Documenting Software Architectures

Nauman bin Ali, Mikael Svahnberg
Nauman.Ali@bth.se
Mikael.Svahnberg@bth.se

Purpose

- Education
- Communication
- Initiate discussions
- Support analysis and evaluations
- Early design decisions
- Resource allocation
  - Organizational resources
  - Hardware resource constraints
- Supports Maintenance

Architecture documentation

- Abstract enough to understand
- Detailed enough to analyze
- Prescribes constraints
- Recounts decisions
- Fulfills different stakeholder needs

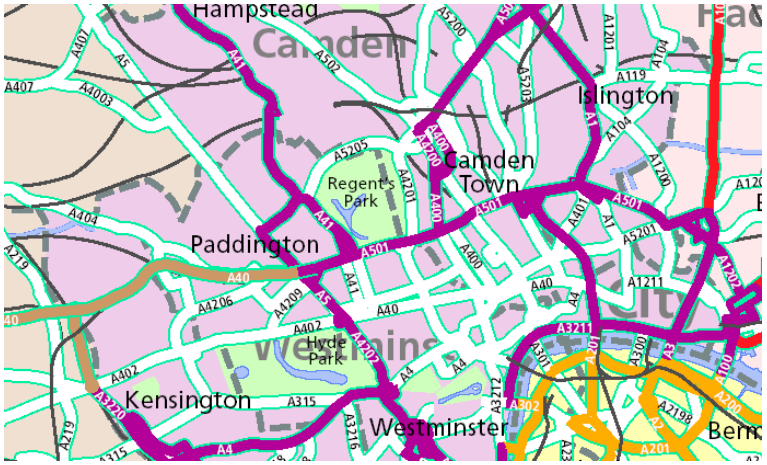
Stakeholders and typical concerns

"One size fits all"

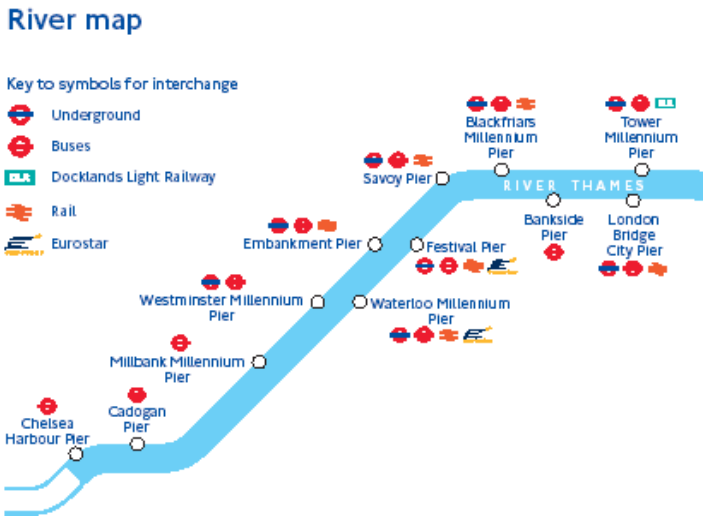
- Architects and Requirements Engineer
  - Negotiation with customer
- Architect
  - Primary communication tool
- Developers
  - Division of work
  - Context
- Testers
  - Basis for test specification
- Management
  - Time and resource Planning
- Customers
  - Insight in the design process
- Quality Assurance team
  - Conformance checking
- Maintainers
  - System understanding
  - Impact analysis

