The Nature of Software

—

What's so special about software engineering?

Philippe Kruchten

Rational Software Canada
638-650 West 41st Avenue
Vancouver, BC, V5Z 2T9 Canada
E-mail: pbk@rational.com

International Conference « The Sciences of Design », Lyon (France), March 15-16, 2002

1 Software engineering?

> *Engineering, medicine, business, architecture and painting are concerned not with the necessary but with the contingent — not with how things are but with how they might be — in short, with* design*.* (Simon, 1996)

Software development slowly evolves from some form of elaborated craftsmanship to become a real software engineering profession. This evolution is not new (Shaw, 1990; Ford & Gibbs, 1996), but oh! so very slow. Although the term 'software engineer' has been pre-empted long ago and adorns the business card of many software developers, the evolution of this industry to a real level of engineering is not easy. Today (in 2001), software development is 40% hacking (code, deliver, let the user test), 20% craftsmanship (lots of oral tradition, and 'death march' projects), and at best 20% engineering.

At the same time, it is obvious that we need engineered software, as software becomes ubiquitous and more and more is at stake: life-critical applications (avionics, medical instrumentation, air-traffic control, etc.), or high monetary costs (financial systems, telecommunications, integrated enterprise systems, etc.). "Error at PC 4AE5H, please reboot" is not an acceptable outcome any longer.

As engineering organizations across North America struggle with the concept of opening their doors to and registering — or even licensing — software engineers, questions naturally arise about what software engineering actually entails (Pour, Griss, and Lutz, 2000; McConnell, 1999). How do we qualify and evaluate software engineers? How do we validate their experiences? A first reaction may be to approach this task in the same way that we have done for all other engineering disciplines. However, software engineering *differs* from structural, mechanical, or electrical engineering in subtle ways. The differences are linked to the *soft*, but rather unkind, nature of *soft*ware. In this article, I shall explore four key differentiating characteristics:

- Absence of a fundamental theory
- Ease of change
- Rapid evolution of technologies
- Very low manufacturing costs

to which I may be tempted to add that "software knows no borders." I will then look at two areas where a strict transposition of approaches that worked in other engineering disciplines have failed so far in software:

- The development lifecycle
- Component-based development

and show some alternatives.

2 Key Characteristics of Software

Design in the world of bits is very different from design in the world of atoms. Bits are fluid, have no weight, can be replicated as low cost, and the techniques to organize the bits evolve very fast. We are right in the heart of the "science of the artificial". But I am mostly speaking about engineering, here, not about science.

2.1 Absence of a Fundamental Software Theory

> *"As we succeed in broadening and deepening our knowledge - theoretical and empirical - about computers, we shall discover that in large part their behavior is governed by simple general laws, that what appeared as complexity in the computer program was to a considerable extent complexity of the environment to which the program was seeking to adapt its behavior."* (Simon, 1996)

We hear you, doctor Simon, but we are still looking. Despite all the research done by computer scientists, there is no equivalent in software for the fundamental laws of physics. This lack of theory, or at least the lack of practically applicable theories, makes it difficult to do any reasoning about software without actually building it. During design, software can be structured and partitioned into chunks, but the real thing (once it crawls inside a computer) is actually totally unstructured, so that anything that goes wrong somewhere can corrupt something somewhere else. In other words, even with the advent of the *object-oriented model* of software, we are still faced with the *Turing or Von Neumann model* of running software (Dryja, 1995). The absence of solid and widely applicable theory also means that the few software engineering standards we do have rely on good practice alone, whereas building codes in other disciplines can trace their rules to sound physical principles. And there are very few software engineering practice standards.

Information theory, despite its name, can account for only a small portion of the type of issues, problems and design decisions we are faced in the construction of software-intensive systems. Formal methods are still confined to research labs, and small subsets of very complex (and expensive) systems — far from a routine practice.

2.2 Ease of Change

Software is, almost by definition, easy to change, so naturally, organizations want to take advantage of this characteristic. There is pressure to change software throughout its entire development and even after it's delivered. If you're building a bridge, you don't have this kind of flexibility. You cannot say, "Hmm, now that I see the pilings, I would like this bridge to be two miles upstream." But it is very, very difficult to change software in a rigorous fashion, with all ramifications of all changes fully understood and completely coordinated. Software is even the element that must be changed or adapted to palliate the lack of flexibility of other engineering elements: "This chip has a defect, we cannot produce a new silicon in less than 3 months, let us ask the software folks to change their thing to work-around this issue." was I told a few years ago.

