Software Architecture

Software Engineering
Alessio Gambi - Saarland University

These slides are based the slides from Cesare Pautasso and Christoph Dorn, and updated from various sources.

References and Readings

- **Textbooks**
  - R. N. Taylor, N. Medvidovic, E. M. Dashofy, Software Architecture: Foundations, Theory, and Practice, Wiley, January 2009.
  - G. Fairbanks, Just Enough Software Architecture: A Risk-Driven Approach, Marshall & Brainerd, August 2010.
  - Amy Brown and Greg Wilson (eds.) The Architecture of Open Source Applications, 2012.

- **References**
  - Mary Shaw and David Garlan, Software Architecture: Pespectives on an Emerging Discipline, Prentice-Hall, 1996
  - Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal Pattern Oriented Software Architecture: A System of Patterns, Wiley, 1996
  - William Brown, Raphael Malveau, Hays McCormick, Thomas Mowbray, Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis, Wiley, 1992
  - Clemens Szyperski, Component Software: Beyond Object-Oriented Programming, 2nd Edition, Addison-Wesley, 2002
  - Len Bass, Paul Clements, Rick Kazman, Ken Bass, Software Architecture in Practice, 2nd Edition, Addison-Wesley, 2003
  - Martin Fowler, Patterns of Enterprise Application Architecture, Addison Wesley, 2002
  - Luke Hohmann, Beyond Software Architecture: Creating and Sustaining Winning Solutions, Addison-Wesley, 2003
  - Ian Gorton, Essential Software Architecture, Springer 2006

Intro and Motivation

Design in the Large

- Objects and methods
- Modules and components
- Large and complex systems
- Systems of systems

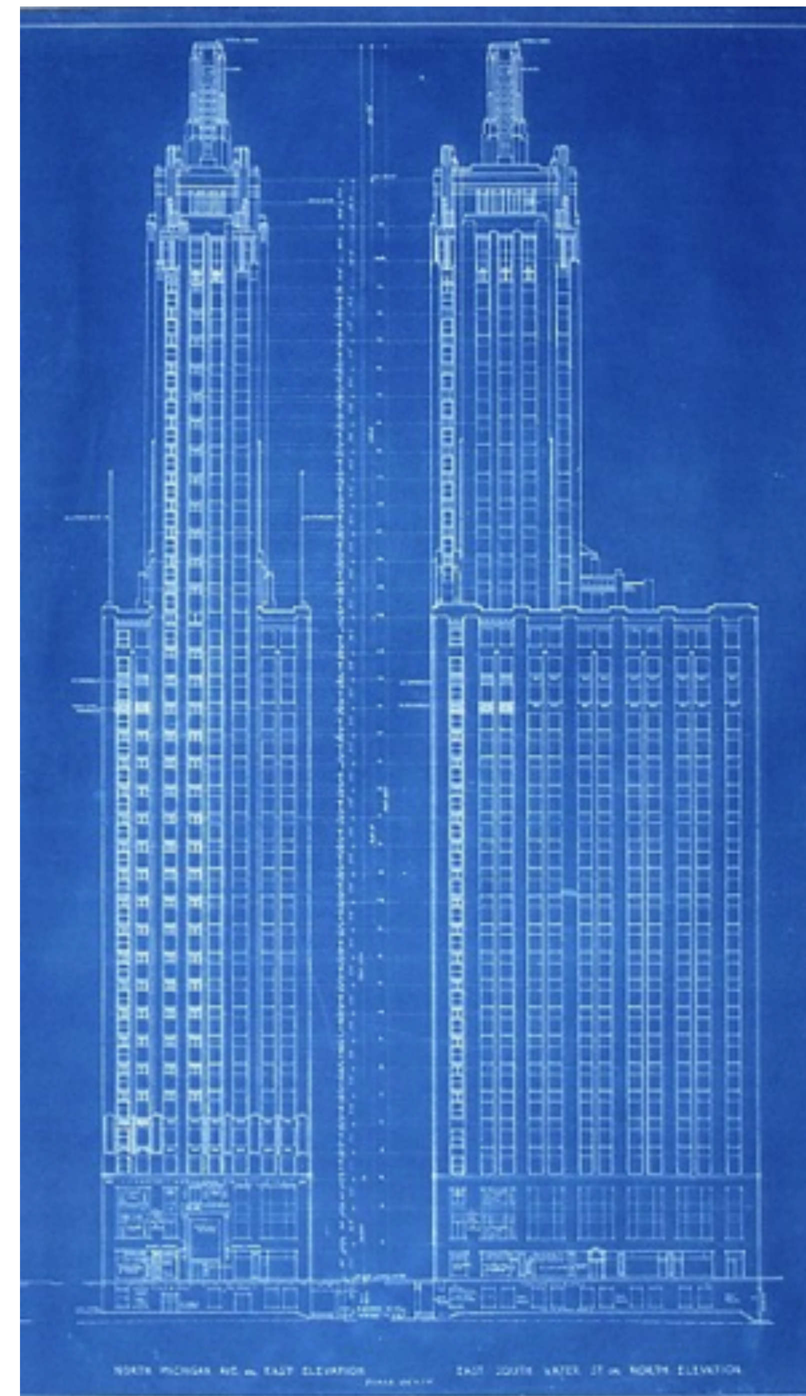
Design in the Large

- Objects and methods
- Modules and components
- Large and complex systems
- Systems of systems

- Size of the team
- Lifetime of the project
- Cost of development

Building software as we build buildings ?

- Software is complex, so are buildings (blueprint)
- Architecture implies a systematic process for design and implementation
- Architects put together pieces and materials, they usually do not invent new materials



It’s just an analogy !

- We know a lot about buildings (2000+ years), much less about software
- Software systems do not obey to physical laws
- Software is a machine; a building is not
- Software deployment has no counterpart in building architecture

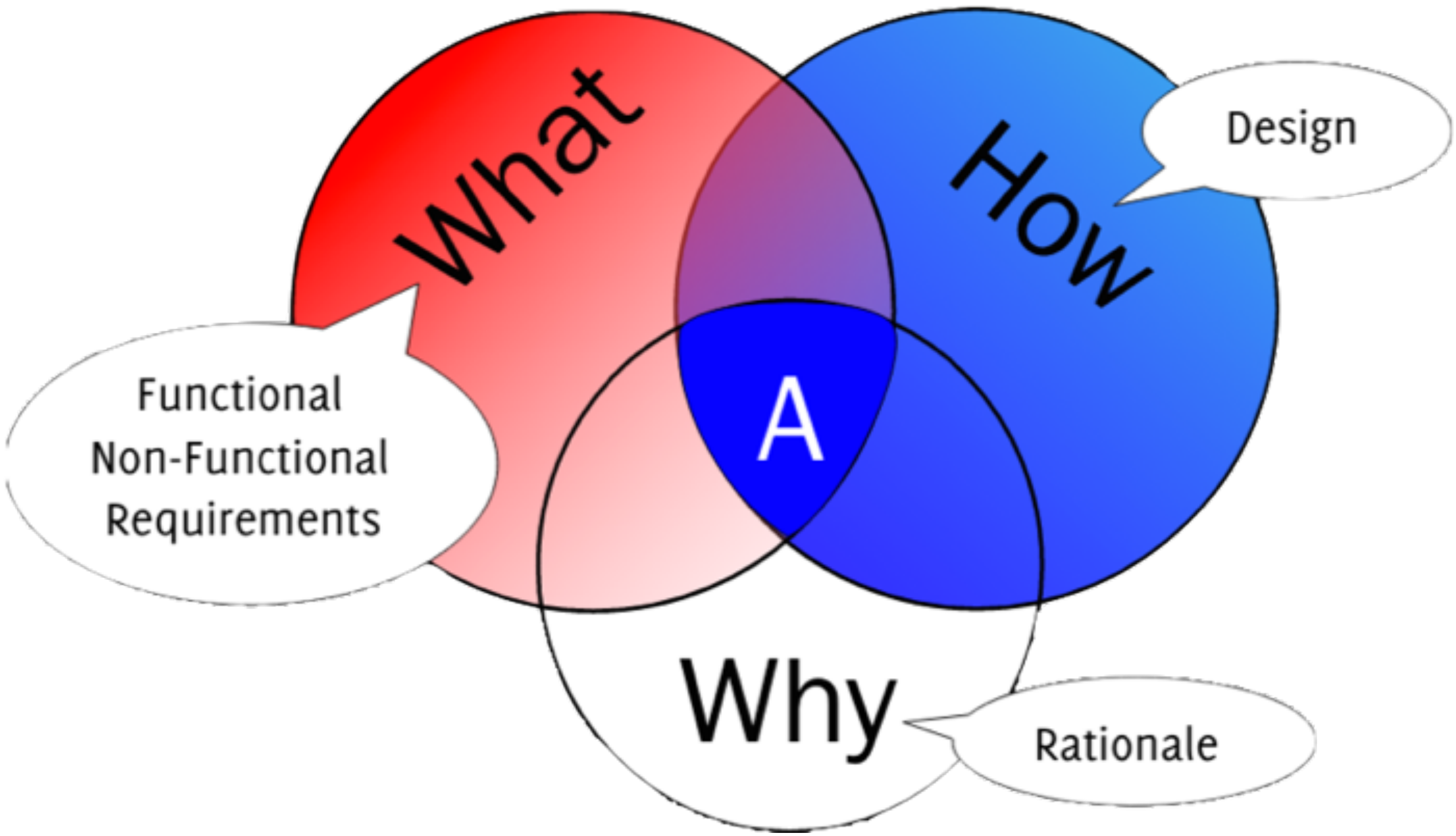
Basic Concepts and Definitions

Software Architecture

A software system's architecture is the set of principal design decisions made about the system.

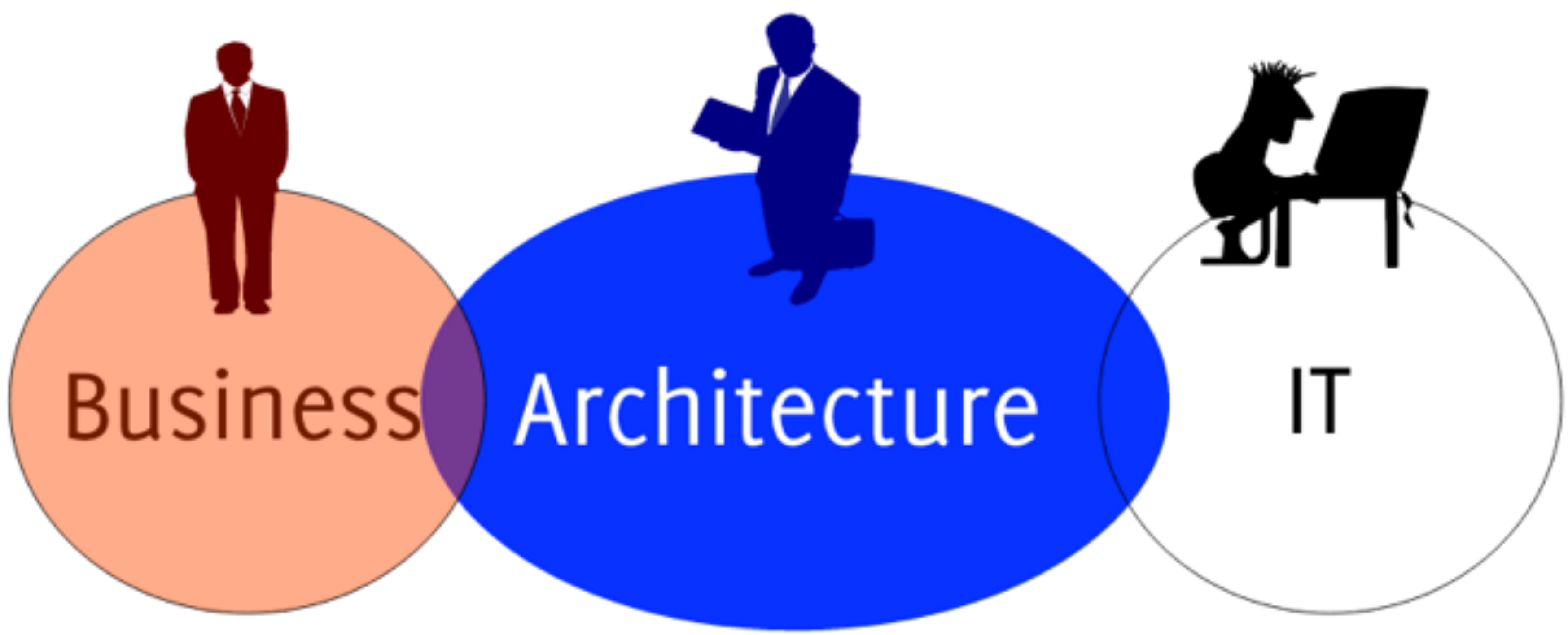
N. Taylor et al.

