Software Architecture

Architectural Design and Patterns. Standard Architectures.

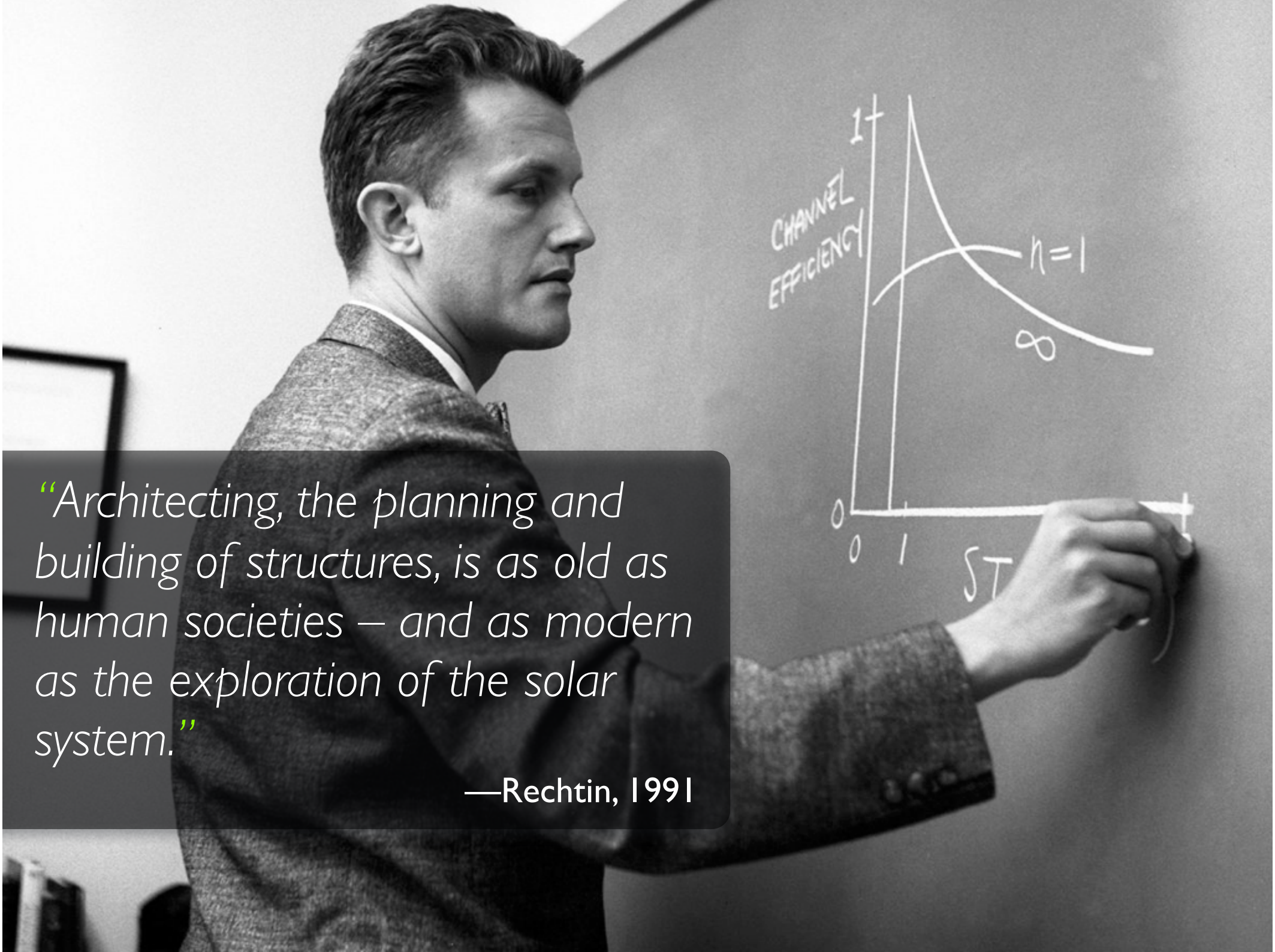
Dr. Philipp Leitner

@xLeitix

University of Zurich, Switzerland

Universität Zürich[UZH]

s.e.a.l.
software evolution & architecture lab

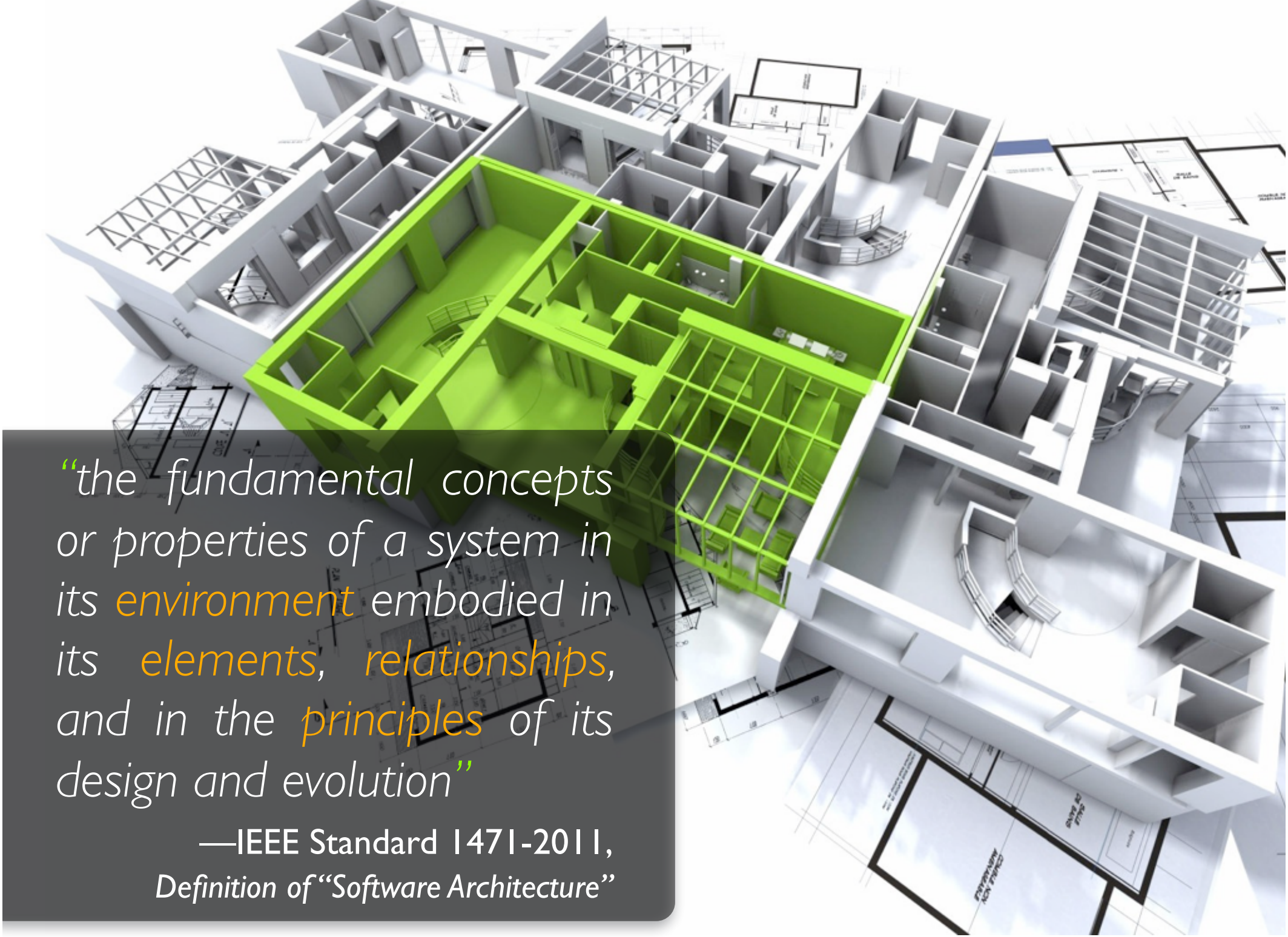


"Architecting, the planning and building of structures, is as old as human societies – and as modern as the exploration of the solar system."

—Rechtin, 1991

Motivation

If the size and complexity of a software system increase,
the global structure of the system becomes more important than the selection of specific algorithms and data structures.



"the fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution"

—**IEEE Standard 1471-2011,**
Definition of "Software Architecture"

About Software Architects

So what does an Software Architect actually do?

