Software Architecture as a Shared Mental Model

Ric Holt,
University of Waterloo, Waterloo, Canada
holt@uwaterloo.ca

Abstract. Software architecture is commonly considered to be the structure of a large piece of software -- commonly presented as a nested set of box and arrow diagrams. This paper takes a different position, claiming that software architecture is most usefully thought of as a mental model shared among the people responsible for software. This point of view leads to a number of principles regarding how we should design, present, and think about software architecture.
Keywords. Software architecture, software visualization, cognitive models, program comprehension.

Background: Styles, Reference Architectures and Views

Thanks to foundational contributions by Garlan & Shaw [2], Perry & Wolf [4], Kruchten [3], and others, we have a clear idea that there are:

1. *styles* or patterns of architecture, e.g., the pipe-and-filter style,
2. *reference architectures*, e.g., the reference architecture for compilers contains a scanner, a parser, a semantic analyzer, etc.,
3. *views* of a software architecture, e.g., the view of the logical hierarchy of the system, or the view of the executing software across hardware systems.

I will assume that these ideas are well understood.

At Least 100 KLOC

I will consider only those software systems which are large enough so that they require years of development by a team (at least three people). In other words, I will only consider *PitL* (programming in the large) and *PitM* (programming in the many). Smaller systems have structure, which perhaps should be called *patterns*, I would not call that structure "software architecture". Generally, PitL and PitM come into play when the source code size reaches roughly 100 to 500 KLOC.

Social Architecture

Systems characterized by PitM are inherently managed by teams and these teams must communicate. Indeed:

Conway's Law. *The structure of a system reflects the structure of the team that created it* [1].

This law leads to:

Position of paper. *The key purpose of software architecture is to facilitate team communication and understanding*.

For a team to communicate, it needs a common ”vocabulary", e.g., a person working on Linux needs to be able to refer to the Memory Manager and to IPC without wasting time wondering if his cohorts understand what he is talking about.

But this mutual understanding must be much deeper than shared phrases. Rather, the understanding must include a shared model of what the system parts are and how they interact. My position is that this shared model is the essence in software architecture. Without this model, the team would be mired in confusion, without a way to work cooperatively in developing and maintaining the software.

In a perfect world, this model would be written down, say in an Architecture Description Language, but we don't have time to wait for perfection. Why? Because software developers never have enough time. Because the mental model must exist for the team to make progress, and once it exists in peoples' minds, much of the advantage of recording it in detail becomes superfluous. Of course, documentation of software architecture is a good thing, but such documentation should recognize its purpose: helping people to think and communicate. Excessive time spent in documenting

a changing architecture is taken away from other demands such as demands for short-time-to-market and high performance. While there are uses for a precisely recorded software architecture, that I will maintain the claim that a shared mental model is the main purpose of a software architecture.

I hope that I have established that one of the key purposes of software architecture is to serve as a shared mental model. This position leads to the question of how we can best use our mental model (our software architecture). This means: how do we best use our mental horsepower --- our personal cognitive engine --- to deal with software architecture. Much of the rest of the paper lists cognitive principles and their relationship to software architecture.

But first, let's consider the fact that software architecture necessarily exists in a social setting. It is a short step from Conway's Law to the concept of ownership, i.e., in PitM it is scarcely avoidable that parts of the system become "owned" by individuals, who are responsible for the correct working of the part, and for understanding the workings of the part. This in turn leads to (sometimes overly) proprietary ownership and territoriality, to positions of influence and power, e.g., I understand this part and I won't explain it to you unless you're nice to me.

It leads to concepts of power and governance: who controls the overall architecture and how are decisions taken to change it. How are decisions about the architecture enforced, e.g., if the team head directs that no-one is to use MFC (Microsoft Foundation Classes), how does he make this decision stick?

These social concepts impinge directly on software architecture, indeed, it is best to consider that they are a part of the architecture. To understand the architecture, Conway reminds us that you need to understand the organization of the team that manages it.

Cognitive Principles

This section presents various cognitive principles and discusses how they can help us create better software architecture.

Students are taught that it is good to learn all they can about the subject at hand; curiously, this lesson works in reverse at the architecture level. At this level, we try to minimize what we learn because this learning comes at the expense of limited time and limited brain power --- and we are easily overwhelmed by complexity.

Cognitive miser principle. *Don't waste brain power* [6].

A key purpose of software architecture is to assist our mental faculties by enforcing or promoting simplicity. This, in spite of the fact that the actual software system is overwhelmingly complex. Our software architecture must abstract away unwanted detail, even if the abstraction sometimes is not true to the actual underlying system. To re-iterate this curious fact: it is often advantageous for the software architecture to *oversimplify* its representation of the corresponding implementation, in order to make it easier think about the architecture. (Of course, one must be honest and recognize the danger of these simplifications.)

Law of maximal ignorance. *Don't learn more than you need to get the job done.*

This corollary of the Cognitive Miser principle explains that the software architecture can protect us from learning too much about the system. We don't have time to learn everything; we barely have time to learn what is essential for today's work. A good software architecture is sparse (light weight), telling us only what we need not know about the shape of the system, its parts and their interactions.

To construct effective mental models for software architecture, we need to know something about how brains work.

Right brain (visual) architecture. *Brain science indicates that spatial reasoning is isolated, usually in the right brain hemisphere.*

Related cognitive facilities are the basis of our understanding of orientation (above, beside, inside, etc.), connectivity, position, texture, colour, etc. When we *visualize* software architecture, we are using these cognitive facilities to represent our mental models. Indeed, for many of us "seeing is believing", i.e., we only understand a model when we can visualize it.

The term *visualization* is often used, in quite a different sense, to mean creating a diagram, usually on a computer screen. We should not confuse that action, carried out by a computer, with the cognitive function of constructing and manipulating a mental model.

