WOLF-GIDEON BLEEK

SOFTWARE ARCHITECTURE

From initial Development to Continuous Evolution

Abstract

The term software architecture is used for a variety of aspects of software systems: from the static structure to the dynamic configuration of infrastructures. In this talk I will conceptualize the Hamburg Software Engineering view on software architecture and introduce a software engineering approach that can be characterized as architecture centric software development. With this emphasis not only the creation but also the constant maintenance and further development of systems is supported from the architecture's point of view.

© Wolf-Gideon Bleek

Before we start 1/4

I'd like to know you a little better.

Quick round:

  Name, Organisation, Your Relation to SW Architecture

Before we start 2/4

Let's take a look at some classical architecture

© Wolf-Gideon Bleek

picture taken from the web – copyright may apply

picture taken from the web – copyright may apply

picture taken from the web – copyright may apply

Before we start 3/4

Some General Remarks

Before we start exploring software architectures let us take a look at common questions and misconceptions.

© Wolf-Gideon Bleek

Buildings are unique

- Many buildings are built once – they are a unicum, but some are built several times (e.g. identical houses in a residential area).
- *Software is unique.*
- *Copies are merely a technical problem.*
- Elements of buildings are produced many times, *software elements are only built once.*

A Common Mistake

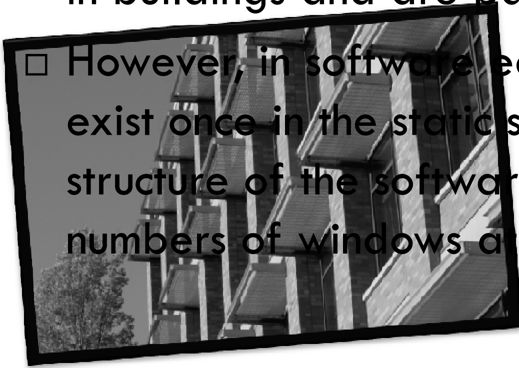
Only carefully apply properties of classic architecture (about the static structure of buildings) to software.

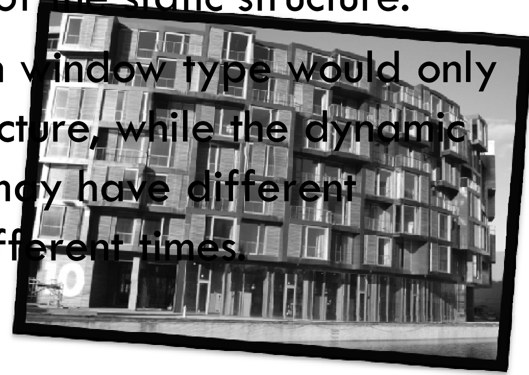
Some characteristics are only due to the physical structure and – in software systems – are reflected by the dynamic structure.

© Wolf-Gideon Bleek

Example 1: Repeating Elements

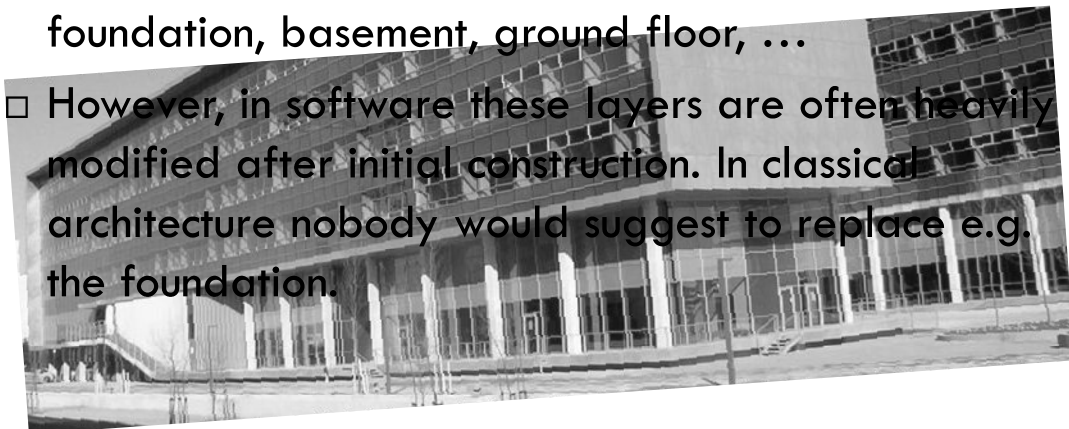
- Windows are an example of a repeating element in buildings and are part of the static structure.
- However, in software each window type would only exist once in the static structure, while the dynamic structure of the software may have different numbers of windows at different times.





Example 2: Layers

- Layering is a typical characteristic of buildings: foundation, basement, ground floor, …
- However, in software these layers are often heavily modified after initial construction. In classical architecture nobody would suggest to replace e.g. the foundation.



© Wolf-Gideon Bleek

Careful

- Architecture is a useful term to discuss Software Systems
- However,
  - not all Characteristics of Classical Architecture apply to Software
  - Software is different!
  - So is Software Architecture!