Abstraction



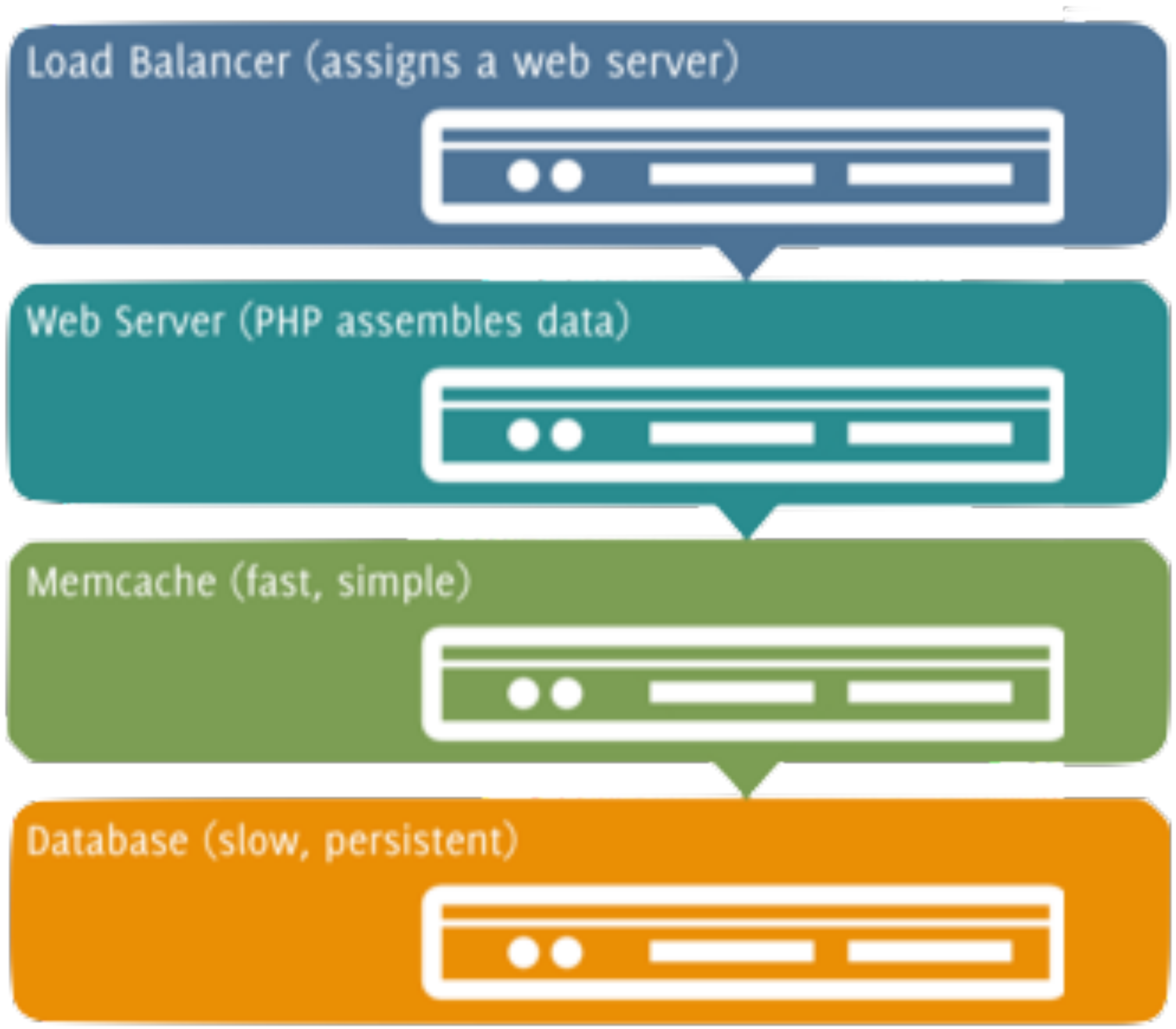
Manage complexity in the design

Communication

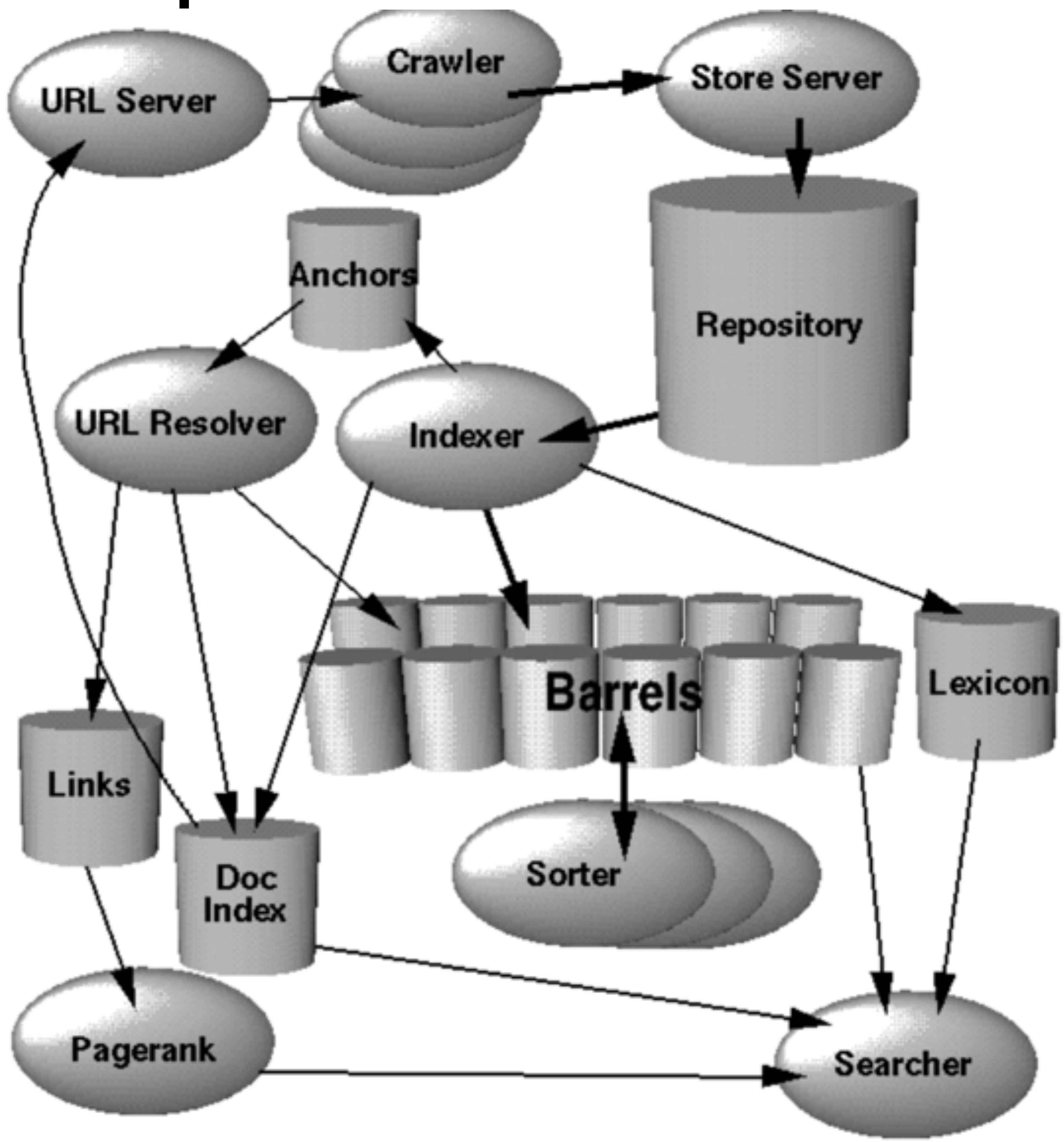


Document, remember and share design decisions among the team

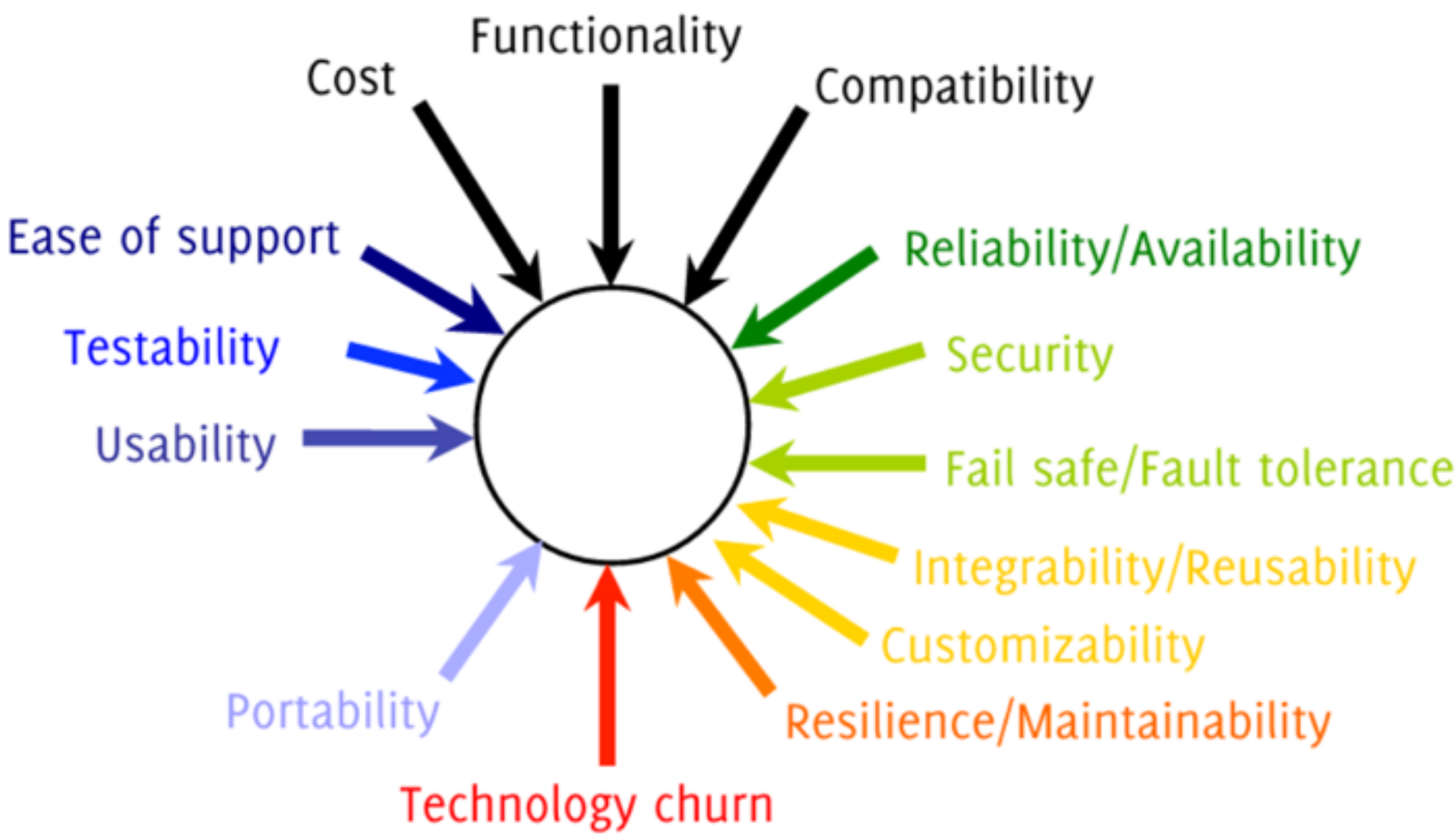
Visualization



Representation

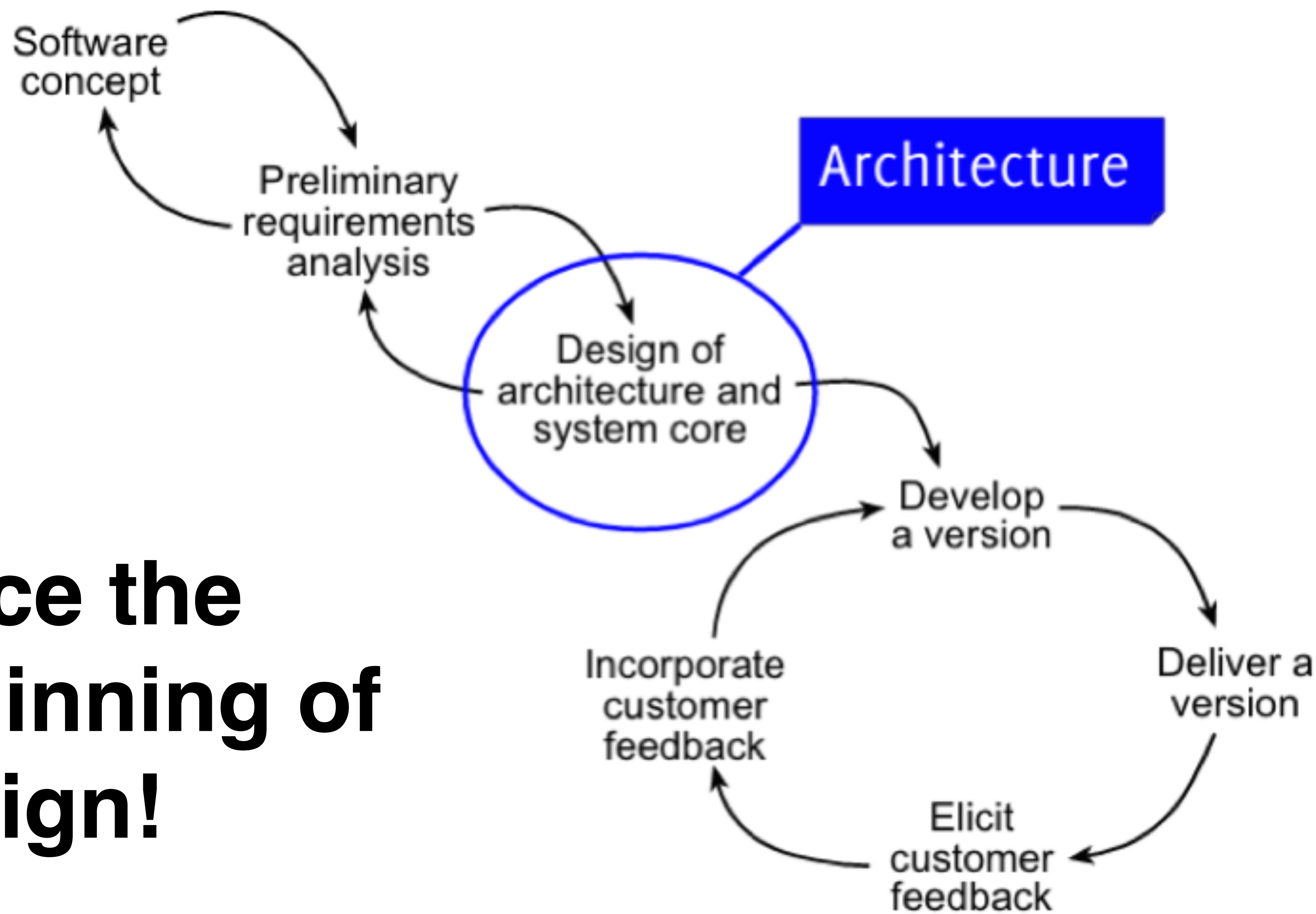


Quality Analysis



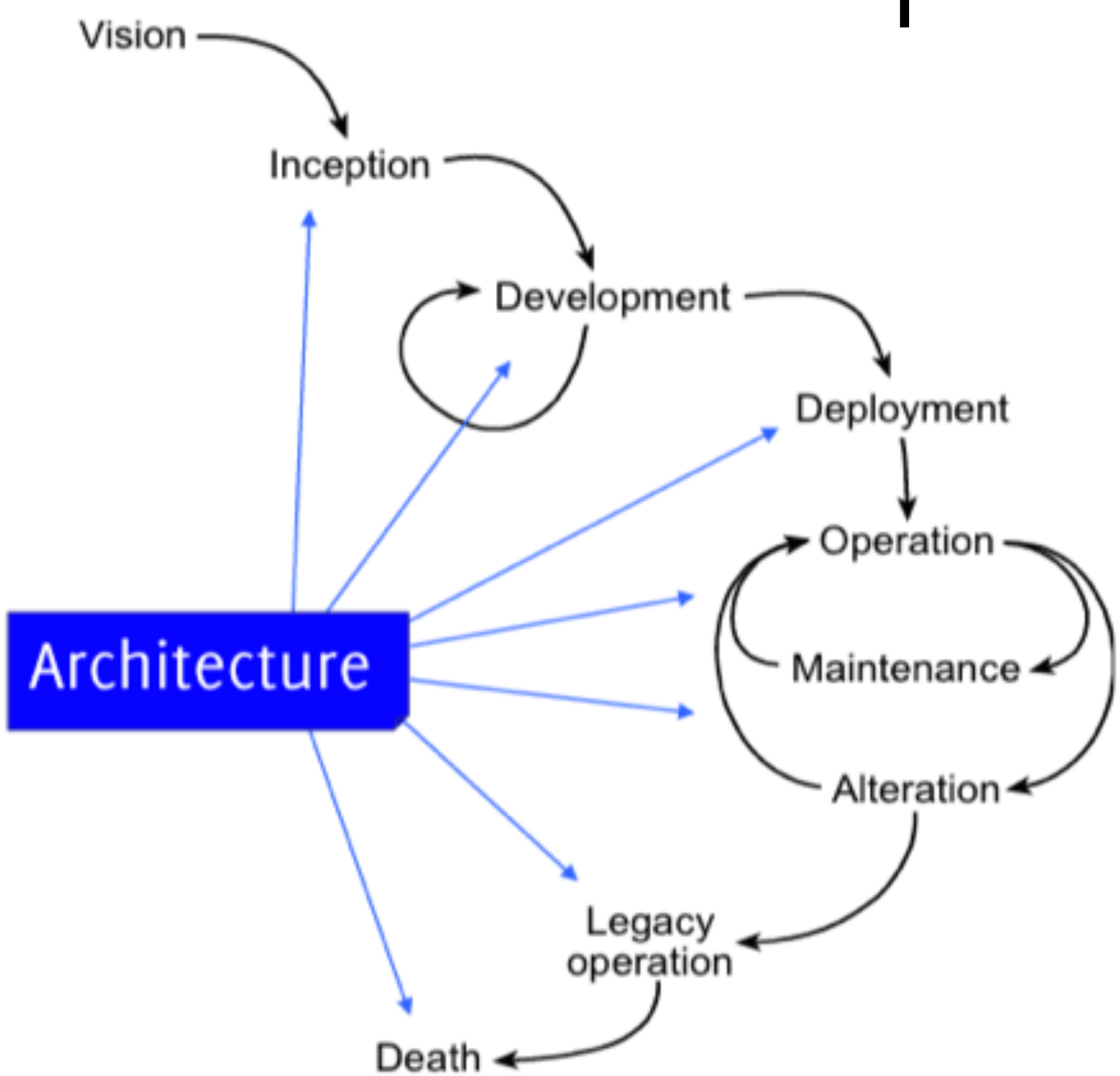
Understand, predict, and control

When Sw. Architecture Start ?



Since the beginning of design!

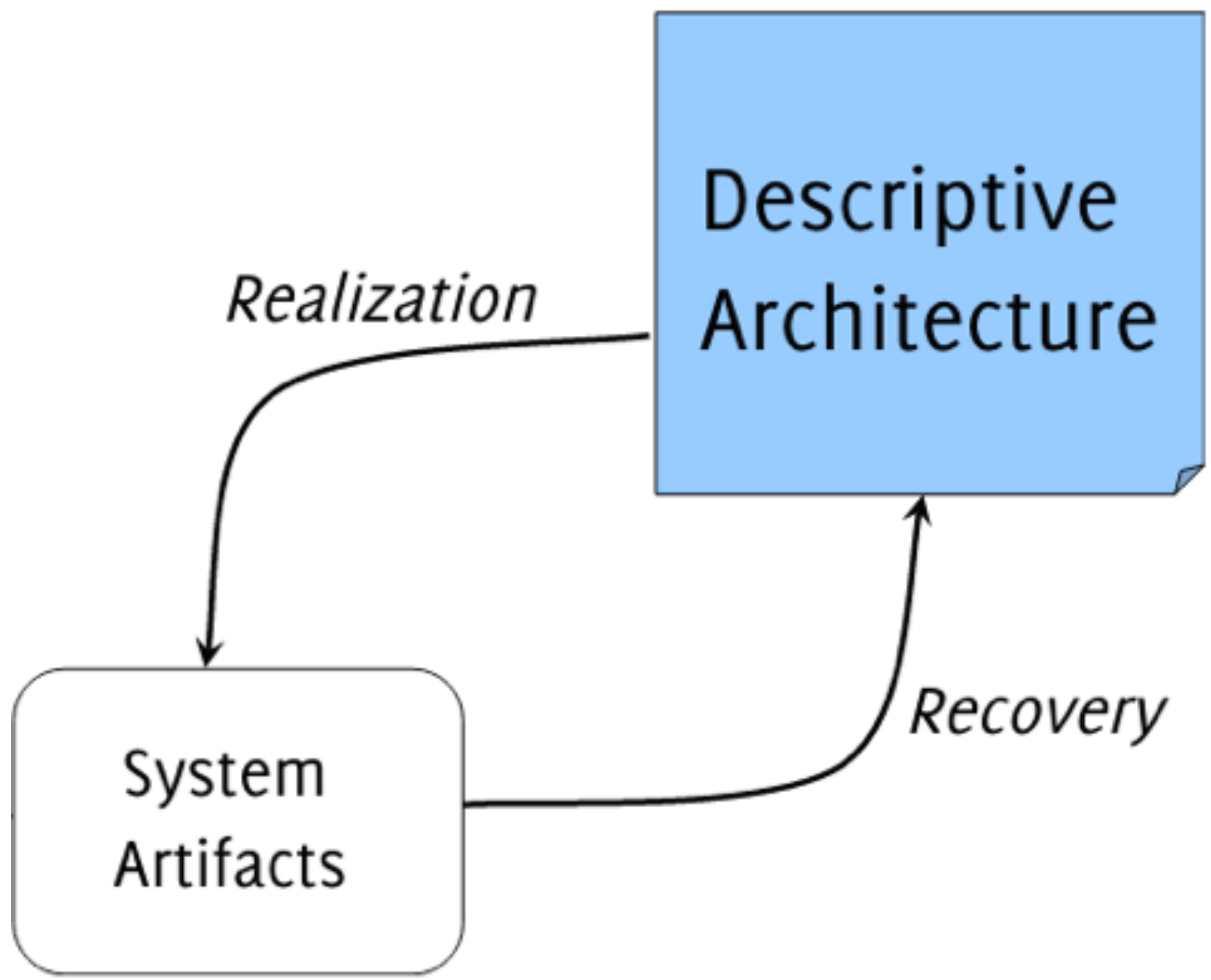
When Sw. Architecture Stop ?



Never!