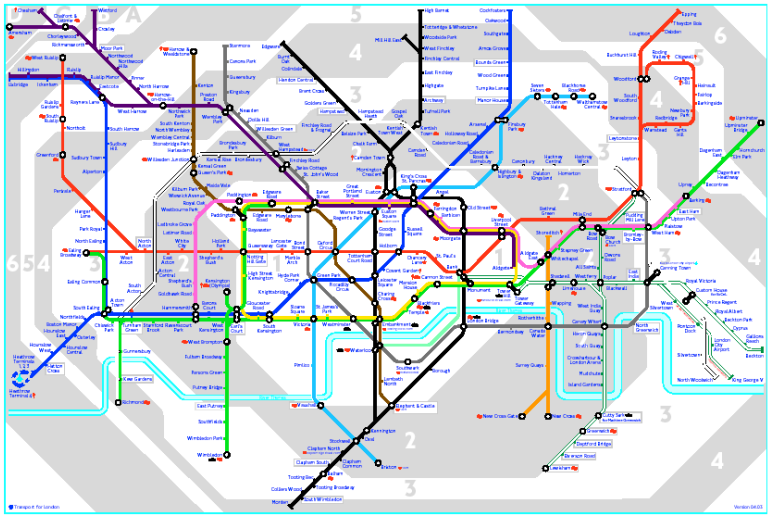
Views

- Other views
  - Electricity
  - Sewage
  - Sightseeing Track



Which of these views represent London?