Before we start 4/4

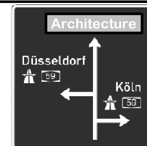
I stand on the work of others

Petra Becker-Pechau
Christiane Floyd
Guido Gryczan
Carola Lilienthal
Joachim Sauer
Axel Schmolitzky
Heinz Züllighoven

to name a few

© Wolf-Gideon Bleek

Overview



- 0. Before we start …
- A. Foundation
  - Software Architecture
  - Perspectives
  - Software Design
  - Quality Requirements
- B. Architecture-centric Software Development
  - Architecture-centric Software Development
  - Designing Software Architecture
  - Architectural Styles
  - Design Criteria
  - Design Patterns, Architectural Patterns
  - Model and Reference Architecture
- C. Continuous Evolution
  - Preserve Architecture
  - Refactoring
  - Renew Architecture

A. Foundation

© Wolf-Gideon Bleek

Questions to start with

Software Architecture:

- Is it layout and distribution of programs in a network?
- Is it the concert of inter-operating programs?
- Is it the runtime structure of objects?
- Is it the static structure of classes?

What is a Software Architecture?

- The Architecture of a Software Systems describes
  - the outside design
  - The inner composition
- The outside Architecture
  - Geographical placement, availability, characteristics of usage
  - Technical Embedding into the base system
    - Operating system, Database, GUI system …
  - Linkage to libraries and frameworks
- The inner Architecture
  - Decomposition of the system into design elements (e. g. modular, object-oriented)
  - Static structure of the design elements and dynamic Interaction of the run-time components

© Wolf-Gideon Bleek

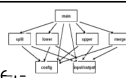
Perspectives on Software Architecture

„The software architecture of a program or computing system is the **structure** or structures of the system, which comprise **software elements**, the **externally visible properties** of those elements, and the **relationships** among them."
– *Bass, Clements, Kazman 2003*

Functional View

Requirements
Co-operation
Usability

Static View

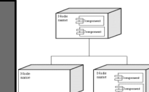


Subsystems, Inter-faces, Layers, Relationships
Responsibilities

Different Schools define different Views:

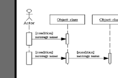
- Kruchten: Logical, Process, Development, Physical View
- Hofmeister et al.: Conceptual, Module, Execution, Code View
- Clements et al: Module, Component and Connector, Allocation View

Distributional View



Machines
Processes
Network

Dynamic View



Interaction
Synchronisation
Data exchange

Software Design – Center of Software Development

- Goals
  - Reduce Complexity
  - Prepare Division of Work
  - Include existing Solutions
  - Enable continuous and further Development
- Quality Requirements
  - Understandability
  - Changeability
  - Reusability
- Means
  - Develop a suitable Software Architecture

Quality Requirement 1: Understandability

By reading the design document one must be able to understand the functionality of the software system!

Levels of Understanding:

- Complete System
  - The Static Structure of the Design Elements should hint at the Dynamic Interaction
- Design Elements
  - Each Design element should consist of a closed set and should not have side effects
- Relationship between Design Elements
  - Direct and indirect Dependencies between Design Elements should be comprehensible

Quality Requirement 2: Changeability

Changes should only have local consequences!

Levels of Changeability:

- Functional Changes
  - Refer to the functionality of the Software System
    affect modeled objects and operations
- Technical Changes
  - Refer to the Base System: Programming Environment, Database, GUI System, Network etc.
- Structural Changes
  - Refer to changes of the inner structure:
    Refactoring = changed structure keeping same behavior

© Wolf-Gideon Bleek

Quality Requirement 3: Reusability

Software Systems or Parts of them should be re-usable!

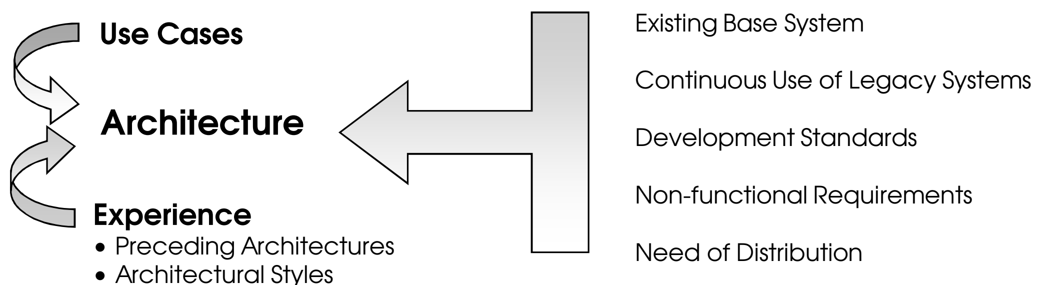
Modes of **Re-Use**

- **Functional**
  - A Solution should be **applicable** to similar problems and should be generalizable to a domain
- **Technical**
  - A Solution should be **transferable** to another base system or used on a class of base systems
- **Investment motivated**
  - Source-Code should be **available** for similar systems, existing libraries and frameworks should be used

Influencing Factors on Software-Architectur

(Jacobson, Booch, Rumbaugh, 1995)

„Architects develop the architecture over several iterations during the inception and elaboration phase. The primary goal of the elaboration phase is to establish a sound architecture in the form of an executable architectural baseline."



