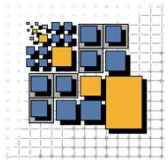


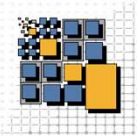
DSG-DSAM-M 2024/25:
- Software Architecture -

Dr. Andreas Schönberger

Distributed Systems Group
Faculty Information Systems and Applied Computer Science
University of Bamberg

If you think good architecture is expensive,
try bad architecture.

What is Software Architecture?



- IEEE 1471-2000 defintion:

  “Architecture is defined […] as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.”

  [interfaces are missing!]

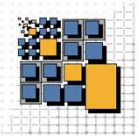
- Informally speaking, software architecture is the building plan of software that describes structure, behavior and guiding principles such as:

  cross-cutting mechanisms, programming conventions, build and development practices, concurrency mechanisms

- Grady Booch:

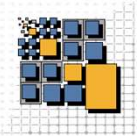
  “Architecture is about everything costly to change”

What is a Good Software Architecture?



- A good software architecture meets its requirements!
  - not less, otherwise stakeholders will be unsatisfied
  - not more, otherwise (some other) stakeholders complain about cost
- A good software architecture follows simple principles
  - high cohesion within components
  - loose coupling between components
  - domain-driven component cut
  - no components can be left out any more
- A good software architecture is well communicated
- A good software architecture is enforced

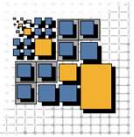
How to Communicate/Enforce Architecture?



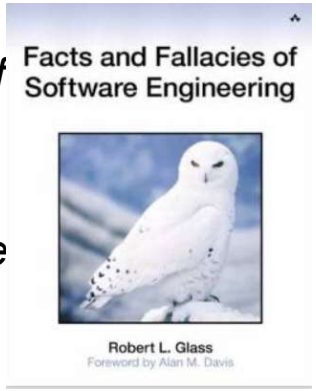
- There are many things to communication and enforcement of architecture. Let us look at documentation first, for
  - Product owner and customers: *they need to accept it*
  - Developers: *they need to implement it*
  - Companion architects: *they need to interface with it*
  - Test managers and testers: *they need to (be able to) test it*
  - Project, quality, line managers: *they need to accept it*
  - Certification authorities: *they need to sign it off*
  - Yourself: *you need to maintain it*
- Many description formats are available, among others
  - Zachman Framework
  - Semantisches Objektmodell
  - **Kruchten: 4+1 View**
  - arc42 template (http://www.arc42.de/)

cf. Prof. Sinz

A few Numbers Related to Documentation



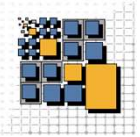
- Robert L. Glass, **Facts and Fallacies of Software Engineering**
- *"Maintenance typically consumes 40 to 80 percent (average 60 percent) of software costs. Therefore, it is probably the most important life cycle phase of software."*
- *"Enhancement is responsible for roughly 60 percent of software maintenance costs. Error correction is roughly 17 percent. Therefore software maintenance is largely about adding new capability to old software, not fixing it."*
- *"About 30% of effort in maintenance is spent on understanding existing software"*



Hence,

- up-to-date and understandable documentation is catalyst to maintaining your software
- do not document in too much detail, because you have to keep it up to date and somebody must be willing to read it

Let's Start Simple – Express what you Build

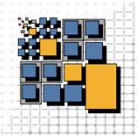


For any software product, the product owner should be able to give a mission statement:

- For *[target customer]*
- Who *[needs]*
- The *[product]*
- Is a *[product category]*
- that *[key benefit; USP]*
- Unlike *[competitor]*
- our product *[competitor differentiation]*

cf. SOM

Let's add Structure - Kruchten's 4+1 Views

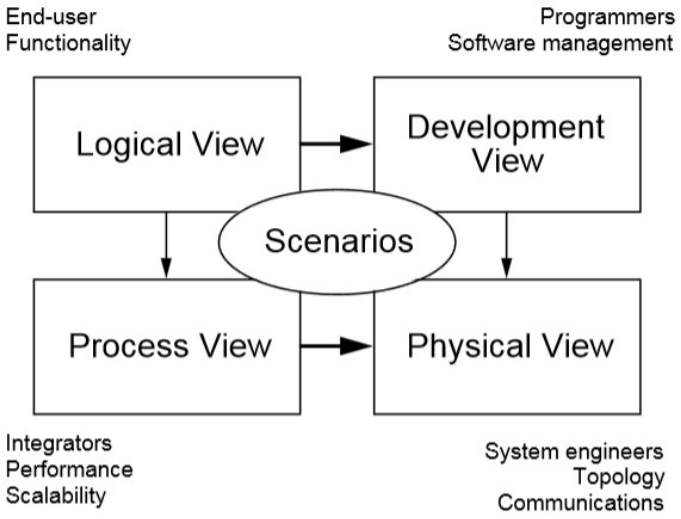


- Why do we actually leverage multiple views?
  ➔ Because capturing all relevant aspects at the same time is too complex
- But remember:
  Models have a purpose! If a model's contents are trivial or overly complicated then their benefit may be 0.
- Textual descriptions are always complementary and sometimes well sufficient.
  So do not use (semi-)formal models for the sake of using models

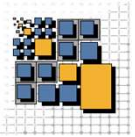
Philippe Kruchten (Rational, IBM) has proposed a renowned framework for documenting software architectures.

You probably will not use the notation, but it is a **really great** tool for checking you cover everything essential!

