h e g

Module 626-1

ARCHITECTURE DU SYSTÈME D'INFORMATION ET DESIGN PATTERNS

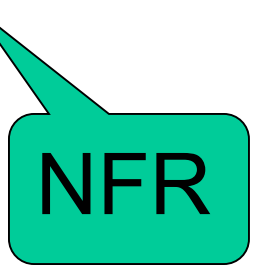
Dr. Stefan Behfar

session 1

Architecting systems

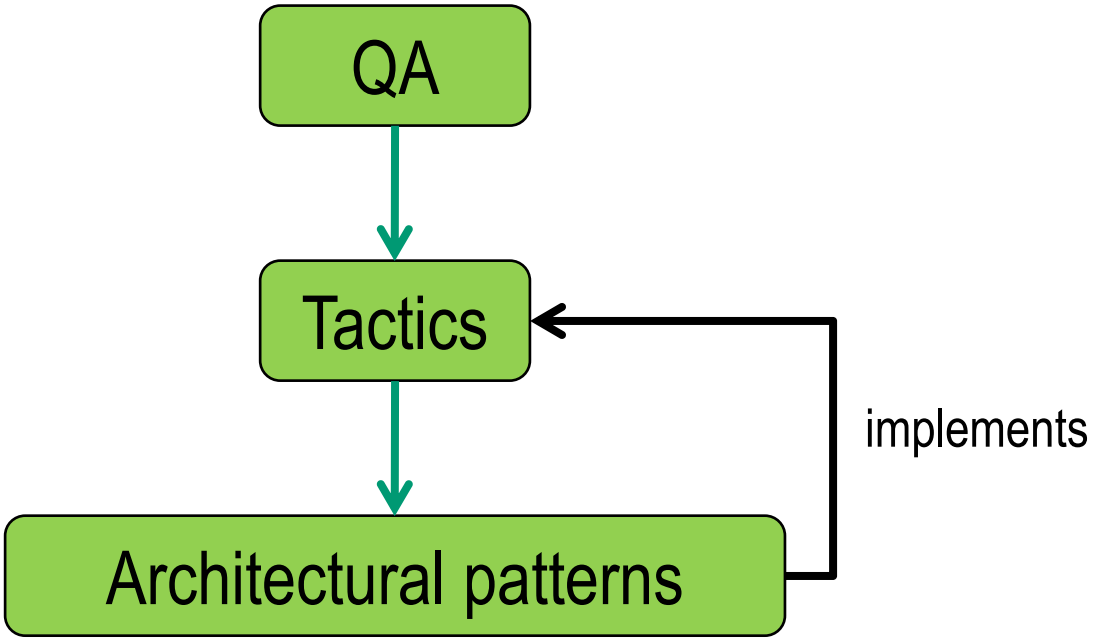
« **Architecting**…is helping determine the relative requirement priorities, acceptable performance, cost and schedule, taking into account such factors as technology risks, projected market size, likely competitors moves, economic trends, political regulatory requirements, project organization and the appropriate «illities» (availability, operability, manufacturability, survivability,…). »

[Rechtin E. – Systems Architecting. Prentice Hall, 1991]



© Philippe Dugerdil, HEG

Designing the high level architecture



© Philippe Dugerdil, HEG

h e g

Designing high level architecture based on NFR

2 concepts:

- Architectural tactic

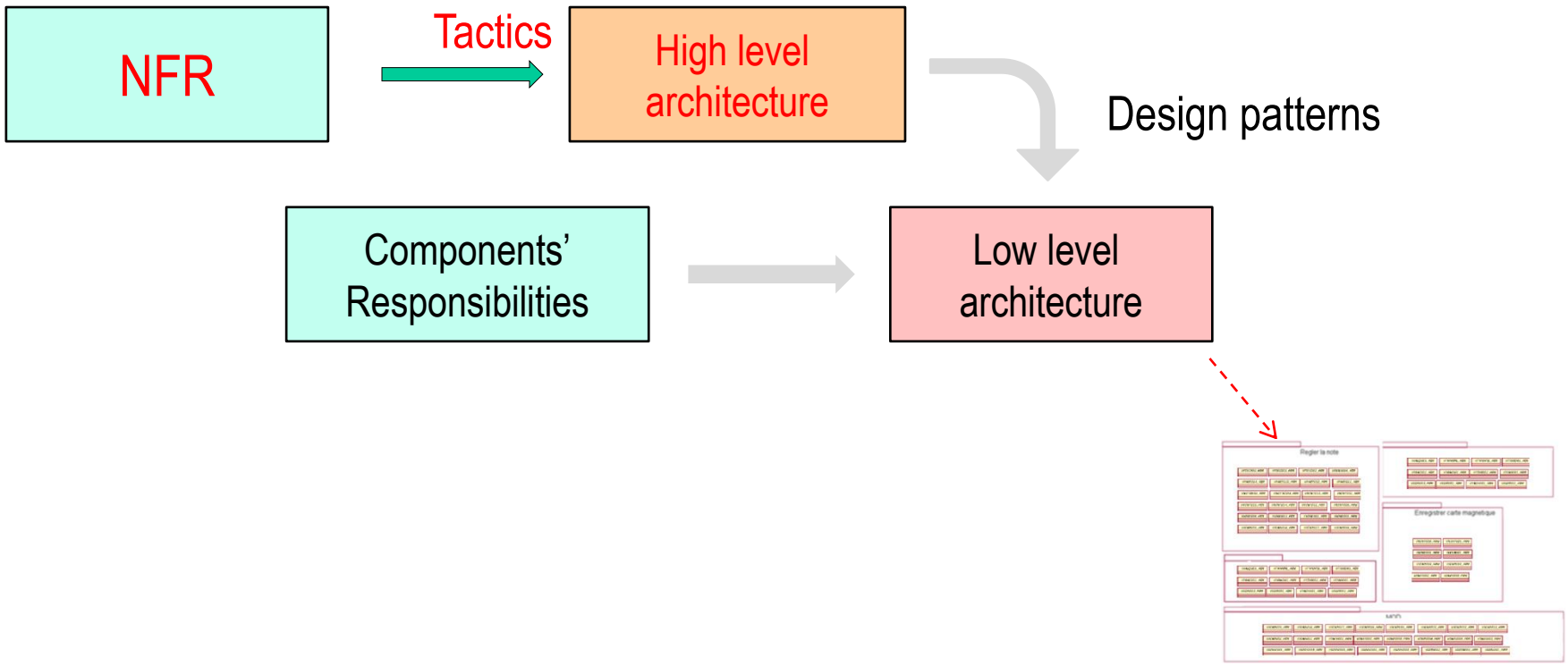
  An architectural tactic is a design decision that helps achieve a specific quality-attribute response. Such a tactic must be motivated by a quality-attribute analysis model.

- Architectural pattern (architectural *style*)

  Architectural patterns express fundamental structural organization schemas for software systems. They provide a set of predefined subsystems, specify their responsibilities and include rules and guidelines for organizing the relationships between them.

© Philippe Dugerdil, HEG

An overview: architecting steps



© Philippe Dugerdil, HEG

h e g

QA: performance

- How long does it take for the system to respond to an event (latency) ?
  - Source of complication: the number of event sources and their arrival sequence.
- Source of performance problems
  - Availability of required resources

© Philippe Dugerdil, HEG

Performance tactics

- Resource demand
  - Increase computational efficiency (better algorithms)
  - Reduce computational overhead (do not waste processor time)
  - Manage event rate (limit computational needs)
- Resource management
  - Introduce concurrency (threads)
  - Maintain multiples copies of either data or computation (cache)
- Resource arbitration
  - First-in first-out
  - Fixed priority scheduling

© Philippe Dugerdil, HEG

Availability tactics

- Availability deals with system failures and their consequences.
  - A failure occurs when the system no longer delivers a service consistent with its specification.
  - A fault becomes a failure when it is observable by the user of the system.
    - Then, one way to avoid a failure is to detect and correct a fault before it becomes observable by the user.

© Philippe Dugerdil, HEG

Availability tactics

- Fault detection
  - Ping / echo
  - Heartbeat
  - Exception
- Fault recovery
  - Voting
  - Active redundancy (hot restart) (mirroring)
  - Passive redundancy (backup)

to avoid failure

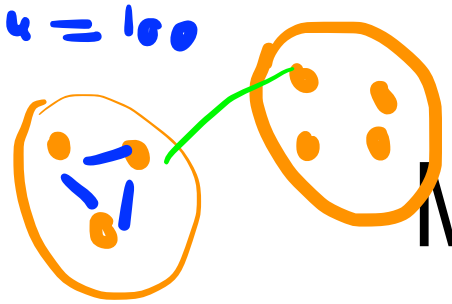
© Philippe Dugerdil, HEG

Why is modifiability an issue ? Dependencies

Kind of dependencies between modules / components:

1. Syntax of function / signature
2. Semantics of function / methods
3. Sequence of calls
4. Name (identity) of an interface
5. Location of a component
6. Quality of service / data
7. Shared access to resources

© Philippe Dugerdil, HEG



Modifiability tactics

- Localize modifications
  - Maintain semantic coherence (low coupling + high cohesion)
  - Anticipate expected change
  - Generalize the module (parameterization)
  - Limit possible options (extension points)
- Prevent ripple effect (limit dependencies)
  - Hide information
  - Maintain existing interface
  - Restrict communication paths
  - Use an intermediary

© Philippe Dugerdil, HEG

- In this session, we will discuss
  - Design in the large system
  - Software architecture
  - Design principles
  - Architectural patterns
  - What do we model

