Chapter 7, System Design – Architecture Organization

Object-**O**riented
Software **C**onstruction

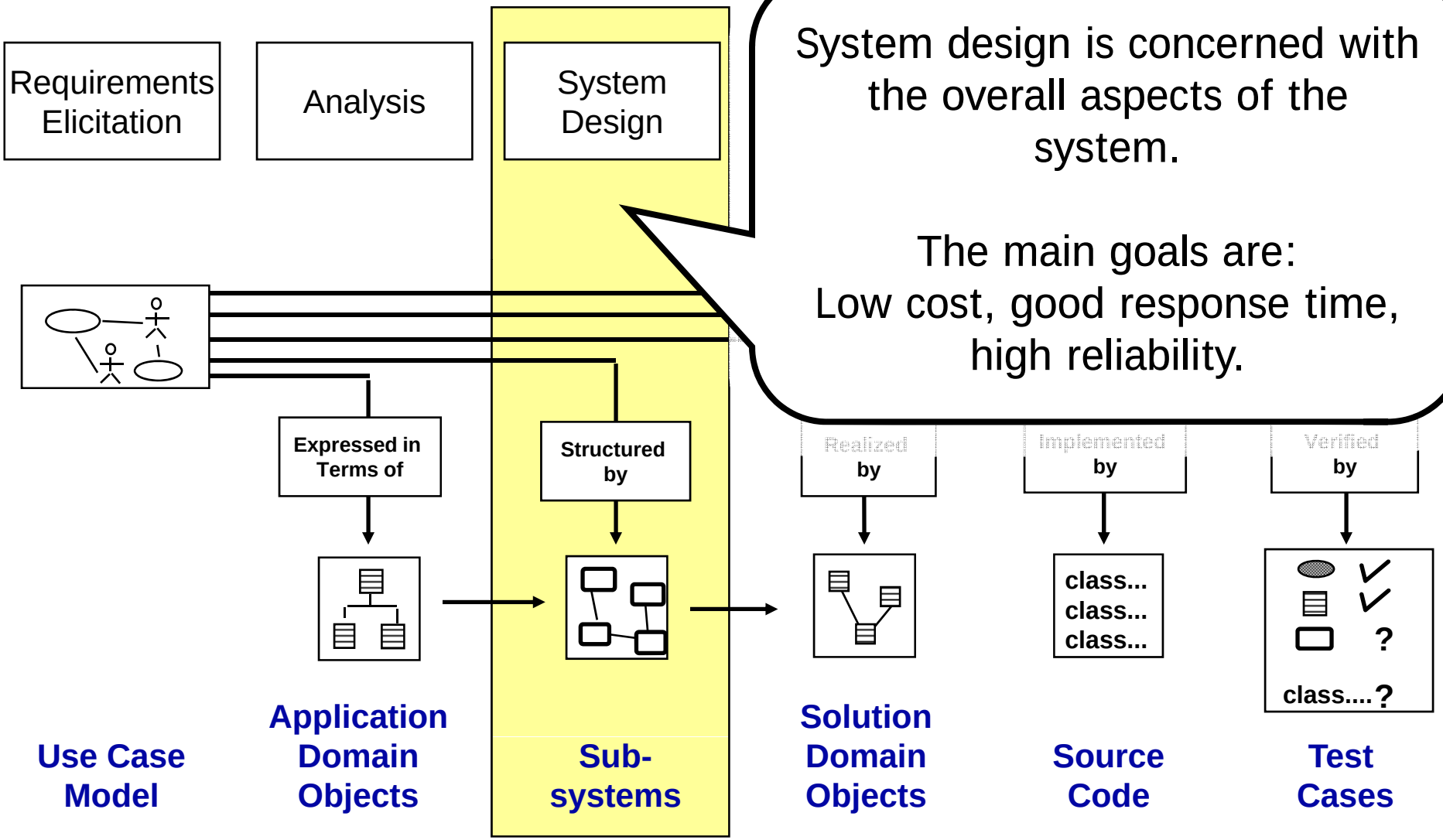
Armin B. Cremers, Tobias Rho, Daniel Speicher &
Holger Mügge
(based on Bruegge & Dutoit)

Overview

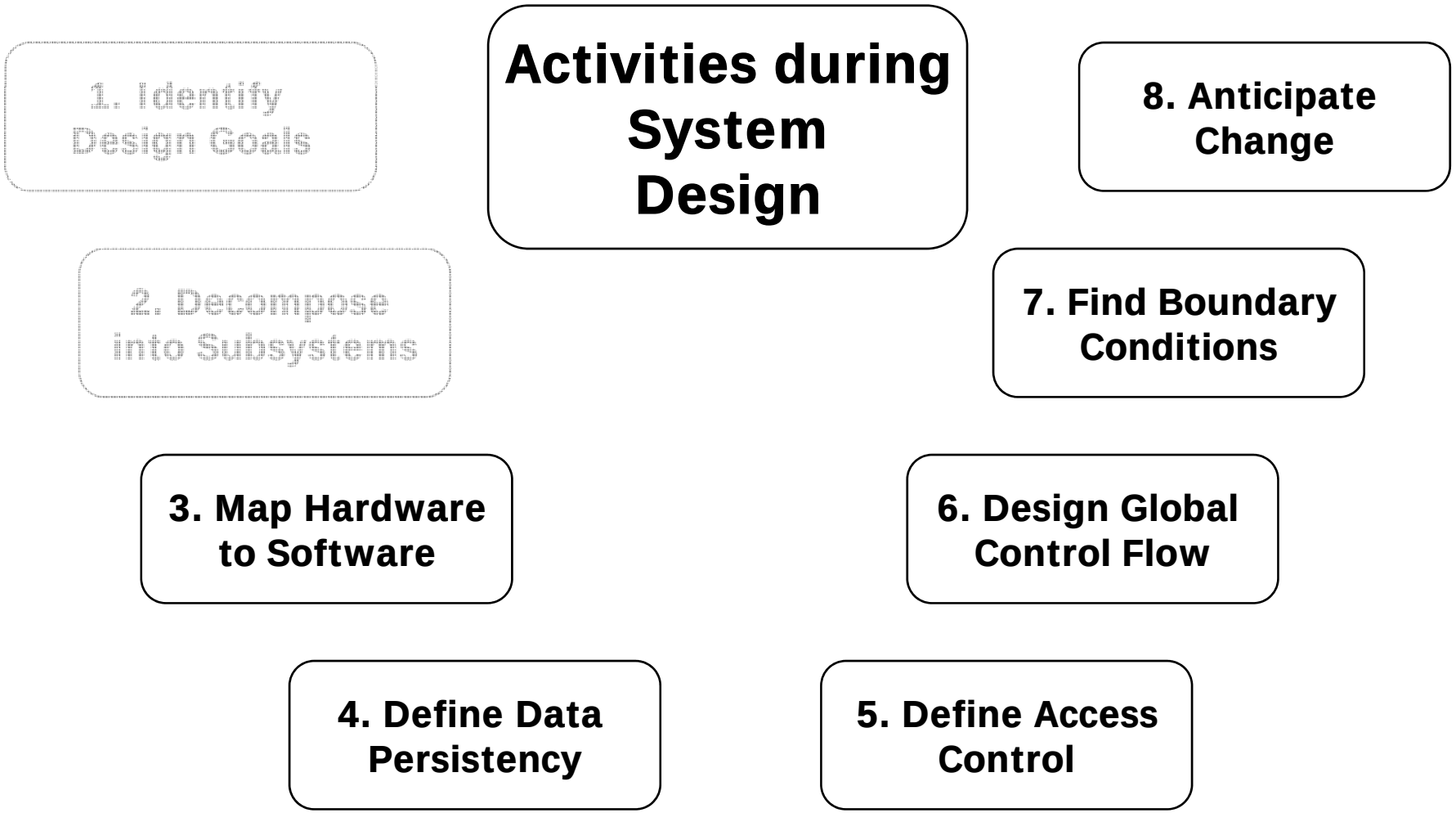
- Where are we right now? (last Lecture):
  - Overview of System Design
  - Subsystem Decomposition
    - Architectural Styles (Client-Server, Peer-to-Peer, …)

