

Object-Oriented Software Engineering
Using UML, Patterns, and Java



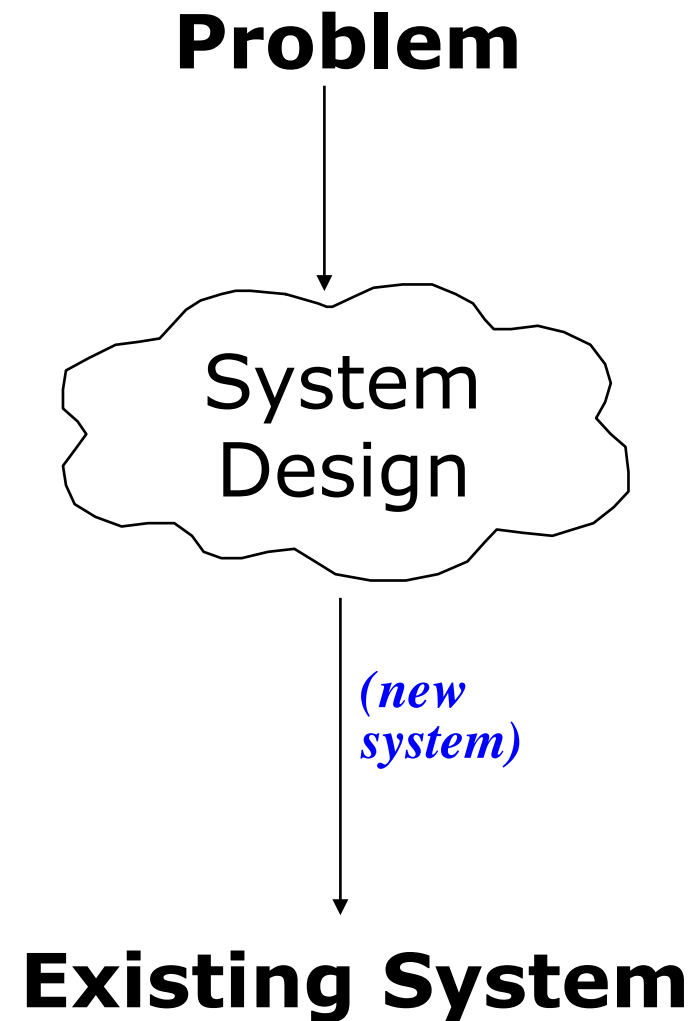
**System Design I:
System Decomposition**

Why is Design so Difficult?

- **Analysis:** Focuses on the application domain
- **Design:** Focuses on the solution domain
 - The solution domain is changing very rapidly
 - Halftime knowledge in software engineering: About 3-5 years
 - Cost of hardware rapidly sinking
 - Design knowledge is a moving target
- **Design window:** Time in which design decisions have to be made.

The Scope of System Design

- Bridge the gap
 - between a problem and an existing system in a manageable way
- How?
- Use Divide & Conquer:
 - 1) Identify design goals
 - 2) Model the new system design as a set of subsystems
 - 3-8) Address the major design goals.



System Design: Eight Issues

System Design



1. Identify Design Goals

Additional NFRs
Trade-offs

2. Subsystem Decomposition

Layers vs Partitions
Architectural Style
Coherence & Coupling

3. Identify Concurrency

Identification of
Parallelism
(Processes,
Threads)

4. Hardware/ Software Mapping

Identification of Nodes
Special Purpose Systems
Buy vs Build
Network Connectivity

5. Persistent Data Management

Storing Persistent
Objects
Filesystem vs Database

8. Boundary Conditions

Initialization
Termination
Failure.