Design in the Large

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

© -Alessio Gambi - Saarland University

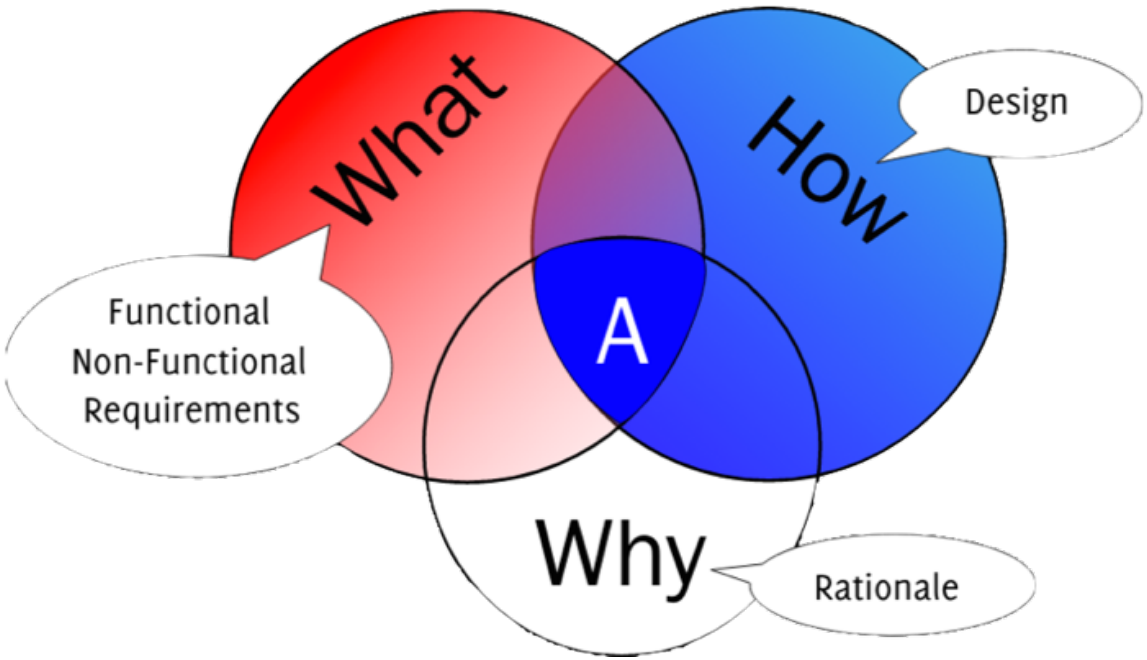
Software Architecture

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

N. Taylor et al.

© -Alessio Gambi - Saarland University

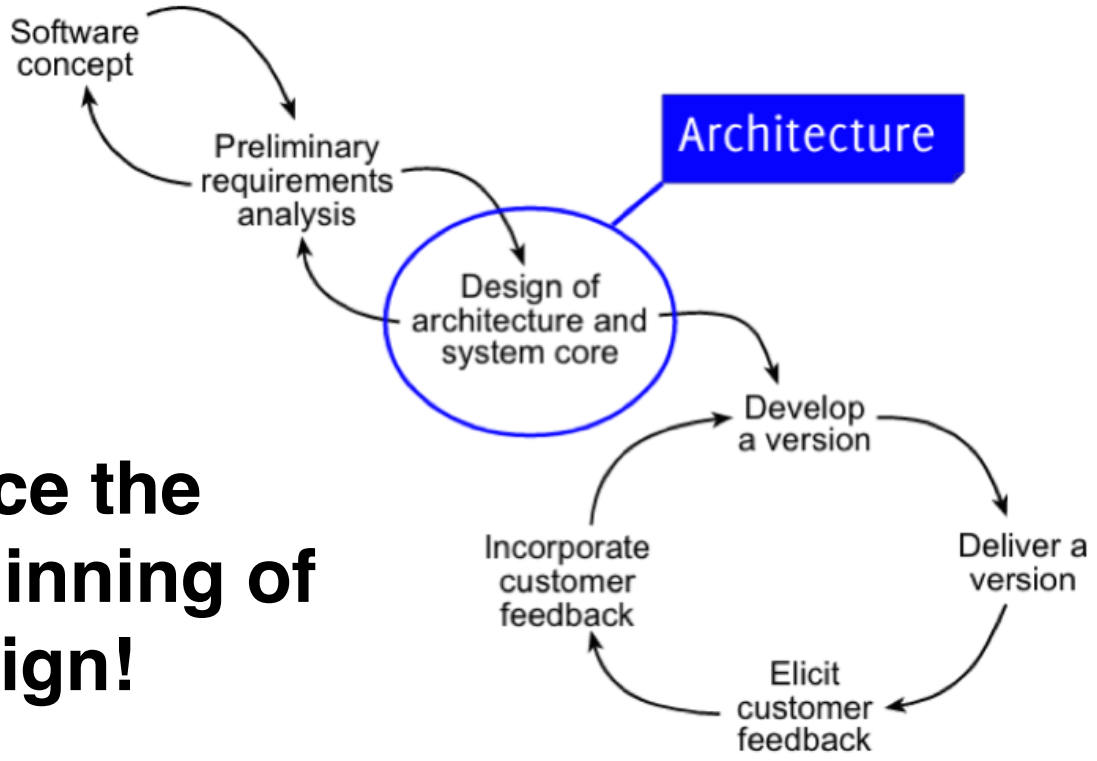
Abstraction



Manage complexity in the design

© -Alessio Gambi - Saarland University

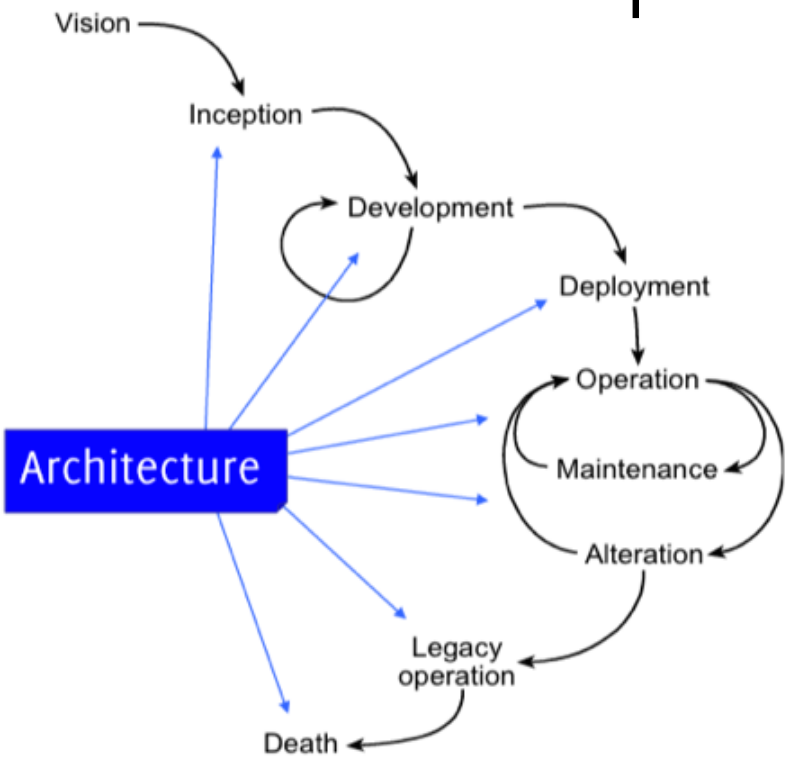
When Sw. Architecture Start ?



Since the beginning of design!

© -Alessio Gambi - Saarland University

When Sw. Architecture Stop ?



Never!

Architecture is NOT a phase of development

© -Alessio Gambi - Saarland University

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

© -Alessio Gambi - Saarland University

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

© -Alessio Gambi - Saarland University

Design Principles

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

© -Alessio Gambi - Saarland University

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.

© -Alessio Gambi - Saarland University

h e g

Problem

- Complex processing with several levels of abstraction from input to output

© Philippe Dugerdil, HEG

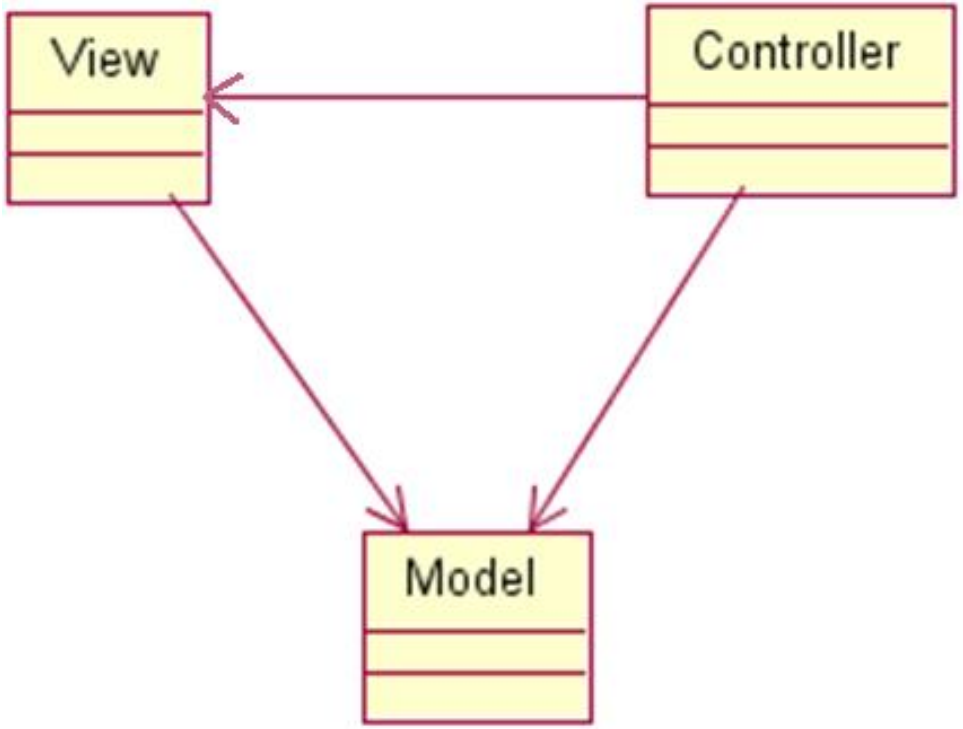
Model-View-Controller

The *Model-View-Controller* pattern is one of the best known and most common patterns in the architecture of *interactive* systems.

© -Alessio Gambi - Saarland University

h e g

Model-View-Controller



Concept designed back in the 70's in Alan's Kay at Xerox. Implemented in Smalltalk 76, Smalltalk 78 then in the first commercial version: Smalltalk-80 by Adele Goldberg's group.

© Philippe Dugerdil, HEG

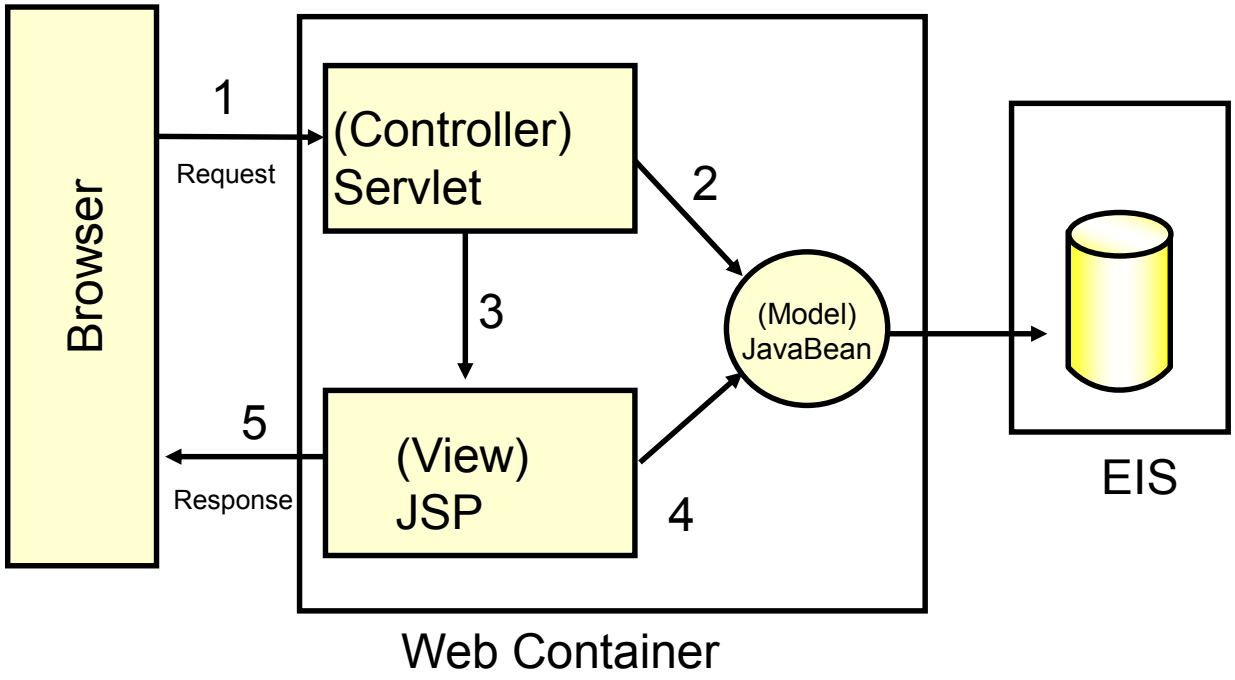
Implemented tactics

Modifiability tactics:

- Anticipate expected change
- Separate concerns

© Philippe Dugerdil, HEG

Exemple: good-old JSP / Servlet

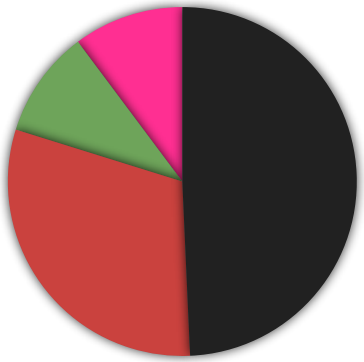


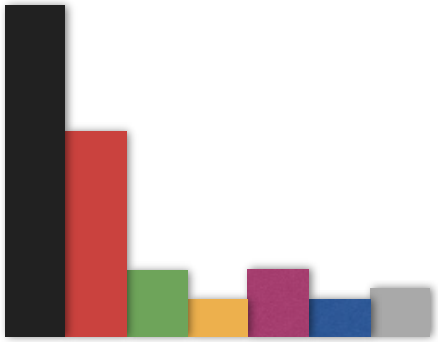
© Philippe Dugerdil, HEG

Example: Election Day

CDU/CSU	41,5 %
SPD	25,7 %
GRÜNE	8,4 %
FDP	4,8 %
LINKE	8,6 %
AfD	4,7 %
Sonstige	6,2 %

Bundestagswahl
22.09.2013





CDU/CSU	0,415
SPD	0,257
GRÜNE	0,084
FDP	0,048
LINKE	0,086
PIRATEN	0,047
Sonstige	0,062

© -Alessio Gambi - Saarland University

Problem

User interfaces are most frequently affected by changes.

- How can I represent the same information in different ways?
- How can I guarantee that changes in the dataset will be instantly reflected in all views?
- How can I change the user interface? (possibly at runtime)
- How can I support multiple user interfaces without changing the core of the application?