B. Architecture-centric Software Dev.

Architecture-centric Software Development

Originally coined by Jaocobson, Booch, Rumbaugh (RUP)
Only within inception and elaboration phase.

However, our experience is that:
During the whole Software Lifecycle Architecture is being developed!

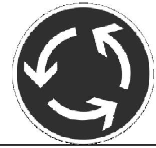
Therefore we argue for different phases:

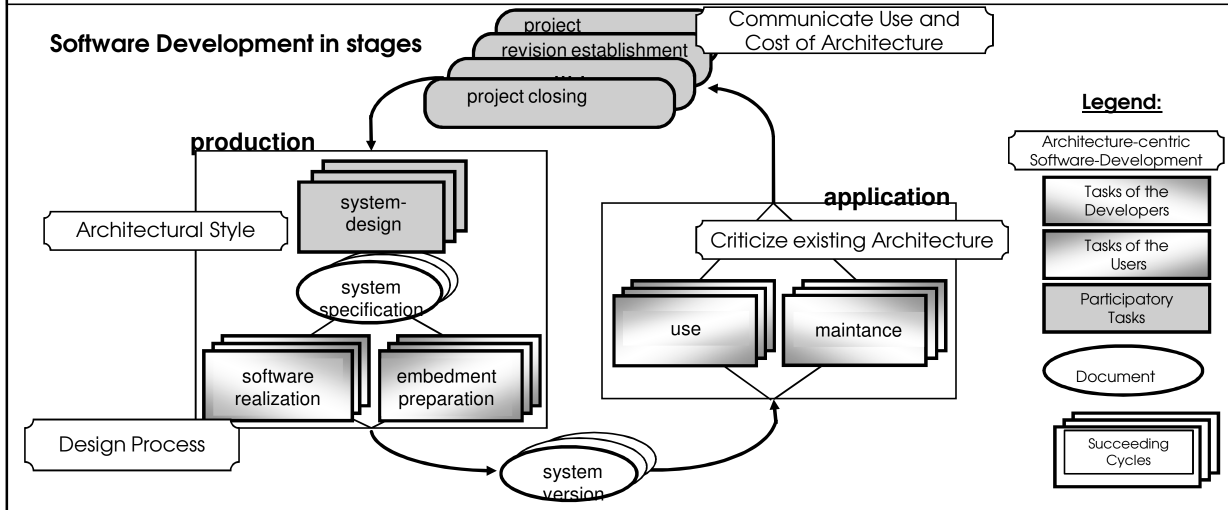
- Design: A new System is developed (in stages)
- Preserve: An existing System is further developed for a new Version
- Renew: A legacy System is refurbished

© Wolf-Gideon Bleek

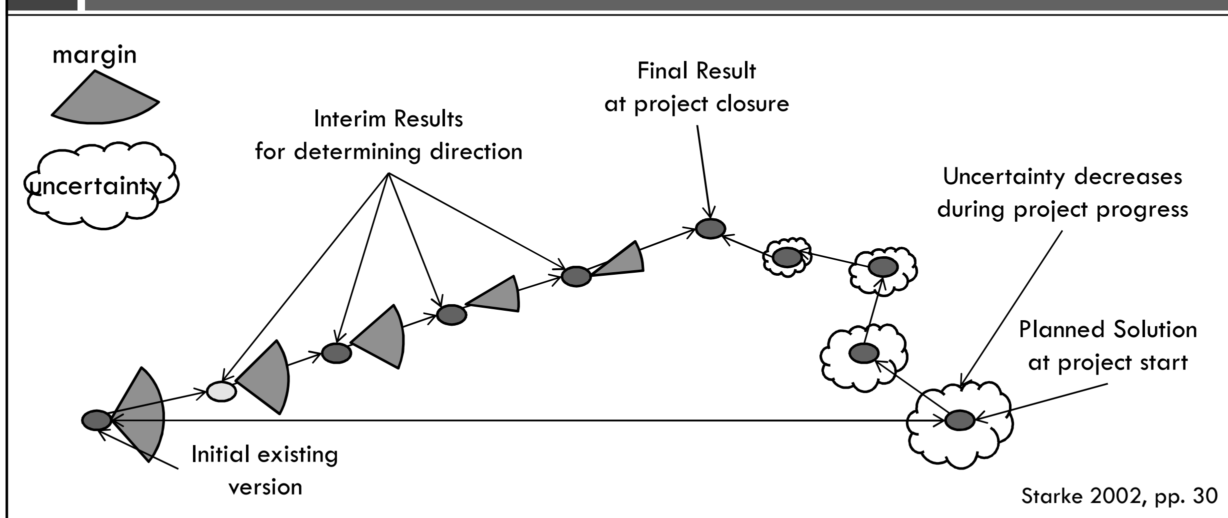
Initial Design of Software Architecture

Applied to the STEPS-Model (Floyd et al. 1989)