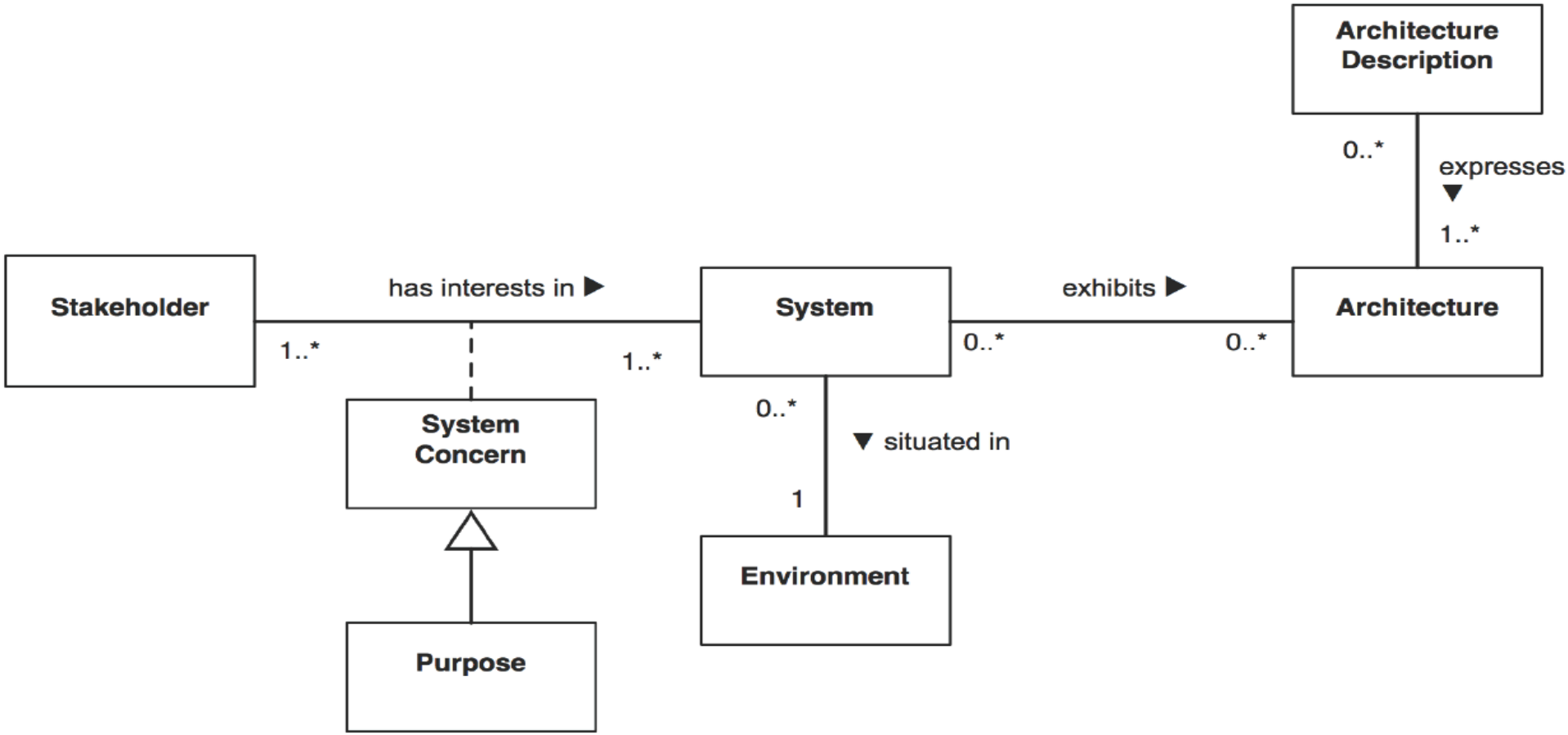




Software Architecture

"Software architecture encompasses the set of significant ***decisions*** *about the* ***organization*** *of a software system including the selection of the* ***structural elements*** *and their interfaces by which the system is composed;* ***behavior*** *as specified in collaboration among those elements;* ***composition*** *of these structural and behavioral elements into larger subsystems; and an architectural style that guides this organization."*

ISO/IEC/IEEE 42010:2011



Architecture views

"*A software architecture is a complex entity that cannot be described in a simple one-dimensional fashion.*"

– Documenting Software Architectures (2002)

No view is the architecture – all views convey it!

– Each view focuses on certain aspects of the system.

Which are the relevant views to use?

– Different views expose different quality attributes to different degrees!

Architecture views

Fixed views

- Kruchten, 1995: the "4+1" approach
- Hofmeister et al.: Siemens Four View model

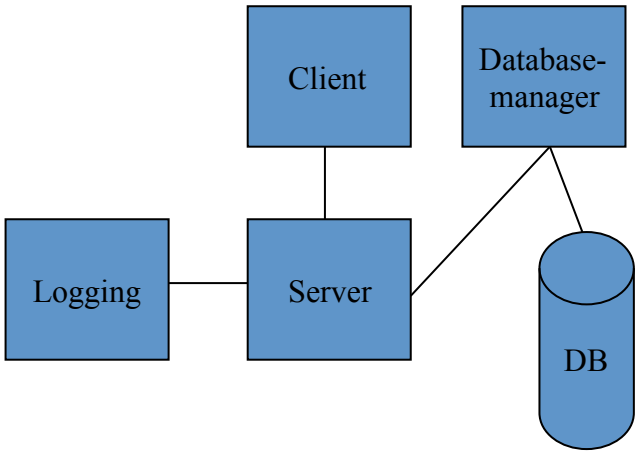
Bass et al.

- Choosing the relevant views
- Documenting a view
  - Documenting behavior
  - Documenting interfaces
- Documenting information that applies to more than one views

Different documentation techniques

- Informal diagrams (e.g. Boxes and lines)
- Structured diagrams
- Formal specification
- Architecture Description Language (ADL)

Boxes and lines: Questions that need answers



- What is the nature of the boxes?
- Do the boxes have similar behaviour?
- What is the significance of the lines?
- How does the behaviour of the parts contribute to the behaviour of the system?
- Does the layout have any meaning?
- Is the sketch realisable?
- How does the architecture operate at runtime?
  - Distribution
  - Multi processor systems

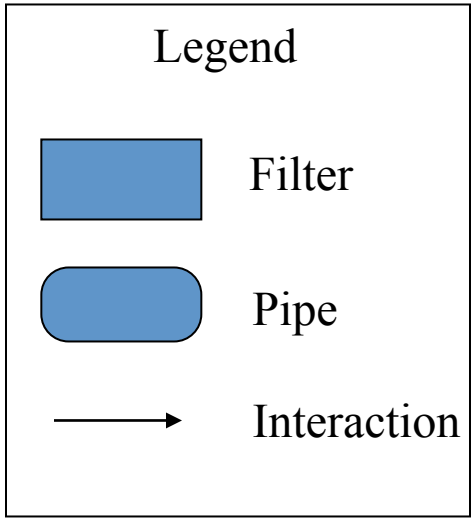
Structured diagrams (1/2)

Add semantics to boxes and lines diagrams!