Again, because of the absence of solid theory, it's hard to validate a change set and its impact without actually doing all the changes. Most of the damage that is done to software is done through changes. Changes are making software degrade, very fast. So ease of change is a great blessing, and a plague at the same time.

2.3 Rapid Evolution of Technologies

Software development techniques, and the environment of software itself, are rapidly changing, at an extreme pace that does not allow for progressively consolidating a body of knowledge. This puts a lot of pressure on companies to train and re-train their software engineers, and some do not really understand why they have to spend four times more per capita on training than do people from other disciplines. This makes the initial training software engineers have received less important, except for the most general education, such as math and so on — especially when this original training occurred twenty-five or more years ago. This rapid evolution also means that it is more and more difficult to maintain and evolve "legacy" systems — as the recent Y2K scramble has demonstrated — because the technologies used some ten years ago are not in use any more, and people who still have mastery over them are rather rare. The norms and standards also have to evolve rapidly to catch up with technology evolution. Finally, software engineering, unlike other disciplines, has not had the benefit of hundreds or thousands of years of experience. This also explain the scarcity of software engineering standards.

2.4 Very Low Manufacturing Cost

First, I would like to note a slight shift in paradigm. Software engineers speak about design, but by this they mean only a high-level description of their intent, and then they think of program construction, as akin to manufacturing. Actually it is not — program implementation is more like preparing a cast in mechanical engineering. Also, for a software engineer, a prototype is roughly equivalent to a scale model; it's pretty incomplete. The real "manufacturing" of software entails almost no cost; a CD-ROM, for example, costs less than a dollar, and delivery over the Internet only a few cents. Often it doesn't matter if the design — that is, the initial program — is a bit wrong; we can just fix it and manufacture it again, as we noted above in the discussion on ease of change.

We hear people refer to this as a "free bug fix release" or a "must-have upgrade." Clearly, this combination — ease of change and low manufacturing cost — has led the software industry into a pretty big mess. And these practices are supported by outrageous licensing policies that allow the designer and manufacturer to assume no responsibility other than, in good cases, a promise that they will re-manufacture the product in a few days or months. You can't do that with a bridge or a car engine because the cost would be huge, and that forces engineers involved in building these things to get them right the first time.

3 Engineering All the Same?

For twenty years, refusing to acknowledge the four factors I described, the software industry has tried hard to pretend that software development could follow the same path as other engineering disciplines. We have failed. We also hoped that science would bring us solutions, but they have not been forthcoming. Why? To answer that question, let's look at two important ways that software engineering departs from other disciplines.

3.1 The shape of the lifecycle — Iterative Development

> *"… both the shape of the design and the shape and organization of the design process are essential components of a theory of design."* (Simon, 1996)

The very rational and straightforward "waterfall" development lifecycle — define and freeze requirements, create and validate the design, implement and test, then deliver — works very well in many disciplines but has failed many times in software engineering. This project lifecycle does not accommodate changes: It does not allow you to really validate much, so you have to rely on your own warm, fuzzy feeling that "the design is OK." Nor does it allow for tactical changes in technology, or take advantage of the low manufacturing cost — except for pushing undue costs on to consumers.

Today, software engineers take a more iterative approach to software development, which allows them to integrate changes, to refine and validate the design based on execution and not just examination, and to accommodate evolution in technology. An iterative approach would be impossible in other disciplines; you cannot build a bridge iteratively, for example. The first to push iterative development to a large scale was Barry Boehm (Boehm, 1988) with his spiral model, and it is now embraced fully by modern software development approaches (Rational, 2001; Kruchten, 1999). Still today, iterative development (because we can) is pooh-poohed by other engineering disciplines (probably because they can't), and its adoption is slow.

Iterative development allows you to continuously verify the quality of a constructed prototype as opposed to demonstrating correctness a priori, based on fixed laws. Software has no laws that can ensure the ultimate product will perform as expected, but iterative development allows you to confront technical risks *earlier* in the development cycle.

Also, software engineering puts more emphasis on some techniques that have less importance in other disciplines, such as requirements management and change management, because requirements and other software artifacts may change throughout the development lifecycle and even after that. They are usually pretty stable in other types of engineering endeavours.

