issue, I will focus on the design rules associated with the first principle and its axiomatic requirements. These rules represent a highly useful, universal design pattern to guide software design. The universal design pattern is a first step toward making software development a predictable, repeatable engineering activity -- so don't forget to come back!

[1]More precisely, the medieval master had no means of formally justifying it. For example, if an apprentice would ask the master, "Why do I have to do it like this?" the master would answer, "Because I said so!" A modern engineer, by contrast, would answer, "Well, according to the laws of static mechanics. . . if we take this formula. . . " and so on, thus providing a formal justification of why things have to be done in a certain way.

[2]Some might argue that the best practices constitute the first principles of software development. But unfortunately that is not the case. The reason is that we do not know whether those best practices are actually good practices. We don't have any understanding of *why* a software development practice deserves to be called good or bad -- other than that it proved to be successful in some past application. Identifying best practices by trial and error might eventually lead to deeper understanding of the first principles and mechanisms that determine the quality of a software development practice. But by itself, identifying best practices does not advance software development past its current stage of a craft.

[3]To be clear, first principles are **not** guiding principles. First principles are well-defined axioms that allow formal verification of software development rules and procedures.

[4]Let me elaborate. If asked whether a design was sound, an engineer might answer, "I applied the formulas of static mechanics to verify it." A software developer, however, would answer, "I'm confident, because it looks good" -- much like an artist would judge a sculpture.

[5]Or the detailed design if the compiler can generate code directly from the design models.

[6]Iterative development only expands the limit of complexity that can be mastered by trial and error.

[7]A popular design approach is to identify nouns in the requirements documents and associate a class with each noun. System functionality is modeled by messages the classes exchange. This design approach invariably results in software architectures that make verifying the axiomatic requirements very difficult.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!