Mommy, where do software architectures come from?

Philippe Kruchten
Rational
6857 Wiltshire street
Vancouver, B.C. V6P 5H2
Canada
pkruchten@rational.com
phone: +1 (604) 231 3132
fax: +1 (604) 278 5625

Introduction

There are two central issues in software architecture: representation and creation.

- *Representation*: how can we describe a software architecture, specify it? What tools, language, notation can be used to describe it? Completely ad hoc strategies have been used by each project. Some advances have been made in the last 3 years in that direction to make a more systematic representation and specification.
- *Creation*: what is the process to create, to produce a software architecture? What happens in the mind of the software architect? Where do the elements come from? Who are the architects?

In this paper we will very briefly describe the software architecture model that we have been using for documenting, specifying a software architecture, then we will attempt to give some hints on the process of *defining* a software architecture.

An Architectural Model[1]

Software architecture deals with the design and implementation of the high-level structure of the software. It is the result of assembling a certain number of architectural *elements* in some well-chosen *forms* to satisfy the major functionality and performance requirements of the system, as well as some other, non-functional (afunctional?) requirements such as reliability, scalability, portability, and availability.

Software architecture = {Elements, Forms, Rationale}[2]

Software architecture deals with abstraction, with decomposition and composition, with style and esthetics. To describe a software architecture, we use a model composed of multiple *views* or perspectives. In order to eventually

[1] This section is extracted from [KRU94]

[2] Formula due to Dewayne Perry & Alexander Wolf [PER92], extended by Barry Boehm to include "constraints" [BOE94]

address large and challenging architectures, the model we propose is made up of five main views (cf. fig. 1):

- The *conceptual* view, which is the object model of the design
- the *dynamic* view, which captures the concurrency and synchronization aspects of the design
- the *physical* view, which describes the mapping of the software onto the hardware and reflects its distributed aspect
- the *static* view, which describes the organization of the software in the development environment.

The description of an architecture, the decisions made, can be organized around these four views, and illustrated by a few selected *scenarios*[3] which become a fifth view. The architecture is fact partially evolved from the scenarios as we will see below.

The design guidelines and rationale for architectural decisions are captured in parallel with these decisions in order to maintain the integrity of the architecture over time. The quality of the architecture is evaluated in terms of usability, simplicity, versatility, robustness, and efficiency. It is assessed by implementing and evaluating a succession of architectural prototypes that evolve into the final system.

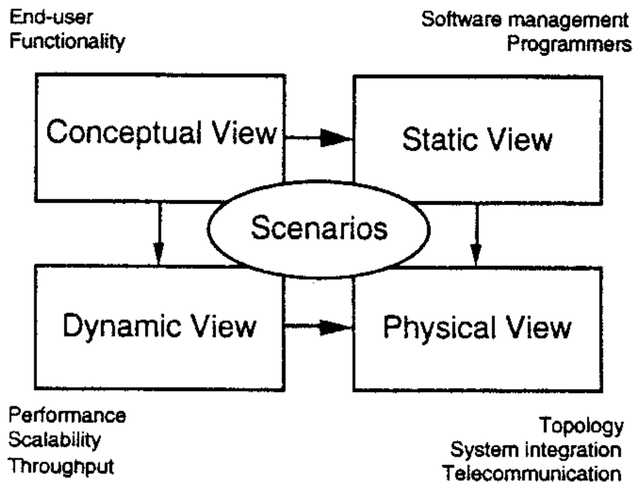


Figure 1 — the "4+1" view model

[3] Ivar Jacobson et al. call them "use cases" [JAC92]

Architecture Sources

There are much less source of information to where architectures come from. It is often hinted that this is a bit black magic, people having 'architectural visions'. Witt et al. [WIT94] give a description of a step by step method to define an architecture, which seems to be aimed at a certain kind of software system (MIS): they proceed with ready-made forms: client/server/transactions, and do not mention where new *forms* come from, especially when designing other kind of software systems (embedded, real-time, command and control, software production, etc.)

There are 3 main sources of architecture, and every software architecture I have seen created used some combination of the three in some proportion: theft, method and intuition.

- *Theft*: most elements of a software architecture are just "lifted" from other architectures the architects happen to be familiar with:

  —the previous system of the same kind,

  —another system with overall similar characteristics,

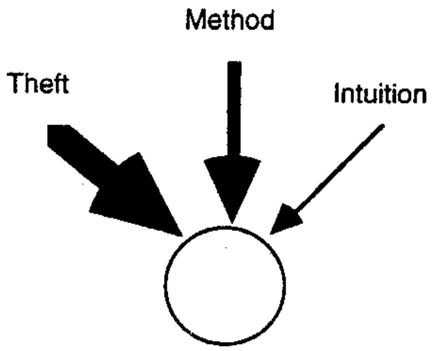
  —some architecture found in the technical literature.