Source: http://dilbert.com

About Software Architects

Job Profile (typically):

Hierarchically somewhere between Dev and Team Lead

Often writes some code, but usually not his core job

Needs to focus on the big picture

- And still understand detailed technical issues

Responsible for strategic decisions

- E.g., which technology or architectural style to use

Responsible for decomposing a system into manageable parts

Responsible for integration of said parts after implementation

Responsible for interface definitions

About Software Architects

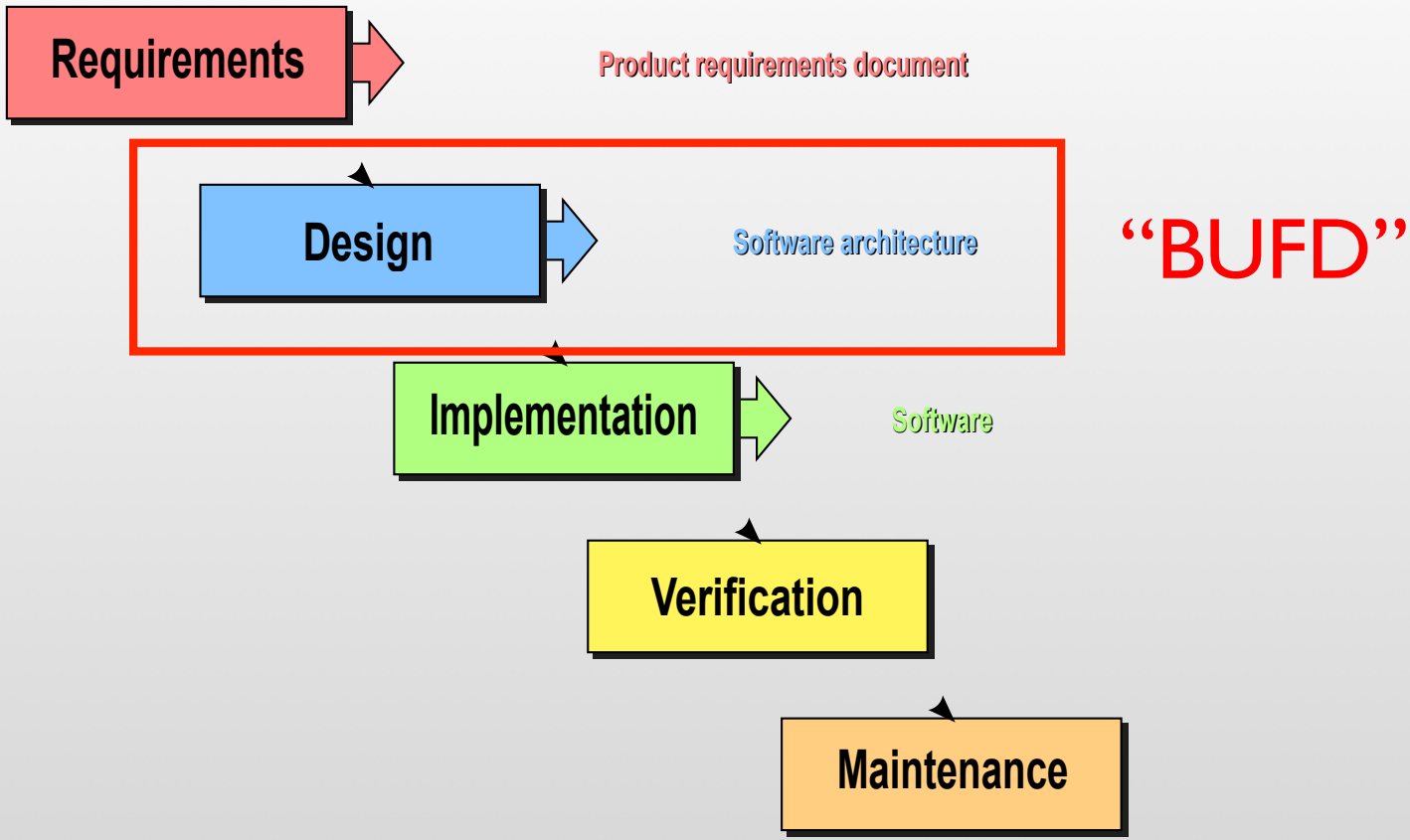
See also:

Ph. Kruchten, "**What do software architects really do?**"
Journal of Systems & Software, vol. 81, pp. 2413-2416, 2008
doi: **10.1016/j.jss.2008.08.025**.

(mandatory reading for this class)