Herbert Simon again:

> *"If you look at any really complex engineering or architectural design you find that the goals are never completely defined until the design is almost finished. At any time in the process of designing you can say, 'There has to be enough space there to allow that door to swing open. So I'll have to set*

> *that as a new constraint and make sure that condition is satisfied.' The very process of design reminds you of new conditions that have to be satisfied.*
>
> *A characteristic of design that is special to it, besides this gradual emergence of goals, is that the largest task is to generate alternatives. There are lots of theories of decision making, a field that has been heavily cultivated by economists and statisticians. But most theories of decision making start out with a given set of alternatives and then ask how to choose among them. In design, by contrast, most of the time and effort is spent in generating the alternatives, which aren't given at the outset.*
>
> *Of course generating alternatives and choosing among them aren't isolated from each other. The process of design is a continual cycle of generating alternatives and testing to evaluate them. The idea that we start out with all the alternatives and then choose among them is wholly unrealistic. If you are designing an important bridge, you might consider two or three basic kinds of bridges and choose one, then go to the next level of detail, and so on. Throughout the design process you are always generating two or three alternatives and choosing among them, and then setting the values of specific parameters to fit the application at hand."* (Simon, 1996)

If you do not limit yourself to the narrower sense of the word "design" used today by software folks, but include in design any activity of software development that include looking at alternatives and making choices — such as: requirements analysis and coding — then you see that Simon's wisdom does apply and implies an iterative approach. However, in other disciplines than software, weeding out the bad alternatives can be done by pure reasoning, using models and laws, or relying on long established practice without going too far in the realization, and at a very small fraction of the cost. You only *consider* the 2 or 3 types of bridges; you do not have to build them to eliminate the wrong ones. Or the wrong ones have been built, have collapsed, and we know about them (e.g.. the collapse of the bridge on the St. Lawrence in Québec city in 1907, or — on film — the famous Tacoma Narrows bridge in 1940).

3.2 Component-Based Development — Software Architecture

> *"… it is typical of many kinds of design problems that the inner system consists of components whose fundamental laws of behavior … are well known. The difficulty of the design problem often resides in predicting how an assemblage of such components will behave."* (Simon, 1996)

Another dream of software engineers is to mimic with software what has happened in electronics or construction — to develop families of standardized parts out of which you can build larger and larger sub-assemblies and ultimately complete systems. This sounds straightforward, and the natural step "beyond programming" (Blum, 1996), but actually very few can do it, despite announcing it for 20+ years. Again, this is due to the lack of a strong underlying theory to rigorously define the components and especially their interface. So we are still far from the situation described by H. Simon, and the difficulty is even greater in software. We do not know the fundamental laws of behaviour of the components, and are mostly unable to describe what happens at the interface, so we are very very far from being able to predict the behaviour of the

composite. The situation has not been helped by the rapid changes in technologies: No component family has had time to settle and develop as a self-sustaining industry. Who is still developing new ActiveX components?

Only recently has a new sub-discipline emerged — *software architecture* — which tries to address, despite the lack of fundamental theory, some of the structural aspects involved in constructing large software programs and reusing software assets across product lines or product families (Shaw & Garlan, 1996).

> *"… complexity frequently takes the form of hierarchy and that hierarchic systems have some common properties independent of their specific content. Hierarchy … is one of the central structural schemes that the architect of complexity uses."* (Simon, 1996)

In software we have too long thought about only *one* decomposition hierarchy, and that this hierarchy is an extension of the system's hierarchy. This was even engraved in stone in widely applied standards, such as DOD-STD-2167A or software configuration standards. Software architecture has taught that there are several intertwined hierarchies, depending on the viewpoint, and that the designer can play on several hierarchies at the same time (IEEE, 2000; Kruchten, 1995). Herbert knew: *"There is not reason to expect that the decomposition of the complete design into functional components will be unique. In important instances there may exist alternative feasible decompositions of radically different kinds."* (Simon, 1996). But "near decomposability" (Simon, 1996) remains a convenient illusion, in software, at the level of our object-oriented blueprints; all the bits end up in the same machine. And there, anything can go wrong.

As we make progresses in the modeling of software, we are now very close to *model-driven design*, where some amount of reasoning can occur on a model whose semantics can be well defined, and out of which code is generated more or less automatically (Soley, 2001). Not there yet, but closer. We just need to add a little bit of science.

4 Conclusion

So, if I agree with David Parnas that software engineering should be treated on an equal footing with other engineering disciplines and not solely as computer science or some kind of enlightened craftsmanship (Parnas, 1999), then I also have to acknowledge fundamental differences that make some of the more traditional approaches to engineering and engineering management inapplicable to software. Software engineering has to develop its own approaches to manage its specificities over the last few years. Iterative development is one, and model-driven development is around the corner.