© -Alessio Gambi - Saarland University

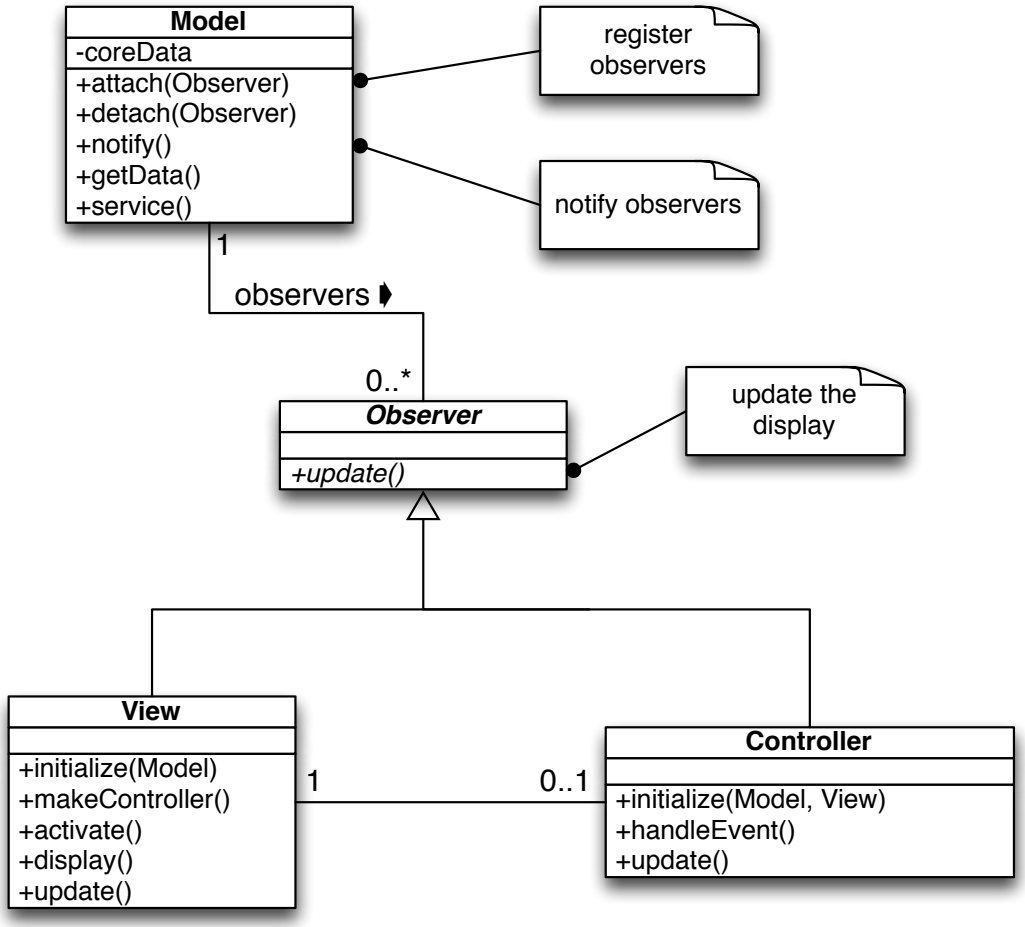
Solution

The *Model-View-Controller* pattern splits the application into three parts:

- The *model* is responsible for processing,
- The *view* takes care of output,
- The *controller* concerns itself with input

© -Alessio Gambi - Saarland University

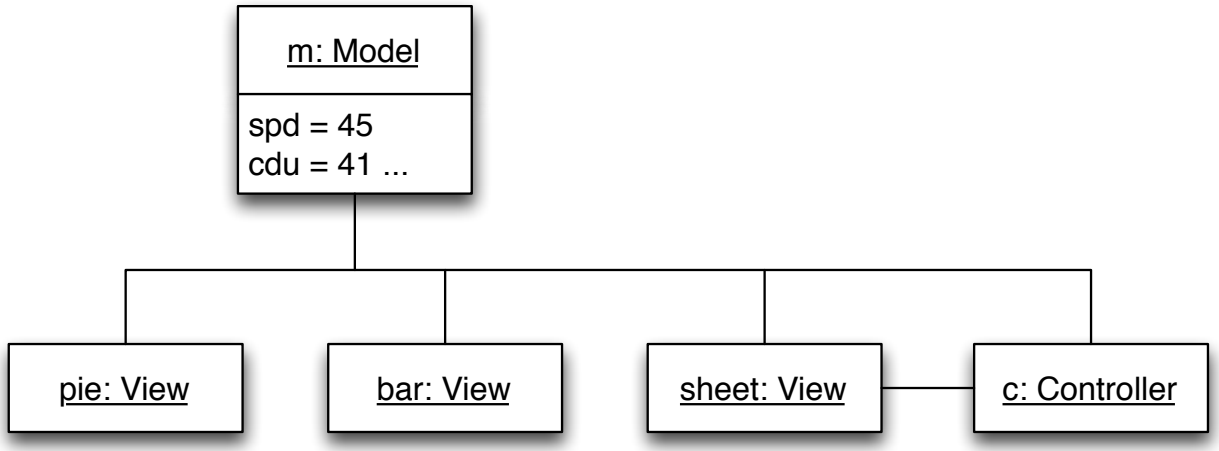
Structure



© -Alessio Gambi - Saarland University

Structure (2)

Each model can *register* multiple *observers* (= views and controllers).



As soon as the model changes, all registered observers are *notified*, and they update themselves accordingly.

© -Alessio Gambi - Saarland University

Participants

The **model** encapsulates core data and functionality; it is independent of any concrete output representation, or input behavior.

Model	*collaboration with*
responsible for	View , Controller
•Core functionality	
•Registering dependent views and controls	
•Notifying registered components on data changes	

© -Alessio Gambi - Saarland University

Participants (2)

The **view** displays information to the user. A model can have multiple views.

View	*collaboration with*
responsible for • Showing information to the user • Possibly creating the appropriate Control • Reading data from Model	Controller, Model

© -Alessio Gambi - Saarland University

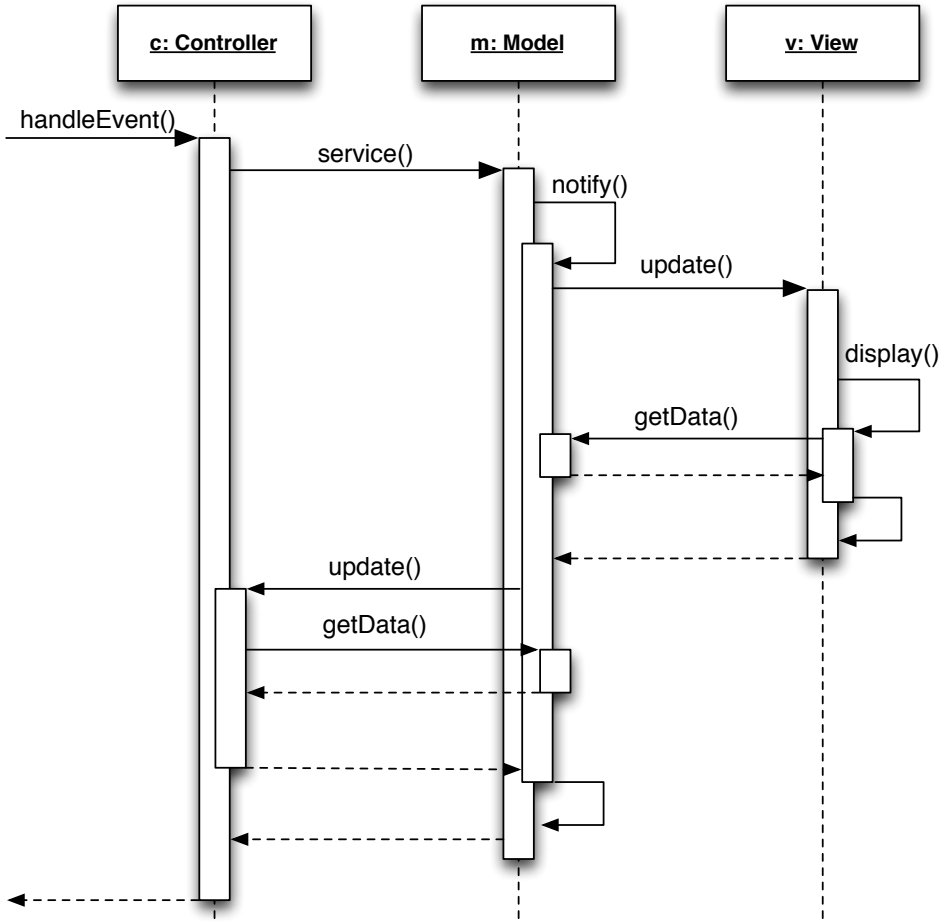
Participants (3)

The controller processes input and invokes the appropriate services of the view or the model; every controller is assigned to a single view; a model can have multiple controllers.

Controller	*collaboration with*
responsible for • Accepting user input • Matching inputs to service invocations (displaying services of the View, or services of the Model)	View, Model

© -Alessio Gambi - Saarland University

Dynamic behavior



© -Alessio Gambi - Saarland University

Consequences of the Model-View-Controller Pattern

Benefits

- multiple views of the same system
- synchronous views
- attachable views and controllers

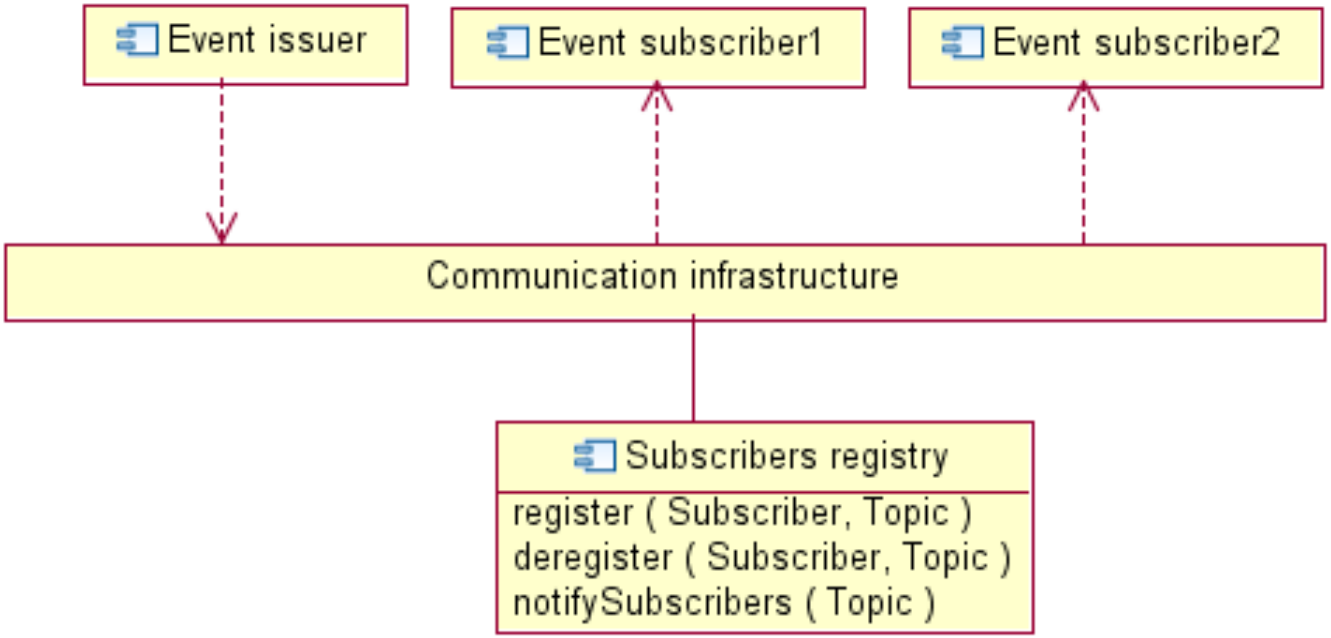
Drawbacks

- increased complexity
- strong coupling between Model and View
- Strong coupling between Model and Controllers (can be avoided by means of the command pattern)

