Dependable Off-The-Shelf based middleware systems for Large-scale Complex Critical Infrastructures

Università degli Studi di NAPOLI "Federico II"
Università degli Studi di FIRENZE
Università degli Studi di MODENA e REGGIO EMILIA
Università degli Studi di NAPOLI "Parthenope"
Università degli Studi di ROMA "La Sapienza"

PRIN Project DOTS-LCCI

Winter School on Hot Topics in Secure and Dependable Computing for Critical Infrastructures

Cortina d'Ampezzo, 16-19/01/2012

Software Architectures for Large-Scale Critical Systems

Stefano Russo

Dipartimento di Informatica e Sistemistica, Università di Napoli Federico II

Laboratorio Nazionale "C. Savy", Consorzio Interuniversitario Naz. per l'Informatica

Stefano.Russo@unina.it

The MobiLab Group

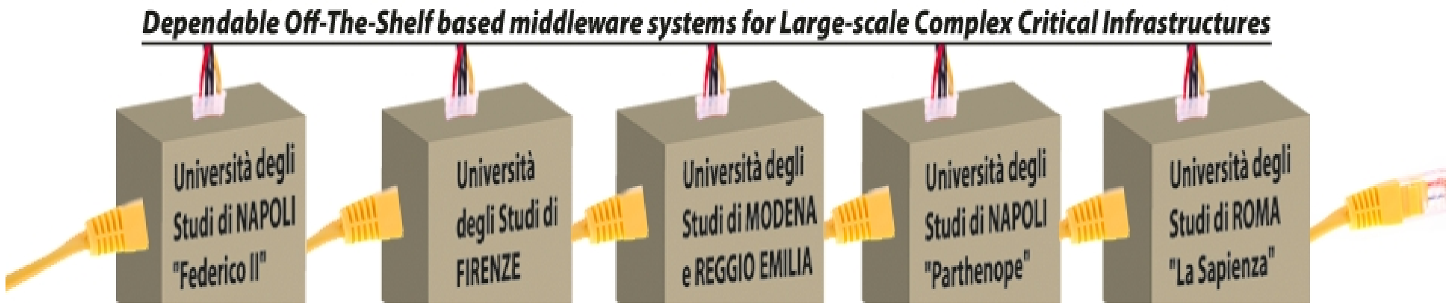
www.mobilab.unina.it

consorzio interuniversitario nazionale per l'informatica

Outline

- Definitions, history, standard, key design principles
- Stakeholders, viewpoints, views
- Types of software assets
- Defining software architectures
- The role of the software architect
- Evaluating software architectures
- Decomposition techniques
- Architectural styles

This presentation is partially based on courseware material produced along with prof. Paolo Ciancarini from University of Bologna for advanced training courses for CINI (Consorzio Interuniversitario Nazionale per l'Informatica).

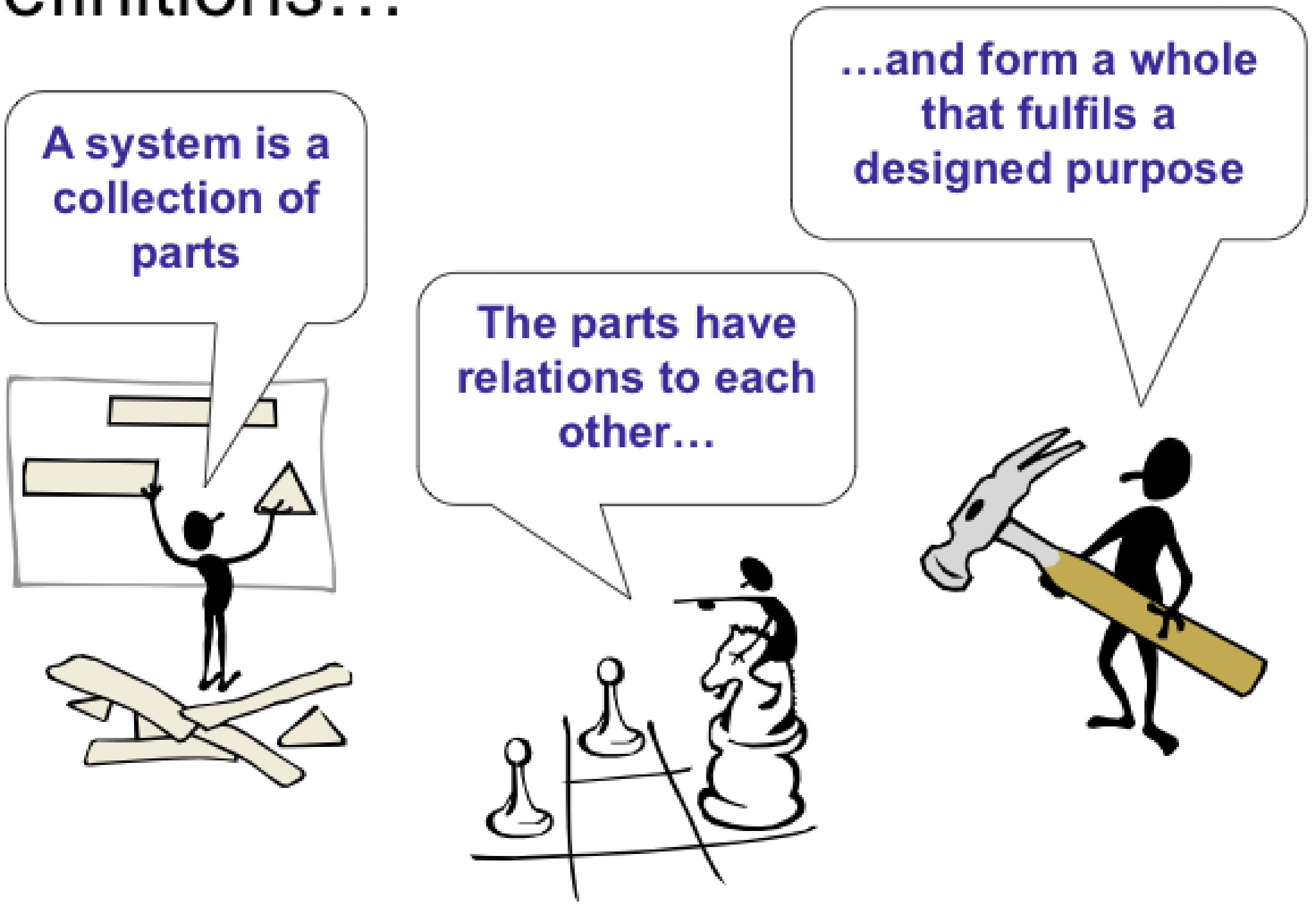

Dependable Off-The-Shelf based middleware systems for Large-scale Complex Critical Infrastructures
Università degli Studi di NAPOLI "Federico II"
Università degli Studi di FIRENZE
Università degli Studi di MODENA e REGGIO EMILIA
Università degli Studi di NAPOLI "Parthenope"
Università degli Studi di ROMA "La Sapienza"

Software Architectures

Definitions, brief history, standards,
key principles, software architecting process,
documenting architectures,
patterns, architectural styles

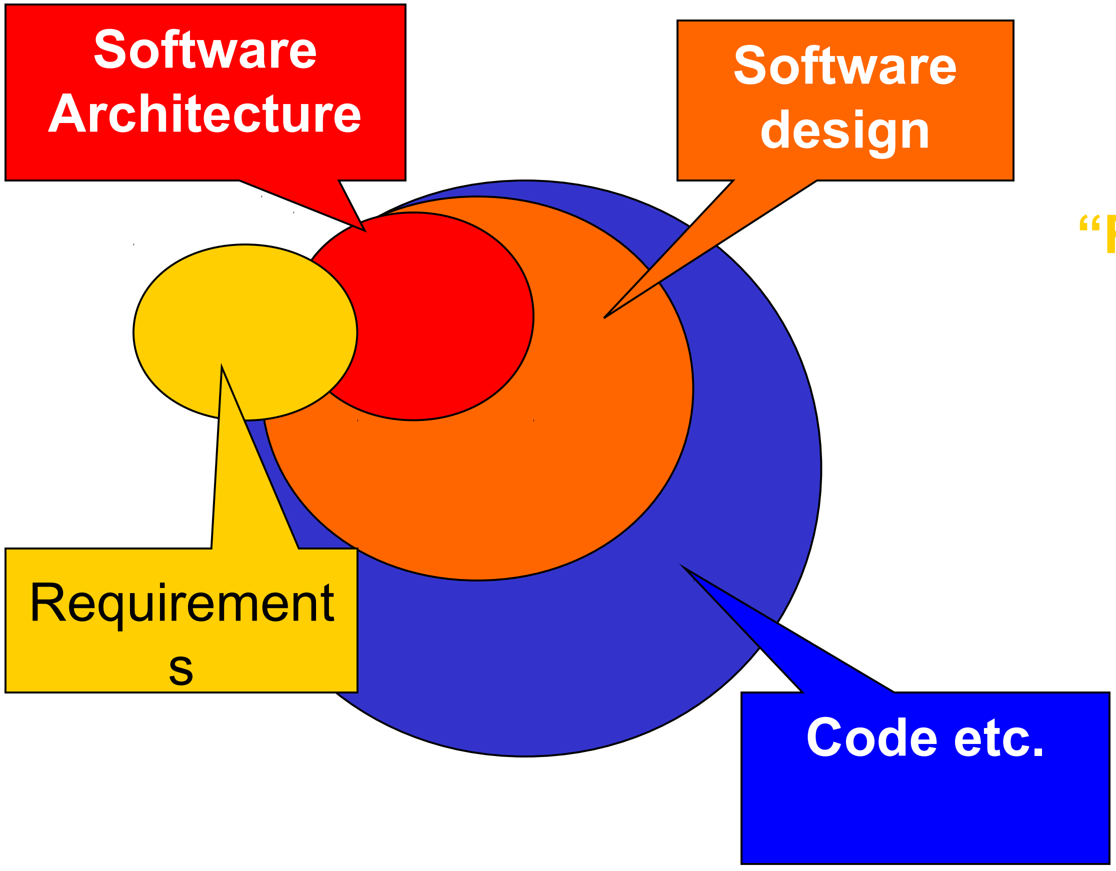
System

Definitions…


A system is a collection of parts
The parts have relations to each other…
…and form a whole that fulfils a designed purpose

What is the role of architecture?

Architecture = design decisions



“Requirements constraints”

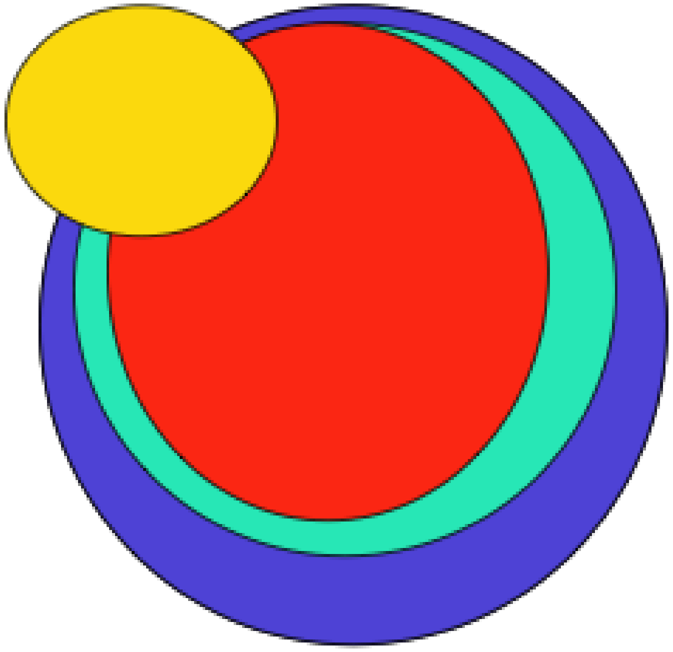
“Design” decisions

Architectural decisions

Other decisions

All choices are binding in the final product

Architecture = design? No!



"Do not dilute the meaning of the term architecture by applying it to everything in sight."

Mary Shaw

Definition – 1/2

- Intuitively, the architecture of a software system represents its internal structure, in terms of the single components of which it is made up and of the interactions among them
- According to the IEEE/ANSI 1471-2000 standard, the architecture of a software system is its basic organization, in terms of its components, of the interactions among components and with the external environments, as well as of the basic, driving, principles for system design and evolution

Definition – 2/2

- From Wikipedia, the free encyclopedia
  - The software architecture of a program or computing system is the **structure** or structures of the system, which comprise **software components**, the **externally visible properties** of those components, and the **relationships** between them.
  - The term also refers to **documentation** of a system's software architecture. Documenting software architecture facilitates communication between stakeholders, documents early decisions about high-level design, and allows reuse of design components and patterns between projects

Software architecture brief history

- The basics of software architectures have been introduced by Dijkstra (1968) and Parnas (1970), who argued:
  - The key role of the structure of a software system, and
  - How tricky the definition of the *right structure* for a given system is
- A great deal of research has been conducted on software architectures during ’90s, mainly focusing on:
  - Architectural styles;
  - How to solve recurrent problems (*pattern*);
  - The definition of languages to describe software architectures (ADL)
  - Documentation of the architecture of a system

Sw engineering vs sw architecture

- Software Engineering is a discipline studying the methods to produce software, the theories at their basis, and the tools to effectively develop and measure the qualities of software systems
- Software engineering deals with limited resources
- It is a discipline strongly empirical, that is based on experience and past projects

- **Is software architecture a discipline**?
  - Claim originated with the book by **Shaw** & **Garlan,** *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996

Software architecture discipline

- The discipline of software architecture is centered on the idea of managing the design complexity of software systems through abstraction and separation of concerns
- The discipline has developed a number of design **styles** and **patterns** that help in designing or integrating software intensive systems
- However,it is hard to find software architects who agree on the right way to architect a software system
- Wrong decisions in crafting the software architecture are a major cause of project cancellation

Why an architecture? – 3/4

- Design is typically driven by functional requirements
- Architecture is driven more by non-functional requirements / software quality attributes

- Some standards in specific domains are posing some emphasis (directly or indirectly) on software architecture
  - E.g, DO-178B in the avionic domain
- Need to trace requirements through architecture up to software modules and code

Why an architecture? – 1/4

- Every software system has an architecture
- Many kind of complex systems demand for putting proper effort in the definition of the software architecture:
  - Large-scale systems
  - Software intensive systems
  - Dependable systems
    - Safety critical systems
    - Mission critical systems
      - Business-critical
      - Operation-critical
  - Long-lived systems
  - Example: software systems for critical infrastructures

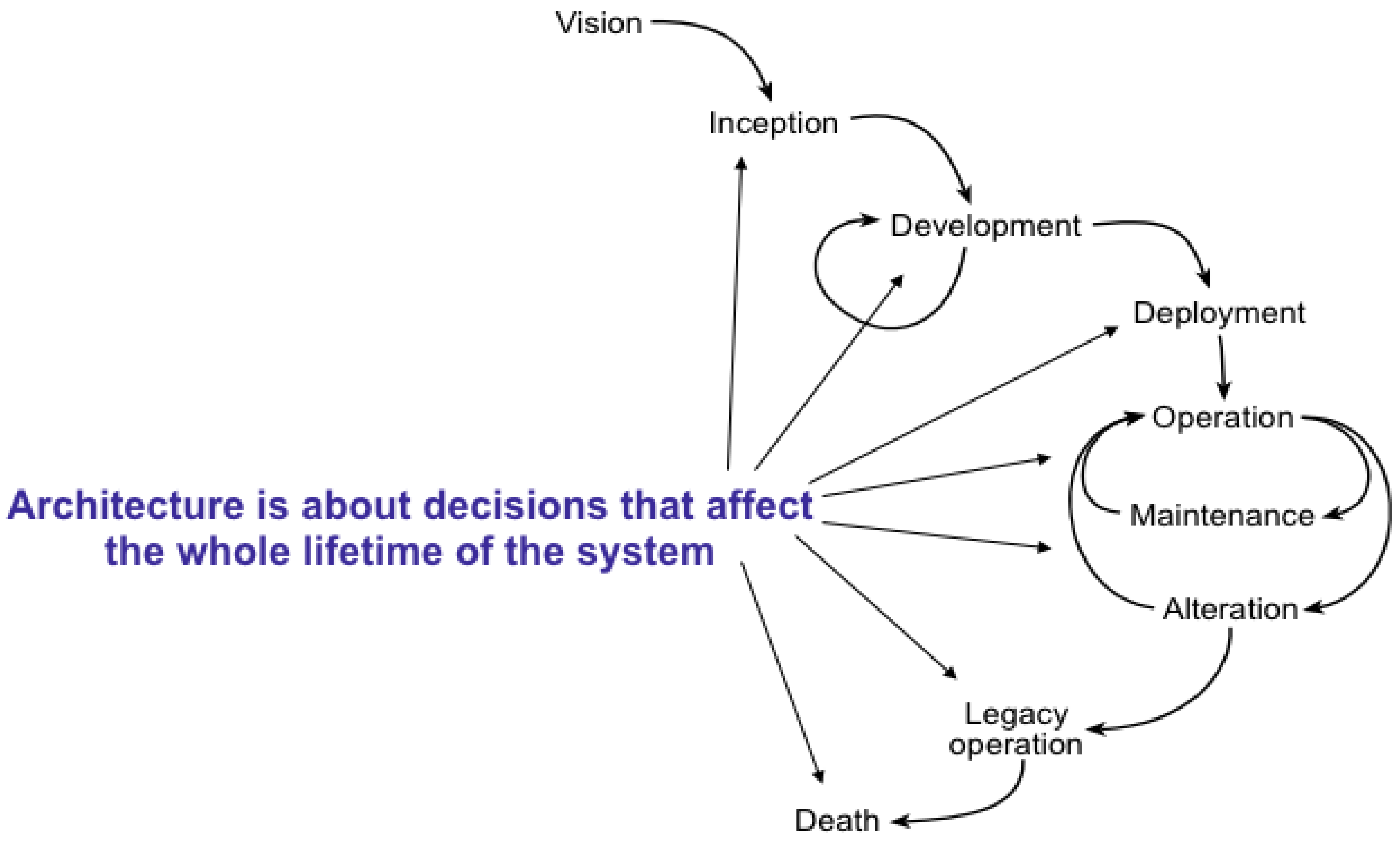
Why an architecture? – 2/4

- In complex systems quality attributes cannot be achieved in isolation
  - Achieving one will have an effect, maybe positive or negative, on others
- Examples:
  - Security and reliability
    - The most secure system has fewest points of failures, the most reliable system has typically the most
  - Portability and performance
    - Portability is usually achieved isolating system dependencies, which introduces overhead
- The architecture definition process helps finding the tradeoffs that ensure meeting requirements and desired quality factors

Why an architecture? – 4/4

- Companies are increasingly perceiving the importance of software architecture
  - Software companies / System integrators
- These companies are setting up ad hoc <u>software architectures divisions</u>, that have to develop a long-term vision of their products/product lines
  - Different from Business Units and from product engineering/development teams → SA div. responsible for enforcing reuse (of any artifacts)
  - A bridge between BUs (sys requirements) and Engineering divisions (software design and implem.)
- Each large project has at least a software architect
  - A senior IT and/or software specialist with proper skills

Architecture and system lifetime



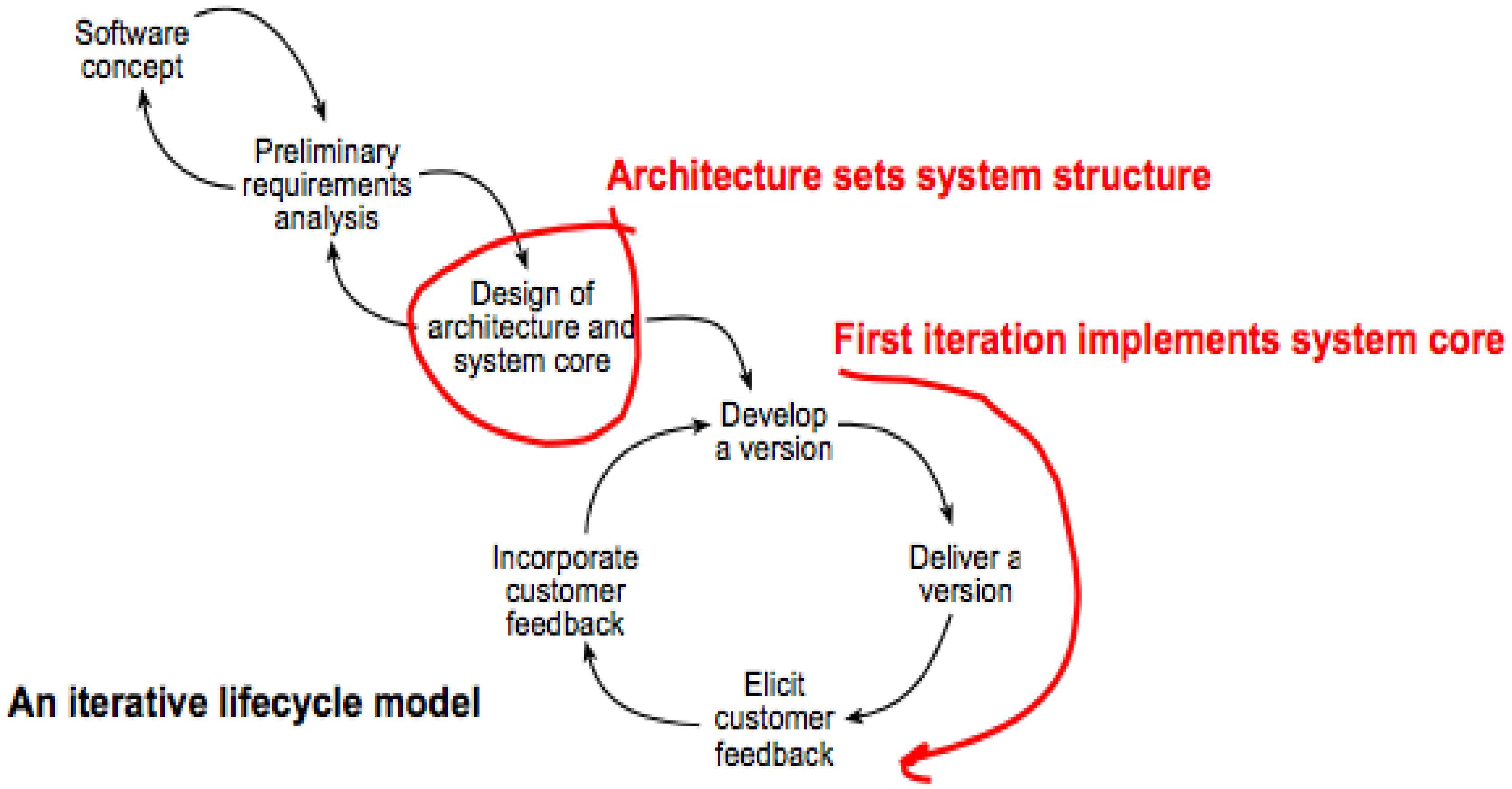
Architectural decisions

- *[Architectural decisions are] conscious design decisions concerning a software system as a whole, or one or more of its core components. These decisions determine the non-functional characteristics and quality factors of the system. [Zimmermann]*
- *Architectural decisions are* ***strategic decisions***

  Architecture = strategic design

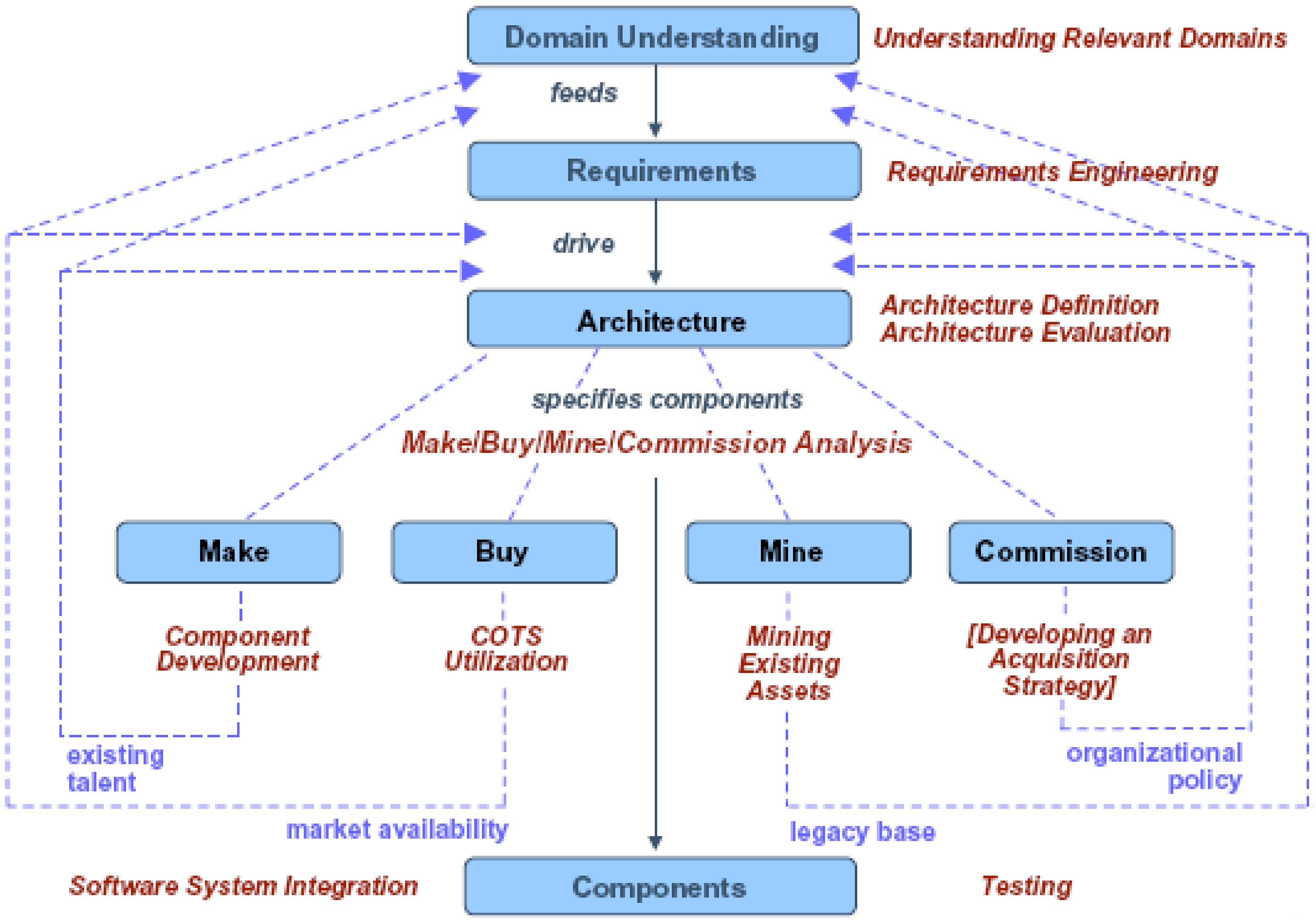
  (Functional) design = tactical design
- *Decision rationale may come from experience, method or some other asset*

The role of architecture in software


Software concept
Preliminary requirements analysis
Design of architecture and system core
Develop a version
Deliver a version
Elicit customer feedback
Incorporate customer feedback
Architecture sets system structure
First iteration implements system core
An iterative lifecycle model

Architecture plays a vital role in establishing the structure of the system, early in the development lifecycle

The role of architecture in software



Software architecture and reuse

- Productivity in software development depends on reuse: algorithms, data structures, components
- A software library is a typical example of code reuse
- A web server is a typical example of component reuse
- A goal of architecture definition is the reuse of components
- An important issue is the reuse of design ideas: major examples are **design patterns** and **architectural styles**

Style: layered

- **An example of architectural styles:**

layered architecture

Portuguese francesinha

Duality of the reuse concept

Reuse design with the reuse of existing software artifacts

Riusability design for the development and maintenance of reusable software artifacts

technological innovation

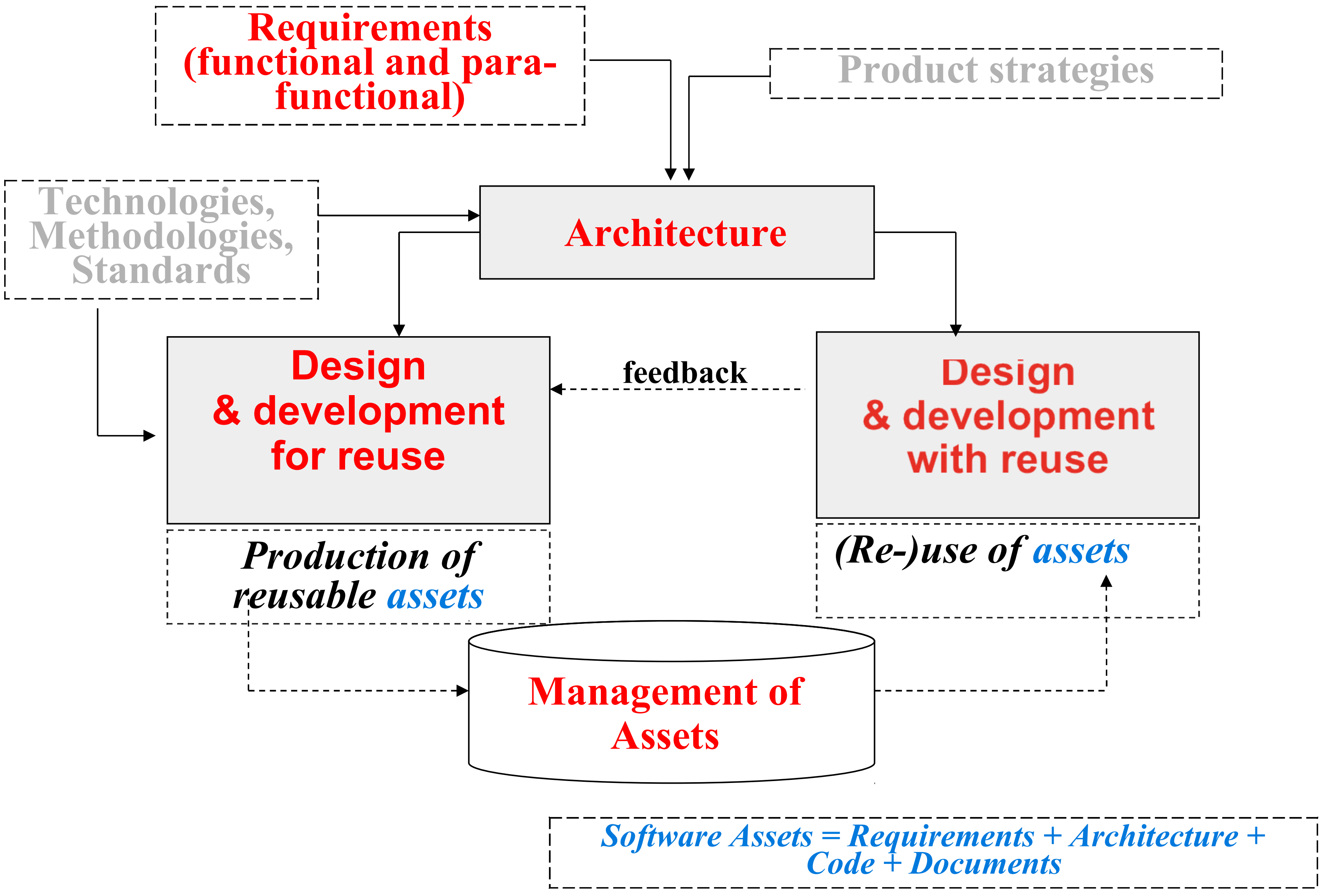
Riuse

Riusability

complexity

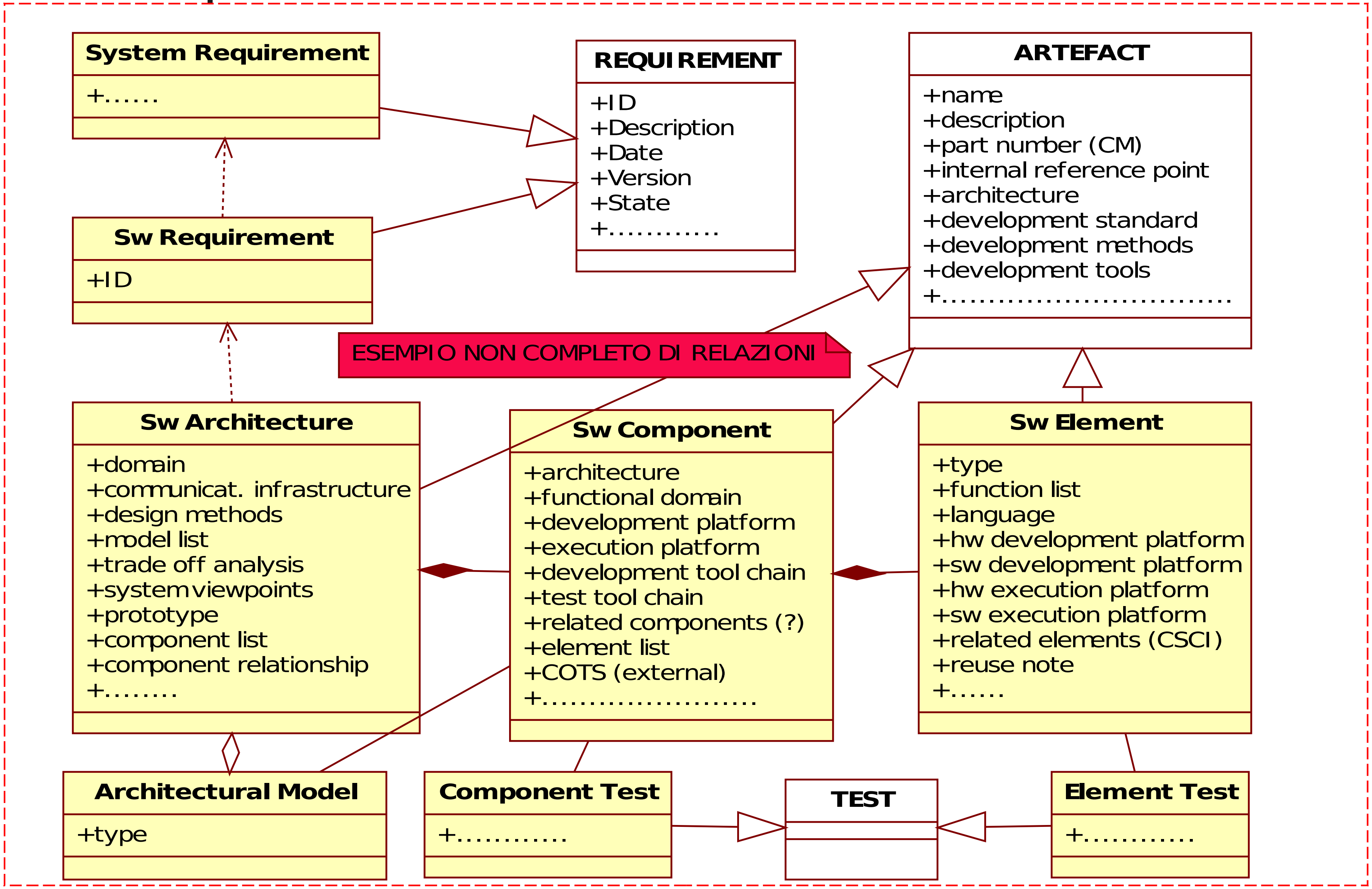
make /buy/ reuse

Reuse in software life cycle

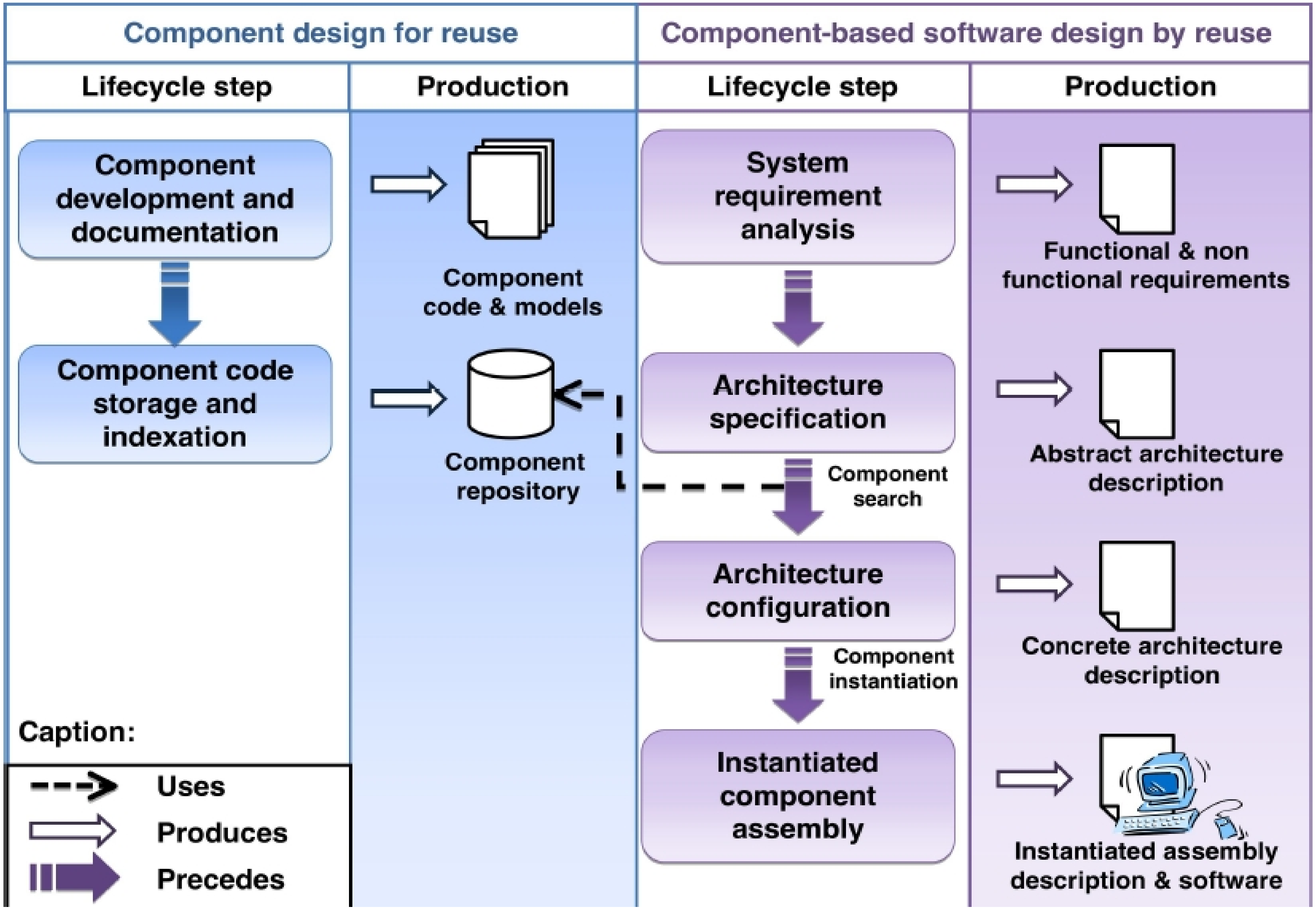


Requirements (functional and para-functional)
Product strategies
Technologies, Methodologies, Standards
Architecture
Design & development for reuse
feedback
Design & development with reuse
Production of reusable assets
(Re-)use of assets
Management of Assets
Software Assets = Requirements + Architecture + Code + Documents

