An engineer is not a carpenter

A. F. M. van der Sijpt

June 2007

Abstract

We show that simple observations made about software architecture point to a set of symptoms and causes of the 'hardness' of software engineering, and software architecture in particular. We present a view on software architecture inspired by these challenges, and give recommendations for both averting them and using them to an advantage.

1 Introduction

The world in which software lives is becoming ever more dynamic. Software becomes more complex, devices that were previously driven by electronics are now controlled by software, and more software is required to 'understand' that the world is becoming networked. Handling this change from an architecture point of view was the focus of project Adaptive Architectures, as carried out at Luminis in 2006–2007.

We required a good understanding of what software architecture, and in particular style, is about. To get some feel for the matters at hand, a start-up exercise has been defined to create a system of software architecture styles, linking them to their quality factors (see Section 4 for a definition of quality factor).

Starting with an incomplete understanding due to lack of information in literature, this project ended up with just that, an understanding of (a) what is software architecture, and (b) why is it so hard to do it actually right.

This document will show some observations made during the project in Section 2. Taking together the observations, we arrive at a set of symptoms of the toughness of software architecture in Section 3 with their basic causes. Section 4 shows the resulting view on software architecture, after which Section 5 provides some recommendations for not only averting typical problems that stem from the observations made in this document, but to actually use them to an advantage.

2 Observations

48% of projects succeeds [Whi99] shows that, for companies of varying sizes, between 15% and 30% of IT projects overran their budgets by more than 50%, and between 25% and 50% overran their schedules by more than 50%. Longer projects tend to overrun more: of projects planned to take less than a year, some 25% doubled their running time, and over 30% of those longer than a year did.

In a more recent publication, [ict07], 48% of IT projects succeed. However, of the remaining projects, 4% fail completely, with 48% having difficulties. Most often, the projects did not deliver what was expected (28%), overran their running time (23%) or budget (11%).

This shows that while more project end correctly, this still represents less than half of the endeavors. The projects that do not deliver as promised are likely to be an artifact of better management techniques: cutting down on functionality, instead of overrunning time or budget.

No consensus, no feuds Although many fields of science will have at least some problems defining 'what they are about', this seems to be especially true in software architecture. This shows in a number of peculiar observations, which will not be a surprise to many practitioners.

It is not really clear what architecture is; a multitude of definitions has been defined by academics and governing bodies, but none of these is really accepted. The same applies to the job an architect does: although it seems logical that an architect is the one who creates the architecture, there is often much more to it, including gathering requirements, coaching engineers and management functions.

The definitions of architecture and the job of an architect have another notable element. The terminology has been borrowed from neighboring fields since the fifties, most notably hardware and civil engineering. Even among practitioners, analogies to neighboring fields are the main way to express views.

Furthermore, there is a 'lack of sides': although there are incompatible ideas out there (e.g., reductionism vs. holism), these do not have ranks of dedicated followers, that try to prove the other side is wrong (like, for instance, intuitionist logic). Instead, it seems that many practitioners do not really think about what side they're on, and why.

The 'ring' With borrowing terminology, the professional connotation of term like *architecture*, *construction* and *engineering* get carried over. However, with the lack of accepted definitions, these titles are free to claim, and so happens.

Tangibility The reductionist view is currently most popular, most likely due to the tangibility and direct applicability of this approach. Reductionism is about chopping up a problem into chunks, working together with defined interfaces and interactions. This is embodied in the box-and-line drawings present in software architecture documents. When focussing on structure, description methods have been devised, known as *architecture description languages*.

Other movements do exist. [SC97] shows a method for classifying architecture styles, acknowledging their potential as 'helpers' in architecting software. However, the same authors have to note in [SC06] that there have been no followers, and the same activities can still bring progress.

Potentially influential was [Fie00], in which the author is dissatisfied with contemporary views of software architecture, and develops a novel vision of software architecture as basis for his own research, breaking with the traditions of Perry & Wolf [PW92] and Parnas [Par72]. The main outcome of Fielding's research, REST, has an avid following of 'RESTafarians'; his founding work is all but ignored.