Known applications: GUI libraries, Smalltalk, Microsoft Foundation Classes

© -Alessio Gambi - Saarland University

Publish-subscribe



© Philippe Dugerdil, HEG

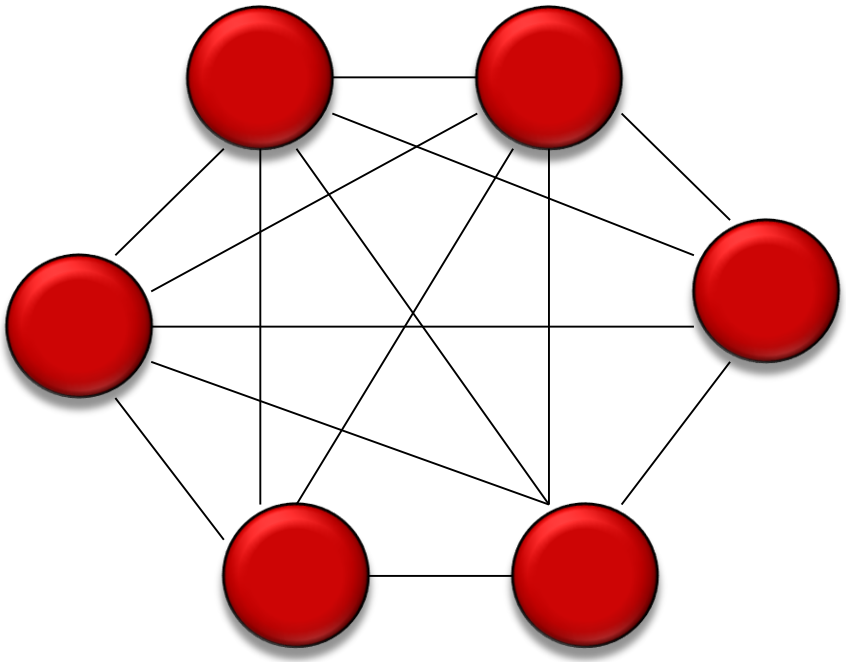
Implemented tactics

Modifiability tactics

- Restrict communication paths
- Use an intermediary
- Standardize collaboration

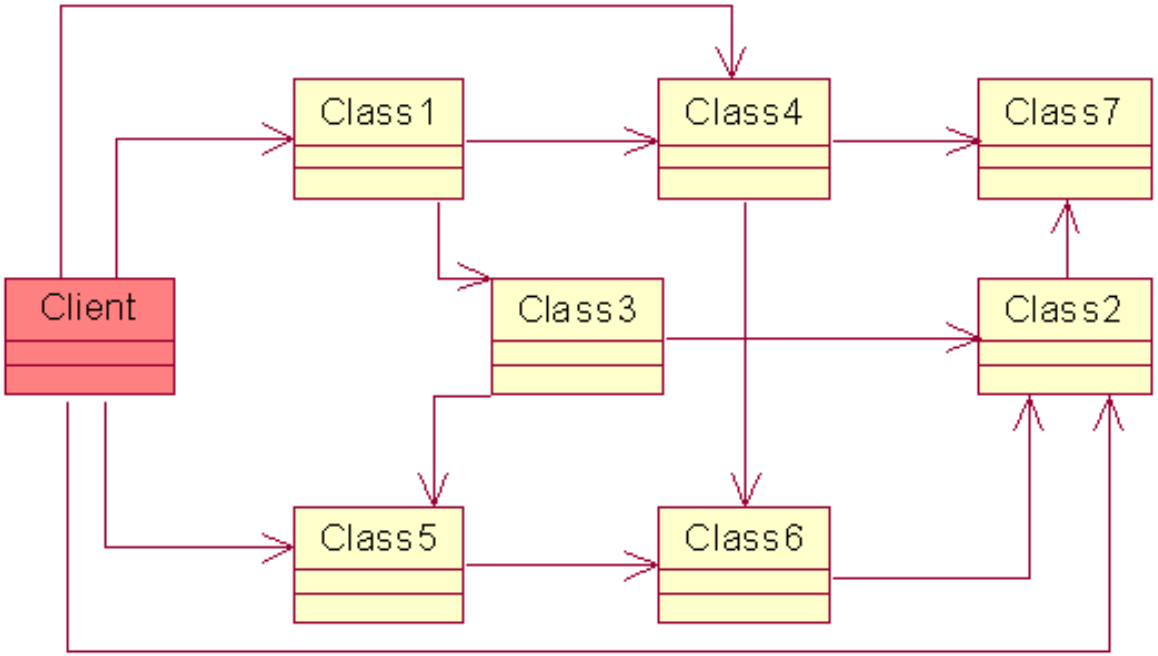
© Philippe Dugerdil, HEG

Problem: point to point communication among components



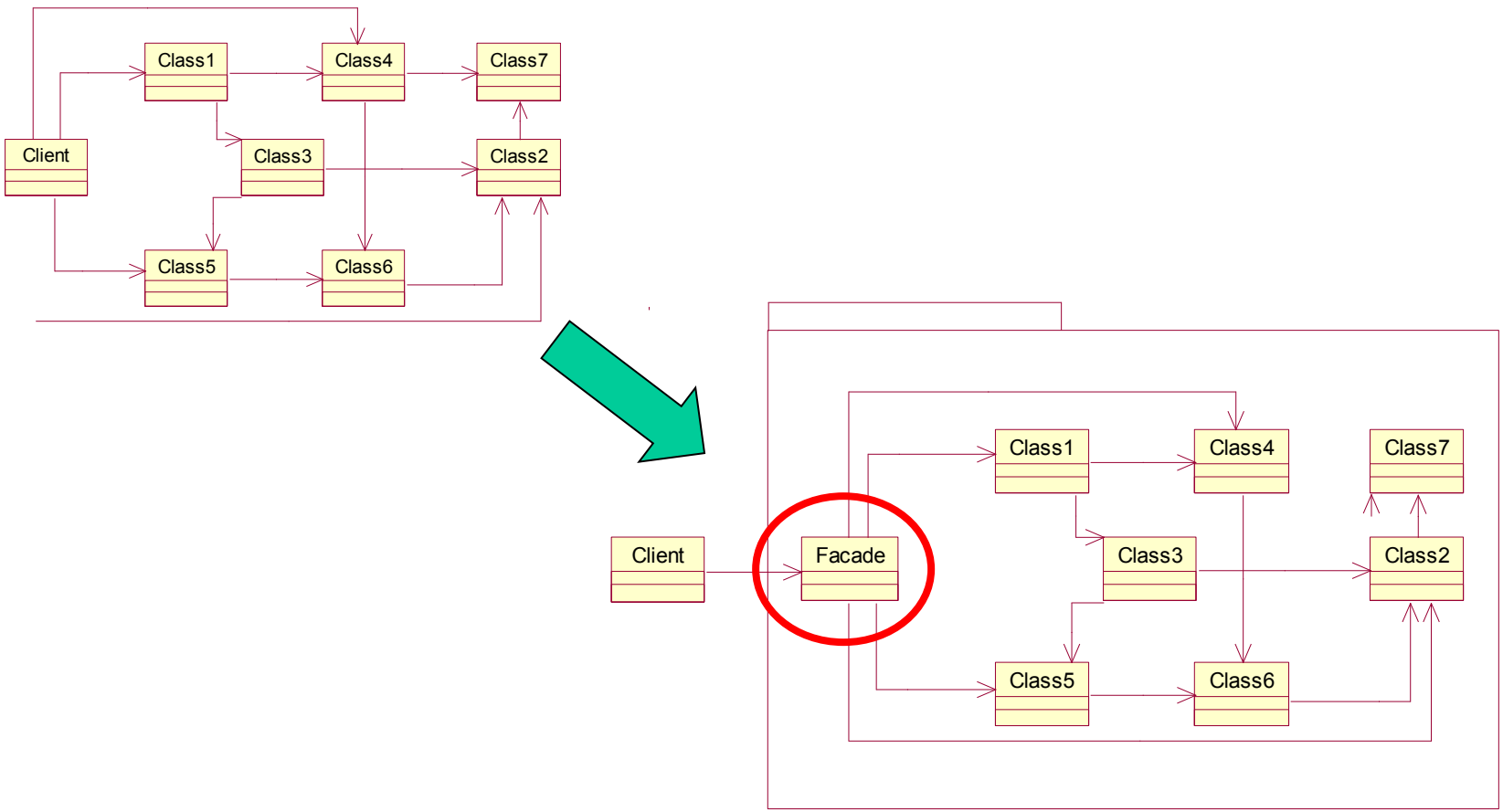
© Philippe Dugerdil, HEG

Problem: too many communications paths



© Philippe Dugerdil, HEG

Facade



© Philippe Dugerdil, HEG

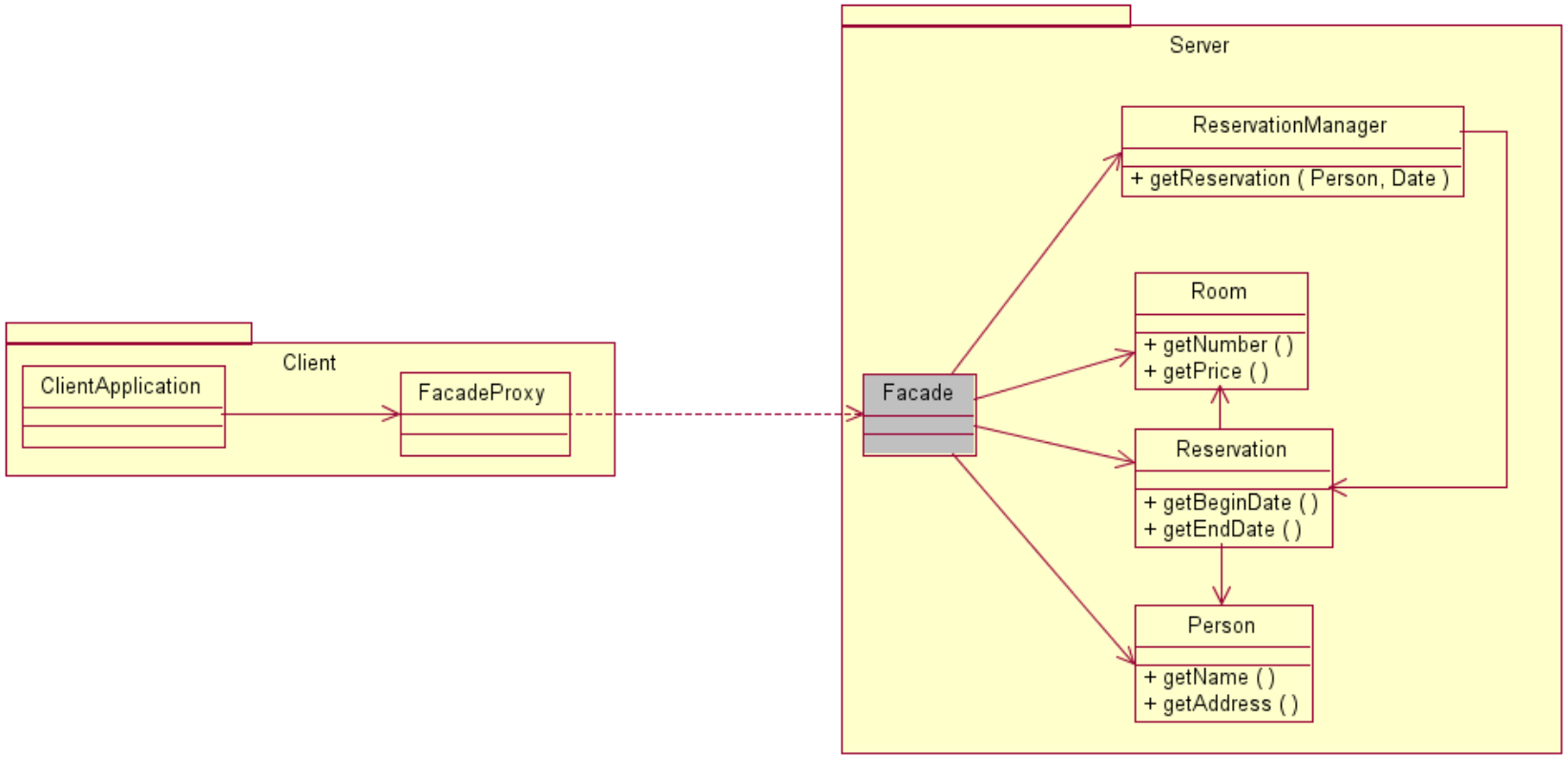
Implemented tactics

Modifiability tactics

- Hide information
- Restrict communication paths
- Use an intermediary

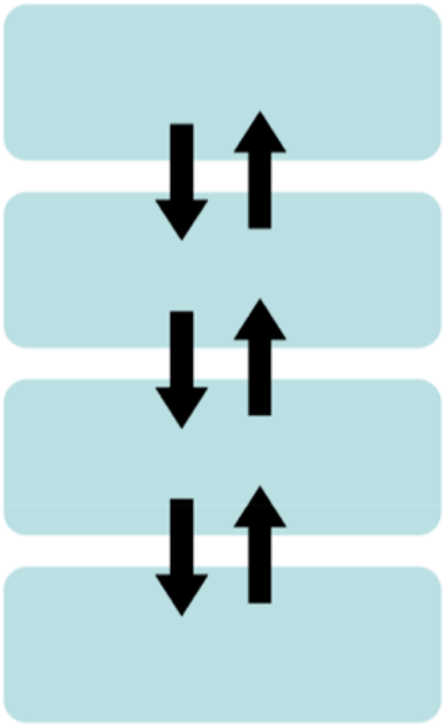
© Philippe Dugerdil, HEG

New architecture: façade & proxy

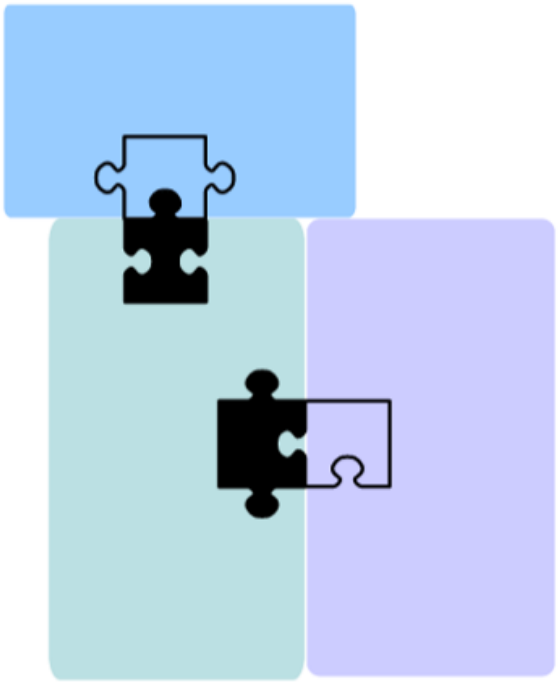


© Philippe Dugerdil, HEG