- Overview of this Lecture:
  - Architecture Organization
    - Refinement of Decomposition
    - Involvement of more technical issues

Software Lifecycle Activities ...system design



Activities during System Design



Aspects for Architecture Organization

- 3. Mapping of Subsystem to Hardware
  - Determine the hardware configuration
- Selection of Off-The-Shelf Components
  - Purchase existing components to realize subsystems more economically
- 4. Design of Persistent Data Management
  - Realize Storage System for objects that outlive a single execution of the system
- 5. Define Access Control
  - Protect Shared objects for concurrent access
- 6. Define Global Control flow
  - Determine the sequence of operations within the subsystems
- 7. Define Boundary Conditions
  - Determine conditions for system initialization and shut-down

3. Hardware Mapping

This activity addresses two questions:

- What is the most appropriate hardware environment?
- How are objects and subsystems mapped on the chosen hardware?
  - Mapping Objects onto Reality: Processor, Memory, Input/Output
  - Mapping Associations onto Reality: Connectivity

Hardware Mapping:
Mapping the Subsystems – Important Issues

- Processor issues:
  - Is the computation rate too demanding for a single processor?
  - Can we get a speedup by distributing tasks across several processors?
  - How many processors are required to maintain steady state load?
- Memory issues:
  - Is there enough memory to buffer bursts of requests?
- I/O issues:
  - Does the response time exceed the available communication bandwidth between subsystems ?

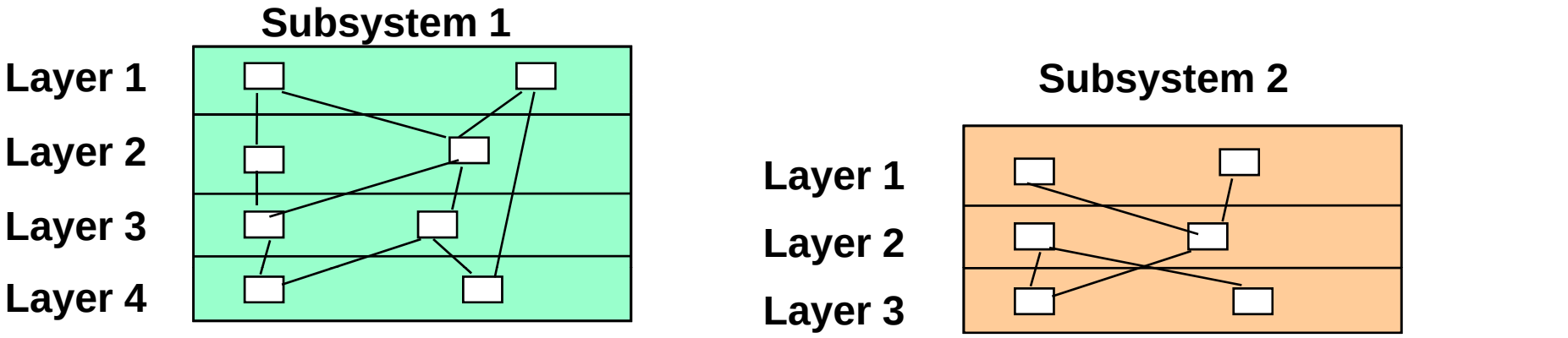
➔ Still worth considering in the 21th century ;-) ?

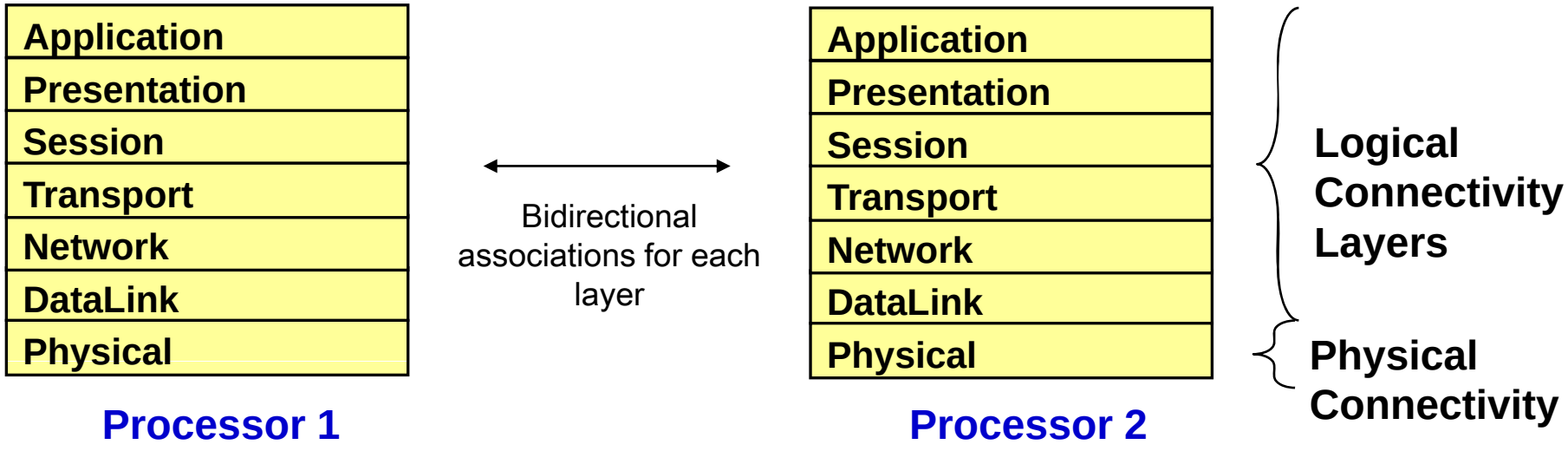
Hardware Mapping:
Mapping the Subsystems Associations

- Describe the *physical connectivity* of the hardware:
  - Which associations in the object model are mapped to physical connections?
  - Which of the client-supplier relationships in the analysis/design model correspond to physical connections?

- Describe the *logical connectivity* (subsystem associations):
  - Identify associations that do not directly map into physical connection. Distinction:
    - Logical Connection within a local system
    - Logical Connection within a distributed system
    - → How should these associations be implemented?