Relationships among artifacts



An architecture goal: reuse of components


Component design for reuse
Component-based software design by reuse
Lifecycle step
Production
Lifecycle step
Production
Component development and documentation
Component code & models
Component code storage and indexation
Component repository
System requirement analysis
Functional & non functional requirements
Architecture specification
Abstract architecture description
Component search
Architecture configuration
Concrete architecture description
Component instantiation
Instantiated component assembly
Instantiated assembly description & software
Caption:
Uses
Produces
Precedes

Standards

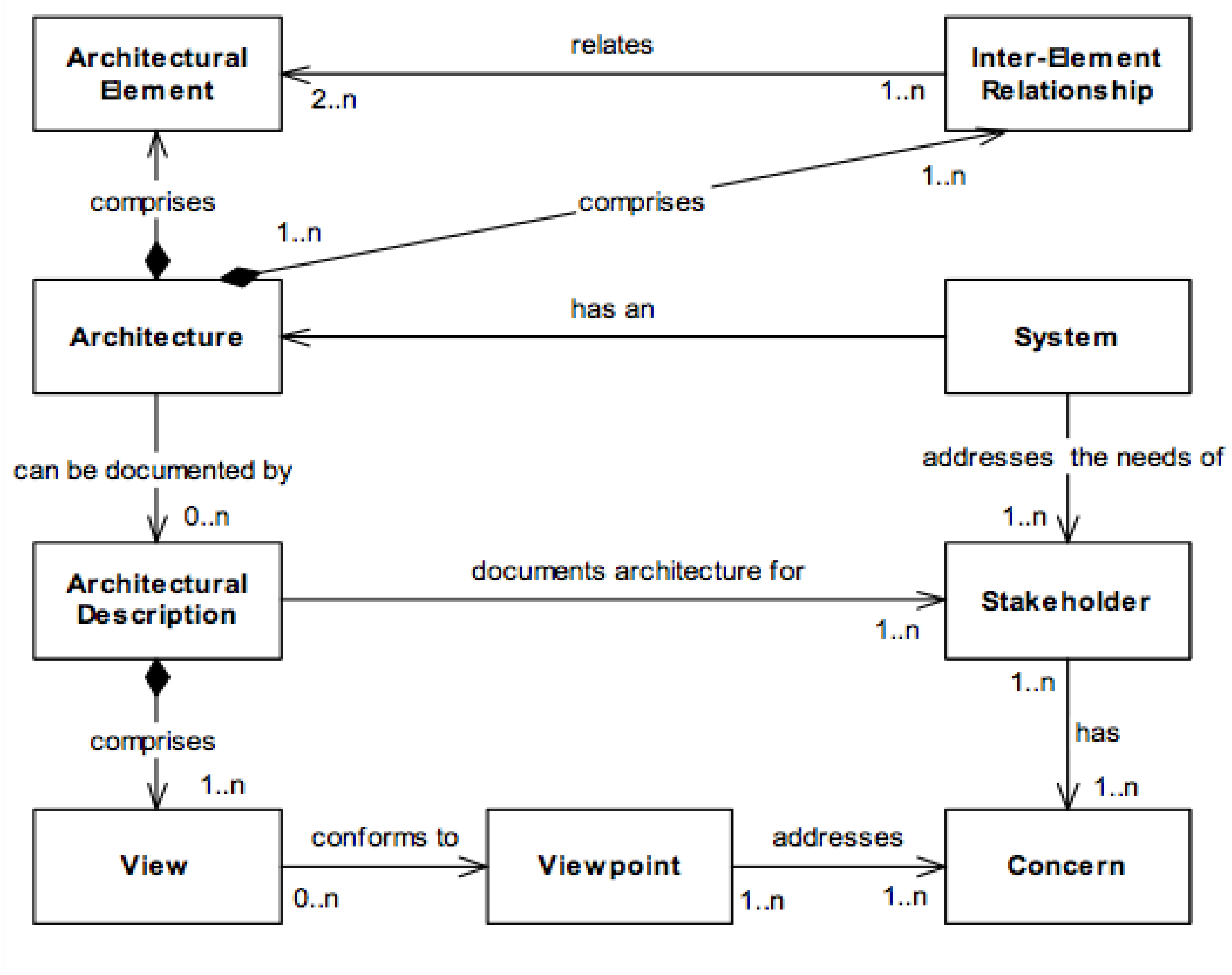
Software architecture standards

- The first standard in the field of software architectures is **ANSI/IEEE 1471-2000: Recommended Practice for Architecture Description of Software-Intensive Systems**
  - better known as IEEE 1471 standards
- In fact it is a “recommended practice”
  - the “weakest” type of IEEE standards, whose adoption and interpretation are the responsibility of the using organization
- It has been adopted by ISO/IEC JTC1/SC7 as ISO/IEC 42010:2007, in 2007

IEEE 1471 standard

- It focuses on the description of an architecture as the concrete artifact representing the abstraction that is software architecture or system architecture.
- IEEE 1471's contributions lie in the following:
  - It provides definitions and a meta-model for the description of architecture
  - It states that an architecture exists to respond to specific stakeholder concerns about the software/system being described
  - It asserts that architecture descriptions are inherently multi-view, no single view captures all stakeholder concerns about an architecture
  - It separates the notion of view from viewpoint, where a viewpoint identifies the set of concerns and the representations/modeling techniques, etc used to describe the architecture to address those concerns
  - It establishes that a conforming architecture description has a 1-to-1 correspondence between its viewpoints and its views

IEEE 1471 conceptual model



Stakeholders and viewpoints

- ***Stakeholders*** are the people for (users) and with (developers) whom we build a system; they have complex, overlapping, and often conflicting needs
- ***Viewpoints*** are ways to structure the description of a system. Viewpoints guide the creation of the system architecture, depicted by a particular view (or set of views) and based on the principle of ***separation of concerns***

Stakeholders

- Customer
- User
- Project manager
- System engineer
- Software architect
- Developer
- Maintainer
- Tester
- Network administrator
- Quality assurance engineer
- …

Stakeholders and their concerns

Maintainer

End User

Customer

Sales

Dev Manager

Developer

Sys Admin

Functionality

Price

Dev Costs

On Time Delivery

Performance

Stability & Maintainability

Ease of Use

Ease of Debugging

Modifiability

Testability & Traceability

Structure & dependency between component

Ease of Installation

Ease of Integration

Architectural description

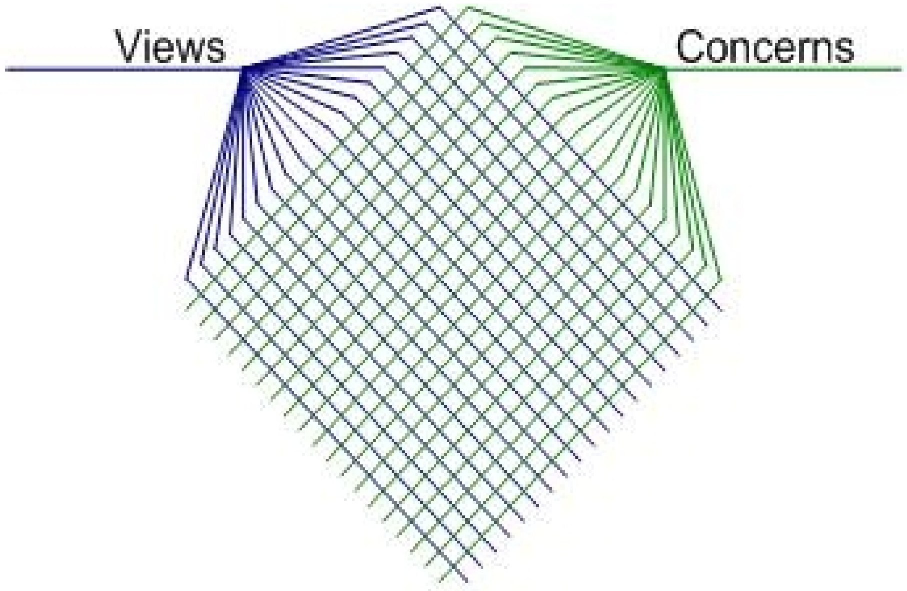
- An **architectural description** (AD) is a set of artifacts which collectively document an architecture in a way understandable by its stakeholders, and demonstrates that the architecture meets their concerns
- The artifacts in an AD include views, models, principles, contraints, etc., to present the essence of the architecture and its details, so that it can be validated and the described system can be built
- The AD can include also relevant information like business drivers, scope or requirements overview

Views

- A **view** is a description of a system according to the perspective (**viewpoint**) of some stakeholder, who has to satisfy some interest (**concern**)
  - **Example**: a user view describes the typical scenarios where a system can be used
- An **architectural view** is a description of some relevant issues of a software architecture
  - **Example**: the architectural view of packages necessary to install a software system, depicting their dependencies

Architectural views and concerns

Code Distribution
Data Storage
Data Transmission
Deployment
Function/Logic/Services
Events
Hardware
Network
System Interface
User Interface
Usage



Accuracy
Availability
Concurrency
Consumability
Customization Points
Environment (Green)
Internationalization
Layering/Partitioning
Maintenance
Operations
Quality
Performance
Regulations
Reliability
Reuse
Security
Serviceability
Support
Timeliness
Usability
Validation

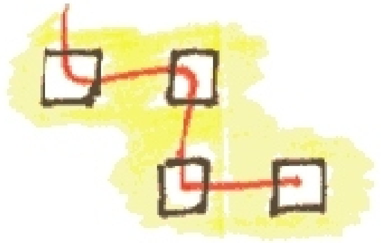
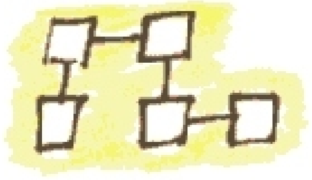
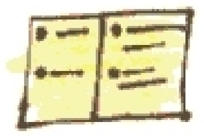
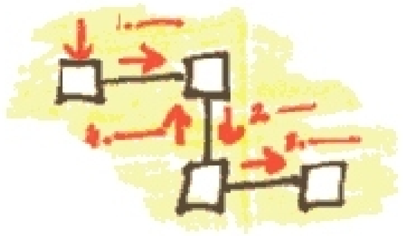
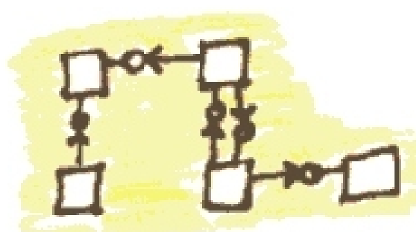
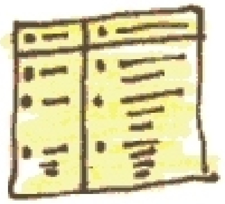
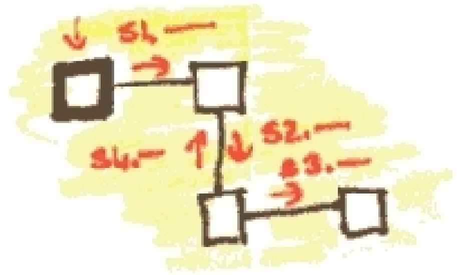
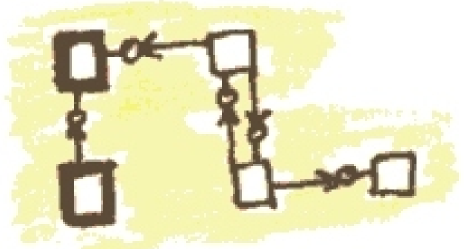
Views and stakeholders

	Module Views					C&C Views	Allocation Views				Other Documentation					
	Decomposition	Uses	Generalization	Layered	Data Model	Various	Deployment	Implementation	Install	Work Assignment	Interface Documentation	Context Diagrams	Mapping Between Views	Variability Guides	Analysis Results	Rationale and Constraints
Project managers	s	s		s			d			d		o				s
Members of development team	d	d	d	d	d	d	s	s	d		d	d	d	d		s
Testers and integrators	d	d	d	d	d	s	s	s	s		d	d	s	d		s
Designers of other systems					s						d	o				
Maintainers	d	d	d	d	d	d	s	s			d	d	d	d		d
Product-line application builders	d	d	s	o	s	s	s	s	s		s	d	s	d		s
Customers							o			o		o			s	
End users						s	s		o						s	
Analysts	d	d	s	d	d	s	d		s		d	d		s	d	s
Infrastructure support personnel	s	s			s	s	d	d	o					s		
New stakeholders	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Current and future architects	d	d	d	d	d	d	d	s	d	s	d	d	d	d	d	d

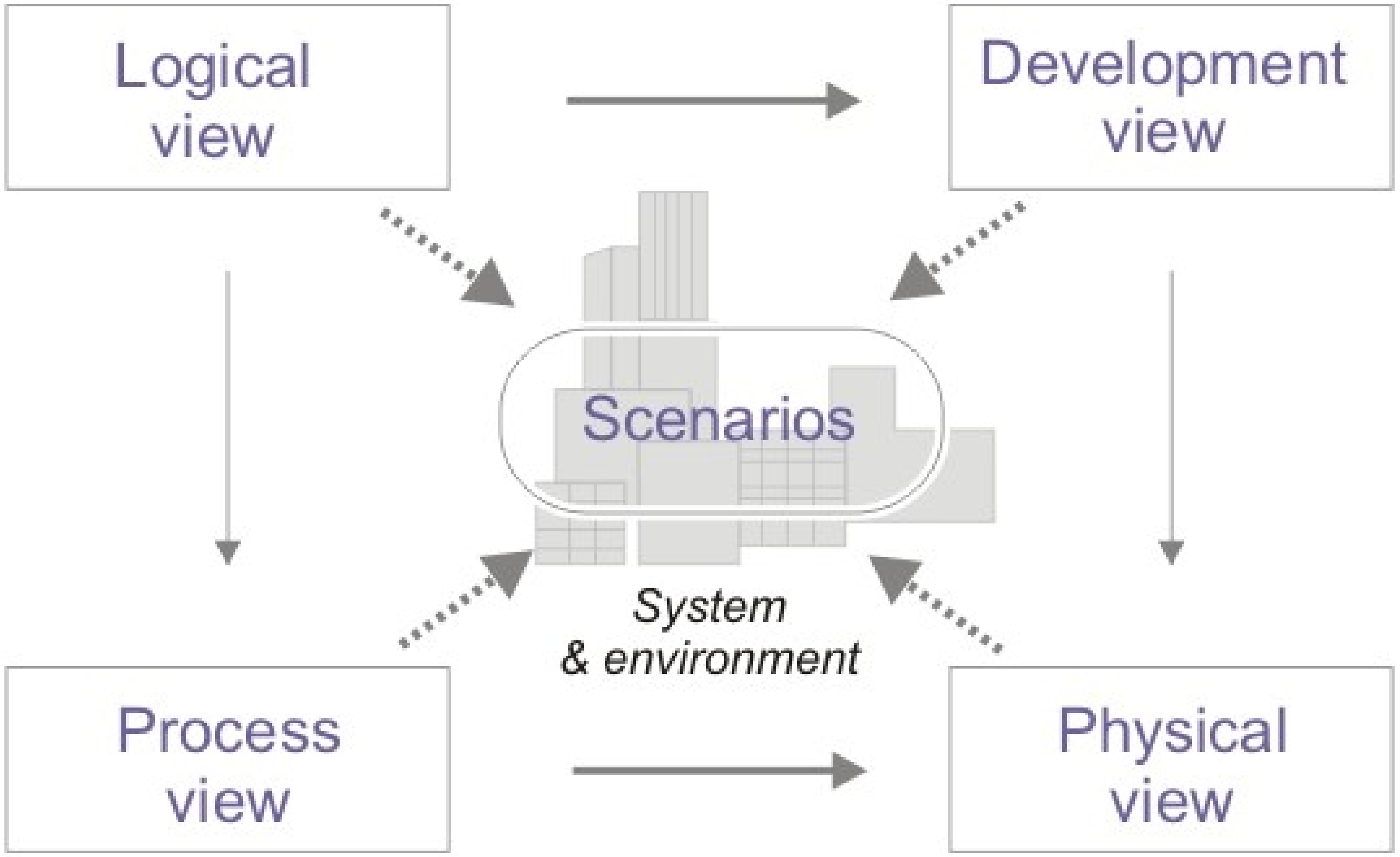
Key: d = detailed information, s = some details, o = overview information, x = anything

Example

- **Structure**: the source code of an operating system has a structure, that for instance separates the kernel (eg. concerning processes and scheduling) from the services (eg. concerning the file system)
- **Behavior**: the operation of an operating system like Unix can be described as a set of concurrent processes which can invoke system calls; each call can raise events which can suspend or activate processes; we can say that a running Unix system is made of processes coordinated by events

	Behavioral View	Structural View
Conceptual Architecture (abstract)	 Collaboration trace	 Architecture Diagram  Informal Component Specs (CRC-R)
Logical Architecture (detailed)	 Collaboration Diagrams	 Architecture Diagram with I/Fs  Interface Specs
Execution Architecture (Process View and Deployment View)	 Collaboration Diagrams showing processes	 Architecture Diagram showing Active Components

Views (Krutchen 4+1)



pkruchten.wordpress.com/architecture/

Views (Rozanski and Woods')

- **Functional** View: runtime elements which deliver functionality, including their responsibilities, interfaces and interactions
- **Information** View: how the architecture stores, manipulates, manages, and distributes information
- **Concurrency** View: state-related structure and constraints
- **Development** View: module organization and related tools
- **Deployment** View: physical environment in which the system runs
- **Operational** View: how the system will be operated, administered, and supported when it is running in its production environment

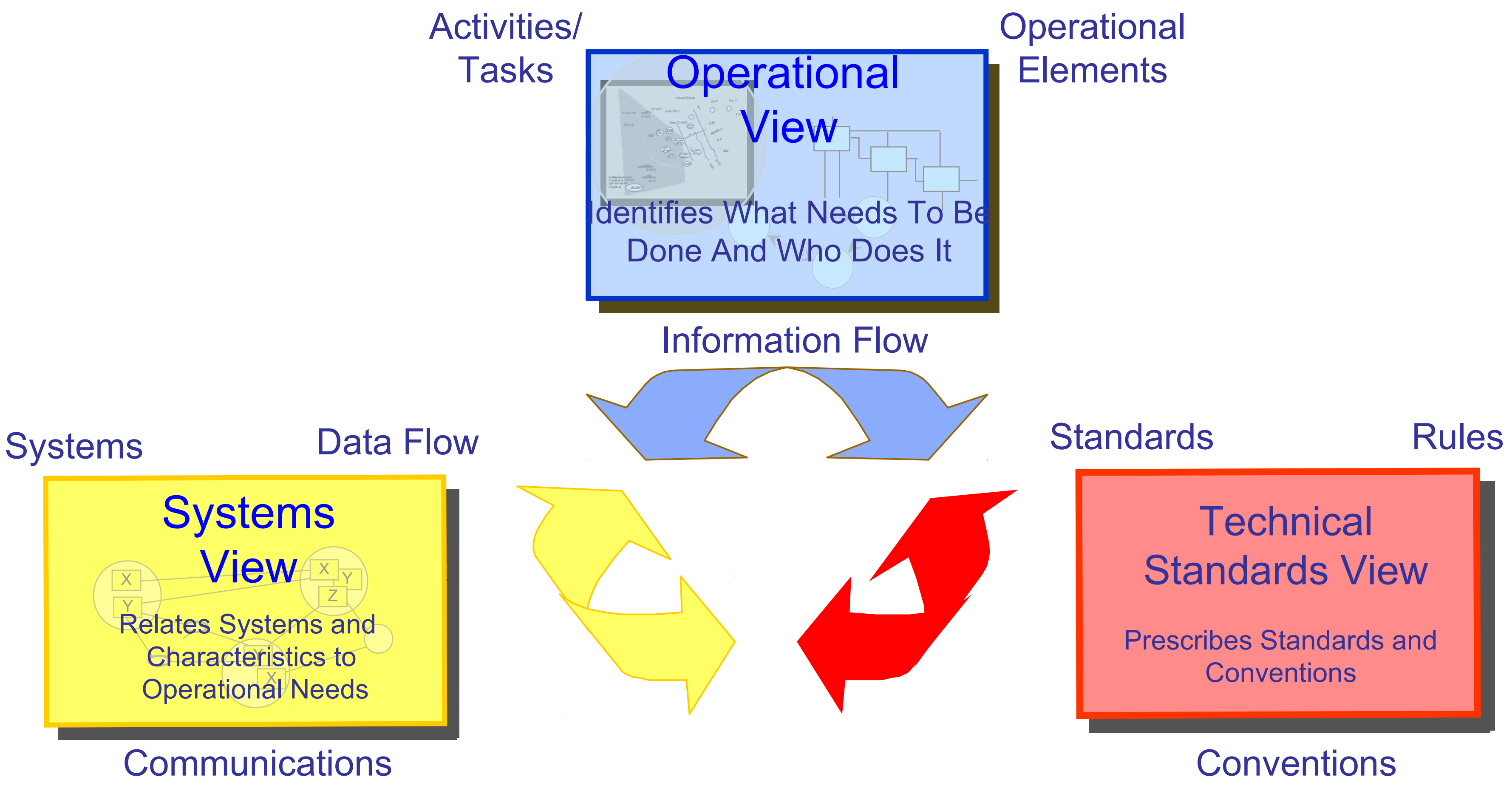
www.viewpoints-and-perspectives.info/

Views (Clements and others)

- Module views
- Components-and-connectors views
- Allocation views

wiki.sei.cmu.edu/sad

Views (DODAF)


Activities/
Tasks
Operational
Elements
Operational
View
Identifies What Needs To Be
Done And Who Does It
Information Flow
Systems
Data Flow
Standards
Rules
Systems
View
Relates Systems and
Characteristics to
Operational Needs
Technical
Standards View
Prescribes Standards and
Conventions
Communications
Conventions

DODAF views

- ***All View***: is the overarching information describing the architecture plans, scope, and definitions
- ***Operational View***: focuses on the behaviours and functions describing the mission aspects
- ***System View***: describes the system and applications supporting the mission functions
- ***Technical Standards View (TV)***: describes the policies, standards and constraints

UML diagrams for documentation

	Use Case View	Logical View		Physical View	Deployment V.
Focus	*Expression of requirements*	*Expressing Behavior*	*Representing Structure*	*Implementing Objects and Classes*	*Deployment of Executable Code*
Diagram	use case diagrams sequence diagrams collaboration diagrams	statechart diagrams	class diagrams object diagrams activity diagrams	component diagrams	deployment diagrams
Element	Actors Use cases Classes Collaborations	Objects Classes Collaborations Interactions Categories		Modules Subroutines Tasks Subsystems	Nodes Modules Main Programs

UML diagrams model a system from different perspectives

Architecture design principles

Defining software architecture – 1/2

- The definition of the architecture is the first high-level step of software design (i.e., architectural design)

Architecture = Design in the large

- Its first aim is the a structural decomposition of the system into sub-entities:
  - *Divide et impera* approach: to develop single entities in simpler than to develop the entire system
  - It allows to perform several development activities at the same time (i.e., different entities developed in parallel)
  - It significantly favors modifiability, reusability as well as the portability of the system

Defining software architecture – 2/2

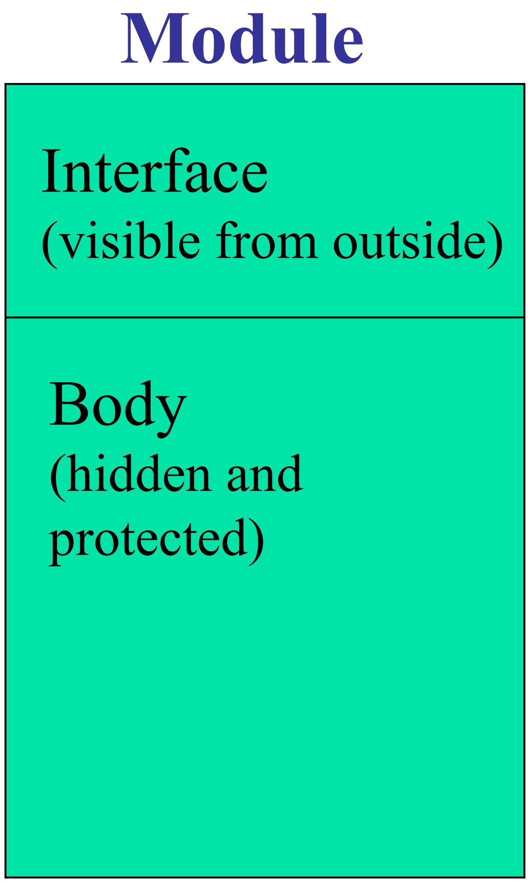
- The definition of the criteria for identification of sub-entities is the preliminary step :
  - A typical criterion is the <u>functional </u>one, i.e., sub-entities may be identified by mapping software functionalities to its parts:
    - E.g., user interfaces, database access, security management…
- Among the key principles for the architecture definition there are (must be!) **High Cohesion** and **Low Coupling** among components
  - Each sub-system has to contain homogeneous modules (e.g., modules providing services which are strictly related one to each other),
  - Minimize interactions between subsystems

Key architectural design principles

- Abstraction
- Modularity
- Simplicity
- Separation of concerns
- Postponing decisions
- Encapsulation
- Information hiding
- Clear interface design
- High cohesion
- Low coupling

Software module

- A **module** is a component that:
  - Provides an abstraction
  - Has a clear separation between:
    - Interface
    - Body
- The **interface** specifies "what" the module is (the abstraction provided) and "how" it can be used
  - The interface is the *contract* between the user and the provider of the abstraction
- The **body** describes "how" the abstraction is implemented

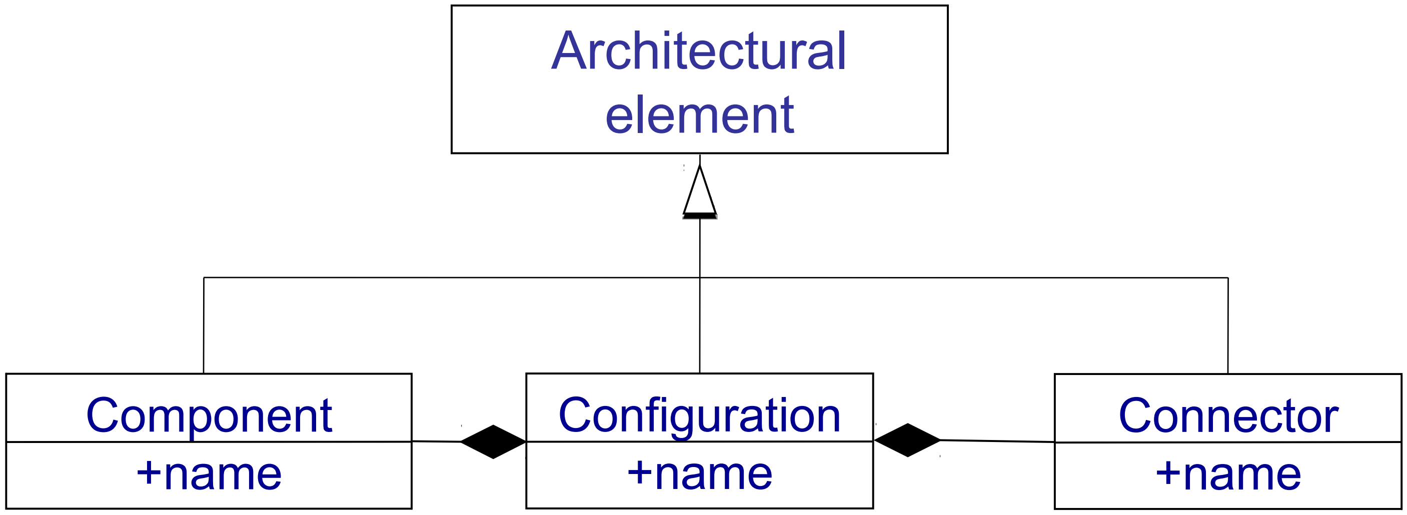


Software component

- A component is a run-time entity, and usually offers no visibility into the implementation structure
- Thus, the concept of module is more abstract, while the concept of component is more physical, however they overlap
- In many architectures there is a one-to-one mapping between modules and components
- We could consider components as modules inserted in an architecture, however a single module can turn into several components, and a single component can correspond to many modules

Main elements of a sw architecture

- Components
- Connectors
- Configurations



Component

- A software component is a **unit of composition** with contractually specified interfaces (including some *ports*) and explicit *dependencies*
- Example: a web server
- A software component can be deployed independently and is subject to composition by third parties
- A *component model* defines rules (standards) for naming, meta data, behavior specification, implementation, interoperability, customization, composition, and deployment of components
- Example: the CORBA Component Model (CCM)