Software Architecture in the Dev Process

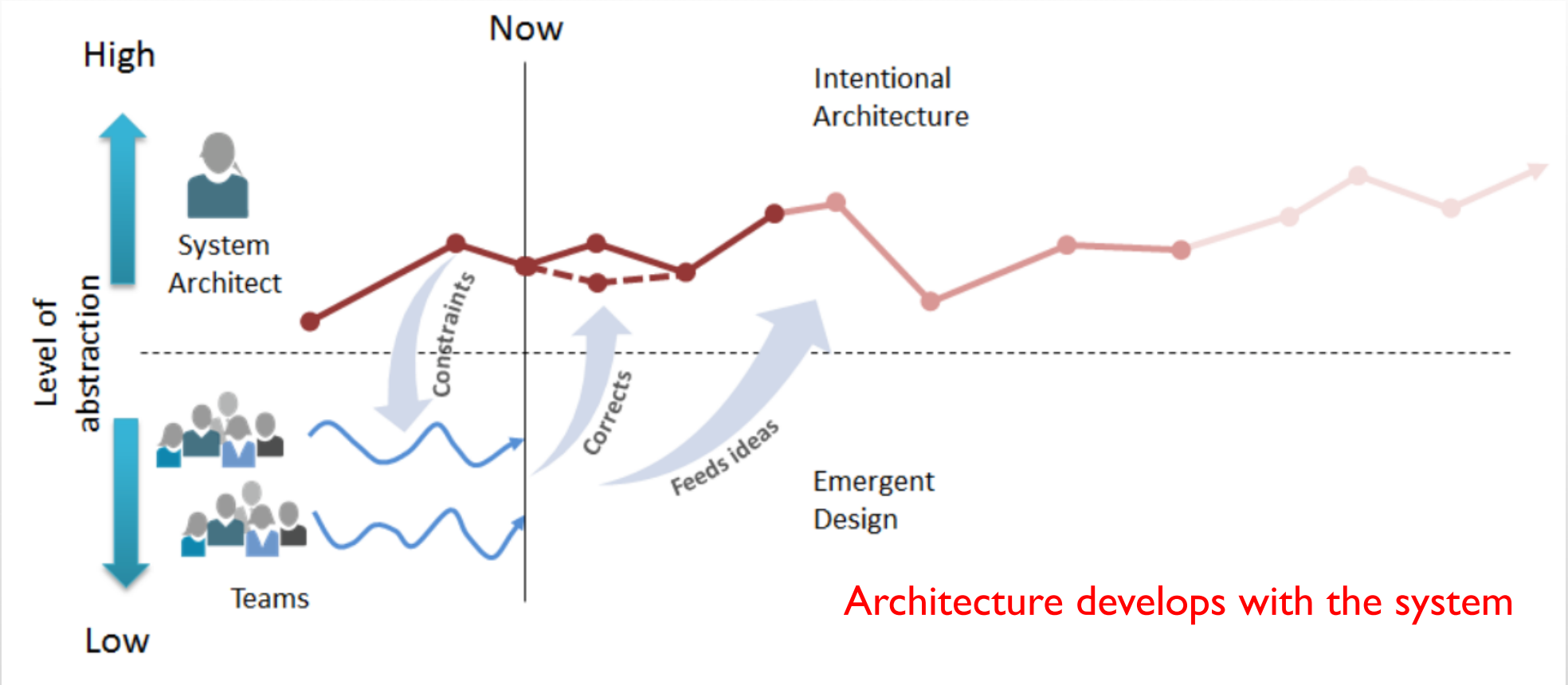
Architecture in the Waterfall Model



Source: http://www.wikipedia.org

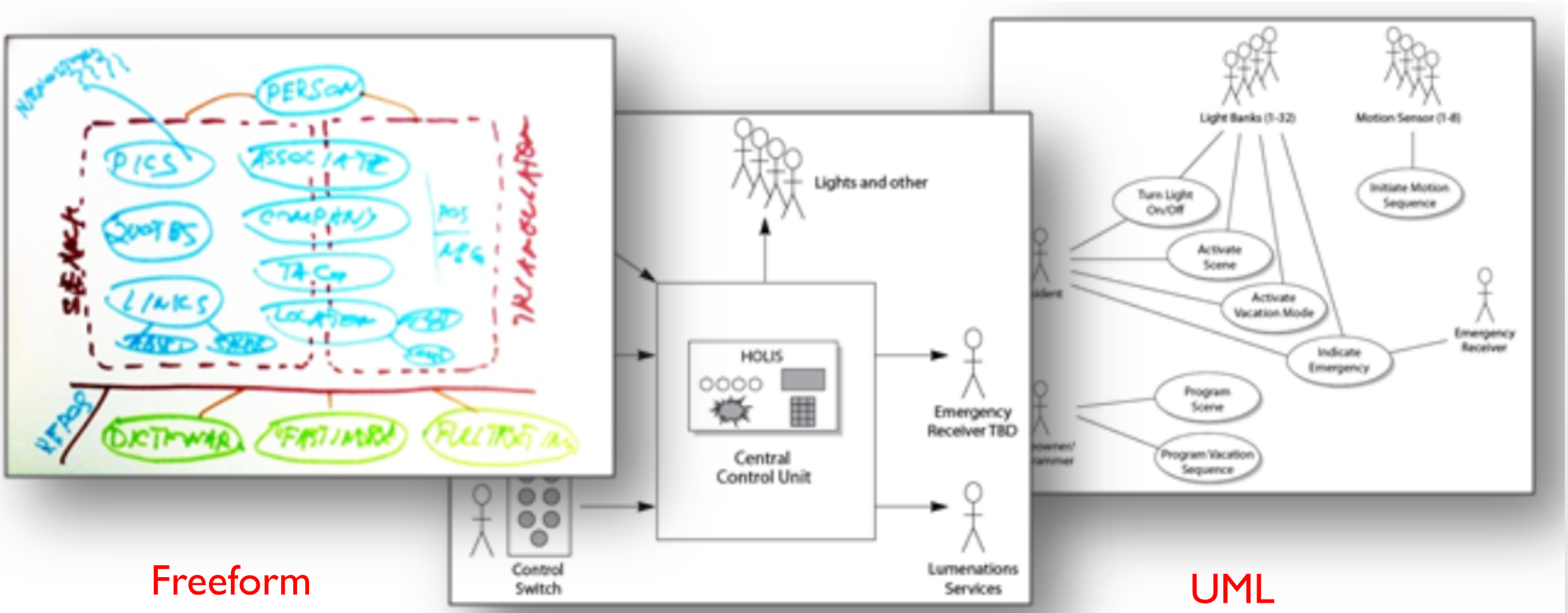
Software Architecture in the Dev Process

Architecture in an Agile Team



Source: http://scaledagileframework.com/the-principles-of-agile-architecture/

Example Architecture



Freeform Diagrams

Architectural Description Languages

UML

Source: http://scaledagileframework.com/the-principles-of-agile-architecture/

Basic Elements of a Software Architecture

Components

Connectors

Constraints

Rationale

What is a Component?

Unit of decomposition of a system

Can be a software package, Web service or module that encapsulates a set of related functions or data.

What is a Connector?

Architectural element that models interactions among components and the rules that govern those interactions

A connector provides interaction duct(s) and transfer of control/data

Components vs. Connectors

Components provide application-specific functionality

Connectors provide application-independent interaction mechanisms

Connectors: Interaction Types

Simple interactions

- Procedure calls
- Shared variable access

Complex interactions

- Client-server protocols
- Database access protocols
- Asynchronous event multi-cast

Roles of Connectors

Protocol specification

- Types of interfaces
- Assurances about interaction properties
- Rules about interaction ordering
- Interaction commitment (e.g., performance)

Nature of Roles

- Communication
- Coordination
- Conversion
- Facilitation

Connectors as Converters

Converters enable the interaction of independently developed components that are mismatched in:

- Type
- Number
- Frequency
- Order

Connectors as Facilitators

Facilitators enable interaction of components by mediation or streamlining. They can:

- govern access to shared information
- ensure proper performance profiles (e.g., load balancing)
- provide synchronization mechanisms (e.g., monitors, guards for critical sections)

What Constraints are there?

Components must be constrained to provide that:

- the requirements are met
- the required functionality is achieved
- no functionality is duplicated
- the required performance is achieved
- modularity is realized

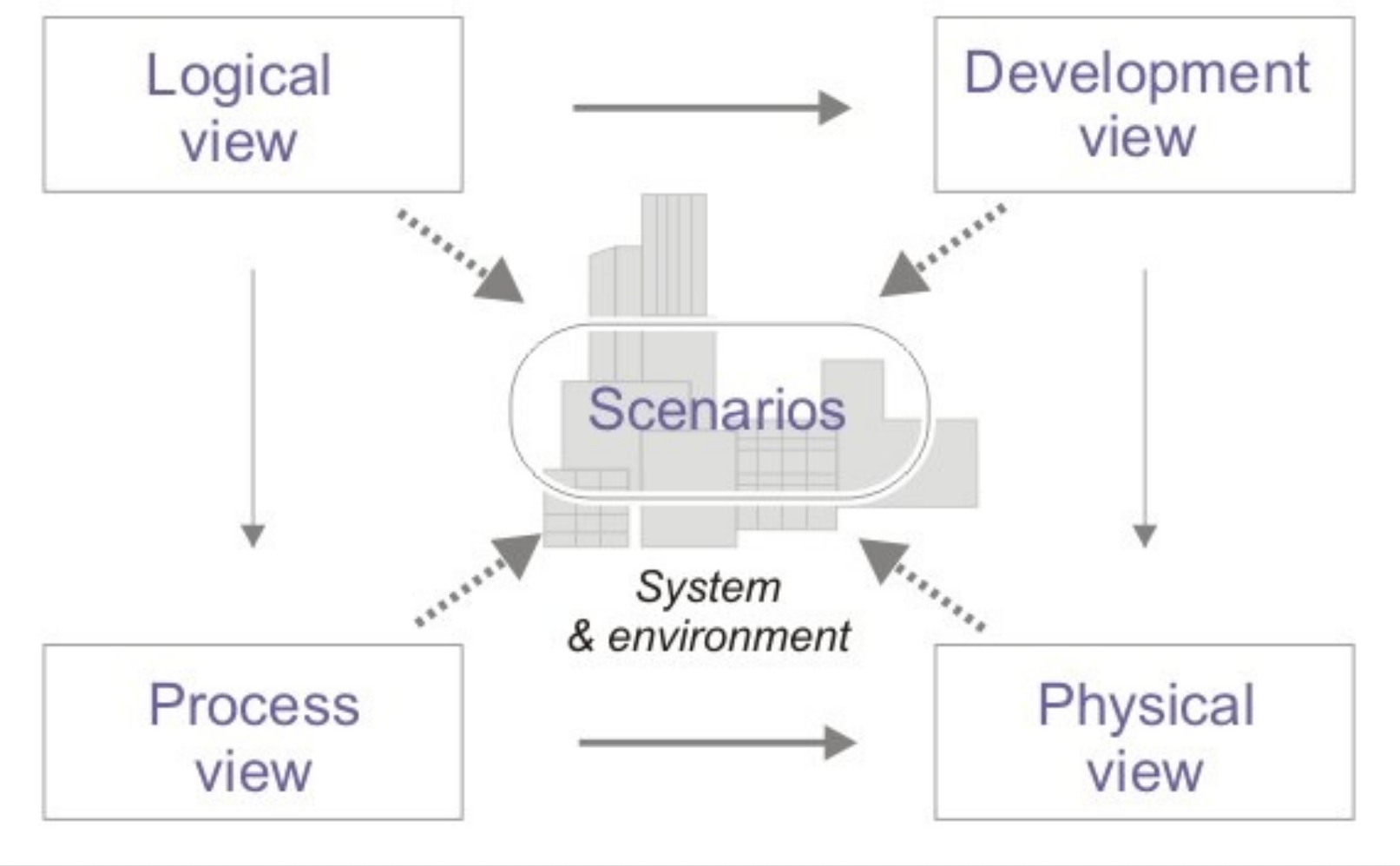
What is the Rationale?

The rationale is the “why?”

