Génie logiciel

Philippe Dugerdil

07.11.2019

Architecting systems

Tactics and patterns

Designing high level architecture based on NFR

2 concepts:

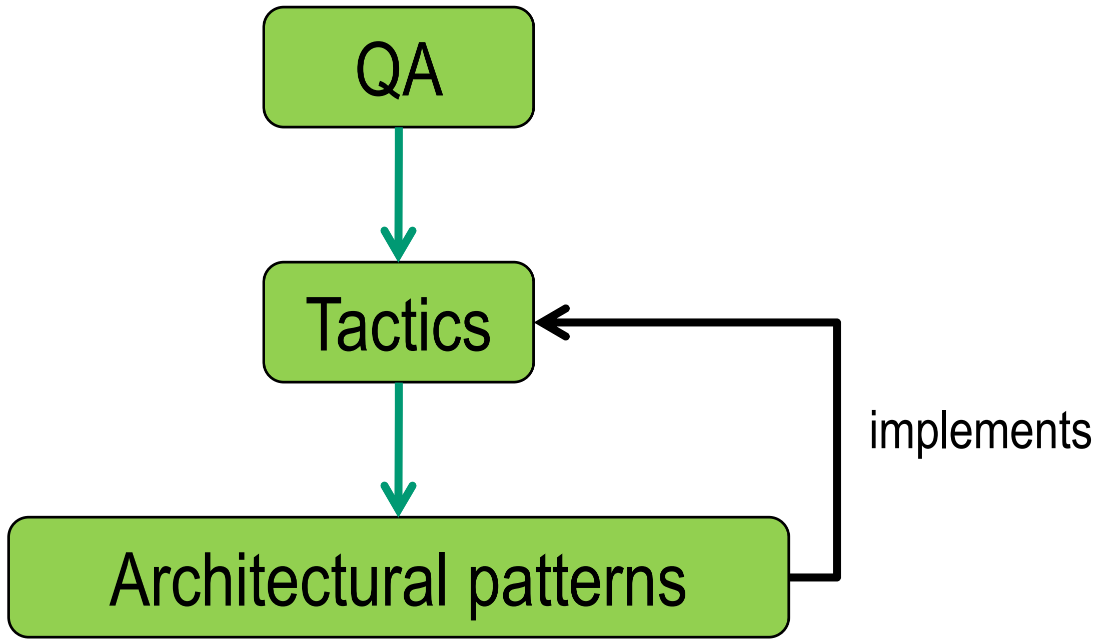
- Architectural tactic

  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.

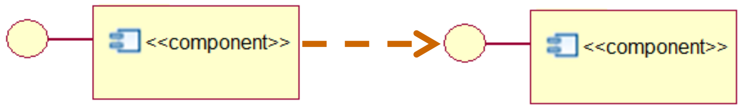
Designing the high level architecture



Why is modifiability an issue ? Dependencies

5 main sources of dependencies among 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



Modifiability tactics

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

QA: performance

- How long does it take for the system to respond to an event (latency) ?
- Source of performance problems
  - Availability of required resources

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)

Availability tactics

- Fault : error in the system.
- Failure : the system no longer delivers a service consistent with its specification.
- A fault becomes a failure when it impacts the service.

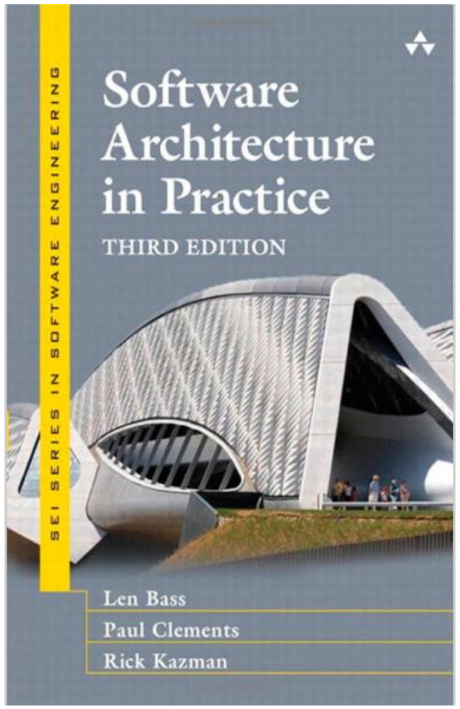
- Availability : avoiding system failures
  - Detect and correct a fault before it becomes a failure

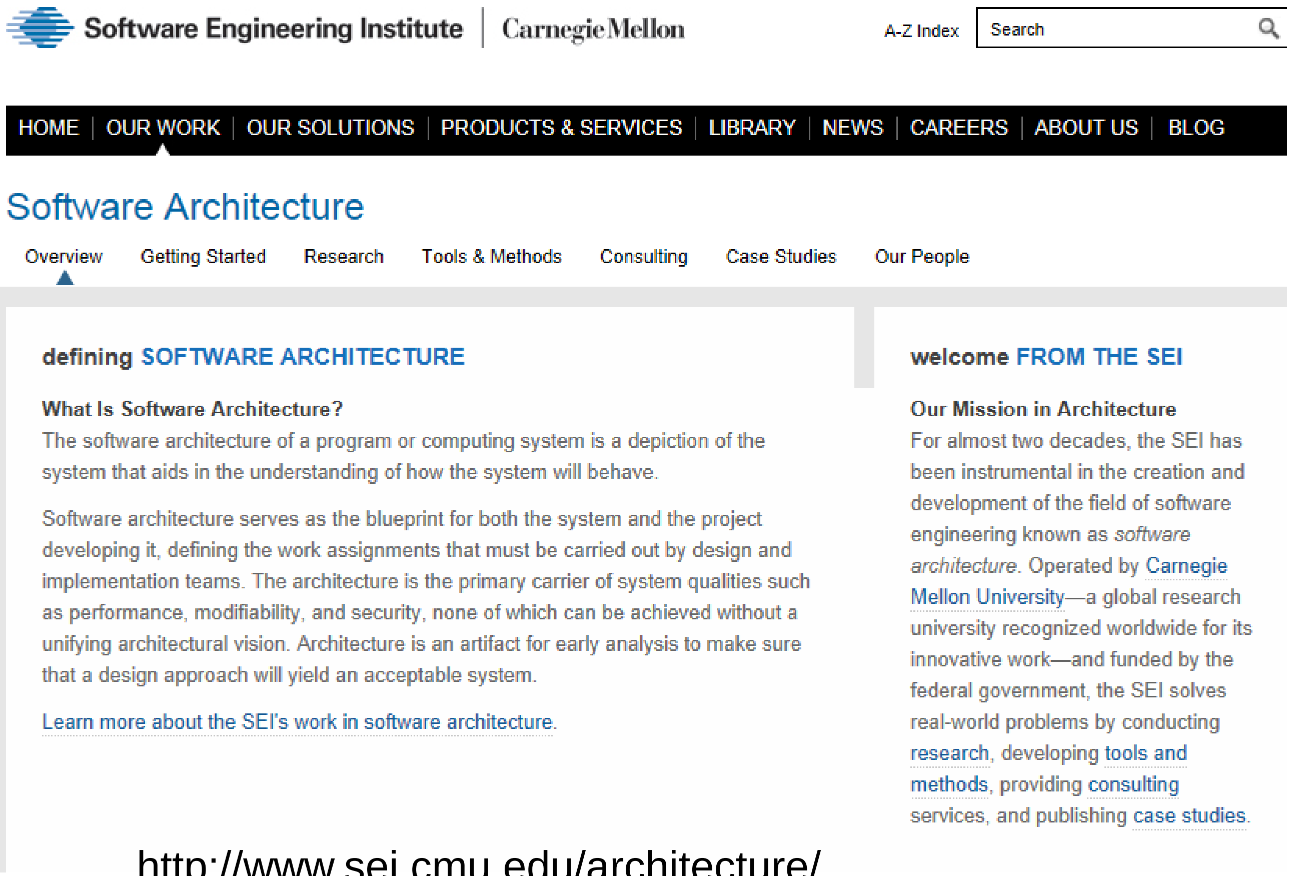
Availability tactics

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

$\left.\begin{matrix} \\ \\ \\ \\ \end{matrix}\right\}$ to avoid failure

Ressources for Tactics


SEI SERIES IN SOFTWARE ENGINEERING
Software Architecture in Practice
THIRD EDITION
Len Bass
Paul Clements
Rick Kazman


Software Engineering Institute | Carnegie Mellon
A-Z Index
Search
HOME | OUR WORK | OUR SOLUTIONS | PRODUCTS & SERVICES | LIBRARY | NEWS | CAREERS | ABOUT US | BLOG
Overview
Getting Started
Research
Tools & Methods
Consulting
Case Studies
Our People

Software Architecture

defining SOFTWARE ARCHITECTURE

What Is Software Architecture?

The software architecture of a program or computing system is a depiction of the system that aids in the understanding of how the system will behave.

Software architecture serves as the blueprint for both the system and the project developing it, defining the work assignments that must be carried out by design and implementation teams. The architecture is the primary carrier of system qualities such as performance, modifiability, and security, none of which can be achieved without a unifying architectural vision. Architecture is an artifact for early analysis to make sure that a design approach will yield an acceptable system.

Learn more about the SEI's work in software architecture.

welcome FROM THE SEI

Our Mission in Architecture

For almost two decades, the SEI has been instrumental in the creation and development of the field of software engineering known as *software architecture*. Operated by Carnegie Mellon University—a global research university recognized worldwide for its innovative work—and funded by the federal government, the SEI solves real-world problems by conducting research, developing tools and methods, providing consulting services, and publishing case studies.