- Provide a legend

Example: Architectural style-specific notation

- Pipes & filters:



Now we know:

- Boxes are filters, representing computational entities
- Rounded boxes are pipes, controlling data flow
- Arrows denote a use relationship

Structured diagrams (2/2)

Notation often comes with a method:

- Siemens Four View model (Hofmeister et al.) uses extended UML:

Notation is irrelevant!

- Booch, OMT, UML...

Consistent use is most important!

Formal specification

E.g. mathematical specification

Advantages:

- Avoids ambiguities
- Produces precise behavioural models
- Permits rigorous analyses

Example: Z, VDM

Pipes and filters in Z

Filter

$filter\_id : FILTER$
$in\_ports, out\_ports : \mathbb{P}\ PORT$
$alphabets : PORT \pfun \mathbb{P}\ DATA$
$states : \mathbb{P}\ FSTATE$
$start : FSTATE$
$transitions : (FSTATE \times (Partial\_Port\_State)) \rightarrow (FSTATE \times (Partial\_Port\_State))$

$start \in states$
$in\_ports \cap out\_ports = \emptyset$
$\mathrm{dom}\ alphabets = in\_ports \cup out\_ports$
$((s_1, input\_observed), (s_2, output\_generated)) \in transitions \Rightarrow$
$\quad s_1 \in states \land s_2 \in states$
$\quad \land\ \mathrm{dom}\ input\_observed = in\_ports$
$\quad \land\ \mathrm{dom}\ output\_generated = out\_ports$
$\quad \land\ (\forall p : in\_ports \bullet \mathrm{ran}(input\_observed(p)) \subseteq alphabets(p))$
$\quad \land\ (\forall p : out\_ports \bullet \mathrm{ran}(output\_generated(p)) \subseteq alphabets(p))$

Pipe

$source\_filter, sink\_filter : Filter$
$source\_port, sink\_port : PORT$
$alphabet : \mathbb{P}\ DATA$

$source\_port \in source\_filter.out\_ports$
$sink\_port \in sink\_filter.in\_ports$
$source\_filter.alphabets(source\_port) = alphabet$
$sink\_filter.alphabets(sink\_port) = alphabet$

System

$filters : \mathbb{P}\ Filter$
$pipes : \mathbb{P}\ Pipe$

$\forall f_1, f_2 : filters \bullet f_1.filter\_id = f_2.filter\_id \Leftrightarrow f_1 = f_2$
$\forall p : pipes \bullet p.source\_filter \in filters \land p.sink\_filter \in filters$
$\forall f : filters;\ pt : PORT \mid pt \in f.in\_ports \bullet$
$\quad \#\{p : pipes \mid f = p.sink\_filter \land pt = p.sink\_port\} \leq 1$
$\forall f : filters;\ pt : PORT \mid pt \in f.out\_ports \bullet$
$\quad \#\{p : pipes \mid f = p.source\_filter \land pt = p.source\_port\} \leq 1$

Architecture description language

Often domain-specific

Formal specifications

Provides:

- Conceptual framework
- Concrete syntax
- Tool sets
- Constraints

Examples:

Adage

- Avionics navigation and guidance

AADL

- Real-time embedded systems

Rapide

- Simulation

Acme

- Interchange language

Darwin

- Distributed message-passing systems

etc.

Problems with ADLs

- No ADL provides the facilities to completely document an architecture!
- Hard to combine different ADLs
- Requires substantial investments:
  - Education
  - Tools