Philippe Kruchten, IEEE Software 12 (6), pp. 42-50, 1995



4+1 View – Scenarios / Use Case View



- Captures significant Use Cases to answer what users do with the system.
- Captures non-functional requirements (NFRs, qualities) that users expect when using the system
- Example given by Kruchten

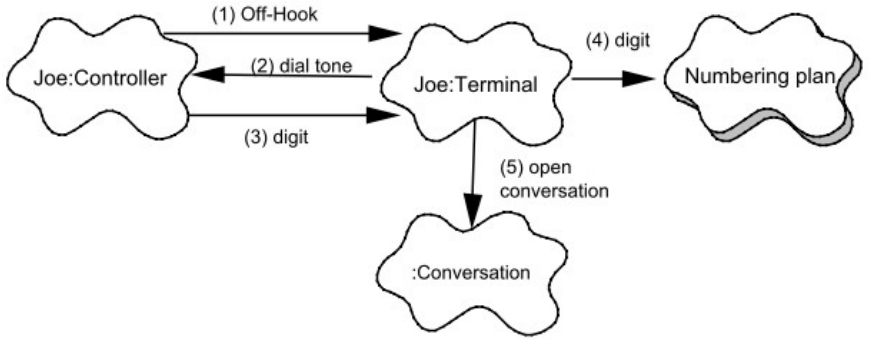
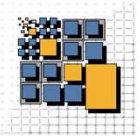


Figure 11 — Embryo of a scenario for a local call—selection phase

- UML / SysML description
  - use case diagrams
  - interaction diagrams

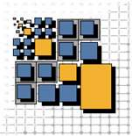
4+1 View – Logical



- Captures the structure of the software:
  Software packages / components + Responsibilities
  + Interfaces

  cf. SOM
- Captures the behavior of the software,
  but only outstandingly important behavior
- Captures the data model
  (not strictly in the paper)
- UML / SysML description formats
  - Class diagrams
  - Activity diagrams
  - State machines
  - Sequence diagrams
  - Block diagrams