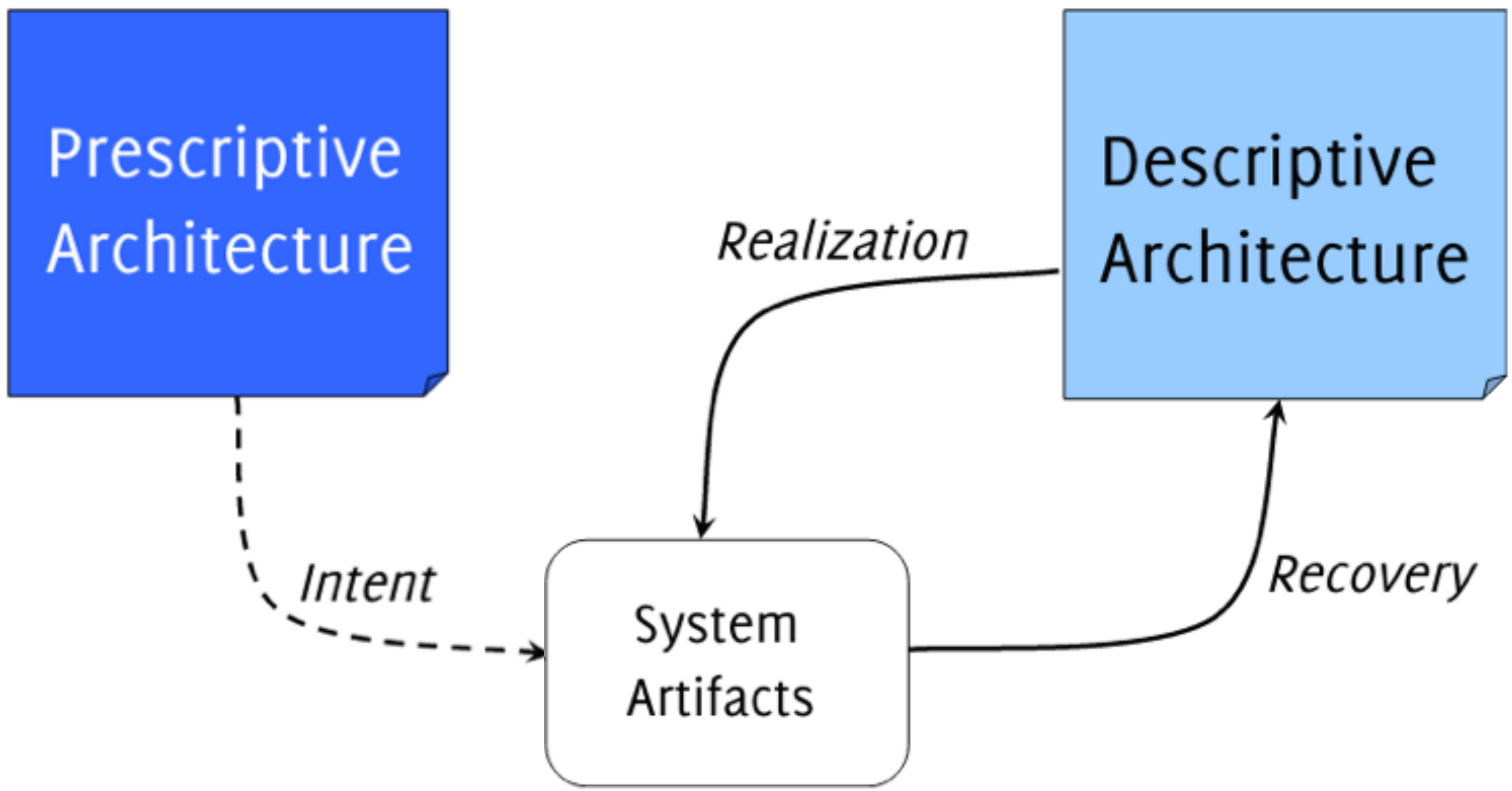
Architecture is NOT a phase of development

Descriptive vs Prescriptive



Every system has a Software Architecture

Descriptive vs Prescriptive



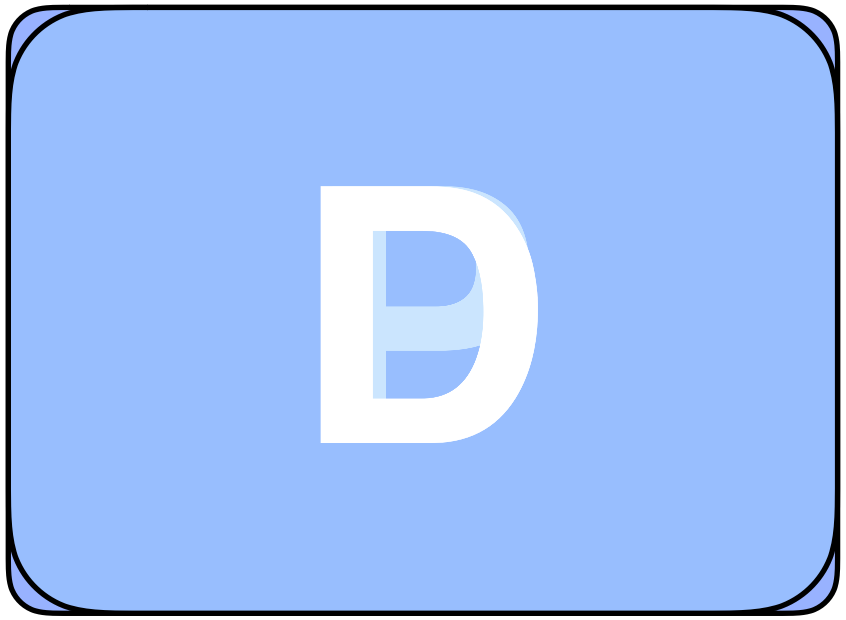
Every system has a Software Architecture

Architectural Evolution

Decisions are made over time
Decisions are changed over time
Decision are made by more than one person

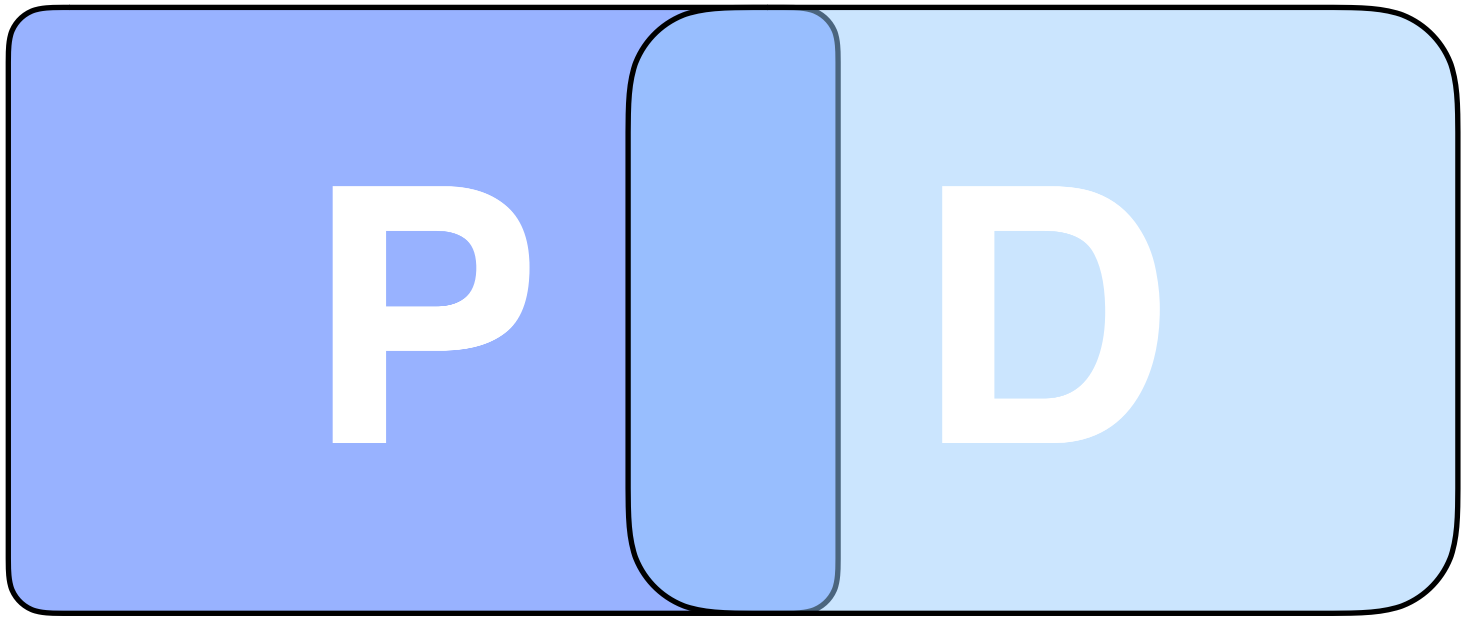
The system architecture changes over time

Architectural Degradation



Ideal P=D

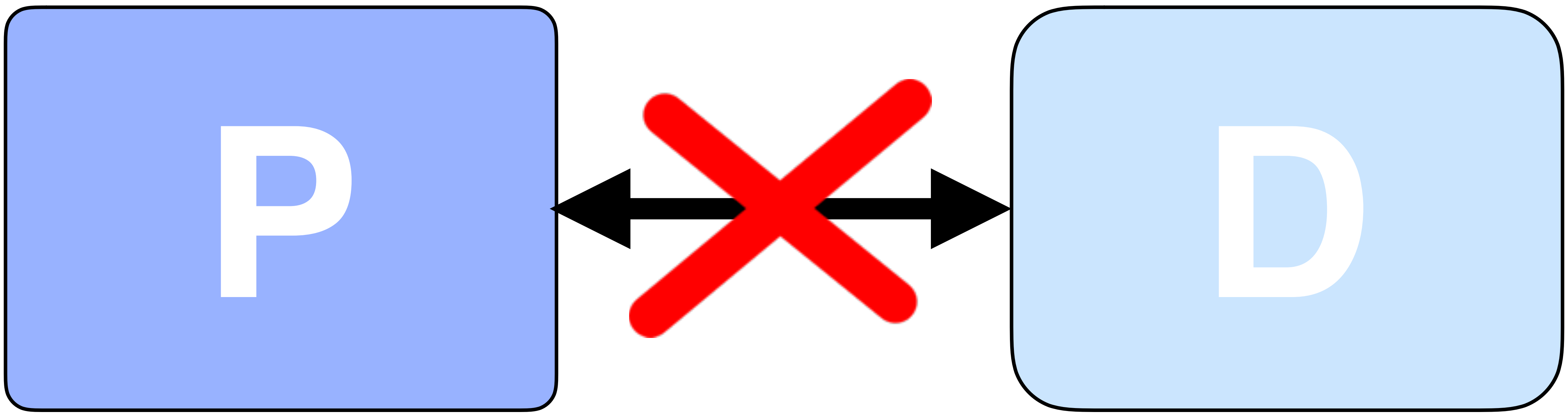
Architectural Degradation



Ideal P=D

Drift P !=D and D does not violate P

Architectural Degradation



Ideal P=D

Drift P !=D and D does not violate P

Erosion P !=D and D violates P

Software Architecture

- Blueprint for construction and evolution

  abstraction • principal design decisions

- Not only about design

  communicate • visualize • represent • quality

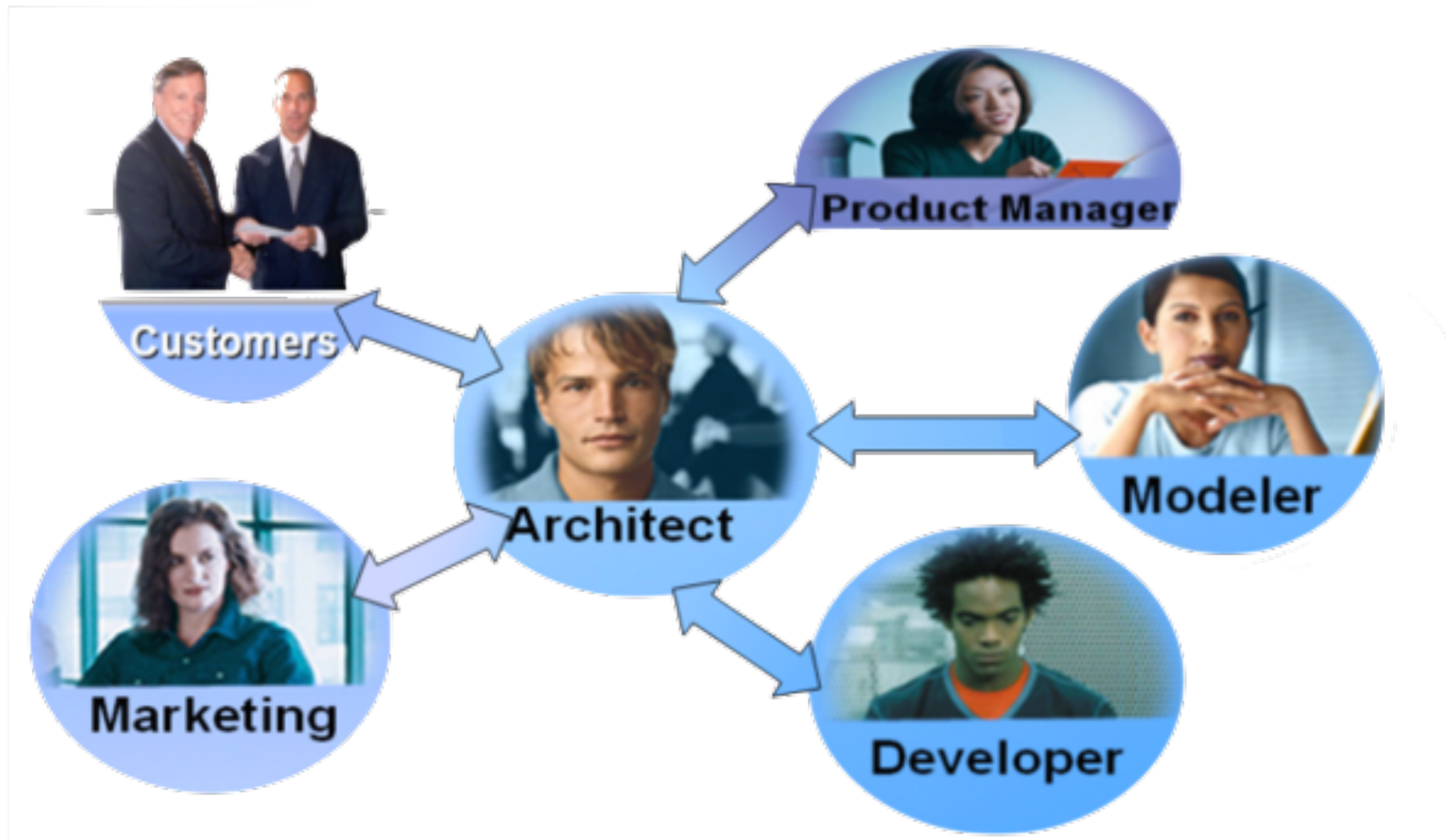
- Every application has one, which evolves

  descriptive • prescriptive • drift • erosion

- Not a phase of development

The Software Architect

Is the one that takes strategic design decision



The Software Architect

Is the one that takes strategic design decision

Communicator

Development Leader

Technology Expert

Risk Manager

Architects as …

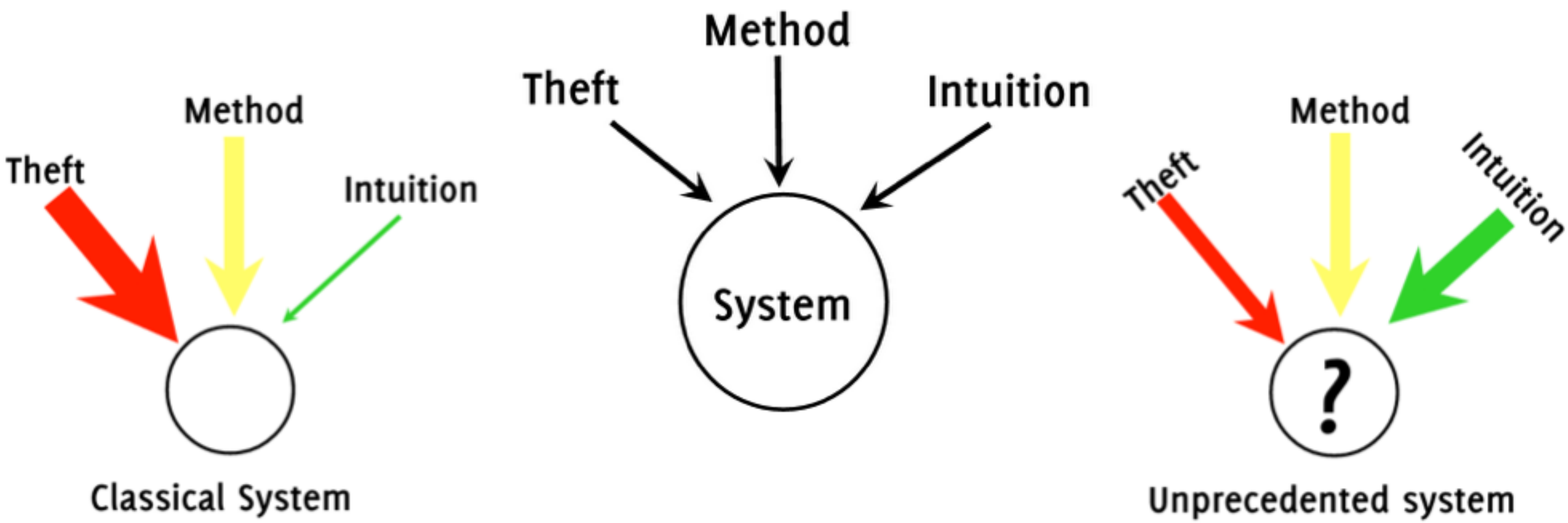
- Software Development Experts
- Consultants
- Domain Experts
- Strategists
- Cost Estimators

Skills and experience:
The best architects are grown, not born

Design

How to Design

Even the best architects copy solutions that have proven themselves in practice, adapt them to the current context, improve upon their weaknesses, and then assemble them in novel ways with incremental improvements.