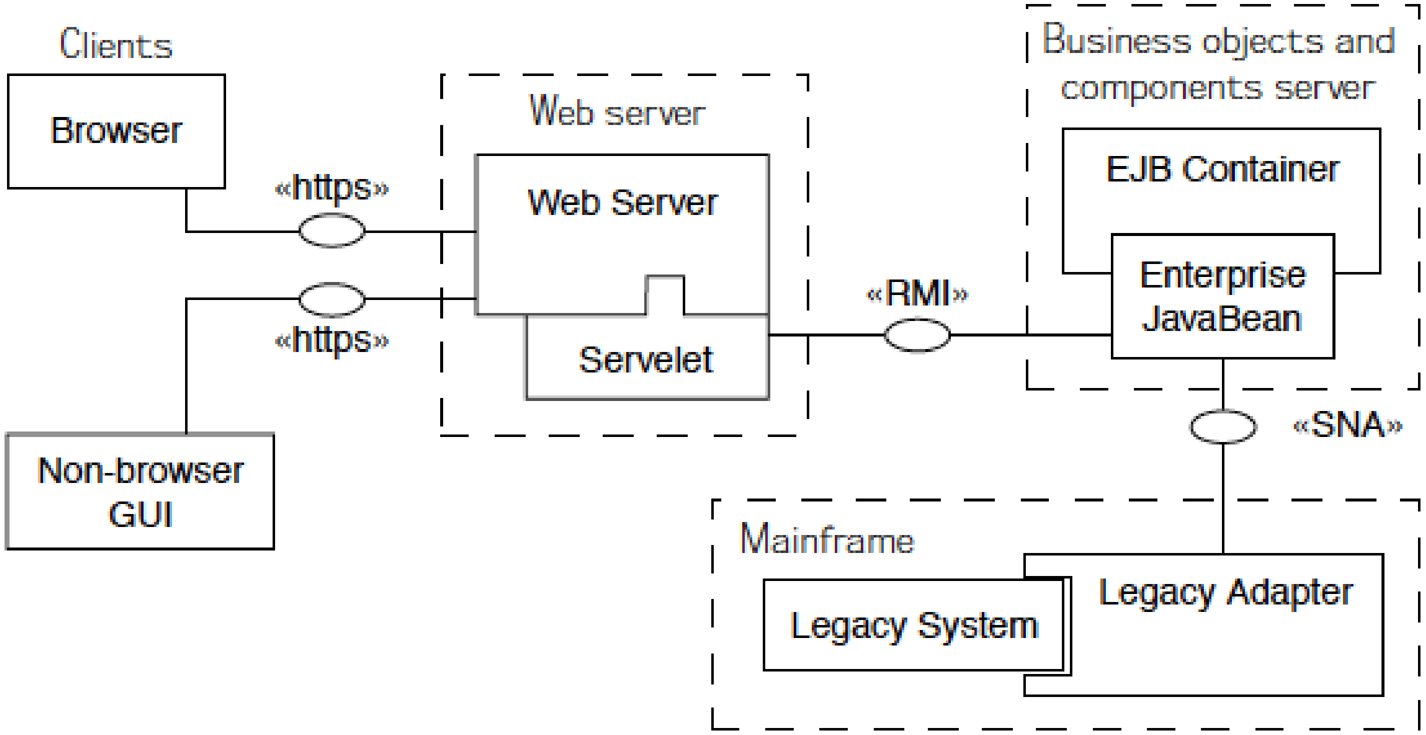
Connector

- A software connector specifies the mechanisms by which components transfer control or data
- Examples: procedure call, protocol, pipe, repository

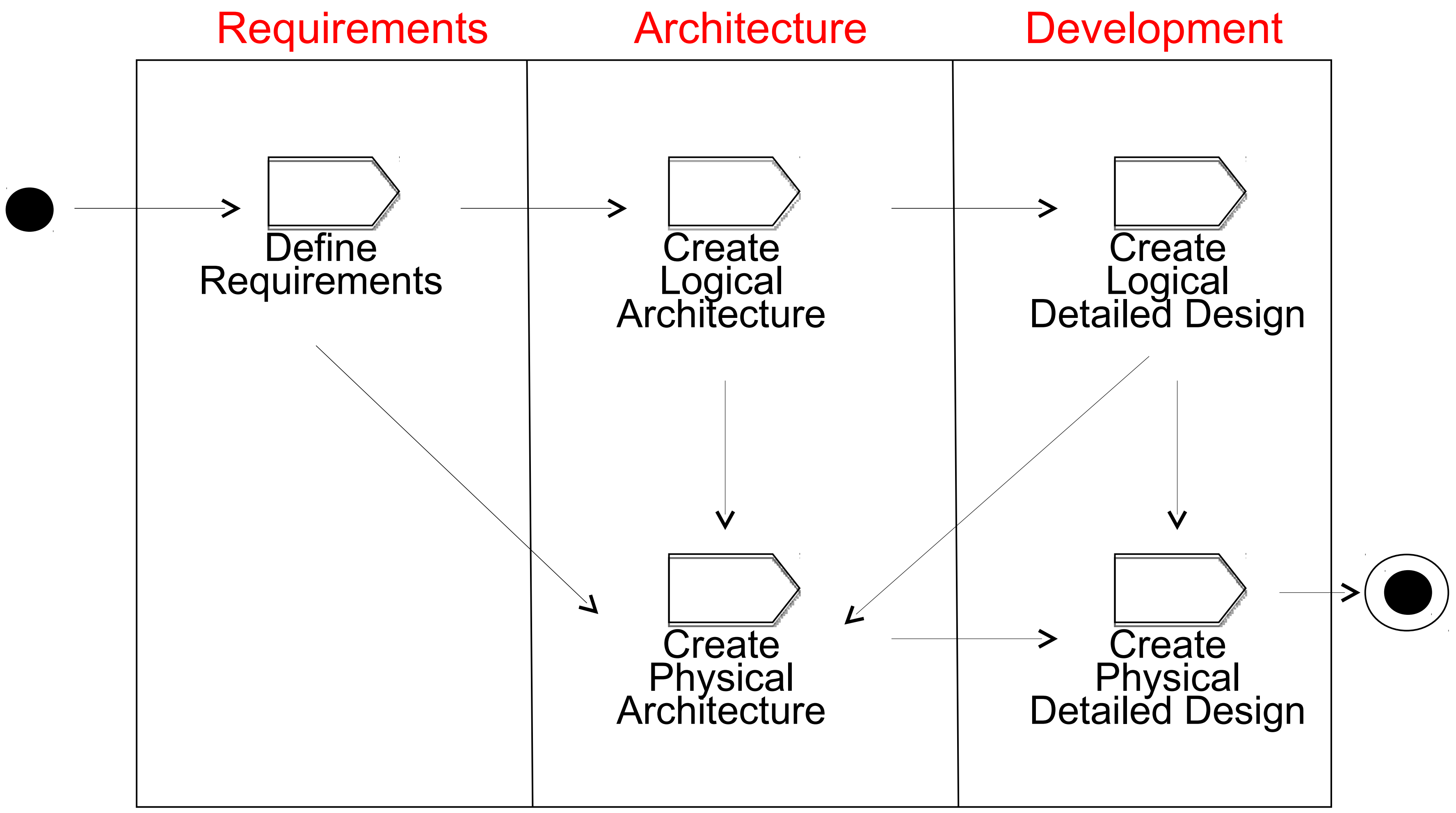
- A connector defines the possible interactions among components: the interfaces of connectors are called **roles**, which are attached to ports of components

Architectural configuration

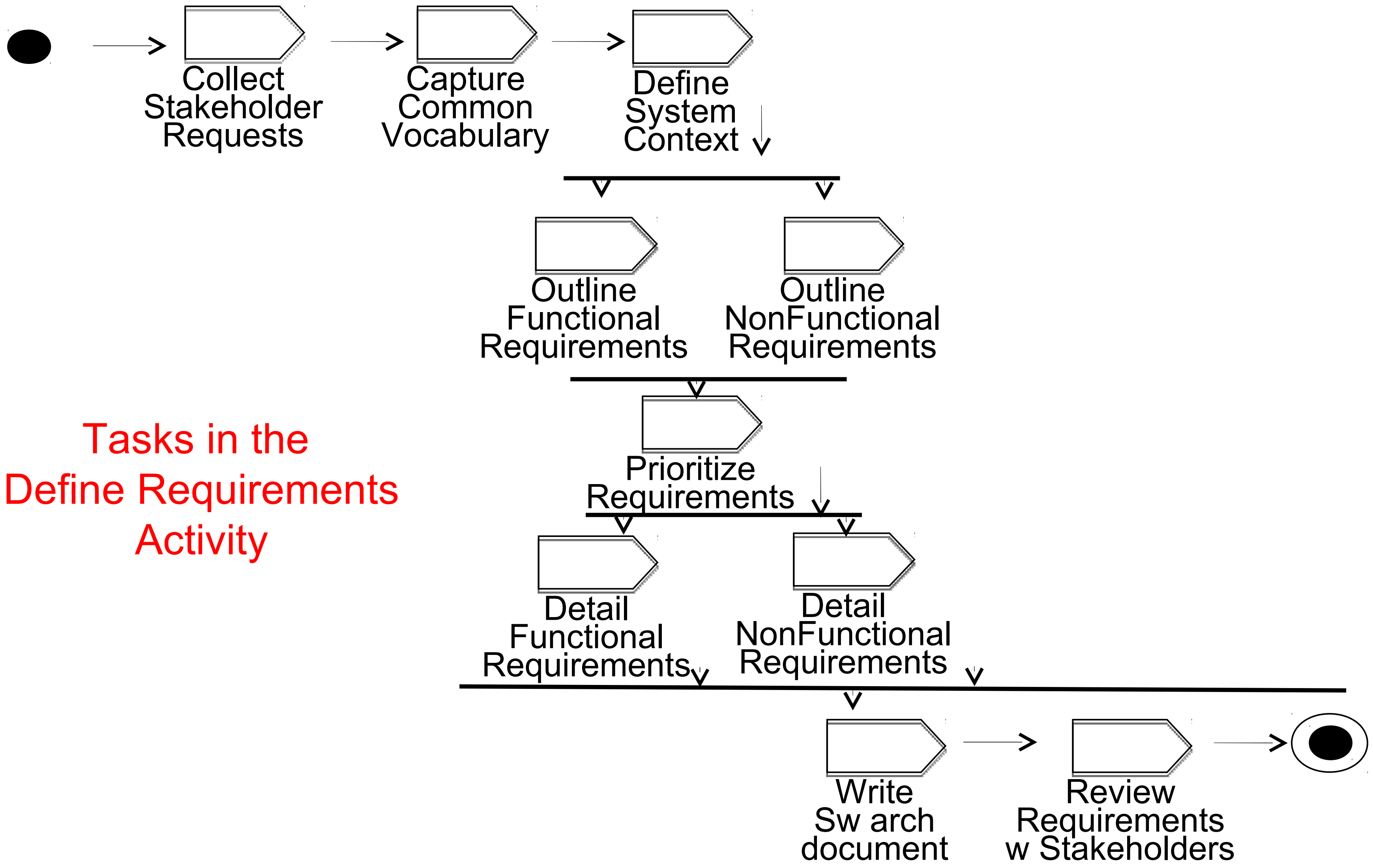
- An architectural configuration is a connected graph of components and connectors that describes basic configuration aspects of a software system
- Example:



A software architecting process
(Eeles & Cripps)


Requirements
Architecture
Development
Define Requirements
Create Logical Architecture
Create Logical Detailed Design
Create Physical Architecture
Create Physical Detailed Design

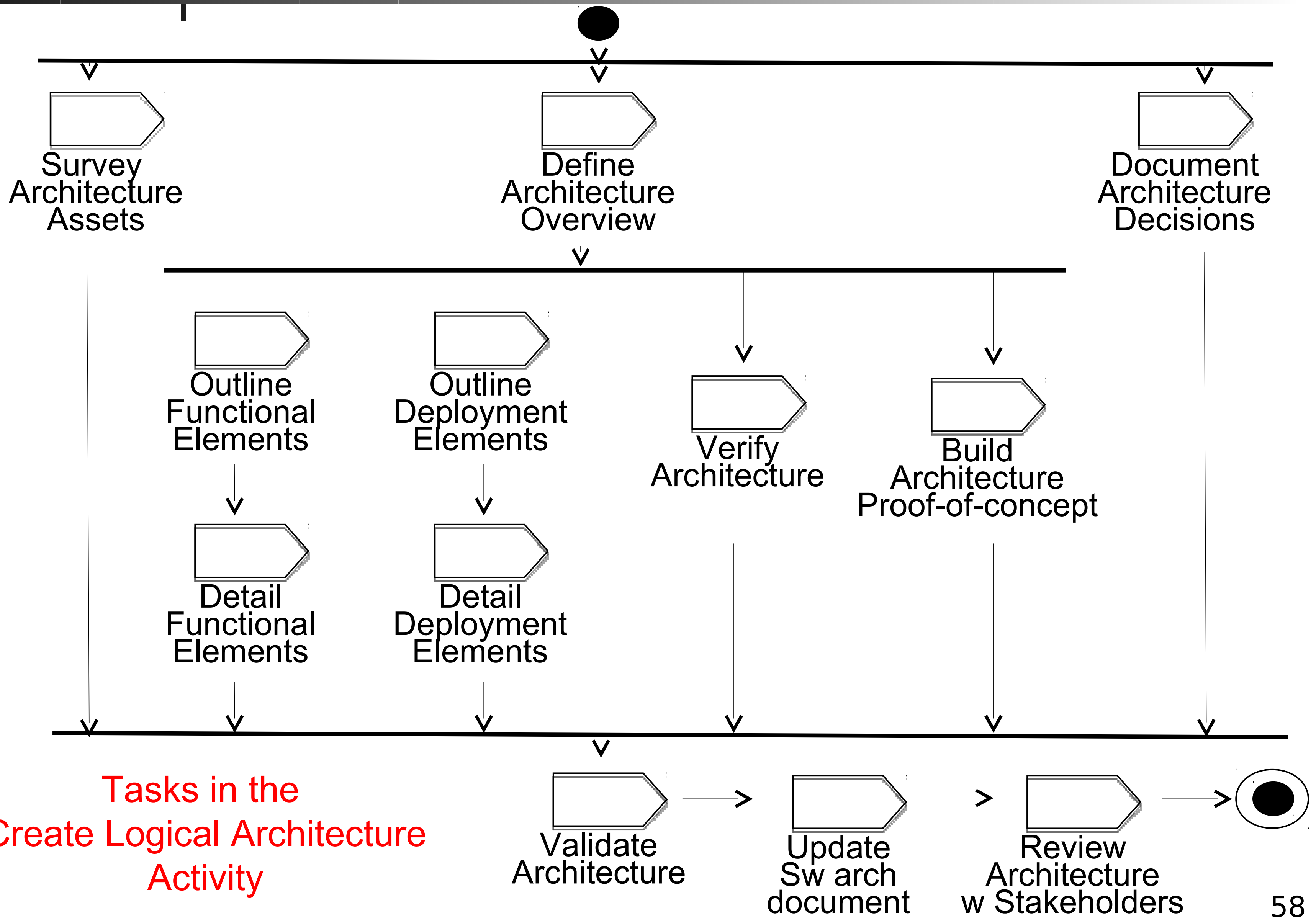
A software architecting process
(Eeles & Cripps)


Collect Stakeholder Requests
Capture Common Vocabulary
Define System Context
Outline Functional Requirements
Outline NonFunctional Requirements
Prioritize Requirements
Detail Functional Requirements
Detail NonFunctional Requirements
Write Sw arch document
Review Requirements w Stakeholders

Tasks in the Define Requirements Activity

A software architecting process

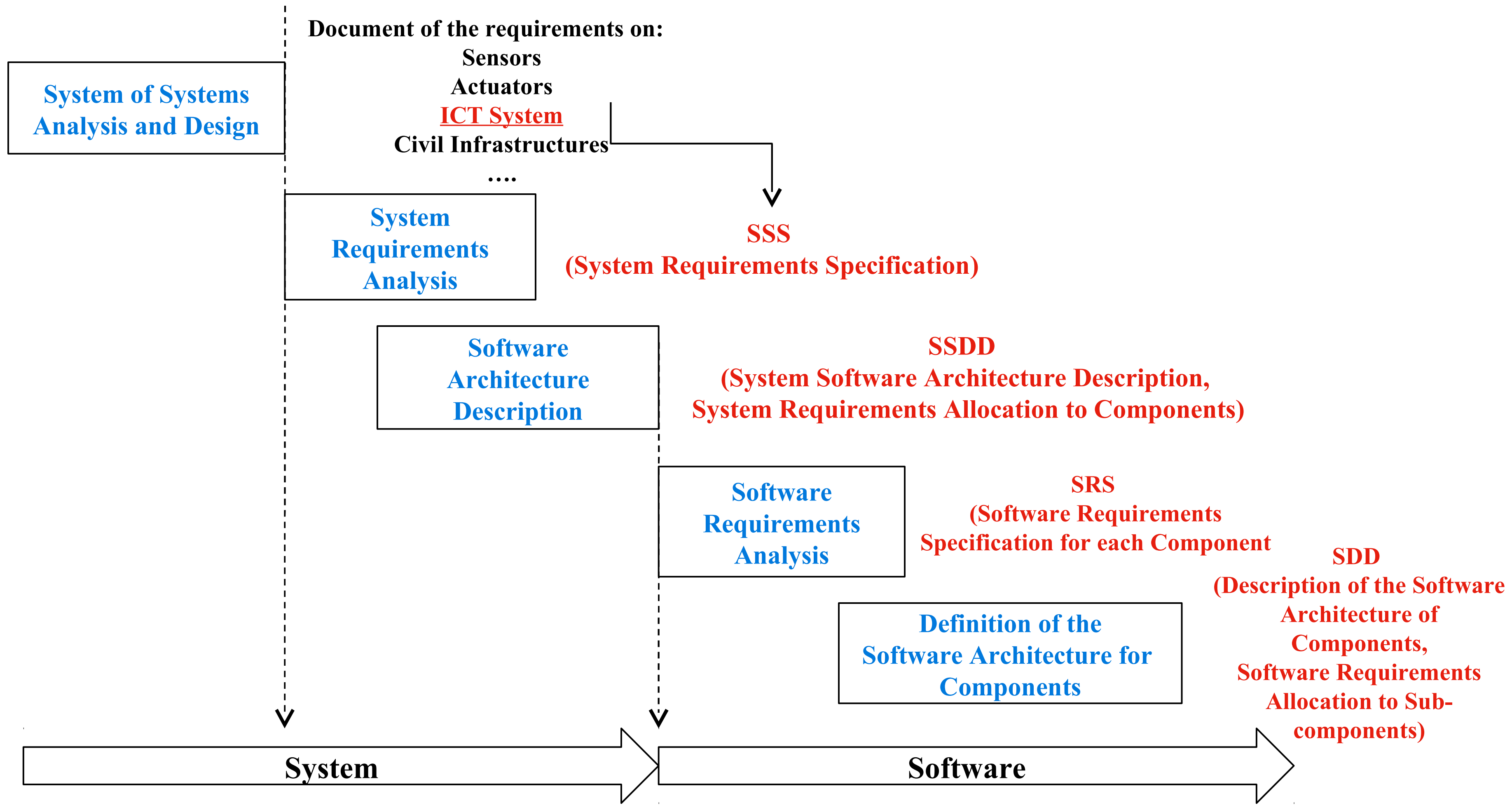
(Eeles & Cripps)



Survey Architecture Assets
Define Architecture Overview
Document Architecture Decisions
Outline Functional Elements
Outline Deployment Elements
Verify Architecture
Build Architecture Proof-of-concept
Detail Functional Elements
Detail Deployment Elements
Validate Architecture
Update Sw arch document
Review Architecture w Stakeholders

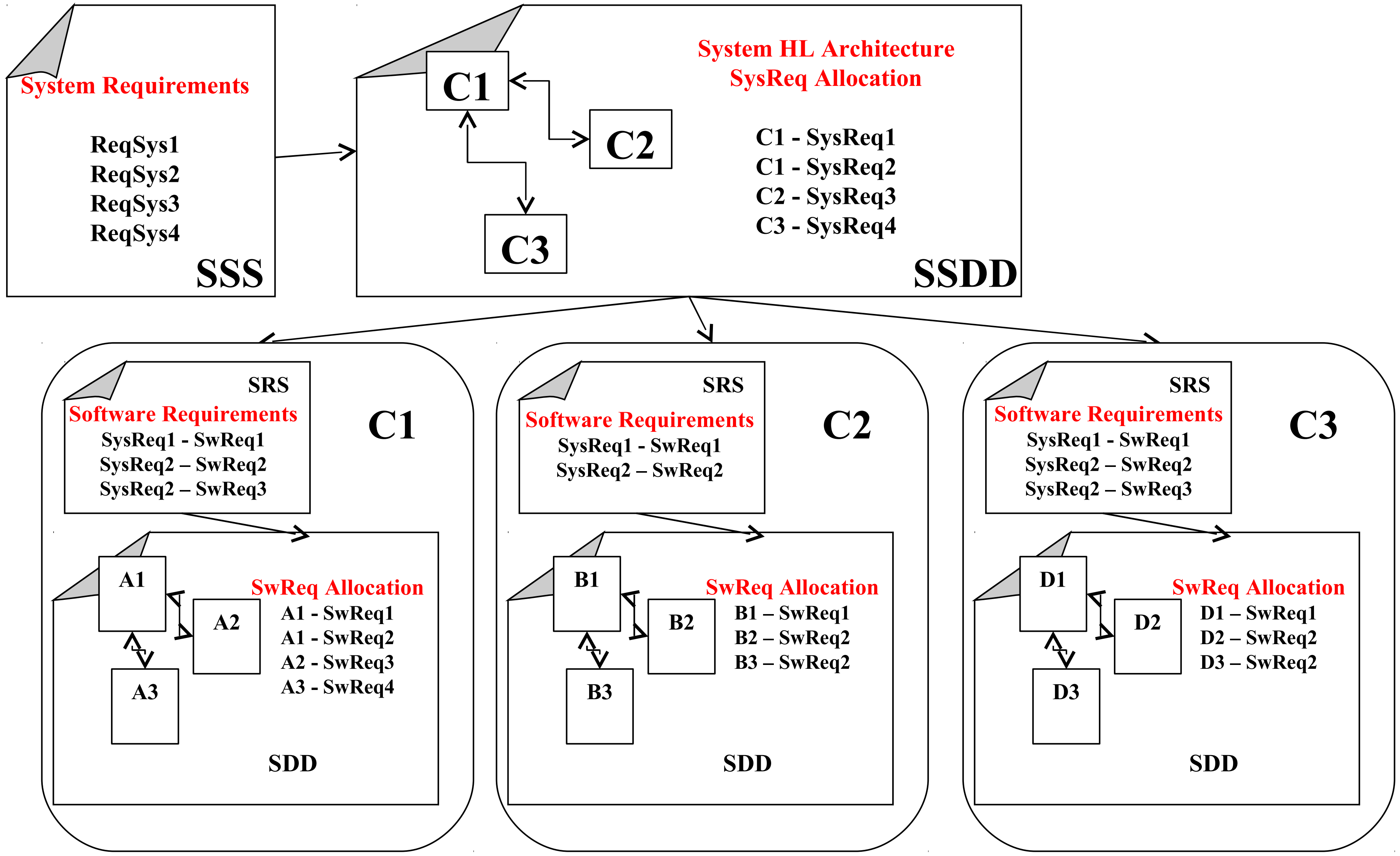
Tasks in the
Create Logical Architecture
Activity

A typical industrial (sub)process – 1/3


Document of the requirements on:
Sensors
Actuators
ICT System
Civil Infrastructures
....
System of Systems Analysis and Design
System Requirements Analysis
SSS
(System Requirements Specification)
Software Architecture Description
SSDD
(System Software Architecture Description, System Requirements Allocation to Components)
Software Requirements Analysis
SRS
(Software Requirements Specification for each Component
Definition of the Software Architecture for Components
SDD
(Description of the Software Architecture of Components, Software Requirements Allocation to Sub-components)
System
Software

Number of requirements ~ 1000 **Number of requirements ~ 10000**

A typical industrial (sub)process – 2/3



A typical industrial (sub)process – 3/3

Decomposition -> Which components in the system?
Assembly -> How are components structurally related?
Delegation -> Which components are delegated the external interfaces to?

SSDD

3. …..

4. System Architecture Design
   - 4.1 System components
   - …
   - 4.2 Relationships among components
   - ….
   - 4.3 Interfaces design
   - ….

5. ….

DECOMPOSITION

ASSEMBLY

DELEGATION

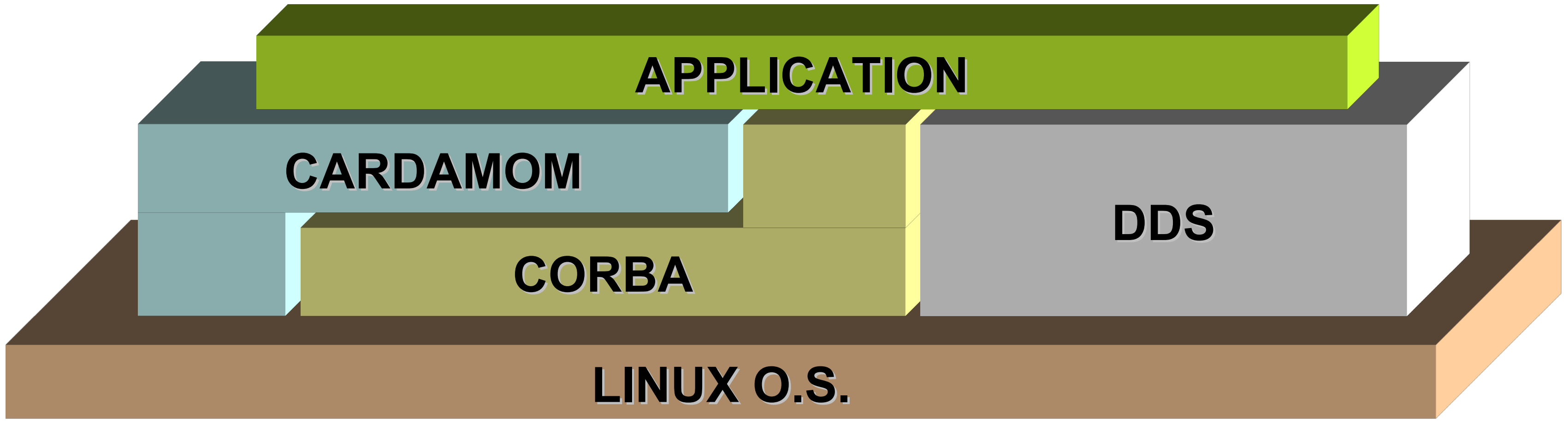
ACTORS

INTERFACES

Example

Example: Air Traffic Control system

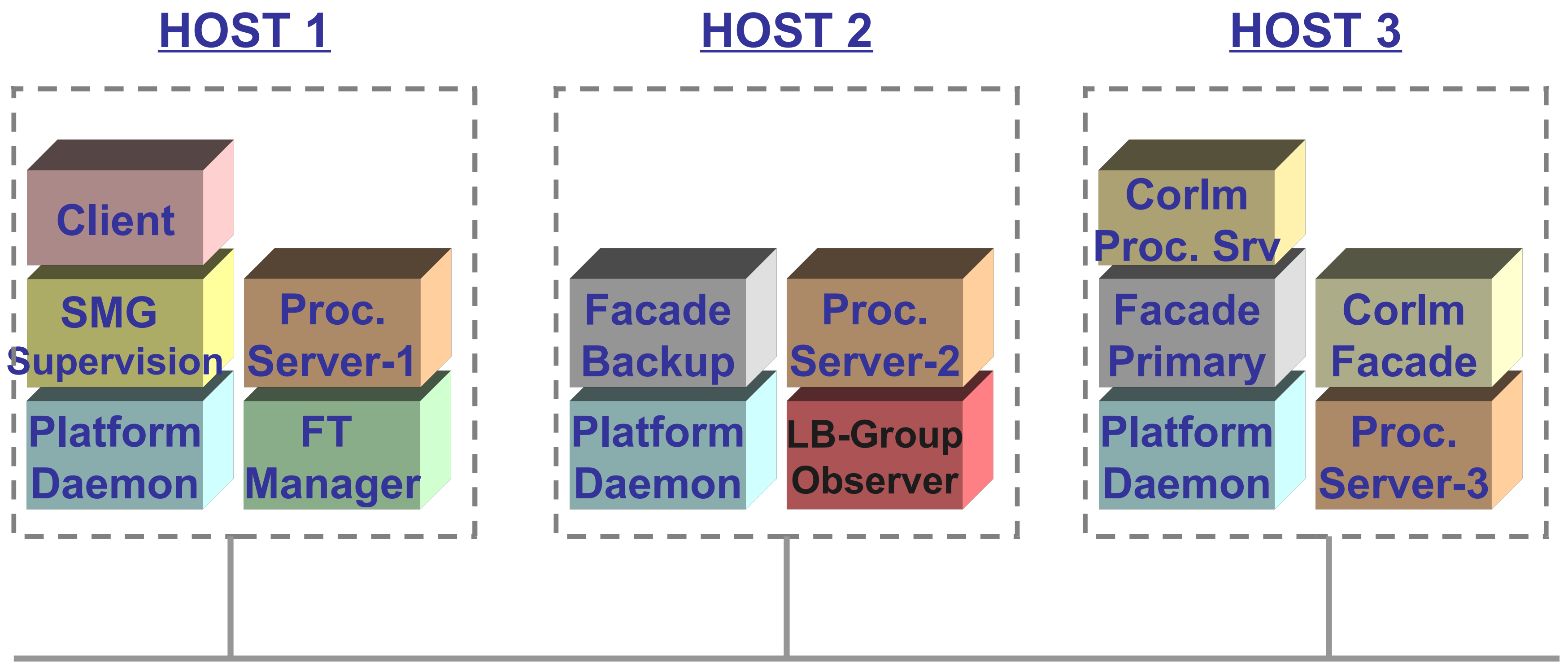
- The scenario provides an example of a real ATC system;
- Layered software organization



The middleware layer

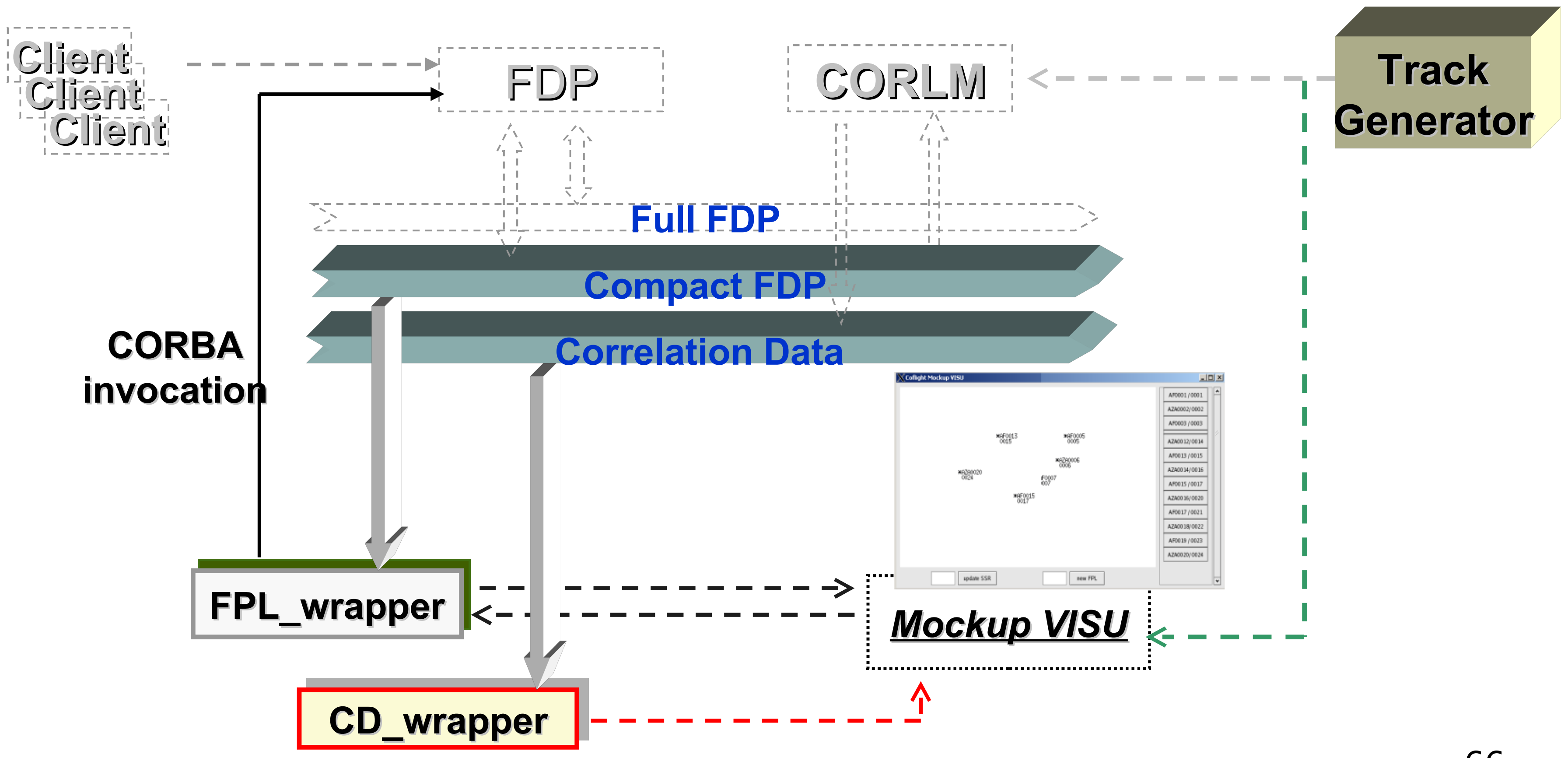
- The application uses the following underlying layers:
  - CARDAMOM services:
    - System Management, Load Balancing, Fault Tolerance, …
  - CORBA
    - Interaction mean between remote objects
  - DDS
    - Data exchange according to the publish-subcribe model