4+1 View – Logical - Kruchten's Notation



Philippe Kruchten,
IEEE Software 12 (6),
pp. 42-50, 1995

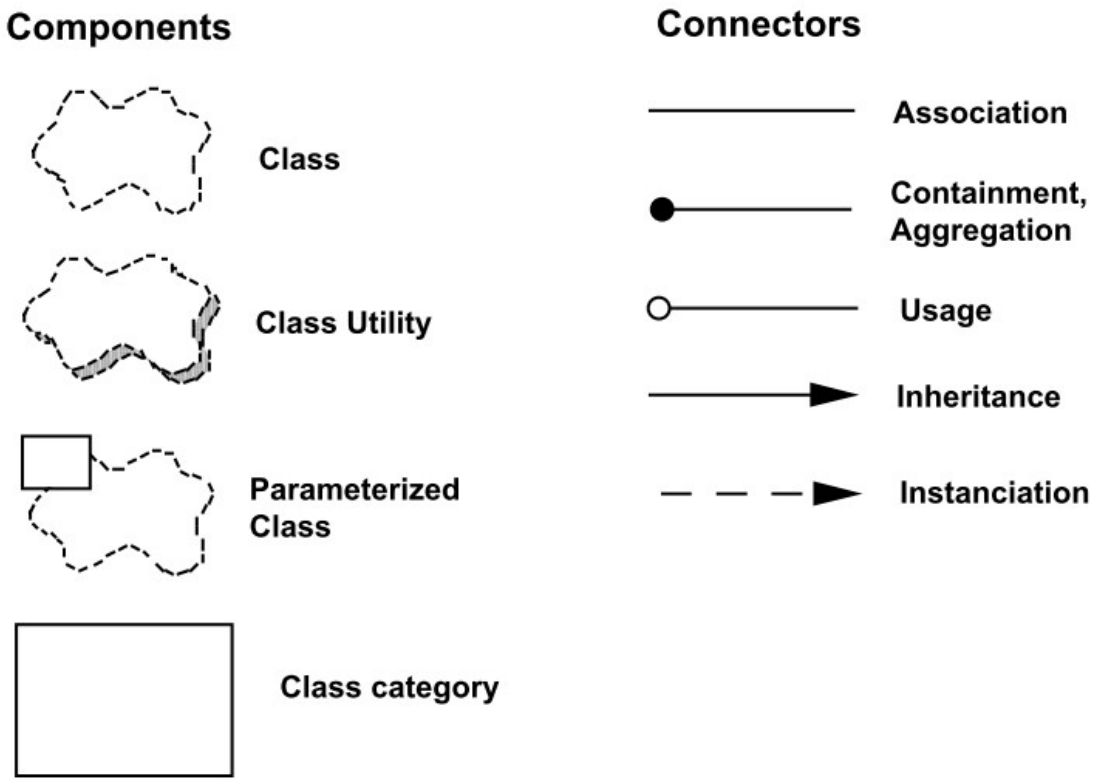


Figure 2 — Notation for the logical blueprint

4+1 View – Logical - Kruchten's Example

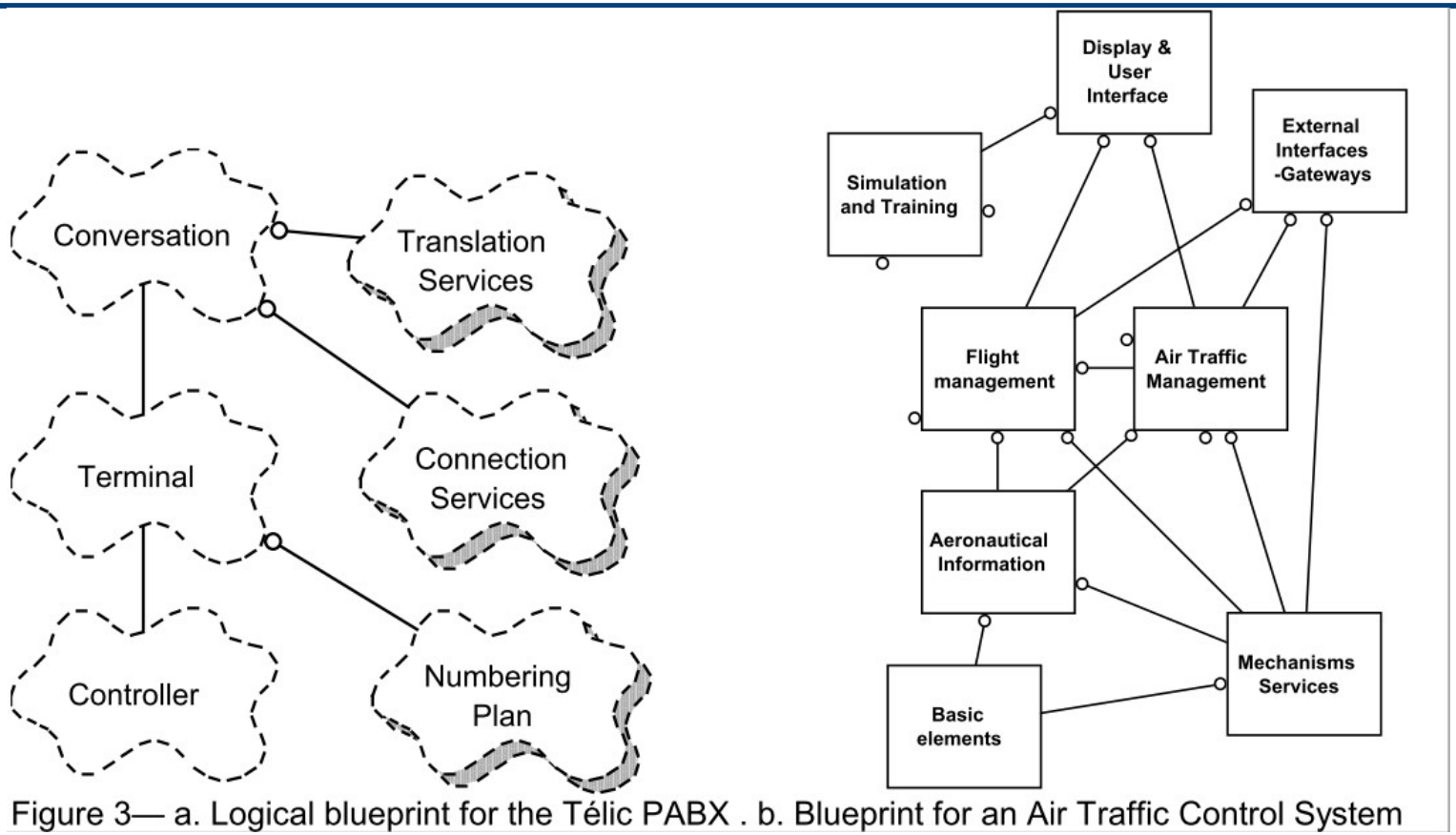
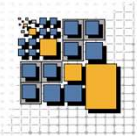


Figure 3— a. Logical blueprint for the Télic PABX . b. Blueprint for an Air Traffic Control System

Philippe Kruchten, IEEE Software 12 (6), pp. 42-50, 1995

4+1 View – Process



- Captures the essentials of how the software maps to computing processes
  - Startup, shutdown, recovery, scheduling, concurrency
  - Refines essential aspects of the logical view