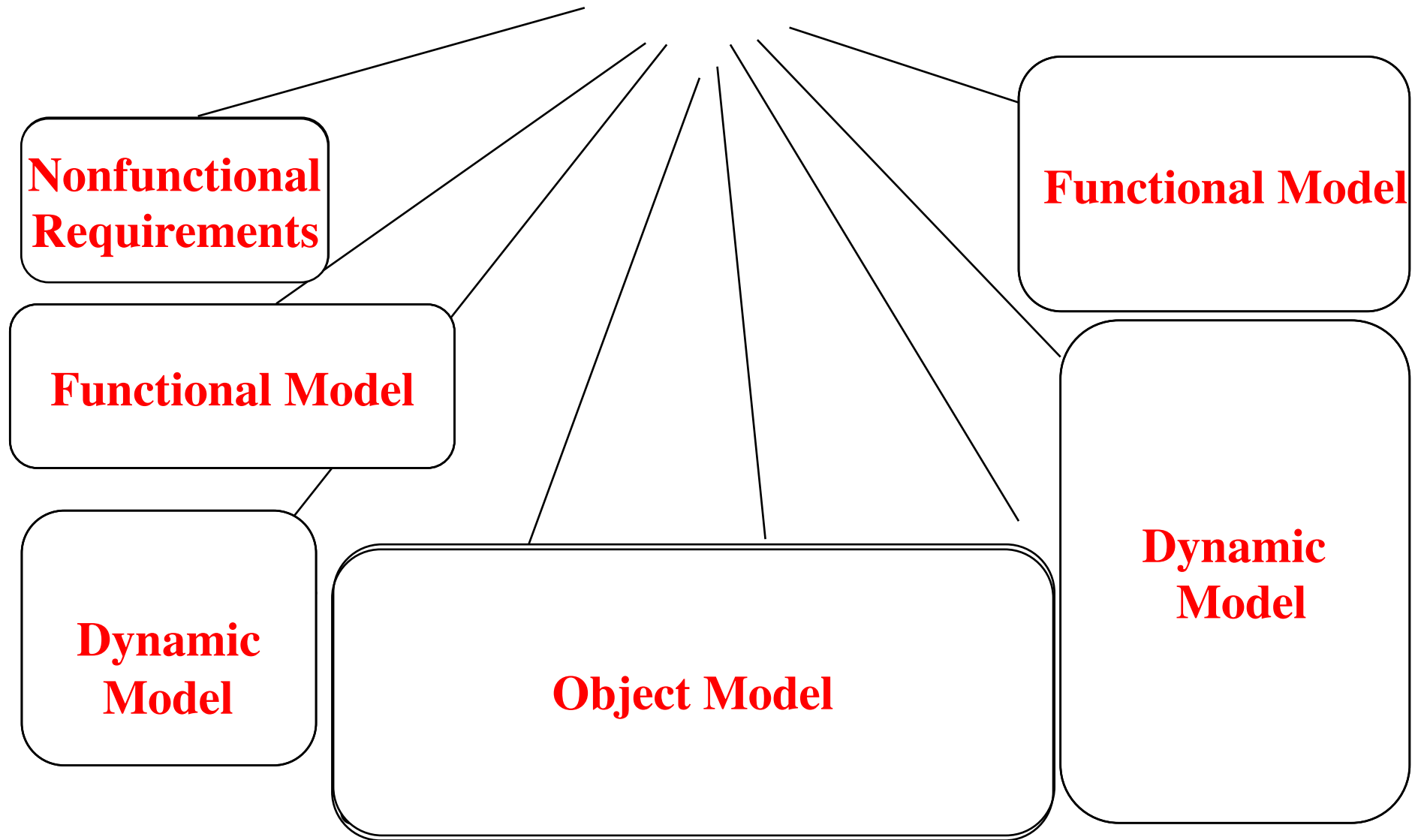
7. Software Control

Monolithic
Event-Driven
Conc. Processes

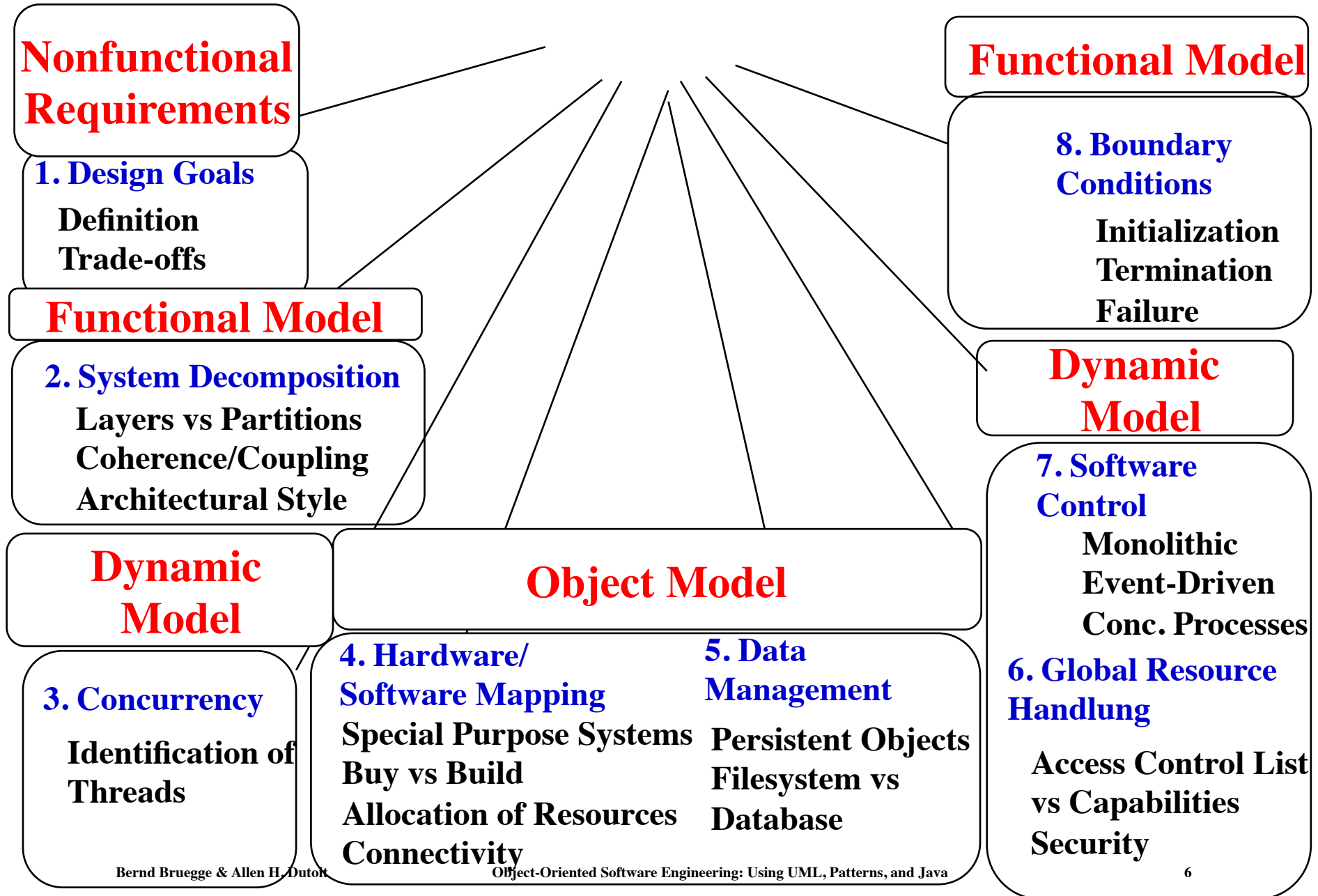
6. Global Resource Handling

Access Control
ACL vs Capabilities
Security

Analysis Sources: Requirements and System Model



From Analysis to System Design



System Design Activities

1) Design Goals

2) System Decomposition

3) Concurrency

4) Hardware/Software Mapping

5) Data Management

6) Global Resource Handling

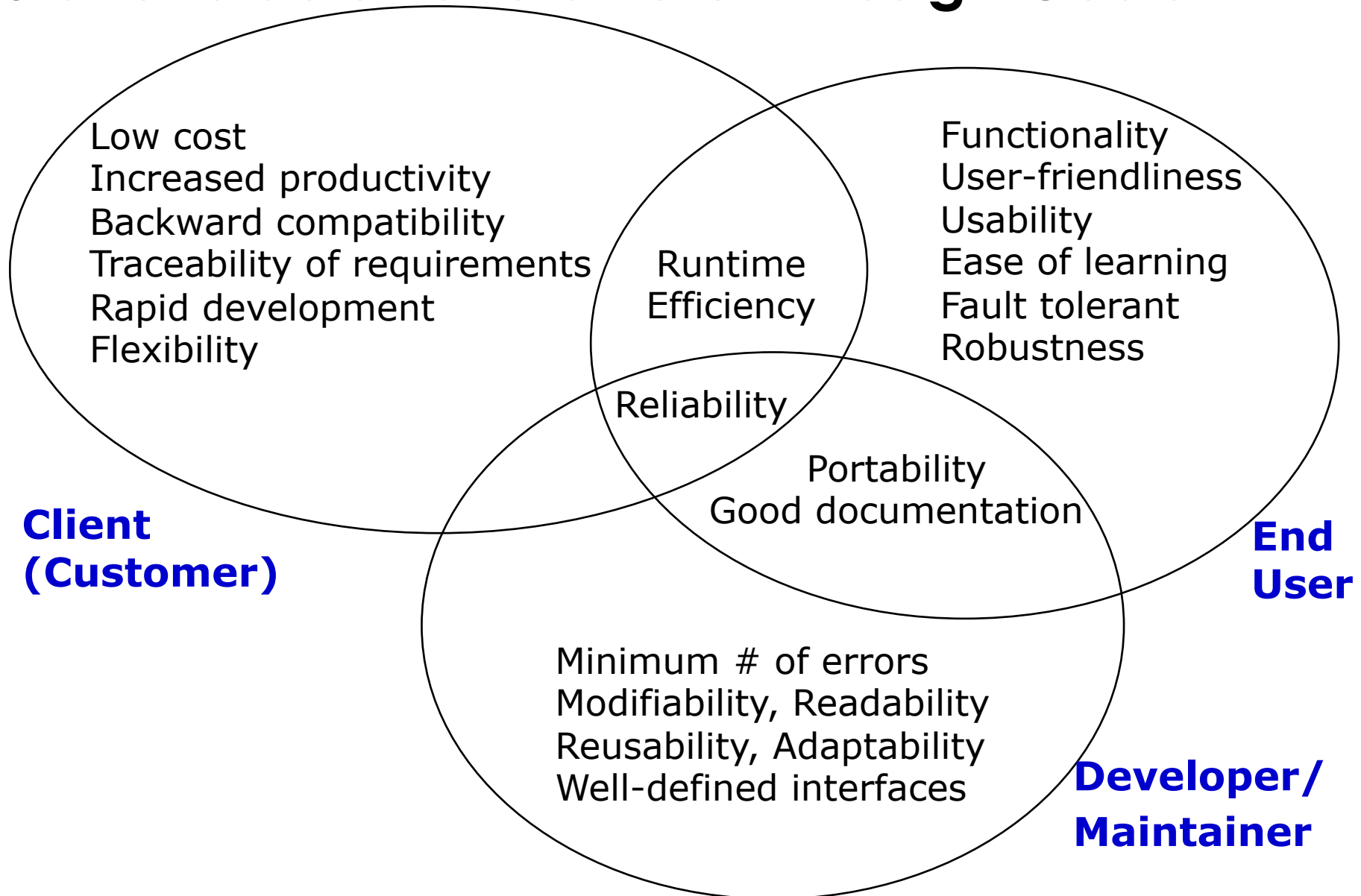
7) Software Control

8) Boundary Conditions

Example of Design Goals

- Reliability
- Modifiability
- Maintainability
- Understandability
- Adaptability
- Reusability
- Efficiency
- Portability
- Traceability of requirements
- Fault tolerance
- Backward-compatibility
- Cost-effectiveness
- Robustness
- High-performance
- ❖ Good documentation
- ❖ Well-defined interfaces
- ❖ User-friendliness
- ❖ Reuse of components
- ❖ Rapid development
- ❖ Minimum number of errors
- ❖ Readability
- ❖ Ease of learning
- ❖ Ease of remembering
- ❖ Ease of use
- ❖ Increased productivity
- ❖ Low-cost
- ❖ Flexibility

Stakeholders have different Design Goals



Typical Design Trade-offs

- Functionality v. Usability
- Cost v. Robustness
- Efficiency v. Portability
- Rapid development v. Functionality
- Cost v. Reusability
- Backward Compatibility v. Readability

System Design Phases

1) Design Goals

2) System Decomposition

3) Concurrency

4) Hardware/Software Mapping

5) Data Management

6) Global Resource Handling

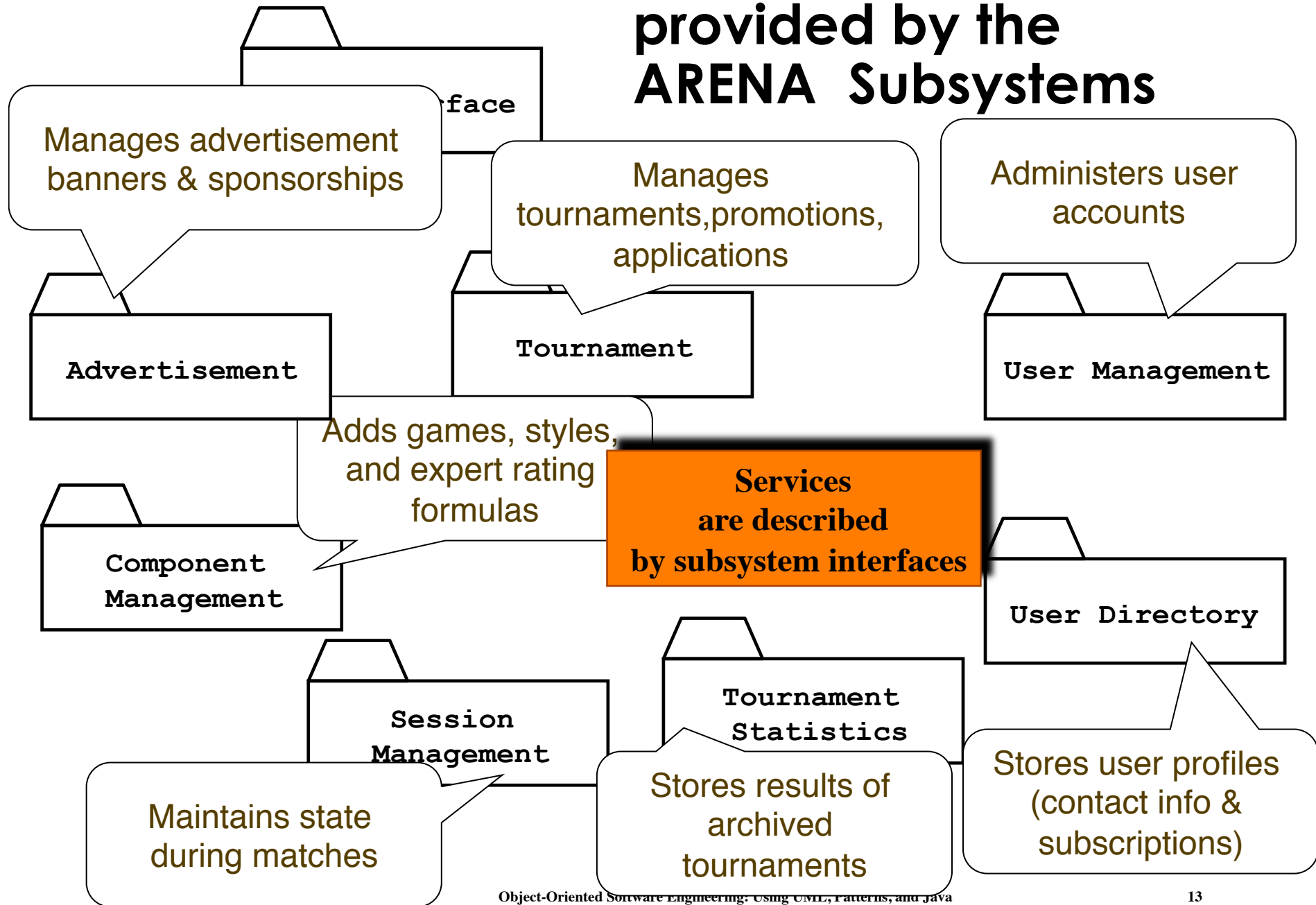
7) Software Control

8) Boundary Conditions

Subsystems and Services

- **Subsystem**
 - Collection of classes, associations, operations, events that are closely interrelated with each other