What is relevant to present

Stakeholders re-visited

- Developers
- Testers
- Management
- Architects

Relevant to Present

Functional Decomposition

Logical Decomposition

- Where does functionality belong

Module Decomposition

- What logical units fit together

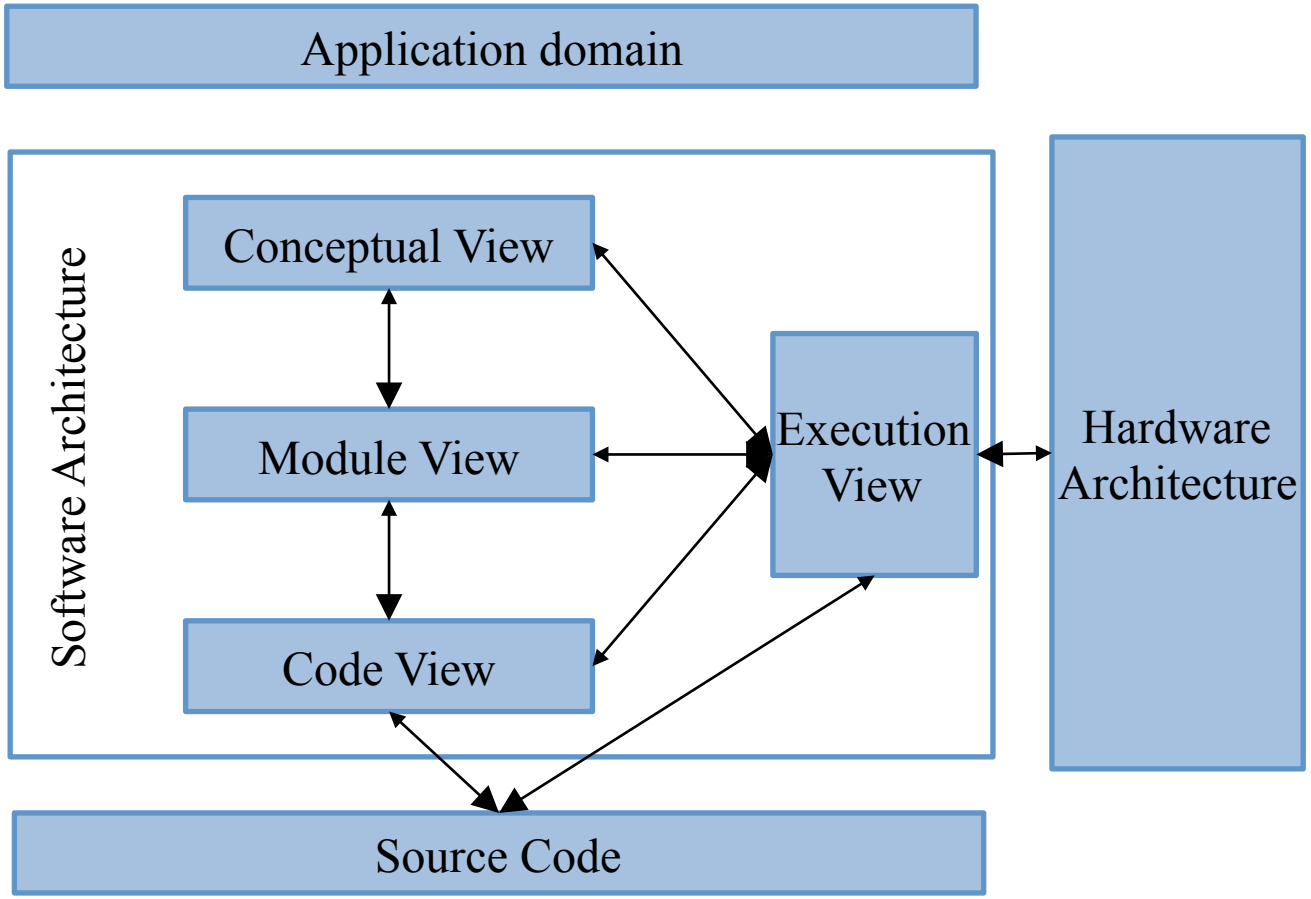
Execution Decomposition

- Distribution across machines

Code Representation

- What files are there, and what versions

Hofmeister's views together



Components & Connectors

Key concepts, Domain terms

Domain experts and developers

Hofmeister's Conceptual View

- Requirements Fulfillment
- Integration of COTS
- Incorporation of Domain-Specific Hardware/ Software
- Product Release Partitioning of Functionality
- Incorporation of Prior Generations, Support for Future Generations
- Support for Product Lines
- Impact of Domain Requirement Changes

Engineering Concerns – Conceptual View

- How does the system fulfill the requirements?
- How are COTS components to be integrated? How do they interact with the rest of the system?
- How is domain specific hardware and/or software incorporated into the system?
- How is functionality partitioned into product releases?
- How does the system incorporate portions of the prior generations of the product and how will it support future generations?
- How are product lines supported?
- How can the impact of changes in requirements or the domain be minimized?