Layered



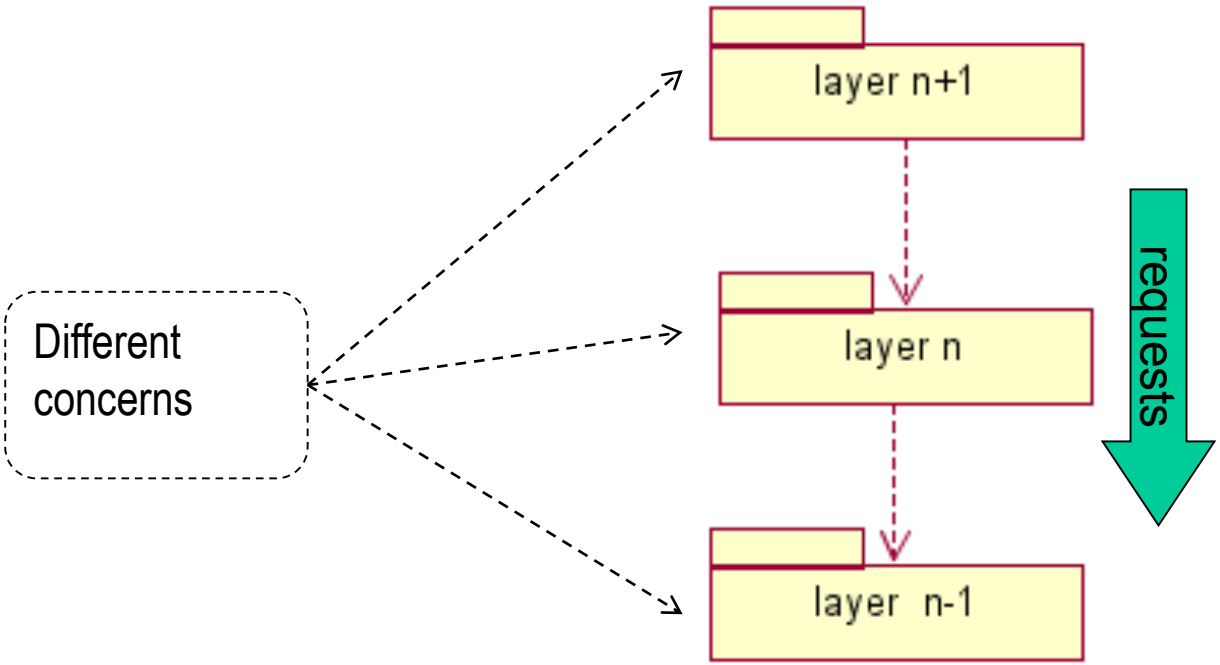
Plug-in



© -Alessio Gambi - Saarland University

h e g

Layers



© Philippe Dugerdil, HEG

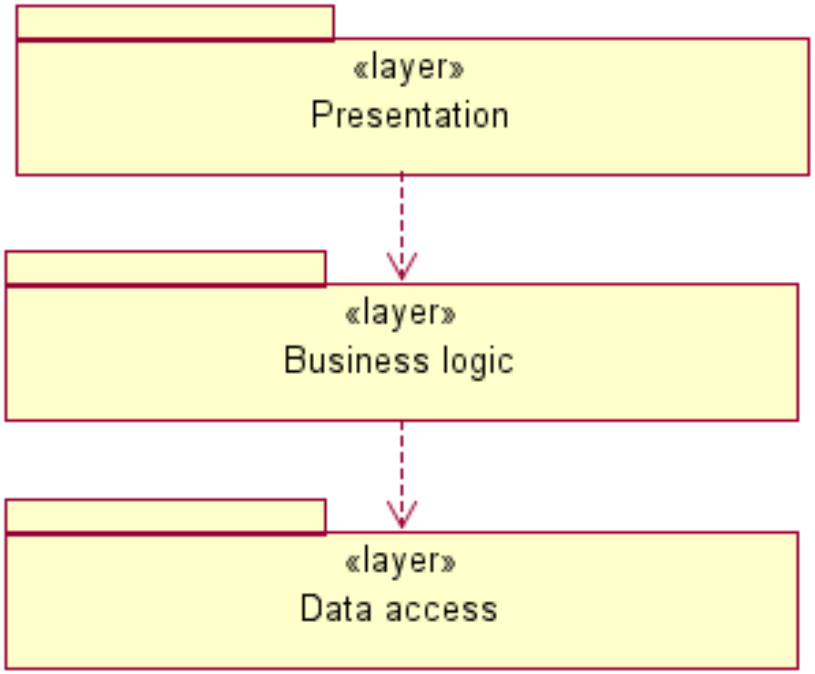
Choosing the layers

- Define the abstraction criterion (concern) each layer provides (targets)
  - Potential changes
  - Reusability opportunities
  - Category of responsibilities
- Choose the number of layers
  - Abstraction levels / number of categories
  - Keep the number of layer small
    - Avoid the performance overhead
- Specify the interface for each layer
  - Each layer should represent a black box for the layer above

© Philippe Dugerdil, HEG

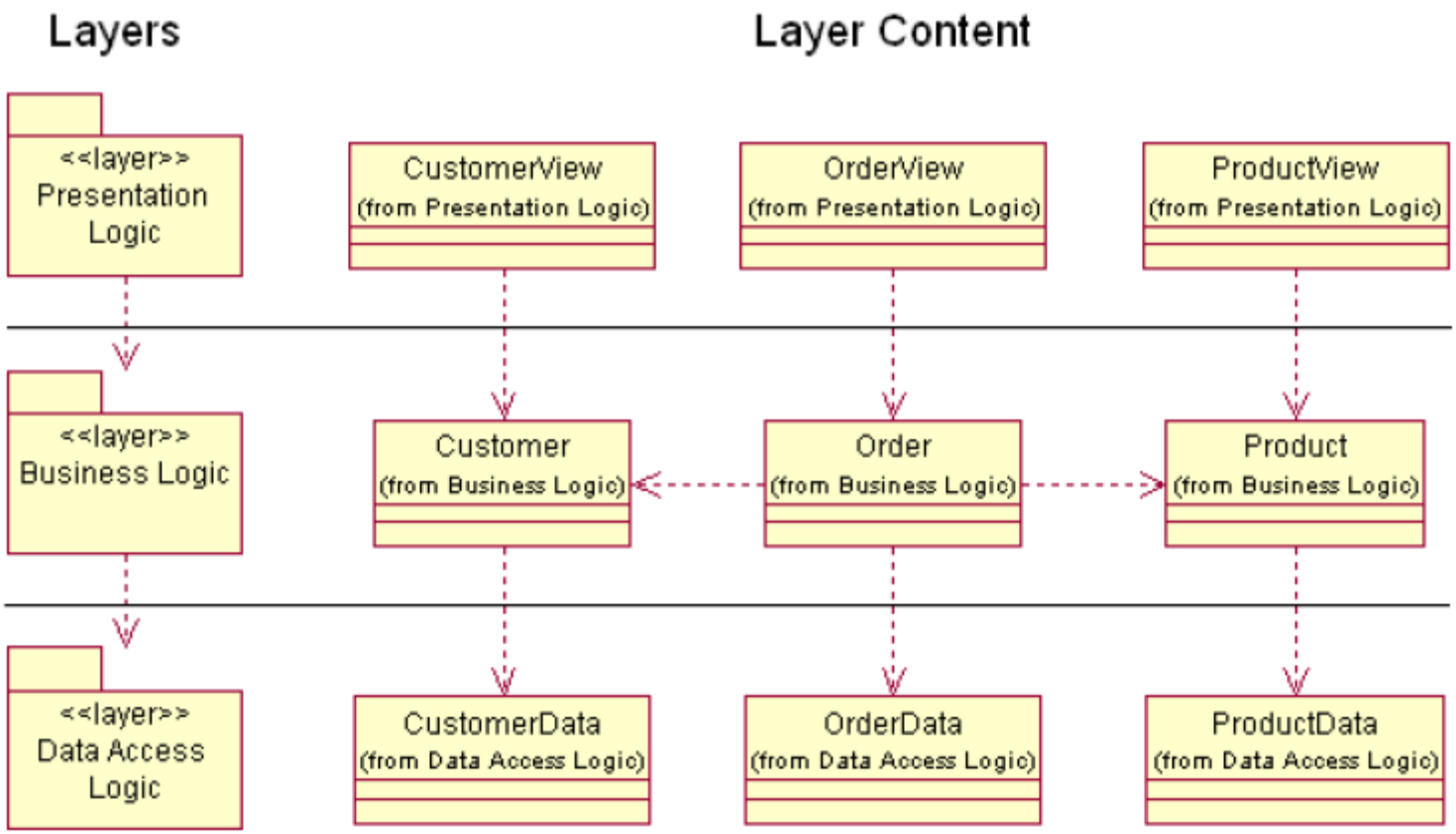
Change in one of the concerns: display, data access

Responsibility-based structure



© Philippe Dugerdil, HEG

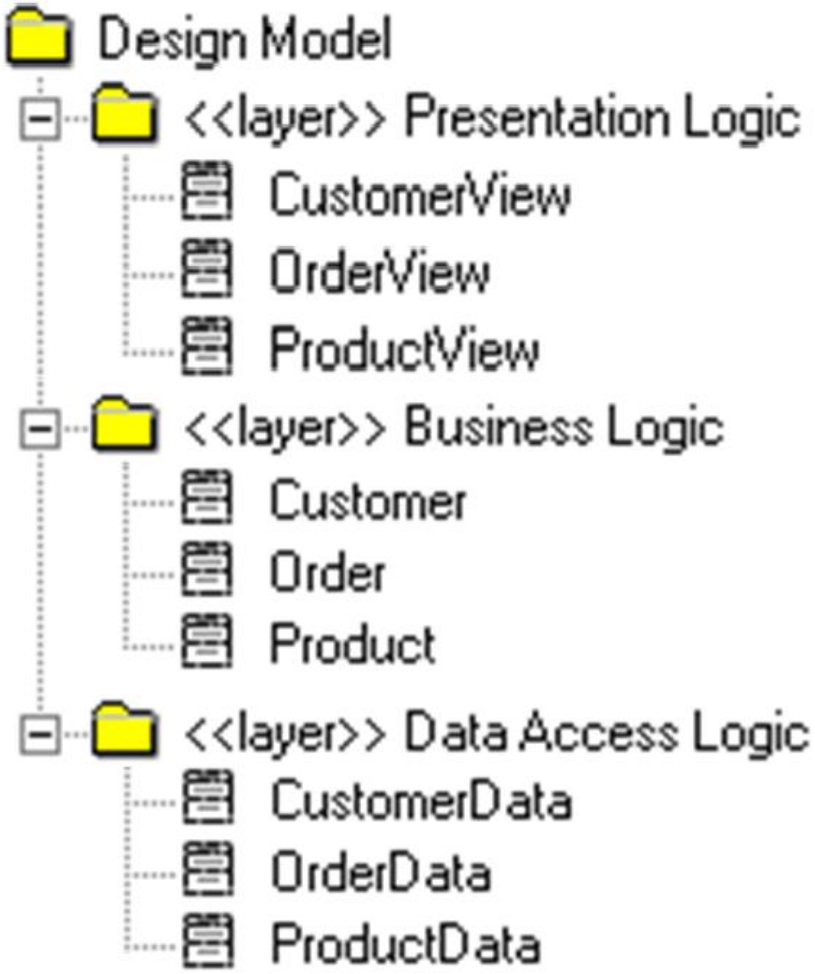
Exemple



[Source: Peter Eeles - Layering Strategies, Rational Software White Paper TP 199, 08/01, 2002]