 - The classes in the object model are the “seeds” for subsystems
- **Service**
 - A group of externally visible operations provided by a subsystem (also called **subsystem interface**)
 - The use cases in the functional model provide the “seeds” for services

Example: Services provided by the ARENA Subsystems



Subsystem Interface and API

- **Subsystem interface:** Set of fully typed UML operations
 - Specifies the interaction and information flow from and to subsystem boundaries, but not inside the subsystem
 - Refinement of service, should be well-defined and small
 - *Subsystem interfaces are defined during object design*
- **Application programmer's interface (API)**
 - The API is the specification of the subsystem interface in a specific programming language
 - **APIs are defined during implementation**
- The terms subsystem interface and API are often confused with each other
 - *The term API should not be used during system design and object design, but only during implementation.*

Subsystems relationships

Coherence and Coupling

Coupling and Coherence of Subsystems

- Goal: Reduce system complexity while allowing change
- **Coherence** measures dependency among classes
 - **High coherence**: The classes in the subsystem perform similar tasks and are related to each other via many associations
 - **Low coherence**: Lots of miscellaneous and auxiliary classes, almost no associations
- **Coupling** measures dependency among subsystems
 - **High coupling**: Changes to one subsystem will have high impact on the other subsystem
 - **Low coupling**: A change in one subsystem does not affect any other subsystem.