- UML / SysML description formats
  - Interaction diagrams
  - Sequence diagrams

4+1 View – Process – Kruchten's Notation

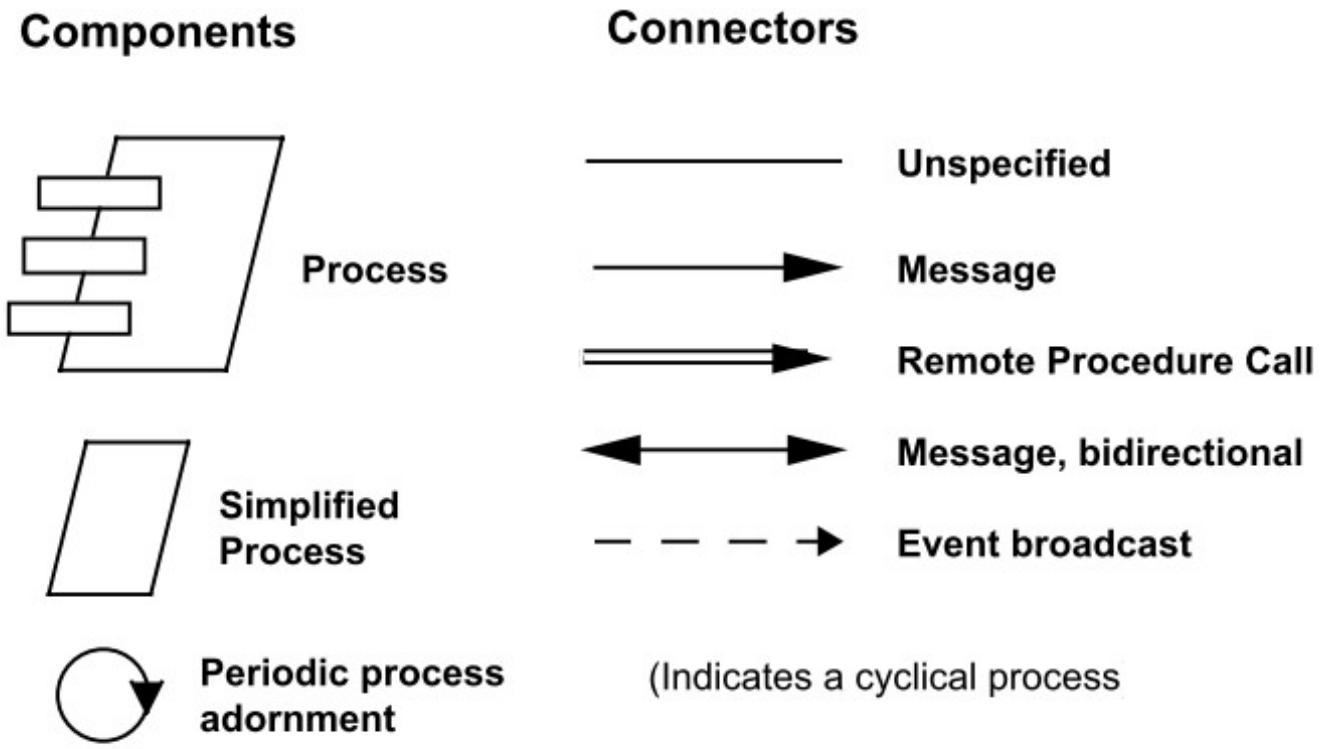


Figure 4 — Notation for the Process blueprint

Philippe Kruchten, IEEE Software 12 (6), pp. 42-50, 1995

4+1 View – Process – Kruchten's Example

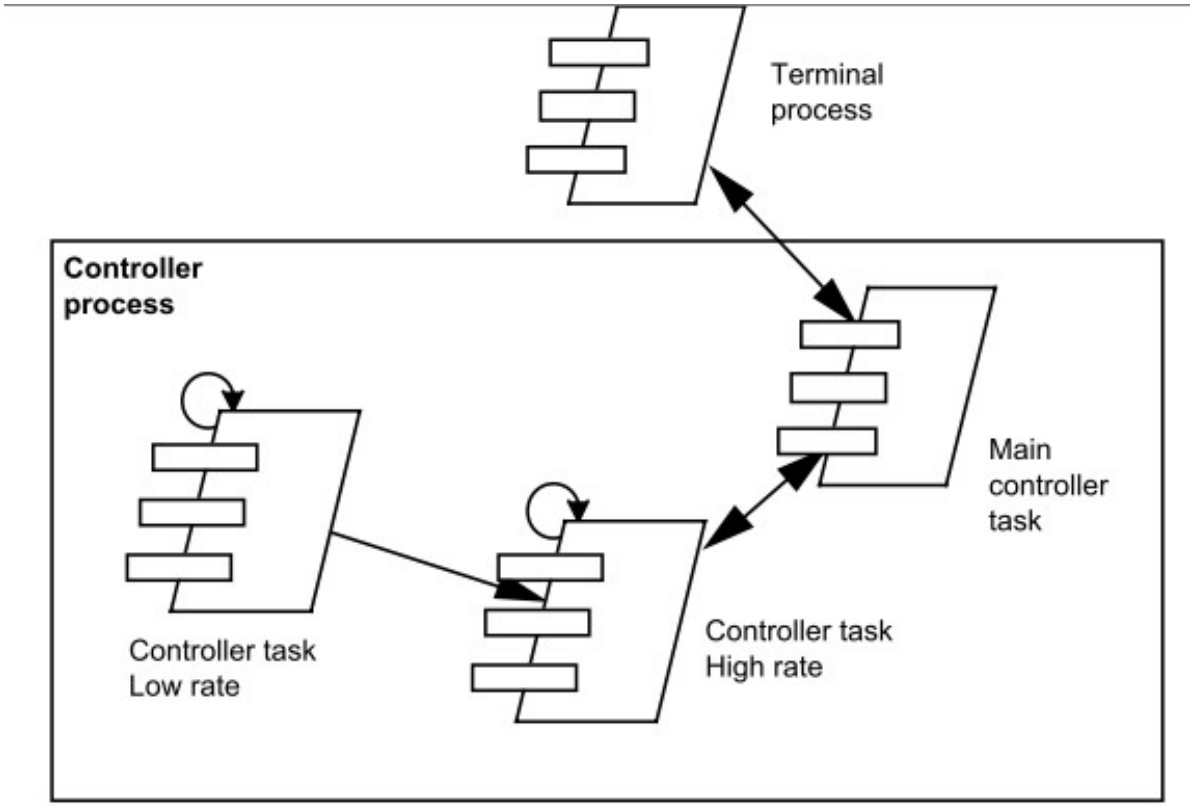
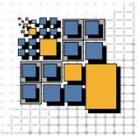