Further more, [BR98] shows our current ways of thinking about software architecture are insufficient, and not much more than structural and behavioral *descriptions*. Both Baragry's doctoral thesis and a summary of that [BR01] have been largely ignored.

What *do* we do? During the project I asked administrators of open source projects rather specific questions about the architectural choices made in their projects. I ended up with general good advice instead of the basic decisions I expected,

- "Decoupling is extremely important in open source projects to ensure minimal commit conflicts and also demarks areas of responsibility."
- "Reliability and efficiency can be derived from [maintainability], clear code can be made more reliable and more efficient."

These men represent some of the most successful open source projects, so there must be a reason for making these statements. After some more correspondence, I decided to ask some of them about the impact of software architecture. This time, I ended up with very different statements.

- "Anyone worth their salt was using object-oriented approaches long before the term object-orientation was coined, or explicit language support existed, for example"

- "qualities are inherent from the strategy [architecture], it is inside, but tactic [engineering] is here to make it seen by others"

There definitely *is* something these people do right given the success of their projects. Yet, they have great trouble pinpointing *what that is*. I do not believe they simply do not want to tell me; I have spoken with a number of software architects from the commercial field and find the same feeling there.

3 Causes

The observations above show a pattern of 'where things go wrong'. There seems to be a 'cloud' of symptoms that explain this phenomenon, all revolving around a 'nucleus' (more on that later on). The symptoms are not separate entities, they all influence each other.

Quality without a name In *The Timeless Way of Building*, Christopher Alexander [Ale79] coins the term *Quality without a name* in civil construction, but immediately shows its applicability in other fields that 'have to do with people'.

The quality 'names' the properties that a building can have, which makes its user 'feel alive'. This is very noticeable, but the actual properties are not identifiable. The quality is not directly engineerable. Alexander states that the quality is

> "... generated, indirectly, by the ordinary actions of the people, just as a flower cannot be made, but only generated from a seed."

The generalized form of this quality applies to all fields of engineering. Software architectures strive for a state of 'least conflict' between opposing interests. During the discussion sessions, this has been articulated as 'harmony'; 'elegance' comes close. Like in construction, it is not possible to identify elements of an architecture that are responsible for this elegance, but it shows throughout the product. This quality makes the difference between a solution that *works* and one that *fits the situation*.

Wickedness When looking at it from various angles, the essence of software architecture seems to shift shapes continuously. A problem keeps changing, and the proposed solution influences the shape of the problem. This might just be a wicked problem, as named by Rittel & Webber [RW73] in social planning. [vdS07] shows a full analysis of software architecture as a wicked subject; I will not go into too much detail here.

Three overarching properties of wicked problems can be identified, which can well be shown to apply to software architecture.

1. Describing the problem *is* the problem.
2. There is no stopping rule, it is never clear when the problem 'has been solved'.
3. Project are constrained in time and money, leaving no opportunity to learn from mistakes.

Formalisms can blind Since the first days of science, formalisms and abstractions have helped practitioners focus on interesting matters, ignoring details. This has allowed tackling ever more complex problems, with essentially the same amount of intellect available.

With the complexity of large software engineering projects, formalisms can prove a problem of their own. Formalisms help abstract away from details, but it cannot be known whether these details are, indeed, of minor importance. This can lead to deliberate denial of potentially crucial factors, because they do not 'fit' the formalism.

However, with the right precautions [BH95], many formalisms can prove a very useful element of the architect's toolbox. So, the problem is not the existence of formalisms as such, it is the possibility that formalisms get misused to cope with the increasing complexity of the problems we face.

Great designers In *No Silver Bullet – Essence and Accidents of Software Engineering*, Frederick Brooks [Bro86] states

> "The central question in how to improve the software art centers, as it always has, on people. (...) Great designs come from great designers. Software construction is a *creative* process. Sound methodology can empower and liberate the creative mind; it cannot inflame or inspire the drudge."

Section 3.1 shows in a more explicit way that the human mind is a prerequisite input to the process. No matter how good the methodology is, how good the procedures are and how great the tools are, it requires great designers to end up with great designs.