Architectural Hoisiting

Design the architecture with the intent to guarantee a certain quality of the system.

- Security: place sensitive data behind the firewall
- Scalability: make critical components stateless
- Persistence: use a database
- Extensibility: design/reuse a plug-in framework

George Fairbanks

What makes a “good” Architecture?

- No such things like perfect design and inherently good/bad architecture
- Fit to some purpose, and context-dependent
- Principles, guidelines and the use of collective experience (*method)*

Design principles - Arch. Patterns - Arch. Styles

Design Principles

- Abstraction
- Encapsulation - Separation of Concerns
- Modularization
- KISS (*Keep it simple, stupid*)
- DRY (*Don't repeat yourself)*

Architectural Patterns

An architectural pattern is a set of architectural design decisions that are applicable to a recurring design problem, and parameterized to account for different software development contexts in which that problem appears.

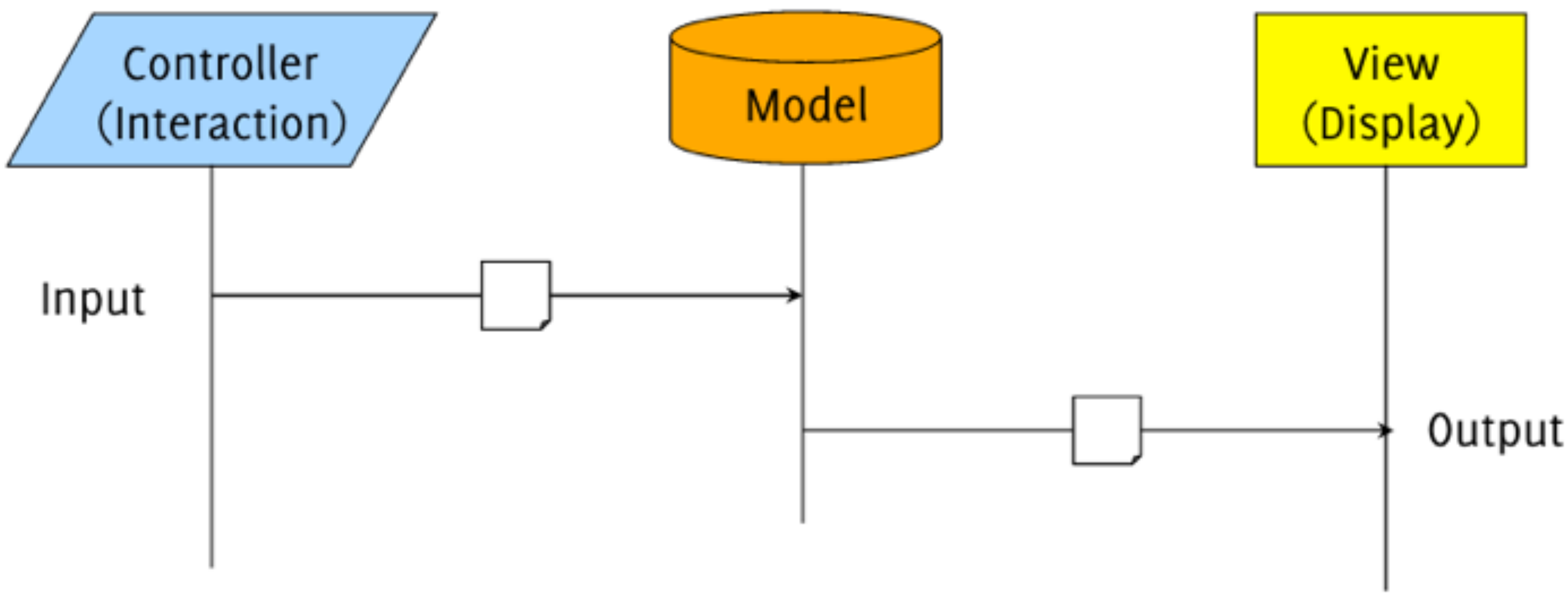
Architectural Patterns

An architectural pattern is a set of architectural design decisions that are applicable to a recurring design problem, and parameterized to account for different software development contexts in which that problem appears.

Layered - Component - Events - Composition

Model-View-Controller

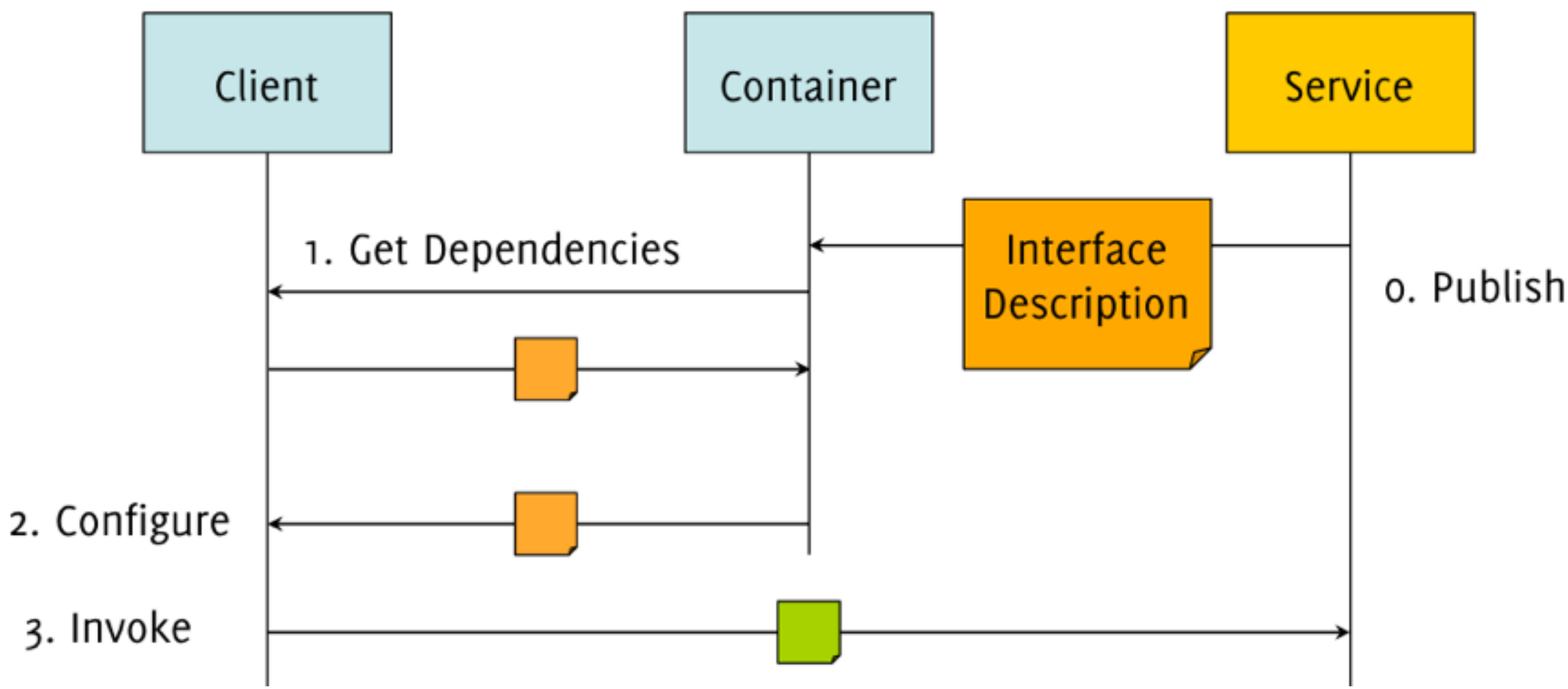
separate content (model) from presentation (output) and interaction (input)



Layered

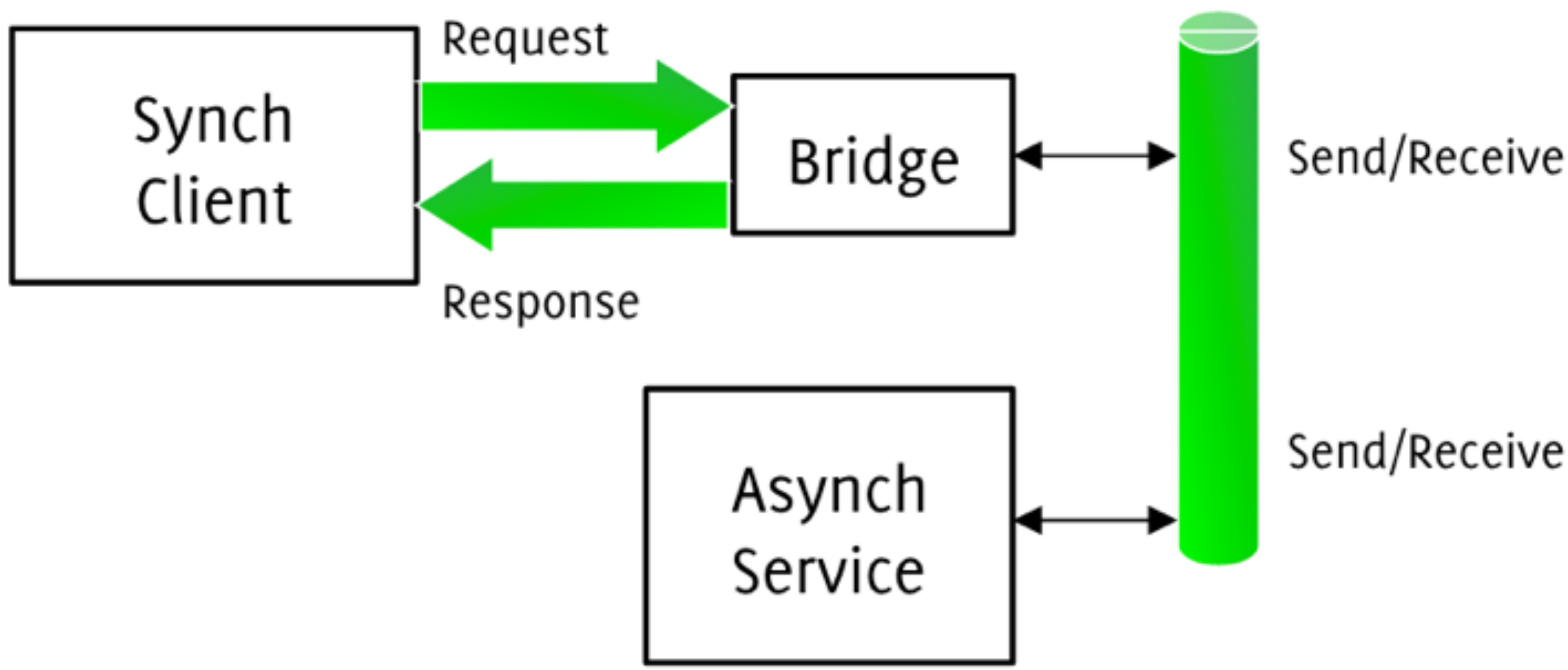
Dependency Injection

use a container which updates components with bindings to their dependencies



Half-Synch/Half-Asynch

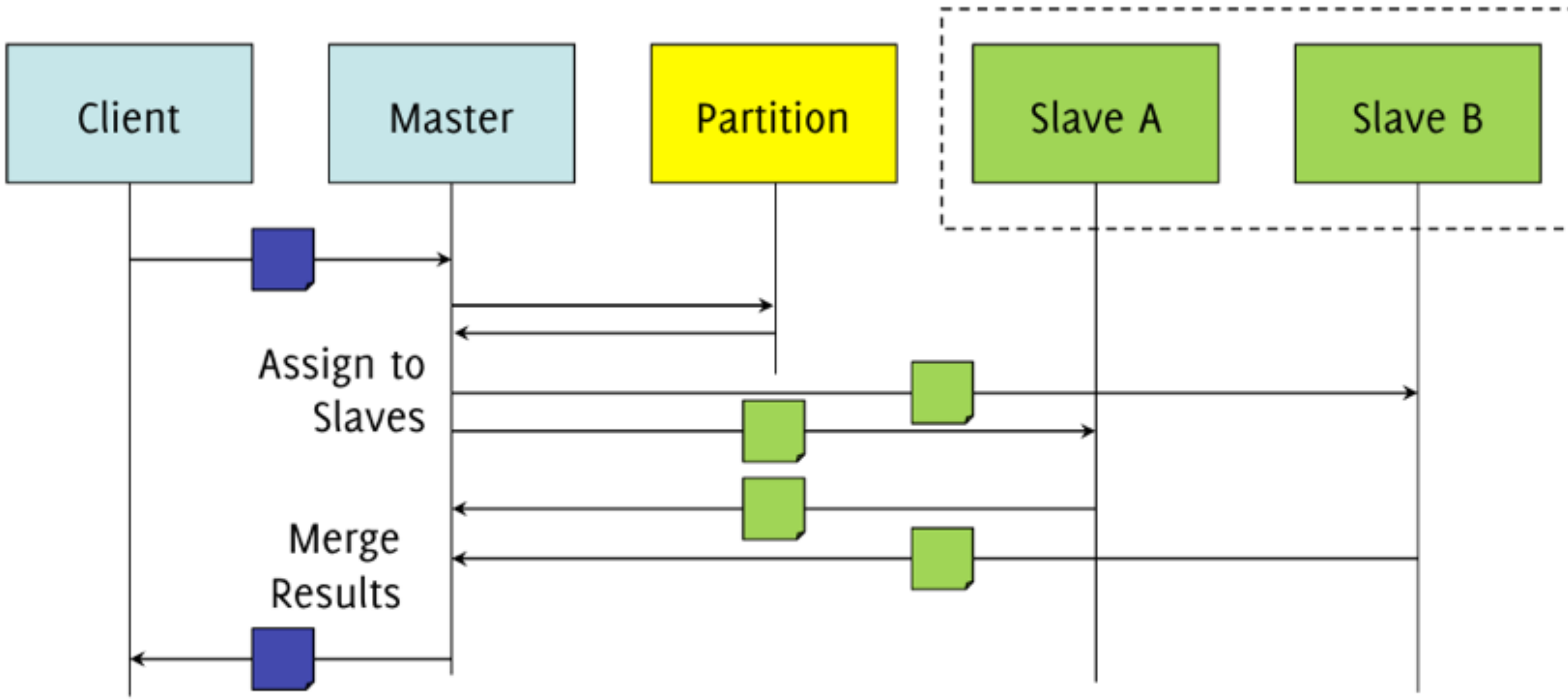
Add a layer hiding asynchronous interactions behind a synchronous interface



Events

Master/Slave

split a large job into smaller independent partitions which can be processed in parallel



Composition

Architectural vs Design Patterns

Express fundamental structural organizations

Specify relationships among (sub-)systems

Vs

Capture roles in solutions that occur repeatedly

Define the relationships among roles

Architectural Styles

Named collections of architectural decisions that are applicable in a development context. They constrain architectural design decisions, are specific to the system within that context, and elicit beneficial qualities in each resulting system

Why Styles?