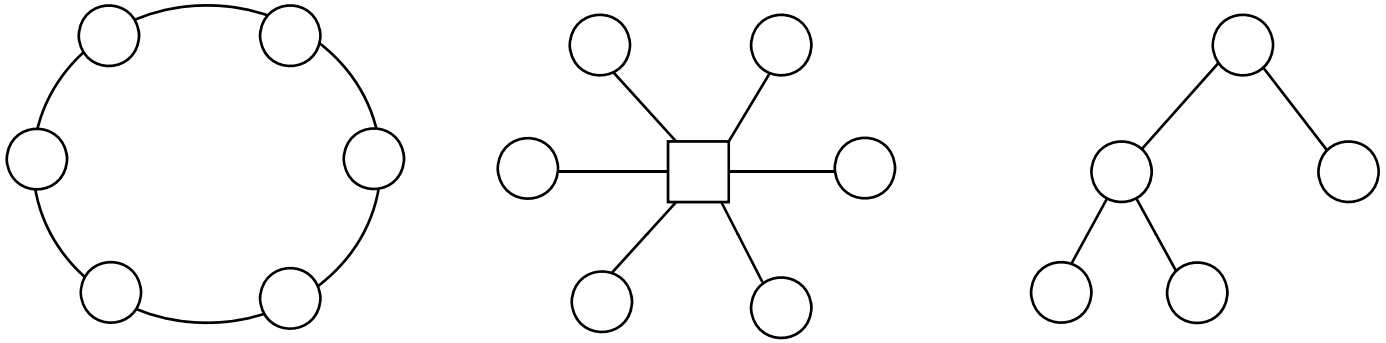
Hardware Mapping:
Logical vs. Physical Connectivity





Hardware Mapping:
Connectivity Mapping Questions

- What is the connectivity among physical units?



- What is the appropriate communication protocol between the subsystems, especially Distributed Systems?
  - Function of required bandwidth, latency and desired reliability, desired quality of service (QoS)
- Is certain functionality already available in hardware?

Software Mapping:
Selecting additional Software

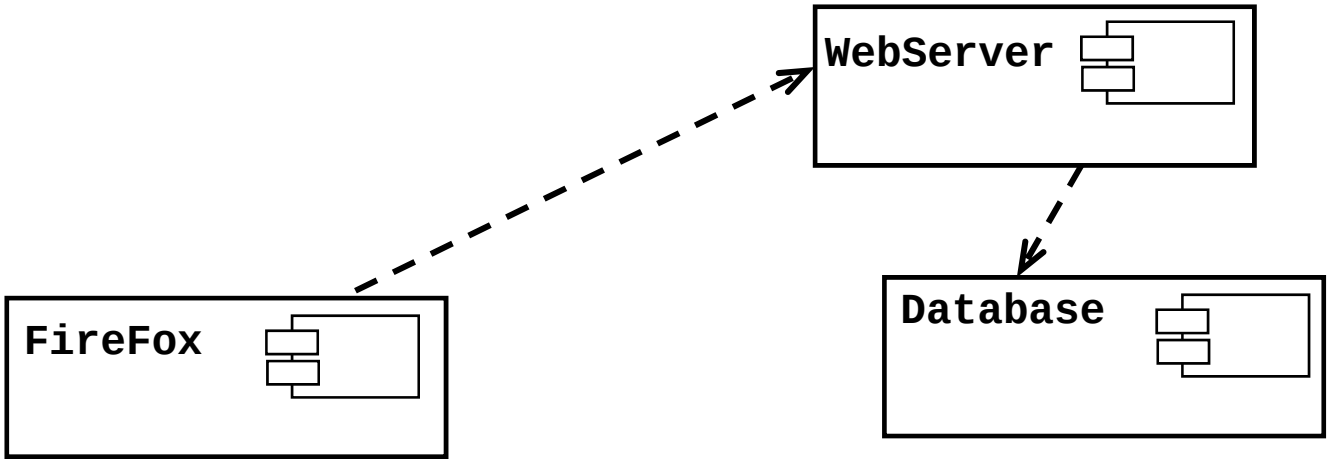
- Selection of existing software subsystems (components, services, frameworks) that realize technical aspects
  - Communication subsystems
  - Data Management subsystems
- COTS (Commercial Off-the-Shelf) Components
  - Products that are designed to be easily installed and to interoperate with existing system components.
  - Mostly no adaptation at client-side is necessary
  - Mass-produced → relatively low cost.
- Selecting a Virtual Machine:
  - Environment including operating system and any additional software components that are needed (e.g. DBMS)
  - Reduces the distance between the system and hardware
  - Reduction of development work (concentration on business code)

Drawing Hardware/Software Mappings in UML (Excurse)

- System design must model static and dynamic structures:
  - Component Diagrams for static structures
    - show the structure at design time or compilation time
  - Deployment Diagram for dynamic structures
    - show the structure of the run-time system