A sample 3-host deployment

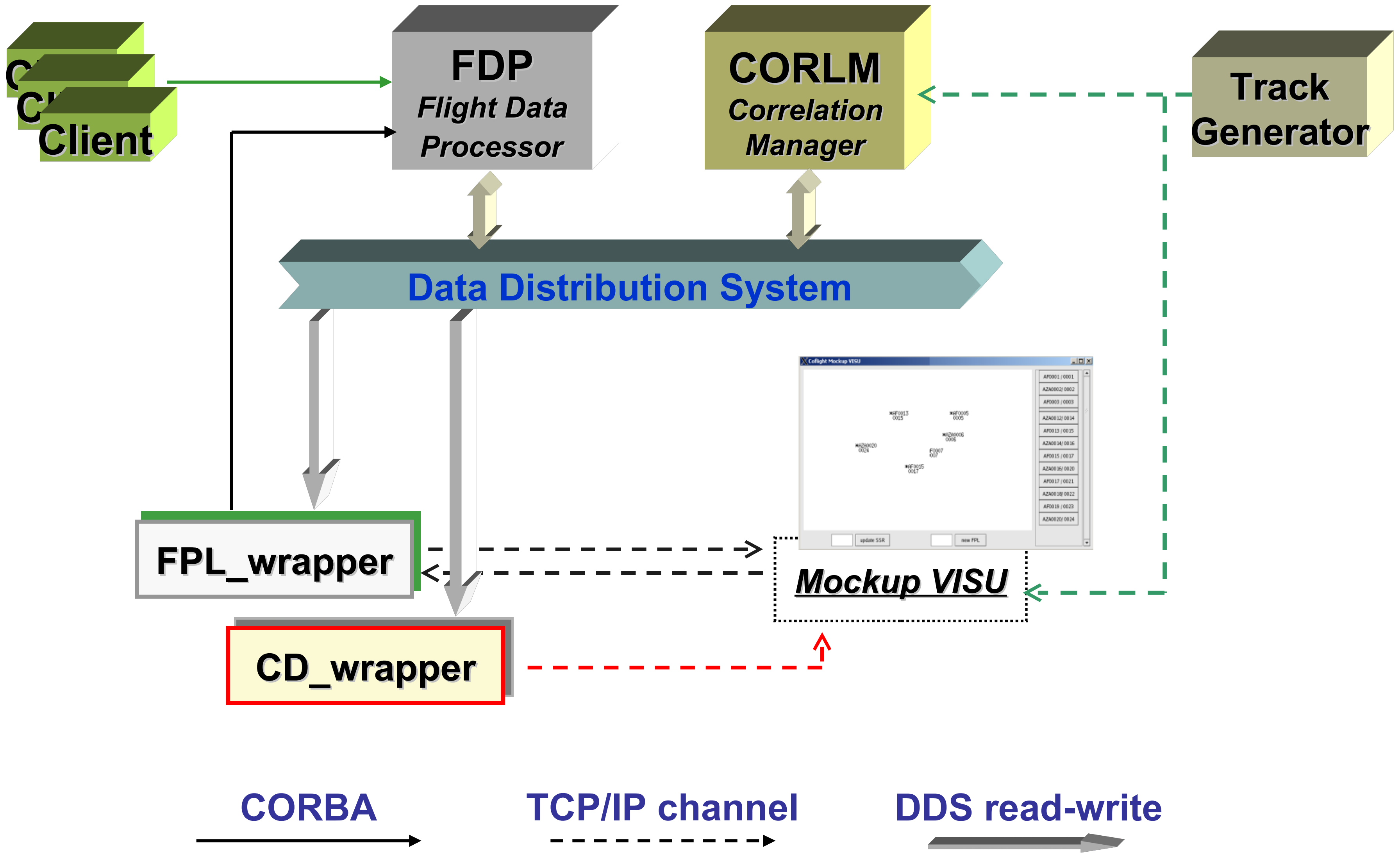


Adding external components

- External components are *de-coupled* from the system using the DDS



System overview

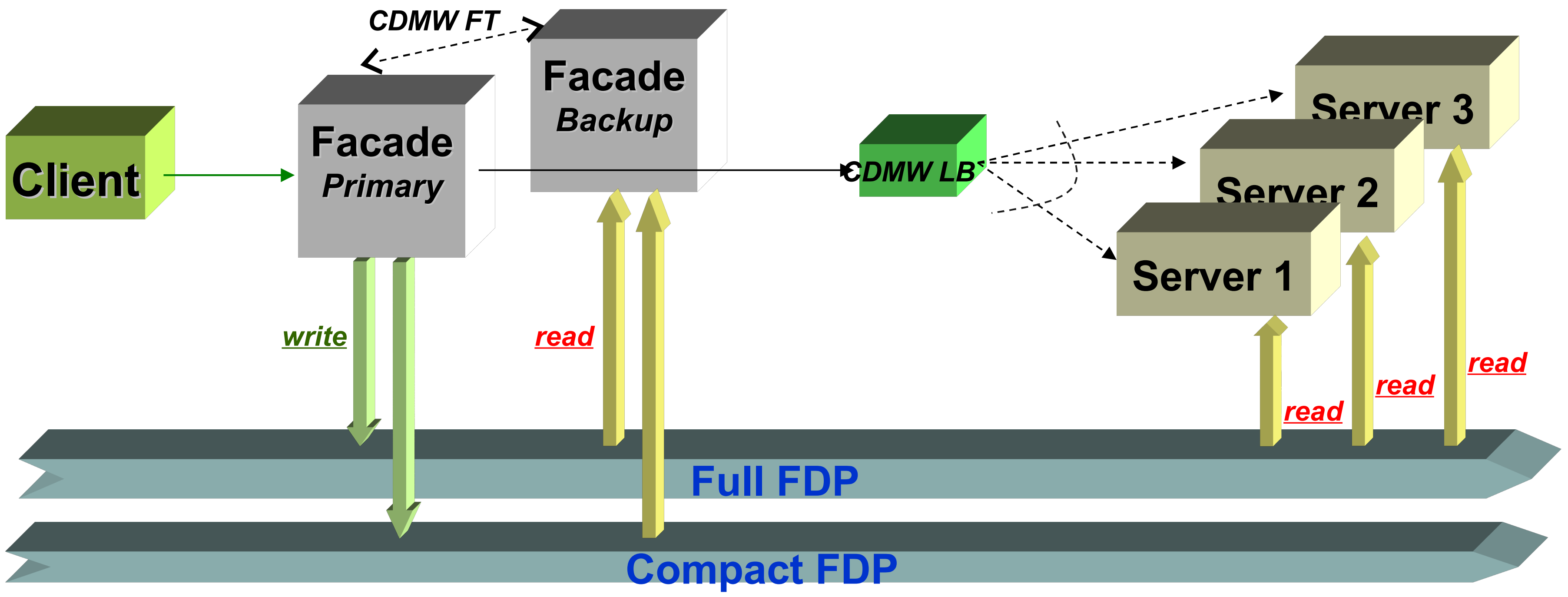


Flight information processing

- FDP – *Flight Data Processor*
  - Manages several flight-plan instances
  - Provides methods to insert, delete, update them
- CORLM – *Correlation Manager*
  - Correlates the FPLs with the radar information (simulated via the track-generator component)
  - Generates the Correlation Data

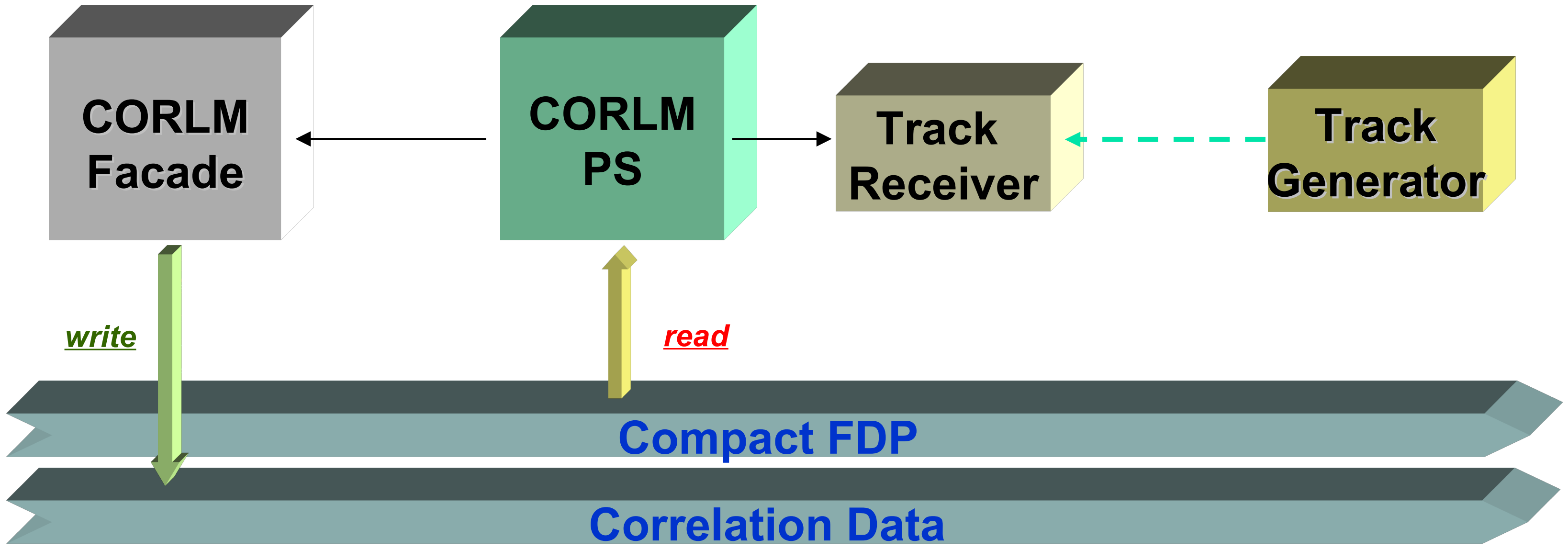
FDP logical organization

- 2 Facade servers replicated via *CDMW Fault Tolerance*
- 3 Processing servers under *CDMW Load Balancing*



CORLM logical organization

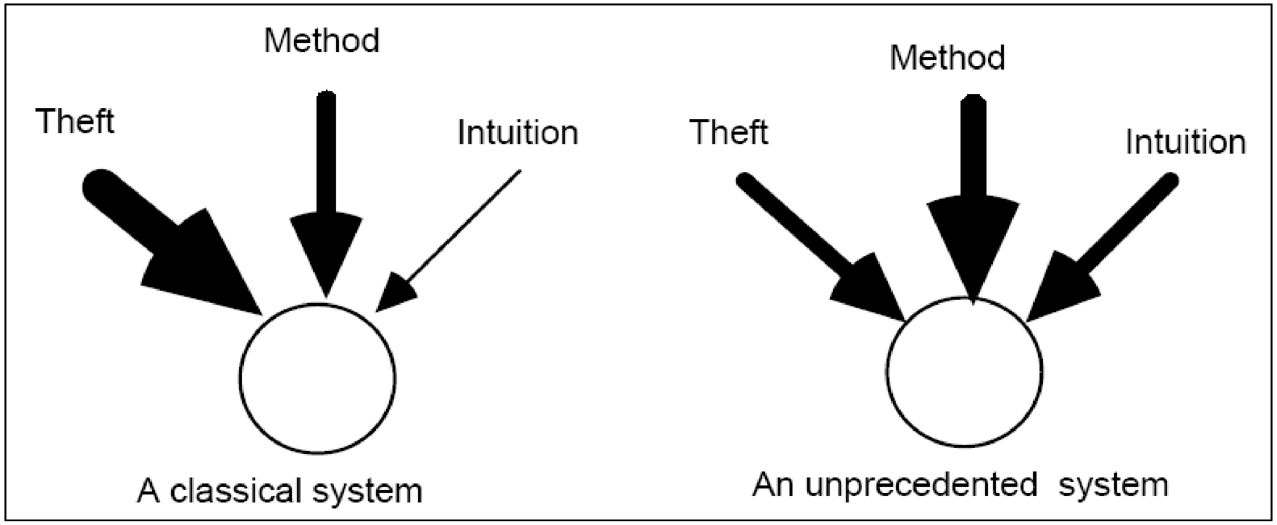
- Track Receiver: manages the radar output (may be simulated with the Track Generator)
- Corlm PS and Corlm Facade



Architectural assets

Sources of Architecture

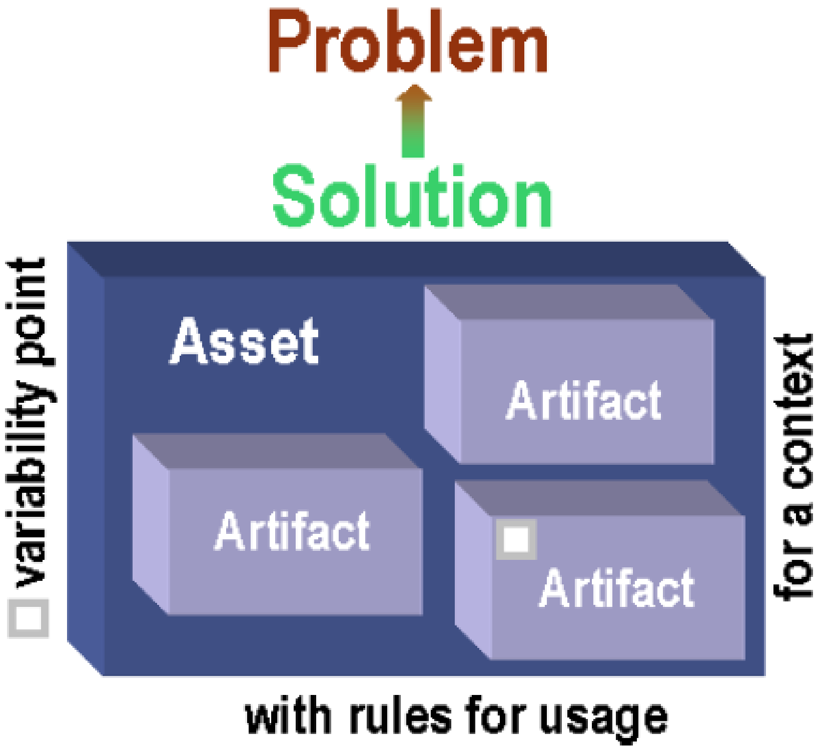
- Theft
  - From a previous system or from technical literature
- Method
  - An approach to deriving the architecture from the requirements
- Intuition
  - The experience of the architect



From "*Mommy, Where Do Software Architectures Come From?*", Philippe Kruchten
1st International Workshop on Architectures for Software Systems, Seattle, 1995

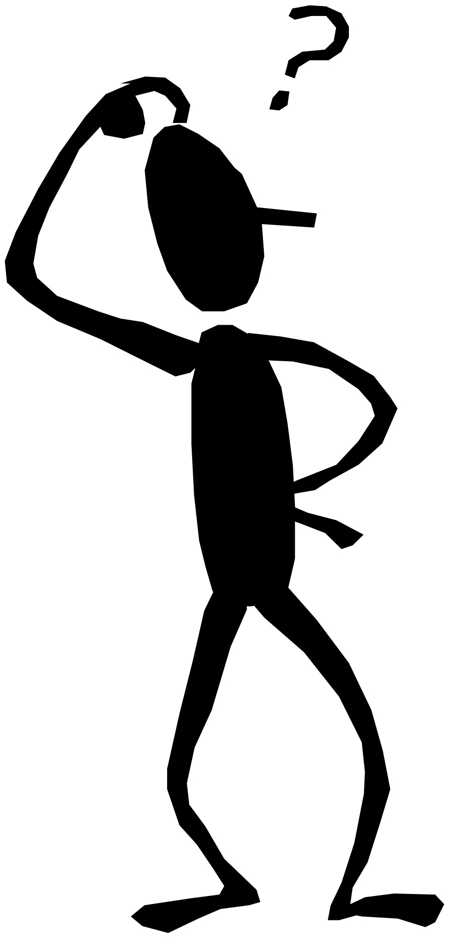
Software asset

- A (reusable) software asset is a set of artifacts providing a solution to a problem in a given context
- An asset may have a variability point, which is a part of the asset that may have a value provided or customized by the asset consumer
- The asset has rules for usage which are the instructions describing how the asset should be used



www.omg.org/spec/RAS

What Types of Architectural Assets are there?



- Reference Architecture
- Design Pattern
- Legacy Application
- Architectural Mechanism
- Pattern Language
- Packaged Application
- Development Method
- Reference Model
- Architectural Decision
- Programming Pattern
- Pattern
- Component Library
- Component
- Architectural Pattern
- Architectural Style
- Application Framework

Pattern

- *[A pattern is] a common solution to a common problem in a given context.* [UML User Guide]
- Pattern types
  - Architectural Patterns
    - Distribution patterns
    - Security Patterns
    - …
  - Design Patterns
  - Programming Patterns
  - Requirements Patterns
  - Testing Patterns
  - Project Management Patterns
  - Process Patterns
  - Organizational Patterns
  - …

Architectural Pattern

- *An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.* [Buschmann]
- Example:

Pattern: Layers

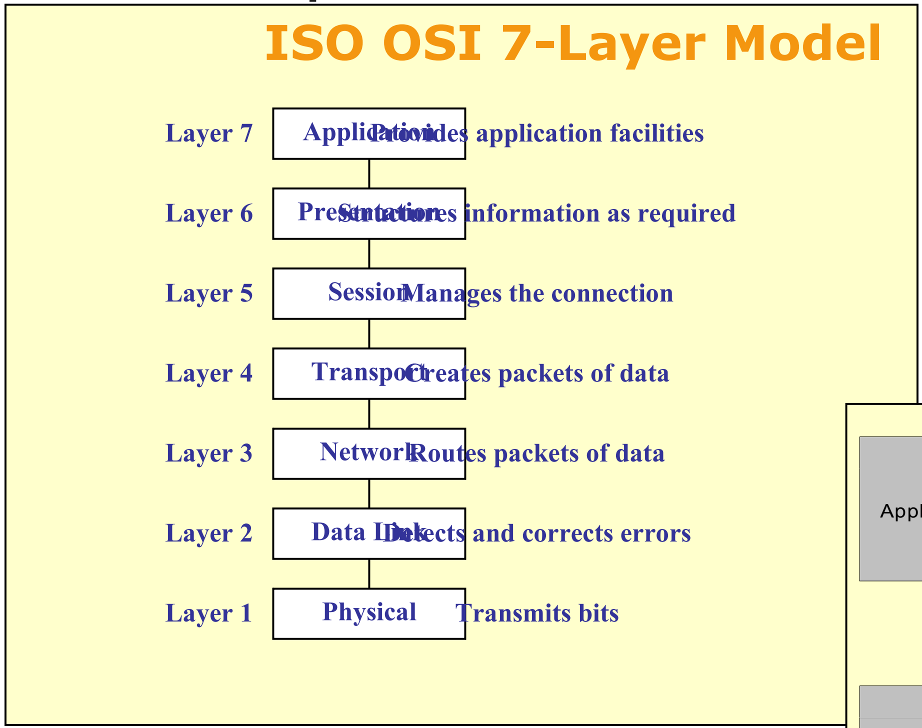
Context
A system that requires decomposition

Problem
High-level elements rely on lower-level elements and the following forces must be balanced:

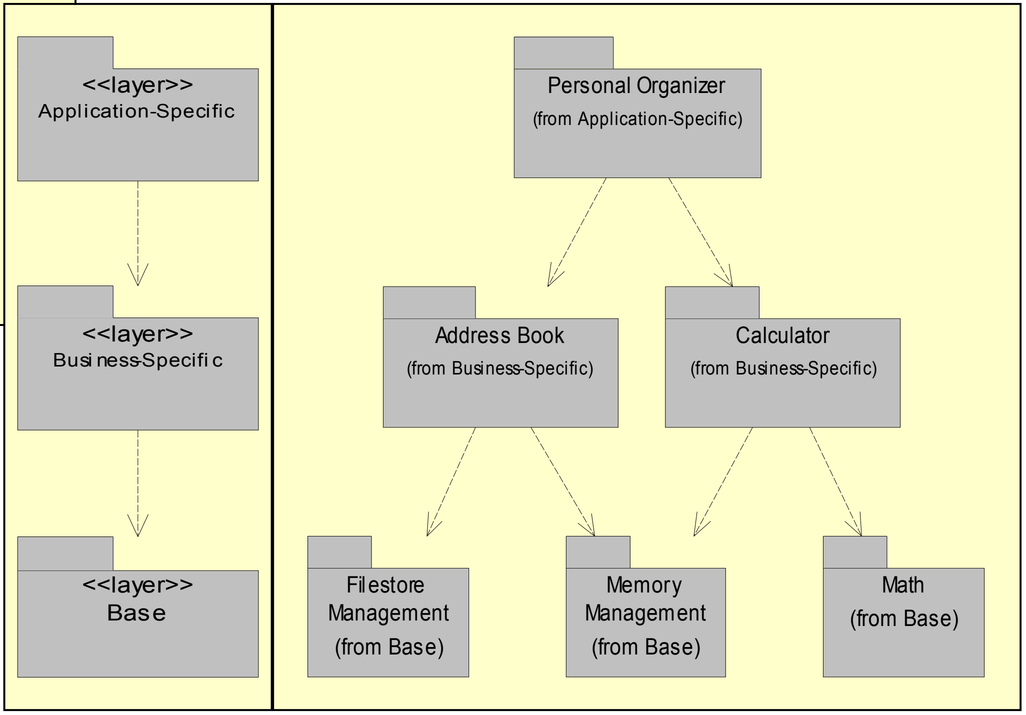
- Interfaces should be stable
- Parts of the system should be exchangeable
- Source code changes should not ripple through the system

Solution
Structure the system into layers

Architectural pattern – Layers

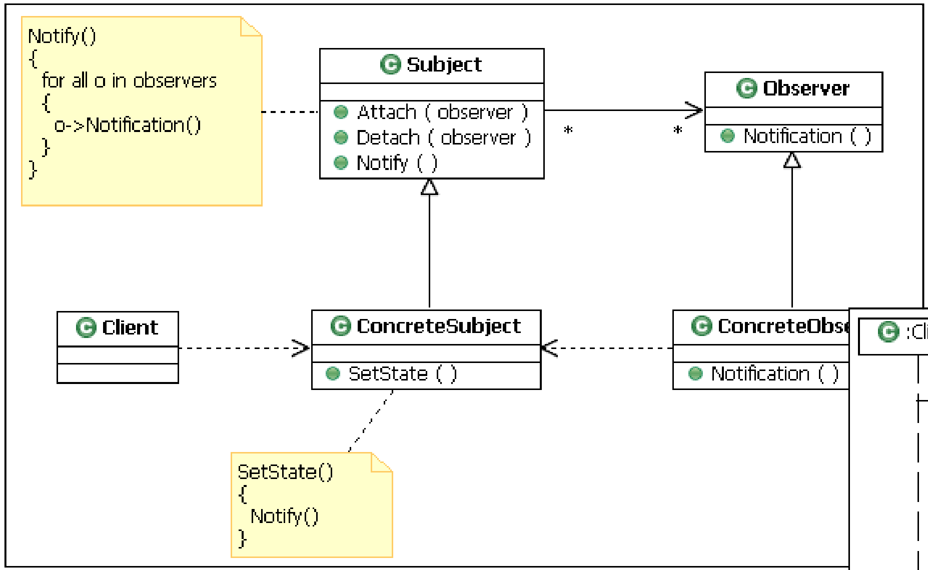


Personal Organizer

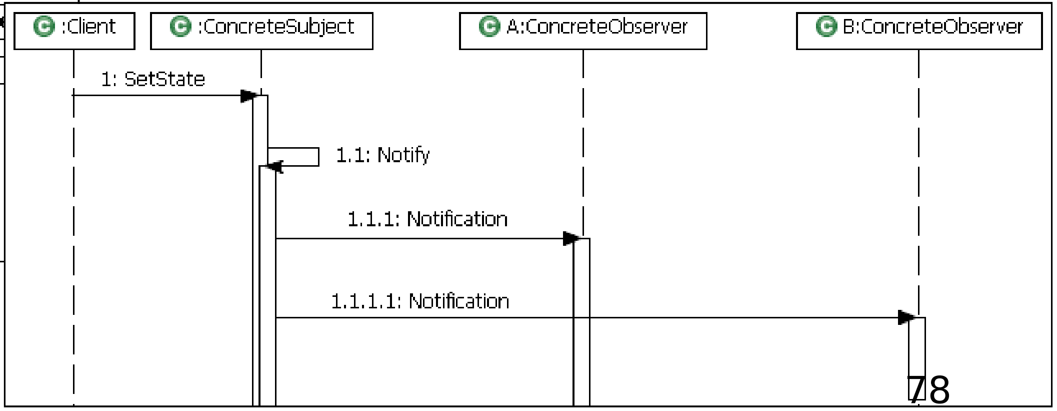


Design Pattern

- *A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context.* [Gamma]



Observer Pattern



Programming Pattern

- *An idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.* [Buschmann]

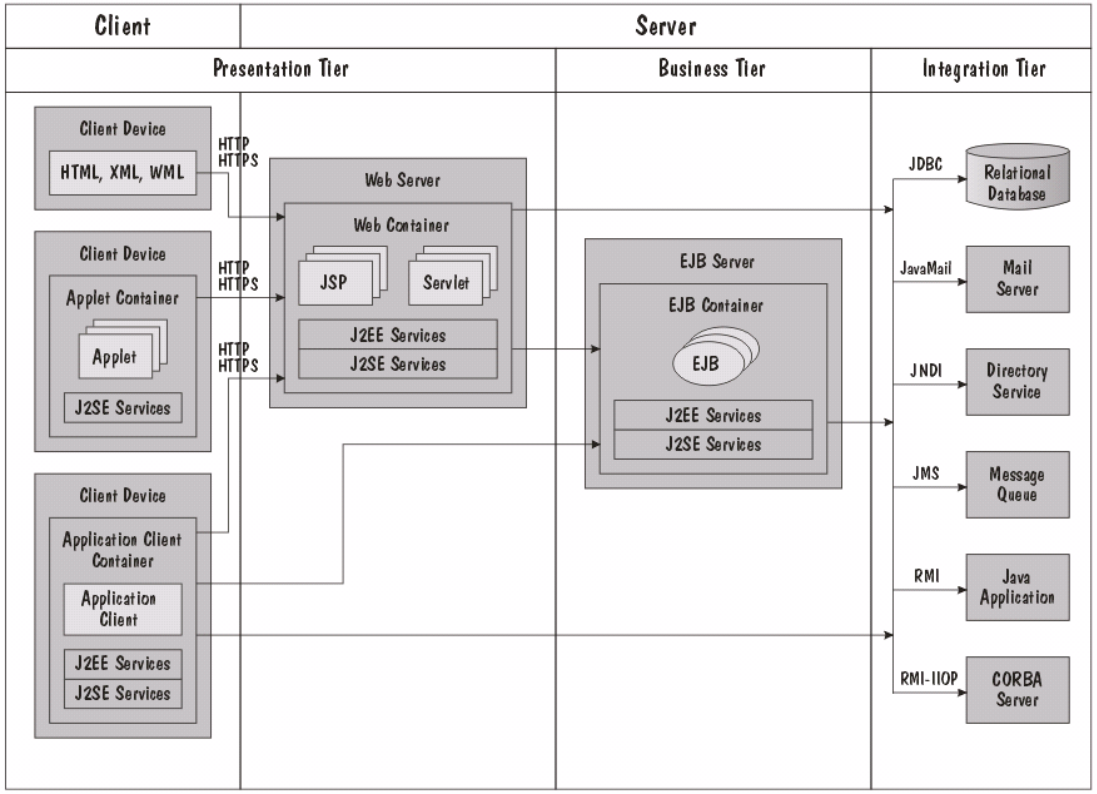
```
// Swap the values of 2 variables
temp = a;
a = b;
b = temp;
```

Architectural Style

- *[An architectural style] defines a family of systems in terms of a pattern of structural organization. More specifically, an architectural style defines a vocabulary of components and connector types, and a set of constraints on how they can be combined. [Shaw]*
- Client-server
  - Supports the physical separation of client-side processing (such as a browser) and server-side processing (such as an application server that accesses a database)
- Event-based
  - Promotes a publish-subscribe way of working, applied across large areas of the architecture
- Pipes-and-filters
  - A series of filters that provide data transformation, and pipes that connect the filters. Examples include compilers, signal processing

Reference Architecture

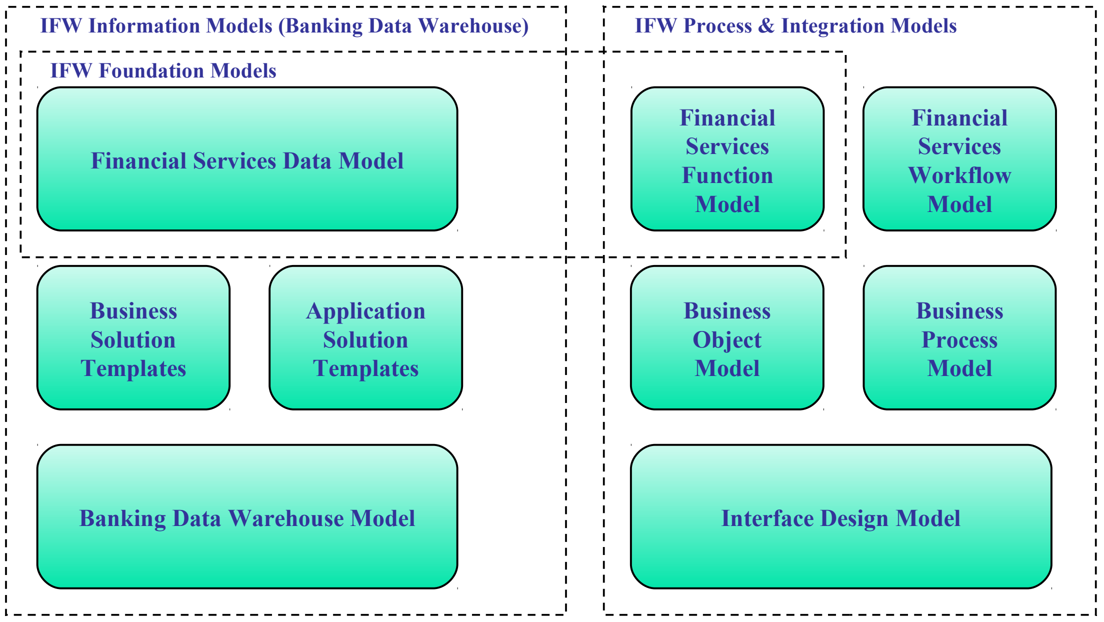
- *A reference architecture is an architecture representation of a particular domain of interest. It typically includes many different architectural patterns, applied in different areas of its structure*
- Examples include J2EE and .NET



Reference Model

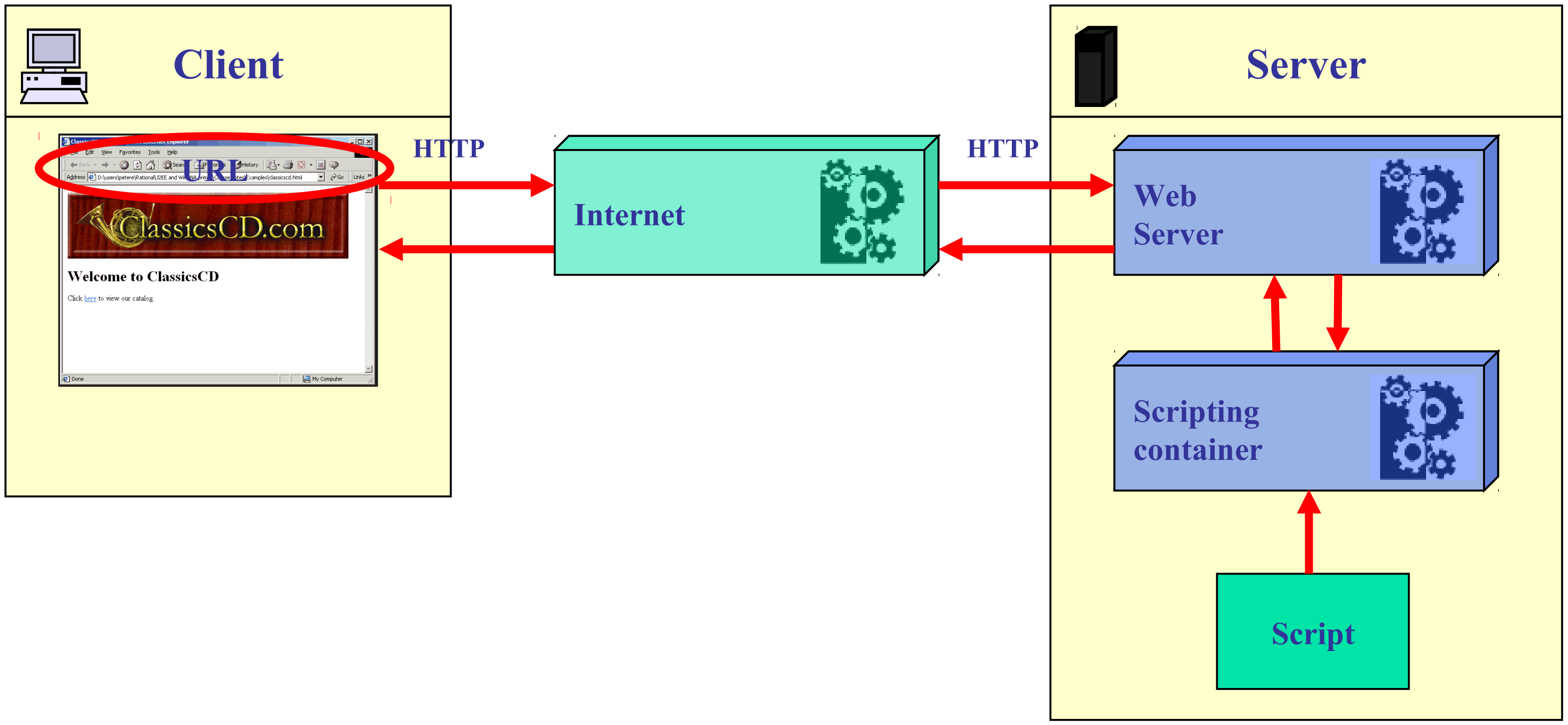
- *A reference model is an abstract representation of entities, their relationships and behavior, in a given domain of interest, and which typically forms the conceptual basis for the development of more concrete elements*
- Examples include a business model, an information model and a glossary of terms

IBM Information FrameWork (IFW)


IFW Information Models (Banking Data Warehouse)
IFW Process & Integration Models
IFW Foundation Models
Financial Services Data Model
Financial Services Function Model
Financial Services Workflow Model
Business Solution Templates
Application Solution Templates
Business Object Model
Business Process Model
Banking Data Warehouse Model
Interface Design Model

Application Framework

- *An application framework represents the partial implementation of a specific area of an application*
- Most widely-known frameworks support user interfaces
  - Java Server Pages
  - ASP.NET