For multi-version software, its design rationale must be documented:

- Decomposition into components
- Connections between components
- Constraints upon components and connections

The 4+1 View Model of Software Architecture



Source: http://www.wikipedia.org

Logical (Conceptual) View

Functional requirements

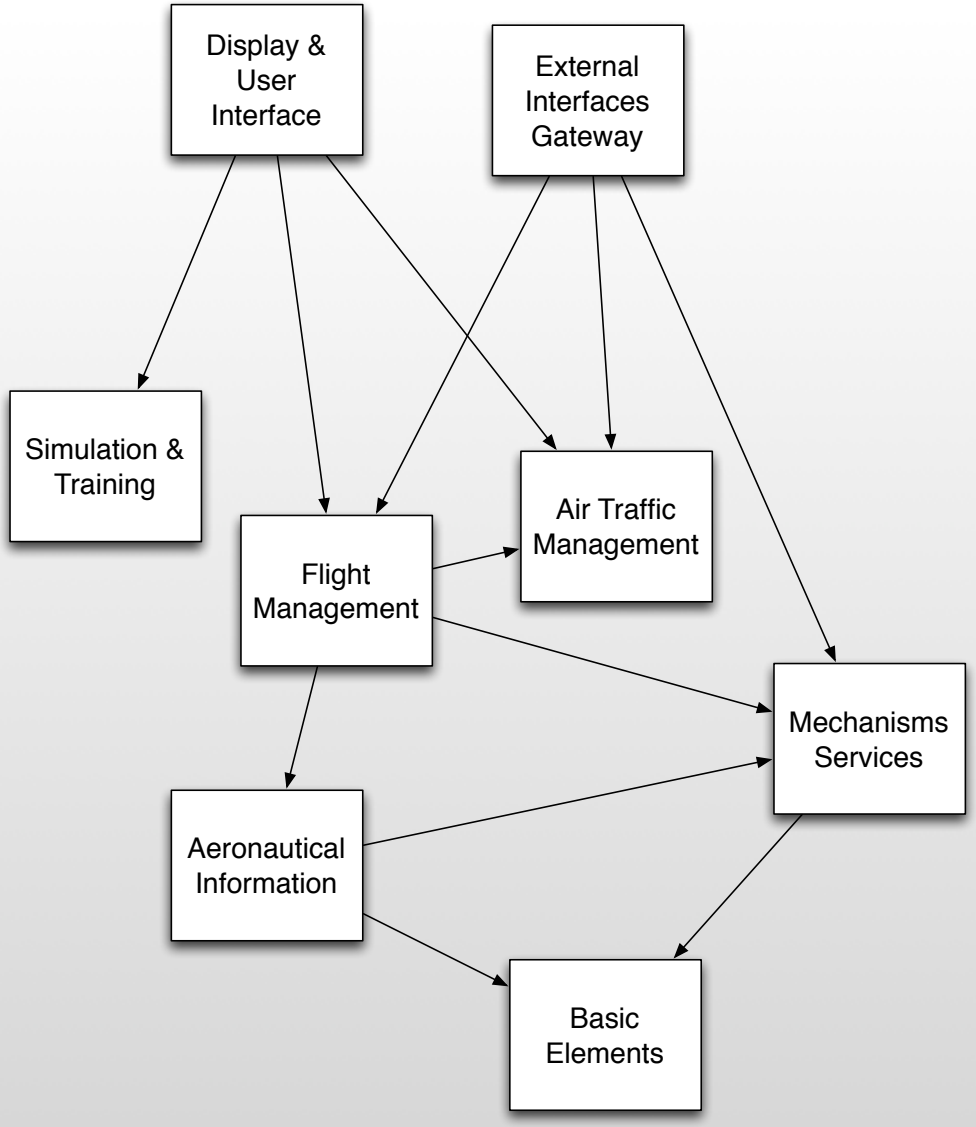
Orientation on problem domain

Communication with experts

Independent of implementation decisions

Logical View: Example

Air traffic management system



Development (Module) View

Organization of modules

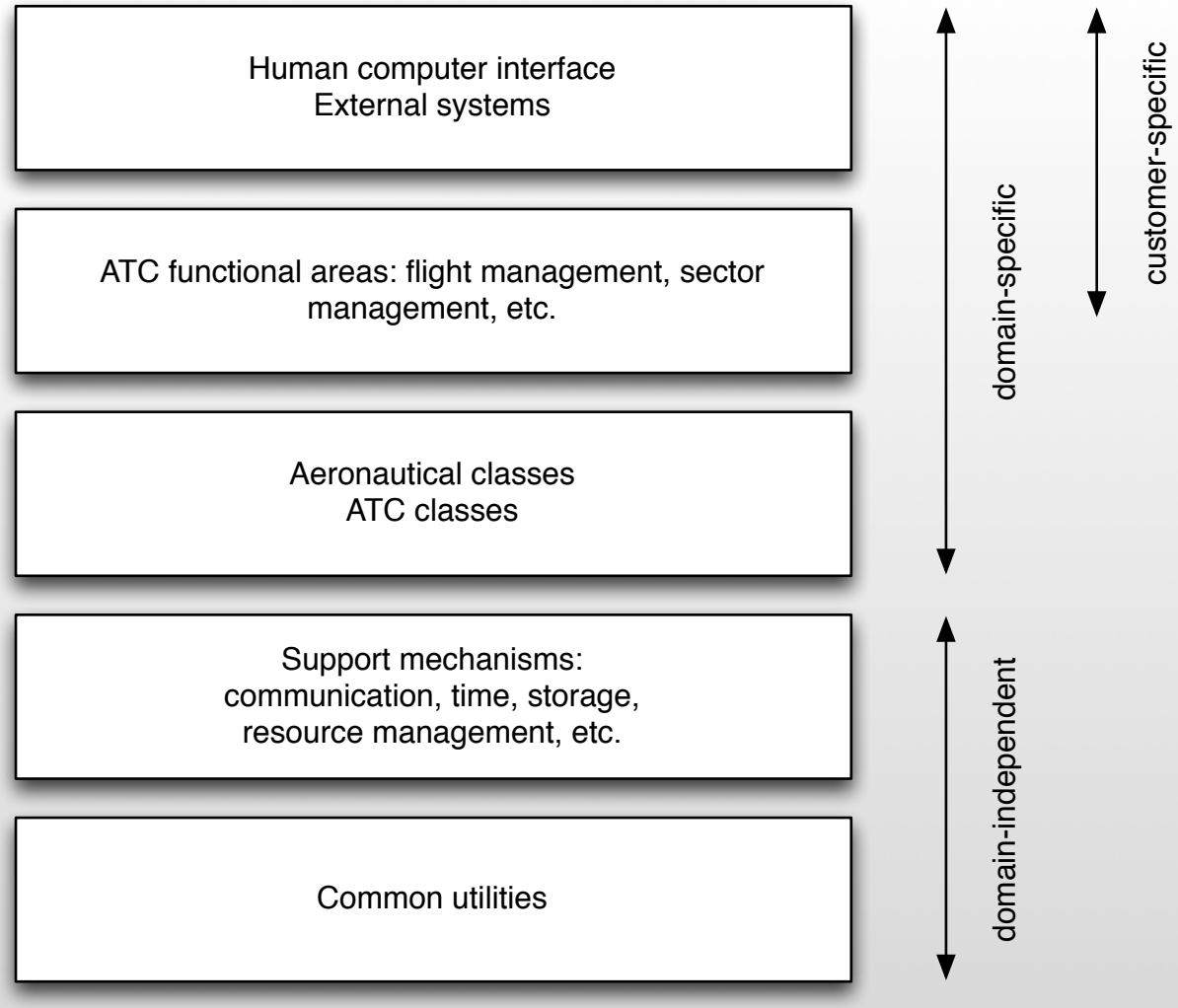
- Subsystems
- Coherent parts in the development
- Allocation of effort (development, maintenance)