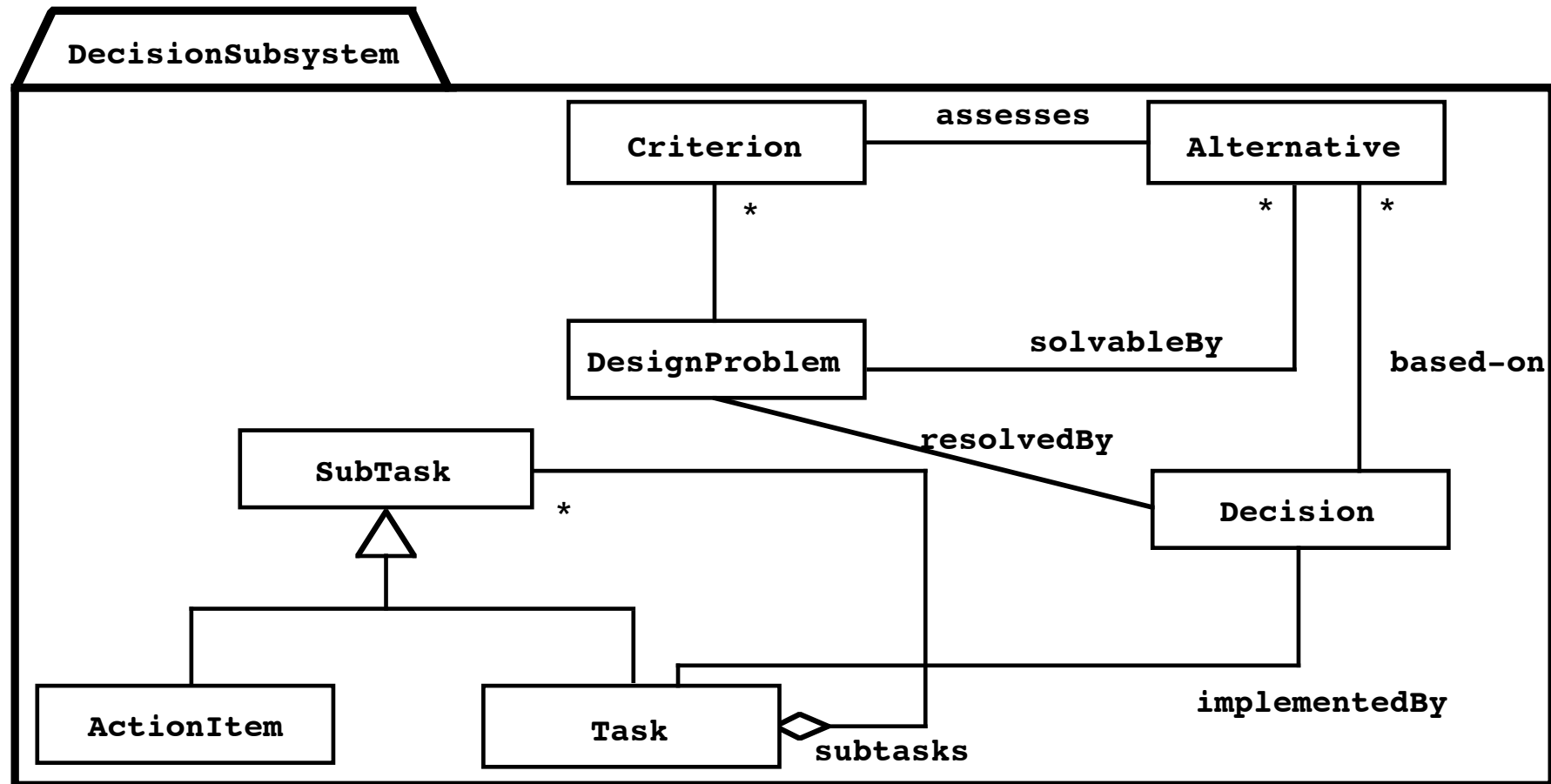
Coupling and Coherence of Subsystems

Good System Design

- Goal: Reduce system complexity while allowing change
- **Coherence** measures dependency among classes
 - ➔ **High coherence:** The classes in the subsystem perform similar tasks and are related to each other via many associations
 - **Low coherence:** Lots of miscellaneous and auxiliary classes, almost no associations
- **Coupling** measures dependency among subsystems
 - **High coupling:** Changes to one subsystem will have high impact on the other subsystem
 - ➔ **Low coupling:** A change in one subsystem does not affect any other subsystem

An example: the Decision tracking system

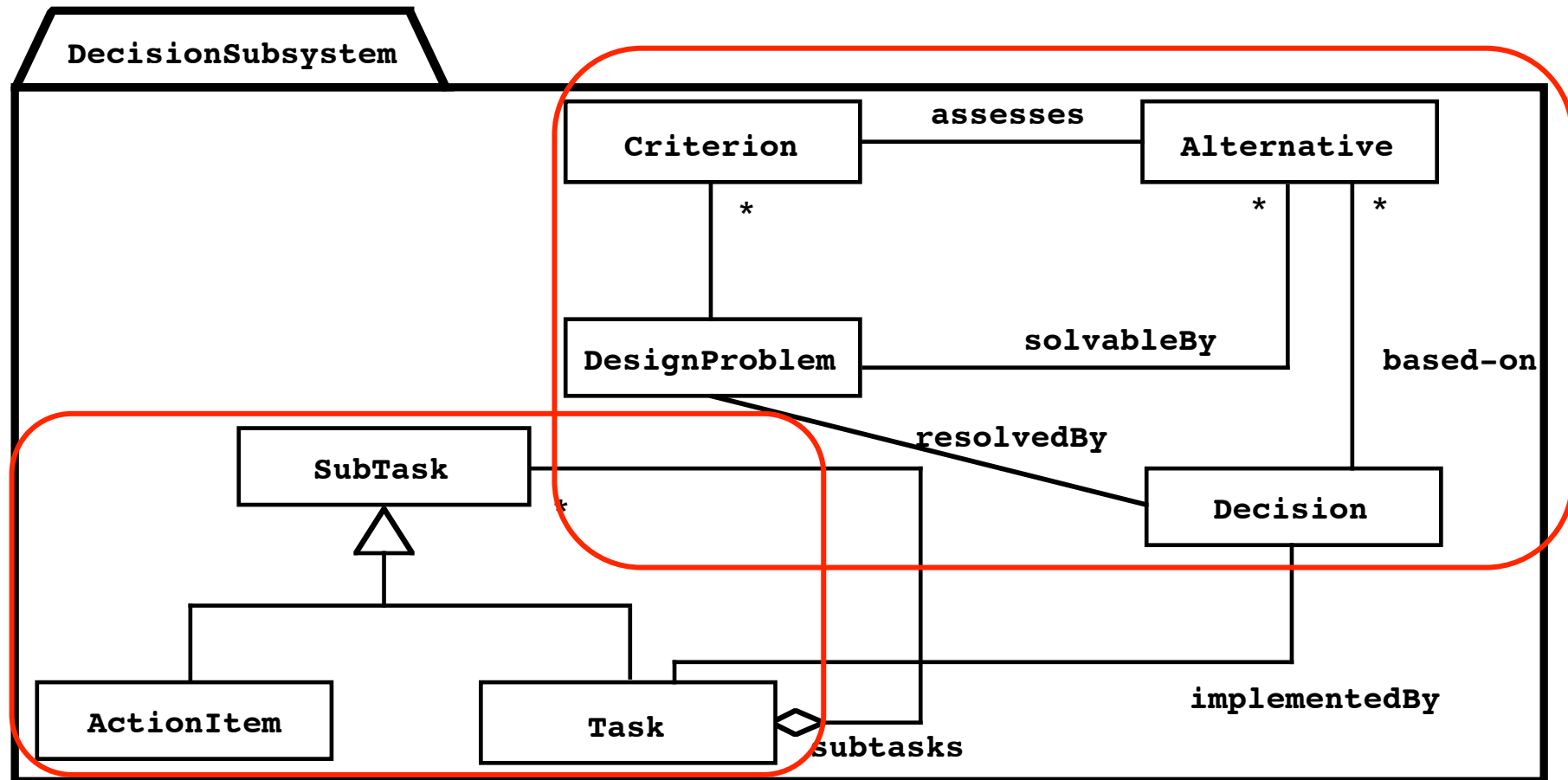
The decision tracking system purpose is to record design problems, discussions, alternative evaluations, decisions, and their implementations in terms of tasks



The DecisionSubsystem has a low coherence: The classes Criterion, Alternative, and DesignProblem have no relationships with Subtask, ActionItem, and Task.