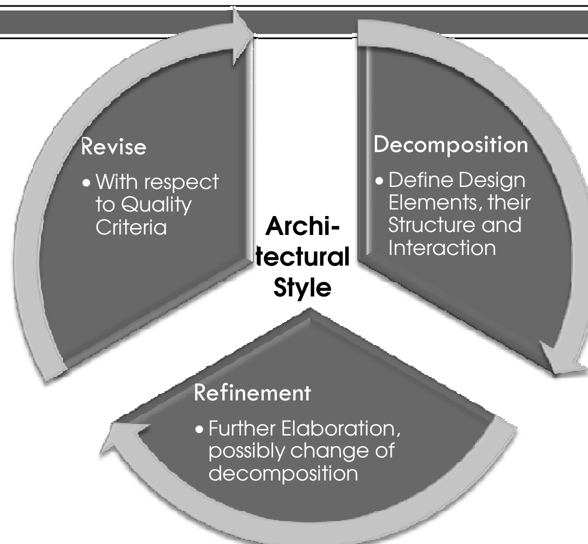
How does an Architecture evolve?



© Wolf-Gideon Bleek

Systematic Process during Design

Software Design is a **Creative Process**, usually consisting of **Many Cycles**, it is oriented towards an **Architectural Style**



Why Architectural Styles?

- Large Software
- Numerous Design Elements
- Complicate Interdependencies
- Multi-Level Decomposition is essential

- Classes of similar Software
- Recurring Design Problems
- Domain specific Systems with similar Structure
- Patterns emerge

Architectural Style: Gothic

The technical term "Architectural Style" was coined by Bruegge & Dutoit 2004

So, what is an Architectural (Software) Style?

- **Architectural Style: Structural Aspects + Construction Rules + Esthetics + …**
  - Here: Interplay of Design Criteria and Architectural Patterns
  - Construction Rules: e.g. **Modular, Functional, Object-Oriented**
- **Design Criteria** – Aspects guiding the Decomposition:
  - High Cohesion, low Coupling,
  - Information Hiding and Encapsulation,
  - Procedural Interfaces and Data Abstraction,
  - Use Hierarchy,
  - Stepwise Abstraction
- **Architectural Patterns**
  - **Design Pattern:** local, standardized Arrangements of Components
  - **Pattern Architectures** (sometimes **Model**- or **Reference-** Architectures): standardized Composition similar Systems
  - Separation of these two is (unfortunately) blurred!

Design Criteria (1): High Cohesion - low Coupling

(Myers 1978, Yourdon & Constantine 1979)

- Cohesion und Coupling
  - define a Qualitative Measure of the Decomposition
  - support Design and Analysis of Software
- A Historic Look
  - Oldest Design Criteria (before 1970):
    - referred originally to seperately compilable units,
    - any programming language
  - Has been further developed for Parnas' Module Concept:
    - referring to the Interplay of Modules and Procedures
  - Has been refined and complemented for object-oriented languages:
    - referring to Classes and Methods

© Wolf-Gideon Bleek

Cohesion (in object-oriented systems)

- Cohesion is used to measure the variety of responsibilities that an element of our object-oriented systems has to fulfill.
- If an element is responsible for exactly one task we call this high cohesion.
- Class Design distinguishes between
  - Cohesion of methods
  - Cohesion of classes
- We strive for a maximum in cohesion.

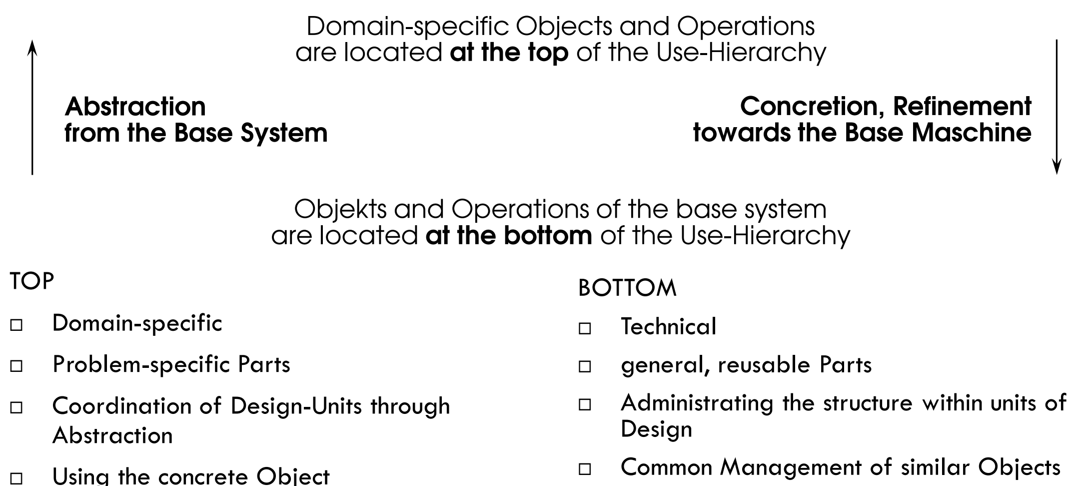
Coupling

- Stands for the dependency between modules
- Should be as low as possible
- Is measured in grades or levels:
  - The highest and most dangerous levels are excluded in modern programming languages (e.g. one modules changes the code of another)
- We strive for Data Coupling:
  - Exchange through procedures (methods) with parameters
  - No cyclic use relationships (e.g. mutual calls)
- Refined for object-oriented Languages:
  - Number of Methods, Number of used Classes, …
- Actions to reduce Coupling
  - Change Interfaces: Selection of Operations and their Parameters