Unconscious talents During the course of this project, I spoke many practitioners. All of them do something 'right' in the choices they make, given the position they are in today. However, when trying to find out what that is, I am not able to identify reasons for that. In *Blink – The Power of Thinking Without Thinking*, Malcolm Gladwell [Gla05] shows the process of 'thin slicing', taking split-second decisions with incomplete information with remarkably good results. It is about relying on the processing power of the subconscious to make decisions that will take a lot of time to rationalize, if they can be rationalized at all.

From *Blink* (emphasis mine),

> "Braden has had a similar experience in his work with professional athletes. Over the years, he has made a point of talking to as many of the world's top tennis players as possible, asking them questions about why and how they play the way they do, and invariably he comes away disappointed. "Out of all the research that we've done with top players, we haven't found a single player who is consistent in knowing and explaining exactly what he does," Braden says. "They give different answers at different times, or *they have answers that simply are not meaningful*." One of the things he does, for instance, is videotape top tennis players and then digitize their movements, breaking them down frame by frame on a computer so that he knows, say, precisely how many degrees Pete Sampras rotates his shoulder on a cross-court backhand."

Maybe what we call architecture is based on a wish to explain what happens in an architect's head. Some professionals can be very good at what they do, but have no way of explaining, or even knowing, *what* they do. Our society wants to have explanations for everything, everything has to be traceable, and all men are created equal, so we want a rationalization of what people do.

This may explain the phenomenon encountered in contacting open source software administrators. They give general 'good advice', while not getting to the key issues. These are not dumb people; they represent the best of the open source software movement. Perhaps they just do not know what it is that makes them choose the right thing, they just do.

3.1 Core

It's all in your head Plato stated that reality as we perceive it is not 'raw', but interpreted by our minds. We create an internal model that strives to explain (and predict) the things we see. New experiences that fit the model are accepted, whereas contradicting experiences either mean something is happening in our perception, or our model is not correct; M.C. Escher's drawings are an example of this mechanism.

Science, too, creates models to explain observations, a "useful model of reality". Consider physics as an example. Through the ages, theories have explained the effects objects have on one another. In the 17th century, Newton's model of physics explained the way an apple falls down, and how a billiard ball transfers its energy to another when hitting it.

Newtonian physics worked well for several centuries. However, when studying small scale or high speed phenomena, it starts to break down. This is where quantum mechanics come in. This,

too, is of a constructivist nature: it explains observations, but we can never know whether the model actually represents reality.

In this respect, Einstein used the metaphor of a man faced with a clock he cannot open. He can construct a model explaining the movement of the hands and the ticking he hears, predicting every observable aspect of the clock with great accuracy. Still, he will never know what *really* happens inside [EI38].

Scientific models can be verified by experiments, as above, or by deriving new properties from accepted axioms.

In creating software, we model parts of reality in our artifacts. However, the artifact itself is no more than a collection of electrical currents; even the attribution of ones and zeroes to states of a transistor is an abstraction. All the things these bits represent only exist as a mental model; we take blobs of bits to mean numbers, objects, customers, processes.

The model can represent any reality, everything that can be imagined can be attributed to a formal construct, and we can create constructs that are arguably representations of all we can imagine. No experiments can verify the suitability of the model, nor can it be established that it is free of contradictions. On the other hand, no basic axioms exist that allow deriving a correct model.

In the end, in software, we can model any reality we can imagine, but we cannot verify its consistency; we simply have no way of exporting them from our mind in all their glory.

The semantic gap The difference between two non-automatically translatable representations of an object is known as the semantic gap. In computer science, it usually means the difference between the problem's description, and its code-counterpart. A typical example is known as the Halting Problem[1]: this can readily be expressed in natural language, but a formal representation can be proven not to return the wanted output on all allowable inputs.

The Von Neumann-machine, which is the basis of all modern computers, is expressionally equivalent to a Turing machine. Elements of this architecture may be aggregated into higher level constructs, such as programming language statements or objects. These help in *narrowing* the semantic gap. However, these constructs can be computationally translated to machine code, meaning that high level languages are still Turing complete.