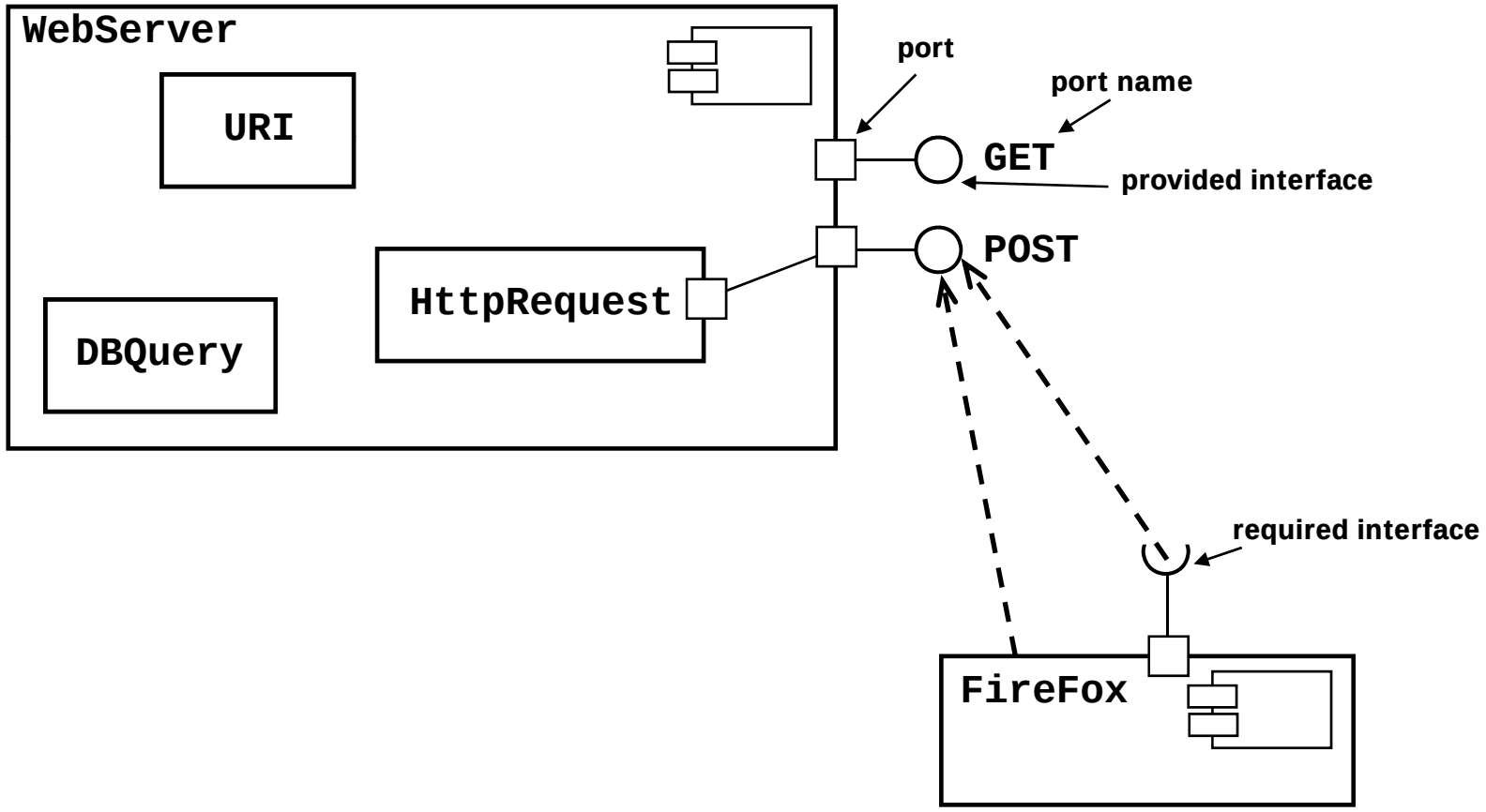
Component Diagram

- Component Diagram (UML 2.0)
  - A graph of components connected by dependency relationships.
  - Shows the dependencies among software components
- Dependencies are shown as dashed arrows from the client component to the supplier component.



Component Diagram

- Components can be refined to include information about the interfaces they provide (through *ports*) and the parts (classes, subsystems) they contain:

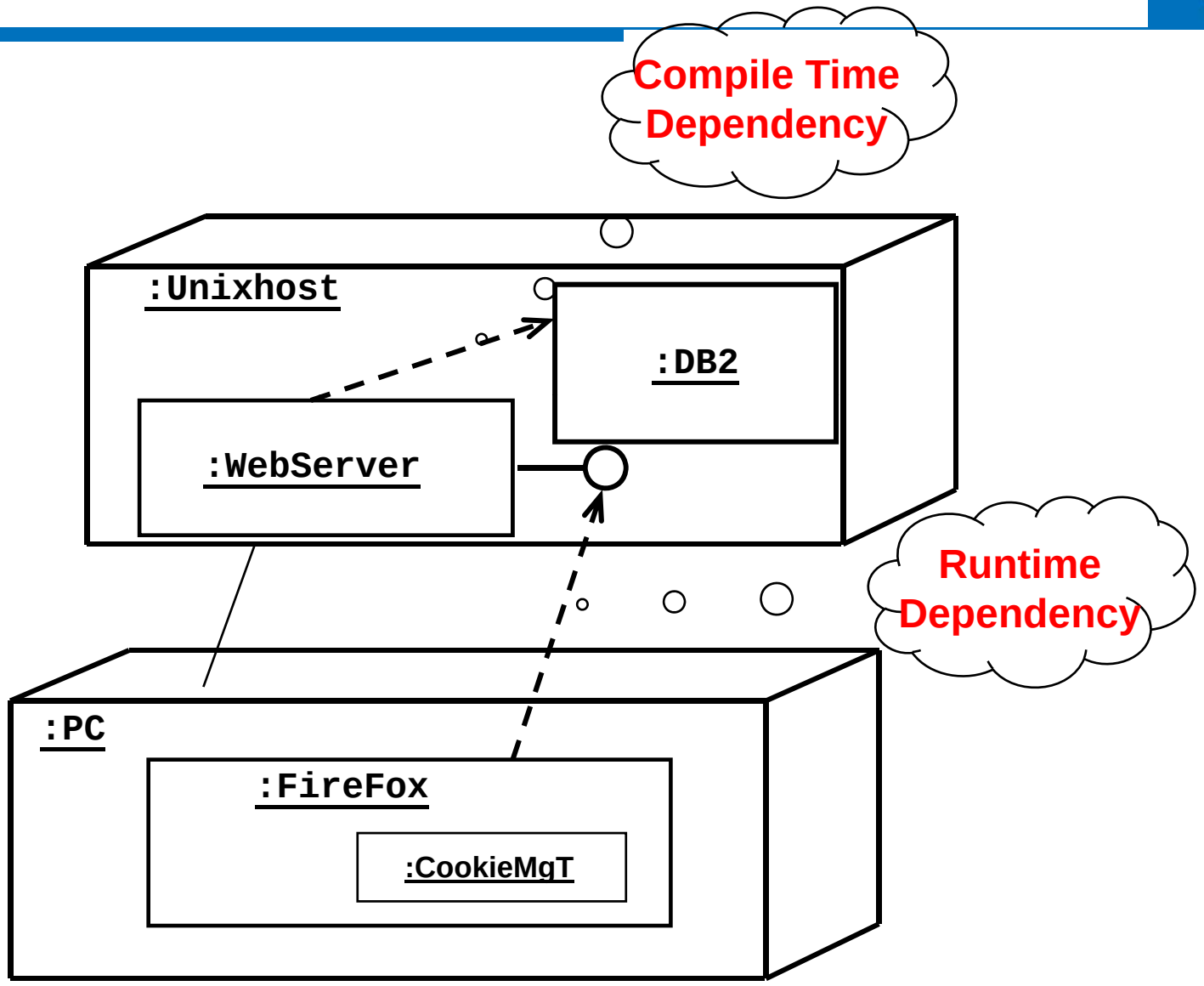


Deployment Diagram

- Deployment diagrams are useful for showing a system design after the following decisions are made
  - Subsystem decomposition
  - Hardware Mapping
  - Selection of additional Software Components

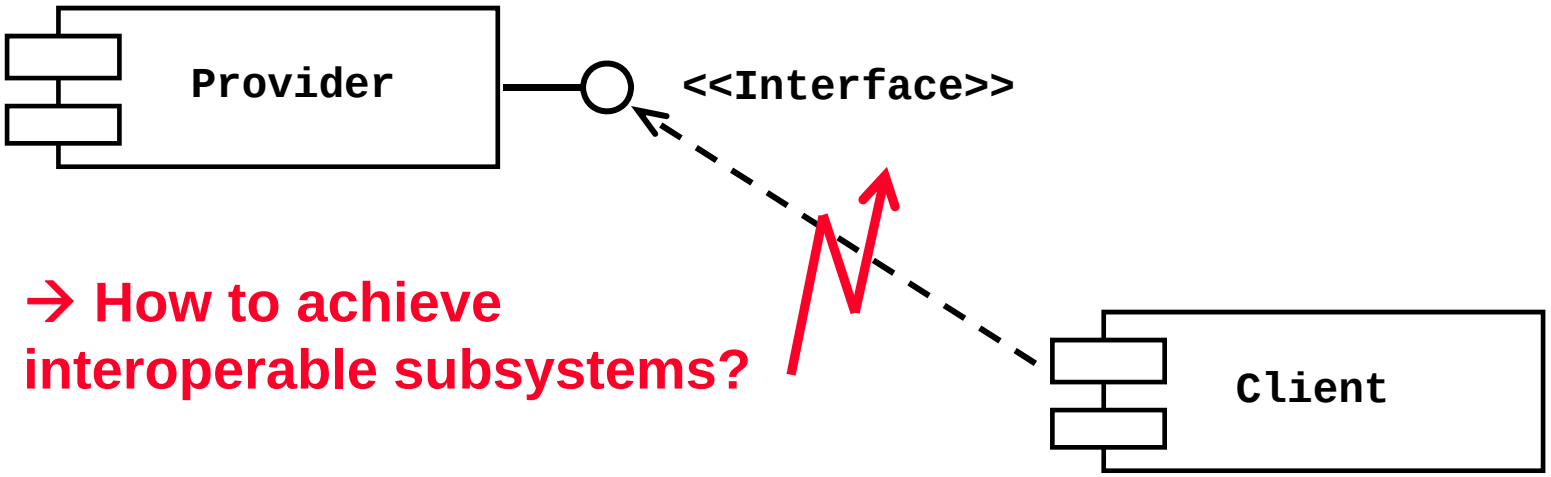
- A deployment diagram is a graph of nodes connected by communication associations.
  - Nodes are shown as 3-D boxes.
  - Nodes may contain component instances.
  - Components may contain objects (indicating that the object is part of the component)

Deployment Diagram Example



Software Mapping:
Problems with Connecting Subsystems

- Connecting subsystems yields additional problems:
  - Subsystems are implemented in different programming languages
  - The intention (semantics) cannot be derived exactly from the interface (Example: difference between POST and GET?!)
  - Subsystems might be built on different abstraction levels / communication layers



→ How to achieve interoperable subsystems?