Figure 5 — Process blueprint for the Télic PABX (partial)

Philippe Kruchten, IEEE Software 12 (6), pp. 42-50, 1995

4+1 View – Physical



- Captures how the software **maps to** a given system
  - Physical deployment
  - Networking

- UML / SysML description formats
  - UML deployment diagrams
  - SysML block diagrams

4+1 View – Physical – Kruchten's Notation

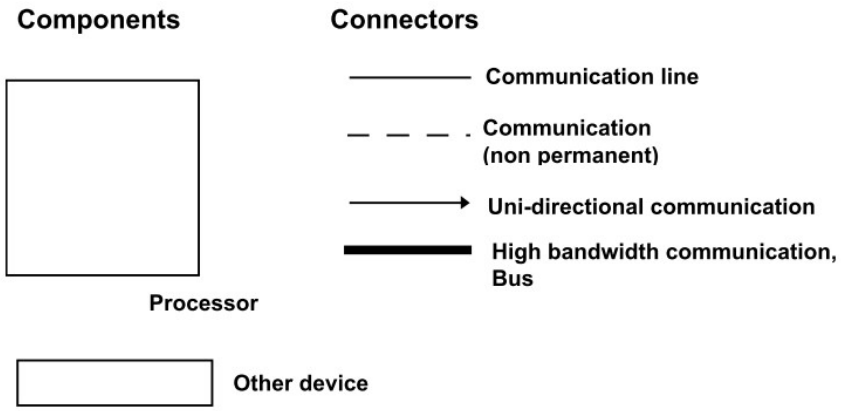


Figure 7 — Notation for the Physical blueprint

Philippe Kruchten,
IEEE Software 12 (6),
pp. 42-50, 1995

4+1 View – Physical – Kruchten's Example

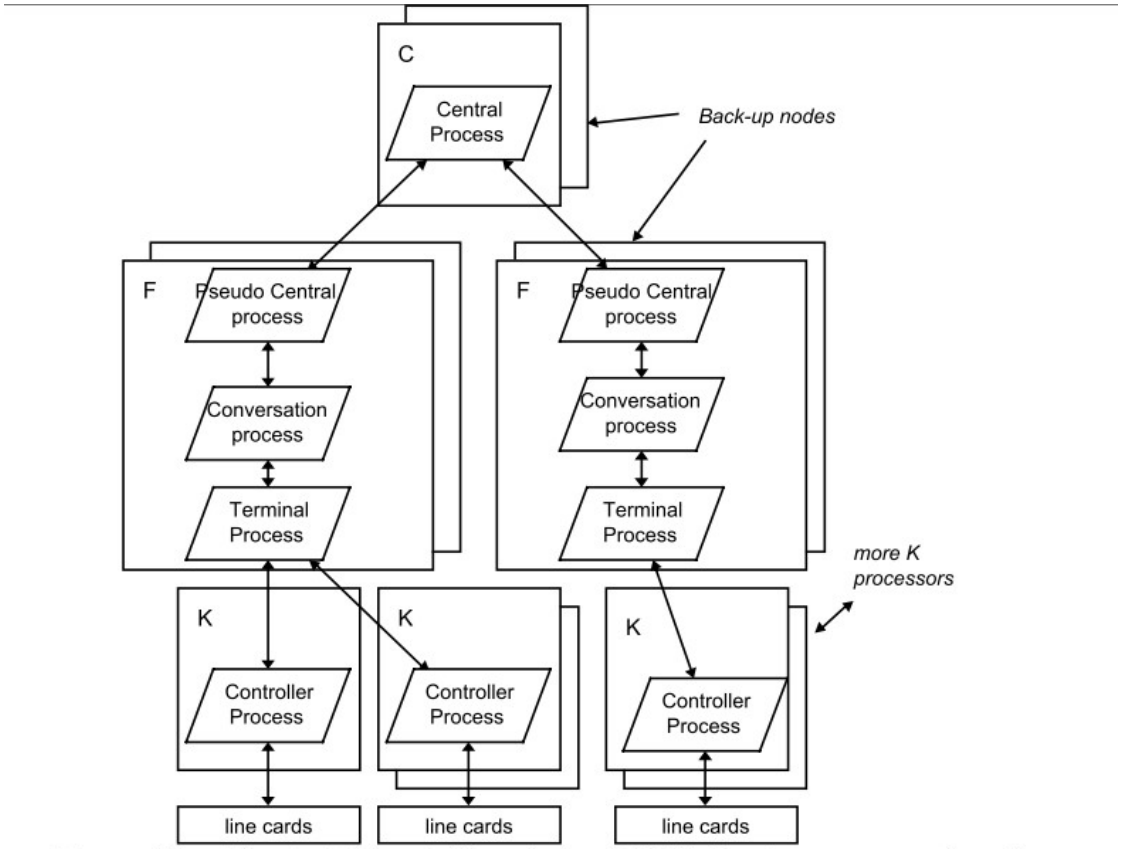
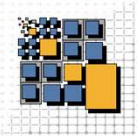


Figure 10 — Physical blueprint for a larger PABX showing process allocation

Philippe Kruchten, IEEE Software 12 (6), pp. 42-50, 1995

4+1 View – Development

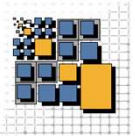


- Captures the technical setup for engineering the system, for example
  - Code organization in file systems
  - Testbeds and test workflow
  - Configuration management