Design Criteria (2): Use-Hierarchy

- Goal: Understandability and Changeability of Designs
- Use-Relation:
  - Dynamic Use : Call of Operations
  - Static Use: Knowledge of Types or Constants
- Use-Hierarchy:
  - If A uses B we require:
    Neither B nor any module used by B makes use of A
- Strong Use-Hierarchy:
  - the System is divided into Layers,
  - use is only allowed from one layer to the next higher layer
- Weak Use-Hierarchy:
  - Layers may be skipped
- Attention: A strict Use-Hierarchy cannot be established in all cases!
  - e.g. using a windowing system

Design Criteria (3): Stepwise Abstraction



TOP

- Domain-specific
- Problem-specific Parts
- Coordination of Design-Units through Abstraction
- Using the concrete Object

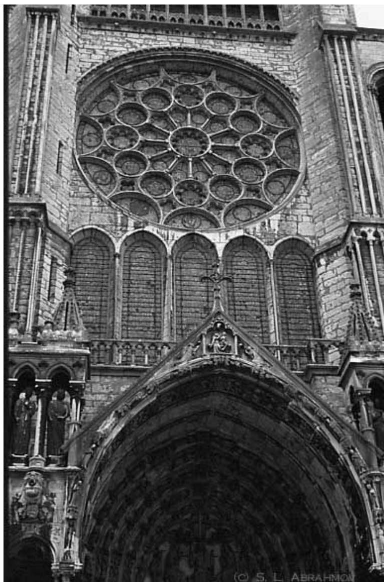
BOTTOM

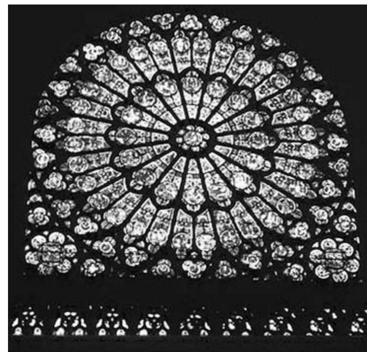
- Technical
- general, reusable Parts
- Administrating the structure within units of Design
- Common Management of similar Objects

© Wolf-Gideon Bleek

Design Patterns

- Describe a repeating Design Problem and a successful Solution
- Are standardized Arrangements of Design Units
- Are applied locally
- Typically there are many different Design Patterns in a Software System
- Common Distinction
  - Creational Patterns
  - Structural Patterns
  - Behavioral Patterns
- Example Singleton: Create exactly one instance of a type





Rosettes in gothic Cathedrals

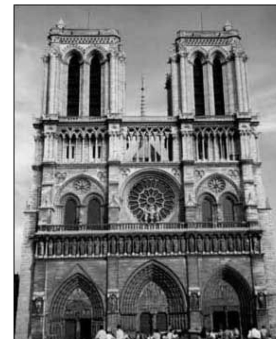
Above: Paris

Left: Chartres

picture taken from the web – copyright may apply

Architectural Patterns

- … describe the Composition of similar Software Systems
- … comprise
  - embedding of large scale systems (e.g. Client-Server-Architecture),
  - interplay with the base system (e.g. window system),
  - the character of the software system (e.g. multi-phase compiler),
  - the way of decomposition (e.g. abstract machine)
- … enrich Architectural Elements

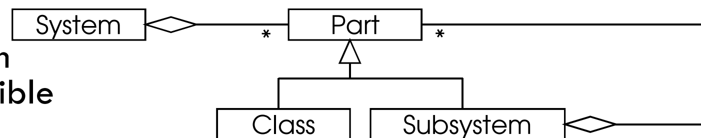


Gothic Cathedrals
outside (Paris, Cologne)
inside (Metz)

picture taken from the web – copyright may apply

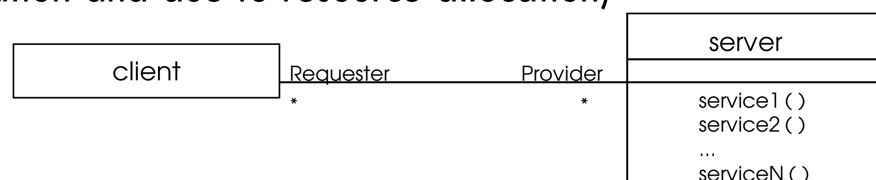
Decomposition into Subsystems

- Subsystems are
  - Part of the designed System
  - Collections of Modules or Classes that belong together
  - frequently assigned to one Developer
  - offering Services for other Subsystems
- A Service is a Set of related Operations with a common purpose
- The Interface of a Subsystem is the collection of Operations offered by a Subsystem
- Recursive Decomposition into Subsystems is possible