A common vocabulary for the design elements

improve communication by shared understanding

A predefined configuration and composition rules

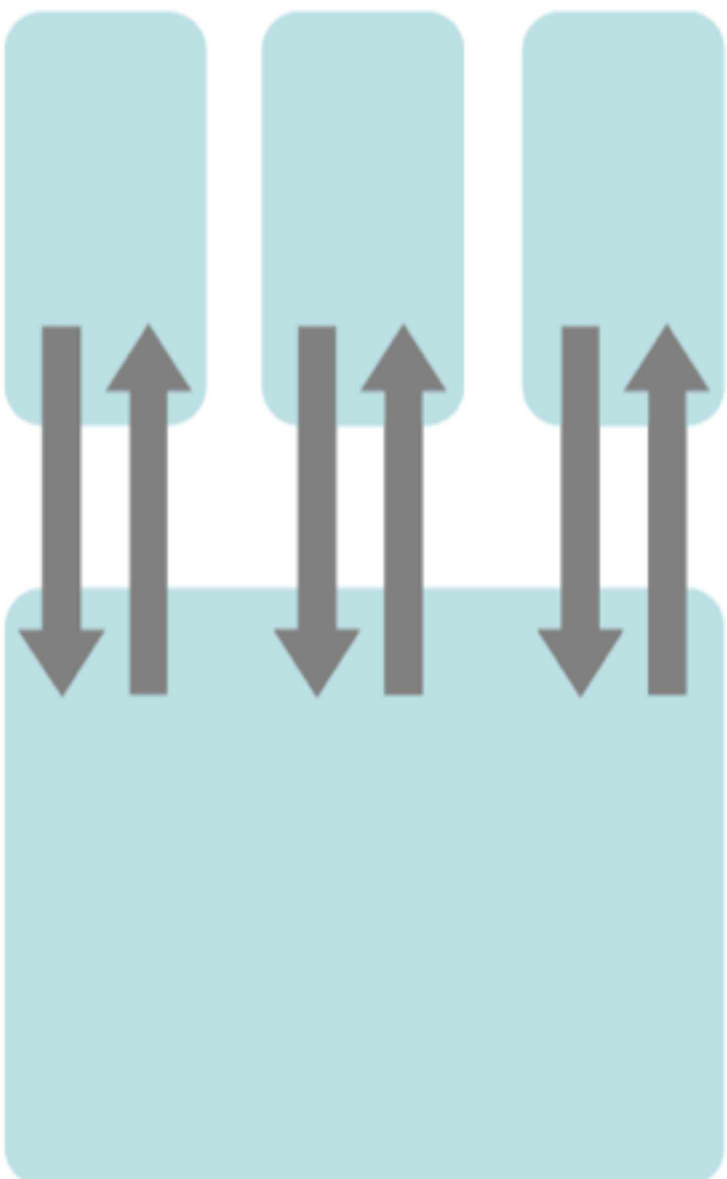
known benefits and limitations

ensure quality attributes if constraints are followed

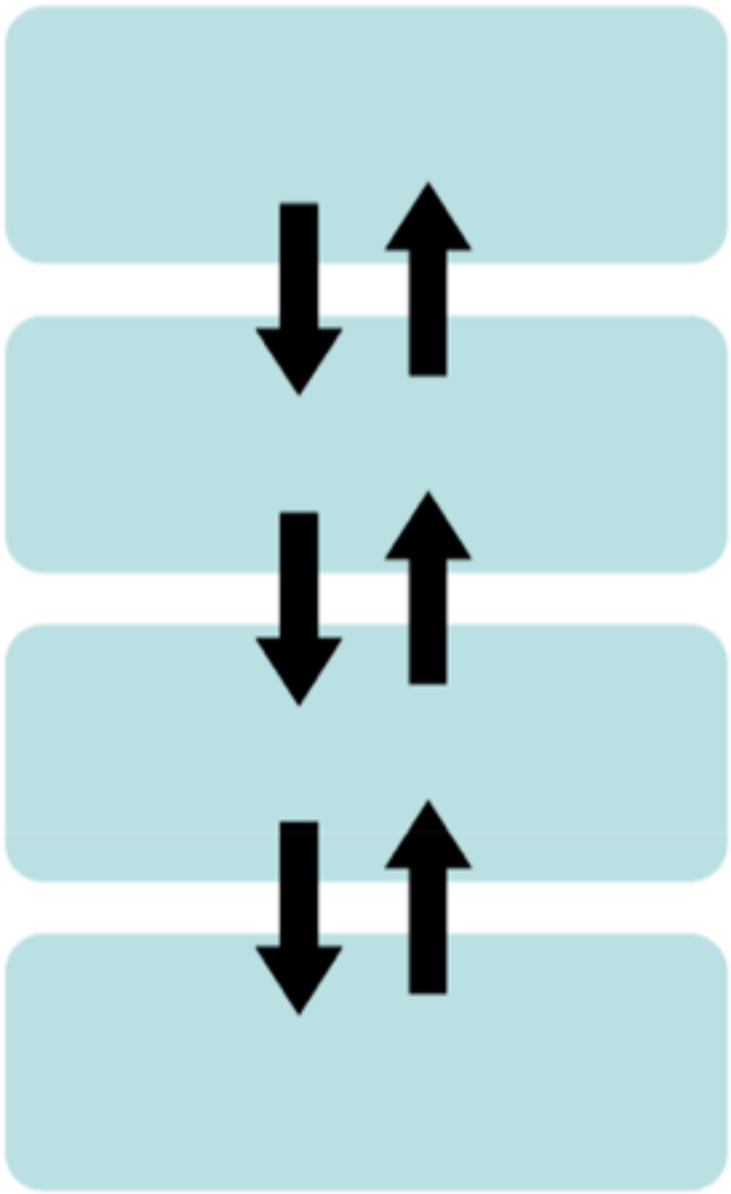
Style-specific analyses and visualizations

Monolithic

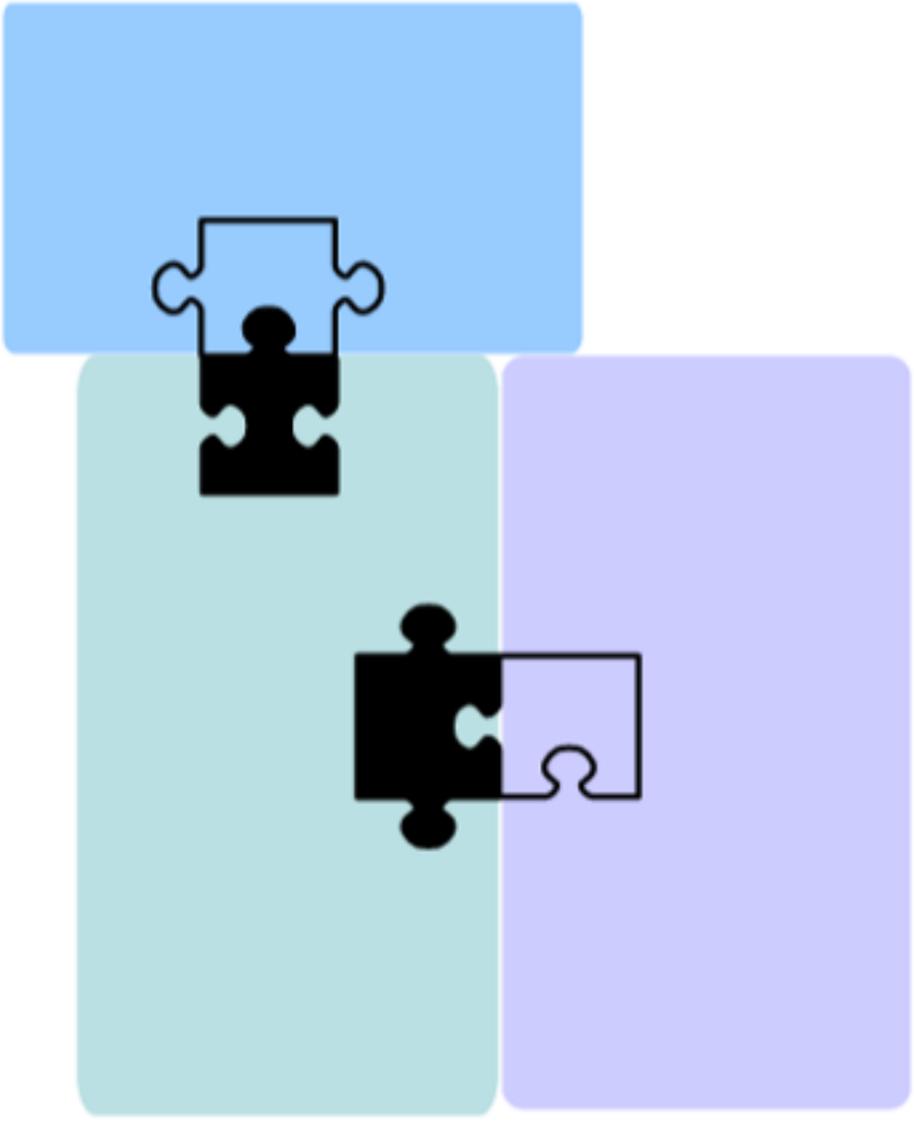
Client/Server



Layered



Plug-in



Styles vs Patterns

General constraints	Fine-grained constraints
Architecture with superior properties	Specific to recurrent problems
Styles must be refined and adapted	The same pattern can be used many times
Usually there is one dominant style	Many patterns are usually combined

Summary

- A great architecture likely combines aspects of several other architectures
- Do no limit to just one pattern, but avoid the use of unnecessary patterns
- Different styles lead to architectures with different qualities, and so might do the same style
- Never not stop at the choice of patterns and styles: further refinement is needed

Modeling

Why modeling?

- Record decisions
- Communicate decisions
- Evaluate decisions
- Generate artifacts

What do we model ?

- The system-to-be (Design model)
  - *Static architecture*
  - *Dynamic architecture*
- Quality attributes and non-functional properties
- The problem (Domain model)
- The environment (System context and stakeholders)
- The design process

Design Model

Boundary Model

System Context
Interfaces/API
Quality Attributes

Externally visible behavior

Internal Model

Software Components
Software Connectors
Component assembly

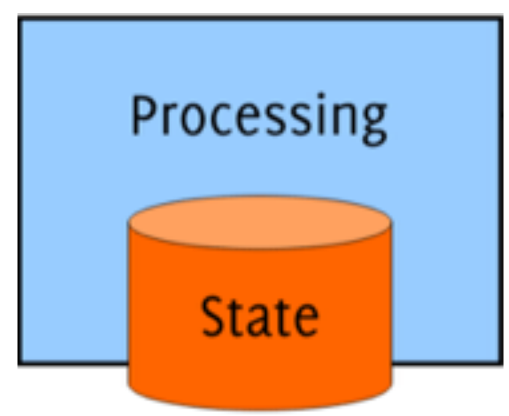
Internal behavior

Software Components

Reusable unit of composition

Can be composed into larger systems

Locus of computation

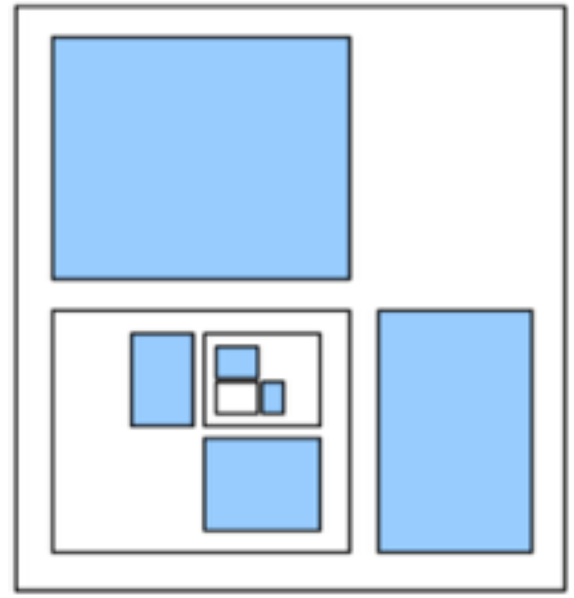


State in a system

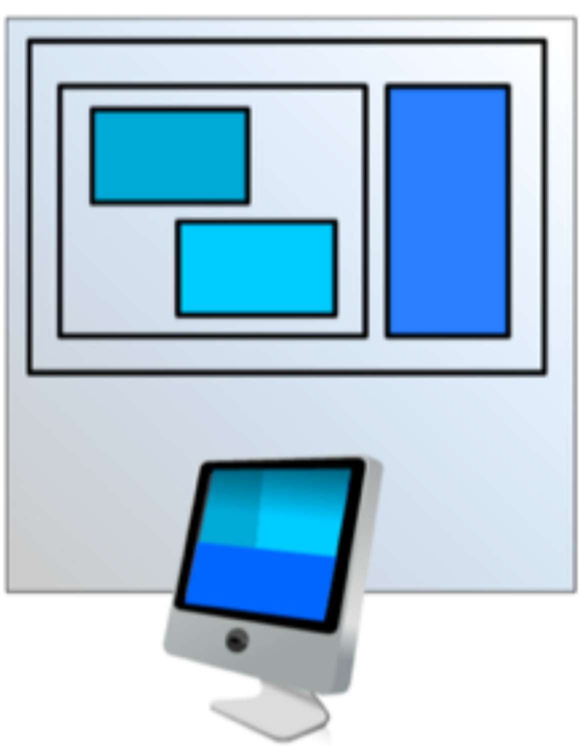
Application-specific — Infrastructure

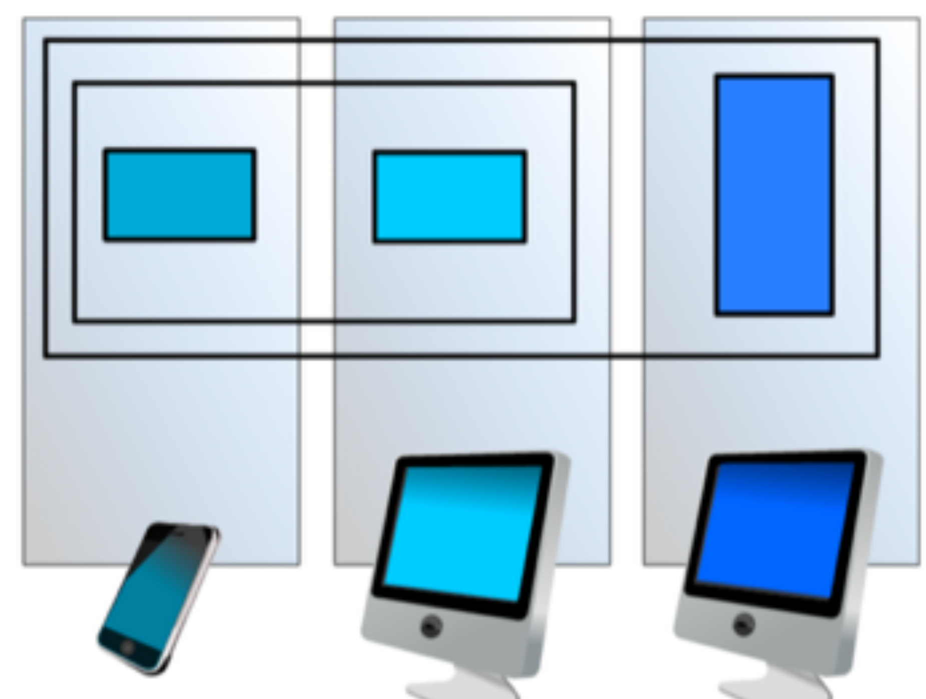
Media Player *Web Server*

Math Library *Database*

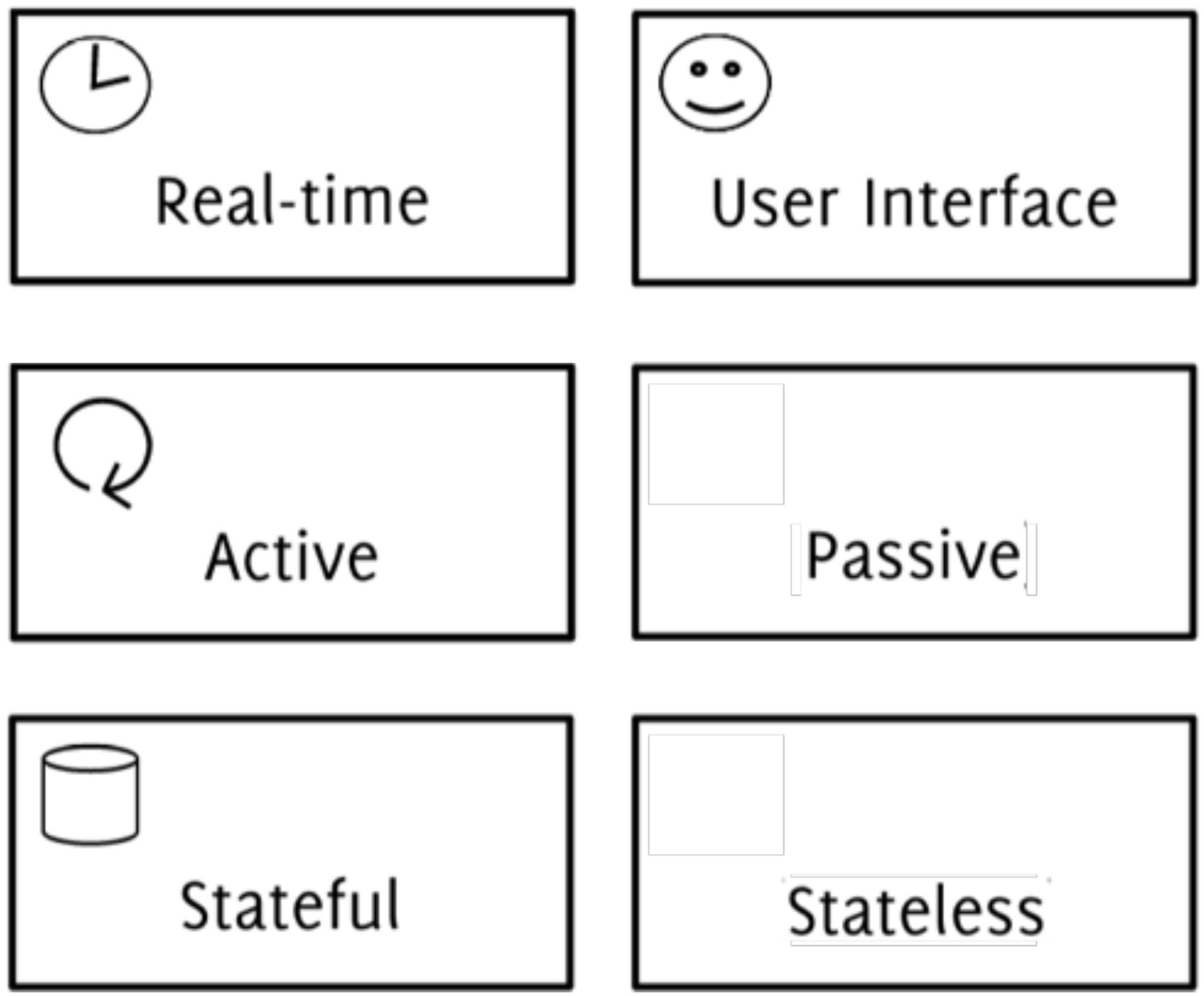


Composition and Distribution





Component Roles



Components

Encapsulate state and functionality
Coarse-grained
Black box architecture elements
Structure of architecture