- UML / SysML description formats

  UML / SysML is not dedicated to this purpose, but UML / SysML diagrams **may** come in handy from time to time, for example activity diagrams for describing a requirements workflow

4+1 View – Development – Kruchten's Notation



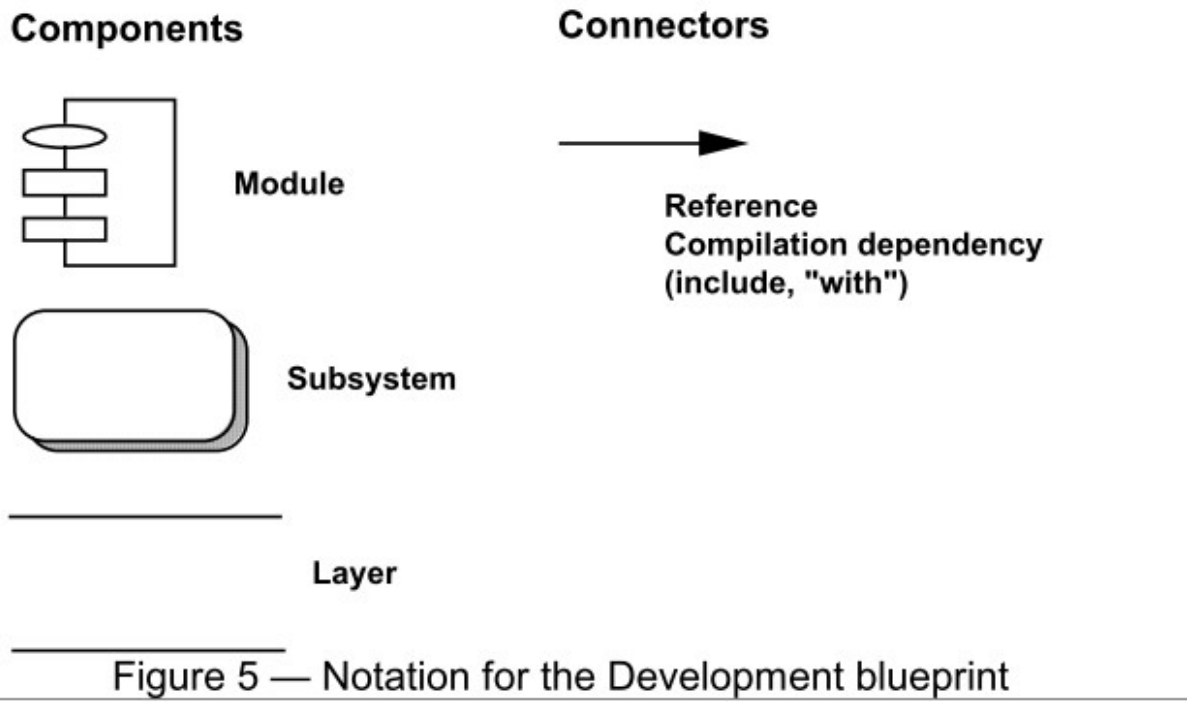


Figure 5 — Notation for the Development blueprint

Philippe Kruchten,
IEEE Software 12 (6),
pp. 42-50, 1995

This notation does the job only partially!

4+1 View – Development – Kruchten's Example

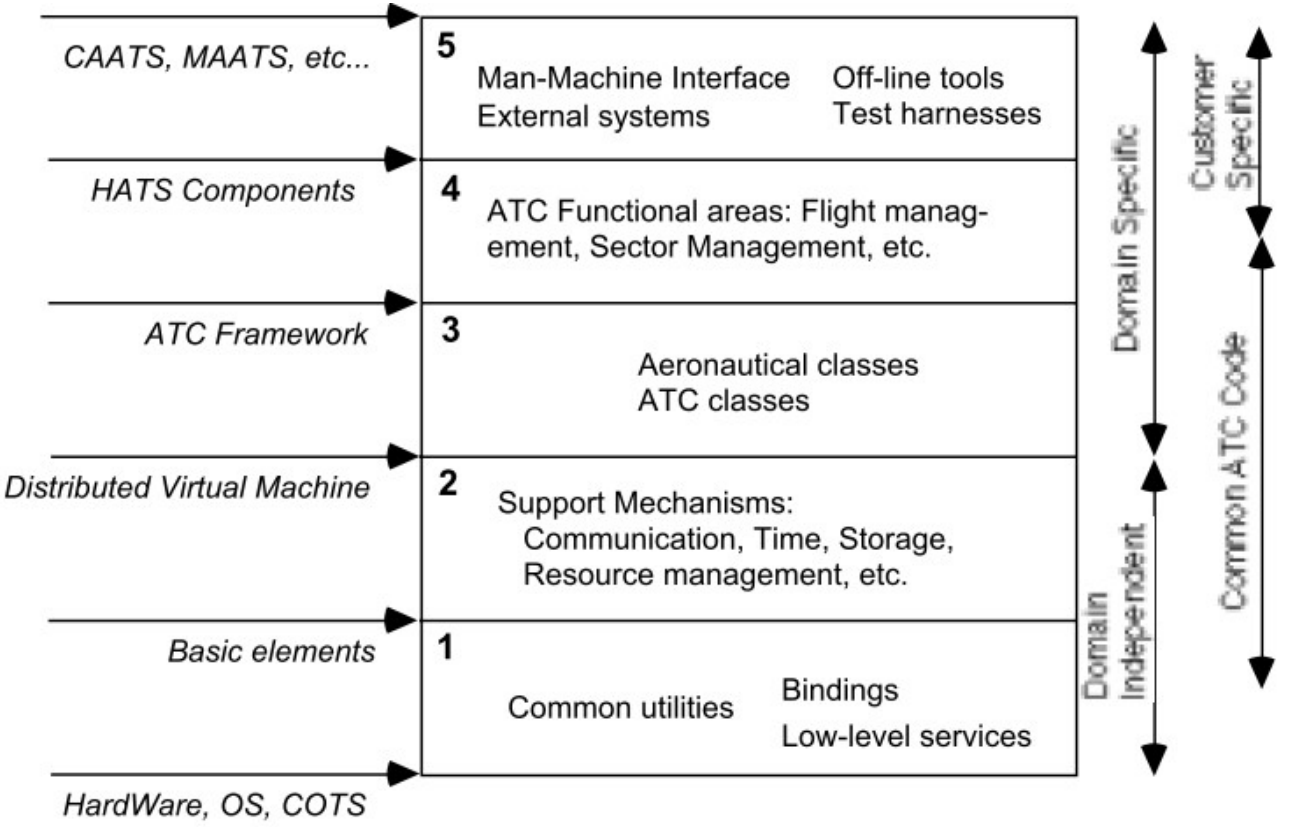
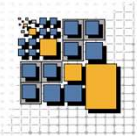