© Philippe Dugerdil, HEG

Representation of the example as Java packages / folders



© Philippe Dugerdil, HEG

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

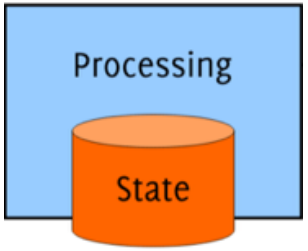
© -Alessio Gambi - Saarland University

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*

© -Alessio Gambi - Saarland University

Components

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

Objects

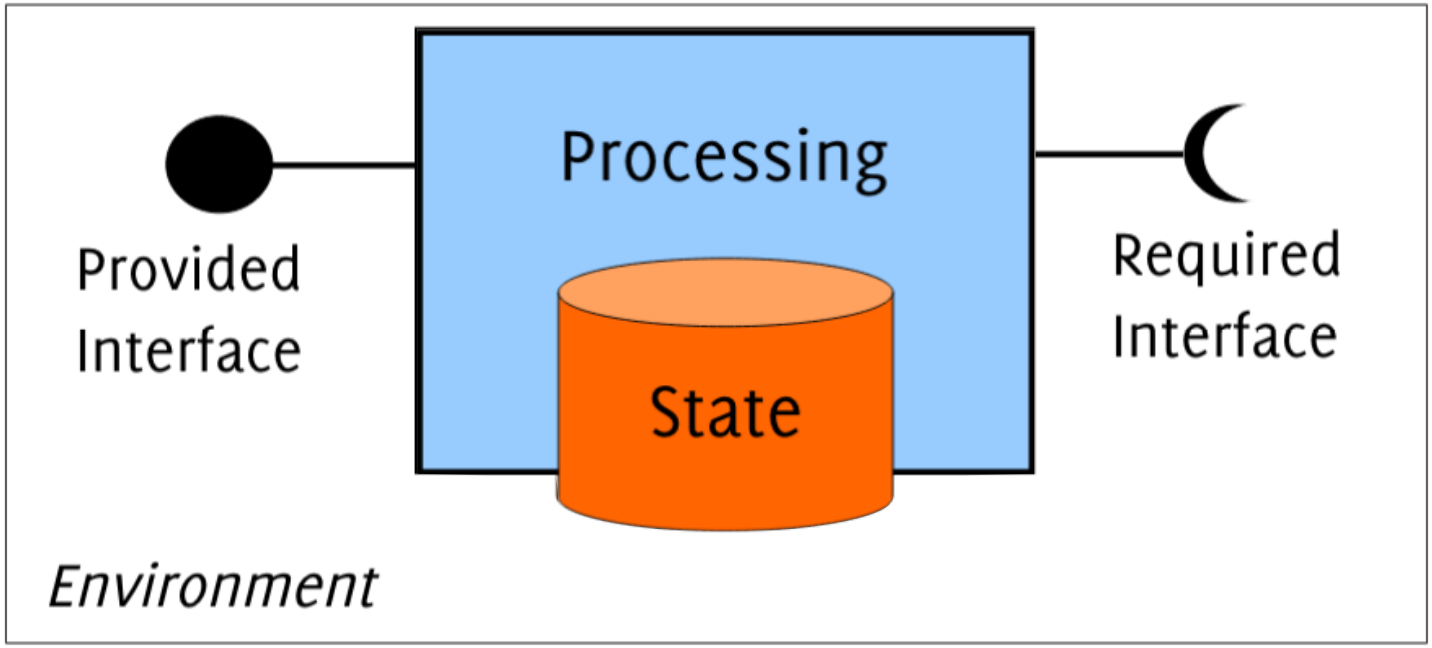
- Encapsulate state and functionality
- Fine-grained
- Can “move” across components
- Identifiable unit of instantiation

Modules

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

© -Alessio Gambi - Saarland University

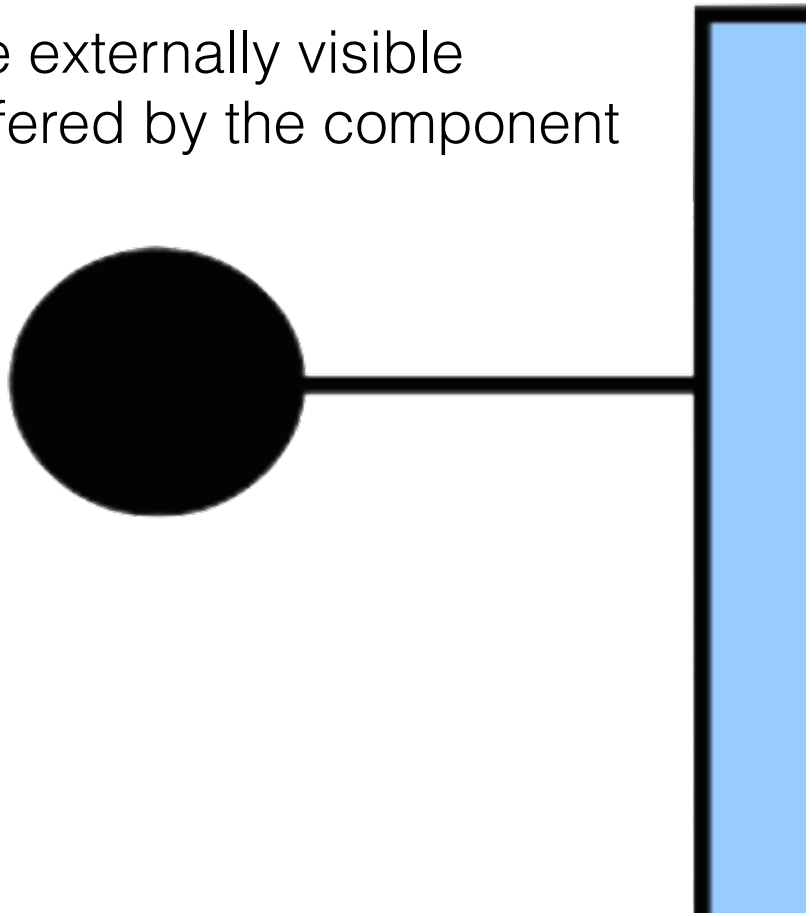
Component Interfaces



© -Alessio Gambi - Saarland University

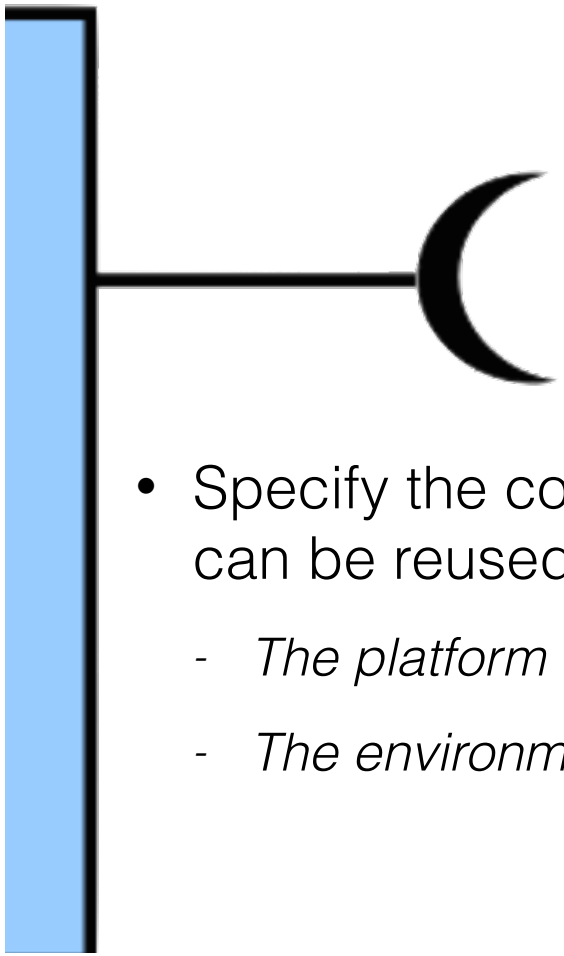
Provided Interfaces

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



© -Alessio Gambi - Saarland University

Required Interface

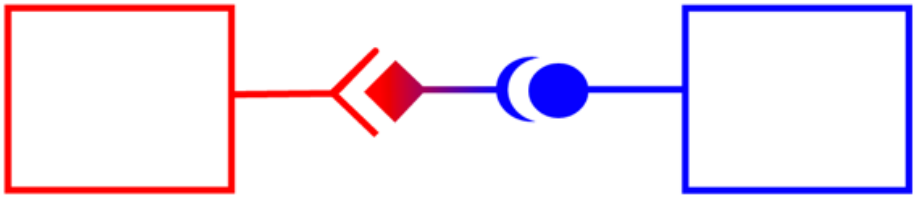


- Specify the conditions upon which a component can be reused
  - *The platform is compatible*
  - *The environment is setup correctly*

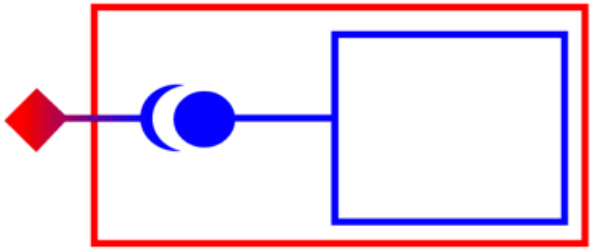
© -Alessio Gambi - Saarland University

Compatible Interfaces

Component interfaces must match perfectly to be connected



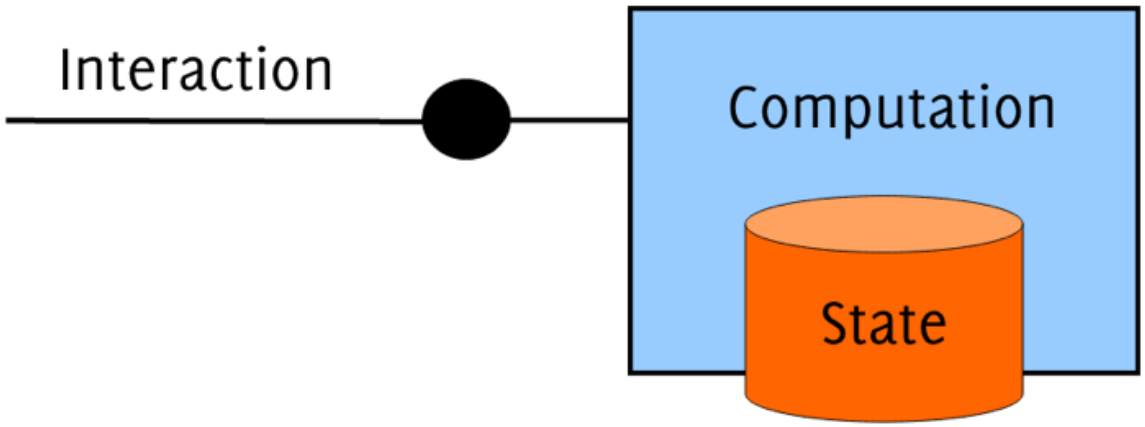
Adapter



Wrapper

© -Alessio Gambi - Saarland University

Software Connectors



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

© -Alessio Gambi - Saarland University

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

© -Alessio Gambi - Saarland University

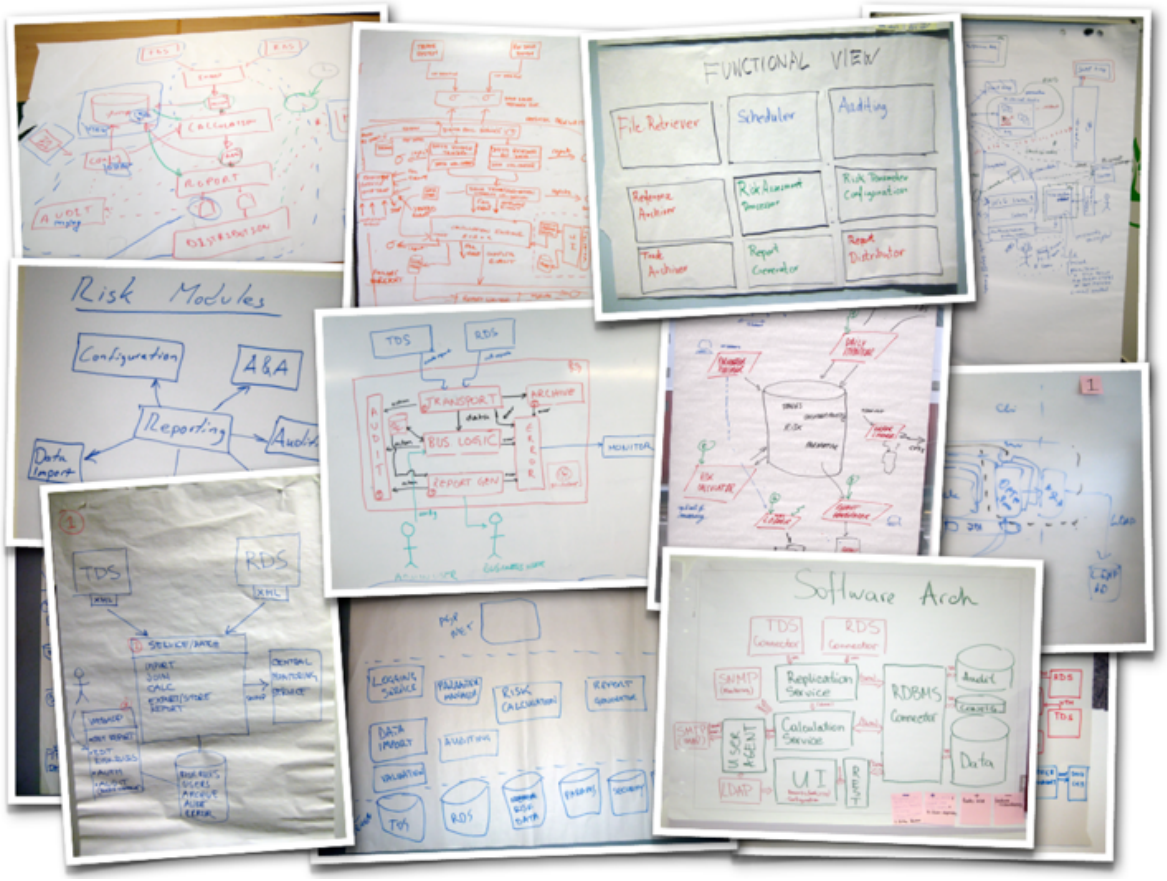
Connectors, not Components!