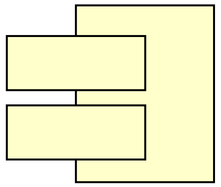


Packaged Application

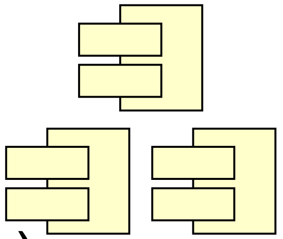
- *A packaged application is a large-grained Commercial-Off-The-Shelf (COTS) product that provides a significant amount of capability (and reuse)*
- Examples
  - Customer Relationship Management (CRM) application (e.g. Siebel)
  - Enterprise Resource Planning (ERP) application (e.g. SAP)
- The amount of custom development required is greatly reduced
- Primary focus is on configuring the application

Component & component library

- Component examples
  - GUI widget (such as a table)
  - Service



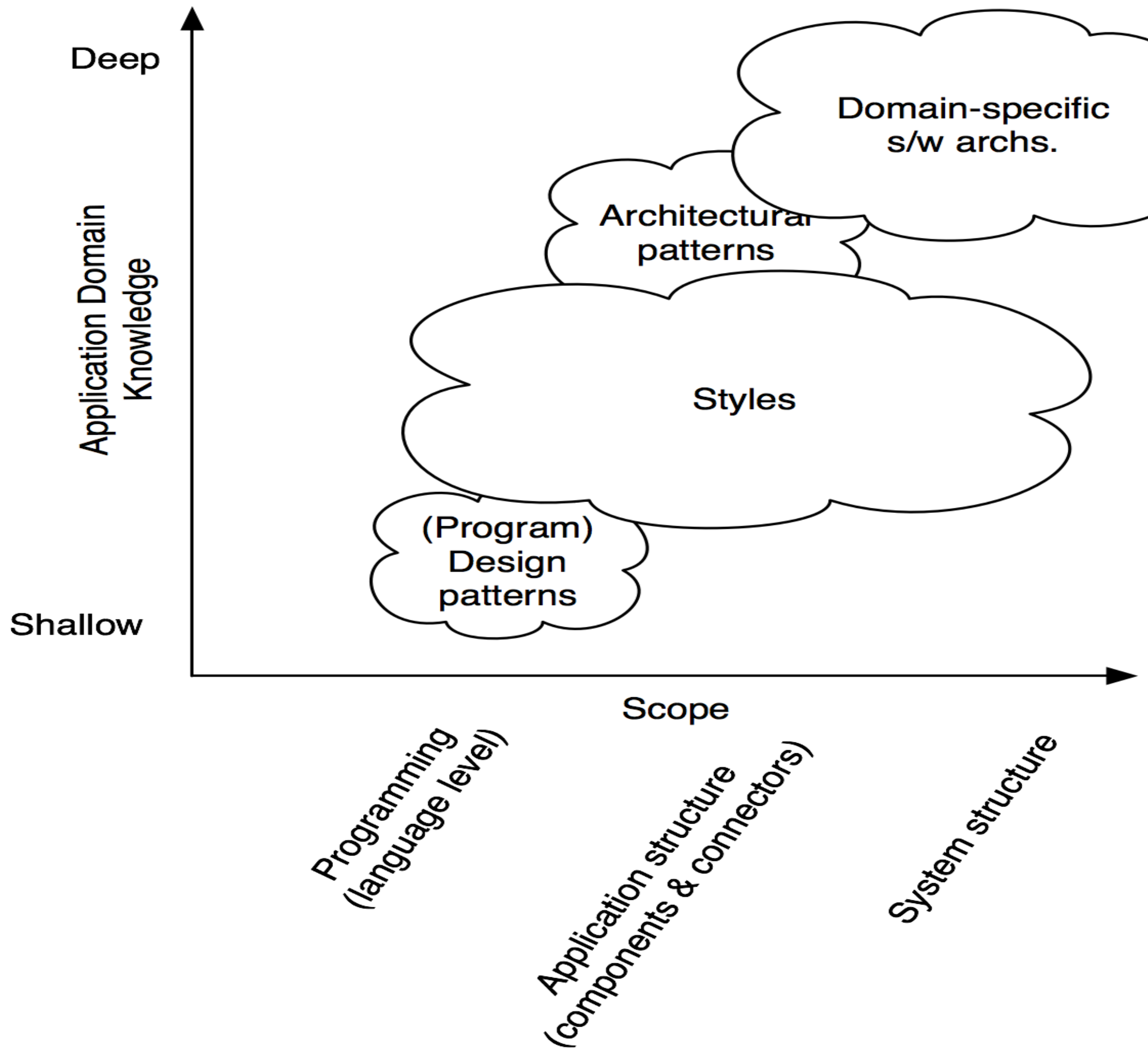
- Component library examples
  - Class libraries (e.g. Java class library)
  - Procedure libraries



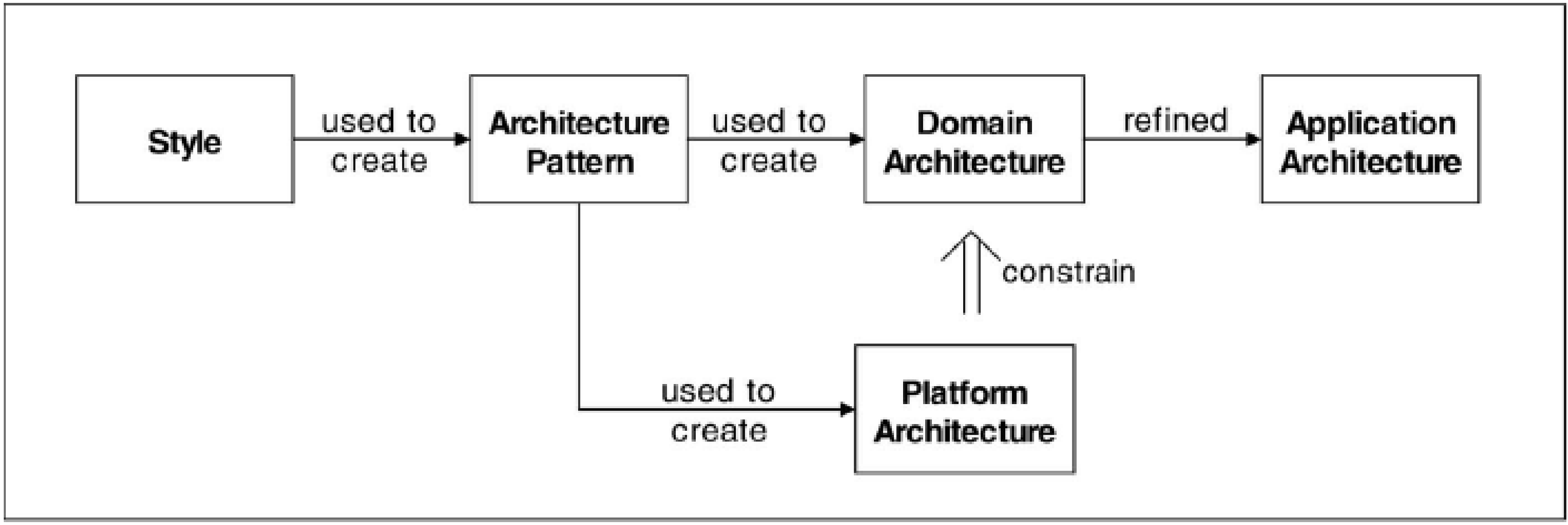
Legacy Application

- *A legacy application is a system that continues to be used because the owning organization cannot replace or redesign it*
- Tends to be a focus on integration rather than new development
- Often results in a focus on enterprise application integration (EAI)

Patterns, styles, and DSSAs


Deep
Application Domain Knowledge
Shallow
Domain-specific s/w archs.
Architectural patterns
Styles
(Program) Design patterns
Scope
Programming (language level)
Application structure (components & connectors)
System structure

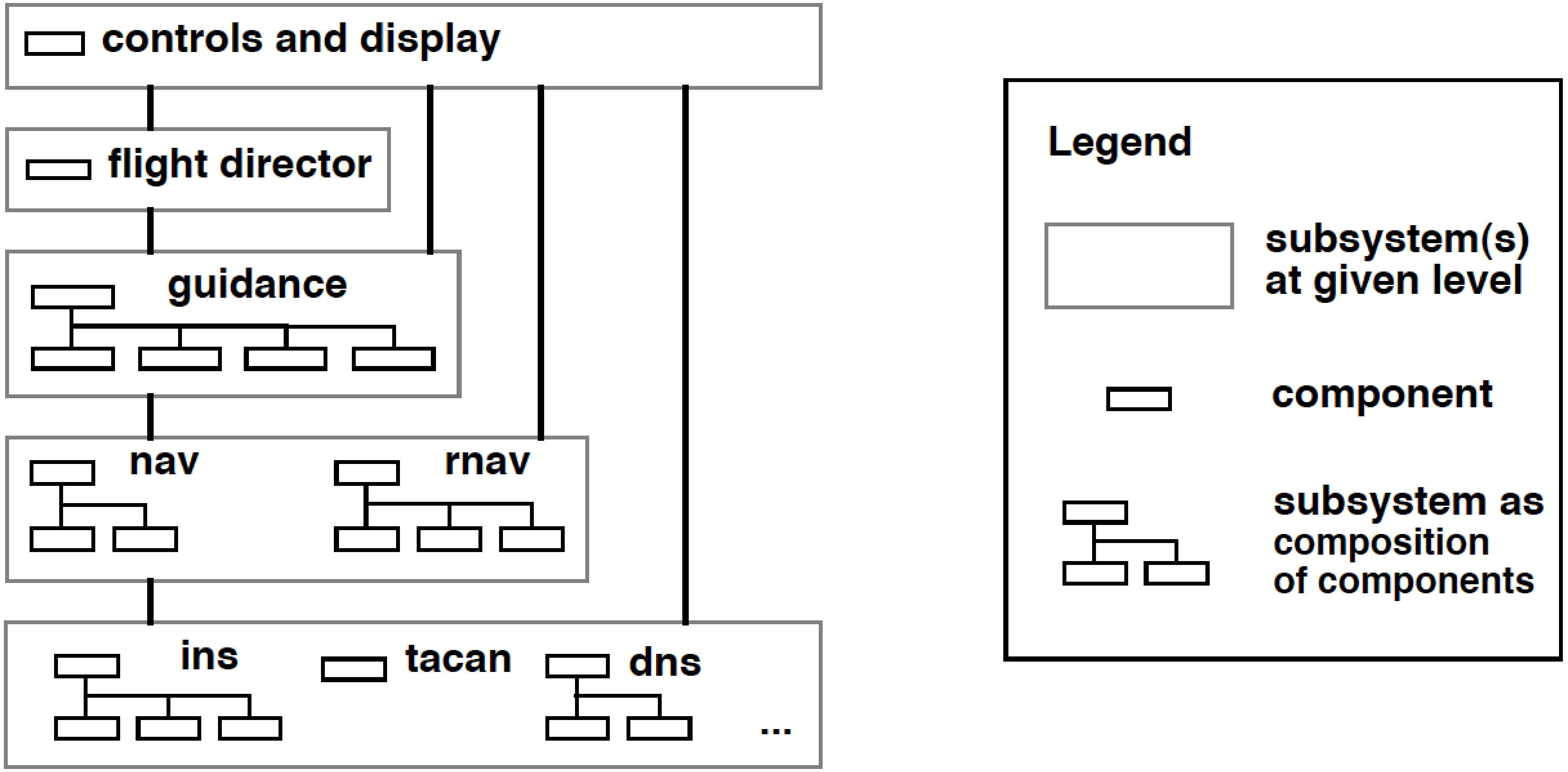
A map of architectural concepts



Domain-Specific Software Architectures

- A DSSA is an assemblage of software components
  - specialized for a particular type of domain,
  - generalized for effective use across that domain, and
  - composed in a standardized structure (topology) effective for building domain specific applications.
- Since DSSAs are specialized for a particular domain they are only of value if one exists for the domain wherein the engineer is tasked with building a new application.
- DSSAs are the pre-eminent means for maximal reuse of knowledge and prior development and hence for developing a new architectural design.

DSSA Example



D.Batory and others, Creating Reference Architectures: An Example from Avionics, 1995

The role of the software architect

The role of the Software Architect

- The role of the software architect involves not just technical activities, but others that are more "political" and strategic in nature, and more like those of a consultant
- Not all good technologists have the skills that make them good architects
- The best architects are good technologists, that command respect in the technical community, but also are good strategists, organizational politicians and leaders

A quotation

The architect must be a prophet...

in the true sense of the term ...
if he can’t see at least ten years ahead,

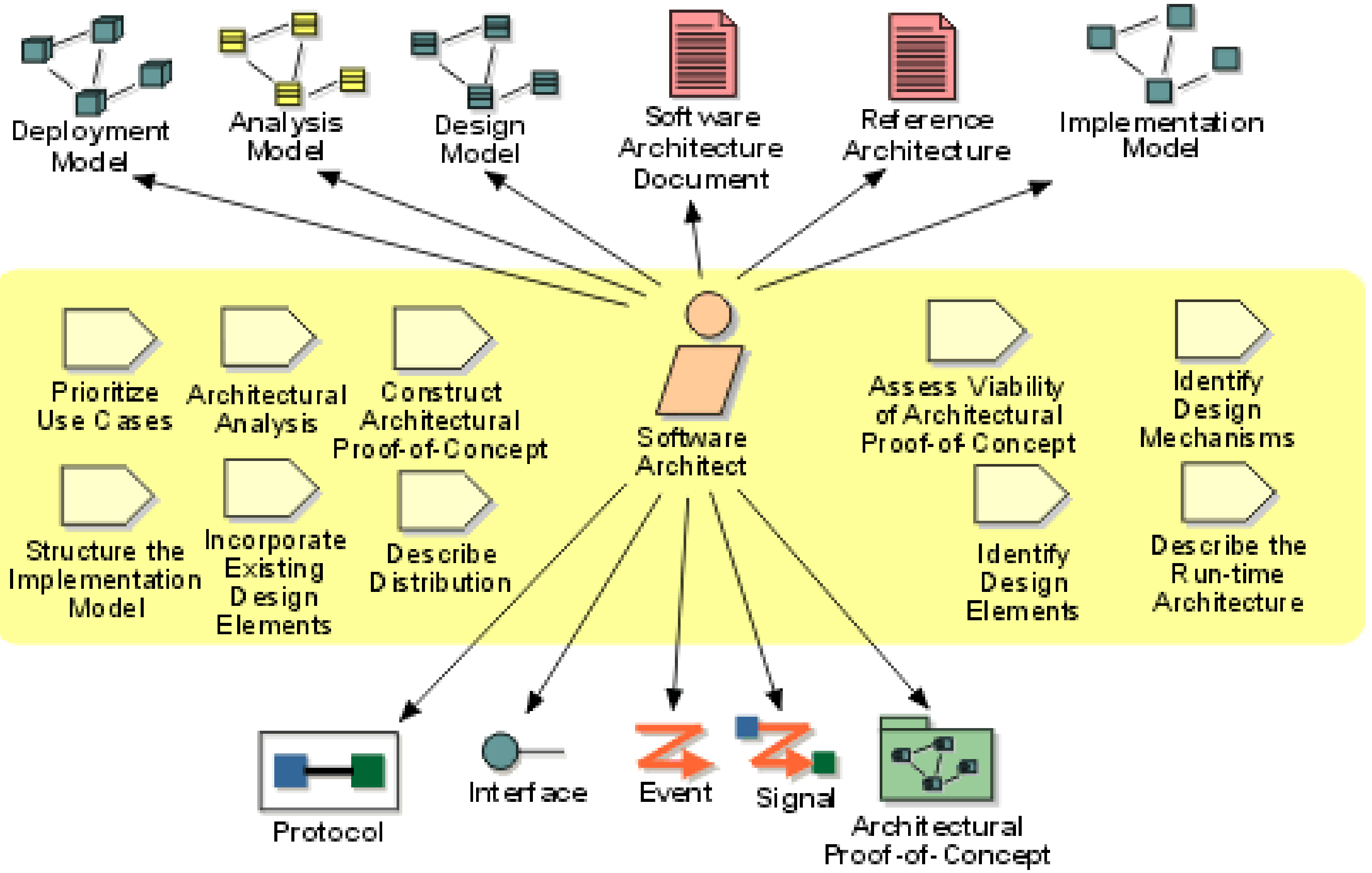
don’t call him an architect

Frank Lloyd Wright

The need

- The best architectures are the product of
  - A single mind or
  - A very small, carefully structured team
    - Rechtin, *Systems Architecting: Creating & Building Complex Systems*, 1991, p21
- Every project should have exactly 1 architect
  - For larger projects, the principal architect should be backed up by architect team of small size
    - Booch, *Object Solutions*, 1996

The role of software architect



Deployment Model
Analysis Model
Design Model
Software Architecture Document
Reference Architecture
Implementation Model
Prioritize Use Cases
Architectural Analysis
Construct Architectural Proof-of-Concept
Software Architect
Assess Viability of Architectural Proof-of-Concept
Identify Design Mechanisms
Structure the Implementation Model
Incorporate Existing Design Elements
Describe Distribution
Identify Design Elements
Describe the Run-time Architecture
Protocol
Interface
Event
Signal
Architectural Proof-of-Concept

Developer/Integrator/Architect skills

Software Developer

- Typically involved in new applications / components
- Ability to compare technologies and solutions, in order to choose the best one
- Analisys, design, programming and testing skills
- Tipically a young professional
- Low cost

Application Integrator

- Reuse of legacy applications (usually badly documented), and use of COTS components
- Faces constraints on OSs, protocols, languages, development environments to be used; and technologies of legacy systems which often are not interoperable
- Re-engineering, reuse, interface design and components adaptation skills
- Familiar with pre-existing systems
- Typically a senior professional
- Medium/high cost

Software Architect

- Master technology trends
- Ability to interface both business/organization key people and technical teams
- Strategic vision
- Organizational skills
- Communication skills
- Ability to design for change
- Manage risk identification and risk mitigation strategies
- Cost estimator
- Leading skill
- High cost

Architects as sw development experts

- Must understand details of software development
  - Principles
  - Methods & techniques
  - Methodologies
  - Tools
- Need not be world-class software programmers
- Should understand consequences of architectural choices
  - Some architectural choices constrain implementation options
  - Some implementation-level techniques & tools constrain architectural choices

Architects as domain experts

- Software engineering expertise is not enough
- Problem domain knowledge
  - Maturity
  - Stability
  - System user profile
- May affect selected & developed architectural solutions
  - Distribution
  - Scale
  - Evolvability
- Requires artifacts that model problem space
  - Not solution space

Architects as communicators

- At least ½ of the job
- Must
  - Listen to stakeholder concerns
  - Explain the architecture
  - Negotiate compromises, acquire consensus
- Need good communication skills
  - Writing
  - Speaking
  - Presenting

Architects communicate with

- Managers
  - Must relay key messages
    - Architecture is useful & important
    - Ensure support throughout project
  - Must listen to concerns
    - Cost
    - Schedule
- Developers
  - Convince them that the architecture is effective
  - Justify local suboptimal choices
  - Listen to problems
    - Tools, methods, design/implementation choices
- Other software architects
  - Ensure conceptual integrity
  - Ensure desired system properties & evolution

Architects also communicate with

- System engineers
  - Coordinate requirements & solutions
  - Explain how architecture addresses key concerns
- Customers
  - Determine needs
  - Explain how architecture addresses requirements
- Users
  - Determine needs
  - Explain how architecture addresses those needs
  - Listen to problems
- Marketers
  - Get/help set goals & directions
  - Explain how architecture addresses marketing objectives

Architects as strategists

- Developing an architecture is not enough
  - Technology is only part of picture
  - Architecture must be right for organization
- Must fit organization’s
  - business strategy and rationale behind it,
  - business practices,
  - planning cycles,
  - decision making processes
- Must also be aware of competitors’
  - Products
  - Strategies
  - Processes

Architects as cost estimators

- Must understand financial ramifications of architectural choices
  - Make or buy
  - Cost of COTS adoption
  - Cost of development for reuse
  - Company’s financial stability & position in marketplace
- Technologically superior solution is not always the most appropriate one
  - Impact on cost & schedule
- Quick, approximate cost estimations are often sufficient
  - Detailed cost estimation techniques can be applied once set of candidate solutions is narrowed down

Architect - Summary

- Designate architect or assemble architecture team to be creators & proponents of common system goal/vision
- Architects must be experienced at least in problem domain & software development
- Software architect is a full-time job
- Charter of software architecture team should
  - Clearly define its roles & responsibilities
  - Clearly specify its authority
- Do not isolate software architecture team from the rest of project personnel
- Avoid pitfalls

Evaluating software architectures

Evaluating architectures – Why?

- Evaluating the candidate architecture before it becomes the project blueprint can be of great economic value
- Useful to:
  - Assess whether the candidate can deliver the expected benefits
    - Assessment against stakeholders’ requirements/software quality attributes/quality managers concerns
  - Evaluate a large software system with a long expected lifetime before acquiring it
  - Compare alternatives
  - Architecture refactoring
- Repeatable, structured architecture evaluation methods are available now

Evaluating architectures – When?

- Evaluation can take place at many points
- As usual: the earlier, the better
  - Software quality cannot be appended late in a project, it must be built in from the beginning
- Typically, when the architecture is finished, before the project commits to expensive development
- But also:
  - Compare two competing architectures
  - Evaluate the architecture of a legacy system undergoing evolution
  - Evaluate the architecture of a system to be acquired
  - Evaluate OTS (Off-The-Shelf) (sub)systems

Evaluating architectures – How?

- Some architecture evaluation methodologies are available
  - ATAM
  - CBAM
- Planned
  - Scheduled well in advance
  - Built into the project’s plan and budget
- Unplanned
  - When the management perceives that the project has risks of failure or needs a correction
  - Reactive – Tension filled

Evaluating architectures – How?

- Preconditions:
  - A few high-priority goals – 3 to 5
  - Availability of key personnel
  - Competent evaluation team
- Cost: a few tens of staff-days in a large project
  - ATAM requires approximately 36 staff-days
- Result:
  - A report describing all issues of concern, along with supporting data
  - Report should include costs of the evaluation and estimated benefits if concerns are addressed
  - Report to be first circulated in draft form among participants
  - Issues should be ranked by potential impact on the project if unaddressed

Evaluating architectures – Benefits

(Especially for planned evaluations)

- Economic / financial
- Preparation for review
- Captured rationale
- Early detection of problems
- Validation of requirements
- Improved final architecture
- Ensuring proper documentation

Architectural styles

Definition of a software architecture

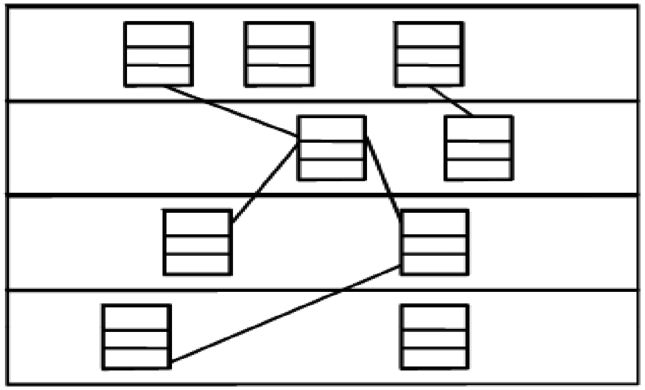
- The architectural definition of a system is generally achieved by breaking up the system into subsystems, following a ***layered*** and/or ***partitions*** based approach (*tiers*).
- These are orthogonal approaches:
  - A layer is a level in charge of providing services (e.g., a user interface)
  - A tier is the organization of several peer modules (within the same layer, in most of the cases)

Layers

- A system which has a hierarchical structure, consisting of an ordered set of layers
- A layer is given by a set of subsystems which are able to provide related services, that can even be realized by exploiting services from other layers
- A layer depends only on lower layers (i.e., layers which are located at a lower level into the architecture)
    - A layer is only aware of lower layers
- **Closed architecture**: the *i-th* layer can only have access to the layer (i-1)-th
- **Open architecture**: the *i-th* layer can have access to all the underlying layers (i.e., the layers lower than *i*)

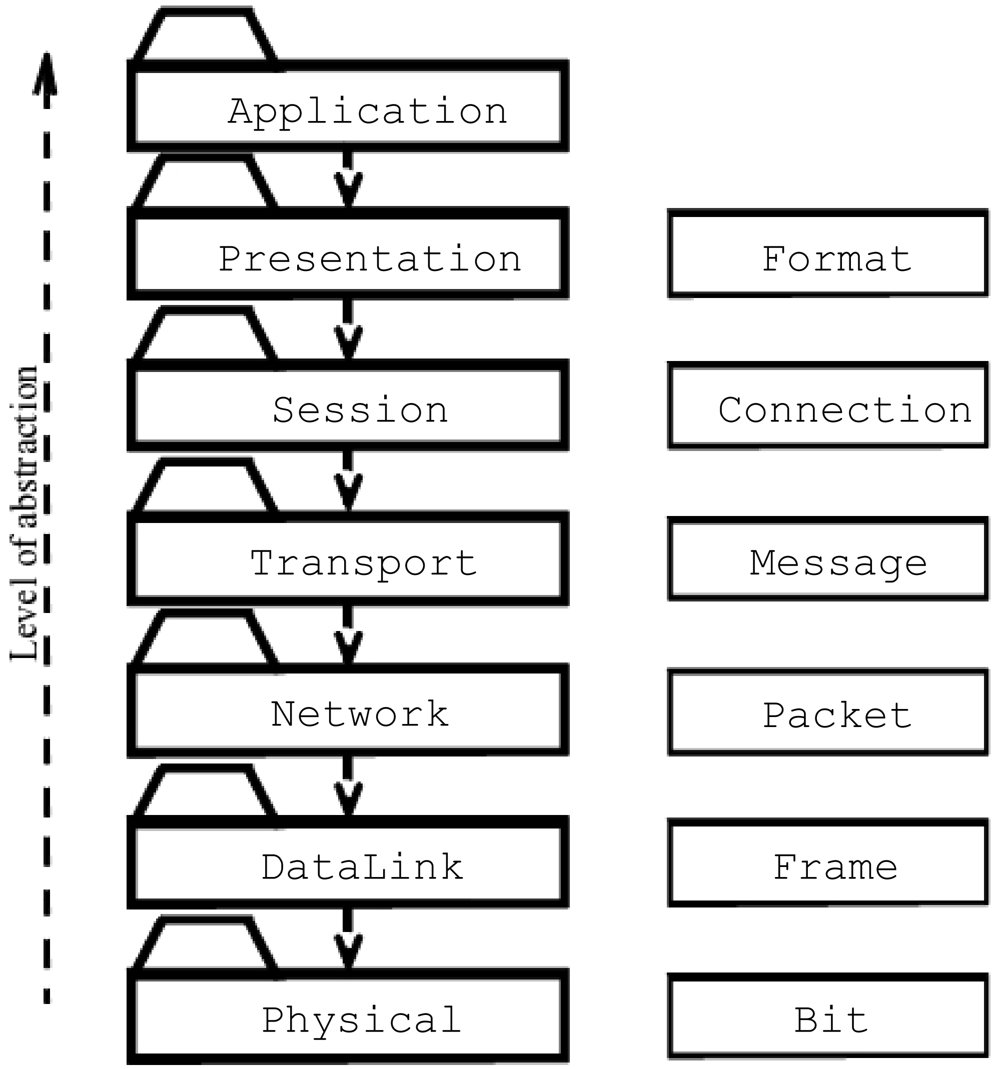
Layers – Closed architecture

- The i-th layer can only invoke the services and the operations which are provided by the layer (i-1)-th
- The main goals are the system maintainability and high portability



An example of closed architecture

- ISO/OSI stack
  - The ISO/OSI reference model defines 7 network layers, characterized by an increasing level of abstraction



Layers – Open architecture

- The i-th layer can invoke the services and the operations which are provided by all the lower layers (the layers lower than i)
- The main goals are the execution time and efficiency

Tiers

- A further approach to manage system complexity consists of *partitioning* the system into peer sub-systems (*peers*), which are responsible for a class of services
- Typically, a complete decomposition of a given system comes from both layering and partitioning
  - First, the system in divided into top level subsystems which are responsible for certain functionalities (*partitioning*)
  - Second, each subsystem is organized into several layers, if necessary, up to the definition of *simple enough* layers

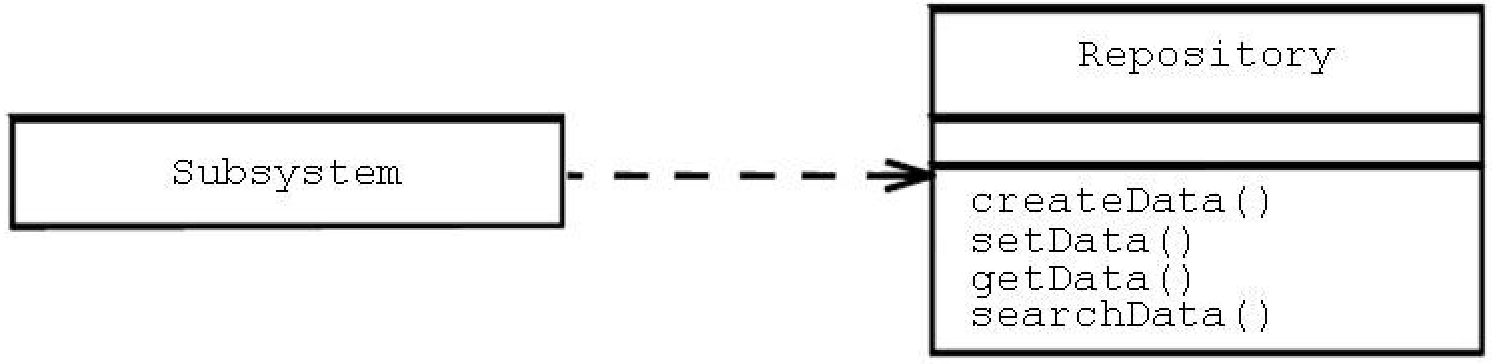
Architectural styles

- Several architectural styles have been defined in the literature of software engineering. They can be used as the basis for software architectures. They include:
  - Repository
  - Client/Server
    - two-tiers; three-tiers; n-tiers
  - Model/View/Controller
  - Distributed objects
  - Distributed components
  - Service-Oriented
  - Peer-To-Peer

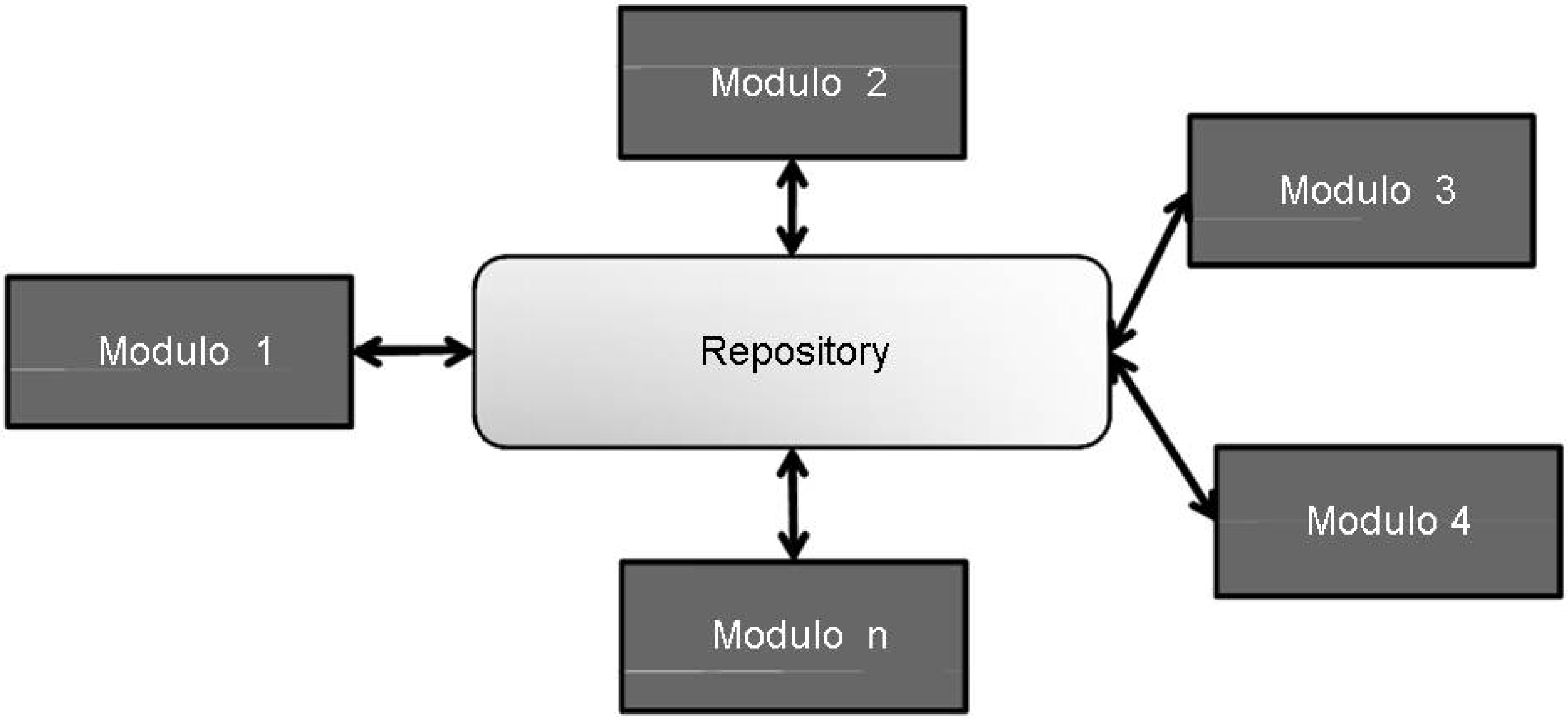
Repository-based architectures

Repository Architecture – 1/4

- The subsystems use and modify (i.e., they have access to) a data structure named **repository.**
- They are relatively independent, in that they interact only through the repository
- The control flow into the system can be managed both by the repository (if stored data have to be modified) and by the subsystems (independent control flow)