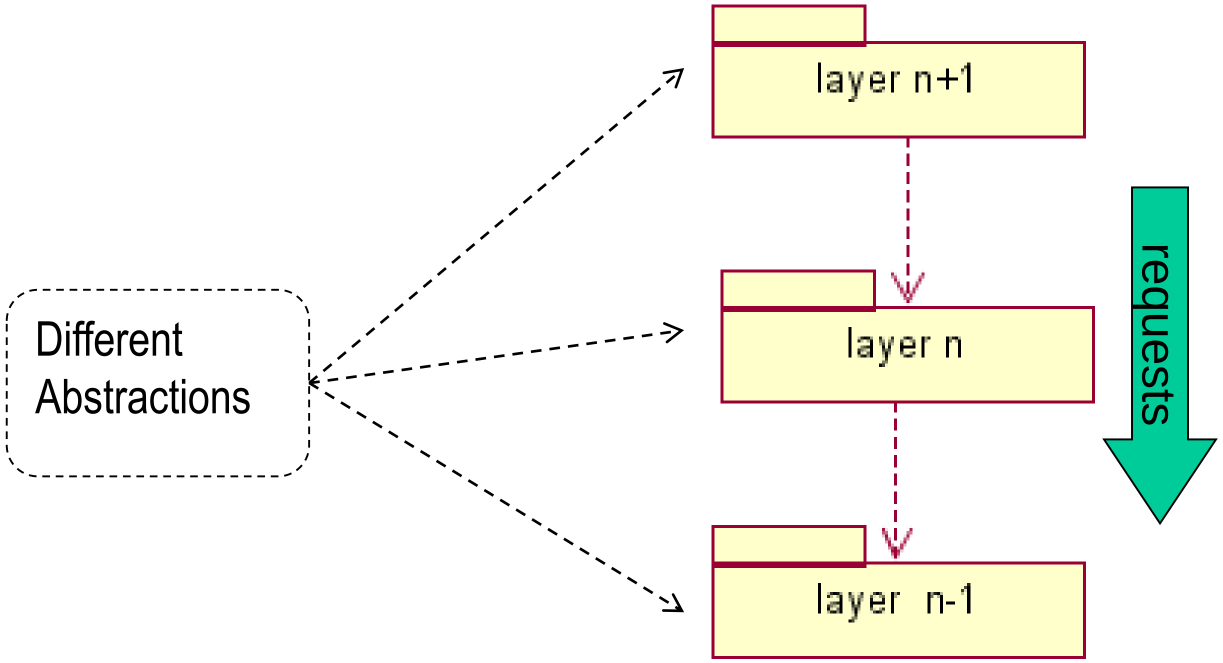
http://www.sei.cmu.edu/architecture/

Patterns

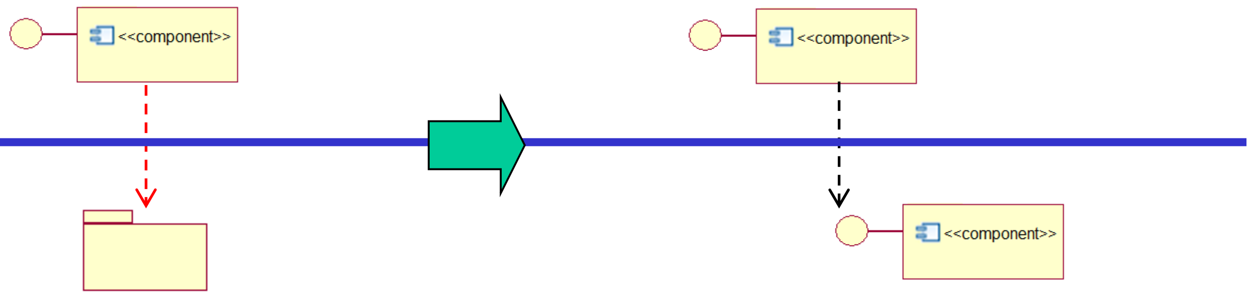
Problem

- Structuring a complex system with levels of abstraction

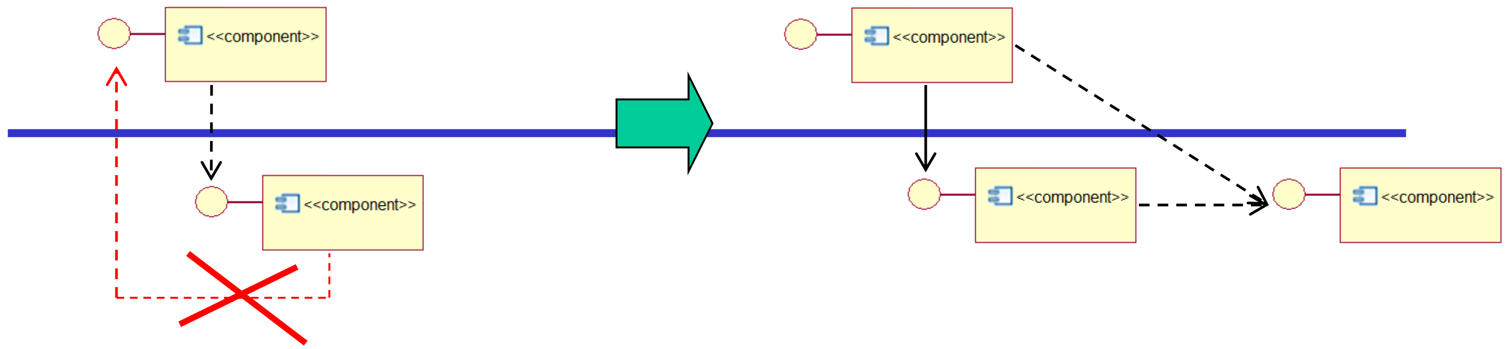
Layers



Layers' dependency rules



Access through interfaces



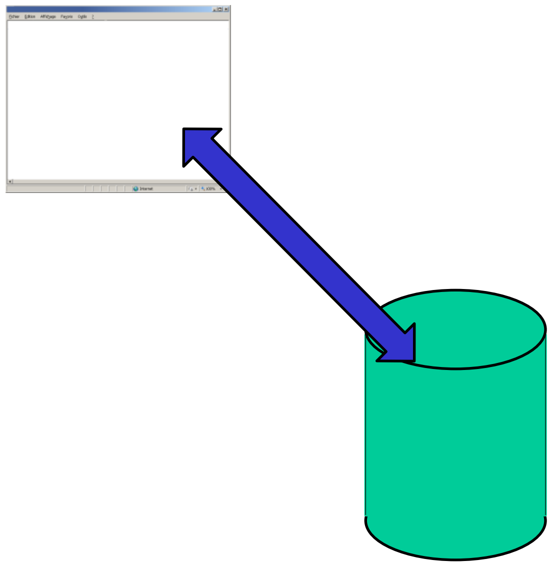
Downward requests only

Implemented tactics

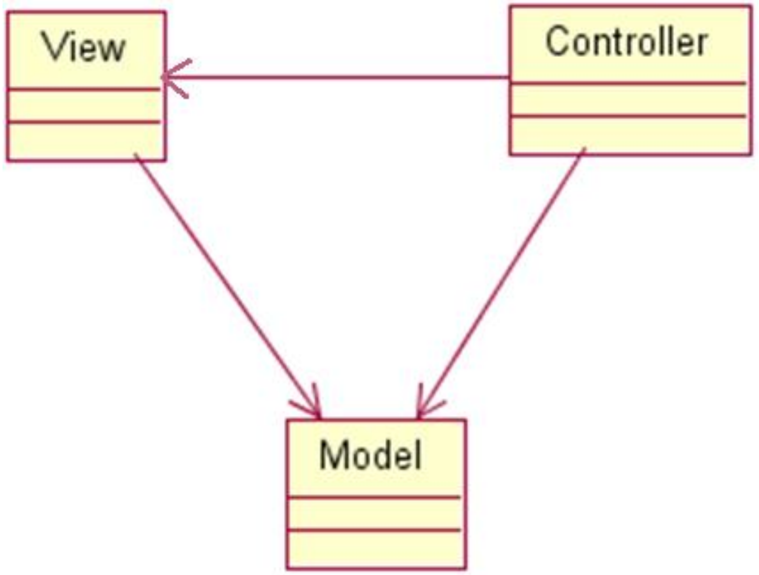
Modifiability tactics:

- Hide information
- Maintain semantic coherence (abstraction at layer level)

Problem: mixing up responsibilities



Model-View-Controller



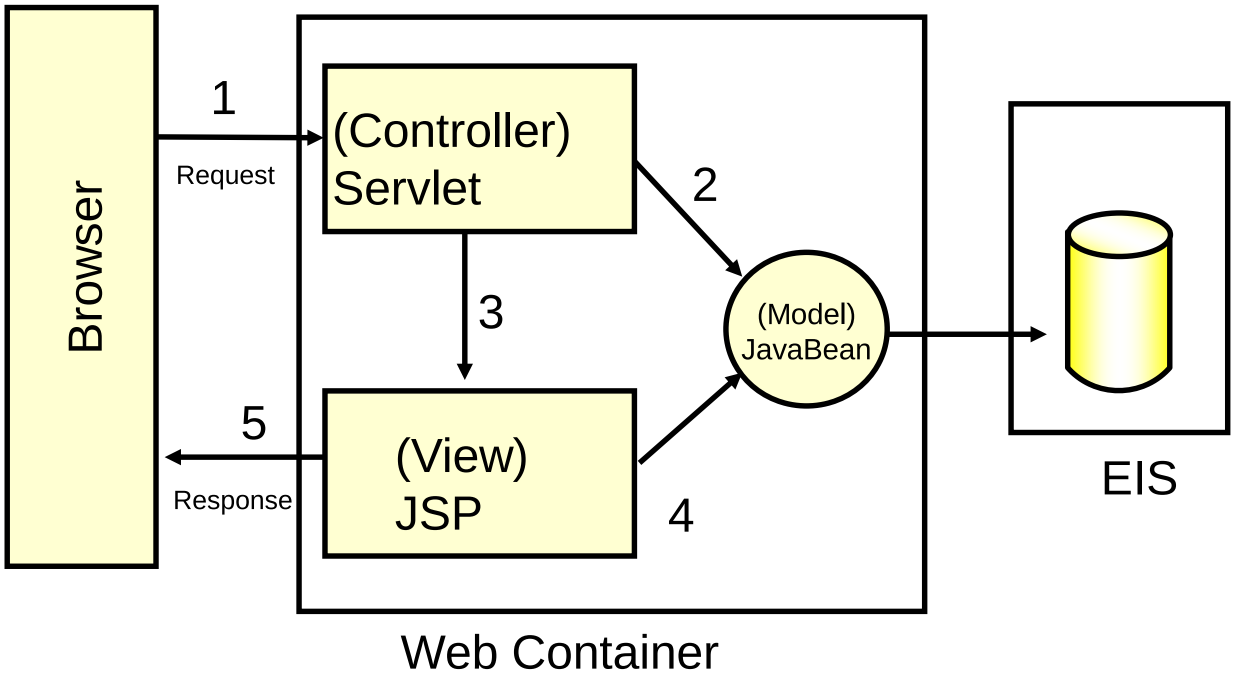
Concept designed back in the 70's in Alan's Kay at Xerox. Implemented in Smalltalk 76, Smalltalk 78 then in the first commercial version: Smalltalk-80 by Adele Goldberg's group.