Software Mapping:
Demand: Interoperability

- Interoperability: The ability of two or more systems or components to exchange information and to use the information that has been exchanged

  IEEE Glossary, 1990

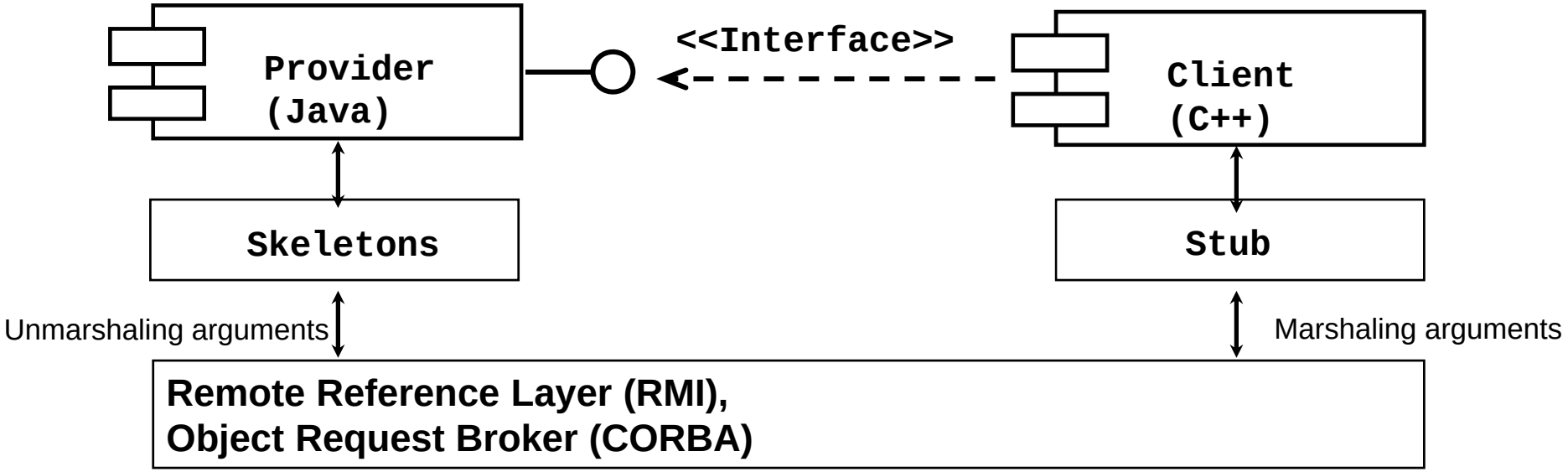
- Interoperability denote systems that work together

  Microsoft Webpage

- In general: Use standards!
- (Selected) Requirements for interoperable Systems:
  - Standards for Communication for distributed systems
  - Semantic descriptions for interfaces
  - Mediator components

Software Mapping:
Standards for distributed systems

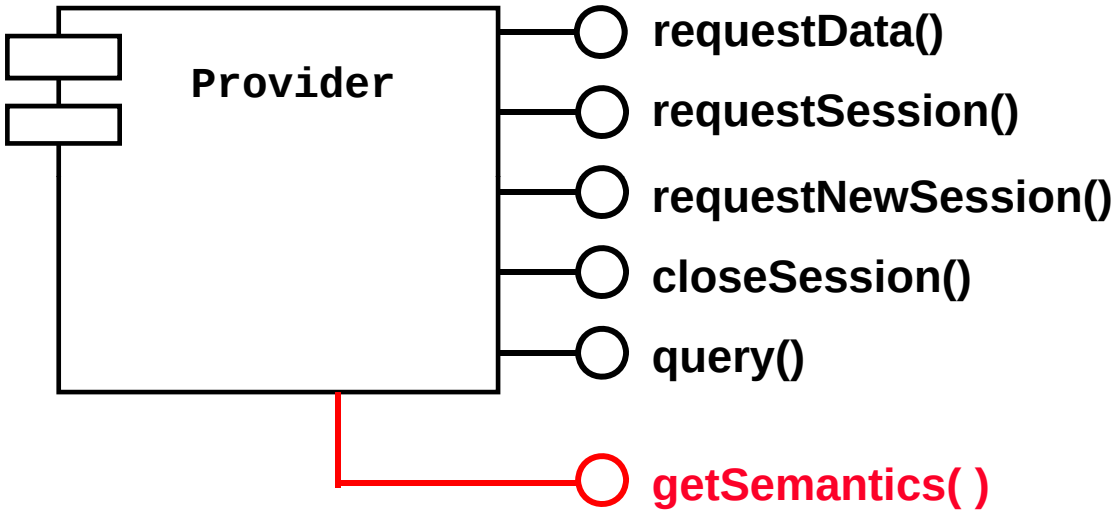
- Standards for integrating *distributed* Subsystems:
  - Middleware Frameworks (CORBA, RMI, COM, SOAP)



  - Interface Definition Languages (IDL) are used to formulate Interfaces of Subsystems in an language neutral way
  - Compiler transforms IDL description to concrete components (Java, C++) and generates Stubs and Skeletons

Software Mapping:
Semantic Descriptions

- Semantic Descriptions (Annotations) are instrumental to clarify the intention of a subsystem



Questions:

→ Order?

→ Any Iterations()?