  If there weren't so much theft, it would be very hard to build taxonomies of architectures or to create architecture description languages. As the domain matures, handbooks of software architectures will emerge out of which it will be easier to steal.[4]
- *Method*: a systematic, conscious, maybe even documented way by which an architecture is derived from the system's requirements and technological constraints, applying well-known transformations or some heuristics; this is what Witt et al. have described. In most cases, the software architects apply heuristics, again derived from experience, and very often those heuristics are not completely spelled out.[5]
- *Intuition*: intuition is the ability to conceive without conscious reasoning; a significant amount of architecture invention is just pure intuition; the eye of the experience software architect recognizes some pattern, or finds another point of view, or changes a way to express things, and Voilà! an architectural element of form appears, which then need to be confronted with the requirements, the architecture already in place, etc.

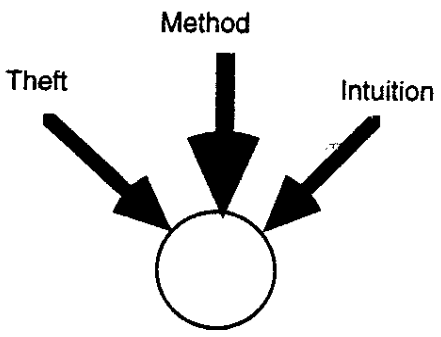
The ratio between those three sources varies according to the experience of the architects and to the degree of novelty or "unprecedentedness" of the system they are designing.

[4] As an example of micro-architectural pattern handbook, see [GAM94]

[5] Eberhardt Rechtin at USC is trying to collect software architecture heuristics, to complement the system architecture heuristics he already documented in [REC91]



A classical system



An unprecedented system

The contribution of the 3 sources can have different profiles; for instance:

- Classical: 80% theft, 19% method, 1% intuition
- Unprecedented: 30% theft, 50% method, 20% intuition

Intuition is a great thing and source of real creativity. But however it needs to be supported by method. The intuitions need to be carefully reviewed, integrated, validated by experiments, prototypes. The higher the reliance on intuition, the greater the risks. A architecture built with a profile of: 50% theft, 0% method, and 50% intuition is doomed to fail.

Method and Process

There are 3 levels at which we can look for a method:

- the macro-process: how does the design of an architecture fits in the overall software life-cycle; the scale is the month, the year
- the micro-process: how do architecture proceed to construct, document, validate the architecture in a methodical fashion; the scale is the day, the week
- the nano-process: what happens in the head of the architect at the scale of the second or the hour.

Iterative process[6]

Witt et al. indicate 4 phases: sketching, organizing, specifying and optimizing, subdivided into some 12 steps. They indicate that some backtracking may be needed. However we think that this approach is far too linear for an ambitious and rather novel project. Too little is known at the end of the 4 phases to validate the architecture. We advocate a more iterative development, were the architecture is actually prototyped, tested, measured, analyzed, and then refined in subsequent iterations. Besides allowing to mitigate the risks associated with the architecture, such an approach has other benefits for the project: team

[6] We described this process in [KRU91].

building, training, acquaintance with the architecture, run-in of procedures and tools, etc.

A scenario-driven approach

The most critical functionality of the system are captured in the form of scenarios (or use cases). By critical we mean: functions that are the most important, the raison d'être of the system, or that have the highest frequency, or that present some significant technical risk that must be mitigated.

Start:

- A small number of the scenarios are chosen for an iteration based on risk and criticality. Scenarios may be synthesized to abstract a number of requirements.
- A strawman architecture is put in place (mostly by theft). The scenarios are then scripted in order to identify major abstractions (classes, mechanisms, processes, subsystems).
- The architectural elements are laid out on the 4 blueprints mentioned above: conceptual, dynamic, static, and physical.
- This architecture is then implemented, tested, measured, and some analysis may detect some flaws.

Loop:

The next iteration can then start by:

- reassessing the risks,
- extending the palette of scenarios
- selecting a few additional scenarios

Then:

- try to script those scenarios in the preliminary architecture
- discover additional architectural elements, or sometimes significant architectural changes.
- update the 4 blueprints
- revise the existing scenarios based on the changes
- upgrade the implementation to support the new extended set of scenario.
- Test. Measure under load, in real target environment.
- All five blueprints are reviewed to detect potential for simplification, reuse, commonality.
- Guidelines and rationale are updated.

End loop

The initial architectural prototype evolves to become the real system. Hopefully after 2 or 3 iterations, the architecture itself become stable: no new major abstractions are found, no new subsystems or processes, no new interfaces. The rest of the story is in the realm of software design, where, by the way, development may continue using very similar methods.