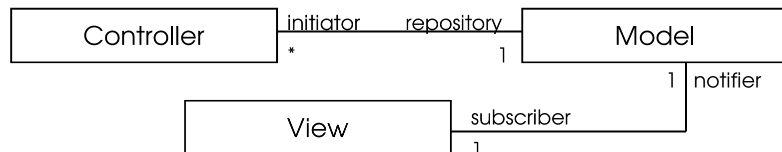
Client/Server-Architecture

- The Service-Subsystem “server” provides commonly accessible operations or services
- One or more Subsystems “client” interact with the server
- The Request for Service is typically performed via a network and will be transformed to conform to technical protocols
- The Flow of Control is usually independent (except for synchronization and due to resource allocation)



Model/View/Controller-Architecture

- Can be seen as Pattern Architecture for a whole System
  - The Subsystem "model" contains the functional model and is domain specific
  - The Subsystem "view" presents this model to a user
  - The Subsystem "control" handles user interaction
- Is related to the (locally applied) Observer Pattern



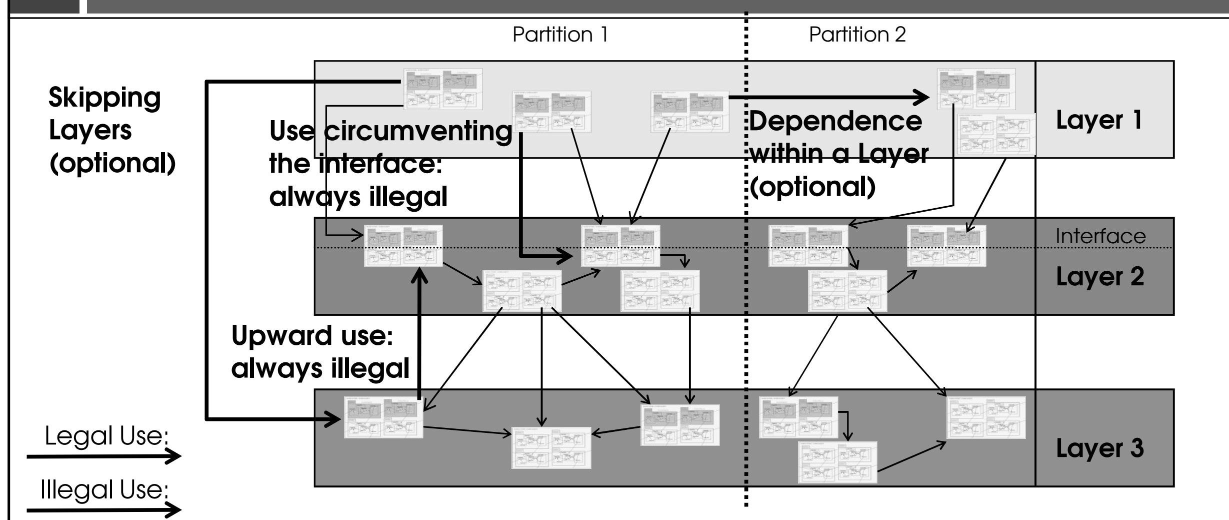
Layers and Partitions

- A hierarchical Decomposition leads to an ordered Set of Layers.
- Each Layer is a Collection of Subsystems (using other layers' services)
- Layers may only access Services from lower Layers (Use-Hierarchy):
  - Closed Architecture: access only to the next Layer (strong Hierarchy)
  - Open Architecture: access also to other lower Layers (weak Hierarchy)
- Partitioning is a way of structuring into coequal Subsystems or Partitions which are only loosely associated
- Commonly a System is structured combining Layers and Partitions

Layer Architecture for interactive Applications

- Basis are three Layers (Three-Tier):
  - Interface layer: contains all objects handling interaction with the outside
  - Application logic layer: contains the functional model's objects and operations
  - Storage layer: realizes persistent data storage, selection and change
- These layers are often refined and differentiated if the application domain requires
- Layers are constituted through their use-relation
  - In general a weak hierarchy
  - Layers are usually depicted top-down or left-to-right

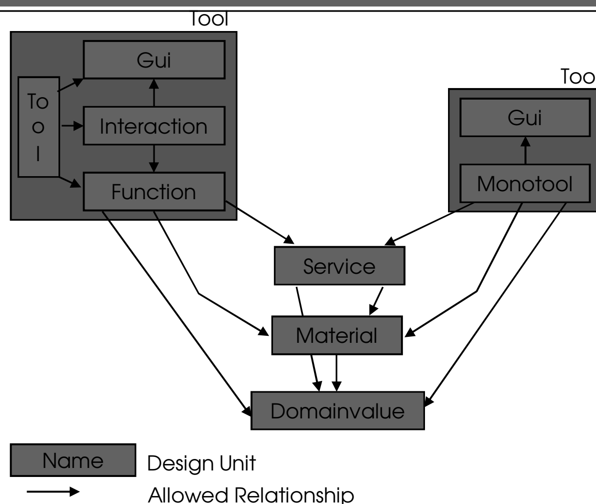
An Example of a Three-Tier-Architecture



Model-/ Reference-Architecture

- Architectural Elements
  - Offer a "Vocabulary": What Elements are there? What are they named?
  - Semantically richer: What is the functional meaning?
- Connections/Associations
  - e.g. Use- or Inheritance-Relation
- Architectural Rules
  - How are Elements developed?
  - How are Element connected?
  - Which Interplay of Elements is expected?
- Example
  - Tools and Material Model-Architecture

Model Architecture – Example Tools and Material



- Metaphors Tools and Material become Design Units
- Class Categories are established:
  - Material
  - Tool
  - Domain value
  - Service
- Detailed Instructions on the Construction of Tools exist
  - e.g. Segregation of Interaction and Function within a Tool
- Design Units are only connected according to rules

C. Continuous Evolution

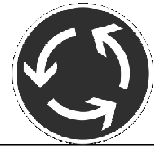
Design Process: How does Architecture come into being?