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

© -Alessio Gambi - Saarland University

Views and Viewpoints



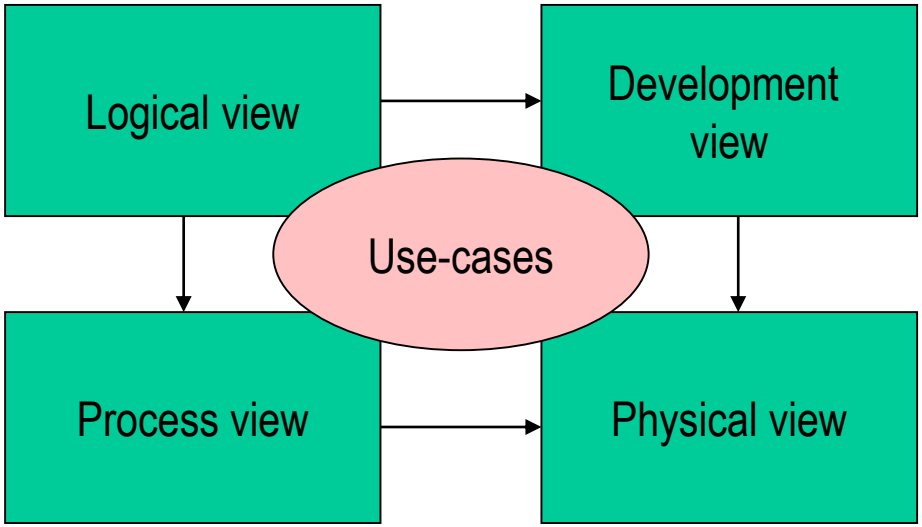
© -Alessio Gambi - Saarland University

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

© -Alessio Gambi - Saarland University

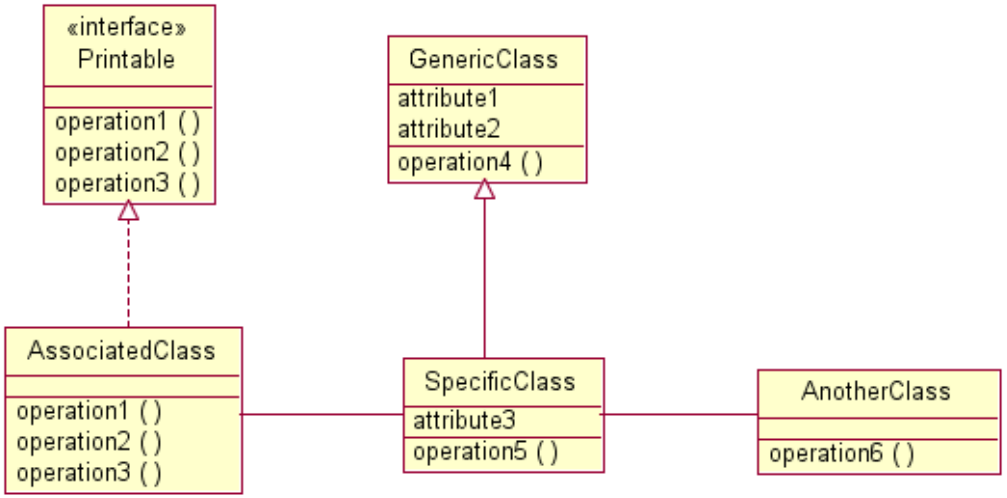
Historical model: Kruchten's 4+1 views (RUP)

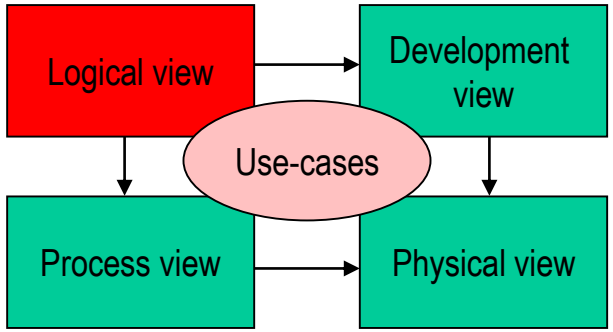


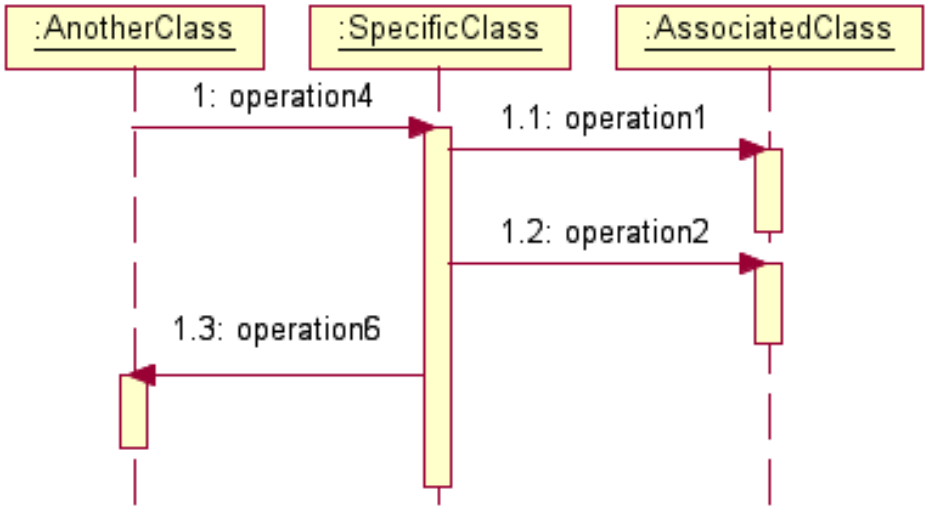
[Source: Kruchten Ph.- The 4+1 View Model of Architecture. IEEE Software 12(6), Nov. 1995]

© Philippe Dugerdil, HEG

Logical view

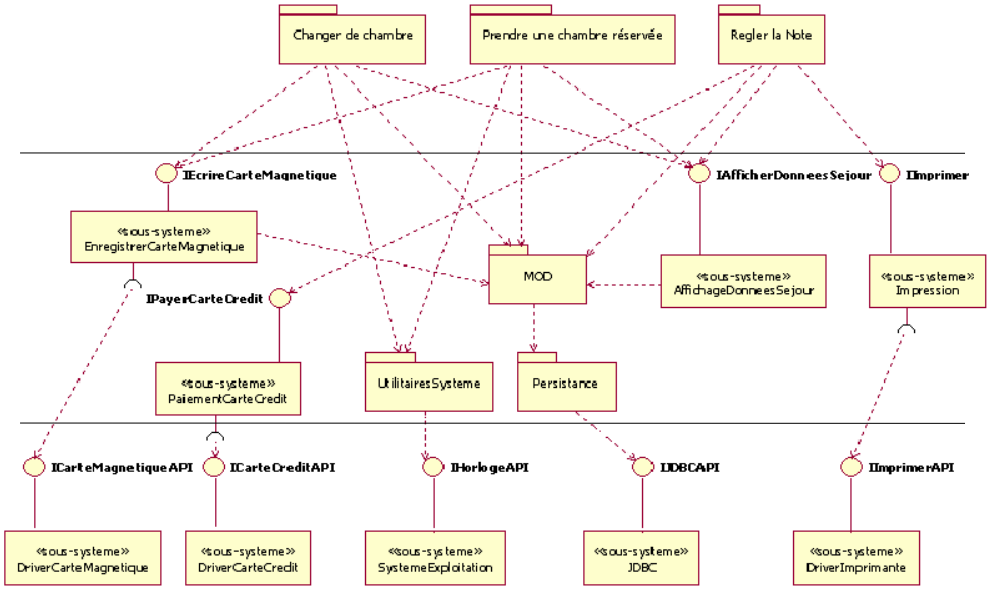


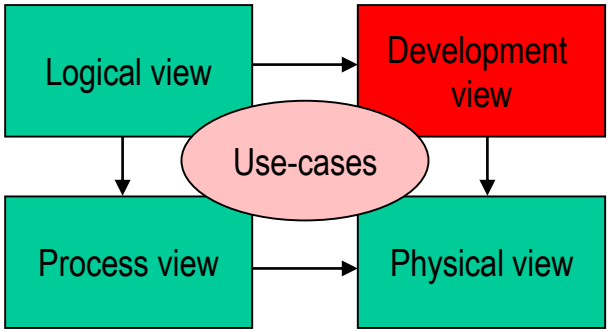


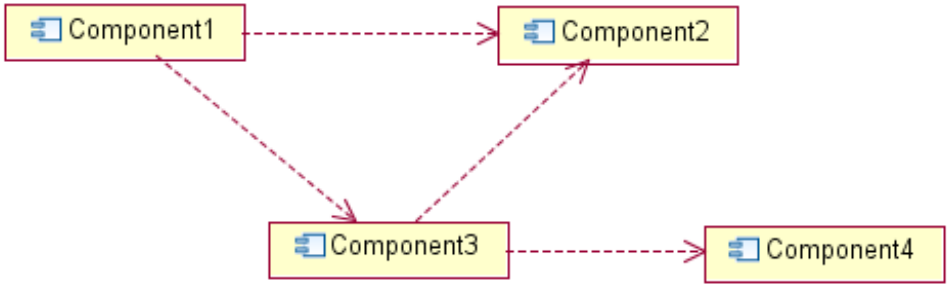


Classes, sequence diagrams, state diagrams, activity diagrams…

Development view



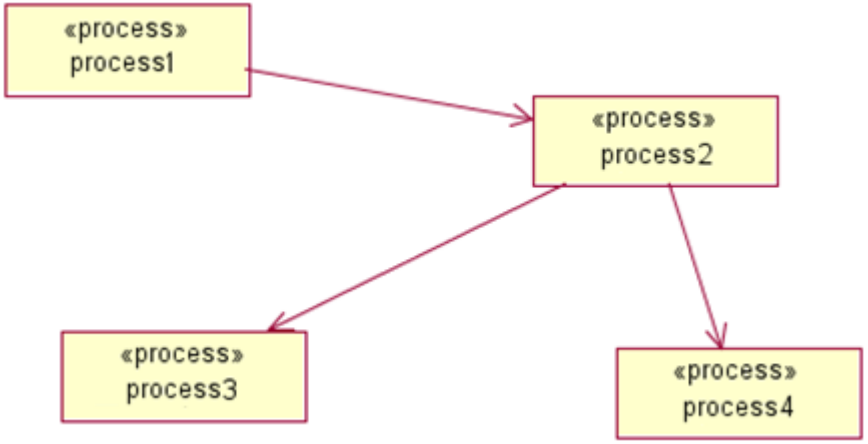


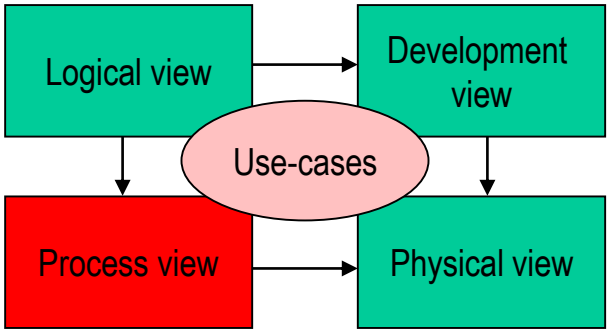


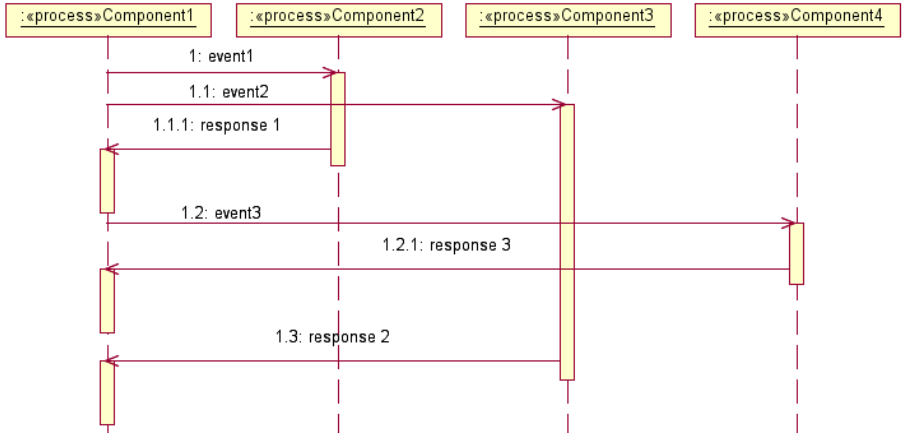
Components, packages, layers, subsystems.