The semantic gap will always exist, no matter how high the level of abstraction in constructs becomes.

The circle The elements above create a circle: in software, any reality can be created and used, but its correctness cannot be verified. This means the mental gap is bridged by non-verifiable constructs, keeping the gap alive since it would only be really closed once a Turing machine can bridge the gap, which cannot be done since the mental constructs are necessary; these cannot be formalized because of the semantic gap, etc.

The circle described above is unlikely to be broken anytime soon. Both elements are caused by the fact that we do not understand how the human mind works ("The mind is a strange and wonderful thing, I'm not sure it'll ever be able to figure itself out. Everything else, maybe, from the atom to the universe, everything except itself."[2]). The only way of breaking the circle is finding this understanding and modeling (the relevant aspects of) the human mind in a Turing machine. Note that some believe that human-like intelligence will arise once computer processors get powerful enough[3].

Until that moment, there is only one tool that can effectively bridge the semantic gap: a human mind able to use sufficiently complex mental constructs, so it has no need to resort to over-abstraction.

[1] `http://en.wikipedia.org/wiki/Halting_problem`, "Given a description of a program and a finite input, decide whether the program finishes running or will run forever, given that input."

[2] From *Invasion of the Body Snatchers*.

[3] For instance, Arthur Koestler's *Ghost in the machine*, which argues the human brain is machine, building upon primitive structures, and 'awakening' when its complexity is sufficiently high.

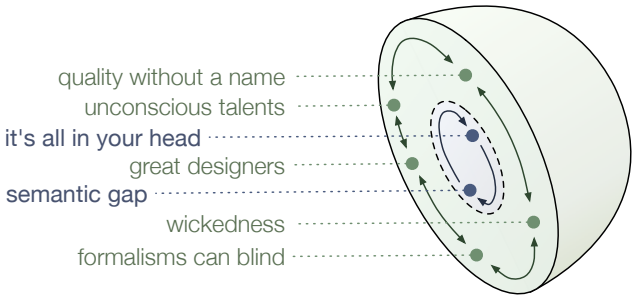

Figure 1: Symptoms and core problems.

3.2 Relations

This document has shown a number of observations, the five symptoms they 'break' on, and two core elements causes this. Figure 1 shows the symptoms and their core.

Figure 1 can be seen as a cross-section of a sphere, in which the sphere represents the 'hard elements' of software architecture. This particular cross-section represent the approach of "using quality factors to make sense of software architecture". Other approaches will create a different cross-section of the sphere, possibly encountering other symptoms. I believe the core is common for all parts of software engineering.

4 Now, what *is* software architecture?

4.1 Introduction

Software architecture uses borrowed terminology, placing the architect in a role where he is the one that envisions the entire system, and is occupied with creating its high-level structure. Like this is not entirely true in construction, I believe it is not in software either.

Architecture is about making a system fit its context. This applies to the system *as a whole*; I do not believe architecture is primarily about structure and 'chopping up' a problem into manageable pieces [Par72]. Rather, it is about *cohesion* and *alignment*.

In construction, blueprints and maquettes are manifestations of the architecture. The actual choices the architect makes are about the *ambiance* of a building; decisions like 'I don't want any corridors ending in three doors'. These embody unshakeable constraints on the architecture, stating what is *required* to make the product fit its context. Note that this can include, but is not limited to, structure.

While the architectural choices above are very articulate, it is possible that they are more 'hidden'. They will manifest themselves in the resulting artifacts, but cannot be articulated as such. Still, they are part of the architecture, and can find their ways into elements like *rationale*.

These views have been inspired mainly by two authors. First, the idea that software is 'something completely different' came from experience during the project, and turns out to have been developed by Jason Baragry [BR98, BR01]. The views on architecture and the resulting artifact have been inspired by Fielding [Fie00].

4.2 A model

An architecture is a set of concepts, guidelines for creating the resulting artifacts. This idea puts styles on the same level of abstraction as architectures, merely a collection of concepts, brought together with a convenient name.