Implemented tactics

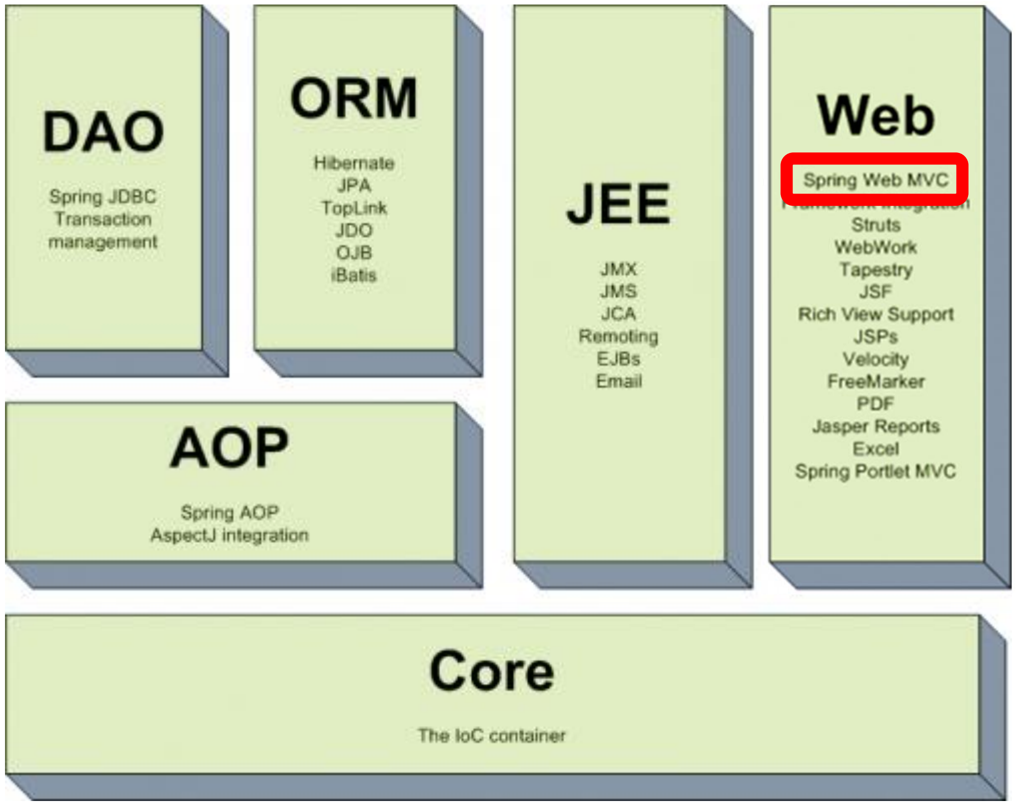
Modifiability tactics:

- Anticipate expected change
- Separate concerns

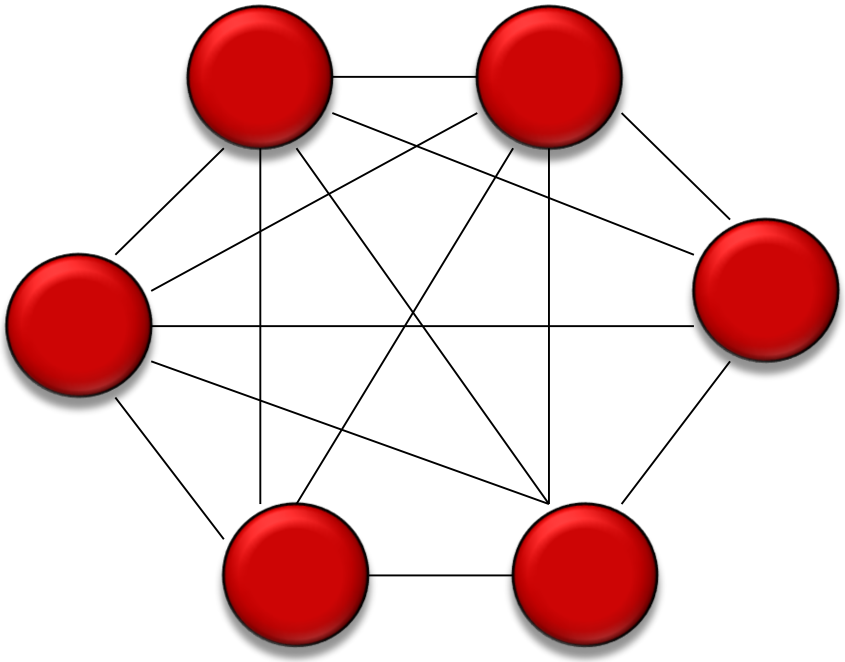
Exemple: good-old JSP / Servlet



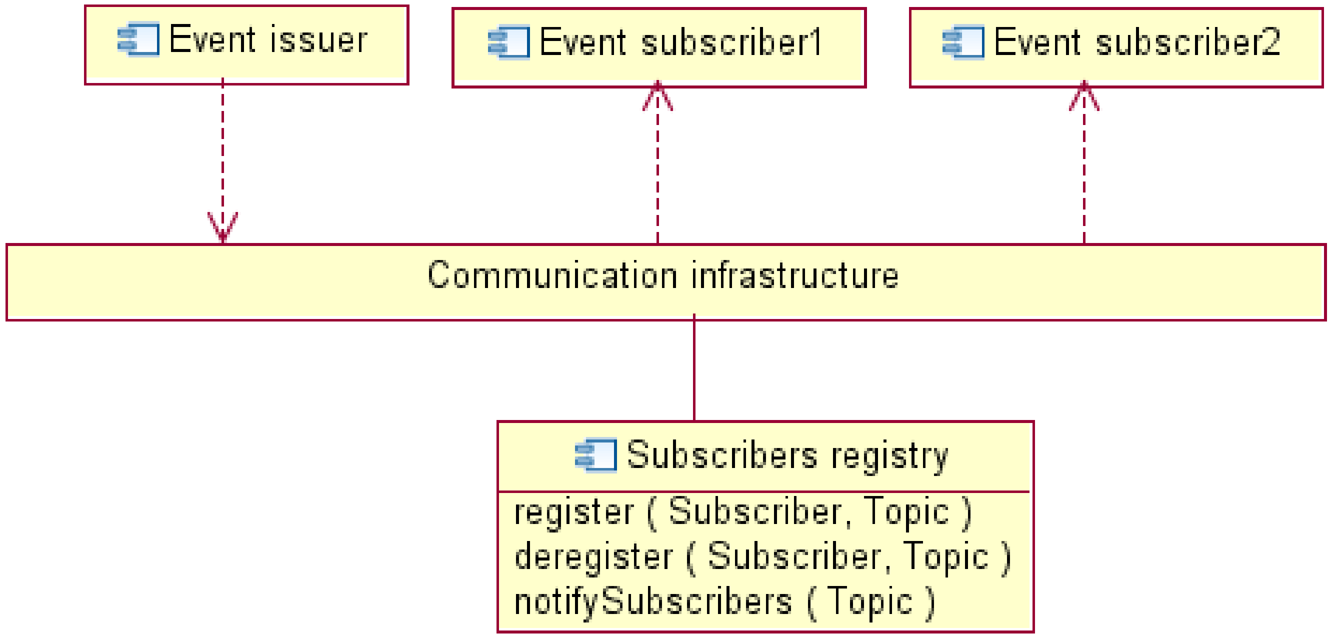
Spring architecture



Problem: point to point communication among components



Publish-subscribe

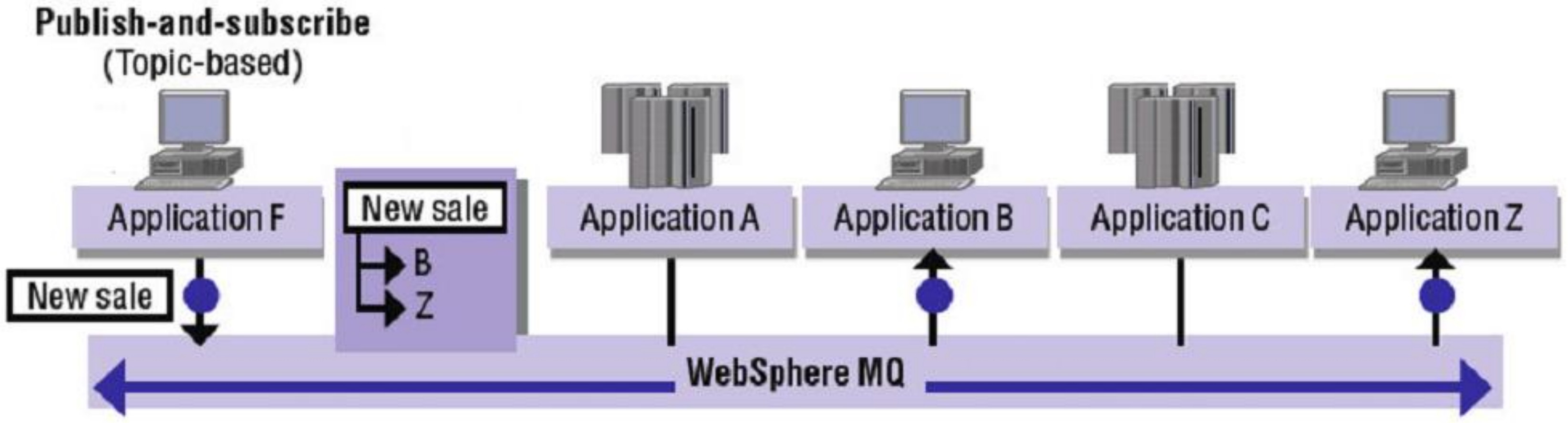

Event issuer
Event subscriber1
Event subscriber2
Communication infrastructure
Subscribers registry
register (Subscriber, Topic)
deregister (Subscriber, Topic)
notifySubscribers (Topic)

Implemented tactics

Modifiability tactics

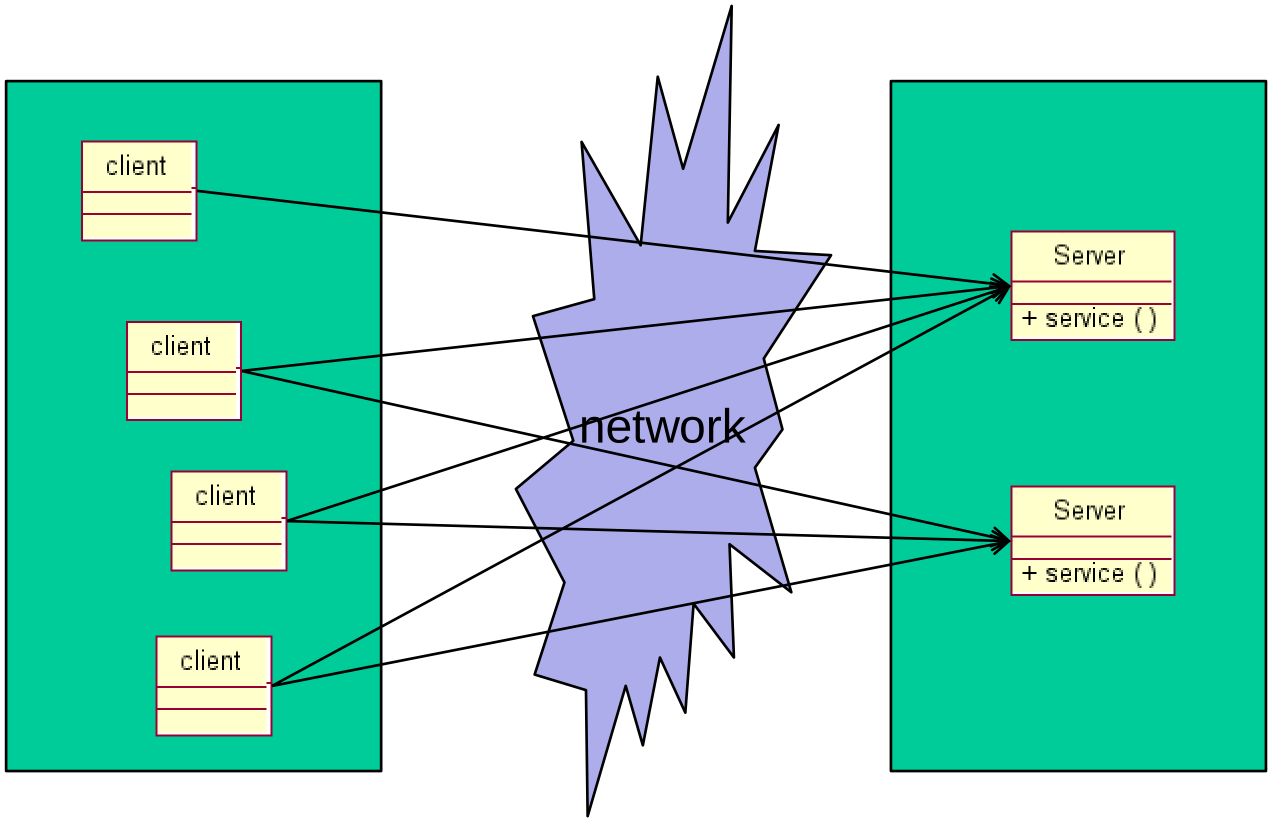
- Restrict communication paths
- Use an intermediary