An example: the Decision tracking system

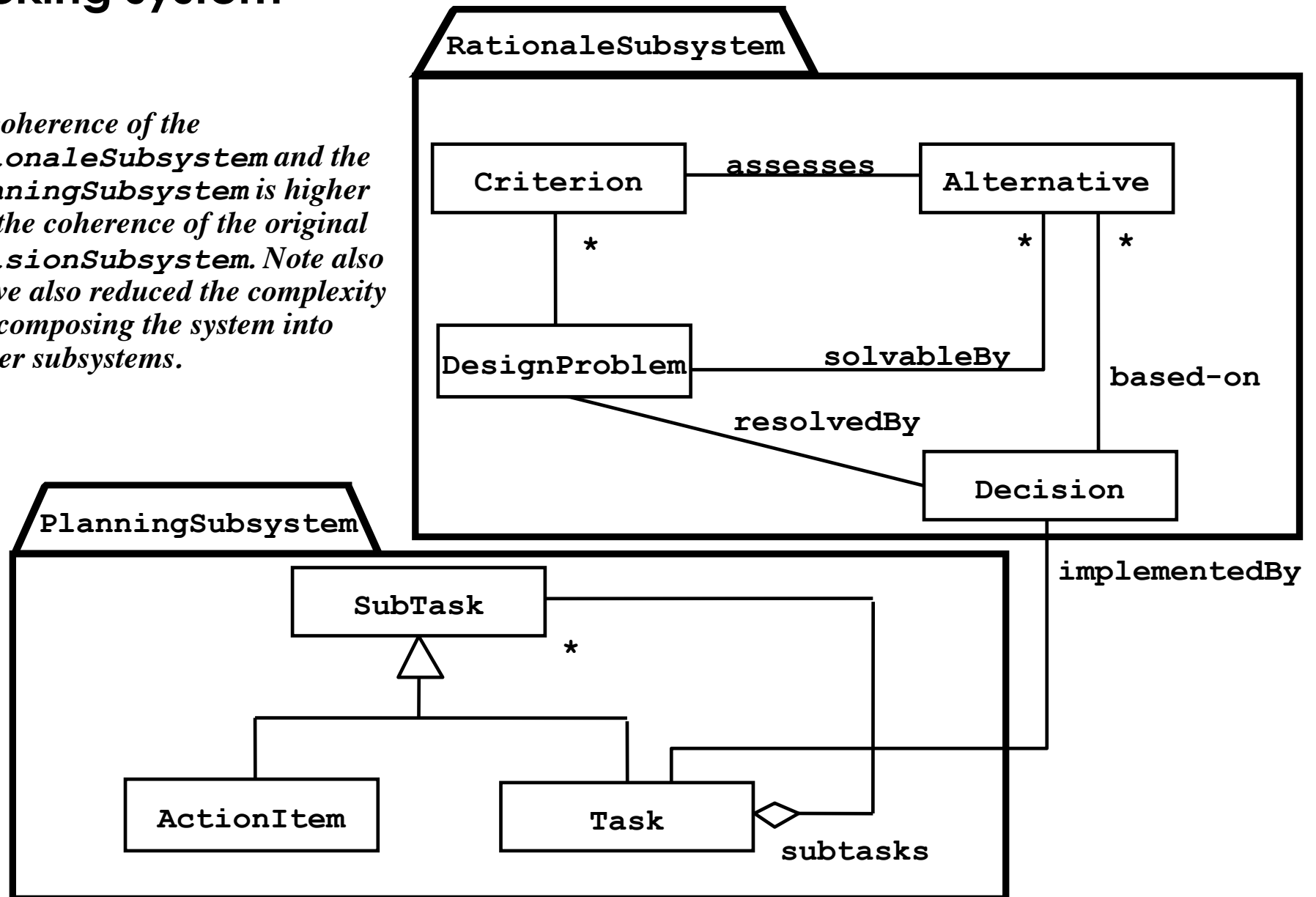
The decision tracking system purpose is to record design problems, discussions, alternative evaluations, decisions, and their implementations in terms of tasks



The DecisionSubsystem has a low coherence: The classes Criterion, Alternative, and DesignProblem have no relationships with Subtask, ActionItem, and Task.

Alternative subsystem decomposition for the decision tracking system

The coherence of the RationaleSubsystem and the PlanningSubsystem is higher than the coherence of the original DecisionSubsystem. Note also that we also reduced the complexity by decomposing the system into smaller subsystems.



How to achieve high Coherence

- **High coherence** can be achieved if most of the interaction is within subsystems, rather than across subsystem boundaries
- Questions to ask:
 - Does one subsystem always call another one for a specific service?
 - Yes: Consider moving them together into the same subsystem.
 - Which of the subsystems call each other for services?
 - Can this be avoided by restructuring the subsystems or changing the subsystem interface?
 - Can the subsystems even be hierarchically ordered (in layers)?