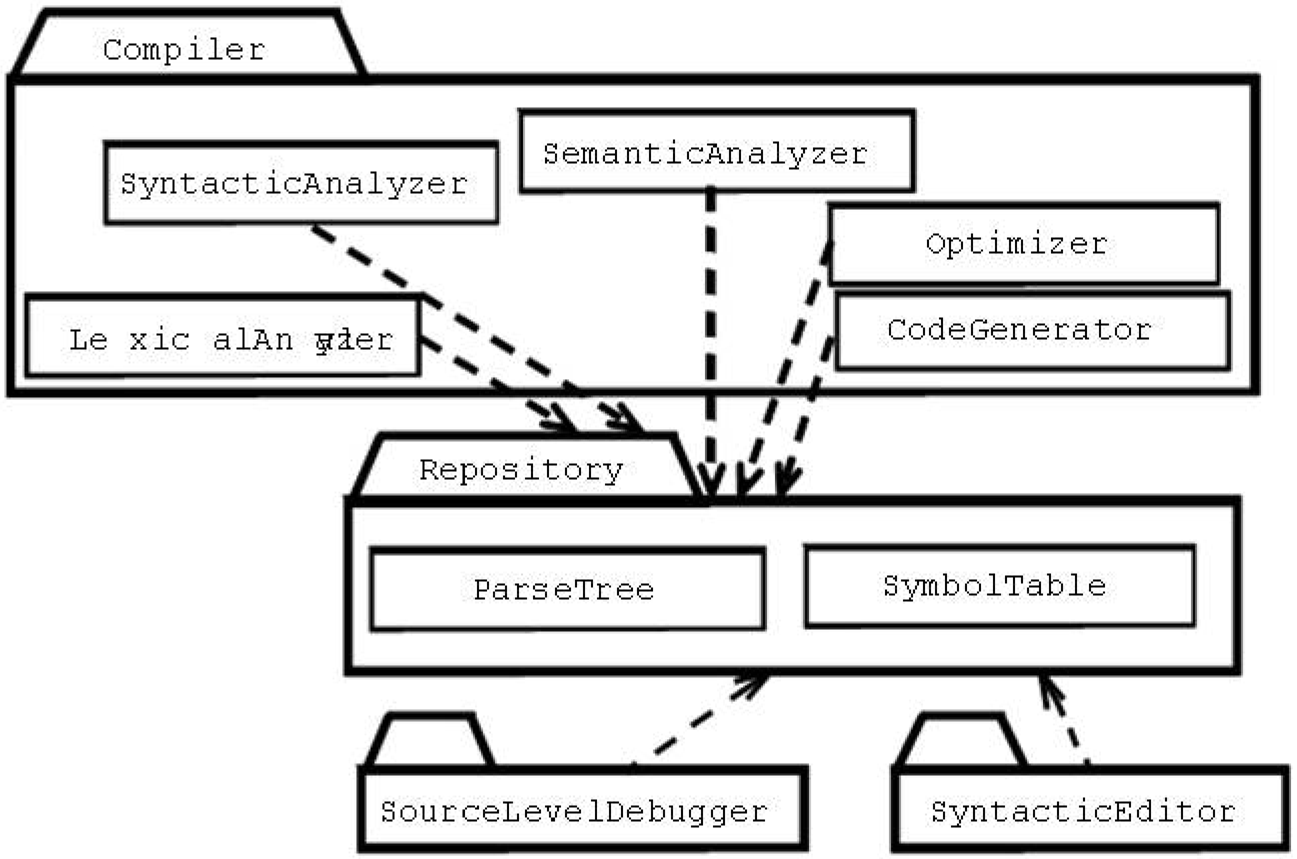


Repository Architecture – 2/4



Repository Architecture – 3/4

- An example: a compiler architecture



Repository Architecture – 4/4

- Benefits
  - It is an effective mean to share huge amount of data: write once for all to read
  - Each subsystems has not to take care of how data are produced/consumed by other subsystems
  - It allows a centralized management of system backups, as well as of security and recovery procedures
  - The data sharing model is available as the *repository schema,* hence it is easy to plug new subsystems
- Pitfalls
  - Subsystems have to agree on a data model, thus impacting on performance
  - Data evolution: it is "expensive" to adopt a new data model since (a) it has to be applied on the entire repository, and (b) all the subsystems have to be updated
  - Not all the subsystems requirements in terms of backup, security are always supported by the repository
  - It is tricky to deploy the repository on several machines, preserving the logical vision of a centralized entity, due to redundancy and data consistency matters

(CSA)

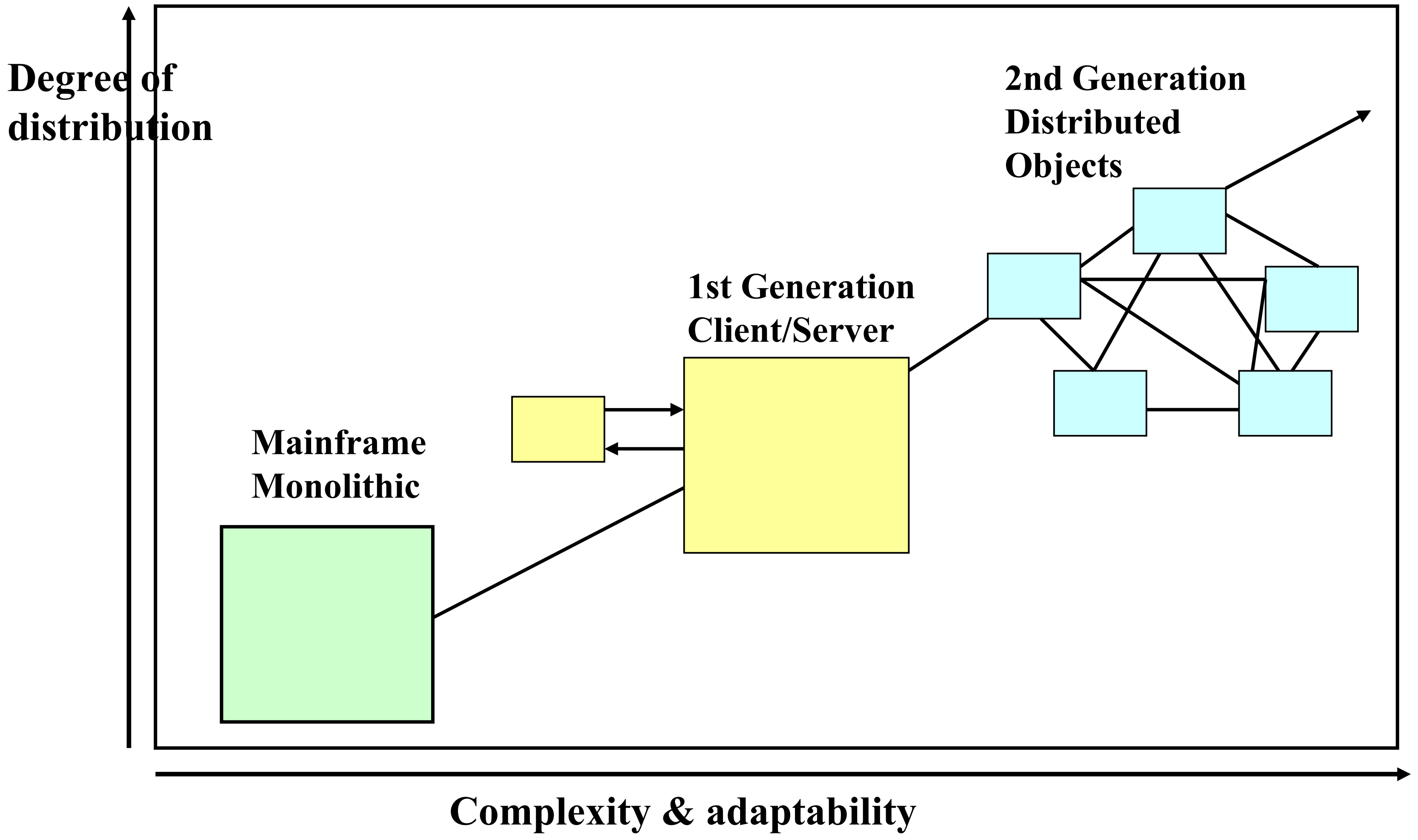
Client-Server Architectures

Client-server paradigm – 1/2

- It has been conceived in the context of distributed systems, aiming to solve a synchronization issue:
  - A protocol was needed to allow the communication between two different entities
- The driving idea is to make unbalanced the partners’ roles within a communication process
  - Passive entities (named ***server***):
    - They cannot initiate a communication
    - … they can only answer to incoming requests (reactive entities)
  - Active entities (named ***client***):
    - They trigger the communication process
    - They forward requests to the servers, then they wait for responses

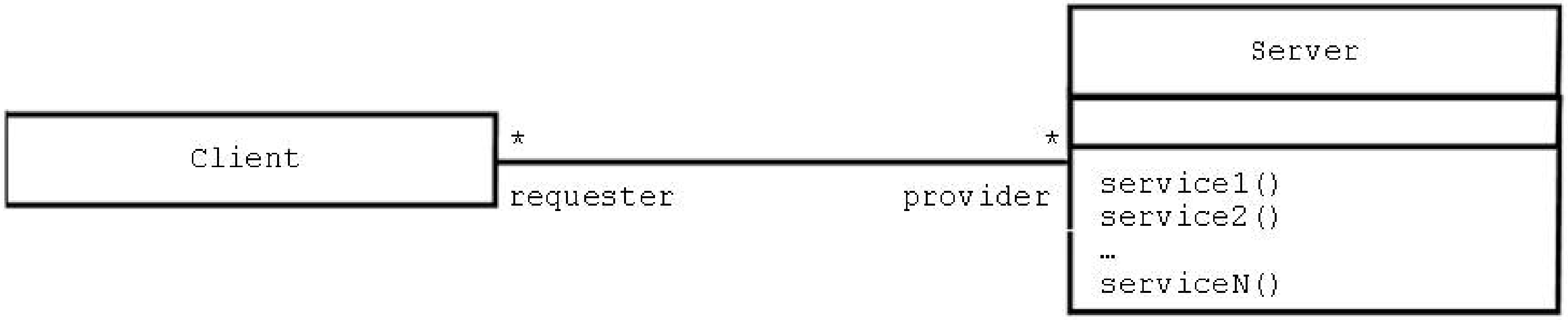
Client-server paradigm – 2/2

- The first generation of software distributed systems was based on the client server paradigm


Degree of distribution
2nd Generation Distributed Objects
1st Generation Client/Server
Mainframe Monolithic
Complexity & adaptability

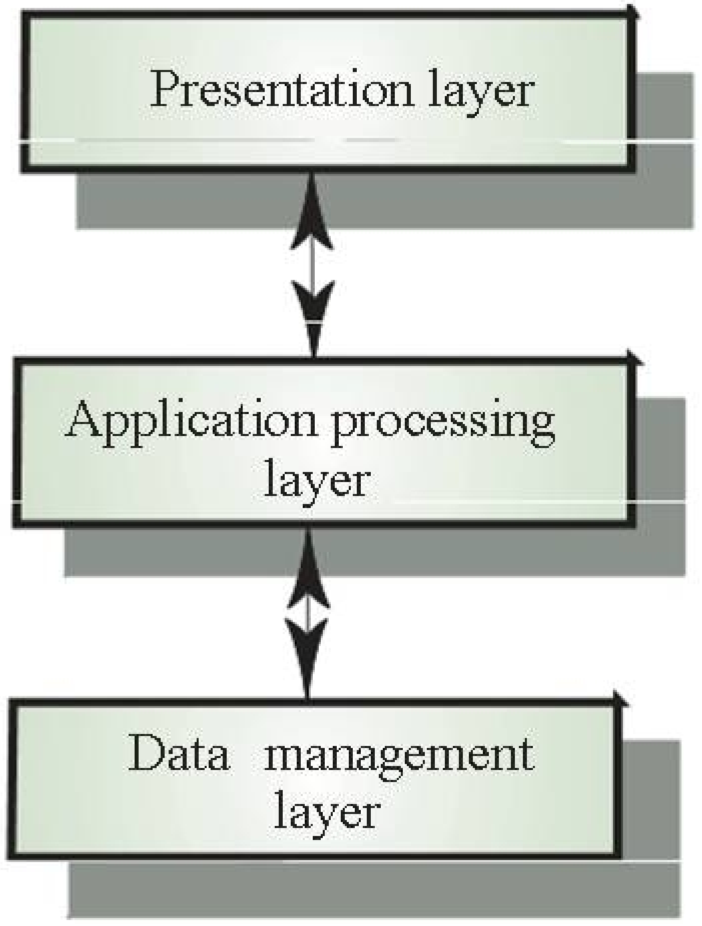
Client-server architecture – 1/3

- From the architectural viewpoint, a C/S software system is made up of components which can be classified according to the *service* abstraction:
  - Who provides services is a server;
  - Who requires a service is a client;
- Clients are typically responsible for the interaction between the system and its users
  - …but a server can be a client for a different service
- Clients are aware of the server interface
- Servers cannot foresee clients' requests



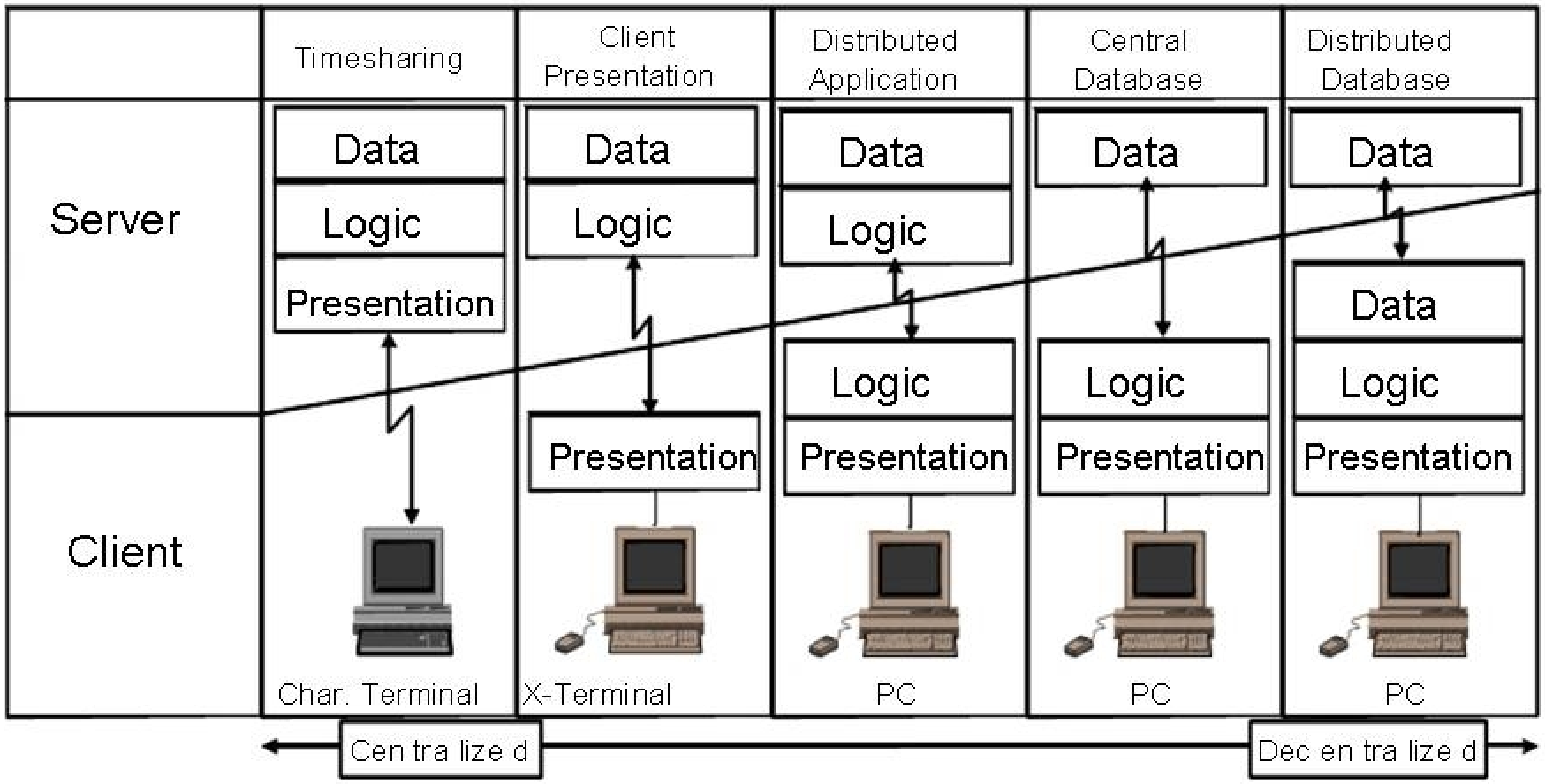
Client-server architecture – 2/3

- In fact, a system can be logically divided into three parts:
  - The **presentation**, which is in charge of managing the user interface (graphic events, input fields check, help..)
  - The actual **application logic**
  - The **data management layer** for the management of persistent data.
- The system architecture is defined according to how these parts are organized :
  - 2-tiered
  - 3-tiered
  - n-tiered



Client-server architecture – 3/3

- 2 tiers Client/Server architectures

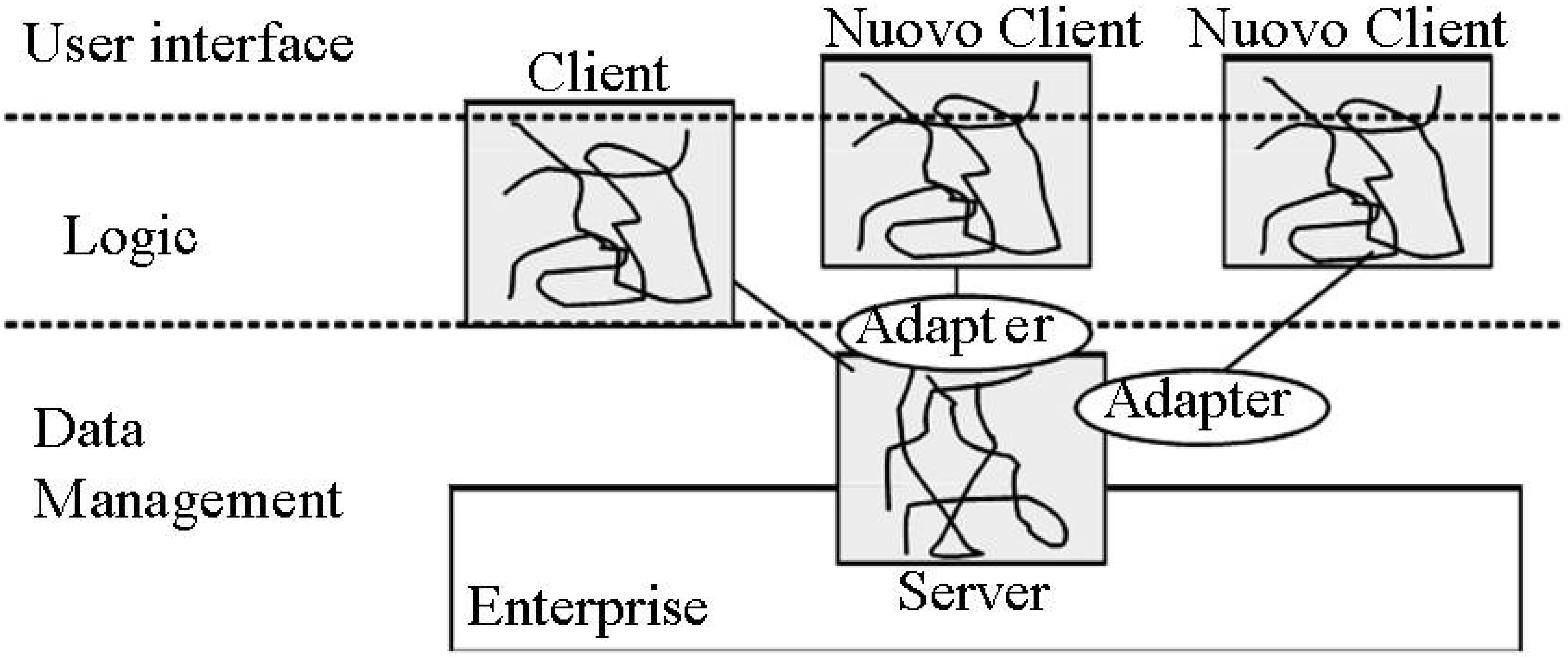


Two-tier C/S architectures – 1/2

- Pitfalls:
  - They exhibit a heavy message traffic since front-end and the servers communicate intensively
  - The business logic is not managed by a ad-hoc component, but is it “shared” by front-end and back-end
    - client and server depend one from each other
    - It is difficult to reuse the same interface to access to different data
    - It is difficult to interact with databases which have different front-end
    - The business logic is encapsulated into the user interface
      - If the logic changes, the interface has to change as well

Two-tier C/S architectures – 2/2

- Recurrent problem:
  - *Business* and *presentation logic* are not clearly separate
    - E.g., if there is a service which can be accessed by several devices (e.g., mobile phone, desktop PC)
      - The same logic but different interface



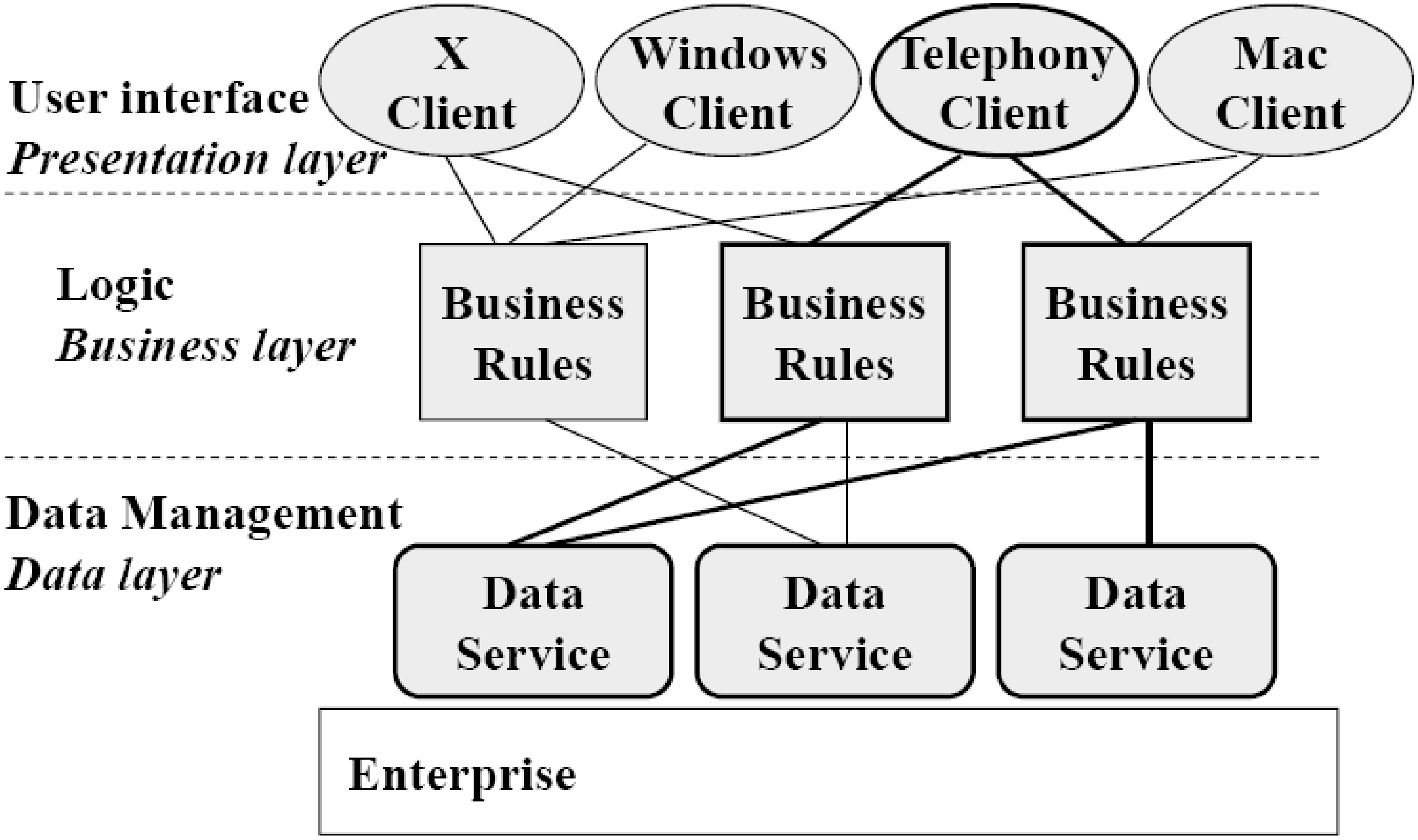
Three-tier architectures – 1/6

- Early 90's; they propose a clear separation among logics:
  - Level 1: data management (DBMS, XML files, …..)
  - Level 2: business logic (application processing, …)
  - Level 3: user interface (data presentation and services)
- Each layer has its own goals, as well as specific design constraints
- No assumptions at all about the structure and/or the implementation of the other layers, i.e.:
  - Level 2 does not make any assumptions neither on how data are represented, nor on how user interfaces are made
  - Level 3 does not make any assumptions on how *business logic works*

Three-tier architectures – 2/6

- Levels 1 and 3 do not communicate, i.e.:
  - The user interface neither receives any data from data management, nor it can write data
  - Information passing (in both the directions) are filterer by the business logic
- Levels work as they were not part of a specific application:
  - Applications are conceived as collections of interacting components
  - Each component can take part to several applications at the same time

Three-tier architectures – 3/6



Three-tier architectures – 4/6

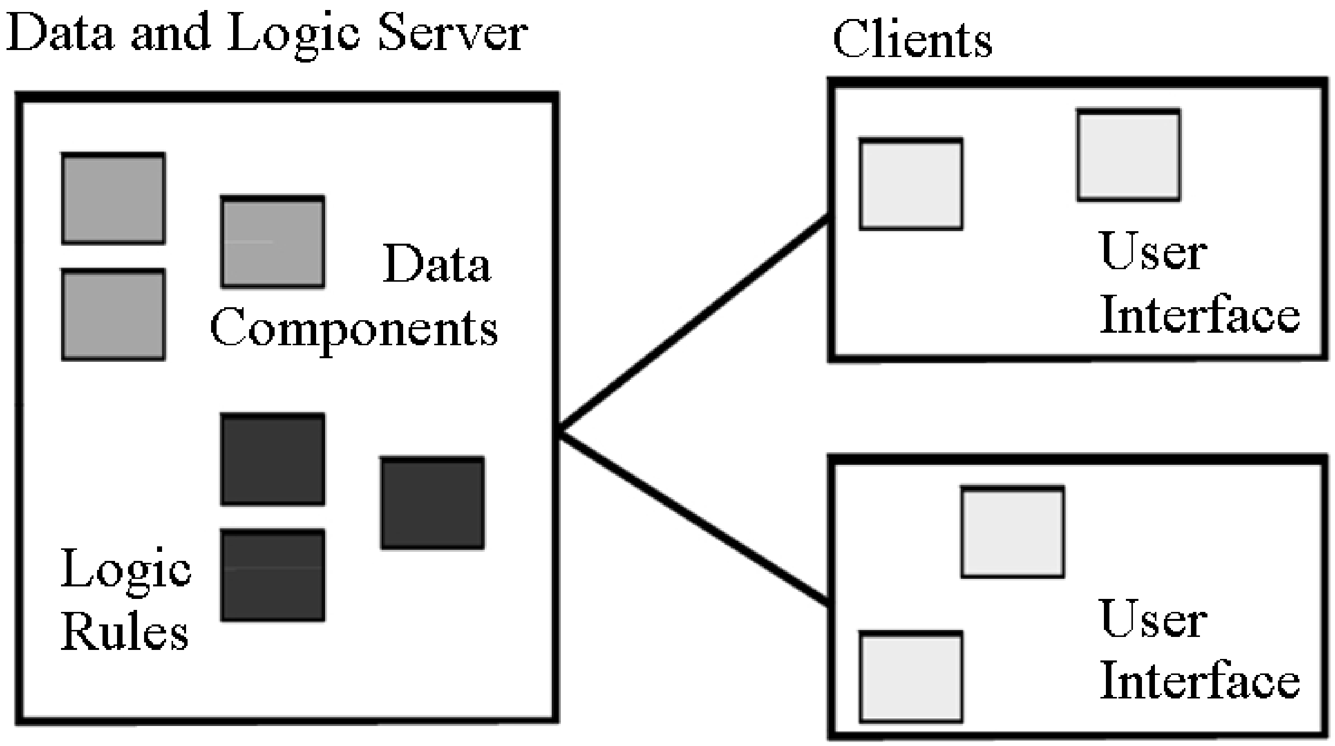
- Benefits:
  - They leverage the flexibility of and the high modifiability of modular systems
    - Components can be used in several systems
    - A change into a given component do not have any impacts on the system (apart from changes into the API)
    - It is easier to localize a bug since the system functionalities are isolated
    - New functionalities can be added to the system by only modifying the components which are in charge of realizing them, or by plugging new components

Three-tier architectures – 5/6

- Pitfalls:
  - Application dimensions and efficiency:
    - Heavy network traffic
      - High latency for service delivering
    - Ad hoc software libraries have to be used to allow the communication among components
      - Many LoCs!
  - Legacy software
    - Many industries make use of preexisting software, monolithic, systems to manage data
    - Adapters have to be implemented to interoperate with the legacy SW

Three-tier architectures – 6/6

- Levels are **logical** abstractions, not physical entities
  - Three-tier architectures can even be realized by using one or two levels of machines



N-tier architectures

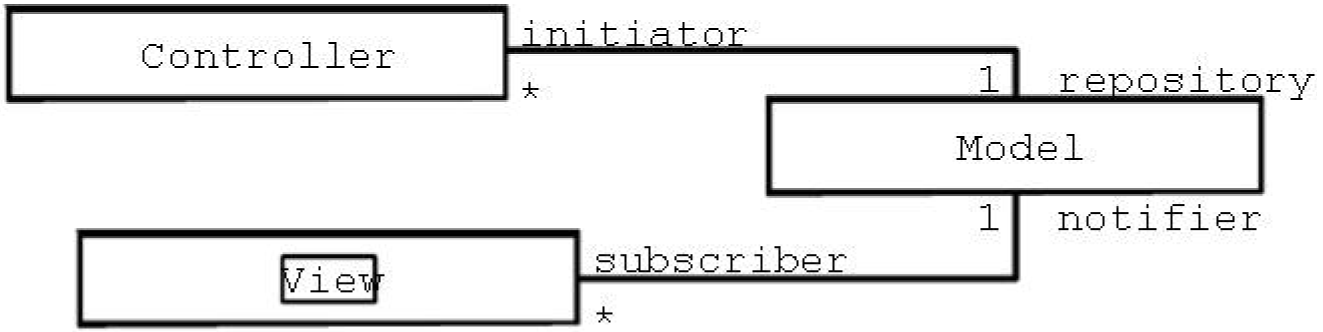
- They provide high flexibility
- Fundamental items:
  - User Interface (UI): e.g., a browser, a WAP minibrowser, a graphical user interface (GUI)
  - Presentation logic, which defines what the UI has to show and how to manage users' requests
  - Business logic, which manages the application business rules
  - Infrastructure services
    - The provides further functionalities to the application components (messaging, transactions support);
  - Data layer:
    - Application Data level

MVC

Model/View/Controller Architectures

MVC Architectures – 1/2

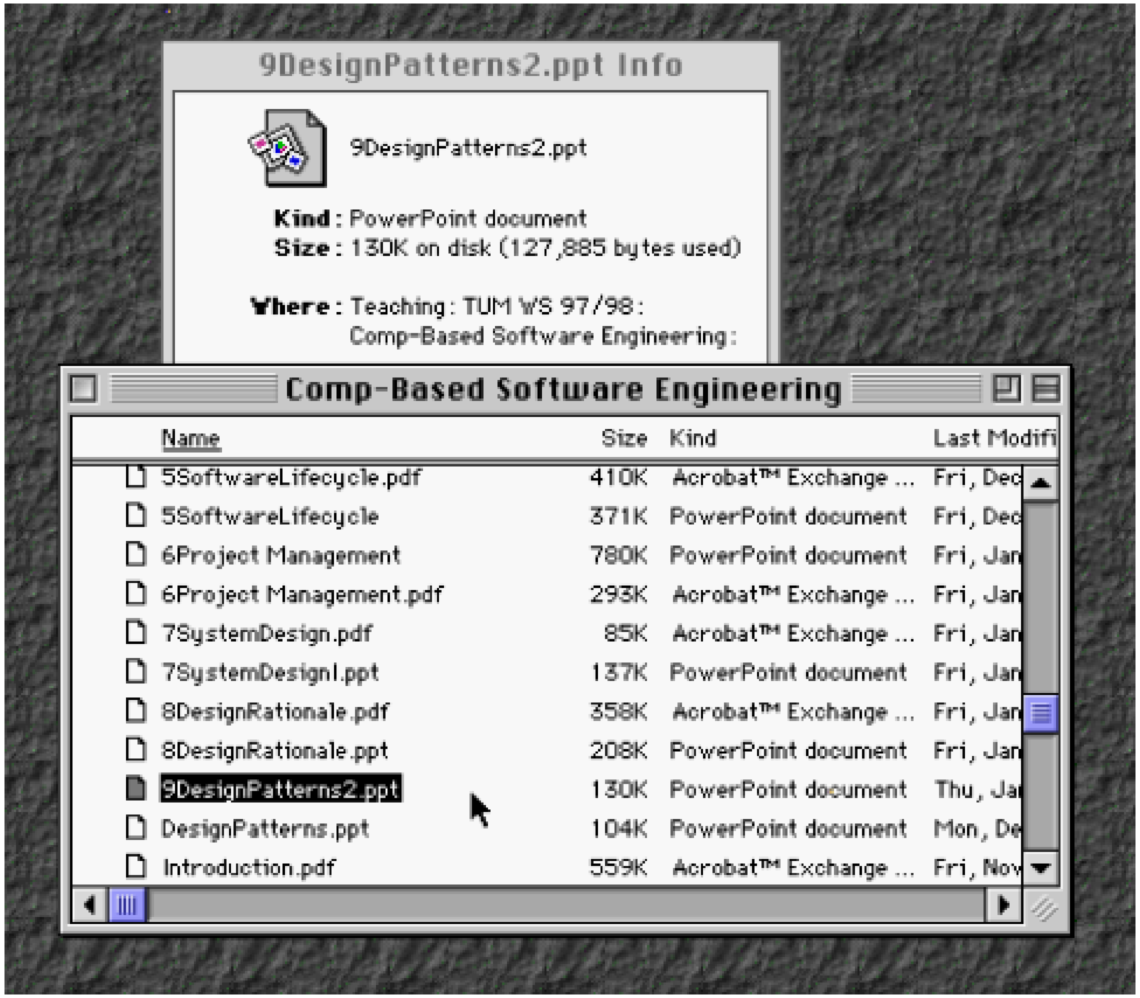
- Three kinds of subsystems:
  - **Model**, which keep the knowledge about the application domain
  - **View**, which takes care of making application objects visible to system users
  - **Controller,** which manages the interactions between the system and it users.