Organization in hierarchical layers

- OSI communication protocols

Development View: Example

Air traffic management system



Physical View

Mapping of software onto existing/ available hardware

Non-functional requirements

- Performance
- System availability
- Fault-tolerance
- Scalability

Physical View in Cloud Computing

Note that the term *Physical View* comes from a time before virtualization

In the cloud, your Physical View will map your components to virtual resources

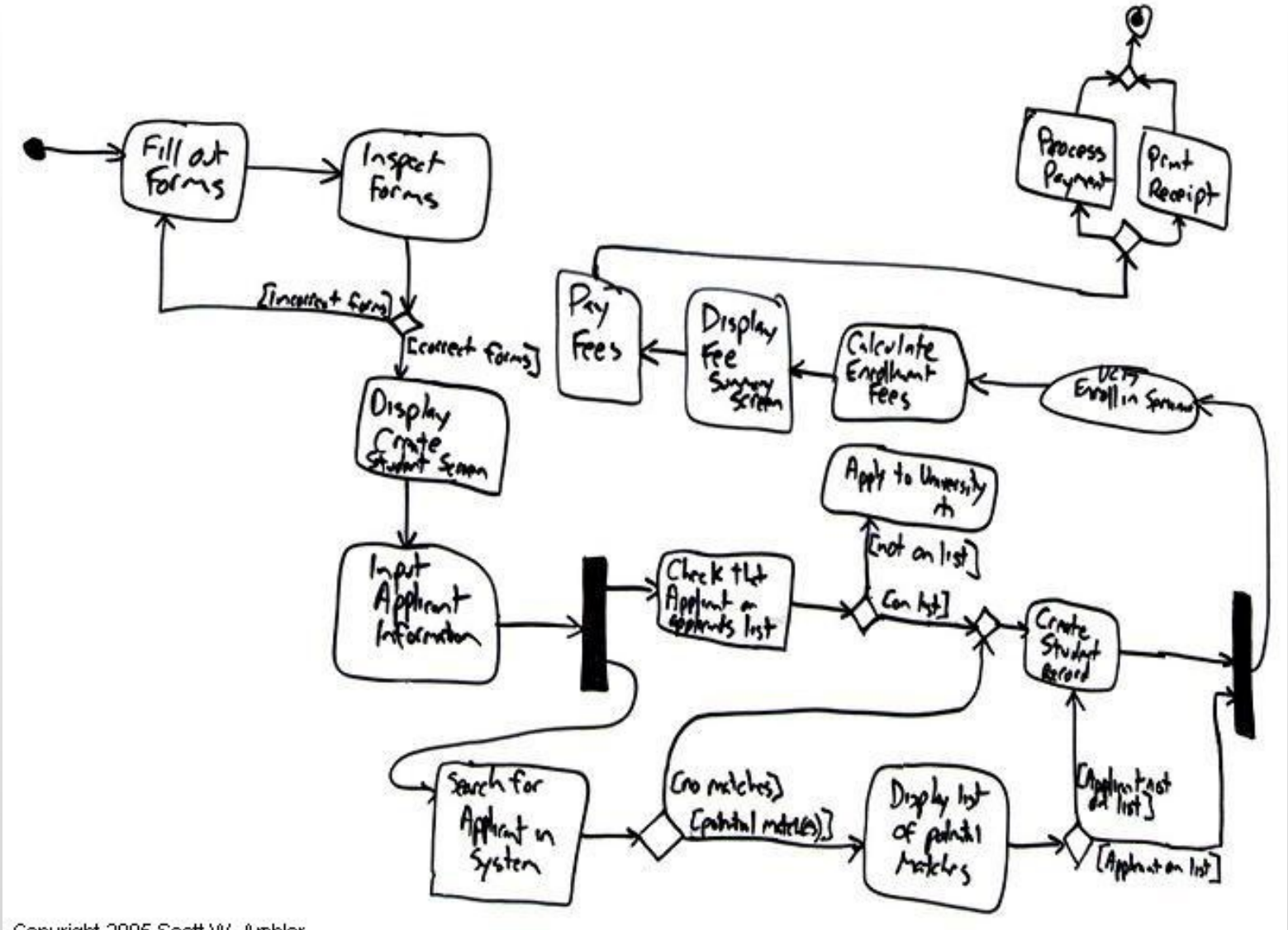
Process View

Dynamic aspects of the run-time processes

- Process creation
- Synchronization
- Concurrency

“What happens at system runtime”

Process View: Example



Scenarios

Instances of uses cases show that the elements of the four views work together seamlessly

Example:

User stories that run through all 4 views …

… and show the importance and rationale of all elements of all 4 views

Architectural Styles and Patterns I

An Architectural Pattern is a named collection of architectural design decisions that are applicable to a recurring design problem parameterized to account for different software development contexts in which that problem appears.

An Architectural Style is a named collection of architectural design decisions that (1) are applicable in a given development context, (2) constrain architectural design decisions that are specific to a particular system within that context, and (3) elicit beneficial qualities in each resulting system.

Architectural Styles and Patterns II

Both provide a common vocabulary to talk about different aspects of software architecture

A pattern is a way of solving a common architectural problem

A style is just a name given to a common architectural design

Architectural Styles

An architectural style defines a family of software systems and their structural organization

It defines components, connectors, and constraints for their usage in concrete software systems

(Some) Common Architectural Styles

Program structure

- Layers
- Client / Server
- Peer-to-Peer
- MVC

Data Flow

- Pipes and Filters
- Event-Driven Architecture
- Blackboard (Factory / Worker)

(Some) Common Architectural Styles

(Many) books are written about architectural styles and patterns.

Some of the most well-known ones:

Buschmann et al.:
Pattern-Oriented Software Architecture, Vol. 1, 2, 4

Fowler:
Patterns of Enterprise Application Architecture

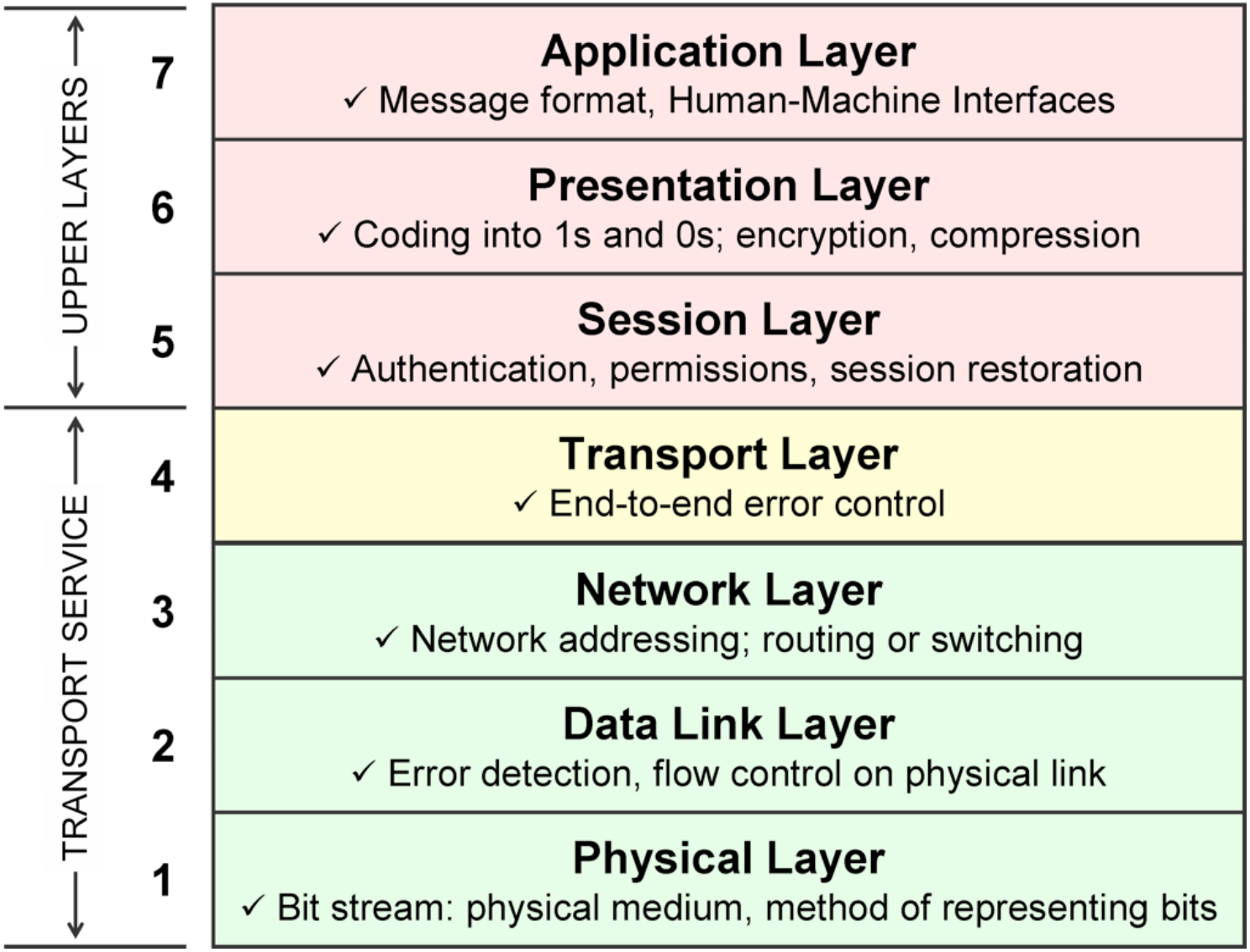
Layered Architecture

Hierarchically organized system

Layers are components

Layer interfaces and protocols are connectors

Layers: Example



Layers: Advantages

Divide and conquer

Locality of changes

Reusability

Layers: Disadvantages

Layers don’t encapsulate everything well

Layers *may* harm performance

Additional layers may increase development effort

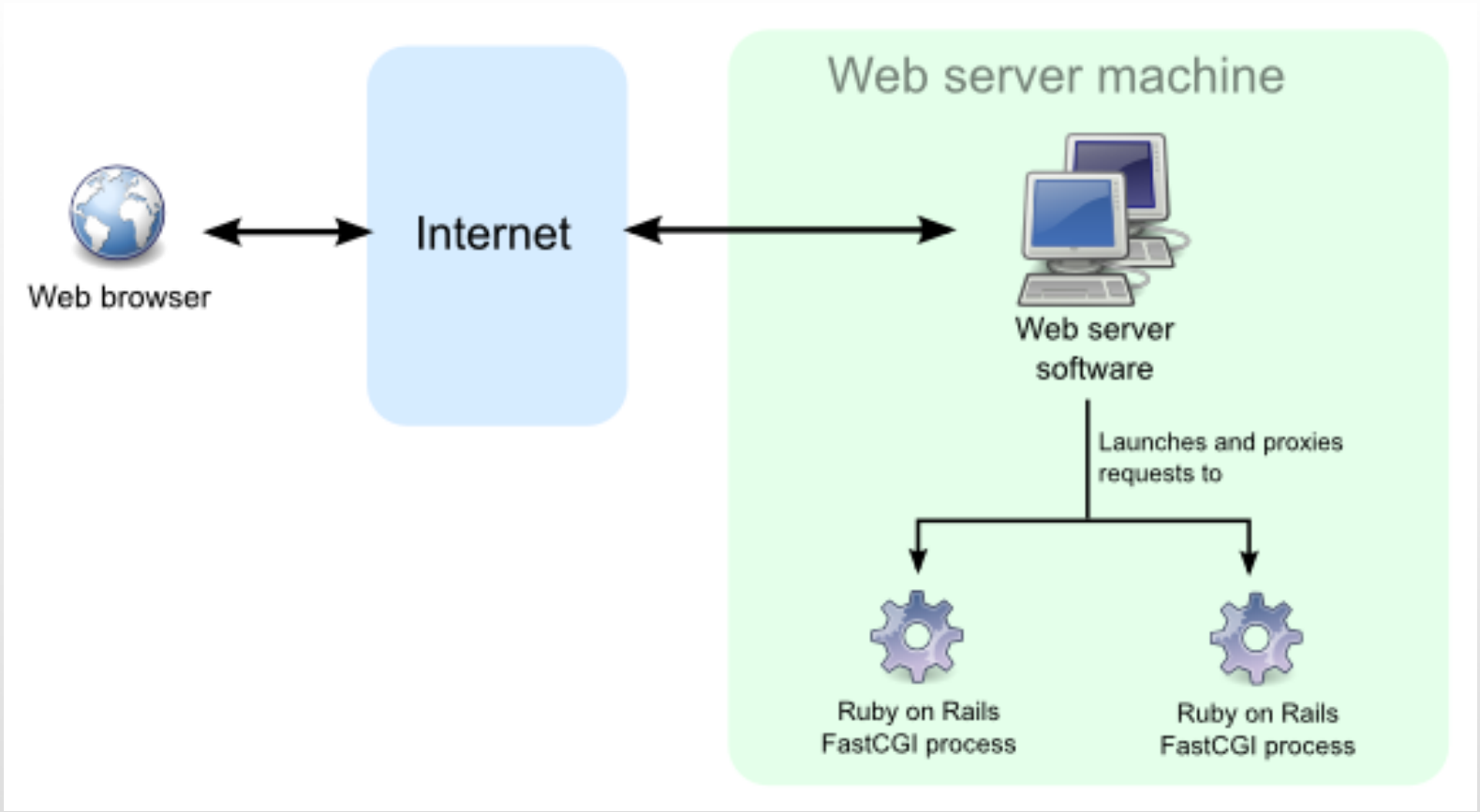
Client / Server

Servers are actively all the time and passively wait for requests

Clients are active sporadically and trigger servers

Servers are (usually) few and fat, while clients are numerous and thin

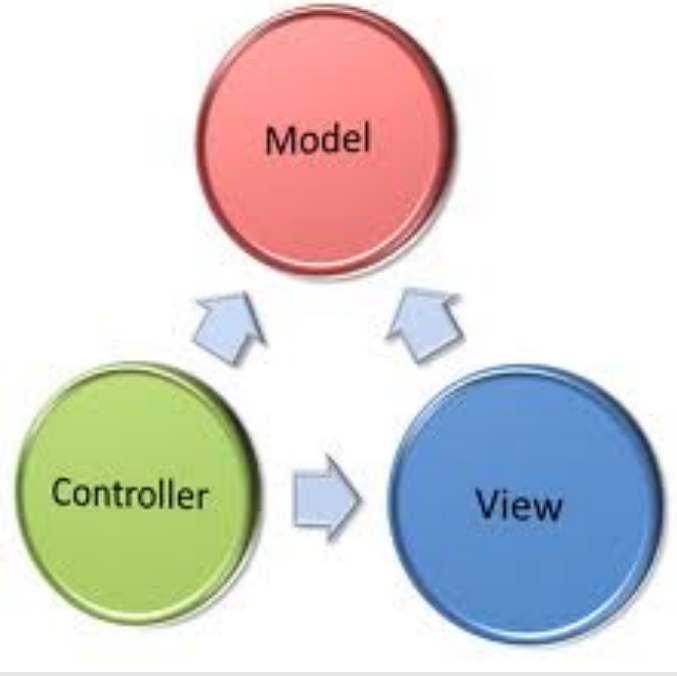
Client / Server Example



Source: http://izumi.plan99.net/blog/index.php/2007/04/05/saving-memory-in-ruby-on-rails/