Example: enterprise service bus

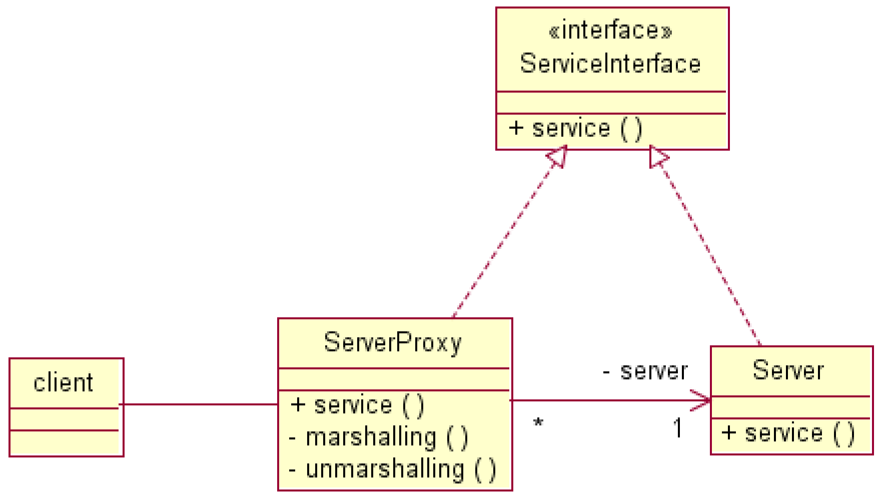


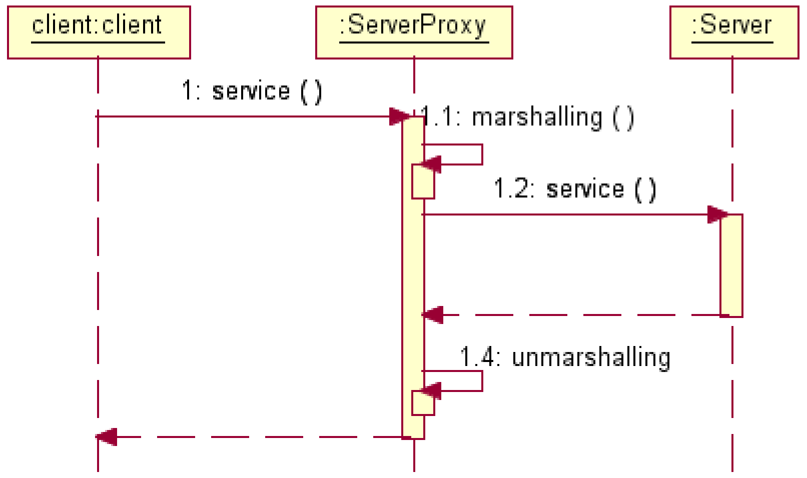
Source: Providing a backbone for connectivity with SOA Messaging, IBM White Paper, June 2009