The nano-process

What happens in the mind of the architect? A certain number of intellectual activities: abstraction, generalization, specialization, induction. There is also a process called "heuristic jump". The architect selects a heuristic, applies it to jump ahead, draws some conclusions, then steps back and explore systematically all consequences of that "jump."

What drives which activity, or which heuristic to apply depends on the specific problem:

- Design getting too complex: abstraction, generalization, trying to find commonality, trying to challenge strange, unorthogonal requirements.
- Discovery of a new requirement: pattern matching, extension
- Performance issue: specialization
- Wide interface, high coupling: reorganize the groupings
- Stuck by too many constraints: ignore half of the requirements (for a while)

etc.

Who are the Architects?

The architect or group of architects collectively must have a solid experience of software development, over the entire life-cycle, and solid domain expertise, at roughly 50/50. The intuition mechanism works only when there is enough material stored deep in the "bottom neurons."

In a team, a maximum of overlap is preferable to really be able to do some teamwork. This can be achieved by cross training: train the software people in the domain and vice versa.

They must have a very good feel for abstraction, able to see the "big picture", but able to dive into technical details as necessary. They are synthesists, more than analysts [WEI88].

They must be great communicator (written and oral), since a large part of their activity will be to explain the architecture to all kind of people inside and outside the organization.

They must be able to negotiate and achieve balance and compromise between all stakeholders: customers, management, developers, etc. as well as internally within the team.

Architects are very curious people, and often in plenty of other fields than just software. This allows them to have more material to steal from and more sources of analogies and metaphors.

The architects must be dedicated to that task, and have proper authority to enforce technical decisions. The size of the architecture team is around 5% to 12% of the total number of developers.

In Myers-Briggs classification, there are often of the xNTy kind, and INTP are even labeled the "Architect," although in a large organization, a mix of personalities does help to achieve a real team [KAT93]

Conclusion

Software architectures come from 3 sources: theft, intuition and method. Experience and wide culture are invaluable for theft and intuition. A process, a systematic way to approach the building of an architecture, as well as a defined way to specify and document it is key to the third aspect. We will be some day able to answer the question: "where are the building codes?" But we are still a bit far from the "software architecture assembly line," there is still many years of long pregnancies and difficult births ahead of us.

Acknowledgements

Many thanks to Alex Bell and Drasko Sotirovski from Hughes Aircraft of Canada for the stimulating discussions that lead to this paper, and to Grady Booch from Rational for the impulse.

References and Other Sources of Inspiration:

GAM94 Erich Gamma, Richard Helm, Ralph Johnsson & John Vlissides, *Design Patterns—Elements of Reusable Object-Oriented Software*, Addison-Wesley (1994) 395p.

GAR93 David Garlan & Mary Shaw, "An Introduction to Software Architecture," *Advances in Software Engineering and Knowledge Engineering*, Vol. 1, World Scientific Publishing Co. (1993)

GLA94 Robert Glass, *Software Creativity*, Prentice-Hall (1994) 268p.

JAC92 Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Övergaard, *Object-Oriented Software Engineering—A Use Case Driven Approach*, Addison-Wesley, Wokingham, England (1992) 528p.

JON94 Capers Jones, *Assessment and Control of Software Risks*, Yourdon Press, 1994, 619p.

KAT93 Jon R. Katzenbach & Douglas K. Smith, *The Wisdom of Teams*, Harper Business Press (1993) 265 p.

KRU91 Philippe Kruchten: "Un processus de dévelopment de logiciel itératif et centré sur l'architecture", *Proceedings of the 4th International Conference on Software Engineering*, Toulouse, France, December 1991, EC2, Paris.

KRU94 Philippe Kruchten & Christopher Thompson, "An Object-Oriented, Distributed Architecture for Large Scale Ada Systems," *Proceedings of the TRI-Ada '94 Conference*, Baltimore, November 6-11, 1994, ACM, 262-271.

PER92	Dewayne E. Perry & Alexander L. Wolf, "Foundations for the Study of Software Architecture," *ACM Software Engineering Notes*, **17**, 4, October 1992, p.40-52.
REC91	Eberhardt Rechtin, *Systems Architecting—Creating and building complex systems*, Prentice-Hall, Englewood Cliffs, N.J. (1991) 333p.
RUB92	Kenneth Rubin & Adele Goldberg, "Object Behavior Analysis," *CACM*, **35**, 9 (Sept. 1992) 48-62
SHA94	Mary Shaw, R. DeLine, D. Klein, Th. Ross, D. Young, & G. Zelesnik, "Abstractions for Software Architecture and Tools to Support Them", February 1994, Private communication.
WEI88	Gerald Weinberg, *Rethinking Systems Analysis & Design*, Dorset House, New York (1988)
WIT94	Bernard I. Witt, F. Terry Baker & Everett W. Merritt, *Software Architecture and Design—Principles, Models, and Methods*, Van Nostrand Reinhold, New-York (1994) 324p.