© Philippe Dugerdil, HEG

Process view

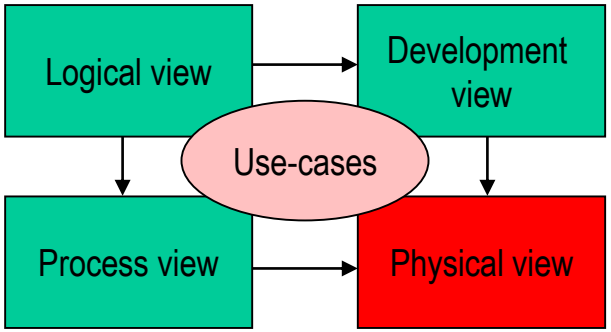


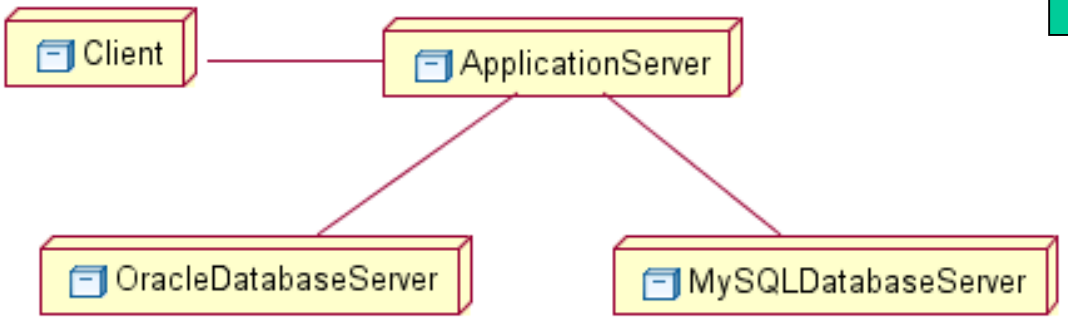




Process, communications, synchronisations, …

Physical view





Physical files, machines, communication channels, protocols,…

© Philippe Dugerdil, HEG

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

© -Alessio Gambi - Saarland University

Music Player Scenarios

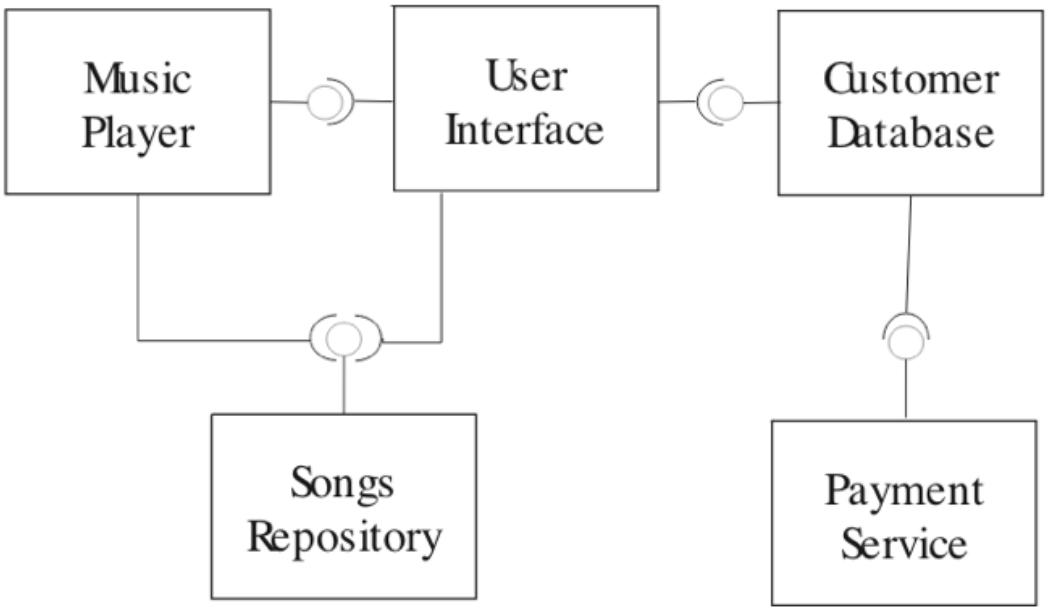
- Browse for new songs
- Pay to hear the entire song
- Download the purchased song on the phone
- Play the song

© -Alessio Gambi - Saarland University

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

© -Alessio Gambi - Saarland University



© -Alessio Gambi - Saarland University

Process View

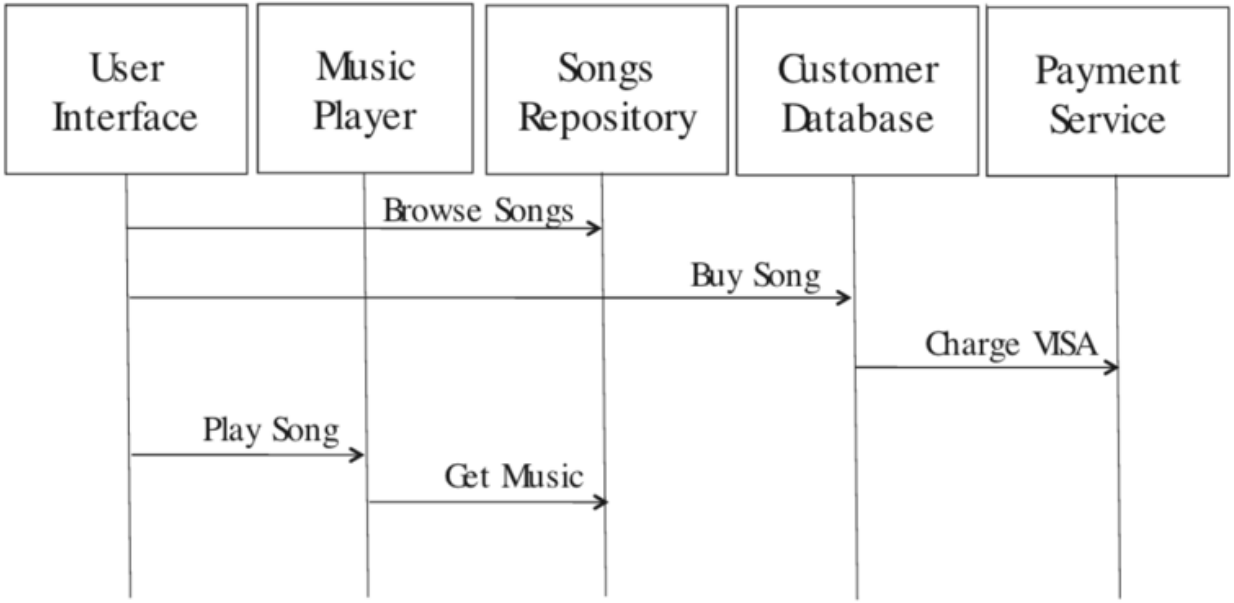
- Model the dynamic aspects of the architecture and the behavior its parts
  - *active components*
  - *concurrent threads*

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

    - **Concern**: Functionality, Performance

    - **Target Audience**: Developers

© -Alessio Gambi - Saarland University



Use Cases: Browse, Pay and Play For Songs

© -Alessio Gambi - Saarland University

Development View

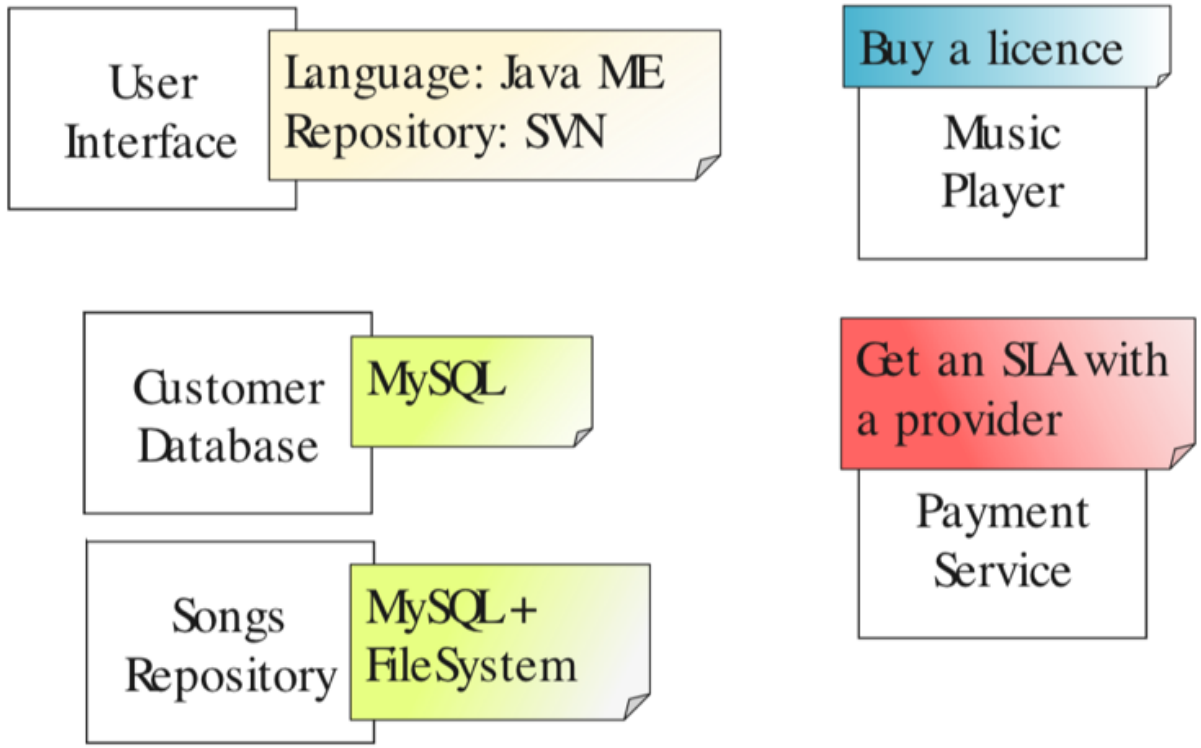
- Static organization of the software code artifacts
  - *packages*
  - *modules*
  - *binaries*

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

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

    - **Target Audience**: Developers

© -Alessio Gambi - Saarland University



© -Alessio Gambi - Saarland University

Physical View

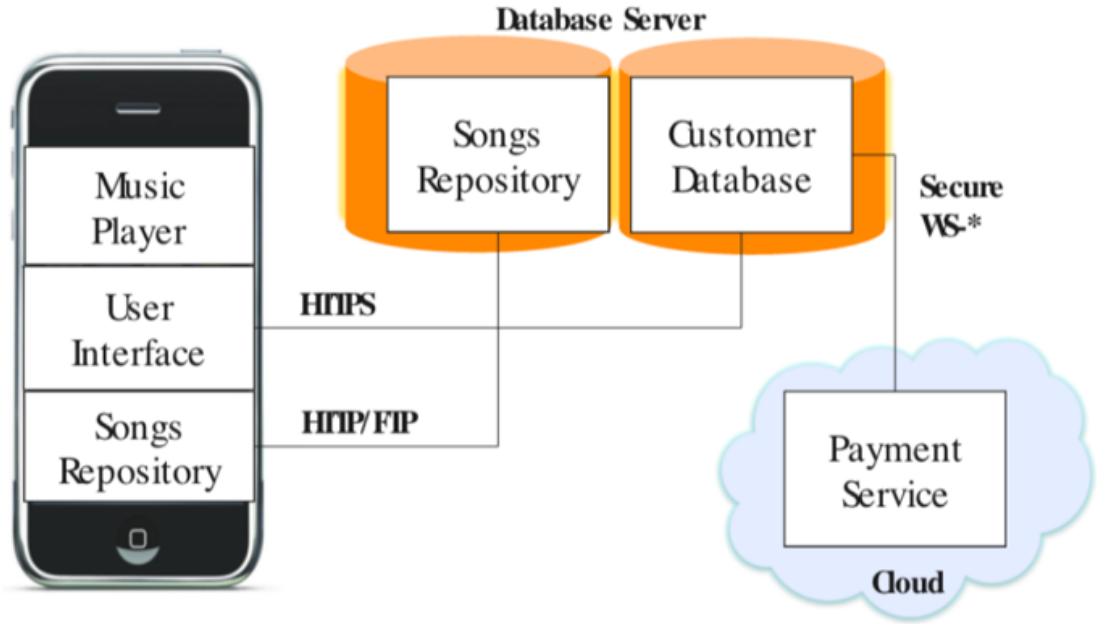
- Hardware environment where the software will be deployed
  - *hosts*
  - *networks*
  - *storage*

- Mapping between logical and physical entities

  - **Concern**: Quality attributes

  - **Target Audience**: Operations

© -Alessio Gambi - Saarland University



© -Alessio Gambi - Saarland University

Reuse and Libraries

Is it possible to reuse existing classes?

Possibly adapter classes are needed.

© Andreas Zeller - Saarland University