Problem: remote connections

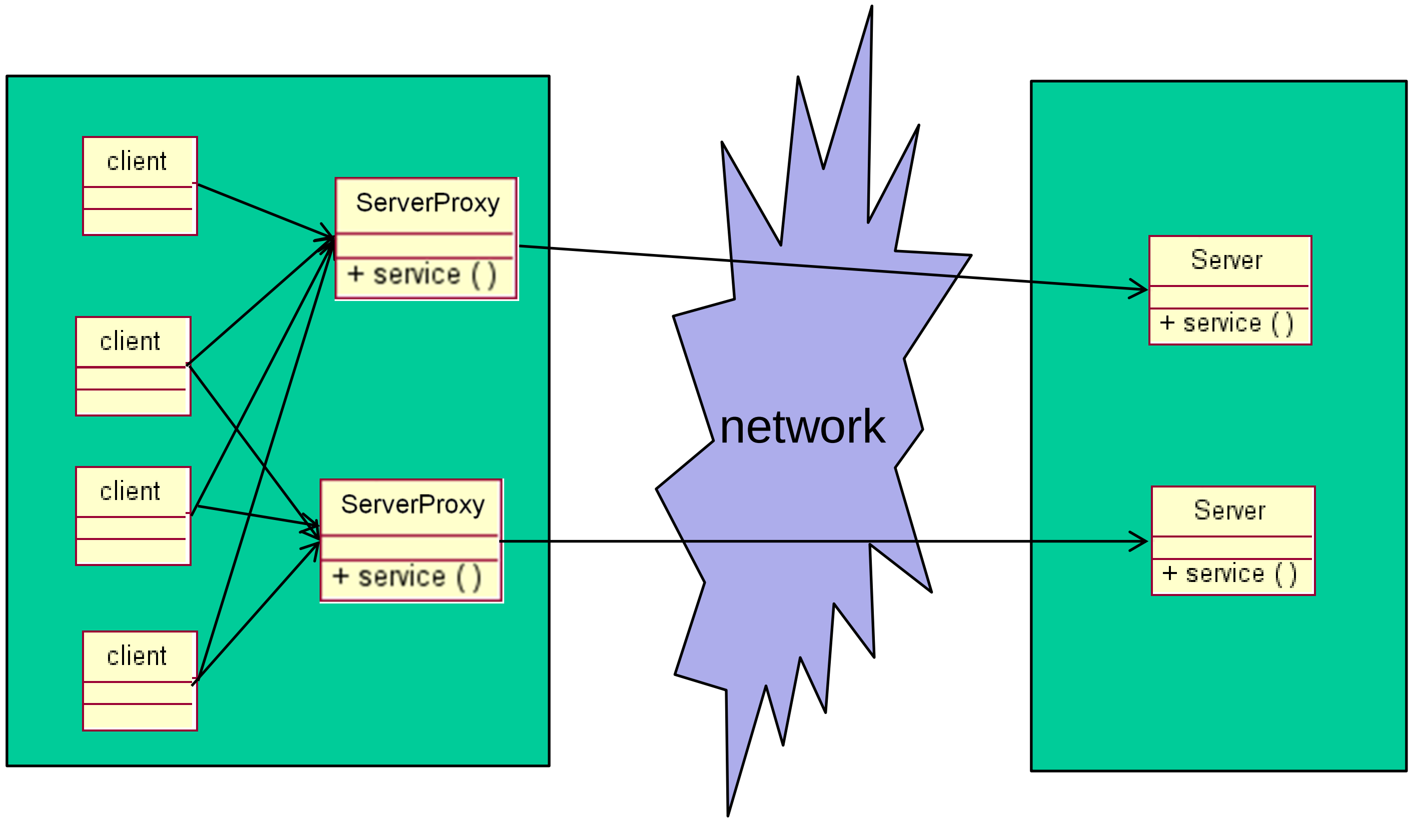


Remote proxy





With proxies…



Implemented tactics

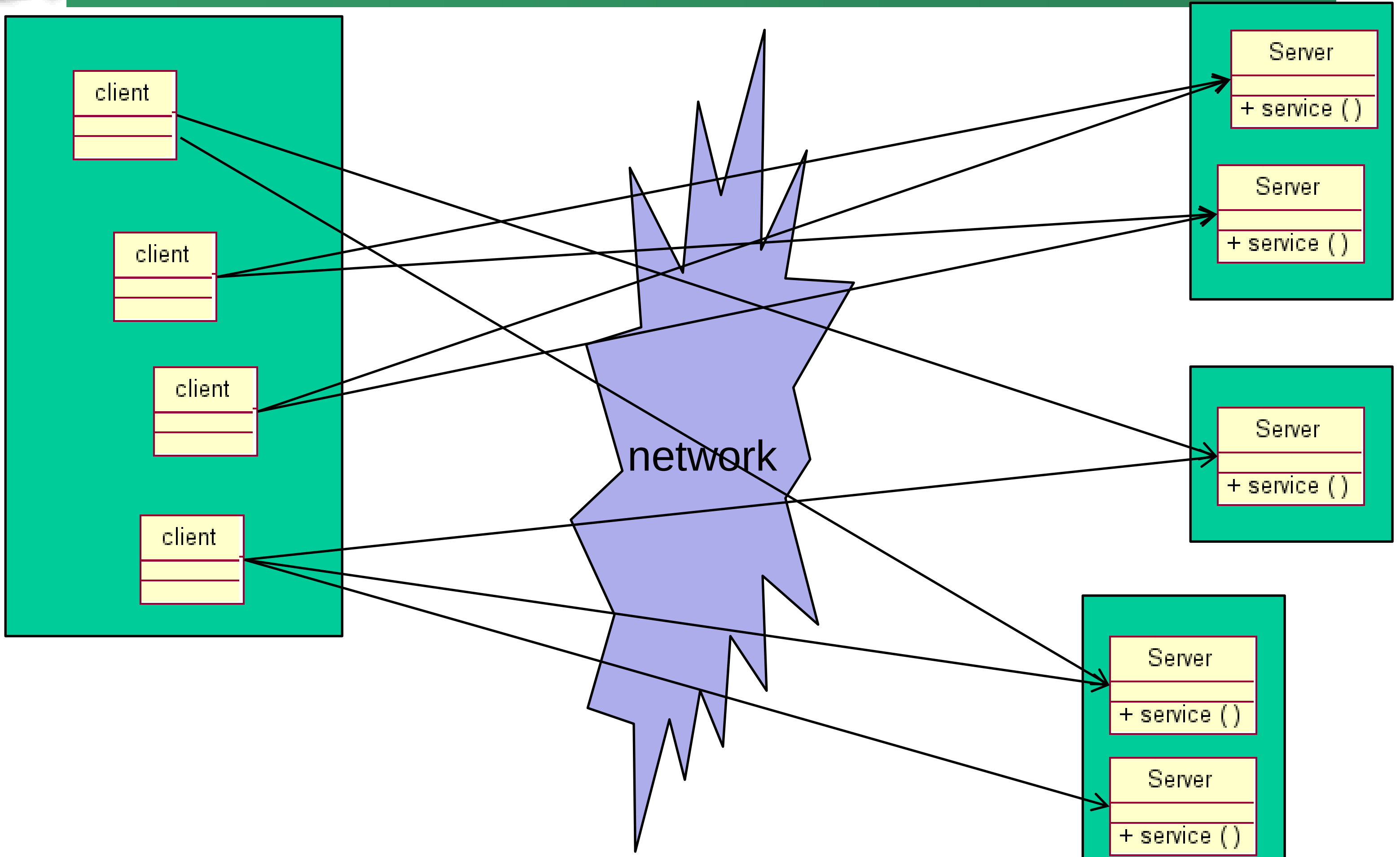
Modifiability tactics

- Restrict communication paths
- Use an intermediary

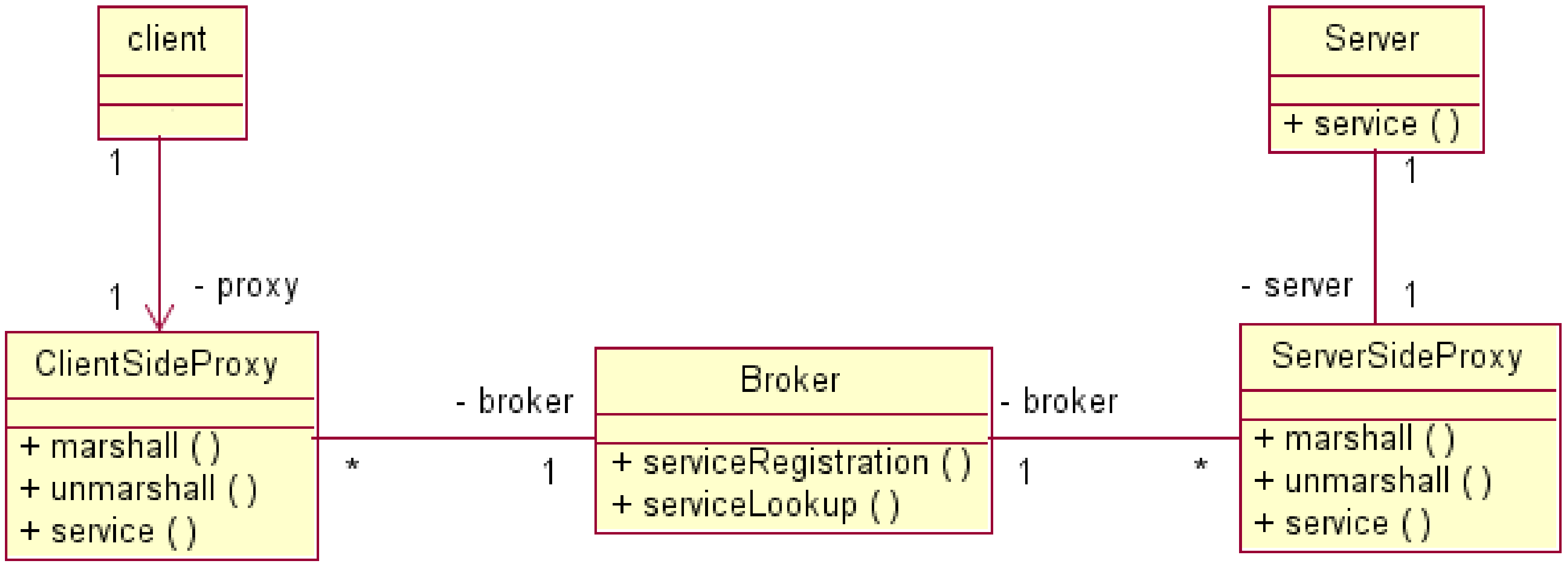
Performance tactics (in case of caching)

- Maintain multiples copies of either data or computation

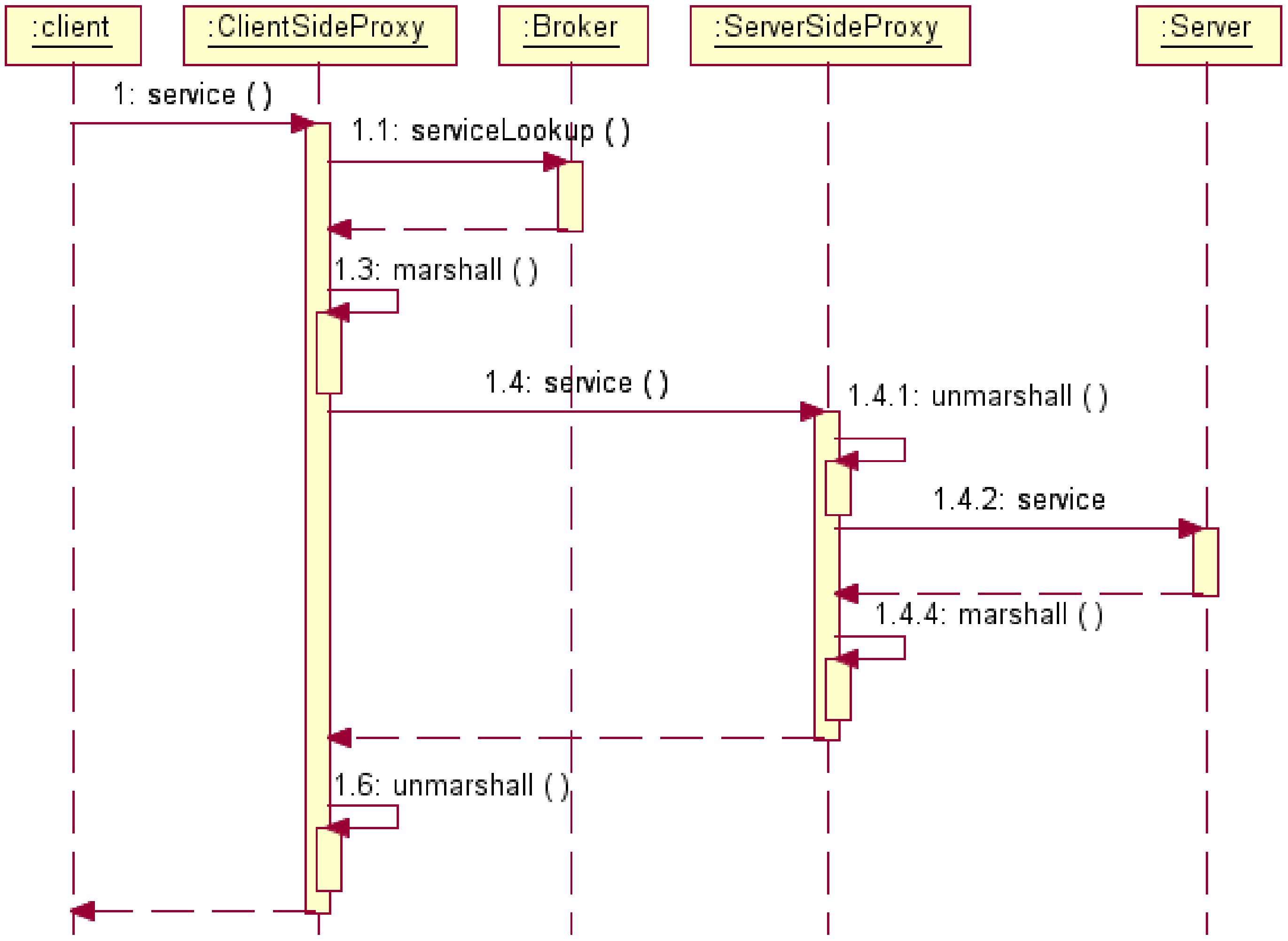
Problem: service location



Broker


client
1
1
- proxy
Server
+ service ( )
1
- server
1
ClientSideProxy
+ marshall ( )
+ unmarshall ( )
+ service ( )
- broker
*
1
Broker
+ serviceRegistration ( )
+ serviceLookup ( )
- broker
1
*
ServerSideProxy
+ marshall ( )
+ unmarshall ( )
+ service ( )

Broker


:client
:ClientSideProxy
:Broker
:ServerSideProxy
:Server
1: service ( )
1.1: serviceLookup ( )
1.3: marshall ( )
1.4: service ( )
1.4.1: unmarshall ( )
1.4.2: service
1.4.4: marshall ( )
1.6: unmarshall ( )