Figure 6 — The 5 layers of Hughes Air Traffic Systems (HATS)

Now we know what to build and document,

but how do we get there?

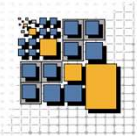
What is Software Architecture Design?



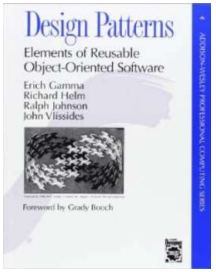
- *Software architecture design maps the problem domain to the solution domain considering forces and risks*
- What are the forces (requirement or constraint)?
  - Functional
  - Non-Functional
  - Infrastructural
  - Non-Technical (organization, processes, business case)
- Architecture design is about taking architectural decisions
  - that have system-wide impact
  - that affect important quality attributes
- Define a baseline architecture
  - as complete as necessary for governing subsequent activities
  - as simple as possible to ensure it can be communicated

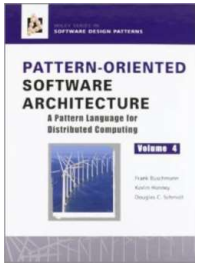
Please check your decisions against reality, in particular do prototype.
There is a lot of truth in code!

Some Best Practices in Software Architecture

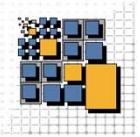


- Neither do BDUF nor NDUF
  - BDUF = Big Design Up Front
  - NDUF = No Design Up Front
- Rely on well-known paradigms, principles and components. This is a big advantage in communicating and enforcing your architecture
- Adopt iterative & incremental development
- Adopt an agile approach
- Adopt a walking skeleton approach
- Adopt design patterns