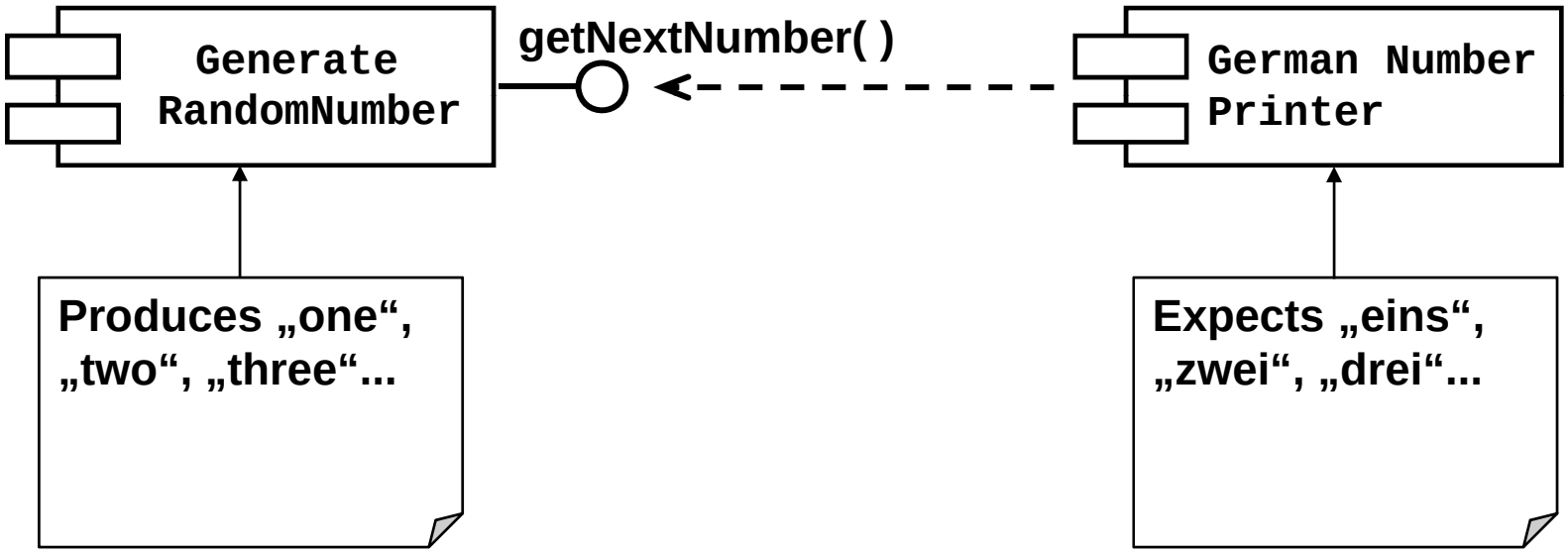
→(Exact) Purpose?

→Dependencies?

- Hardly implemented in standard technologies, only in academic fields or in terms of proprietary solutions
- New Approach: **Ontology**: Notation for describing the semantic concepts of an interface
  - To-Date: Promising, but still hardly implemented

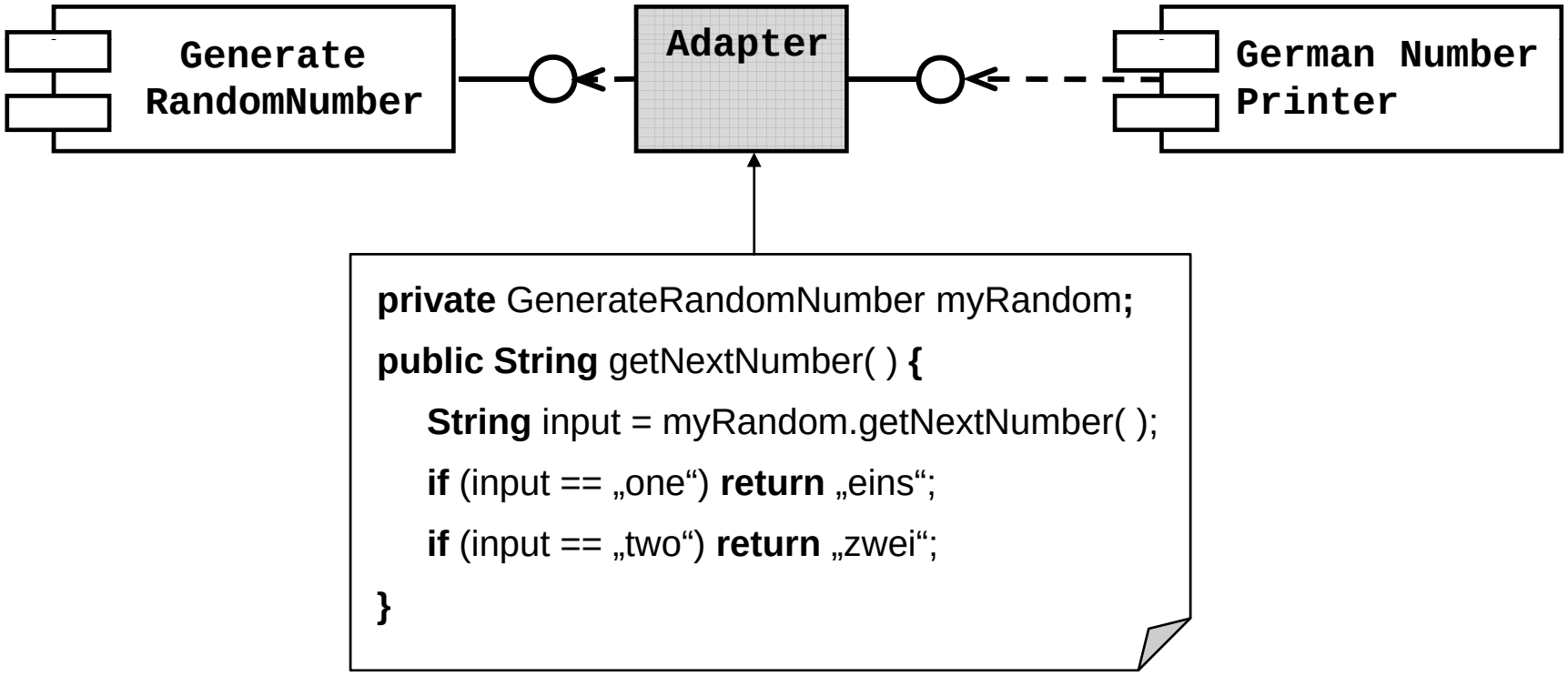
Software Mapping:
Adapters

- Problem: Incompatible Interfaces and/or different data formats (syntax):



Software Mapping:
Adapters

- Problem: Incompatible Interfaces and/or different data formats (syntax):
- Solution: Adapter (implements both interfaces and transforms data in an format, understandable for the client)



4. Data Management

- Some objects in the models need to be persistent
  - In particular Entity Objects
  - Boundary Objects as well (e.g. user preferences in forms)
- A persistent object can be realized with one of the following
  - Files
    - Cheap, simple, permanent storage
    - Low level (Read, Write)
    - Applications must add code to provide suitable level of abstraction
  - Database
    - Powerful, scalable, portable
    - Supports multiple writers and readers

File or Database?

- When should you choose a file system?
  - For extensive data (images)
  - For lots of raw data (core dump, event trace)
  - Temporary Data that is kept only for a short time

- When should you choose a database?
  - Data that require access at fine levels of details by multiple users and/or applications (concurrent access)
  - Data that must be ported across multiple platforms (heterogeneous systems)
  - Rollback and Transactions play an important aspect

Database Management System

- A collection of routines that enables you to store, modify, and extract information from a database
- Main functions (realization depends on Vendor)
  - Concurrent (write) access to the stored data
  - Transaction Management
  - Rollback Mechanisms
  - Crash Recovery
  - Optimizer for complex Queries
  - Integrity Checks
  - Authorization

Relational Databases

- Data are stored in tables that comply with a predefined type called a (relational) schema
  - Column represents an attribute of a schema
  - Row represents data item as a tuple of attribute values
  - Several tuples in different tables represent the attribute values of an object
- Mapping rules to convert object-oriented model to a relation scheme
- Ideal for querying large data sets with complex queries
- SQL is the standard language defining and manipulating tables
- Leading commercial databases:
  - DB2 (IBM)
  - Oracle
  - MySQL

Object-Oriented Databases

- Support of all fundamental object modeling concepts
  - Stores data as objects and associations
  - Classes, Attributes, Methods, Associations, Inheritance
- Reduce the time for the initial development of storage subsystem:
  - Easy Mapping of Object model to Database schema
- Disadvantage:
  - Slower than relational databases even for simple queries
- Commercial Vendor
  - ObjectStore
  - Realization as “add-ons” for Relational DB (DB2) → Sense?