VS

Objects

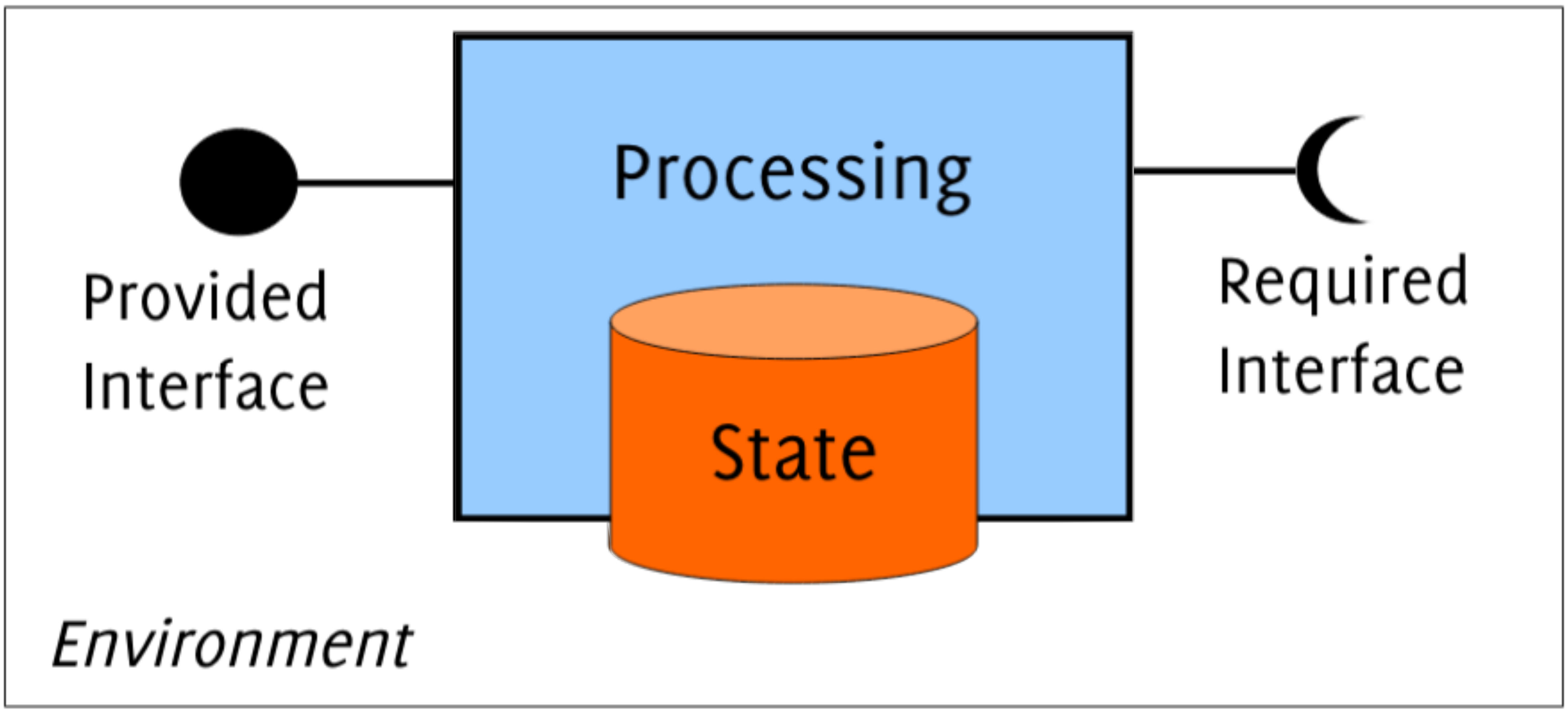
Encapsulate state and functionality
Fine-grained
Can “move” across components
Identifiable unit of instantiation

VS

Modules

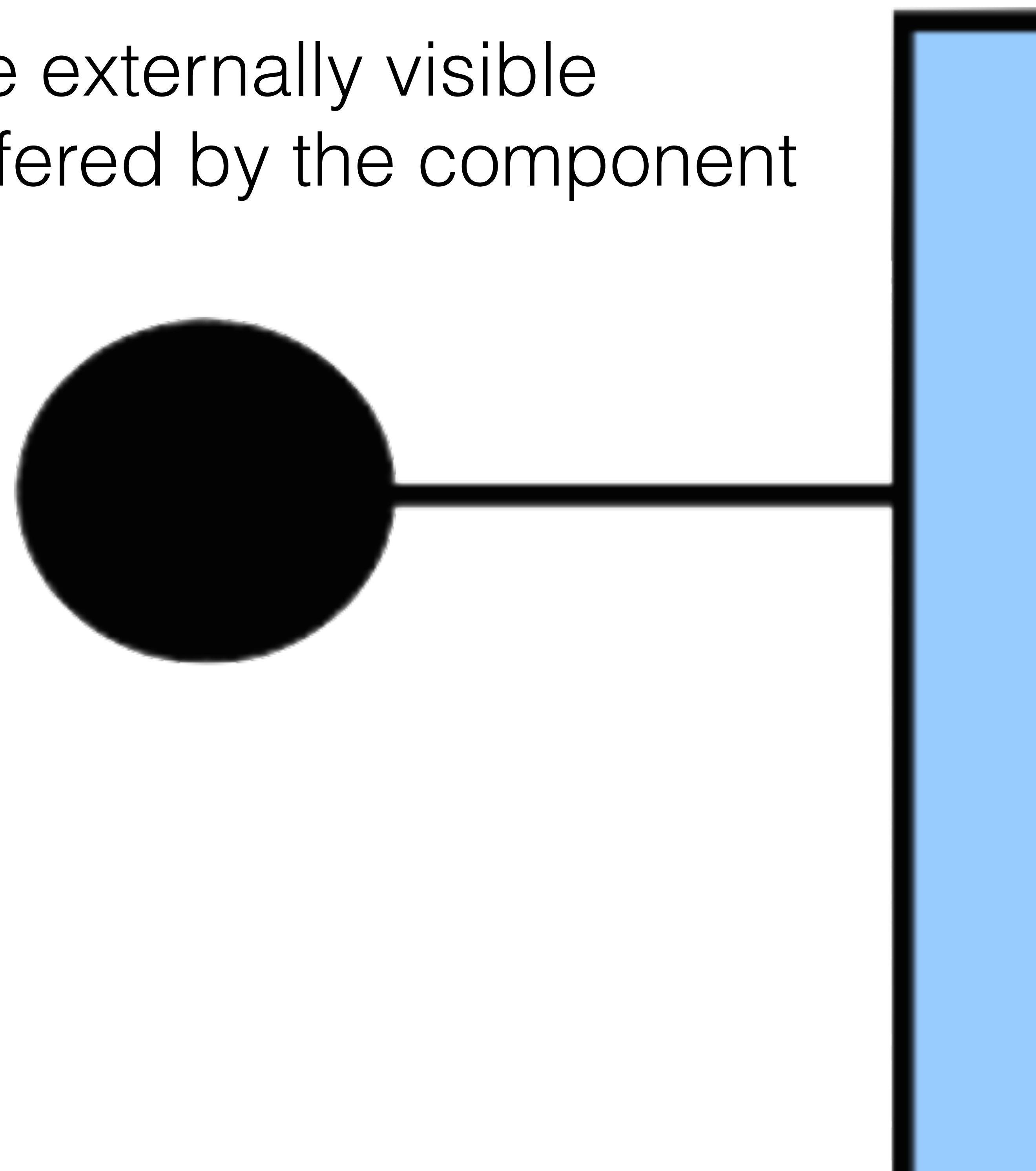
Rarely exist at run time
May require other modules to compile
Package the code

Component Interfaces



Provided Interfaces

- Specify and document the externally visible features (or public API) offered by the component
    - *Data types and model*
    - *Operations*
    - *Properties*
    - *Events and call-backs*

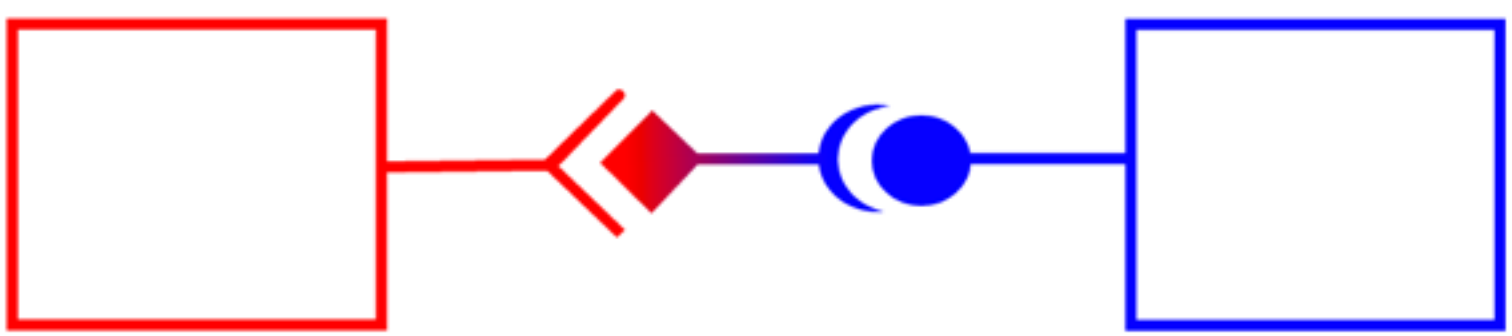


Required Interface

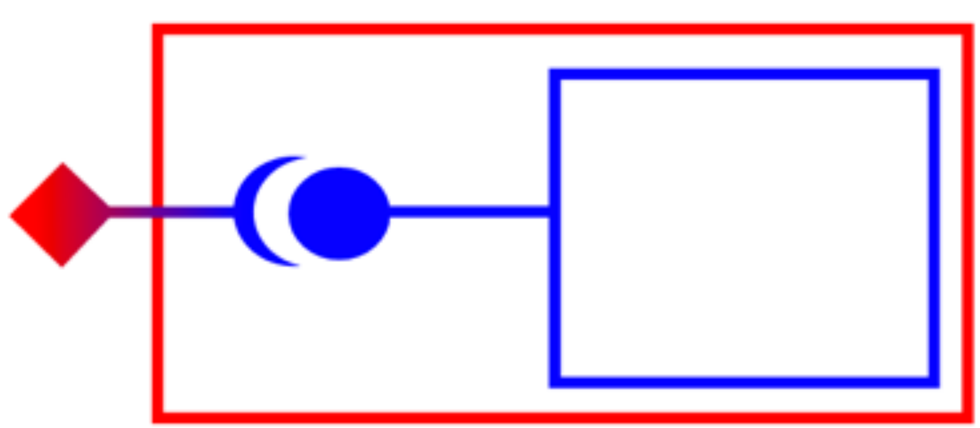
- Specify the conditions upon which a component can be reused
  - *The platform is compatible*
  - *The environment is setup correctly*

Compatible Interfaces

Component interfaces must match perfectly to be connected

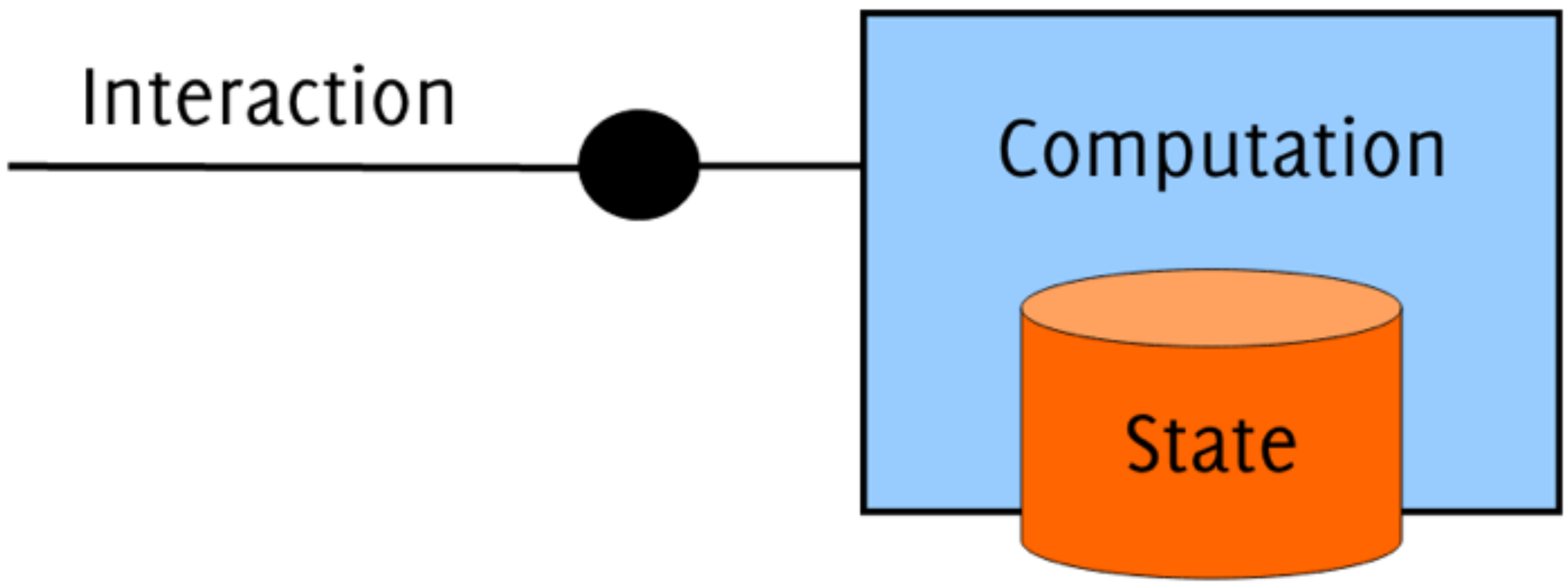


Adapter



Wrapper

Software Connectors



Model static and dynamic aspects of the **interaction** between component interfaces

Connector Roles

- Communication

  deliver data and transfer of control, support different communication mechanisms, quality of the delivery

- Coordination

  control the delivery of data, separate control from computation

- Conversion

  enable interaction of mismatched components

- Facilitation

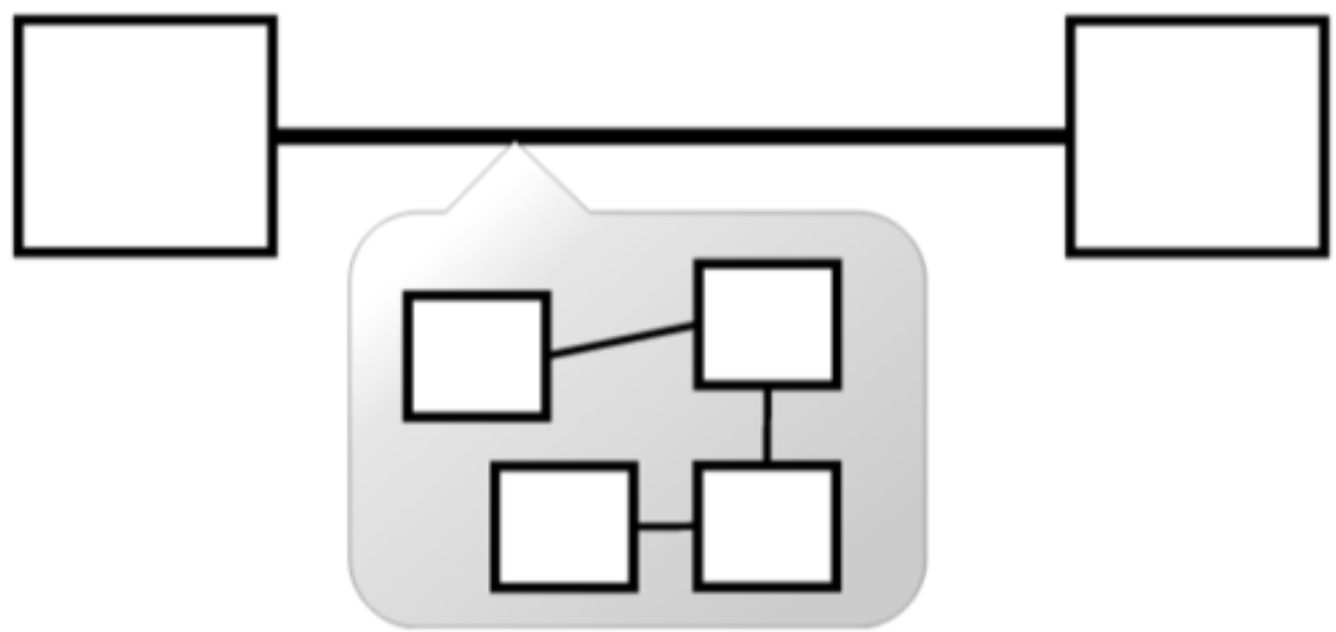
  mediate the interaction among components, govern access to shared information, provide synchronization

Connectors, not Components!

Connectors are not usually directly visible in the code, which is not true for components

Connectors are mostly application-independent, while components can be both application-dependent or not

Connectors are abstractions

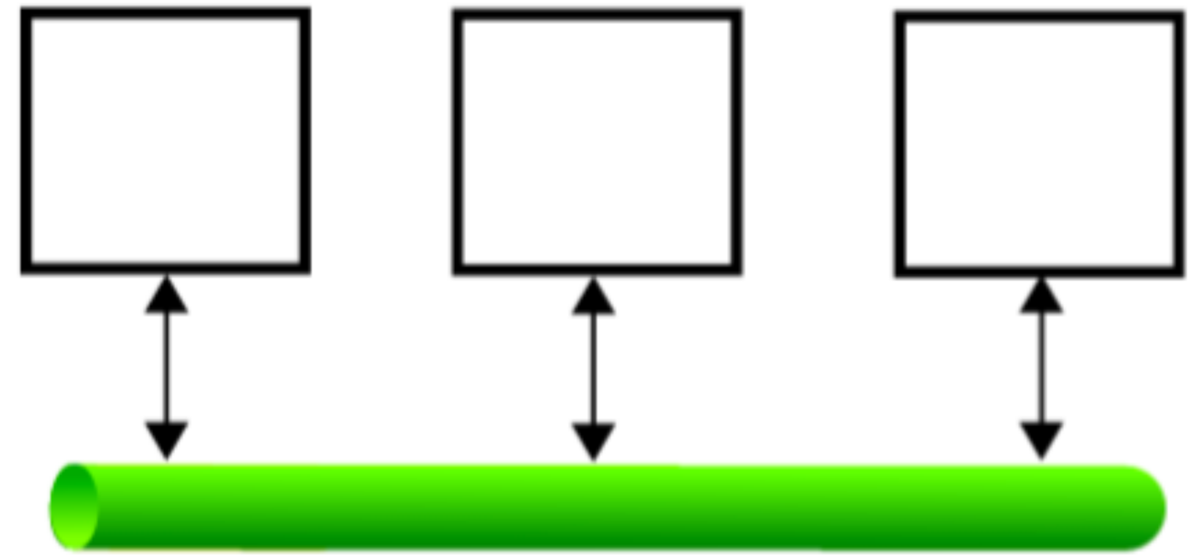


When to hide components inside a connector ?