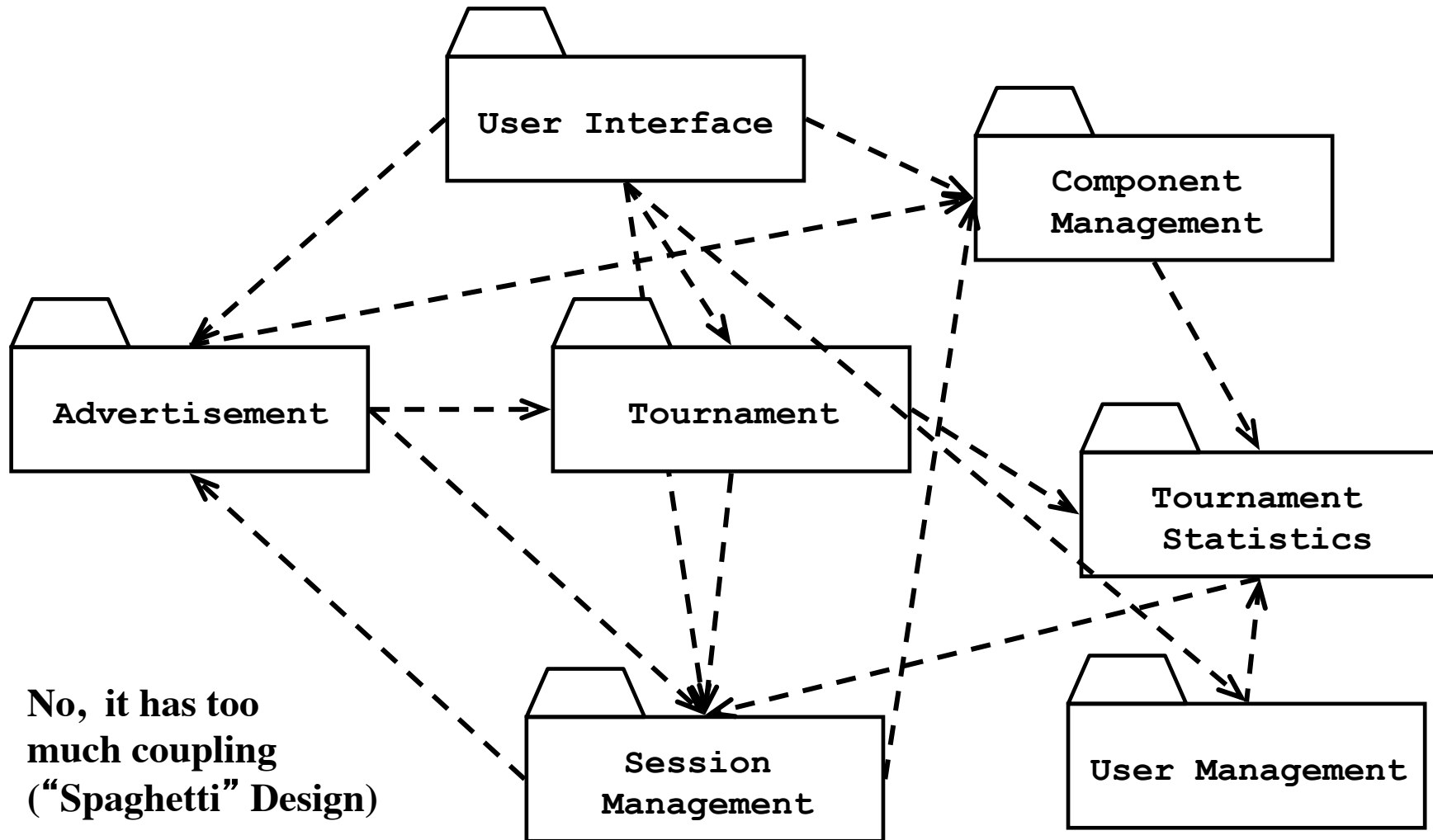
How to achieve Low Coupling

- **Low coupling** can be achieved if a calling class does not need to know anything about the internals of the called class (**Principle of information hiding**, Parnas)

David Parnas, *1941,
Developed the concept of
modularity in design.



Is this a Good Design?



No, it has too much coupling ("Spaghetti" Design)

Dijkstra's answer to "Spaghetti Design"

- Dijkstra revolutionary idea in 1968
 - Any system should be designed and built as a hierarchy of layers: Each layer uses only the services offered by the lower layers



Edsger W. Dijkstra, 1930-2002
Formal verification: Proofs for programs
Dijkstra Algorithm, Banker's Algorithm,
Gotos considered harmful, T.H.E.,
1972 Turing Award

Architectural Style vs Architecture (Terms Definition)

- **Subsystem decomposition:** Identification of subsystems, services, and their relationship to each other
- **Architectural Style:** A pattern for a subsystem decomposition
- **Software Architecture:** Instance of an architectural style.

Examples of Architectural Styles

- ➔ Layered Architectural style
 - Service-Oriented Architecture (SOA)
 - Client/Server
 - Peer-To-Peer
 - Three-tier, Four-tier Architecture
 - Repository
 - Model-View-Controller
 - Pipes and Filters

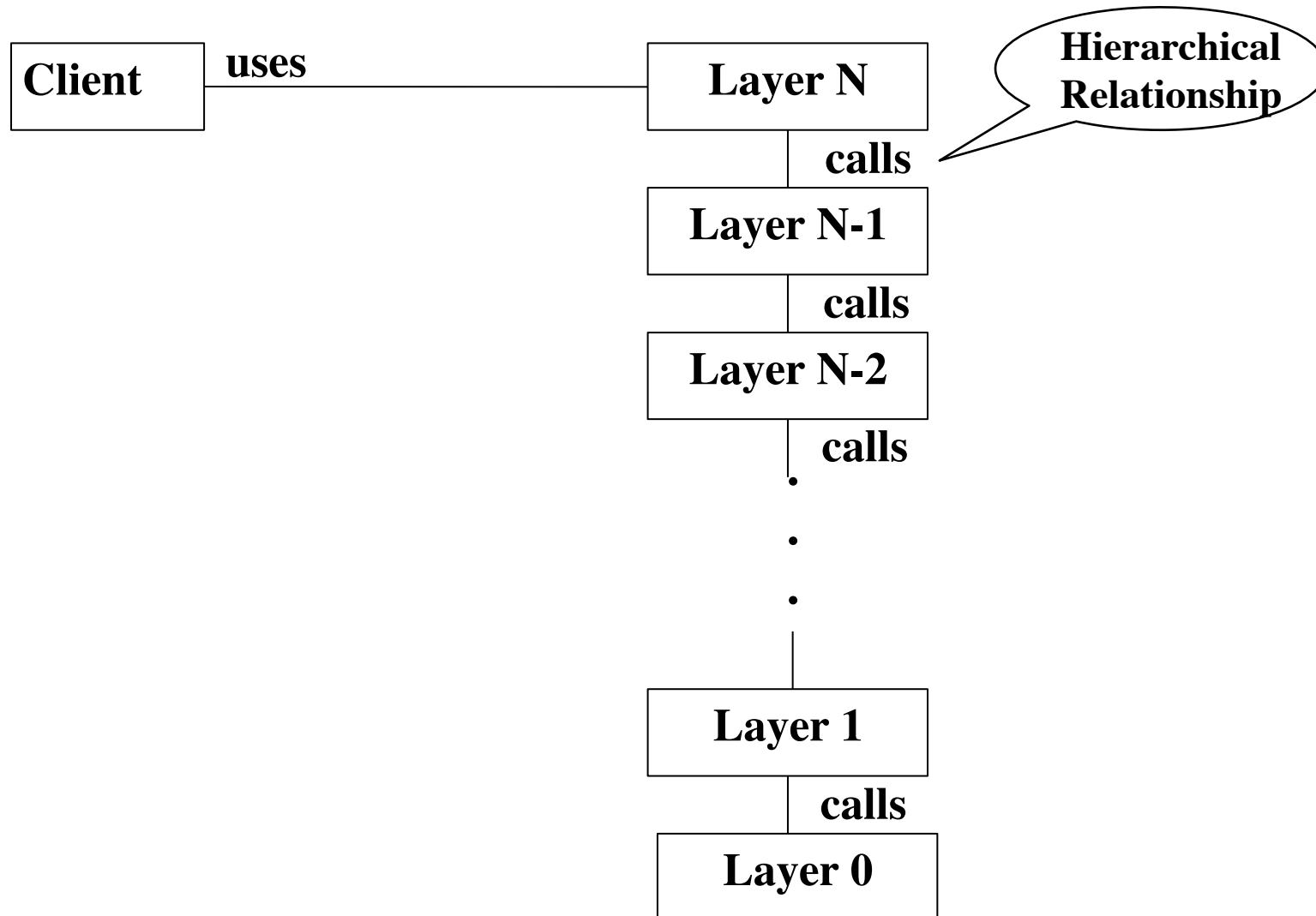
Partitions and Layers

Partitioning and layering are techniques to achieve low coupling.