Module View

- Mapping of Product to Software Platform
- Usage of System Support/Services
- Support for Testing
- Dependencies between modules
- Reuse of modules
- Insulation from changes in COTS, software platform or standards

Mapping of Components to Subsystems and Modules

Realization of conceptual model

Engineering Concerns – Module View

- How is the product mapped to the software platform?
- What system support/services does it use, and exactly where?
- How can testing be supported?
- How can dependencies between modules be minimised?
- How can reuse of modules and subsystems be maximised?
- What techniques can be used to insulate the product from changes in COTS software, in the software platform, or changes to standards?

Mapping of Modules to Runtime Entities

Memory Usage

Hardware Assignment

Flow of Control

Performance

Availability

Execution View

- Performance, Recovery and Reconfiguration Requirements
- Balancing of Resource Usage
- Concurrency, Replication, and Distribiution Requirements
- Impact of changes to runtime platform

Engineering Concerns – Execution View

- How does the system meet its performance, recovery and reconfiguration requirements?
- How can one balance resource usage (for example, load balancing)?
- How can one achieve the necessary concurrency, replication and distribution without adding too much complexity to the control algorithms?
- How can the impact of changes in the runtime platform be minimised?

Mapping of Runtime Entities to Deployment Components

Executables, Libraries

Mapping of Modules to Source components

Production of deployment components from source components

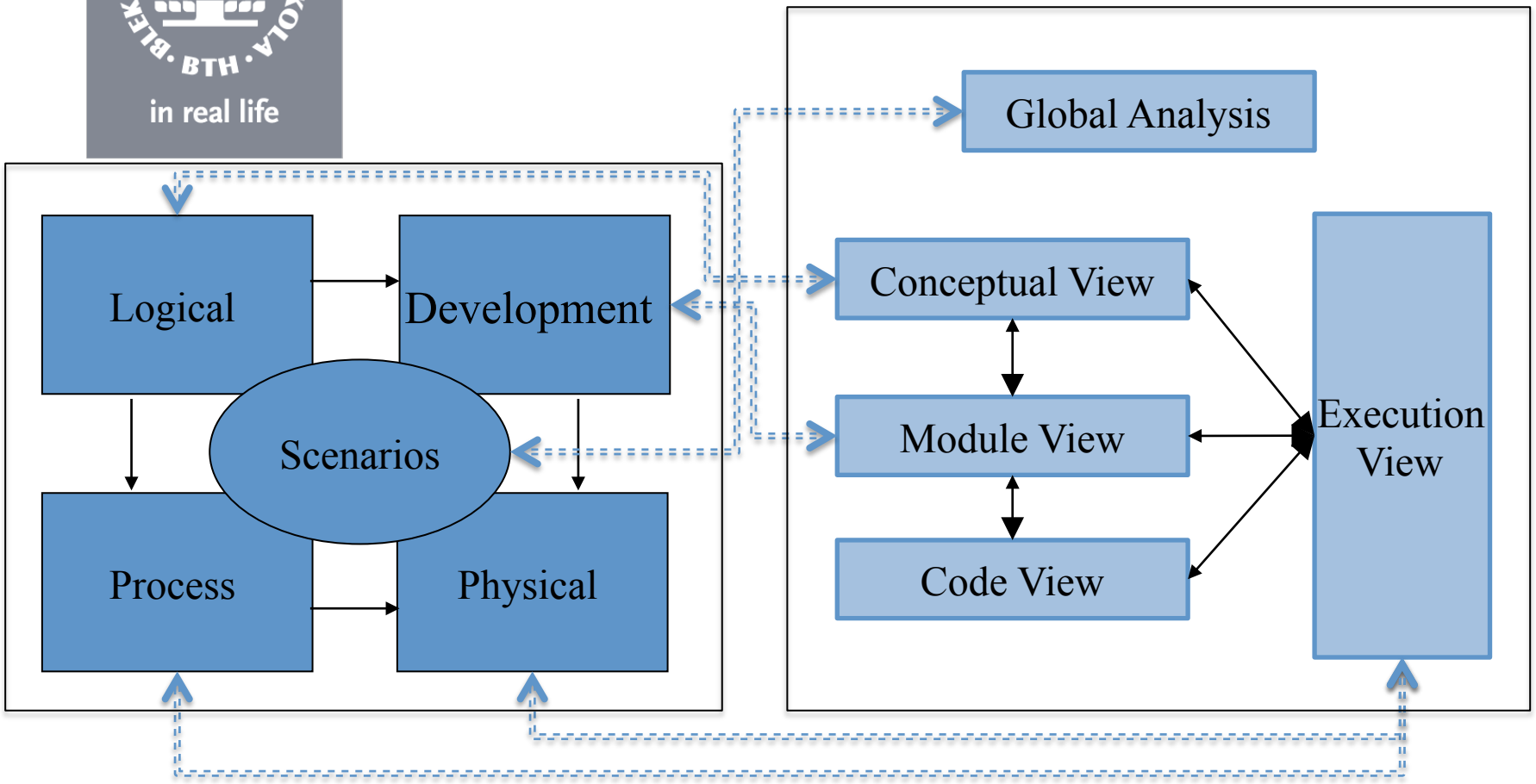
Code View

- Reduction of time and effort for product upgrades
- Version and release management
- Reduction of build time
- Tools for development environment
- Support for integration and testing

Engineering Concerns – Code View

- How can the time and effort for product upgrades be reduced?
- How should product versions and releases be managed?
- How can build time be reduced?
- What tools are needed to support the development environment?
- How are integration and testing supported?

Siemens 4 views and Krutchrn 4+1



Agile approach

- In cases when the documentation is not enough...
  - It's always possible to go back to the whiteboard!