Iterative & Incremental Development

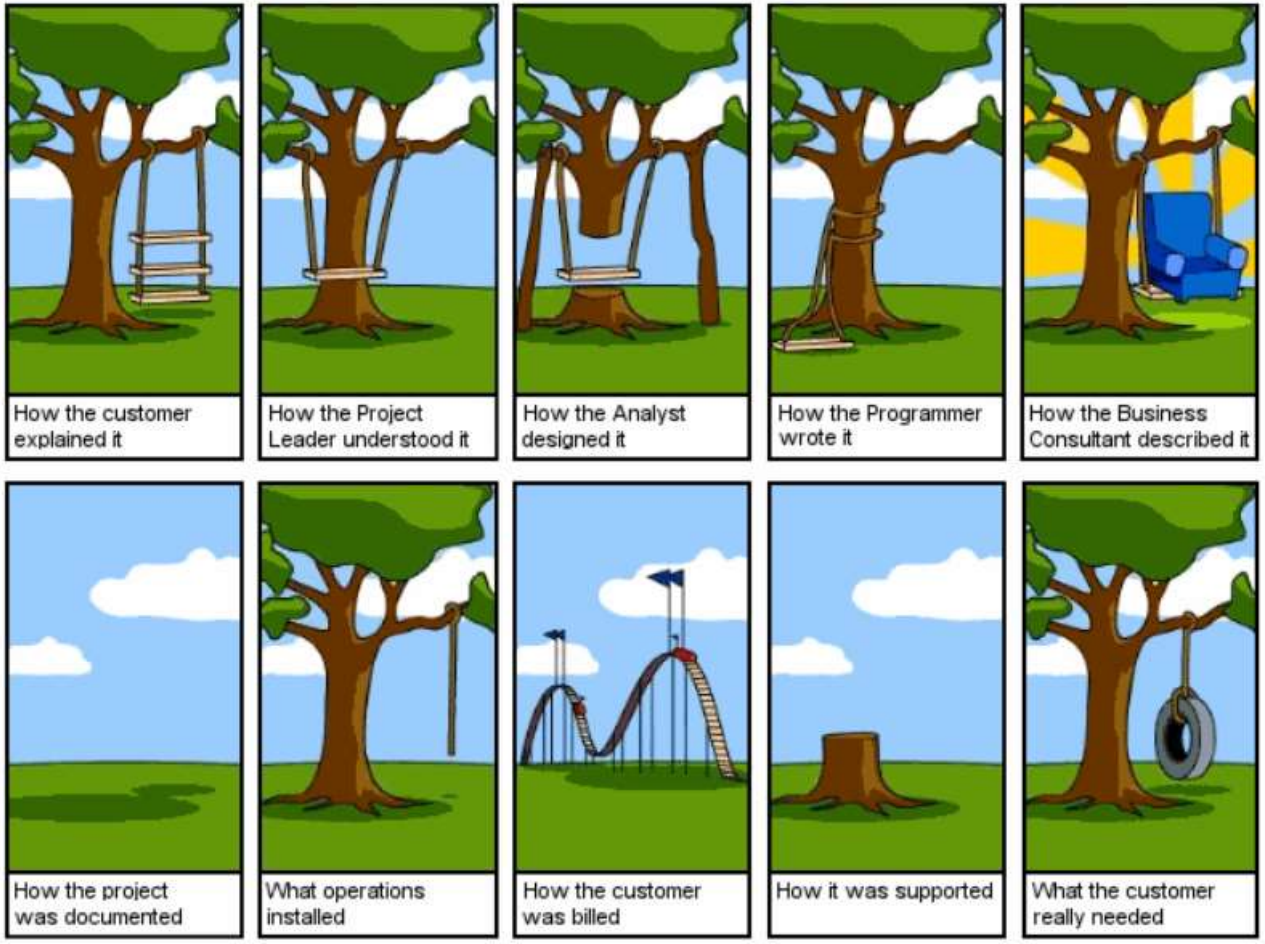


- An iteration is a sequence of activities performed within a period of time (4-8 weeks), typically from requirements to testing

Is 4-8 weeks DevOps style?

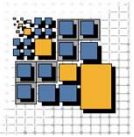
- An increment is a stable executable and testable software release that makes one step forward
- Per iteration
  - do one or more end-to-end slices
  - in product quality
  - addressing major functional or non-functional requirements
- Test early, fail early!
- Take architectural design decisions that must either
  - implement an architectural requirement
  - address a major risk

Why Iterative-incremental Development



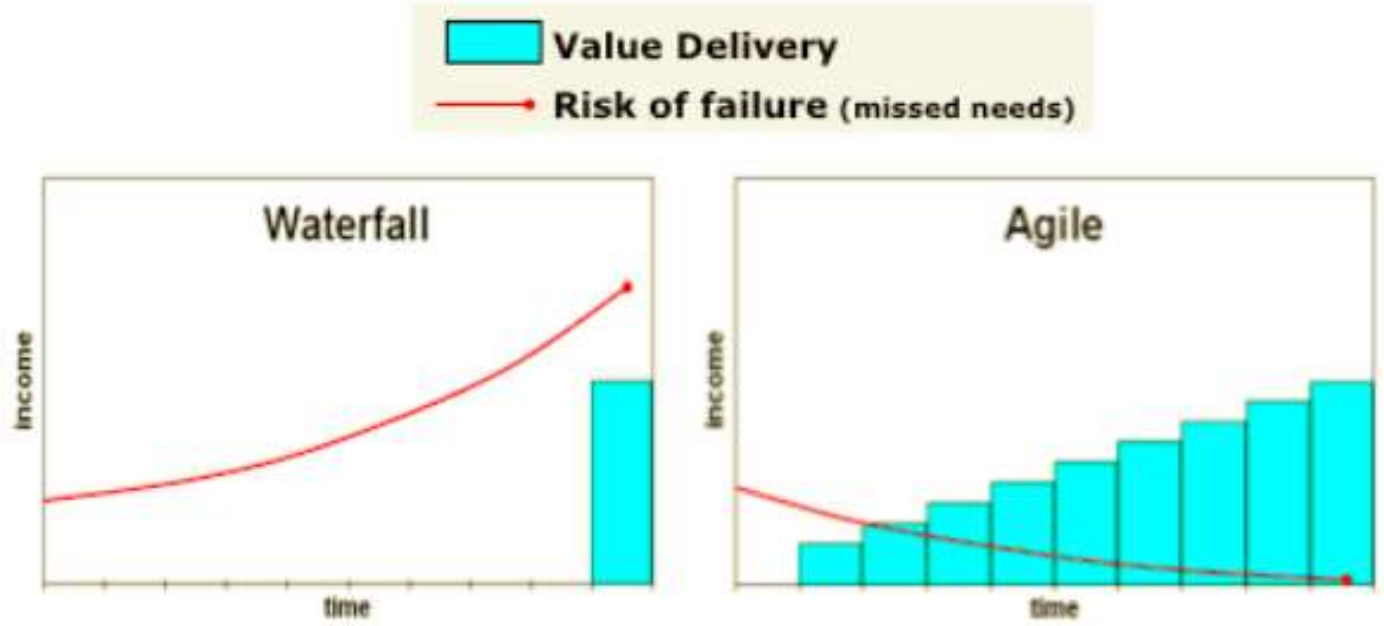
http://codinghorror.typepad.com

Iterative vs. Waterfall

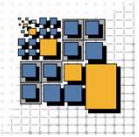


- Software Engineering must embrace change, so try to remove risks as early as possible

https://www.rallydev.com



Manifesto for Agile Software Development



We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

[Please remember:
there is no serious software development without the items on the right]