Both styles and architectures *can* support certain quality factors, given that the implementation does not break them. A system can only exhibit desirable non-functional properties when both

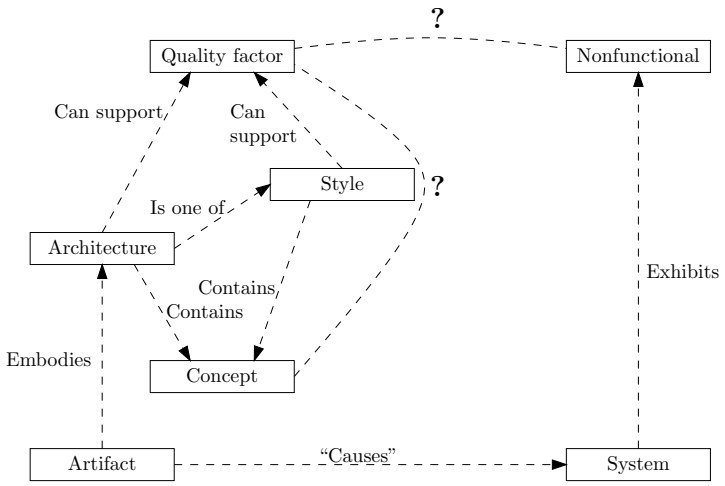

Figure 2: The context of software architecture.

the architectural support and engineered implementation are in place. The artifacts that embody the architecture give rise to the system. This system will exhibit measurable, non-functional properties. This cohesion is shown in Figure 2.

In this figure, dashed lines represent the named relations of the previous paragraph. Cardinalities are apparent, and have been left out. Two relations are marked with a question mark. The link between concepts and quality factors is what this project tried to uncover. The second questionmarked connection is between non-functional properties and quality factors. This projects has assumed them to be the same. I believe they actually *are* the same, or at least one-on-one mappings can be formulated.

4.3 The components

Quality factor "*Quality factors* are the (non-)functional properties that an architecture *can* support, given an upholding of the architecture's concepts in all relevant artifacts."

Architecture "An *architecture* is a set of concepts, brought together to make the product fit its context."

Style Following Perry & Wolf's view, a *style* is a less specific architecture.

> "A style is a collection of concepts to be called by a convenient name, possibly accompanied by a rationale for this combination of concepts."

Concept The most intangible element of this model is the *concept*. These embody the choices made by the architect, the leading principles of an architecture. The introduction has shown that concepts in construction are decisions that could be applied to all buildings, but have been chosen to be included in just *this* situation.

> "A *concept* is a fundamental choice—articulated in a pattern, guideline or even some examples—making the product fit its context, and should be recognizable throughout the product."

This principle means that the key concepts making up the architecture should fit on no more than a few pages. However, rationale and elaboration of the concepts can take up quite some more paper, but are not considered part of the architecture.

Artifact *Artifacts* are the embodiment of the architecture, tangible 'stuff' that shows the concepts the architecture is made of. This can include documents, but also code. In all of these, the influence of the architecture has to be held up.

The element that is most commonly seen as 'the' architecture, the system's structure with modules and interfaces, requires some extra attention. When the context requires a fixed structure, this can very well be a concept; however, when other concepts dictate a some structure, it will only be part of the artifacts. Furthermore, in systems with a dynamic structure, it will not even be part of any document, but will be dictated at runtime by governing concepts.

System The system is the physical manifestation described in the artifacts, i.e., the system behavior.

Non-functional The non-functional properties of a system indicate 'how well' a system does its job.

5 Conclusion

With the problems shown in Section 2, it seems a miracle that so many project actually end well. Granted, not all is bad. We work in a field that attracts bright minds, and often they get their say. However, [Whi99] shows that larger projects have a greater chance of failing. It seems our approaches that work reasonably well, have problems scaling up.

5.1 An engineer *is not* a carpenter

There are some important aspects that sets an engineer apart from a carpenter; these properties apply to all who have to do with creating software, including architects.

Software is not wood The tools a software engineer has to work with are largely mental (Section 3.1), though aides have been created to help the engineer visualize his ideas.

Laws of gravity Artifacts that are created in real life can be touched and handled. When a carpenter builds something, it will soon be apparent when something is crooked. Due to the mental aspects of software, we do not have this luxury.