- New Hypes
  - XML Databases (mostly add-ons, many open source projects)

5. Defining Access Control

- In multi-user systems different actors have access to different functionality and data.
  - Example:
    - System Admin has unlimited access to any data
    - Client only has read access to data
- Definition of access right during analysis:
  - Associate different use cases with different actors.
- During system:
  - Determining which objects are shared among actors.
  - Define access rights for each objects (→ access matrix)
- Subtopics here:
  - Authentication
  - Encryption

Access Matrix

- We model access on classes with an access matrix.
  - The rows of the matrix represents the actors of the system
  - The column represent classes whose access we want to control.

- **Access Right:** An entry in the access matrix. It lists the operations that can be executed on instances of the class by the actor.

Access Matrix: Example

Objects / Actors	PressRelease	Cashflow	
External Organizer	read( )		
Manager	read( ) write( ) submit( ) Reject( )	read( ) compute( ) annoyAbout( )	
Employee	read( ) adjust( )	read( )	

Authentication and Encryption

- Authentication
  - Process of verifying the association between the identity of a user (or calling subsystem) and the system
    - User name / password
    - Smart Cards
    - Biometric Sensors (analyzing patterns of blood vessels in eyes / fingers
- Encryption
  - Translating a message (plaintext) into an encrypted message (ciphertext).
  - A key is used to encrypt (sender) and decrypt (receiver) a message

- Both approaches are fundamentally difficult problems
  - → Select Off-the-Shelf packages !

Different Options for Access Control

Static:

- Global Access Tables (as shown)
- Access Control Lists (who may does what on a given object)
- Capabilities (given an actor: what can he do to what objects?)
- Rules
  - more simple to write and read

Dynamic

- Rules
- Proxies

6. Design Global Control Flow

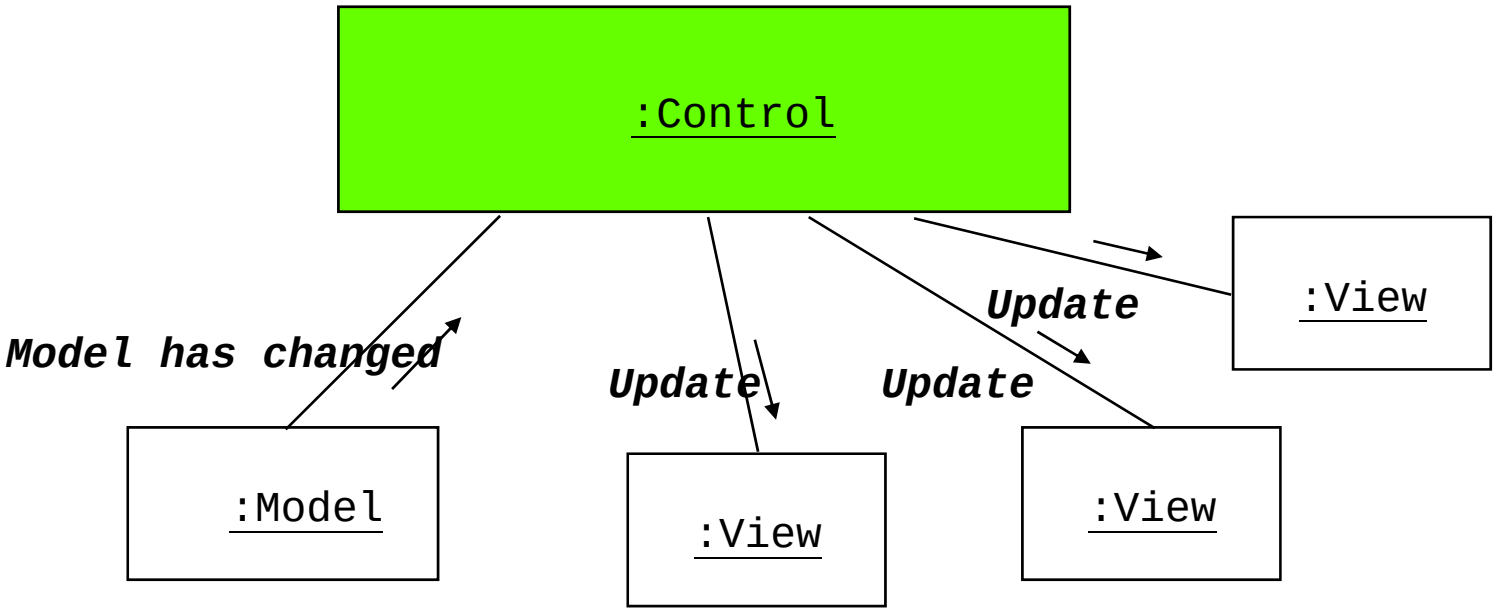
- Control Flow is the sequencing of actions in a system
  - Deciding which operations will be executed in which order
  - Some order is represented explicitly in the code (e.g. order of statements)
  - Some order is given implicitly by (object-oriented) language semantics (e.g. overloading, type of object decides which method version is executed)
- Control flow is modeled by control objects.
  - Record external events
  - Store temporary states about events
  - Issue the right sequence of method calls on both boundary and entity objects

Design Global Control Flow

- Three main control models:
  - Procedure-driven control
    - explicit
    - easy to understand (most often … ;-))
    - hard to change
  - Event-driven control
    - observer pattern (aka publisher subscriber)
    - callback methods in frameworks
    - somewhat implicit (you have to know where to look)
    - more flexible, reusable
  - Thread-based control
    - parallel execution (often only pseudo parallel)
    - implicit
    - sometimes difficult to understand, test, maintain
    - enable flexible UIs and non-blocking behavior in distributed systems