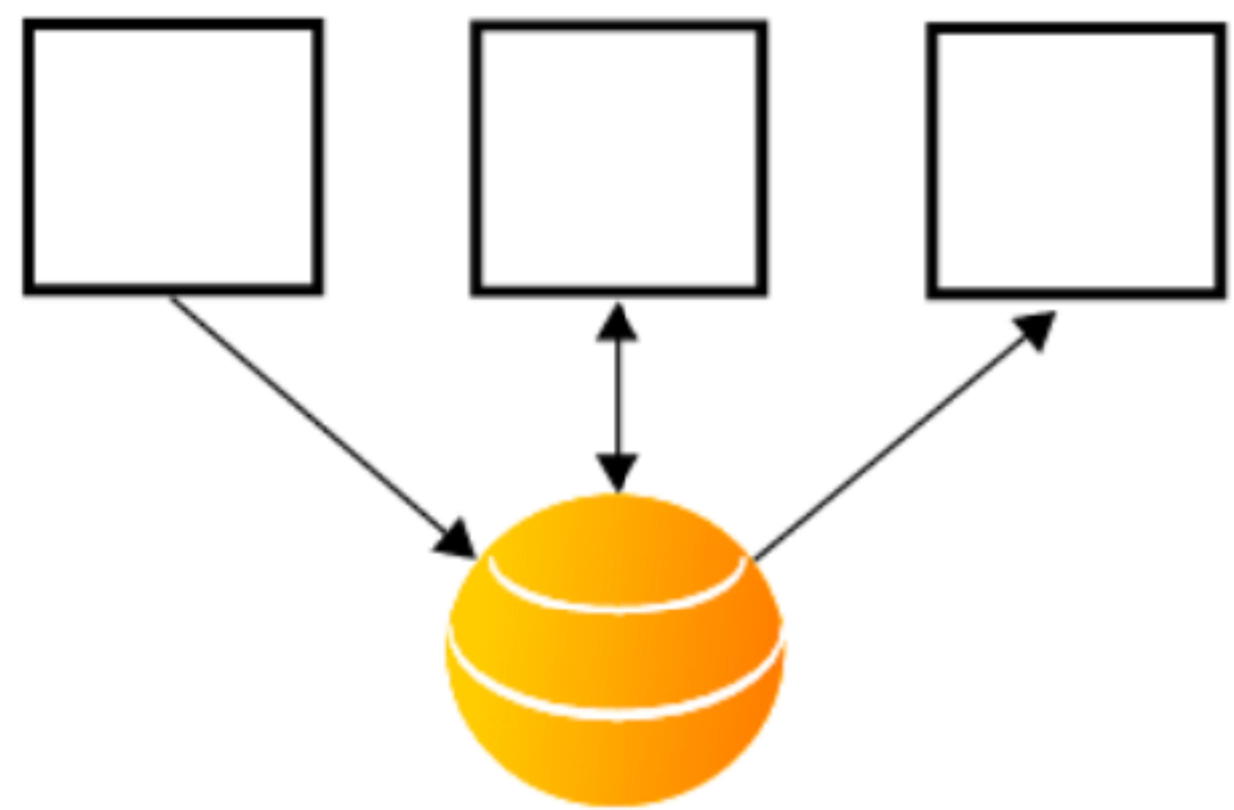
Remote Procedure Call

Stream

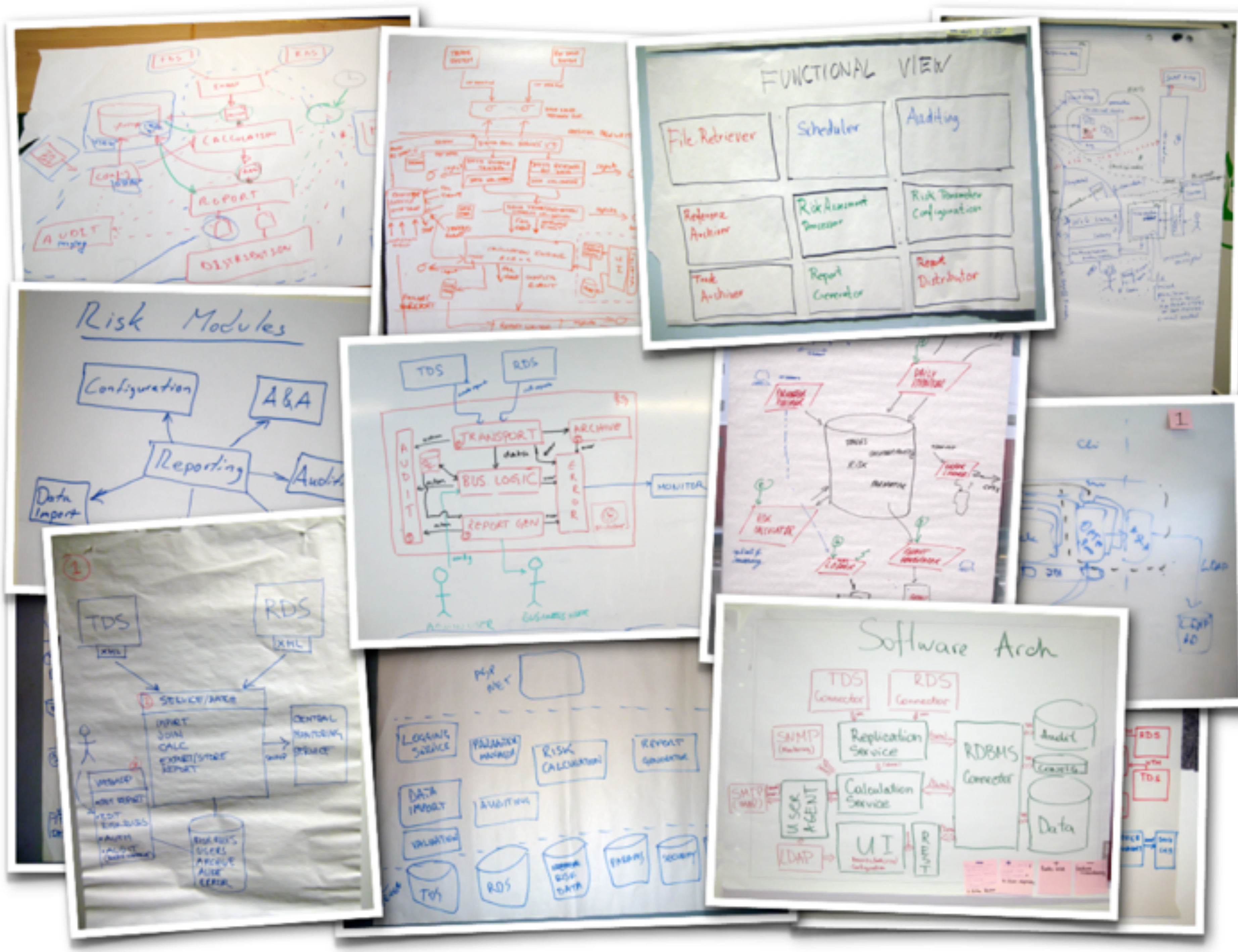
Message Bus

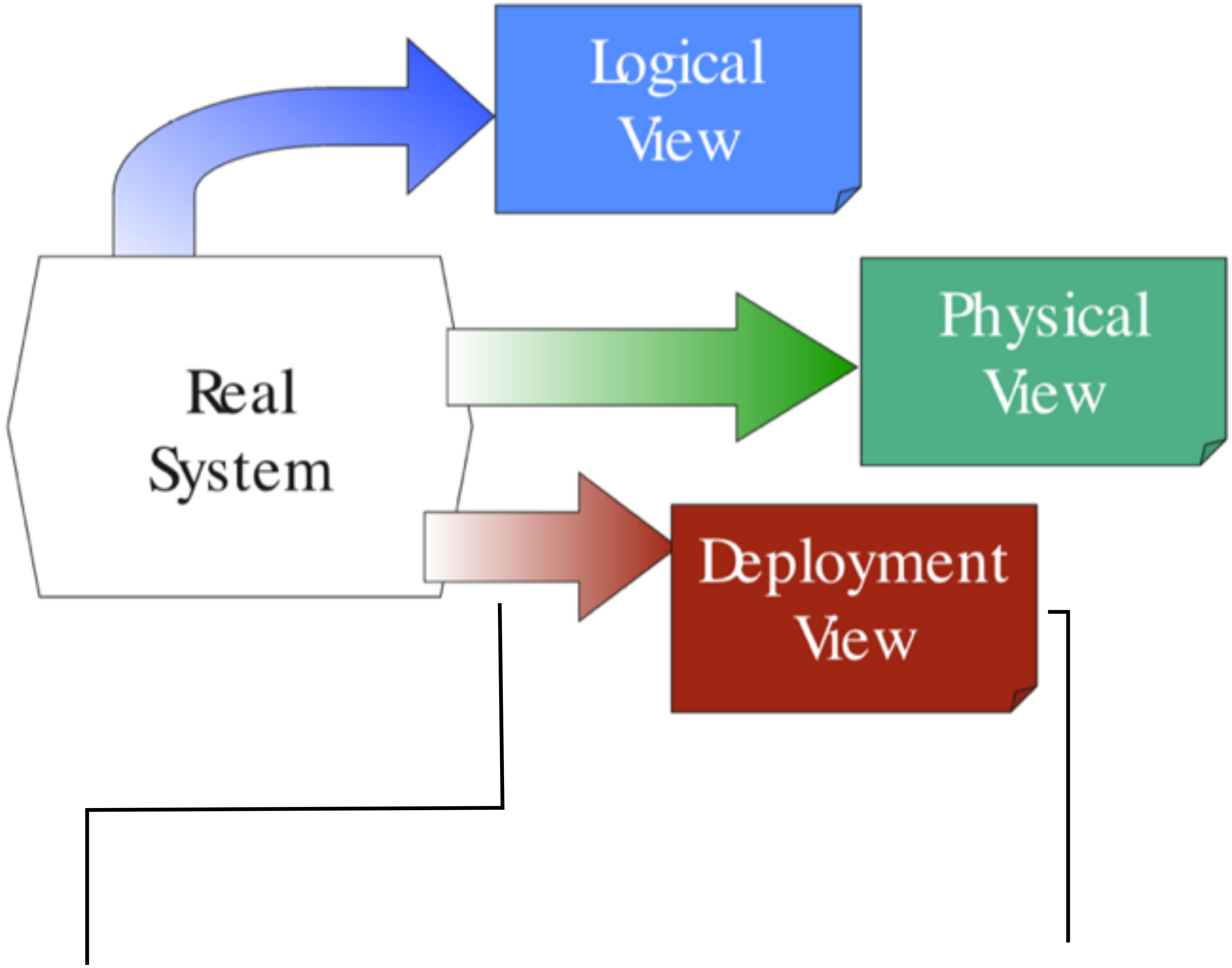


The Web



Views and Viewpoints





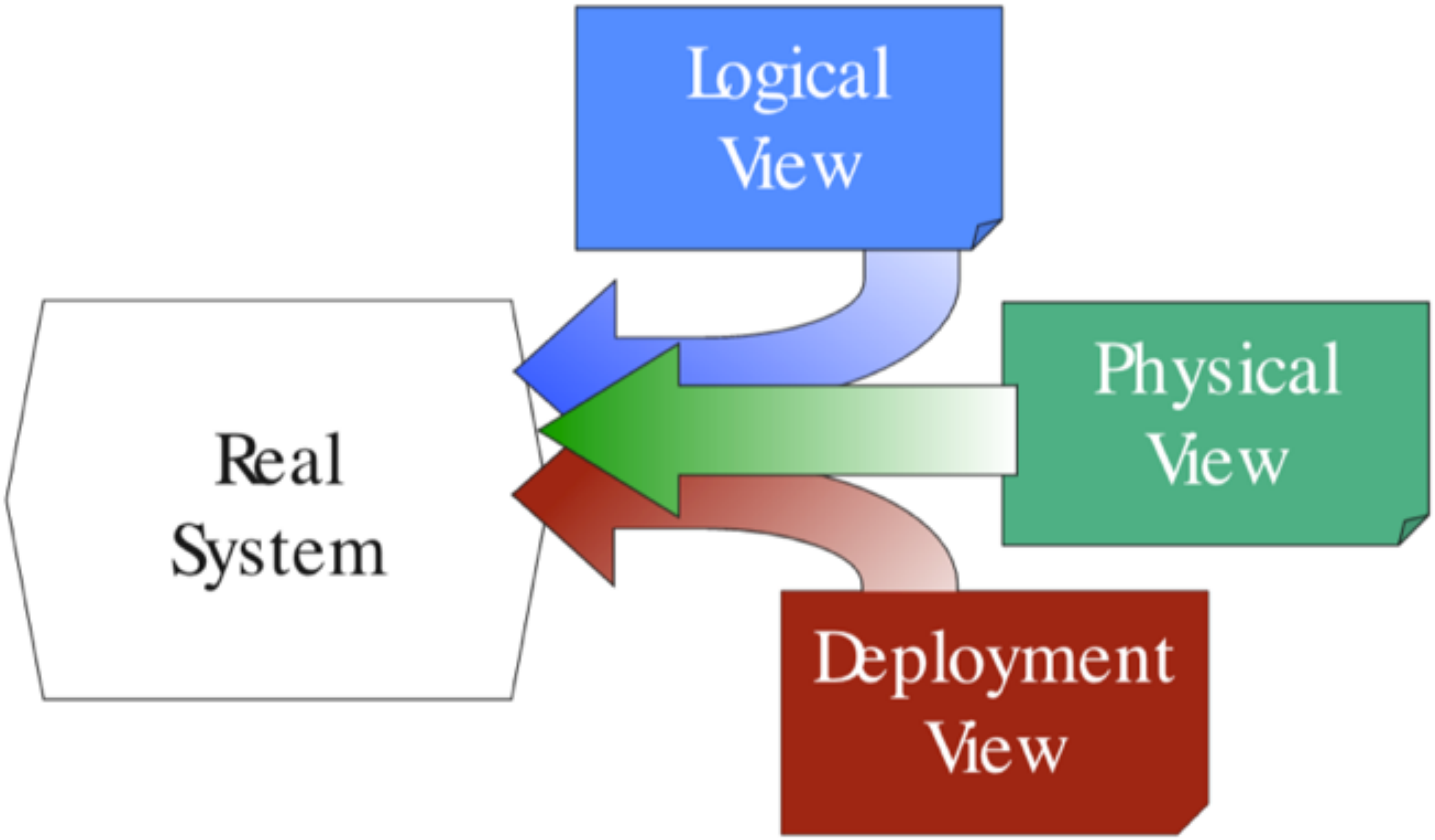
Viewpoint

The common concerns shared by a view

View

A subset of related architectural design decisions

Consistency



Views are not always orthogonal and might become inconsistent if design decision are not compatible

How many views?

- 5 by Taylor et al.: Logical, Physical, Deployment, Concurrency, Behavioral

- 3 by Bass et al.: Component & Connector, Module View, Behavior

- 4+1 by Kruchten: Logical, Physical, Process, Development, and Scenarios

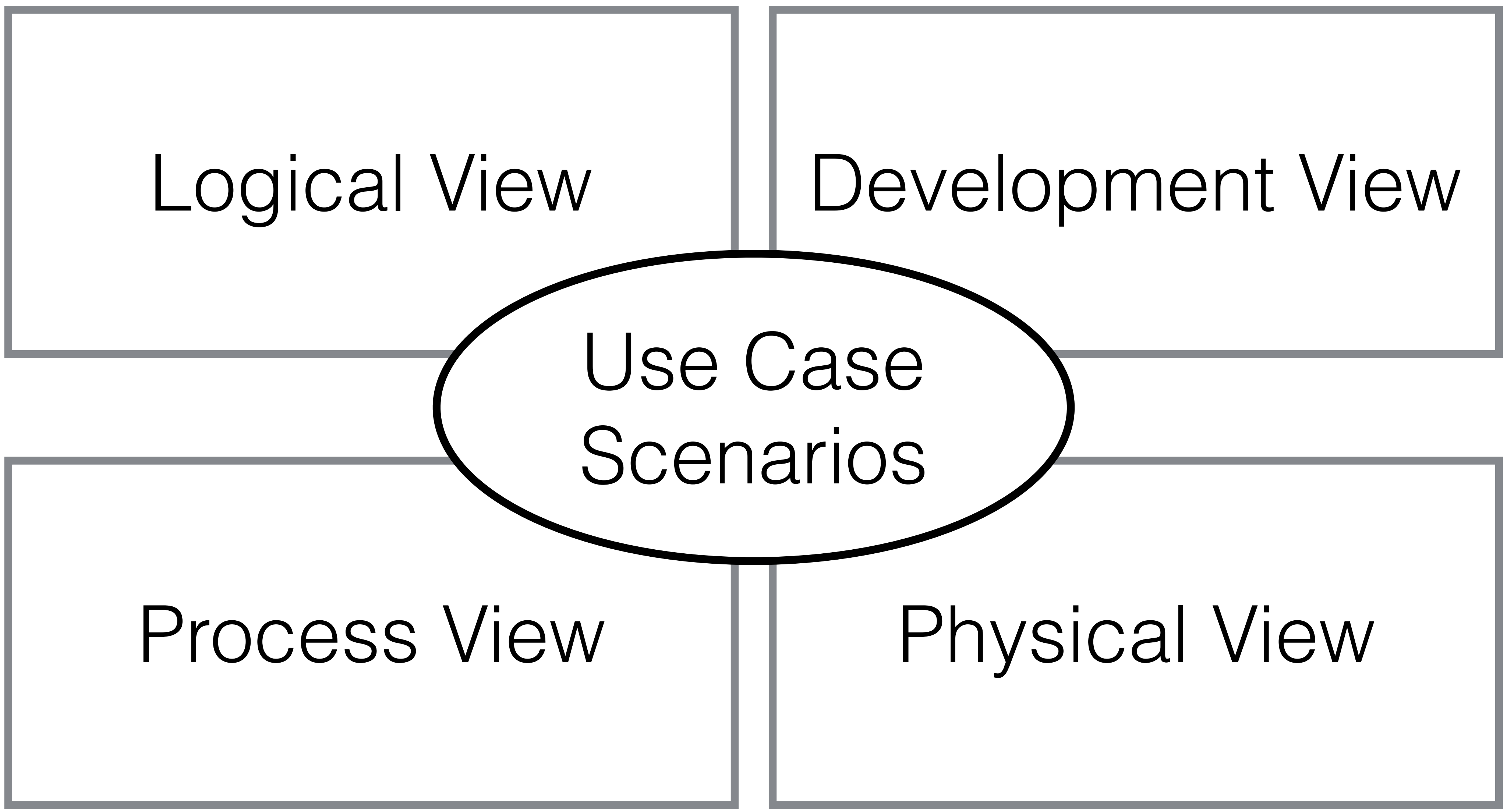
How many views?