A large system is usually decomposed into subsystems using both, layers and partitions.

- **Partitions** vertically divide a system into several **independent (or weakly-coupled) subsystems** that provide services on the same level of abstraction.
- A **layer** is a subsystem that provides services to a **higher level of abstraction**
 - A layer can only depend on lower layers
 - A layer has no knowledge of higher layers

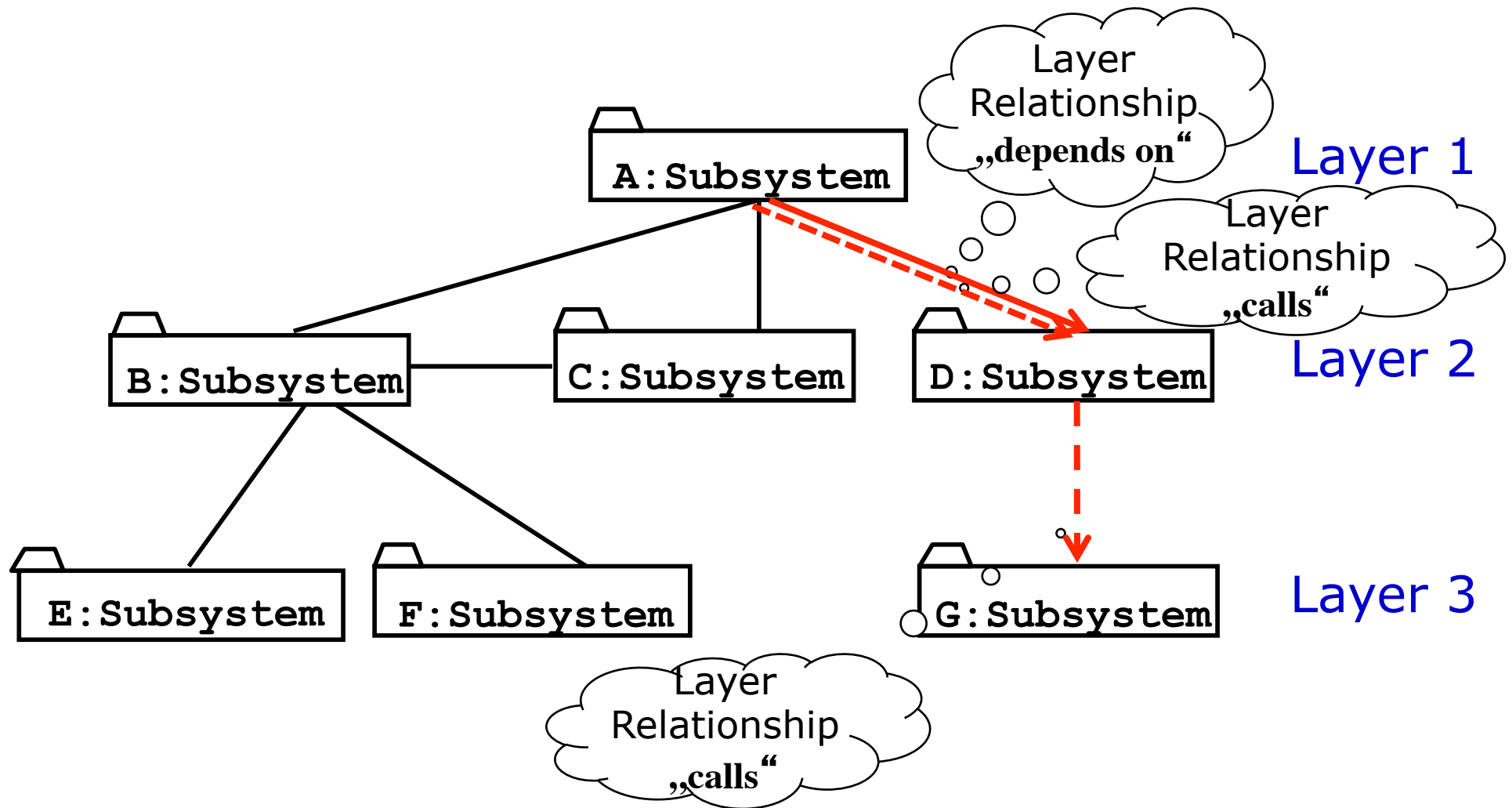
The Layered Architectural Style



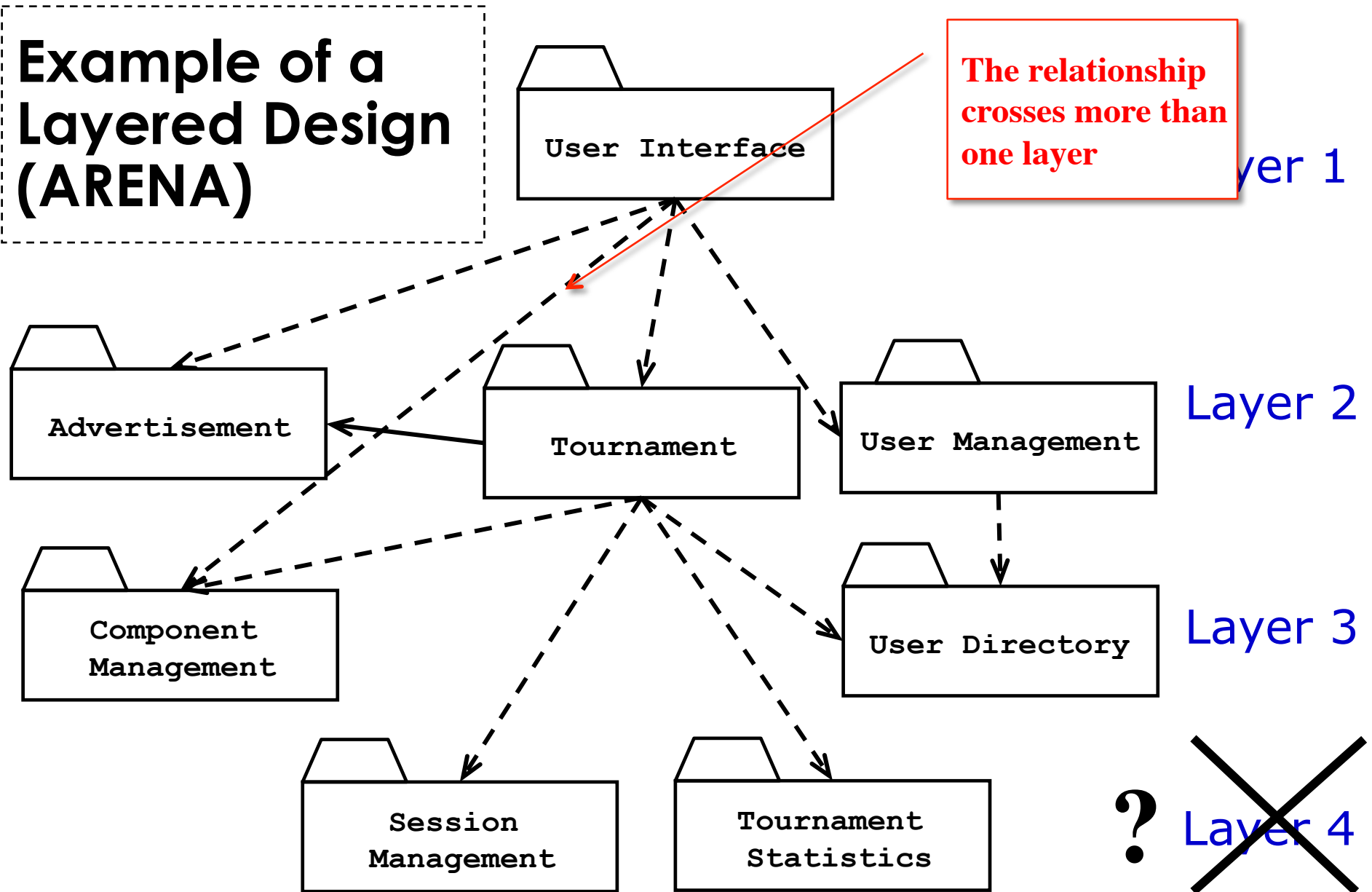
Hierarchical Relationships between Subsystems