Decide on Software Control

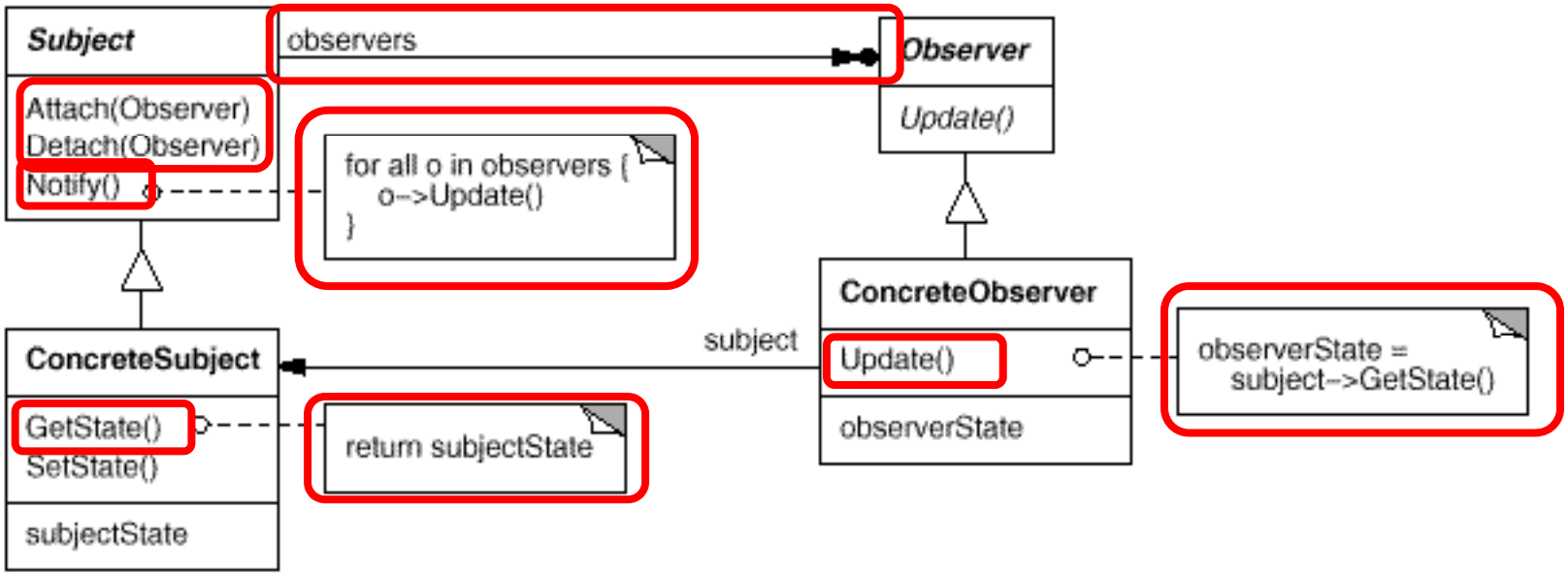
- Event-driven control
  - Control object runs a loop managing external events
    - Example (Data) Model has changed
  - Whenever an event becomes available, it is dispatched to appropriate receiver components
  - Very flexible, good for the design of graphical user interfaces, easy to extend



Event-Driven Control Flow
Observer Pattern

Two phases

1. Configuration (who observes whom)
2. Notification about changes



Decide on Software Control

- Thread-based (or decentralized) control
  - Control resides in several independent objects:
    - Each external event is assigned a single Thread
    - Each new Thread handles an event
  - Possible speedup by mapping the objects on different processors

7. Boundary Conditions (1/2)

- Most of the system design effort is concerned with steady-state behavior.
- However, the system design phase must also address the initiation and finalization of the system. This is addressed by a set of new use cases called *administration use cases*
  - Initialization describes how the system is brought from an non initialized state to steady-state ("startup use cases").
    - How does the system start up?
    - What data need to be accessed at startup time?
    - What services have to registered?
    - What does the user interface do at start up time?
    - How does it present itself to the user?

7. Boundary Conditions (2/2)

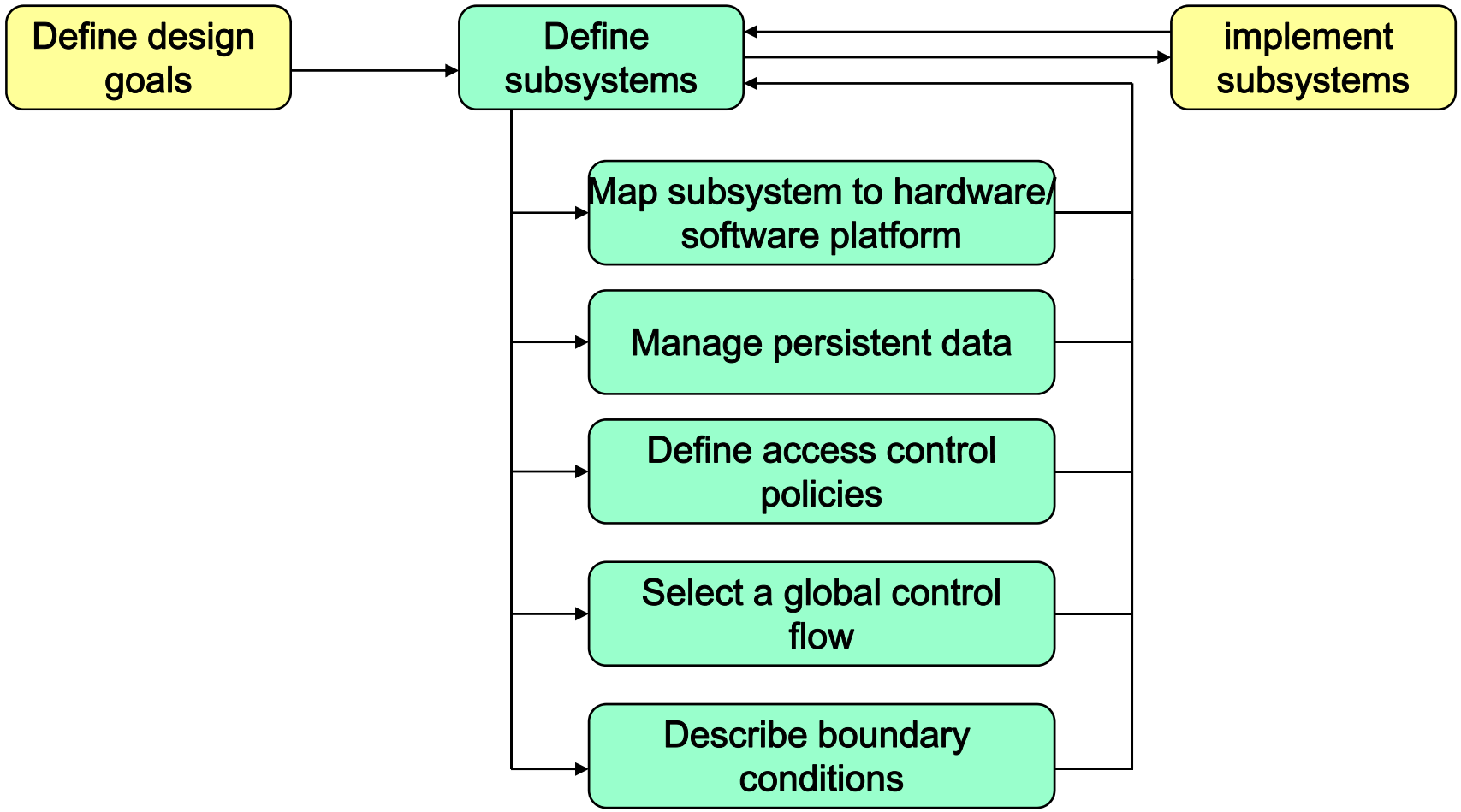
- Termination
  - Describes what resources are cleaned up and which systems are notified upon termination ("termination use cases").
    - Are single subsystems allowed to terminate?
    - Are other subsystems notified if a single subsystem terminates?
    - How are local updates communicated to the database?

- Failure
  - Many possible causes: Bugs, errors, external problems (power supply).
  - Good system design foresees fatal failures (“failure use cases”).
    - How does the system behave when a node or communication link fails? Are there backup communication links?
    - How does the system recover from failure? Is this different from initialization?

8. Anticipating Change

- … in the next lecture …

Architecture Organization
Iterative Process



Summary

- Hardware/Software mapping
- Persistent data management
- Global resource handling
- Software control selection
- Boundary conditions

- Each of these activities revises the subsystem decomposition to address a specific issue. Once these activities are completed, the interface of the subsystems can be defined.