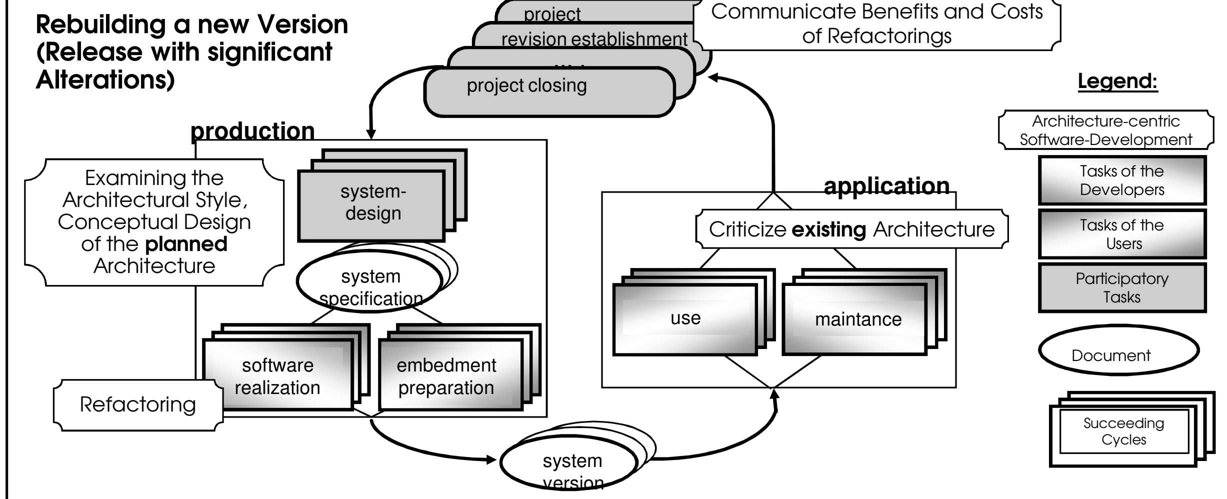
- **Waterfall Process**: Design of the complete Architecture **before** Implementation
- **Iterative/incremental Processes**: Design with **each** Iteration
- **Agile Methods**: Emergent Design – Architecture "arises"
  - Complete Design before Implementation (Big Upfront Design) is unwanted
  - Things have to be repeated three times to be appropriate
  - However, an architectural Style is taken as foundation

© Wolf-Gideon Bleek

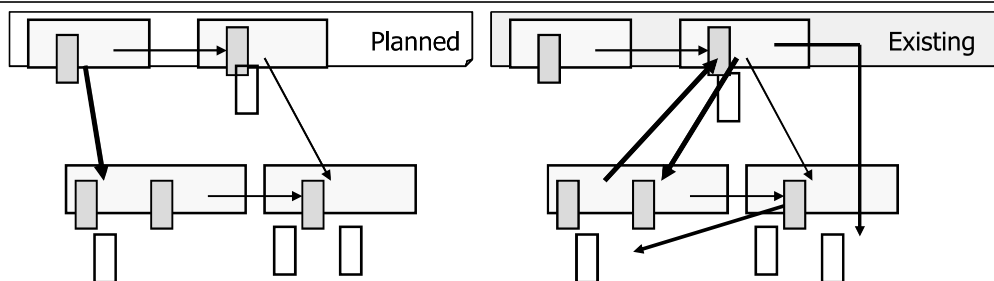
Preserving Architecture

Applied to the STEPS-Model (Floyd et al. 1989)





Consistency Check Planned and Existing Architecture



- consistent Dependencies
- planned but not existing Dependencies (➔ redundance?)
- not planned but existing Dependencies (➔ rule violation)
- interface violation

Refactoring

Agile Methods

- Refactoring comprises
  - Removing Duplicates
  - Simplify complex program logic
  - Clean up ambiguous code

Fowler describes "*bad smells*" *that help developers to identify part in their system that are in need of a refactoring.*

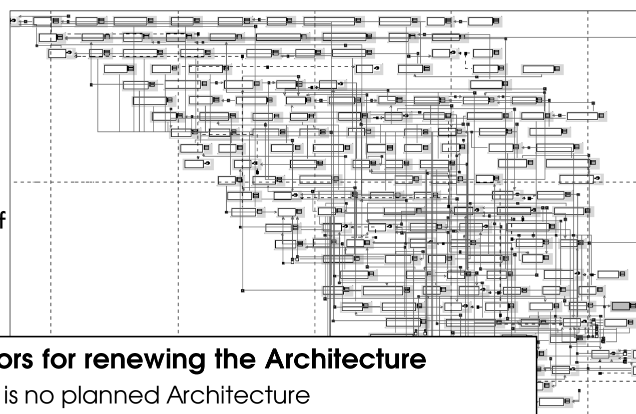
Architecture-centric Software Development

- Refactoring is prerequisite for further development
- Refactoring is used to transform the existing Architecture into the planned Architecture
- Refactoring can only be conducted on the basis of tests

Renewing Architecture – When do we need this?