MVC

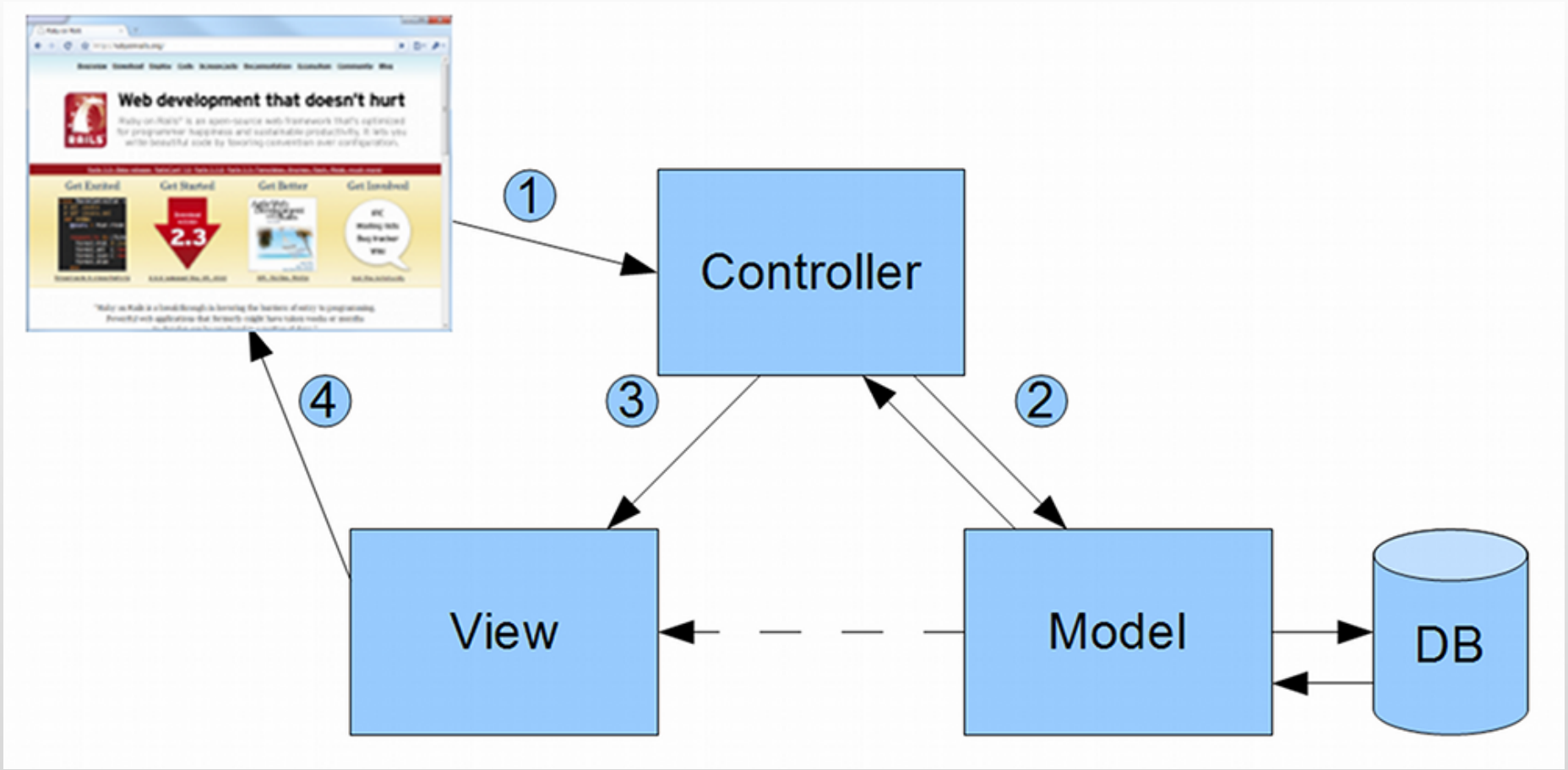
Model-View-Controller



Models hold data
Views present data
Controllers implement business logics

MVC - Example

Ruby on Rails (Ruby-based Web framework)



Source: https://leanpub.com/rails3dot0-astudentmanual/read

MVC: Advantages

Separation of concerns

Supports TDD (Test-Driven Development) well

Maps well to REST (Representational State Transfer) principles

MVC: Disadvantages

Usually, every change (e.g., adding new feature) touches models, views, and controllers

It is not usually possible to assign related model, view and controller to different teams

Client / Server: Advantages

Simple, time-tested style

Particularly suited for distributed applications

Updates to the server are easy to roll out (allows e.g., for partial deployment)

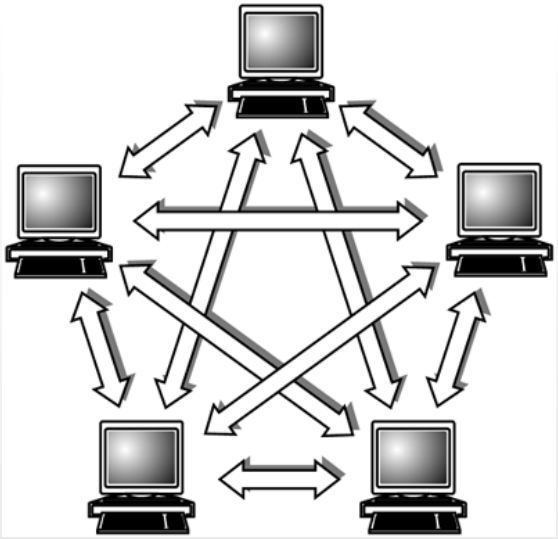
Clients can often be thin and run on cheap hardware

Client / Server: Disadvantages

Servers overload easily

Servers are a single point of failure

Peer-to-Peer



Components are both server and client at the same time

Everybody is (in principle) communicating with everybody

Two styles:

“pure” P2P

“hybrid” P2P with central servers or superpeers

Peer-to-Peer Example

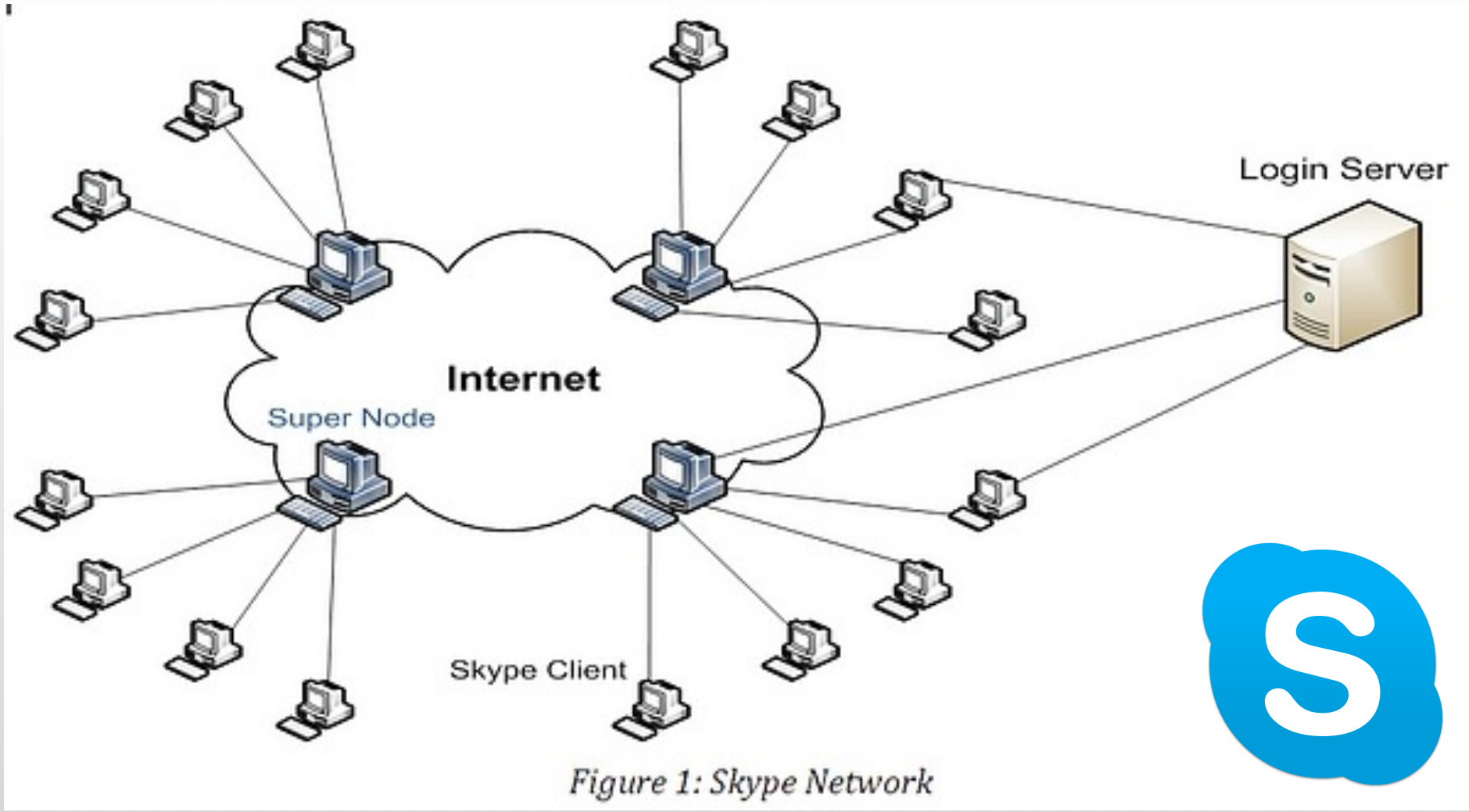


Figure 1: Skype Network

Source: http://kb.fortinet.com/kb/viewContent.do?externalId=FD30776

Peer-to-Peer Advantages

Each new participant also adds new resources —> no natural bounds to system scale (scalability!)