Implemented tactics

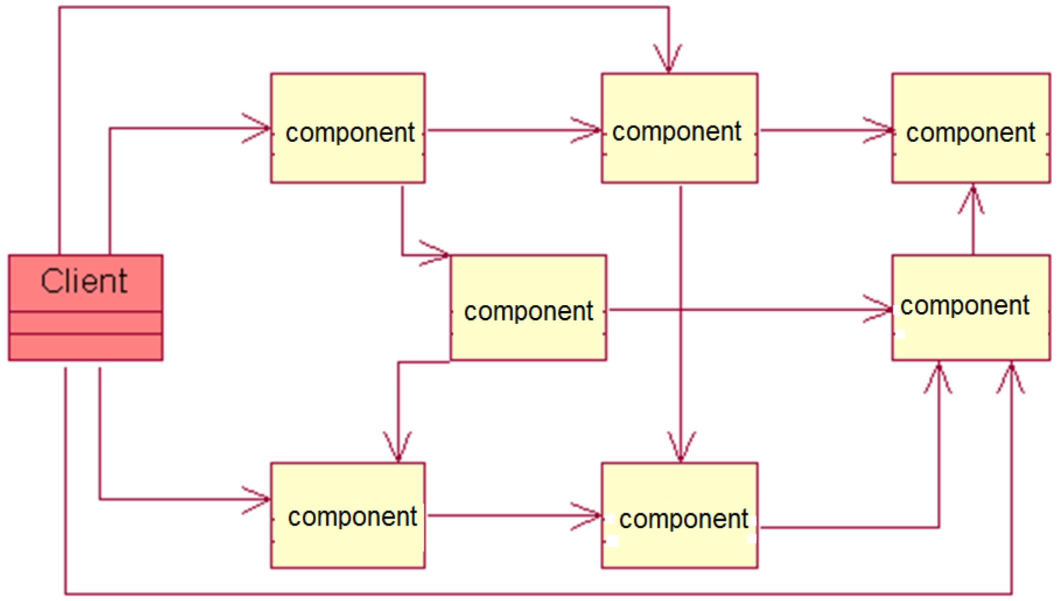
Modifiability tactics

- Anticipate changes
- Use an intermediary

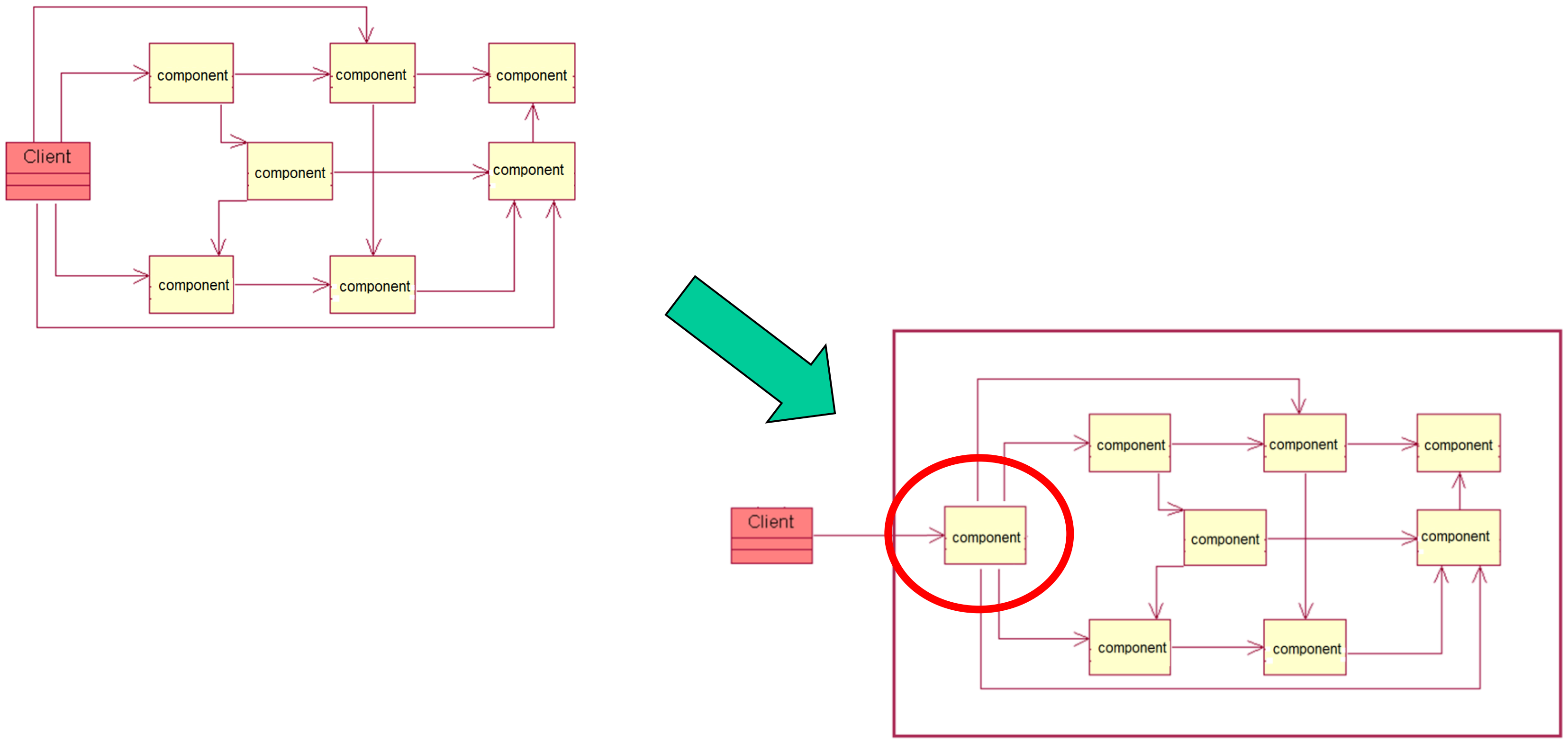
Performance tactics

- Maintain multiples copies of either data or **computation**
  (in case of load balancing)

Problem: too many communications paths



Facade

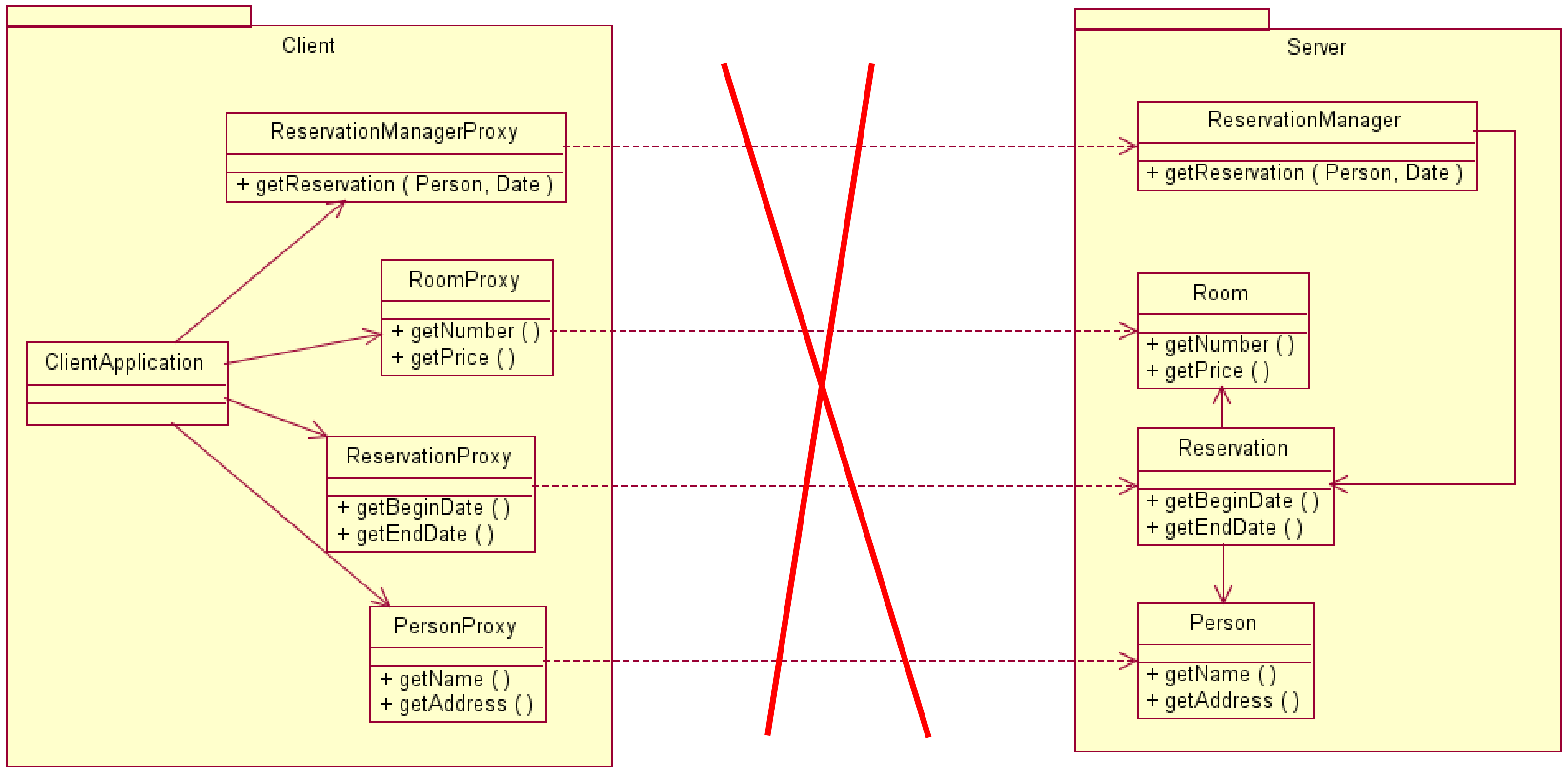

Client
component
component
component
component
component
component
component
Client
component
component
component
component
component
component
component
component

Implemented tactics

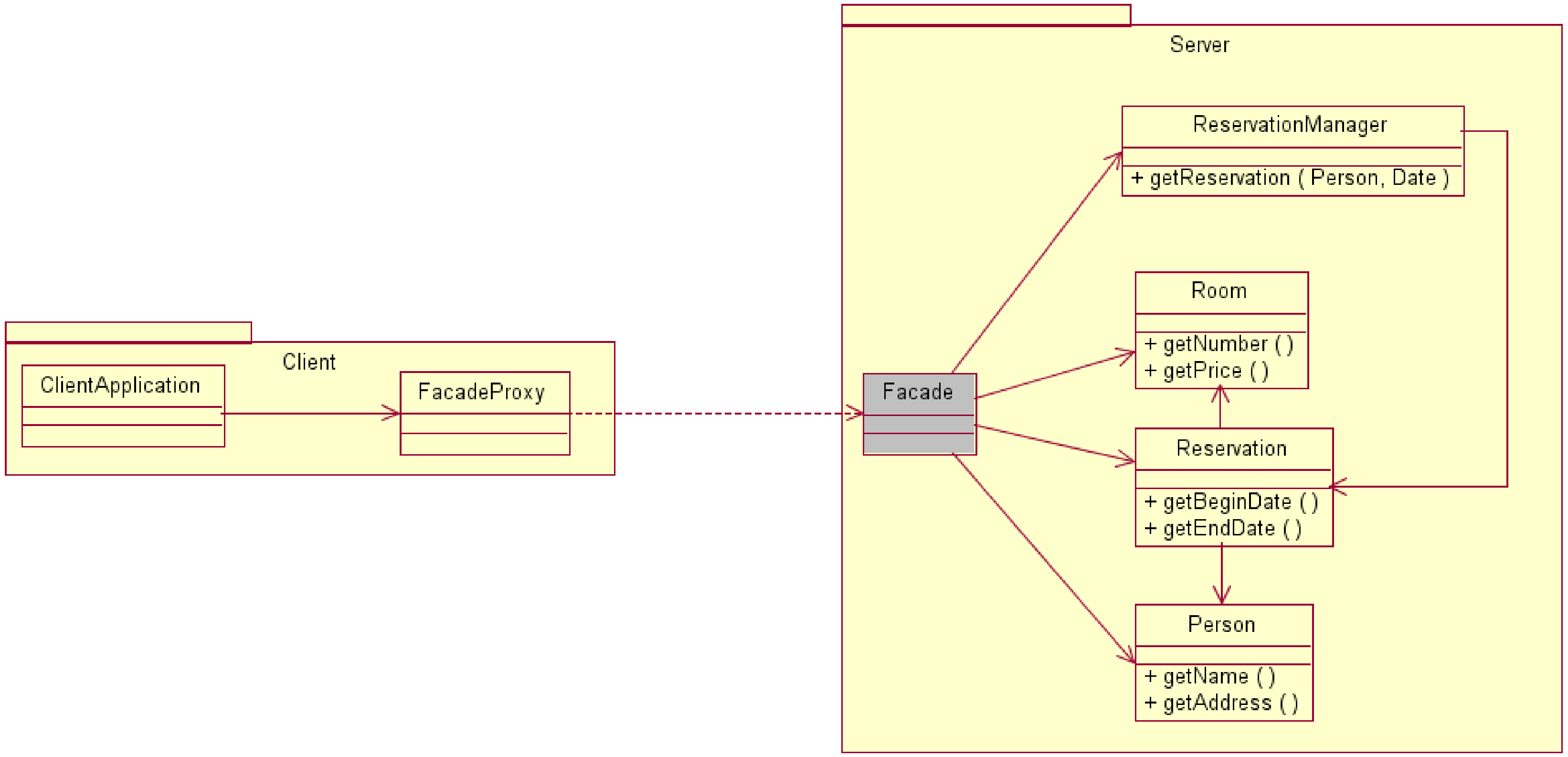
Modifiability tactics

- Hide information
- Restrict communication paths
- Use an intermediary

Example: client-server objects & proxies



New architecture: façade & proxy



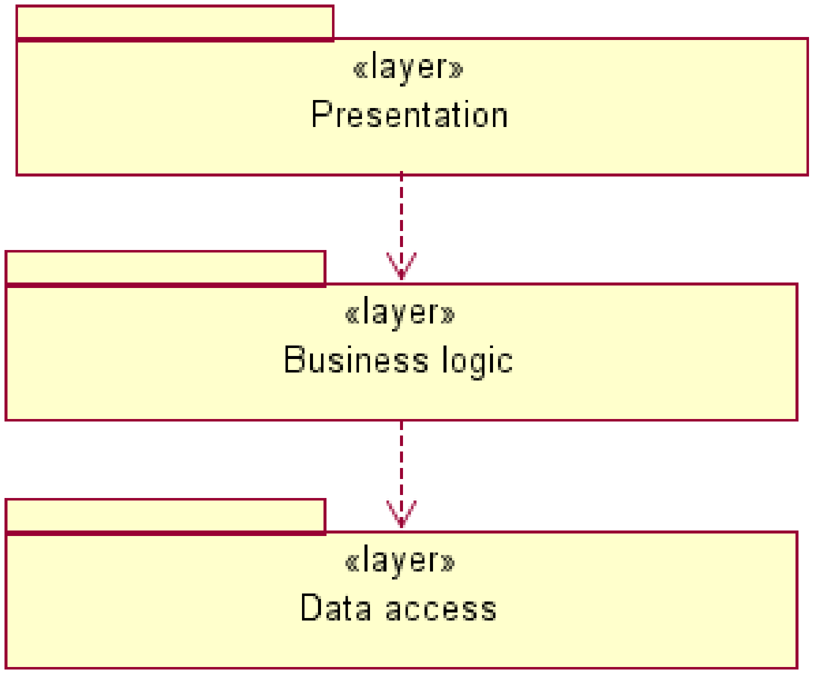
“Layers” is the most common pattern

How to choose layers?