  - Time saved from unnecessary documentation
  - Educate new people by talking to old people
  - Face to face communication
- What about:
  - Availability of architects
  - Overhead to answer the same questions
  - Distributed development teams
  - Maintenance team that inherits the system
  - New developers and architects
  - Employee turnover
- Have a realistic, workable architecture rather than merely well-documented one

"Working software over comprehensive documentation"

Rule 1: Write documentation from the reader's point of view

- A document is written once, read several times.
  - Optimise for "efficiency"!
- It is the most polite thing to do.
- A document written for the reader will be read.
- Avoid unnecessary jargon!

Rule 2: Avoid unnecessary repetition

Record information in one place only.

- Makes the documentation easier to use!
- Makes the documentation easier to change!

But:

- Could repeat for clarity...
- Could repeat for making a different point...

If repetition lowers the reader's cost, do it!

Rule 3: Avoid ambiguity

Architecture design suppresses certain details.

- Does this make the design ambiguous?

Unplanned ambiguity occurs

- when documentation can be interpreted in more than one way, and
- when at least one of the ways is incorrect!

Fight ambiguity with a well-defined notation.

- Explain the notation used!

Rule 4: Use a standard organisation

Establish a standard, planned organisation scheme!

Stick to it!

Helps...

- ...the reader navigate and search the documentation.
- ...the writer plan and organise the contents.
- ...to ensure completeness.

Rule 5: Record rationale

Do not only document design decisions:

- Also document why the decision was made.
- Document the alternatives that were rejected.

The rationale is important for redesign, maintenance etc.

Requires discipline...

- ...but helps saving much time.

Rule 6: Keep documentation current

Change/update the documentation as you make changes.

Incomplete/out-of-date documentation serves no purpose.

Do *not*...

- ...update documentation with non-persistent decisions.
- ...update for every little design decision!

Rule 7: Review documentation for fitness of purpose

Who determines whether a document contains the right information?

- The reader/audience!

What's next

Architecture styles and patterns