- 5 by Taylor et al.:
  Concurrency, Behavioral

- 3 by Bass et al.:
  View, Behavior

- 4+1 by Kruchten: Logical, Physical, Process, Development, and Scenarios

4+1



Philippe Kruchten

Use Case Scenarios

- Unify and link the elements of the other 4 views
- Scenarios help to ensure that the architectural model is complete with respect to requirements
- The architecture can be broken down according to the scenarios and illustrated using the other 4 views

Music Player Scenarios

- Browse for new songs
- Pay to hear the entire song
- Download the purchased song on the phone
- Play the song

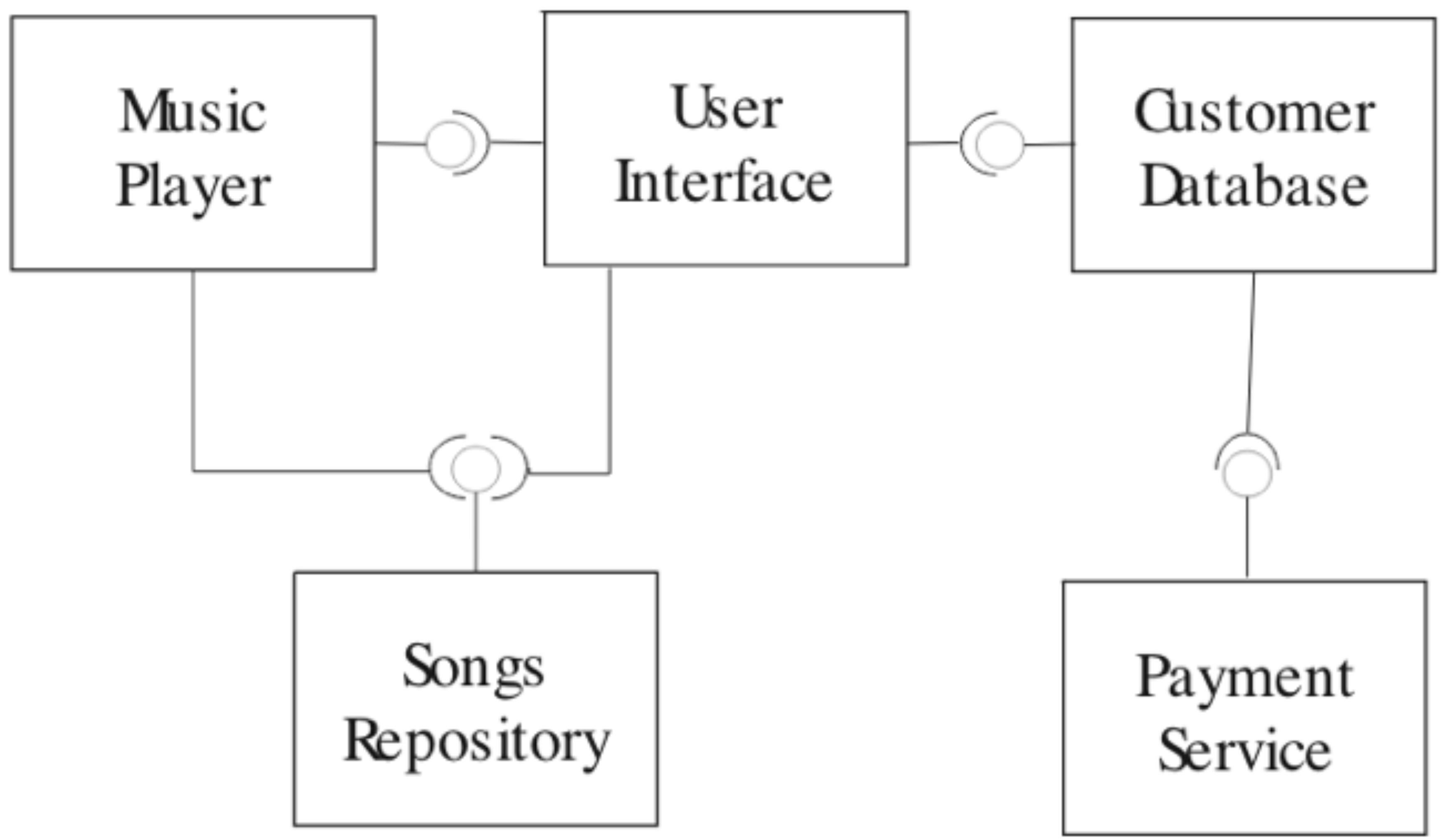
Logical View

- Decompose the system structure into software components and connectors

- Map functionality (use cases) onto the components

  - **Concern**: Functionality

  - **Target Audience**: Developers and Users


Music Player
User Interface
Customer Database
Songs Repository
Payment Service

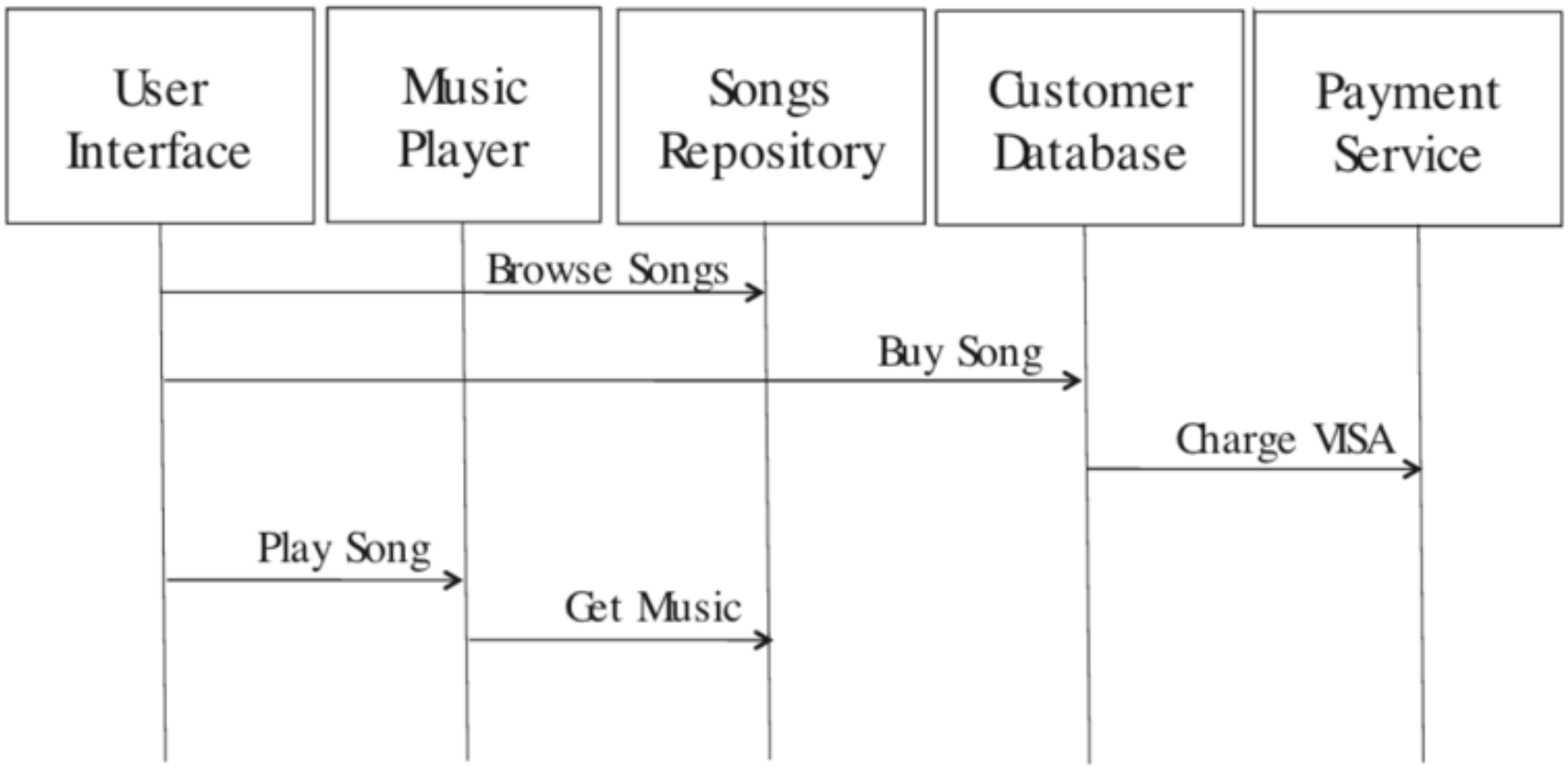
Process View

- Model the dynamic aspects of the architecture and the behavior its parts
  - *active components*
  - *concurrent threads*

- Describe how processes/threads communicate
  - *RPC*
  - *Message bus*

- **Concern**: Functionality, Performance

- **Target Audience**: Developers



Use Cases: Browse, Pay and Play For Songs

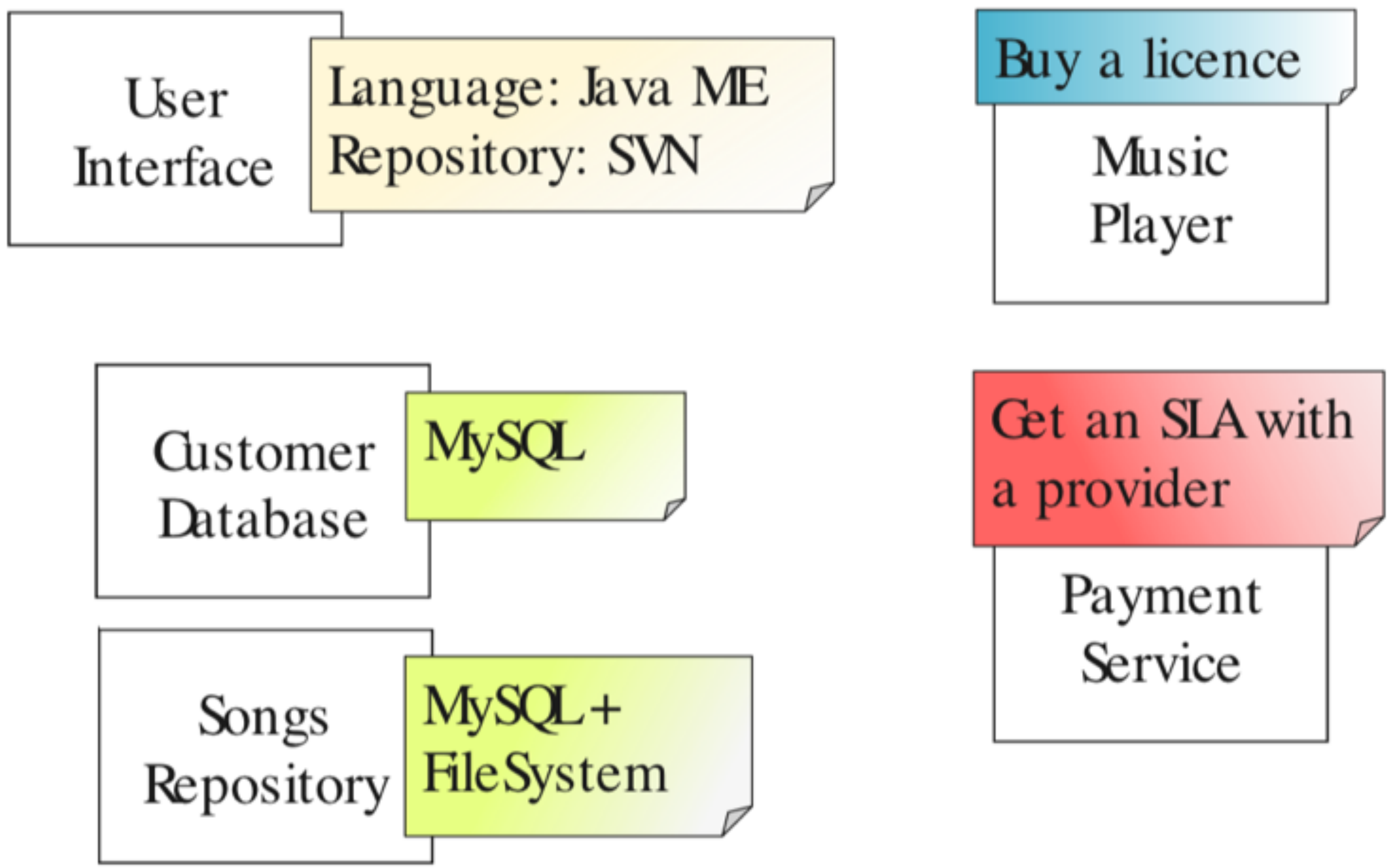
Development View

- Static organization of the software code artifacts
  - *packages*
  - *modules*
  - *binaries*

- Mapping between the elements in the logical view and the code artifacts

  - **Concern**: Reuse, Portability, Build

  - **Target Audience**: Developers

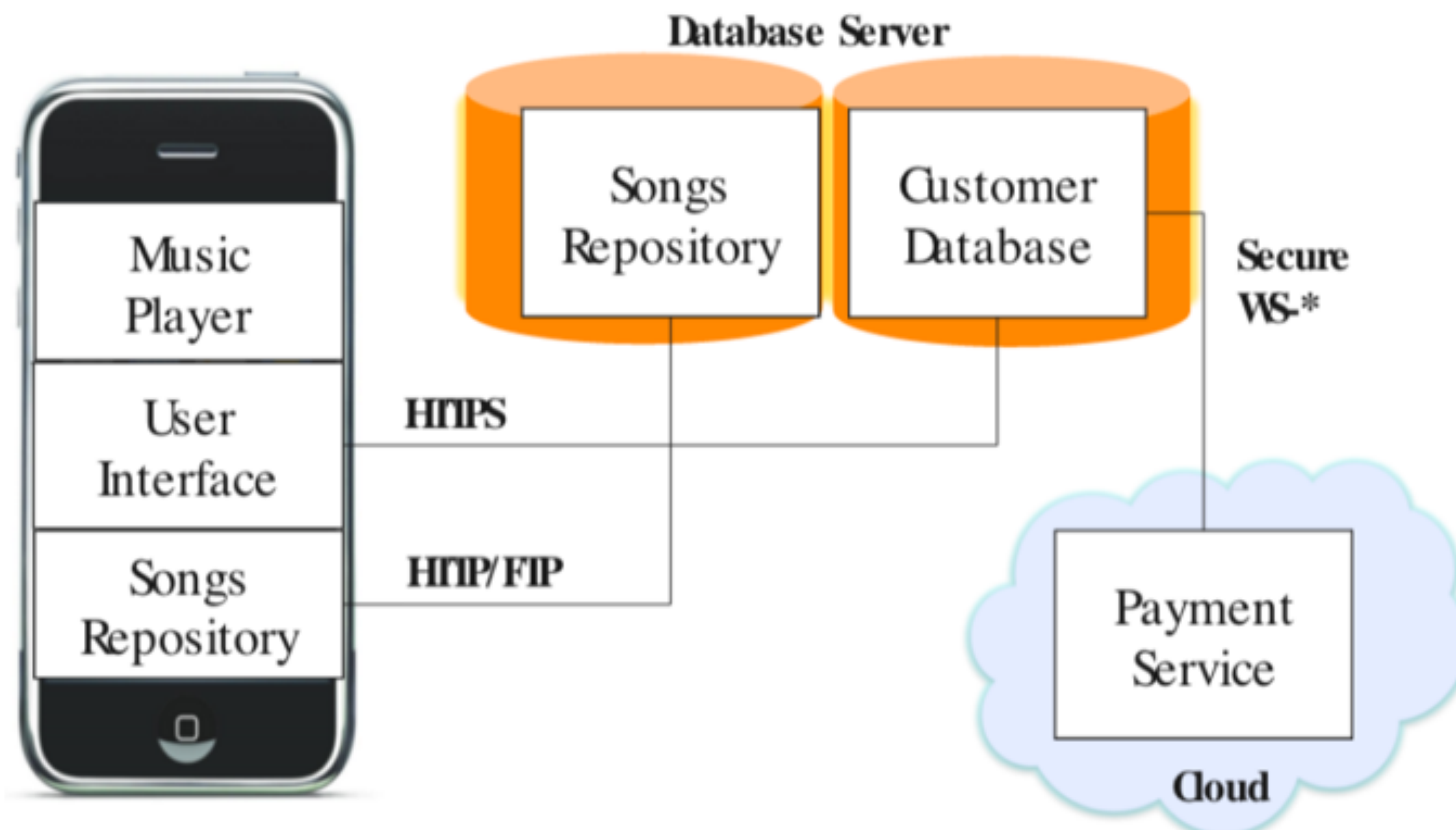

User Interface
Language: Java ME
Repository: SVN
Buy a licence
Music Player
Customer Database
MySQL
Get an SLA with a provider
Songs Repository
MySQL+ FileSystem
Payment Service

Physical View

- Hardware environment where the software will be deployed
    - *hosts*
    - *networks*
    - *storage*

- Mapping between logical and physical entities

- **Concern**: Quality attributes
- **Target Audience**: Operations


Database Server
Music Player
User Interface
Songs Repository
Songs Repository
Customer Database
HTTPS
HTTP/FTP
Secure WS-*
Payment Service
Cloud