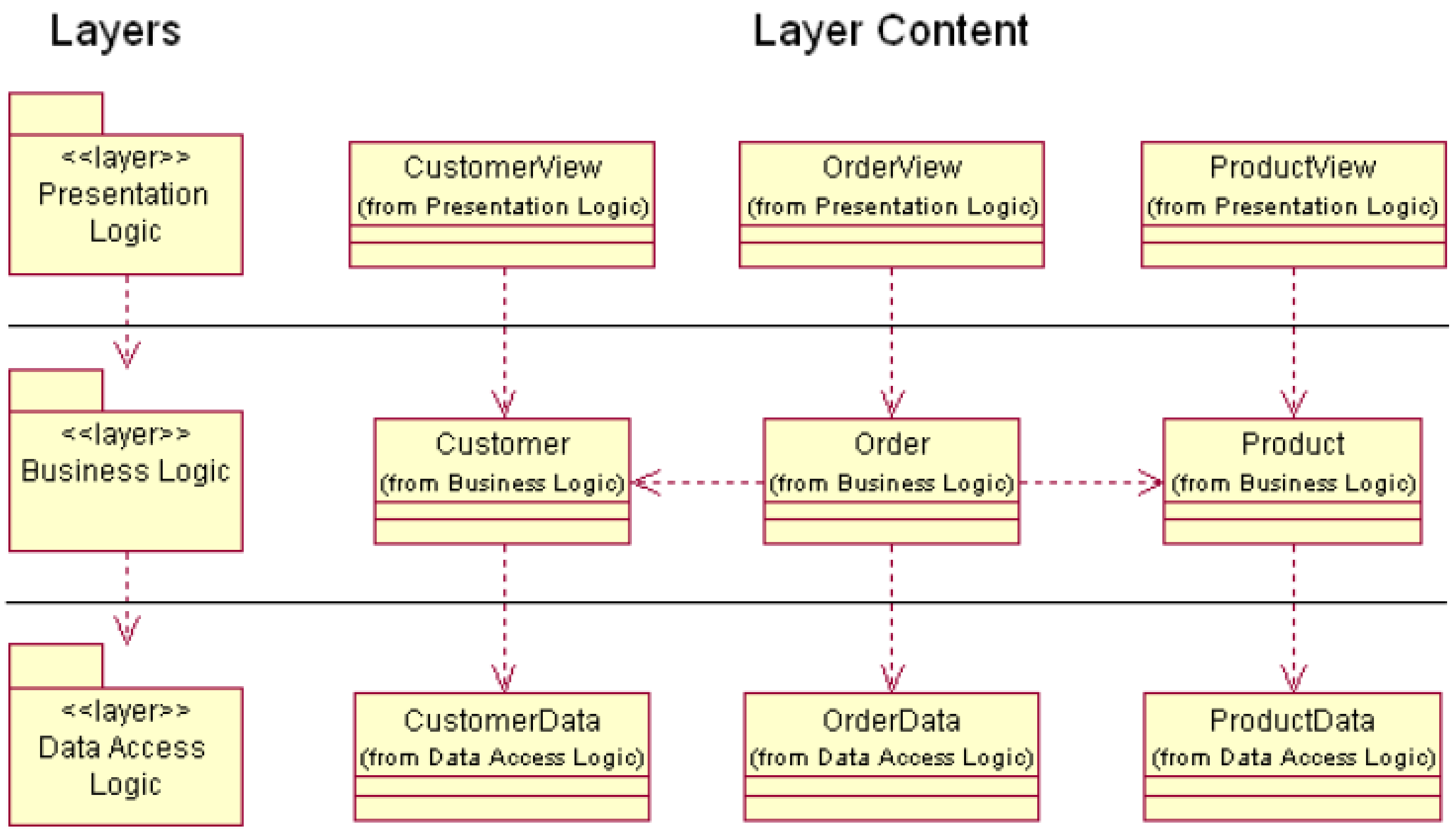
Anticipate changes !

Change in one of the concerns: display, data access

Responsibility-based structure

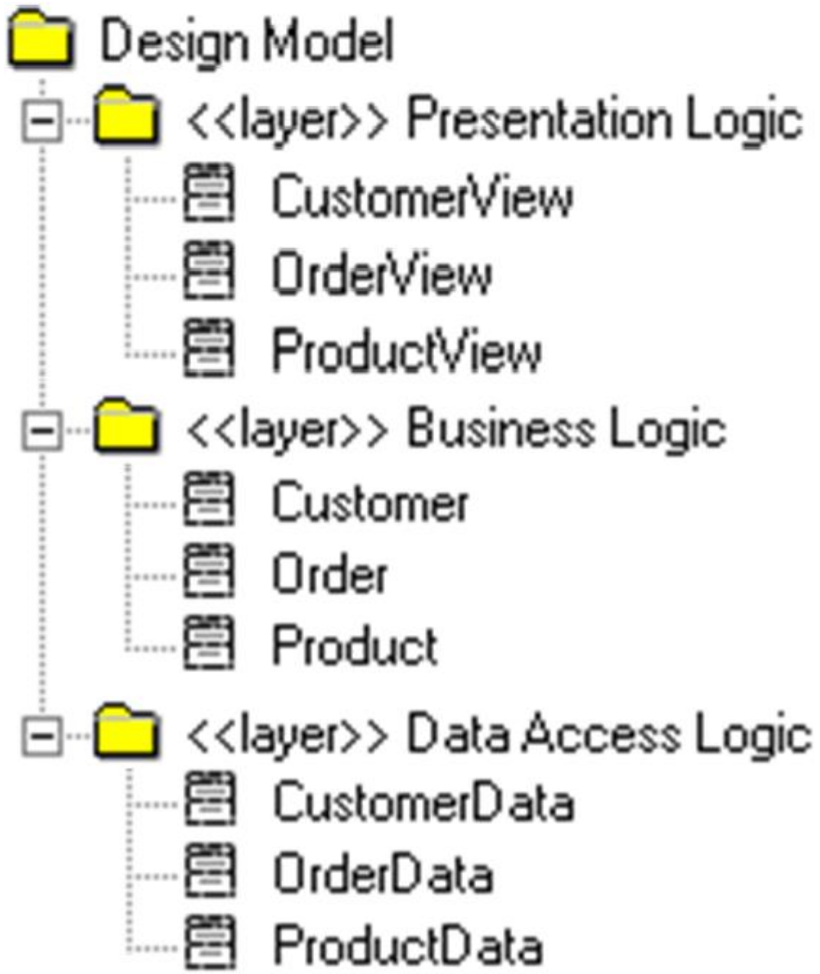


Exemple



Layers
Layer Content
<<layer>> Presentation Logic
<<layer>> Business Logic
<<layer>> Data Access Logic
CustomerView (from Presentation Logic)
OrderView (from Presentation Logic)
ProductView (from Presentation Logic)
Customer (from Business Logic)
Order (from Business Logic)
Product (from Business Logic)
CustomerData (from Data Access Logic)
OrderData (from Data Access Logic)
ProductData (from Data Access Logic)

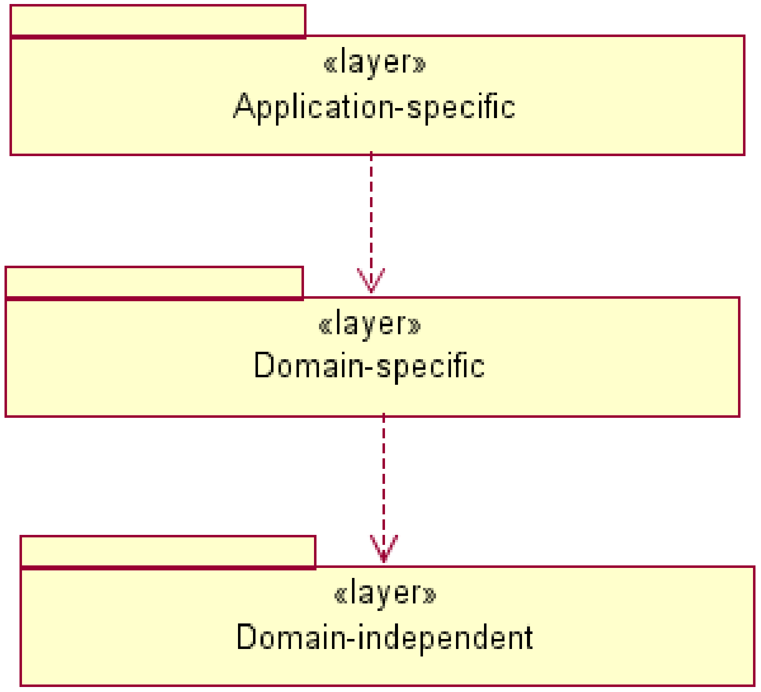
Representation of the example as Java packages / folders