As the software engineering profession matures, let us not fall again and again in the trap of trying to mimic or force-fit approaches that did work elsewhere, and let us looks at what makes the true nature of software, what distinguish its object, totally artificial, from that of other disciplines, closer to the natural sciences. It is engineering, but not quite the same.

There is one aspect, which I left aside: *software knows no border*. Bits have no weight. This ultimate characteristic of software is the one that is today dramatically changing this industry. It does not matter much whether software is specified in Stockholm, designed in Cupertino, coded in Bangalore, tested in Warsaw, and used in New York. The radical differences in costs and quality will create some imbalance, out of which the industry of software can only grow better: a push for quality, consistency, reliability; a way out of the "software crisis" of the last 20 years, which has led to poor quality and almost total irresponsibility. (I am not pointing at anybody in particular.) Software development will remain a craft for a long time, but we can foresee an inversion of the ratios; the hacking will vanish, the craft will be 40%, the engineering 60%. And we, engineers, will take any help from our colleagues on the computer science side, as they close the gap.

Biographical notice

Philippe Kruchten is director of process development at Rational Software.

He has about 28 years of experience of software development in telecommunication, aerospace, defense, command and control systems, and software tools. Like civil engineers in the 19th century, who would go where big bridges and dams were built, he gathered this experience around the world, going anywhere large software projects would take him: France, UK, Sweden, USA, Australia, and Canada. Half of that time—14 years—were with Rational Software, as a technical consultant, software architect, and development manager. He is currently in charge of the development of the Rational Unified Process®, an electronic software-engineering handbook (Kruchten, 1999; Rational, 2001). He has a diploma in mechanical engineering and a doctorate degree in information systems from French institutions. He is based in Vancouver, British Columbia, and is a registered professional (software) engineer in this province of Canada.

References

Boehm, Barry (1988). "A Spiral Model of Software Development and Enhancement," *IEEE Computer*, 21 (5), May 1988, pp 61-72.

Blum, Bruce I. (1996). *Beyond Programming—To a new era of design*, Oxford University Press, New York, NY.

Dryja, Michael (1995). "Looking to the Changing Nature of Software for Clues to its Protection," *University of Baltimore Intellectual Property Law Journal* 3 (2), pp.109-144.

Ford G, and Gibbs N. (1996). *A Mature Profession of Software Engineering*, Technical report, Software Engineering Institute, Pittsburgh, Penn., SEI/CMU-96-TR-004.

IEEE (2000). *Recommended practice for architectural description of software intensive systems*, IEEE standard 1471.

Kruchten, Philippe (1995). "The 4+1 view model of architecture," *IEEE Software*, 6 (12), pp. 45-50.

Kruchten, Philippe (1999). *The Rational Unified Process—An Introduction*, Addison-Wesley-Longman, Reading, Mass.

Kruchten, Philippe (2000). "Putting the Engineering into Software Engineering," *Innovations*, 4 (1), January 2000, pp.23-24.

Kruchten, Philippe (2001). "The Nature of Software: What's So Special About Software Engineering?" *The Rational Edge*, October 2001,

http://www.therationaledge.com/content/oct_01/f_natureOfSoftware_pk.html

Lord, Thomas (2001). *The Nature and Economics of Software Research and Development*, http://regexps.com/nature/nature.html

McConnell, Steve (1999). *After the Gold Rush: Creating a True Profession of Software Engineering*. Microsoft Press.

Parnas, David (1999) "Software Engineering programs are not Computer Science programs ," *IEEE Software*, December 1999, 19-30.

Pour, Gilda, Griss, Martin, and Lutz, Michael (2000) "The Push to Make Software Engineering Respectable." *IEEE Computer*, May 2000, 35-43.

Rational Software (2001). *Rational Unified Process v2002a*, http://www.rational.com/products/rup/index.jsp

Shaw, Mary (1990) "Prospects for an Engineering Discipline of Software," *IEEE Software* 7 (6), pp.15-24.

Shaw, Mary, and Garlan, David (1996). *Software Architecture — Perspectives on an Emerging Discipline*. Prentice Hall, Upper Saddle River, NJ.

Simon, Herbert (1996), *The Sciences of the Artificial*, 3rd edition, The MIT Press, Cambridge, Mass. (first edition in 1969)

Soley, Richard (2000). *Model-driven Architecture*, Object Management Group, http://www.omg.org/mda/