MVC Architectures – 2/2

- Model subsystems do not depend on any View and/or Controller subsystems. Changes into their state are communicated to the viewer (View subs) by means of a “subscribe/notify” protocol
- MVC represents a meeting point between the repository and the three tier architecture:
  - The Model implements the centralizes data structure;
  - The Controller manages the control flow: it receives inputs from users and forwards messages to the Model
  - The Viewer pictures the model.

MVC: An example – 1/2



9DesignPatterns2.ppt Info

9DesignPatterns2.ppt

Kind : PowerPoint document
Size : 130K on disk (127,885 bytes used)

Where : Teaching : TUM WS 97/98 : Comp-Based Software Engineering :

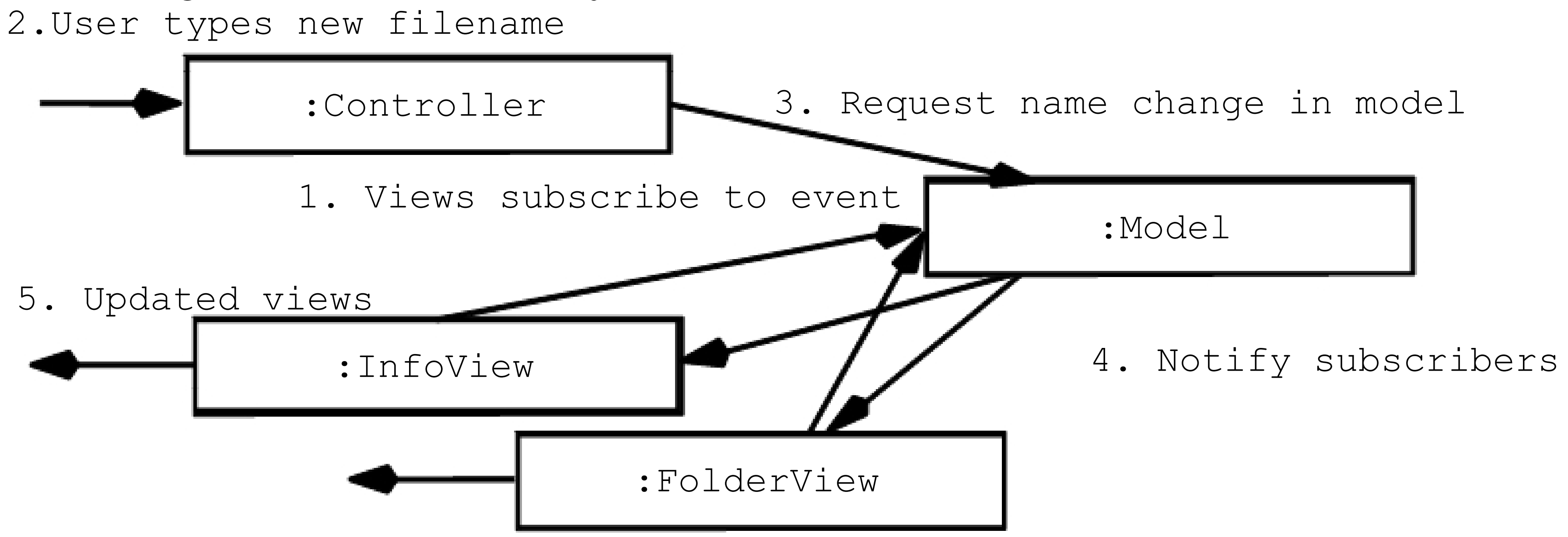
Comp-Based Software Engineering

Name	Size	Kind	Last Modifi
5SoftwareLifecycle.pdf	410K	Acrobat™ Exchange ...	Fri, Dec
5SoftwareLifecycle	371K	PowerPoint document	Fri, Dec
6Project Management	780K	PowerPoint document	Fri, Jan
6Project Management.pdf	293K	Acrobat™ Exchange ...	Fri, Jan
7SystemDesign.pdf	85K	Acrobat™ Exchange ...	Fri, Jan
7SystemDesignI.ppt	137K	PowerPoint document	Fri, Jan
8DesignRationale.pdf	358K	Acrobat™ Exchange ...	Fri, Jan
8DesignRationale.ppt	208K	PowerPoint document	Fri, Jan
9DesignPatterns2.ppt	130K	PowerPoint document	Thu, Ja
DesignPatterns.ppt	104K	PowerPoint document	Mon, De
Introduction.pdf	559K	Acrobat™ Exchange ...	Fri, Nov

- Two different views of a file system:
- The bottom window visualizes the Component Based Software Engineering folder
- The up windows, instead, visualizes information related to the 9DesignPatterns2.ppt file
- The file name is visualized into three different places

MVC: An example – 2/2

- 1. Both InfoView and FolderView subscribe the changes to the model File when created
- 2. The user enters the new filename
- 3. The Controller forwards the request to the model
- 4. The model actually changes the filename and notifies the operation to subscribers
- 5. InfoView and FolderView are updated so that the user perceives the changes in a consistent way


2.User types new filename
:Controller
3. Request name change in model
1. Views subscribe to event
:Model
5. Updated views
:InfoView
4. Notify subscribers
:FolderView

The benefits of MVC architectures

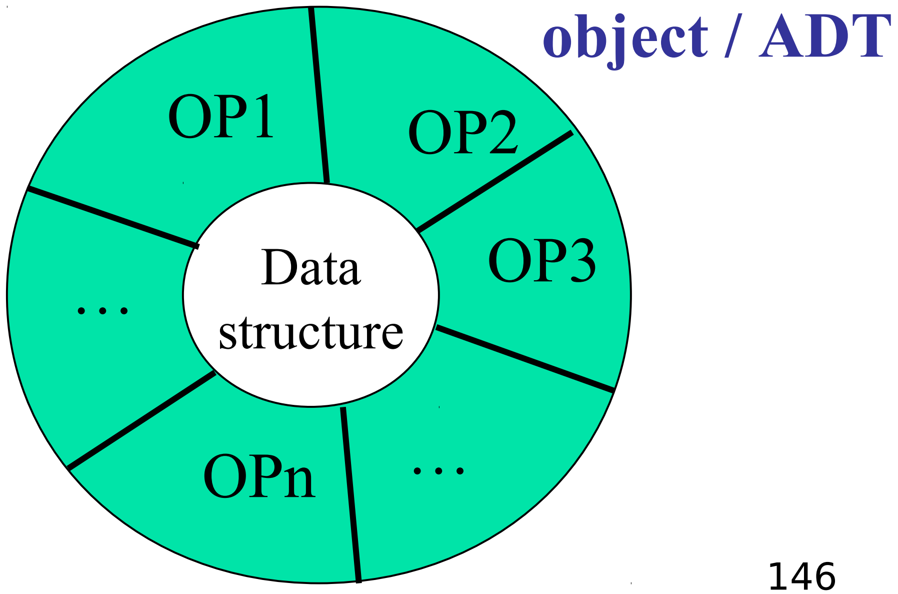
- The main reason behind the separation subsystems (Model, View and Controller) is that user interface are changed much more frequently then the knowledge about the application domain (Model)
- The MVC architecture is effective in the case of interactive systems, especially when several views of the same model have to be provided

DOA

Distributed Objects Architectures

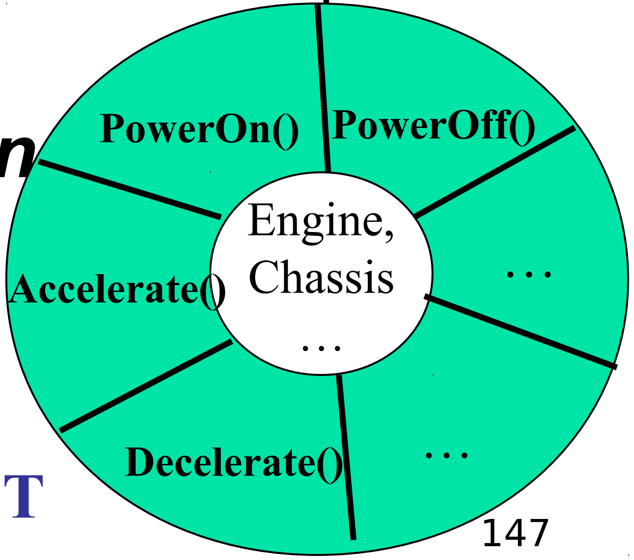
Objects

- An object is the encapsulation of a concrete data structure into the operations (those and only those) to access (read/write) it
- In typed languages, an alternative definition of object is an instance of an **Abstract Data Type**
  - ADT: the specification of a data structure and of the operations to access it
- Untyped languages (eg., Smalltalk) are typically interpreted
- Typed languages (C++, Java) are compiled
  - the compiler performs static type-checking



The power of objects

- In programming languages, subprograms (procedure / functions) and objects are “good” software modules
  - abstraction, interface, body, information hiding
- ... but subprograms provide only a ***functional abstraction***
  - The *hidden information* is the algorithm that implements the functionality
- Objects provide ***data abstraction***
  - The hidden information are:
    - data structures encapsulated
    - algorithms inside operations



A Car ADT

Object-orientation principles

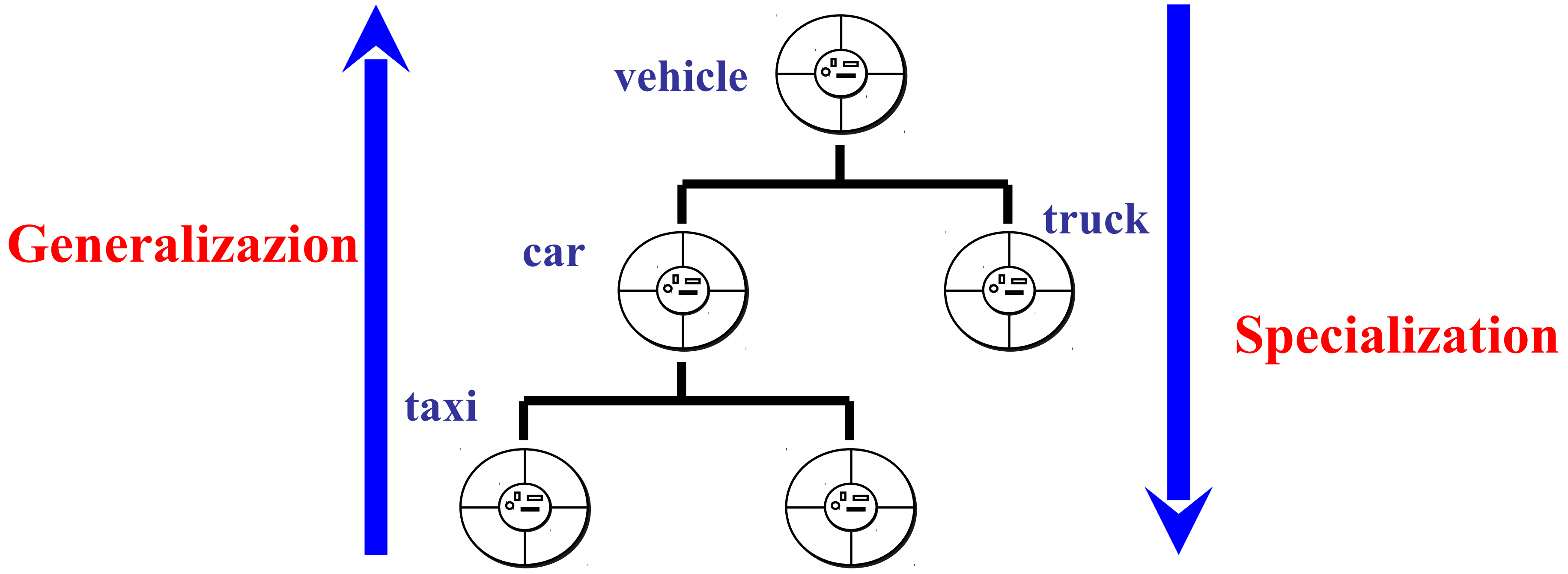
- **Object-based** techniques based on:
  - Classes
    - a way to implement abstract data types
  - Objects
    - instances of classes
- **Object-oriented** techniques are based on:
  - Classes
  - Objects
  - Inheritance, and / or
  - Polymorphism

Inheritance

- Inheritance induces a hierarchical structure into the software architecture
- It allows to build generalization/specialization relationships between classes
  - A class (**base class**) offers services that are common to a set of derived classes (**subclasses**)
  - Subclasses offer specialized services/behavior with respect to the base class
- Example:
  - Base class/type: Animal
  - Derived classes/types: Dog, Cat, Horse, …

Gen/Spec hierarchies

- **Generalization**: the process of moving from particular to general
- **Specialization**: the process of moving from general to particular



The benefits of inheritance

- Inheritance offers the advantage of reducing the development time, since **it minimizes the code to <u>add</u> when**:
  - **defining a new user type** (class) that is inherently a subtype of an existing one
  - **adapting an existing class** to new needs

Polymorphism

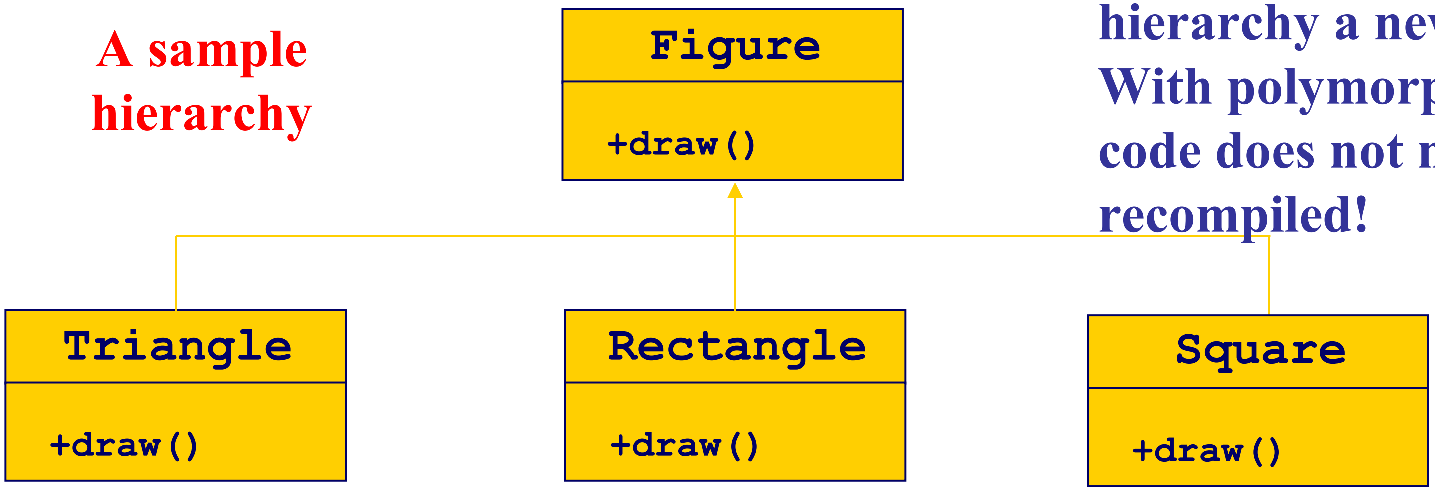
- Polymorphism is the property of an entity to assume several forms
- In O-O languages, polymorphism refers to the property of a reference (to an object) to refer in different moments to objects of different classes

existing code (eg,: main program):

for i = 1 to N do A[i]->draw()

What about adding to the hierarchy a new class Circle? With polymorphism, the existing code does not need to be recompiled!

A sample hierarchy



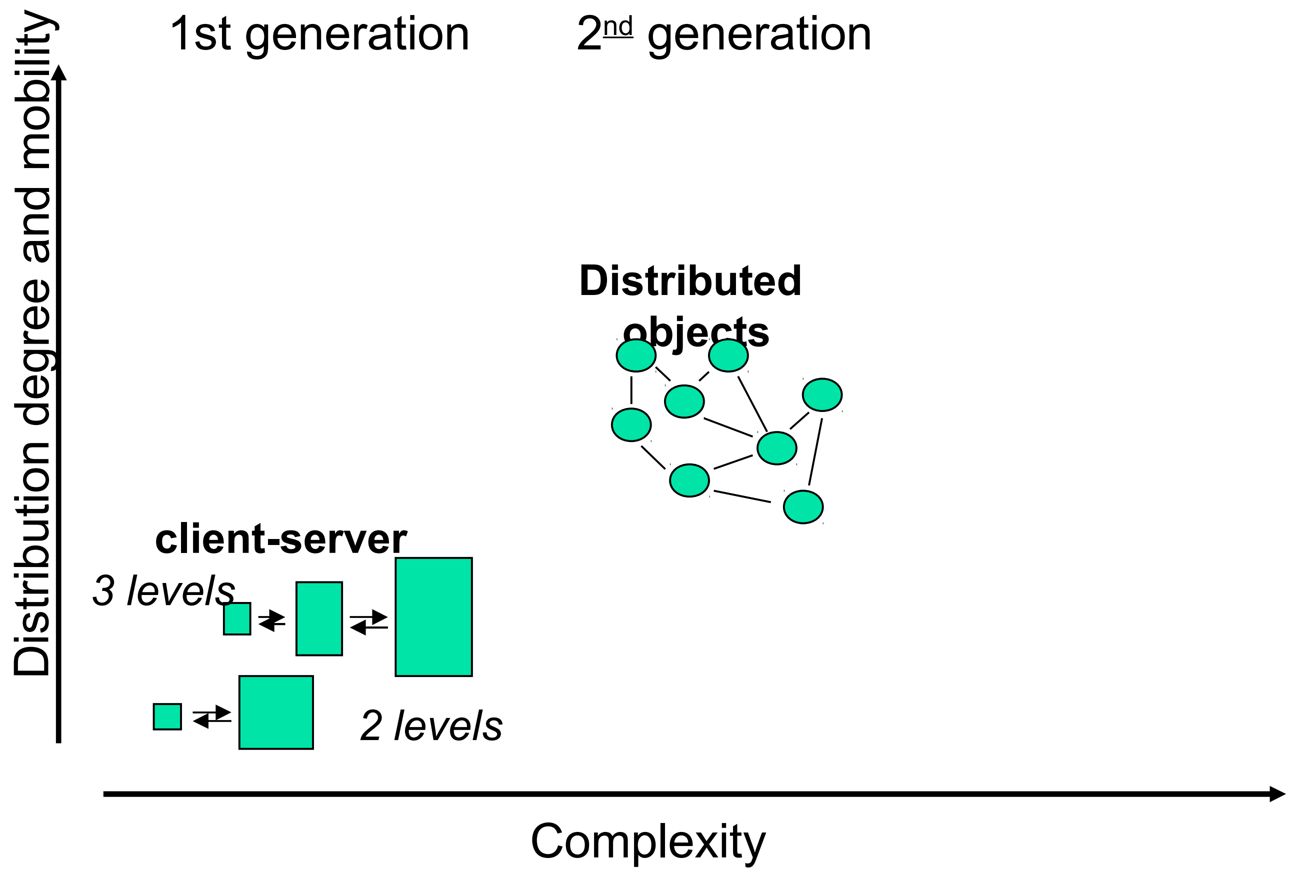
The benefits of polymorphism

- Polymorphism offers the advantage of reducing the development time, in that **it minimizes the existing code to <u>modify</u> when adding new user types** (classes)
- For this reason, **polymorphism supports the property of extensibility** of a software systems

- Polymorphism is typically implemented in O-O languages by means of *late binding*
  - *Late binding* consists in defining run time (instead of compilation time) the code to execute upon method invocation, depending on the actual (current) object type

Distributed objects arch. – 1/4

- The second generation of software distributed systems is based on DOA architectures



Distributed objects arch. – 2/4

- Distributed objects architectures extend the principles of object orientation to a distributed environment
- Schematically:

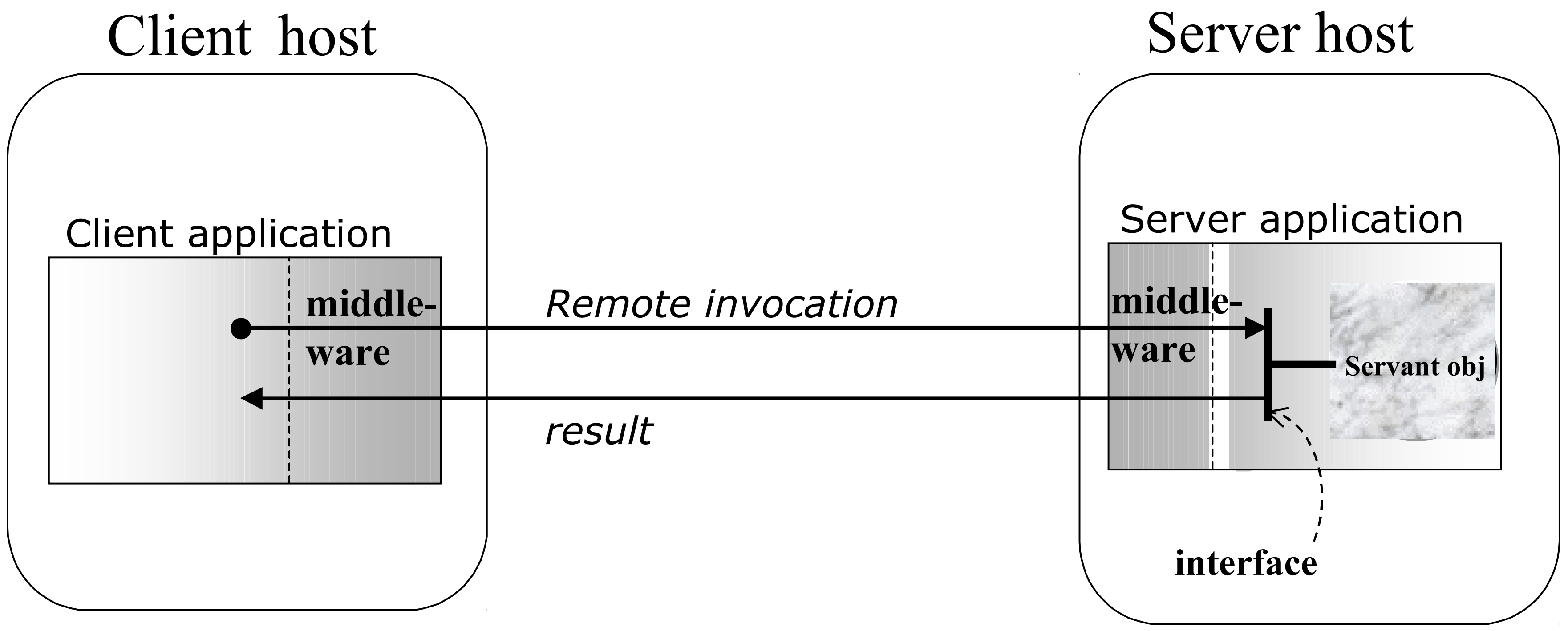
DO = OOP + C/S

Distributed objects = object-oriented programming + client-server

- Both objects and servers:
  - Are passive entities
  - Are able to provide services (operations)
  - Have got an interface
  - Are unaware of the clients

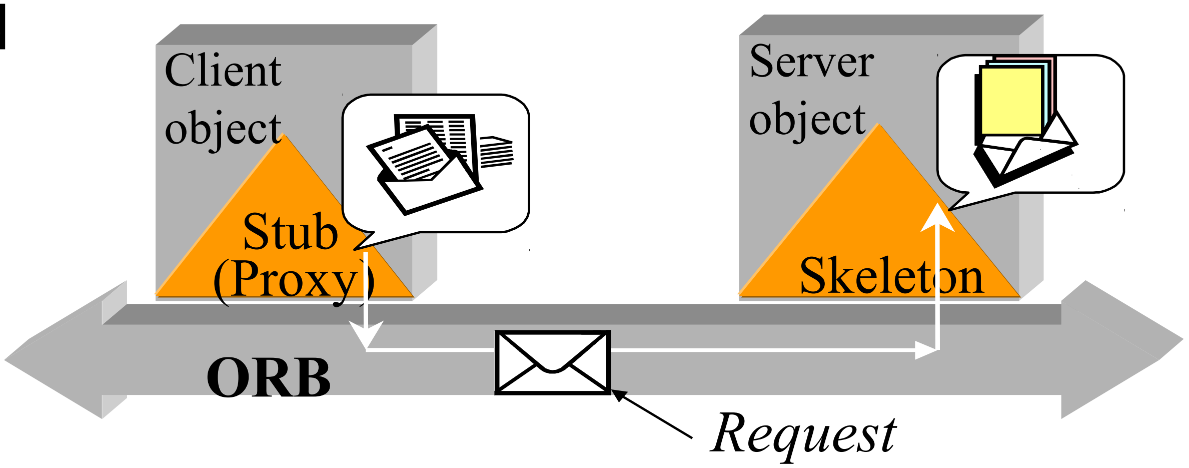
Distributed objects arch. – 3/4

- In a DOA, components are objects which are deployed on different machines and which communicate by invoking remote methods



Distributed objects arch. – 4/4

- DOAs are based on ***middleware platforms*** (DOM), that allow to call a remote method as if it were local
- DOA platforms may be
  - based on an Object Request Broker
  - not ORB-based



- Examples of DOM technologies are:
  - CORBA (OMG standard); ORB-based, language independent
  - Java RMI; not ORB-based, language-dependent

DOA pitfalls

- In O-O techniques, the reusable software modules (classes) are not executable entities
- In O-O applications, dependencies between objects are not made explicit in their interfaces
  - An object's interface states what it offers (the services it provides), not what it uses (the object/services it requires)
- Interactions among objects are hidden in the application code
- Objects are often a too fine-grained abstraction for large distributed complex systems

→ from objects to components

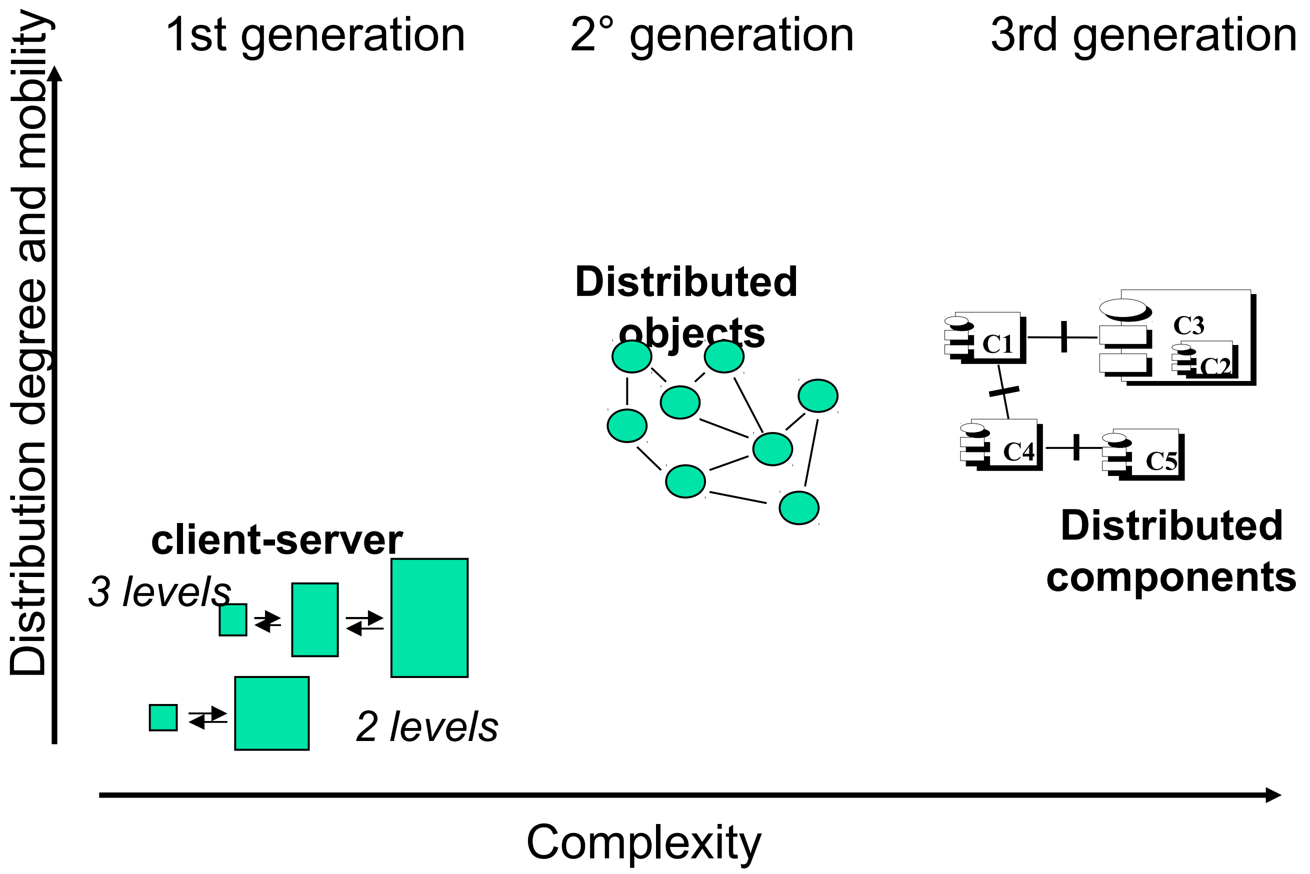
Component-based Architectures

Components

- Component:
  - An executable software module with a clear interface
  - Dependencies from other modules are stated explicitly in its interface
  - Can be released and executed autonomously
  - Can be assembled with other modules to build more complex modules
- Component Based Software Engineering (**CBSE**)
  - Better reuse opportunities wrt O-O
    - Reuse whole applications, not only classes
- The **Component Model** (CM) is an evolution of the O-O model

Component-based arch. – 1/2

- The third generation of distributed software systems
  - Components are coarser grained than objects

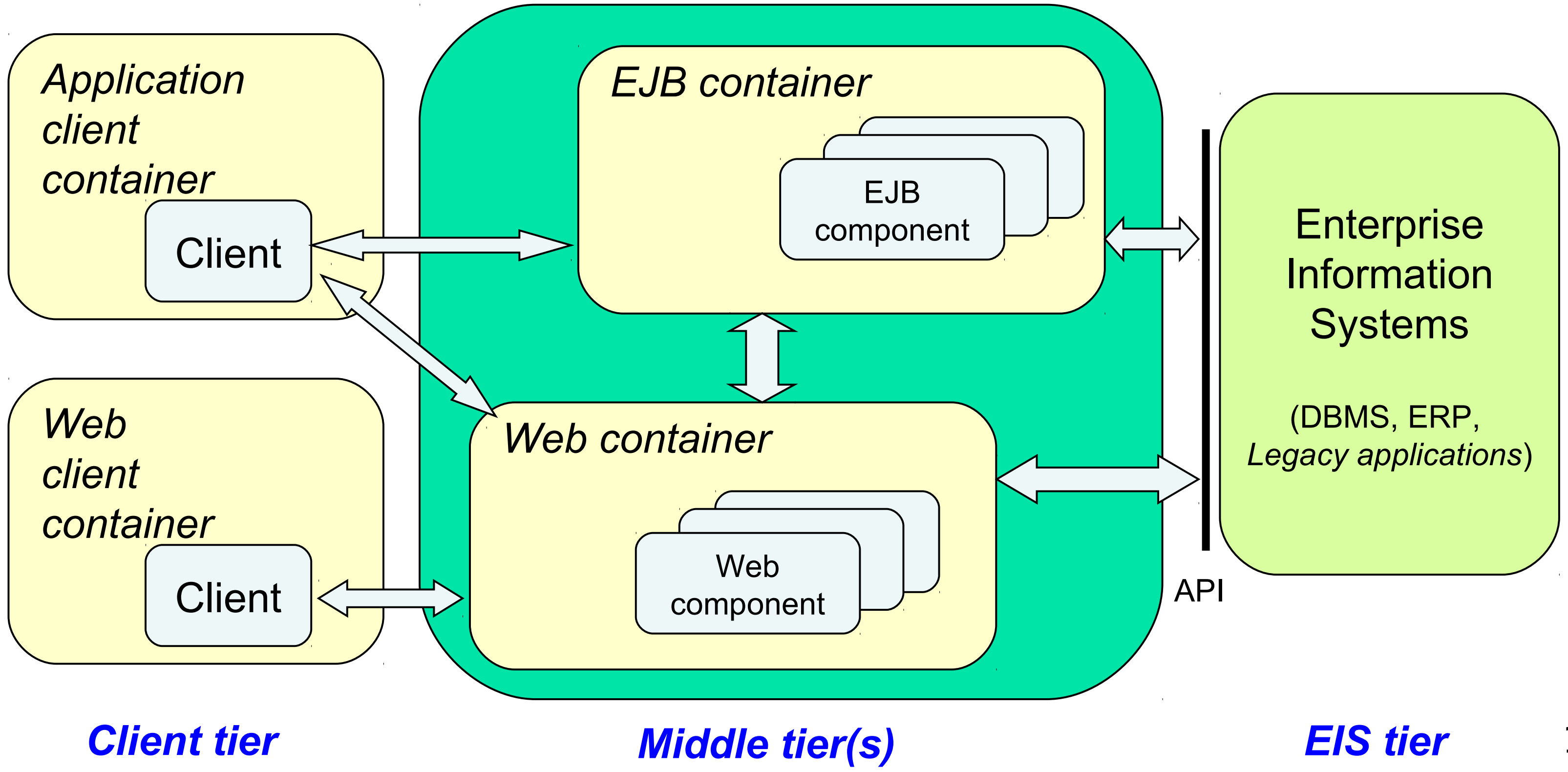


Component-based arch. – 2/2

- Component-based architectures rely on component middleware platforms
- Component platforms are based on containers
- **Container**:
  - Components' execution environment
  - Manages components' life cycle (creation, activation, execution, de-activation)
    - Example: a server has to be running when a client issues a request; a component can be active but not in execution till requested, in order to avoid resource consumption
  - Offers pre-defined standard services
    - Security,
- Advantage: much of the development effort is shifted from programming to assembly/deployment