Our brains are marvelously adapted for visualization. How else can we walk through our homes without damaging ourselves? How can we give directions for finding a restaurant? We use these powerful mental facilities to help us proficiently think about and talk about software architecture. We make hand gestures to show flow of data. We say a particular subsystem calls "down" to another. We draw one box inside another to show it is contained. We draw a big box to represent a subsystem containing much code. We draw connecting arrows to illustrate interacting subsystems. We show data flow going from left to right, etc.

Law of minimal change. *When the software changes in a modest way, our model for it should also change in a minimal way.*

For example, visualization of software architecture of two sequential releases of Linux should have similar layouts, colours, connectivity, etc. Why? Because sequential releases do not involve large changes in the software architecture (well, not very often, anyway). Unfortunately, some tools purporting to help us understand the structure of a system produce automatic layouts, which ignore the layouts of previous versions of the system.

Once a team has invested the time to visualize the architecture of its software system, they should not change their mental model of the system unnecessarily. Each change of this model in their heads takes time, causes confusion and is error prone. Think of the change as installing a new piece of software, while dis-installing old software, in the heads of each team member.

Law of position permanence. *Visualizations of versions of a system should show corresponding parts of the system at roughly the same positions with roughly the same sizes and shapes*.

Position permanence follows from the fact that our mental models are commonly positional, e.g., we may visualize the support library as lying in the bottom right corner of the picture of the entire system. We should not unnecessary change these aspects of the model.

Left brain (verbal) architecture. *Much mental representation of a software architecture is verbal or logical information,*.

For exampe, "the Memory Manager implements virtual memory, and has an internal structure inherited from Minix". This verbal information must be efficiently linked to the right brain architecture, i.e., to the visualization.

Note that the left brain, as well as the right brain, benefits from appropriate use of principles such as the Cognitive Miser, Maximal Ignorance, Minimal Change (e.g., do not needlessly change terminology), etc.

Cleanliness and Simplicity

As I have already noted, a key purpose of software architecture is to provide a shared mental model that is easy to understand and to visualize. There are important further advantages of visualization, as I will now discuss.

Law of ugliness hiding. *Unobserved ugly parts of a system stay ugly*.

(Apologies to David Parnas for perverting his "information hiding" slogan.) They stay ugly because we don't know that they are ugly. The converse of this law is that once we see ugly things, we tend to fix them.

Aesthetic principle. *Visualization leads to cleanliness.*

The Aesthetic principle, applied to software architecture, means that things that are messy to think about probably ought to be simplified. People instinctively like things to be simple and clean, so when they see an ugly thing, they try to fix it. These laws regarding ugliness and cleanliness explain why software visualization leads to better software.

There is a deep need for simplicity of software architecture: our brains are the essential machinery we use for solving serious problems in software architecture. If we fail to feed our brains with good mental models, which fit well with our cognitive abilities, we will not do a good job as software architects.

One way we "add simplicity" to a complex software architecture is to factor off complexities, which correspond to:

Platform assumptions. *Each system rests on a set of mechanisms and assumptions, called here "platforms"*. For example, the system is written in Java, runs on Unix, uses call backs for GUI, runs each transaction concurrently, etc. These must be learned but should not be considered part of the (central) architecture. This approach simplifies the central architecture. Aside: Clashing platform assumptions make system merging difficult [5].

Conclusion

This paper takes the position that software architecture is most usefully thought of as a means for sharing thoughts among developers of a system. This point of view leads us to focus on how we think about software architecture and how we should optimize ways of representing architecture -- to improve our thinking and communication.

I am not saying: do not develop or study software architecture. Rather, I am warning that overly mechanized or detailed approaches to software architecture are self defeating.

When teaching about or designing software architecture we should remember that architecture is intimately intertwined with the social structure of the development team. We should remember that architecture is used largely within peoples' heads, to think about what the architecture is and how to change it, and then to communicate these ideas to other people. We should remember that our cognitive facilities are highly limited and at the same time are highly tuned for certain kinds of operations such as spatial reasoning.

Irony of simplicity. *The hard part is making it easy.*

The hardest part of creation of good software architecture is figuring out how make it easy to understand.

Acknowledgements. Thanks to Matt Armstrong for his helpful suggestions on an early draft of this paper. Thanks to the IWPC referees for a number of corrections and good suggestions.

References

1. Ivan T. Bowman and Richard C. Holt., “Reconstructing Ownership Architecture To Help Understand Software Systems”, *IWPC '99: International Workshop on Program Comprehension*, Pittsburgh, May 5-7, 1999.
2. Garlan, David and Shaw, Mary, “An Introduction to Software Architecture”, *Advances in Software Engineering and Knowledge Engineering*, Volume 1, World Scientific Publishing Co., 1993.
3. Kruchten, P.B., “The 4+1 Views Model of Architecture”, *IEEE Software*, Nov 95, pp 42-50.
4. Perry, Dewayne E. and Wolf, Alexander L., “Foundations for the Study of Software Architecture”, *ACM SIGSOFT Software Engineering* Notes, 17:4, October 1992 pp 40-52.
5. Garlan, David and Allen, R., and Ockerbloom, J., “Architectural Mismatch or, Why it's Hard to Build systems out of Existing Parts”, *Proceedings of the 17th International Conference on Software Engineering* , April 1995.
6. Psybox web site, http://www.psybox.com/web_dictionary/dictionaryWebindex.htm.
   Psybox definition: "Cognitive miser – Theory from the social cognition field, whereby human beings are seen as having limited processing capacity with which to deal with an infinitely complex and ever-changing environment. They must therefore make the best use possible of these resources and treat them in a miserly manner, utilizing heuristics, schemas, etc."