Usually few or no single point of failure

Usually no single point of control

Peer-to-Peer: Disadvantages

In pure P2P, message flooding is a problem

"Soft" state between peers

Protocols tend to get complicated

Software updates often not easy

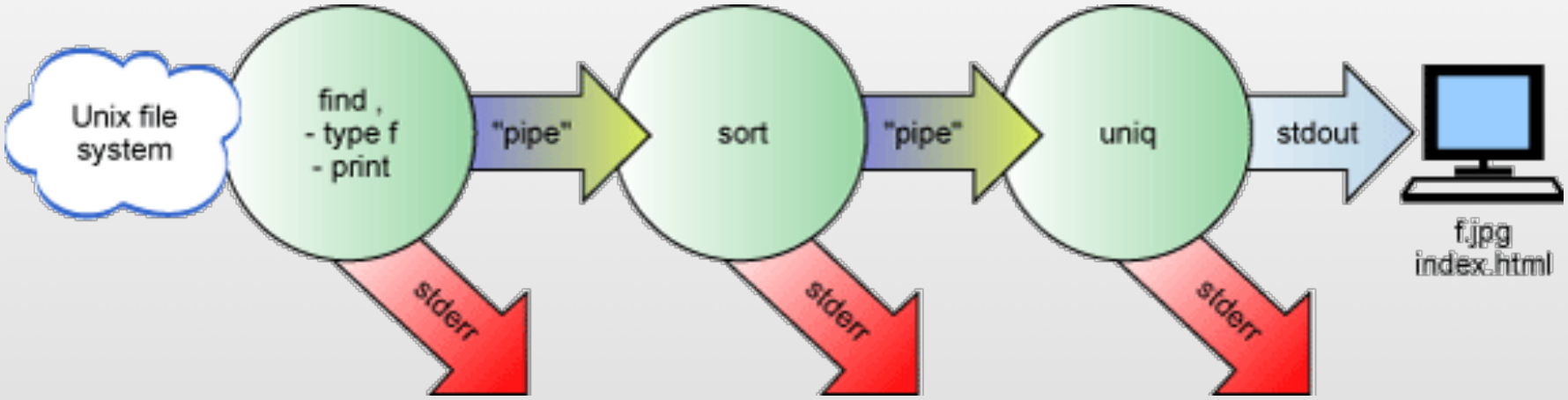
Pipes and Filters

Filters are the components that read an input data stream and transform it into an output data stream

Pipes are the connectors that provide the output of a filter as input to another filter

Pipes and Filters: Example

find . -type f -print l sort l uniq



Pipes and Filters: Advantages

Simple; no complex component interactions

Filters as black-box; substitutable

High maintainability and reusability of individual components

Pipes and Filters: Disadvantages

Whether separation into processing steps is feasible strongly depends on application domain and problem

Filters require a common data format

Redundancy in parsing/unparsing

Process overhead

Event-driven Architecture (EDA)

Components are event emitters (publishers, agents) or consumers (subscribers, sinks)

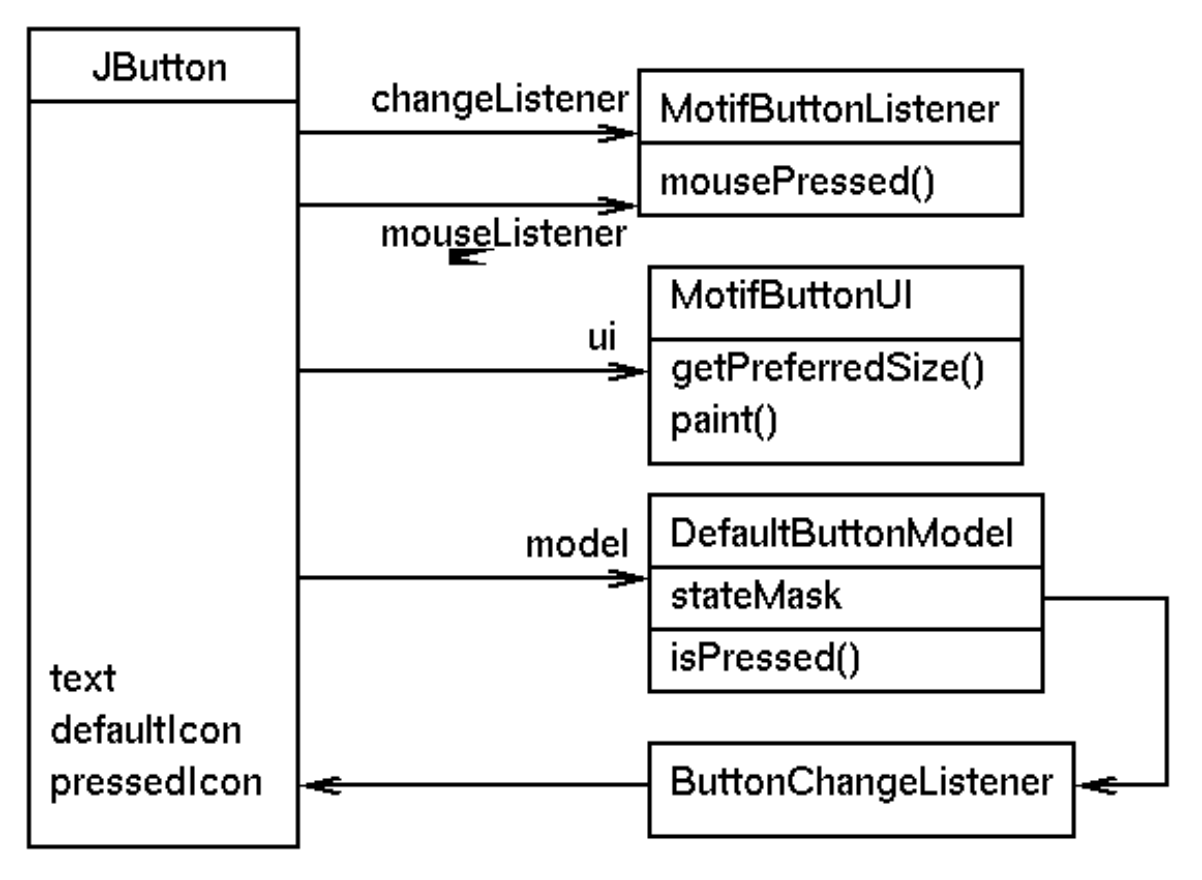
An event dispatcher distributes events to subscribers

—> fundamentally **asynchronous**

EDA Example

Java Swing

(or more generally - most GUI frameworks)



EDA: Advantages

Components are loosely coupled

High (runtime) extensibility and reusability; components can be easily exchanged

Especially apt to unpredictable and asynchronous environments

Often “feels” fast and responsive

EDA: Disadvantages

No guarantees regarding execution or order of event processing

Data exchange other than with events problematic

Component behavior is tightly coupled with execution environment

Difficult to “grok”

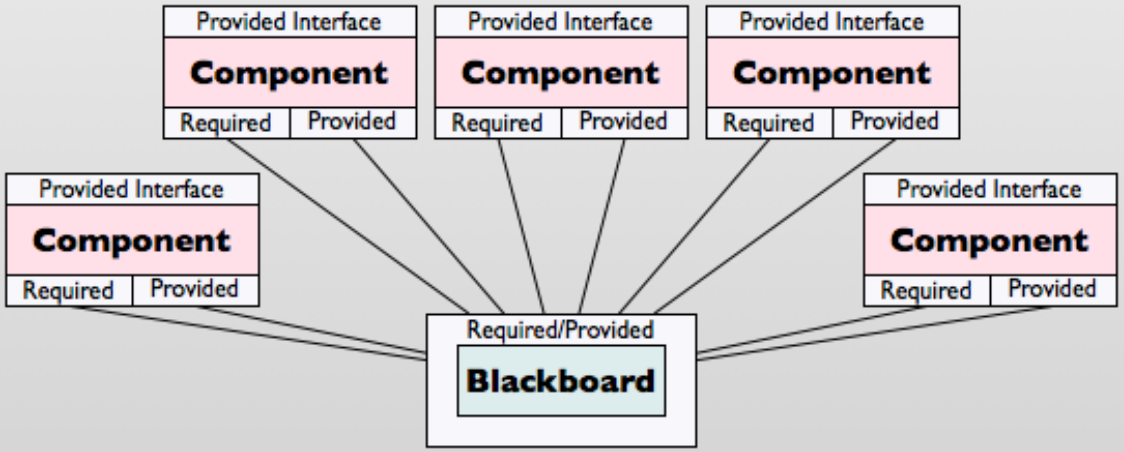
Blackboard

Two kinds of components:

- Central data management
- Independent components for computation (knowledge sources)

Activation of computation:

- Database trigger
- Actual state



Blackboard: Advantages

High changeability and maintainability

Reusable knowledge sources

Factory/Worker: support for fault-tolerance, robustness and redundancy because of loose coupling of workers

Blackboard: Disadvantages

Hard to test

Difficult to establish good control strategy

Low efficiency

Failure to Maintain Architectural Discipline?

Failure to Maintain Architectural Discipline?

Common reasons:

- “pragmatism” (we used whatever made sense at the time)
- time pressure
- “agile” / “lean” (minimal product)
- architecture follows organizational structure

Conclusions

For non-trivial software systems, a proper architecture needs to be designed (either upfront or in parallel to the code)

Architectures consist of components, connectors and constraints (+ rationale)

The 4+1 Model provides means to derive and document such an architecture

Architectural styles provide software engineers a common language for describing and reasoning about software architecture

Next Weeks

Next week:

- We will look at components and (Micro-) services as basic architectures for distributed systems
- The week after we will discuss ATAM, a concrete evaluation model for software architectures