A component middleware: J2EE

- Sun Java 2 Enterprise Edition (J2EE)
- Three-tiered architecture
- Four types of components
  - generic client, web client, web server, Ent. Java Beans

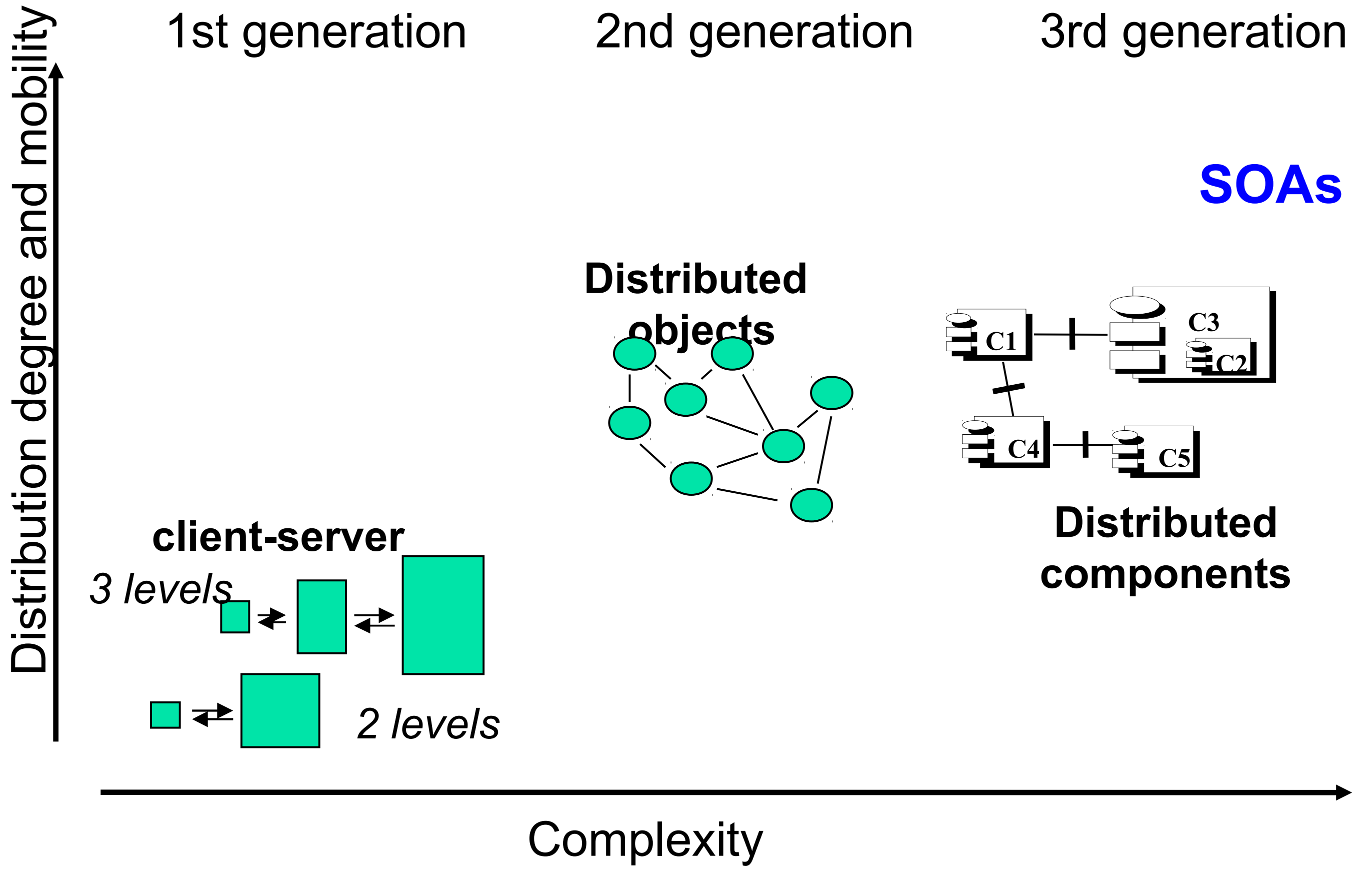


(SOA)

Service-Oriented Architectures

Service-oriented architectures – 1/3

- SOAs represent an evolution of component based architectures (i.e., the 3th generation of distributed software systems)



Service-oriented architectures – 2/3

- Service-oriented architecture (SOA) is an approach to <u>loosely coupled</u>, <u>protocol independent</u>, <u>standards-based</u> distributed computing where **software resources available on the network are considered as Services**
- SOA is believed the enterprise technology solution that provides the agility and flexibility the business users have been looking for by leveraging the integration process through composition of the services spanning multiple enterprises

Service-oriented architectures – 3/3

- The following guiding principles define the ground rules for development, maintenance, and usage of the SOA:
  - Reuse, granularity, modularity, composability, componentization, and interoperability
  - Compliance to standards (both common and industry-specific)
  - Services identification and categorization, provisioning and delivery, and monitoring and tracking

SOA principles – 1/2

- Service **encapsulation** - Many web-services are consolidated to be used under the SOA Architecture. Often such services have not been planned to be under SOA
- Service **loose coupling** - Services maintain a relationship that minimizes dependencies and only requires that they maintain an awareness of each other
- Service **contract** - Services adhere to a communications agreement, as defined collectively by one or more service description documents
- Service **abstraction** - Beyond what is described in the service contract, services hide logic from the outside world

SOA principles – 2/2

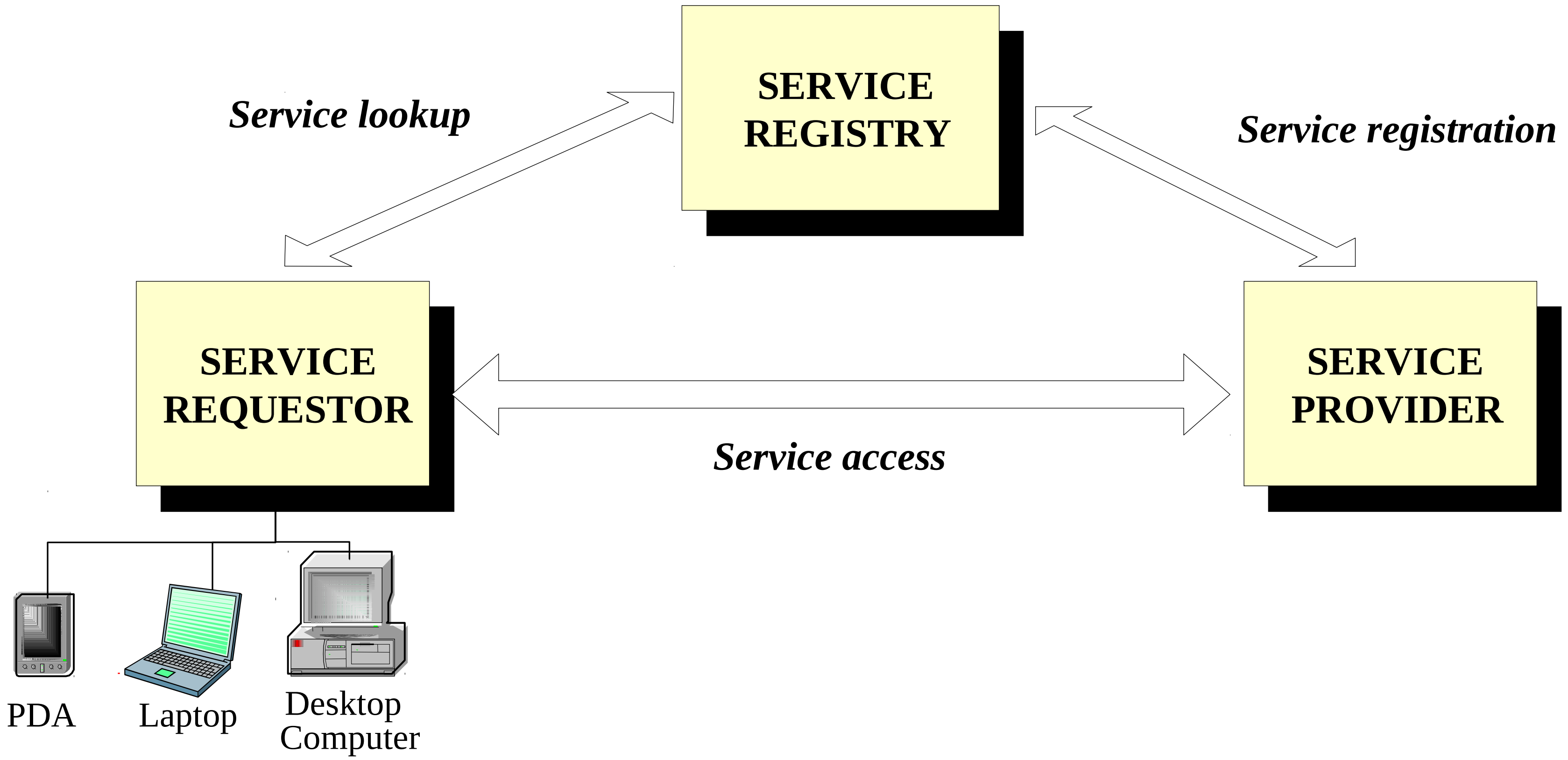
- Service **reusability** - Logic is divided into services with the intention of promoting reuse
- Service **composability** - Services can be coordinated/assembled to form composite services
- Service **autonomy** – Services have control over the logic they encapsulate
- Service **optimization** – All else equal, high-quality services are generally considered preferable to low-quality ones
- Service **discoverability** – Services are designed to be outwardly descriptive so that they can be found and assessed via available discovery mechanisms

SOA building blocks – 1/2

- Service **Consumer** (also known as Service **Requestor**): it locates entries in the Registry using various find operations and then binds to the service provider in order to invoke one of its services
- Service **Provider**: it creates a service and publishes its interface and access information to the Service registry
- Service **Registry** (also known as Service **Broker**): it is responsible for making the Service interface and implementation access information available to any potential service requestor

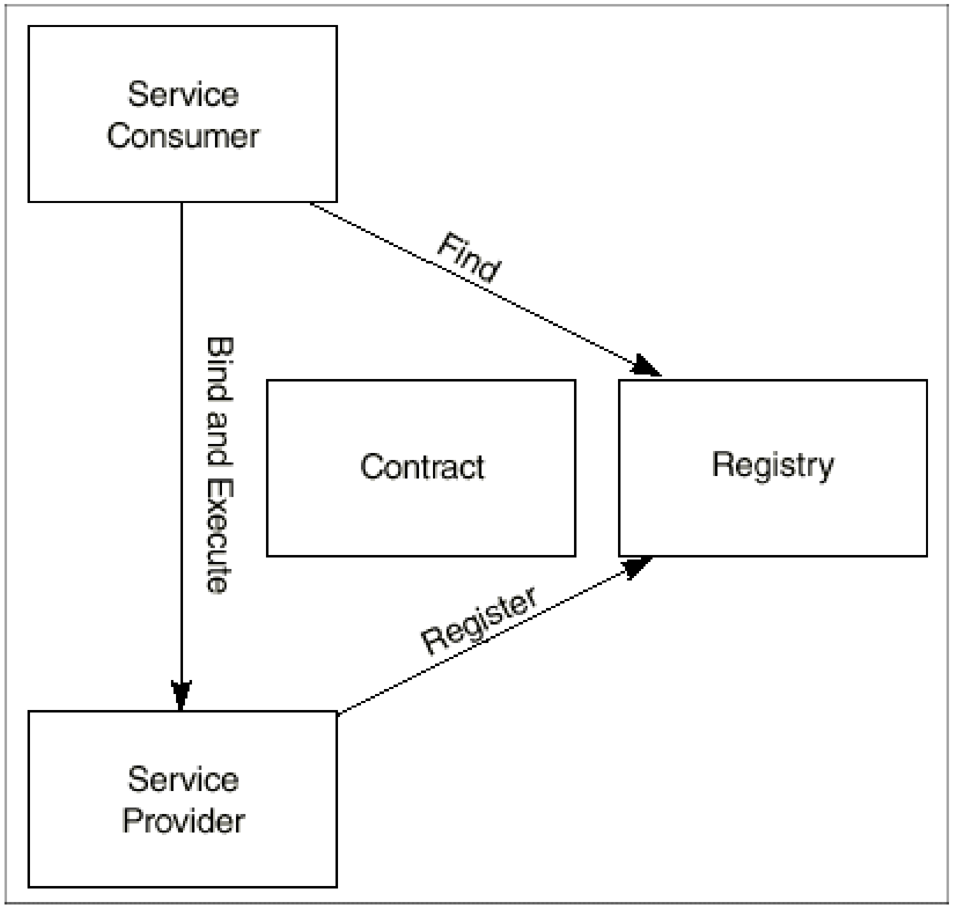
SOA building blocks – 2/2

- Service registration (provider) and lookup (requestor) in the registry
- Service access

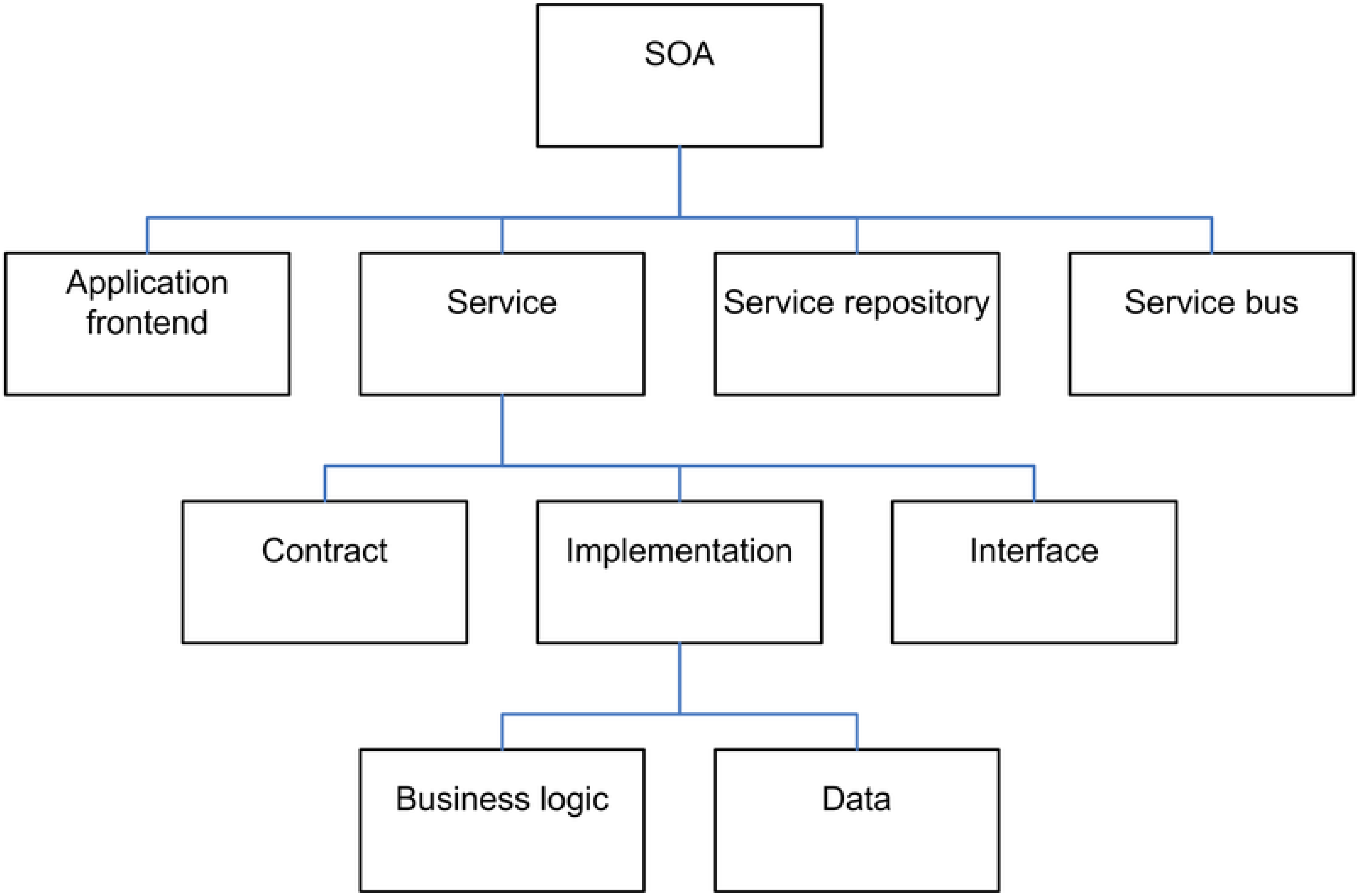


SOA model

- 1 Service Consumer
- 2 Service Provider
- 3 Service Registry
- 4 Service Contract
- 5 Service Proxy
- 6 Service Lease



SOA elements



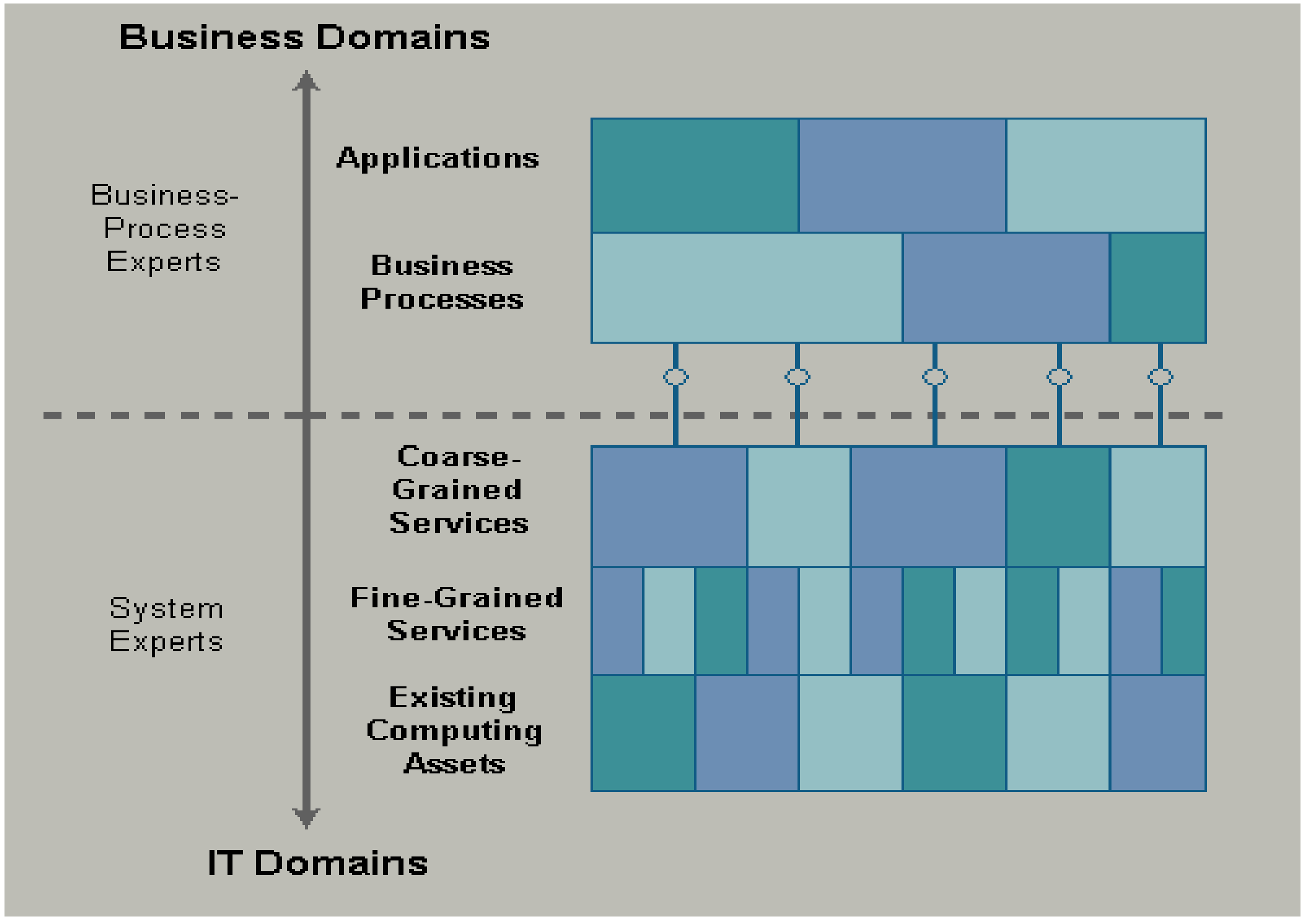
SOA characteristics

- The software components in a SOA are services based on standard protocols
- Services in SOA have minimum amount of interdependencies
- Communication infrastructure used within an SOA should be designed to be independent of the underlying protocol layer
- Offers coarse-grained business services, as opposed to fine-grained software-oriented function calls
- Uses service granularity to provide effective composition, encapsulation and management of services

Service granularity

- Service granularity refers to the scope of functionality a service exposes
- Fine-grained services provide a small amount of business-process usefulness, such as basic data access
- Coarse-grained services are constructed from fine-grained services that are intelligently structured to meet specific business needs.

Service composition



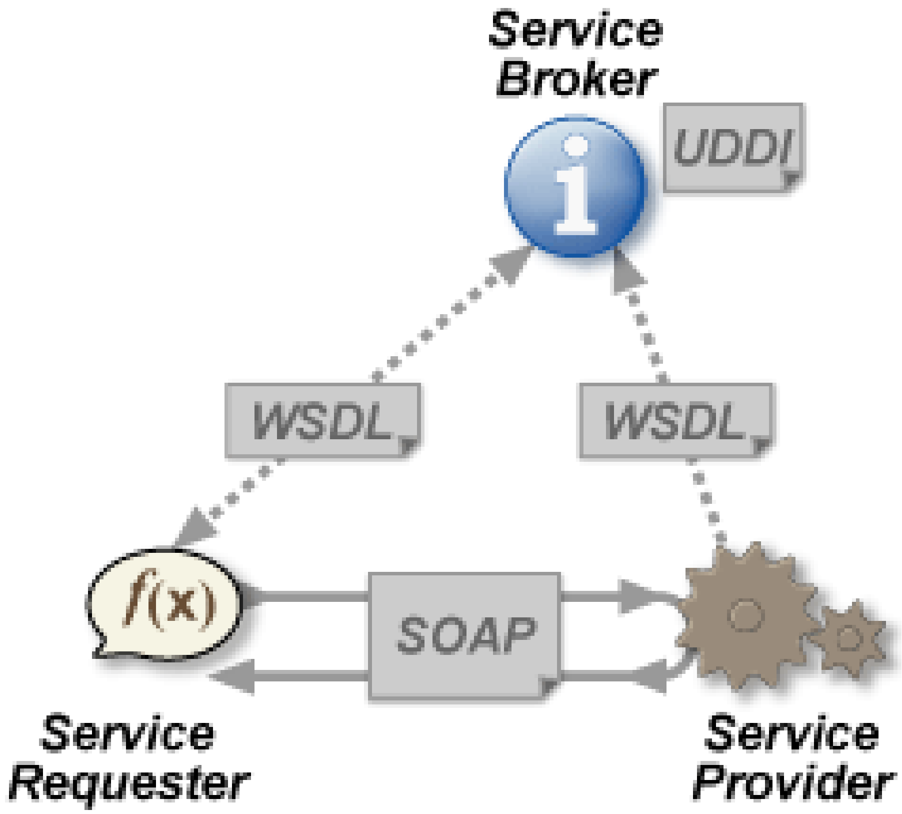
Jeff Hanson, *Coarse-grained Interfaces Enable Service Composition in SOA*, JavaOne, August 29, 2003

Web Services

- A W3C standard suite for SOAs
- Definition:
  - ‘A Web service is a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. A Web service supports direct interactions with other software agents using XML based messages exchanged via internet-based protocols.’

WS underlying technologies

- The basic standards for web services are:
  - XML (Extensible Markup Language)
  - SOAP (Simple Object Access Protocol)
  - WSDL (Web Services Description Language)
  - UDDI (Universal Description, Discovery and Integration)



SOA-related standards

- XML 1.0 fairly stable, although Schema are in the process of replacing DTDs (currently Schema 1.1 being worked on).
- SOAP 1.2
- WSDL 2.0 (coming out, 1.2 current)
- UDDI version 3 (Aug 2003)
- BPEL 1.1 (Business Process Execution Language)
- choreography description language (web services work flows)
  - started January 2003

Web Services Architecture

- Web Services involve three major roles
  - Service Provider
  - Service Registry
  - Service Consumer

- Three major operations surround web services
  - Publishing – making a service available
  - Finding – locating web services
  - Binding – using web services

Service publication

- In order for someone to use your service they have to know about it
- To allow users to discover a service it is published to a registry (UDDI)
- To allow users to interact with a service you must publish a description of it’s interface (methods & arguments)
- This is done using WSDL

Making a service available

- Once you have published a description of your service you must have a host set up to serve it.
- A web server is often used to deliver services (although custom application – application communication is also possible).
- This functionality has to be added to the web server. In the case of the *Apache* web server a ‘container’ application (*Tomcat*) can be used to make the application (servlet) available to *Apache* (deploying).

WS and existing protocols

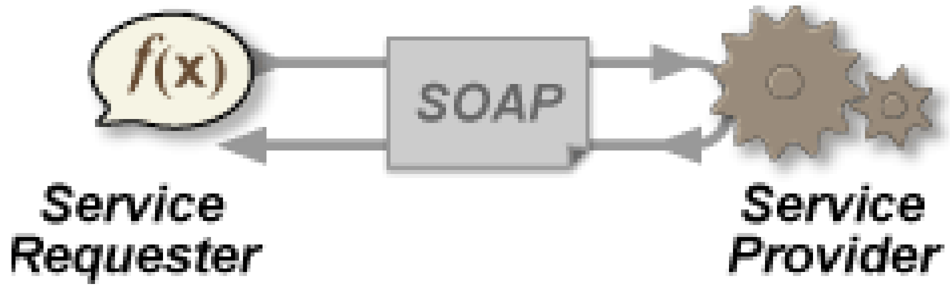
- Web services are layered on top of existing, mature transfer protocols
- HTTP, SMTP are still used over TCP/IP to pass the messages
- Web services (like grids) can be seen as a functionality enhancement to the existing technologies

WS and XML

- All Web Services documents are written in XML
- XML Schema are used to define the elements used in Web Services communication

WS and SOAP

- SOAP is the protocol actually used to communicate with the Web Service
- Both the request and the response are SOAP messages
- The body of the message (whose grammar is defined by the WSDL) is contained within a SOAP “envelope”
- “Binds” the client to the web service



WSDL

- Describes the Web Service and defines the functions that are exposed in the Web Service
- Defines the XML grammar to be used in the messages
- Uses the W3C Schema language

WS and UDDI

- UDDI is used to register and look up services with a central registry
- Service Providers can publish information about their business and the services that they offer
- Service consumers can look up services that are available by
  - Business
  - Service category
  - Specific service

Web Services and SOAs

- Web Services are an open standards-based way of creating and offering SOAs
- Web Services are able to exchange structured documents that contain different amounts of information, as well as information about that information, known as **metadata**
- In other words, Web Services can be coarse grained. Such coarse granularity is one of the most important features of SOAs

Re-architecture to SOA

- Encapsulate software components, applications and underlying systems with Web Services interfaces
- Compose (virtualizing) these fine-grained functional Web Services into coarse-grained business services

Re-architecture to SOA: benefits

- Provides location independence: Services need not be associated with a particular system on a particular network
- Protocol-independent communication framework renders code reusability
- Offers better adaptability and faster response rate to changing business requirements
- Allows easier application development, run-time deployment and better service management
- Loosely coupled system architecture allows easy integration by composition of applications, processes, or more complex services from other less complex services
- Provides authentication and authorization of Service consumers, and all security functionality via Services interfaces rather than tightly-coupled mechanisms
- Allows service consumers (ex. Web Services) to find and connect to available Services dynamically

Why not SOA

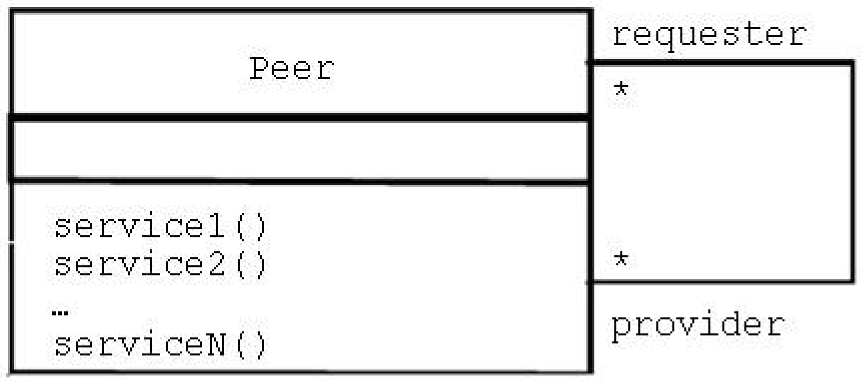
- For a stable or homogeneous enterprise IT environment, SOA may not be important or cost effective to implement
- If an organization is not offering software functionality as services to external parties or not using external services, which require flexibility and standard-based accessibility, SOA may not be useful
- SOA is not desirable in case of real time requirements because SOA relies on loosely coupled asynchronous communication

P2P

Peer-to-peer Architectures

Peer-to-peer architectures – 1/3

- They can be considered a generalization of the client/server architecture
- Each subsystem is able to provide services, as well as to make requests
  - Each subsystem acts both as a server and as a client



Peer-to-peer architectures – 2/3

- In P2P architectures, all nodes have the same abilities, as well as the same responsibilities. All communications are (potentially) bidirectional
- The goal:
  - To share resources and services (data, CPU cycles, disk space, …)
- P2P systems are characterized by:
  - Decentralized control
  - Adaptability
  - Self-organization and self-management capabilities

Peer-to-peer architectures – 3/3

- Functional characteristics of typical P2P systems
  - File sharing system
  - File storage system
  - Distributed file system
  - Redundant storage
  - Distributed computation
- Nonfunctional requirements
  - Availability
  - Reliability
  - Performance
  - Scalability
  - Anonymity

P2P architectures: brief history

- Although they were proposed 30 years ago, they mainly evolved in the last decade
- File sharing systems contributed to increase the interest (Napster, 1999; Gnutella, 2000)
- In 2000, the Napster client has been downloaded by 50 millions users
  - It has recorded a traffic peak of 7 TB in a day!
- Gnutella followed Napster’s footprint
  - The first release has been delivered on March 14th, 2003, by (14 hours!)
  - Host servers are listed at gnutellahost.com

The phases of a P2P application

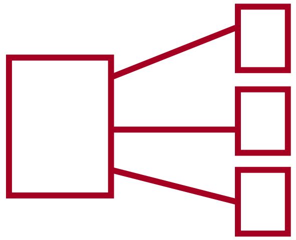
- A P2P application is organized in three phases:
  - **Boot**, during which a peer finds the networks and actually performs the connections (P2P boot is made rarely)
  - **Lookup**, during which a peer looks for a provider for a given service/information (generally providers are SuperPeers)
  - **Resource sharing**

P2P classification – 1/2

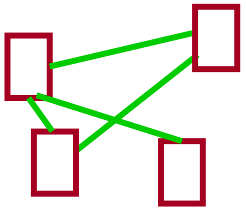
- P2P applications can be categorized as follows:
  - **Pure P2P applications**:
    - All the phases are based on P2P
  - **P2P applications** :
    - lookup and resource sharing are performed according to the P2P paradigm;
    - Some SERVERS are used to make the boot;
  - **Hybrid P2P applications** :
    - The resource sharing is performed according to the P2P paradigm;
    - Some SERVERS are used to make the boot;
    - Specific peers are used to perform lookup.

P2P classification – 2/2

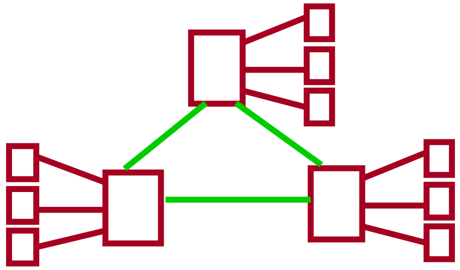
- With respect to lookup phase:
  - Centralized Lookup
    - Centralized index of providers



  - Decentralized Lookup
    - Distributed index



  - Hybrid Lookup
    - Several centralized systems which are linked into a decentralized system



P2P: scalability – 1/2

- The performances of a P2P system (e.g., the time to find a file) become worse as the number of nodes increases (linearly)
  - The “work” for a node increases linearly with respect to the number of nodes
- The protocols used by Napster and Gnutella are not scalable
- A second generation of P2P protocols has been realized to improve the scalability which support DHT (Distributed Hash Table).
  - Examples are Tapestry, Chord, Can, Viceroy, Koorde, kademlia, kelips …

P2P: scalability – 2/2

- The degree of scalability of a given protocol depends on the efficiency of the adopted routing algorithm (lookup)
- In this directions, two are the main objectives:
  - To minimize the number of message to complete the lookup
  - To minimize, for each node, information about other nodes in the network
- DHT differ into the routing strategy they adopt

References

- **Shaw** & **Garlan,** *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996
- **Bass, Clemens, Kazman**, *Software Architecture in Practice*, 2nd ed, Addison Wesley, 2003 `softarchpract.tar.hu`
- **Rozanski & Woods,** Software systems architecture: Working with Stakeholders Using Viewpoints and Perspectives, Addison Wesley, 2005
- **Garland**, *Large-Scale Software Architecture: A Practical Guide using UML*, Addison-Wesley, 2005
- **Taylor & Medvidović & Dashofy**, *Software Architecture: Foundations, Theory, and Practice*, Wiley 2009
- **Clements**, *Documenting Software Architectures: Views and Beyond*, The SEI Series in Software Engineering, Addison-Wesley, 2010
- **Clemens, Kazman, Klein**, *Evaluating Software Architectures: Methods and Case Studies*, SEI Series in Software Engineering, Addison-Wesley, 2001

Web sites

- www.sei.cmu.edu/architecture
- www.iso-architecture.org
- www.viewpoints-and-perspectives.info
- www.softwarearchitectureportal.org
- enterprise-architecture-wiki.nl/mediawiki/index.php/Main_Page
- www.bredemeyer.com/definiti.htm
- www.booch.com/architecture
- www.ivencia.com/index.html?/softwarearchitect/