## Software Engineering: Theory and Practice

**Architecture**

*Tom Verhoeff*

Eindhoven University of Technology
Department of Mathematics & Computer Science
Software Engineering & Technology

Feedback to T.Verhoeff@TUE.NL



## What is Software Architecture?

- The fundamental organization of a system
- embodied in its components,
- their relationships to each other and
- to the environment, and
- principles guiding its design and evolution.

From: IEEE Standard 1471



## Why Software Architecture?

- Organizes communication about the *solution* domain.
- Facilitates parallel construction by a team.
- Improves ability to plan work, track progress.
- Improves verifiability (makes it easier to get it to work):
  - Allows early review of design.
  - Allows unit testing of separate components.
  - Allows stepwise integration (no “big bang”).
- Improves maintainability: doc.; changes affect few components.
- Improves possibilities for reuse.



## Architecture Description: Ingredients

- Stakeholders
- Viewpoints
- Architectural views
- Inconsistencies and conflicts among views
- Rationale, alternatives and why they were not chosen

Compare to architectural description of buildings: spaces and doors, water supplies and drains, electricity, heating/cooling, fire safety, . . .

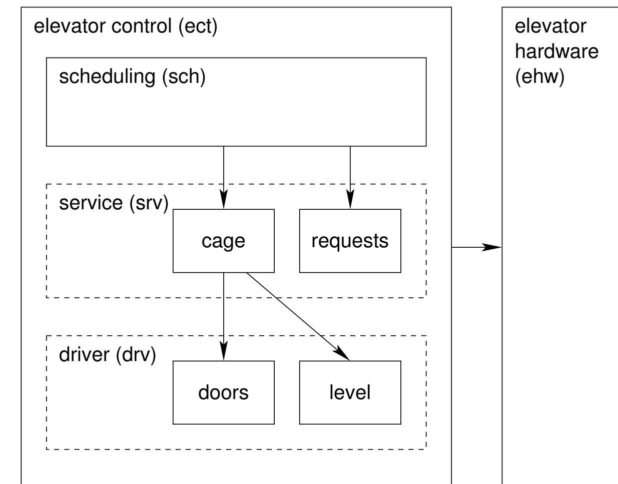
## Kruchten's 4 + 1 Views

| | Static<br>(Structure) | Dynamic<br>(Behavior) |
|---|---|---|
| Abstract | Logical | Process |
| Concrete | Development<br>concerns code in files | Deployment<br>concerns processors |

+ Use case scenarios traced through the architecture



## Example of Logical View






## How to Design an Architecture

Almost any architecture can be made to work, that is, can be made to provide required *functionality*.

*Extra-functional* requirements should drive the architectural design: understandability, verifiability, efficiency, maintainability, . . .

Approaches: Top down, bottom up, yo-yo, functional decomposition, data distribution

KISS: Keep It Simple, Stupid

Consider alternatives and compare them: on paper, by experiment.



## Top down versus bottom up

- Top down: starts from high-level requirements
  - Most important decisions made with least information
  - Requirements are never completely known
  - Risk to reinvent lower-level solutions, instead of reusing them
  - No working code possible until you hit the bottom
- Bottom up: starts from implementation technology
  - Provides no guidance for clear modular structure
- Yo-yo: interleave top-down and bottom-up approaches

## Design Guidelines

- Trace design items and design decisions to requirements
- Minimize coupling between components
- Maximize coherence of components (keep related things together)
- Resolve cross-cutting issues at the architectural level
- Consider alternatives (mention them in the documentation)
- Maximize reusability (through generalization, abstraction)
- Experiment with focused exploratory prototypes



## References

- Example: *Anagrams Architectural Design Document*