A module is not a wall When a contractor orders a mason to build a wall, he has fairly good idea what it will look like when finished. When an IT architect hands over responsibility for a module to an engineer, he has no idea what the engineer will come up with; perhaps, there are some ideas, but the same problems can likely be solved in many ways.

This last difference brings us to a core observation: in construction, one man designs a solution, and the other builds it. We consider the man designing the solution, the architect, to be the creative man, and the man who builds it, the mason, to be the laborer. In software, designing the solution *is* the solution; the computer is the laborer.

5.2 Recommendations

Be aware [vdS07] shows that software architecture can reasonably be classified as a wicked problem. So, the same recommendation from [RW73] can be used here: be aware. Be aware of the typical pitfalls that are situated in the field of software architecture. Be aware of the difficulties that are described in Section 3, and learn to recognize them.

Trust good people And finally, be aware that creating software is a creative, human activity. Only the human mind can bridge the semantic gap. Tools can help the mind in this, but they might also obscure the gap, leading us to believe it has disappeared. No single methodology works for all systems; specific methodologies might prove useful for very specific domains, but even then there is a semantic gap that needs to be closed. A good understanding of 'what we are doing' is indispensable.

It has been quoted earlier, but [Bro86]

> "great designs come from great designers"

Creating 'good' stuff requires a keen mind that is aware of the problems it might face. This means that sets of competencies do not make people replaceable. The study has shown that good architects have some traits in common; unfortunately, these seem to be hidden in the subconscious, making it impossible to directly screen people for having 'great designer'-genes.

Still, what is it that makes these great designers? Based on what I've seen and heard in the past months, I don't believe a typical background can be pointed out. Skills taught at universities are indeed useful, if only in training the mind in understanding complex things; however, controlled case work usually delineates the world of the problem very strict, while exactly this framing is a skill of a great designer. The only binding factors for great designers now seem to be a keen mind, a great amount of 'understanding'. From that point on, as Brooks states, the designers must be 'grown'.

References

[Ale79] Christopher Alexander. *The Timeless Way of Building*. Oxford Press, 1979.

[BH95] Jonathan P. Bowen and Michael G. Hinchey. Ten commandments of formal methods. *Computer*, 28(4):56–63, 1995.

[BR98] Jason Baragry and Karl Reed. Why is it so hard to define software architecture? In *APSEC*, page 28. IEEE Computer Society, 1998.

[BR01] Jason Baragry and Karl Reed. Why we need a different view of software architecture. In *WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, page 125, Washington, DC, USA, 2001. IEEE Computer Society.

[Bro86] Frederick P. Brooks. No silver bullet - essence and accidents of software engineering (invited paper). In *IFIP Congress*, pages 1069–1076, 1986.

[EI38] Albert Einstein and Leopold Infeld. *The evolution of physics: the growth of ideas from early concepts to Relativity and quanta*. 1938.

[Fie00] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. Chair-Richard N. Taylor.

[Gla05] Malcolm Gladwell. *Blink: The Power of Thinking Without Thinking*. Little, Brown, January 2005.

[ict07] Ernst & Young ICT Barometer — ICT projecten, June 2007.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.

[RW73] Horst Rittel and Melvin Webber. Dilemmas in a general theory of planning. *Policy Sciences*, 4(2):155–169, June 1973.

[SC97] Mary Shaw and Paul Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *COMPSAC '97: Proceedings of the 21st International Computer Software and Applications Conference*, pages 6–13, Washington, DC, USA, 1997. IEEE Computer Society.

[SC06] Mary Shaw and Paul Clements. The golden age of software architecture: A comprehensive survey. (CMU-ISRI-06-101). Pittsburgh, PA: Institute for Software Research International, Schoool of Computer Science, Carnegie Mellon University, February 2006.

[vdS07] A. F. M. van der Sijpt. Software architecture as a wicked problem. luminis® internal, February 2007.

[Whi99] Brenda Whittaker. What went wrong? unsuccessful information technology projects. *Information Management and Computer Security*, 7(1):23–29, 1999.