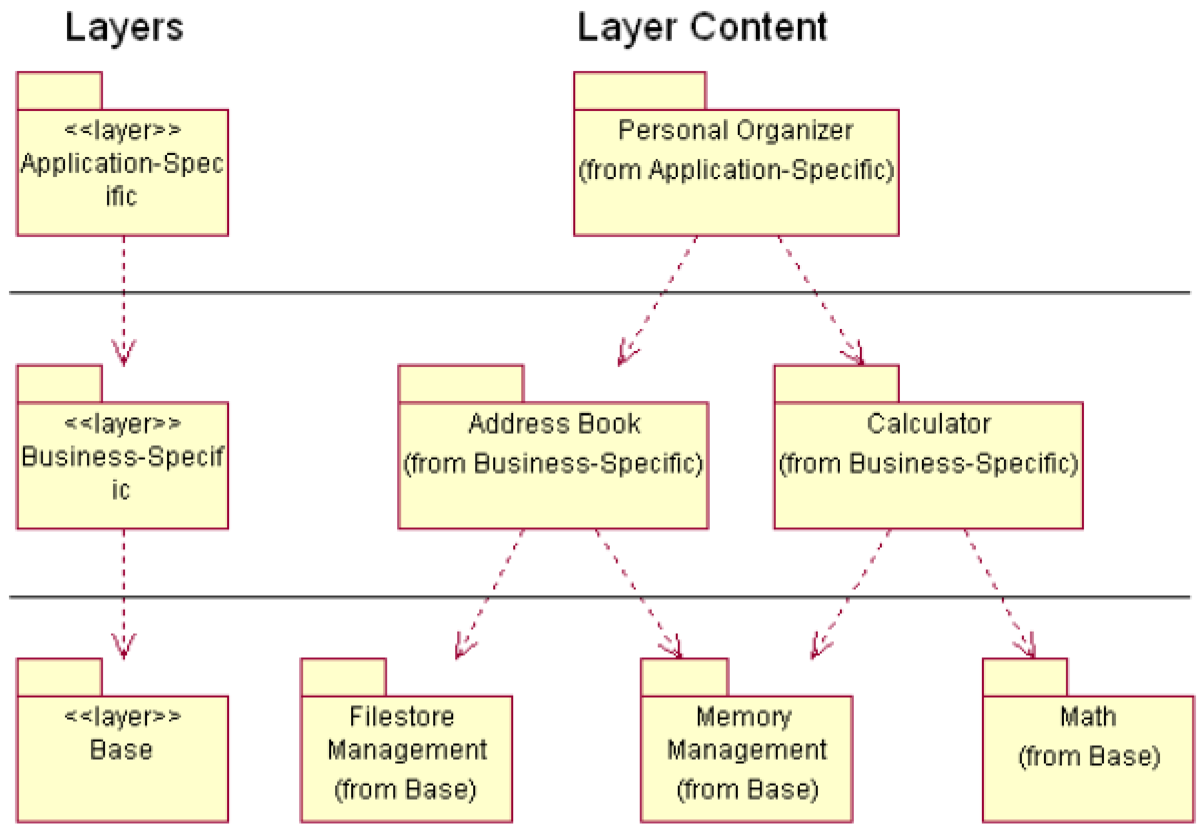
Design Model
- <<layer>> Presentation Logic
  - CustomerView
  - OrderView
  - ProductView
- <<layer>> Business Logic
  - Customer
  - Order
  - Product
- <<layer>> Data Access Logic
  - CustomerData
  - OrderData
  - ProductData

Change in functionalities : adding / modifying functions of the software

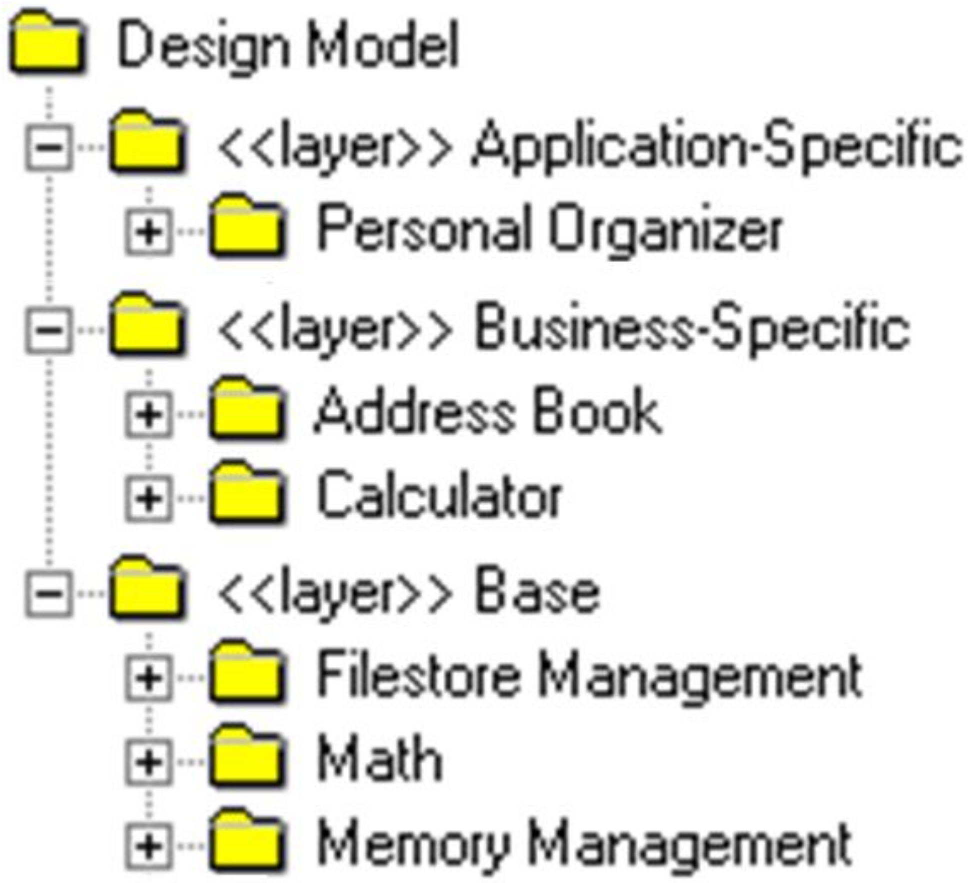
Reuse-based structure



Exemple


Layers
Layer Content
<<layer>> Application-Specific
<<layer>> Business-Specific
<<layer>> Base
Personal Organizer (from Application-Specific)
Address Book (from Business-Specific)
Calculator (from Business-Specific)
Filestore Management (from Base)
Memory Management (from Base)
Math (from Base)

Representation of the example as Java packages / folders

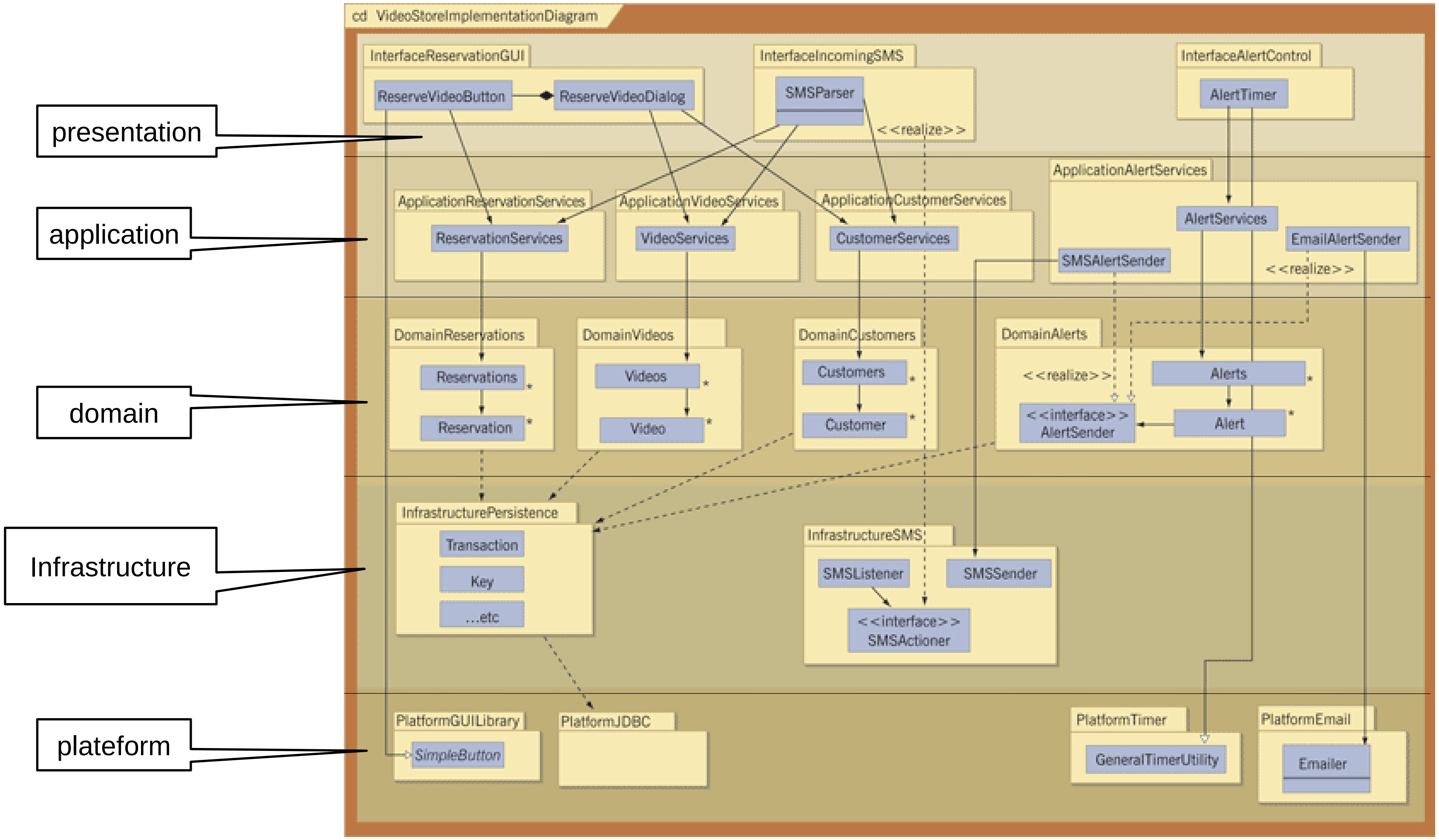


- Design Model
  - <<layer>> Application-Specific
    - Personal Organizer
  - <<layer>> Business-Specific
    - Address Book
    - Calculator
  - <<layer>> Base
    - Filestore Management
    - Math
    - Memory Management

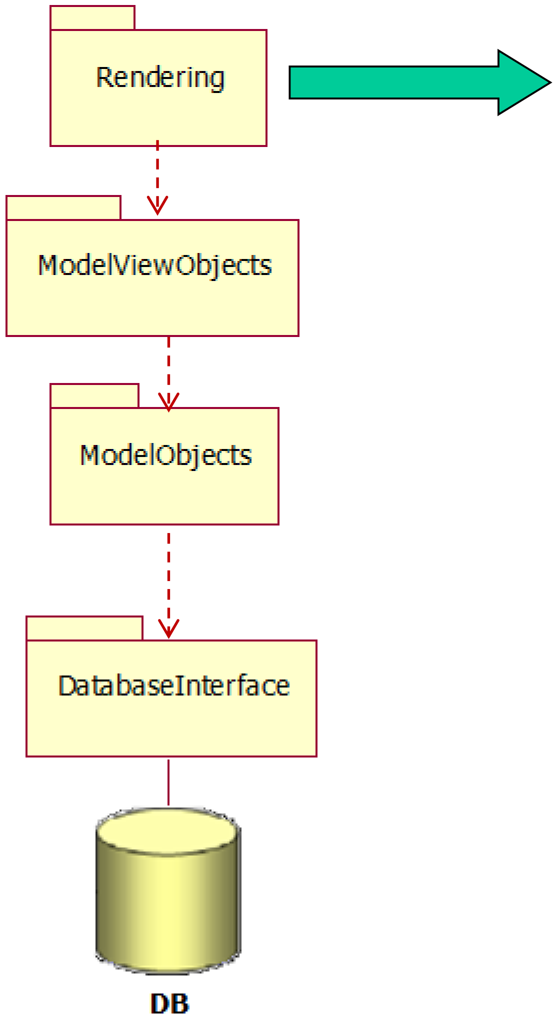
Simple case study: video shop

- Customers can register to receive sms or email alerts when a video they chose is available
- Customer can reserve the video by responding to the alert.
- Customers can search the video database through a GUI interface and reserve the selected video.

Video shop
Abstraction / layer responsibility ?



Another example: EvoSpaces


Rendering
ModelViewObjects
ModelObjects
DatabaseInterface
DB