Causes

- Bad educated developers or missing experience in the team
- Project stress, pressure of time or missing overview
- Transfer of development activities to people that do not know the Intention of this Architecture; Architecture will be alienated
- Changes of the technical Environment
- Heterogeneous Technologies

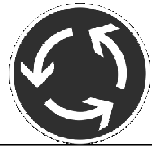


Indicators for renewing the Architecture

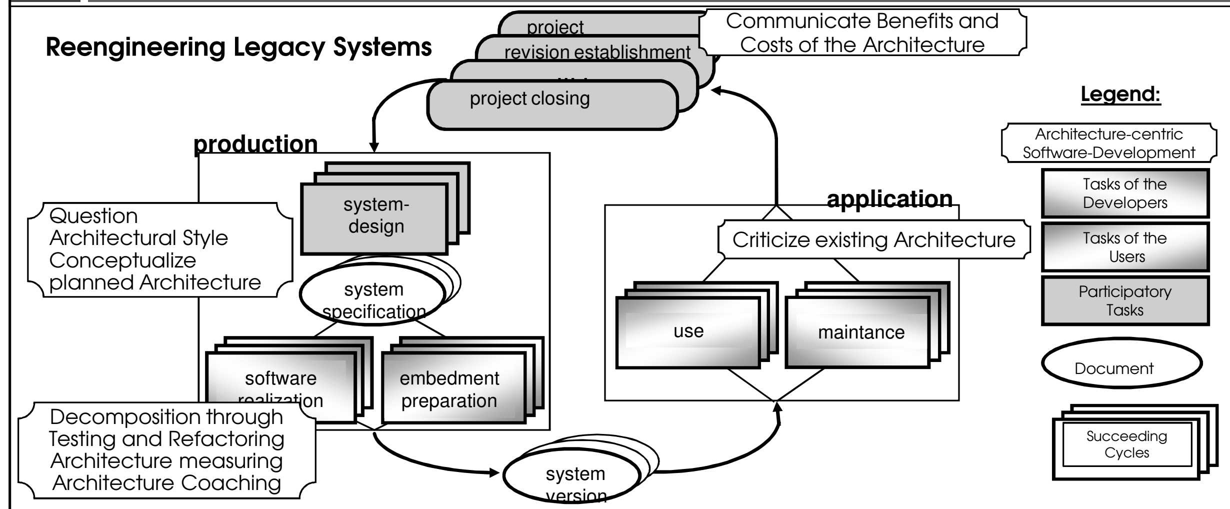
- There is no planned Architecture
- The existing Architecture has substantial problems
- The Architecture is not documented

© Wolf-Gideon Bleek

Renewing Architecture



Applied to the STEPS-Model (Floyd et al. 1989)



Conclusion

- The inner Architecture describes the Structure of a Software System
  - Static View: Decomposition into Design Units and their Relation
  - Dynamic View: Interaction of Run-time Units
  - Functional View: Decomposition with reference to Functional Model
  - Distributional View: Division among Machines and Processes
- An Architectural Style
  - consists of Concepts for Decomposition, Design Criteria and Architectural Patterns
  - supports Design, Preserving and Renewing of an Architecture
- Software Developers should know a repertoire of Architectural Styles an be able to apply them thoughtfully
- Architecture-centric Software Development introduces the Stages Designing, Preserving and Renewing an Architecture.

Mange tak

Har du en spørgsmål?

References

- Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Boston, San Francisco: Addison Wesley, 2003.
- Bruegge, B. & Dutoit, A.H.: Object-Oriented Software Engineering. Using UML, Patterns, and JavaTM, 2. Auflage. Upper Saddle River, NJ: Pearson Education, 2004.
- Gamma, E., Helm, H., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Software. Reading, MA: Addison Wesley, 1994.
- Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Devlopment Process. Reading: Addison Wesley Longman, 1998
- Myers, G.J.: Composite/Structured Design. New York: Van Nostrand Reinhold, 1978.
- Nagl, M.: Softwaretechnik. Methodisches Programmieren im Großen. Berlin: Springer, 1990.
- Parnas, D. L.: On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM, 15[12], 1972, S. 1053-1058.
- Yourdon, E., Constantine, L.L.: Structured Design: Fundamentals of a discipline of computer program and systems design. Englewood Cliffs, NJ: Prentice Hall International, 1979.
- Züllighoven, H.: Object-Oriented Construction Handbook, dpunkt.verlag/Copublication with Morgan-Kaufmann Oktober 2004, ISBN 3-89864-254-2.