- There are two major types of hierarchical relationships
 - Layer A “depends on” layer B (compile time dependency)
 - Example: Build dependencies (make, ant, maven)
 - Layer A “calls” layer B (runtime dependency)
 - Example: A web browser calls a web server
 - Can the client and server layers run on the same machine?
 - Yes, they are layers, not processor nodes
 - Mapping of layers to processors is decided during the Software/hardware mapping!
- UML convention:
 - Runtime relationships are associations with dashed lines ----->
 - Compile time relationships are associations with solid lines. ———>

Example of a System with more than one Hierarchical Relationship



Example of a Layered Design (ARENA)

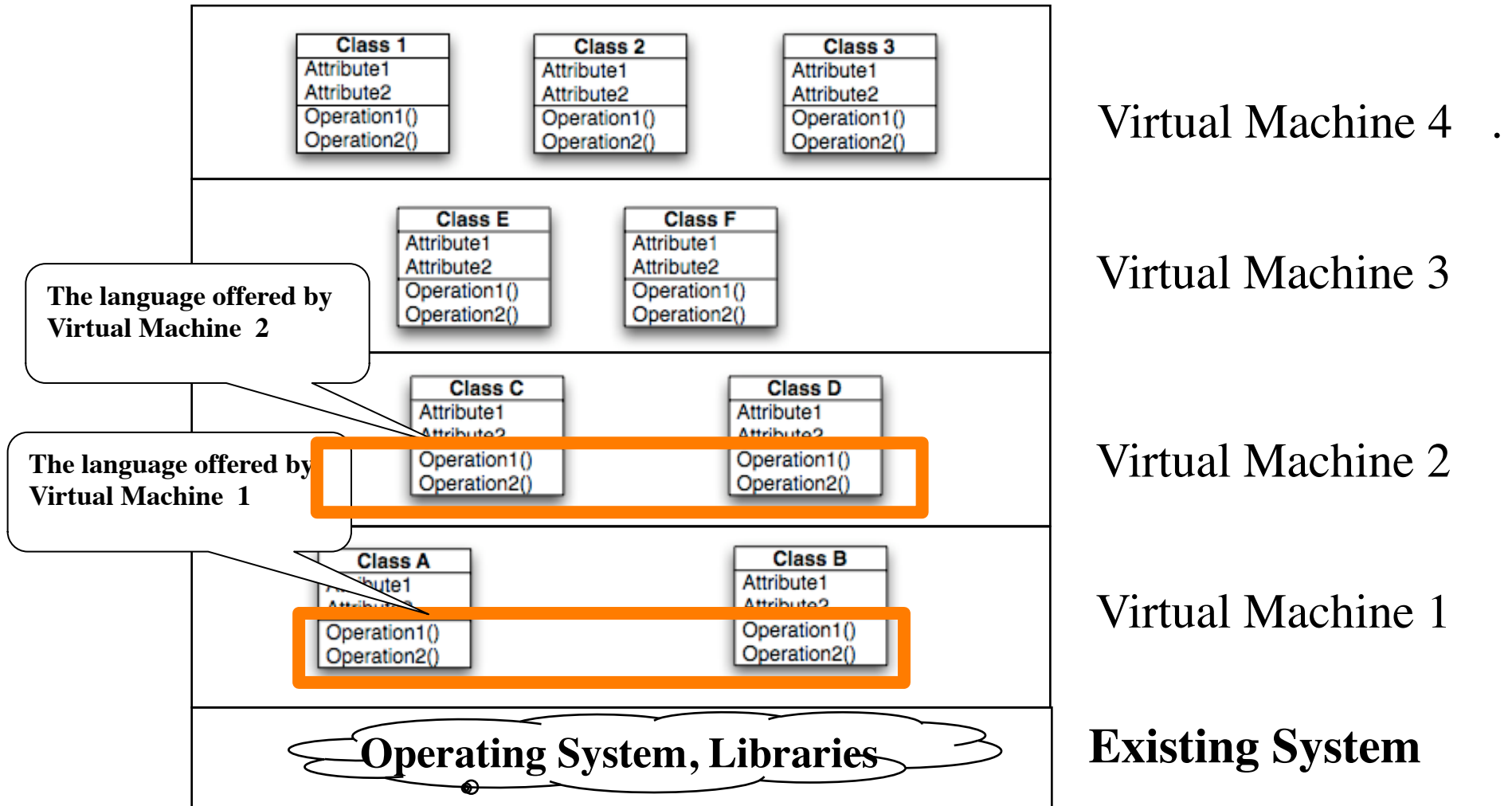


Virtual Machine

- A **virtual machine** is a subsystem connected to higher and lower level virtual machines by "provides services for" associations
- A virtual machine is an abstraction that provides a set of attributes and operations
- The terms **layer** and **virtual machine** can be used interchangeably
 - Also sometimes called “level of abstraction”.

Building Systems as a Set of Virtual Machines

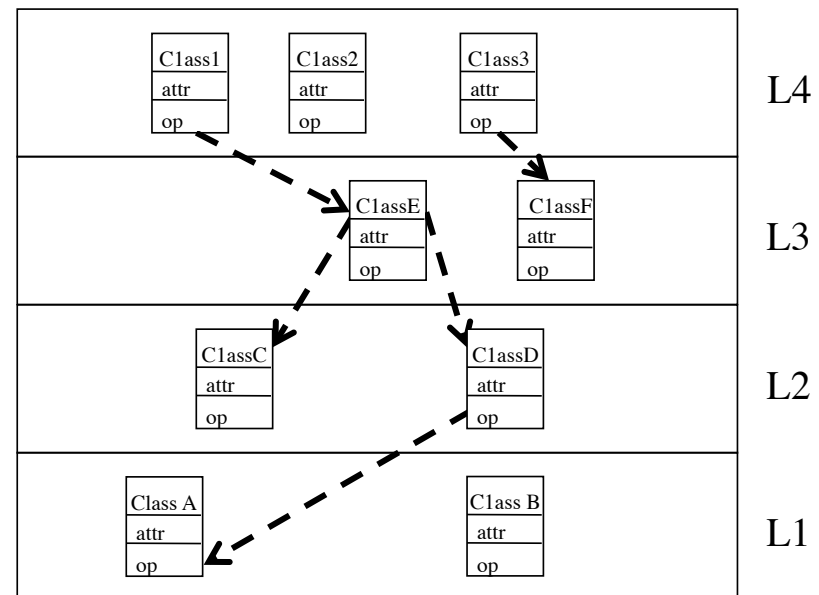
A system is a hierarchy of virtual machines, each using language primitives offered by the lower machines.



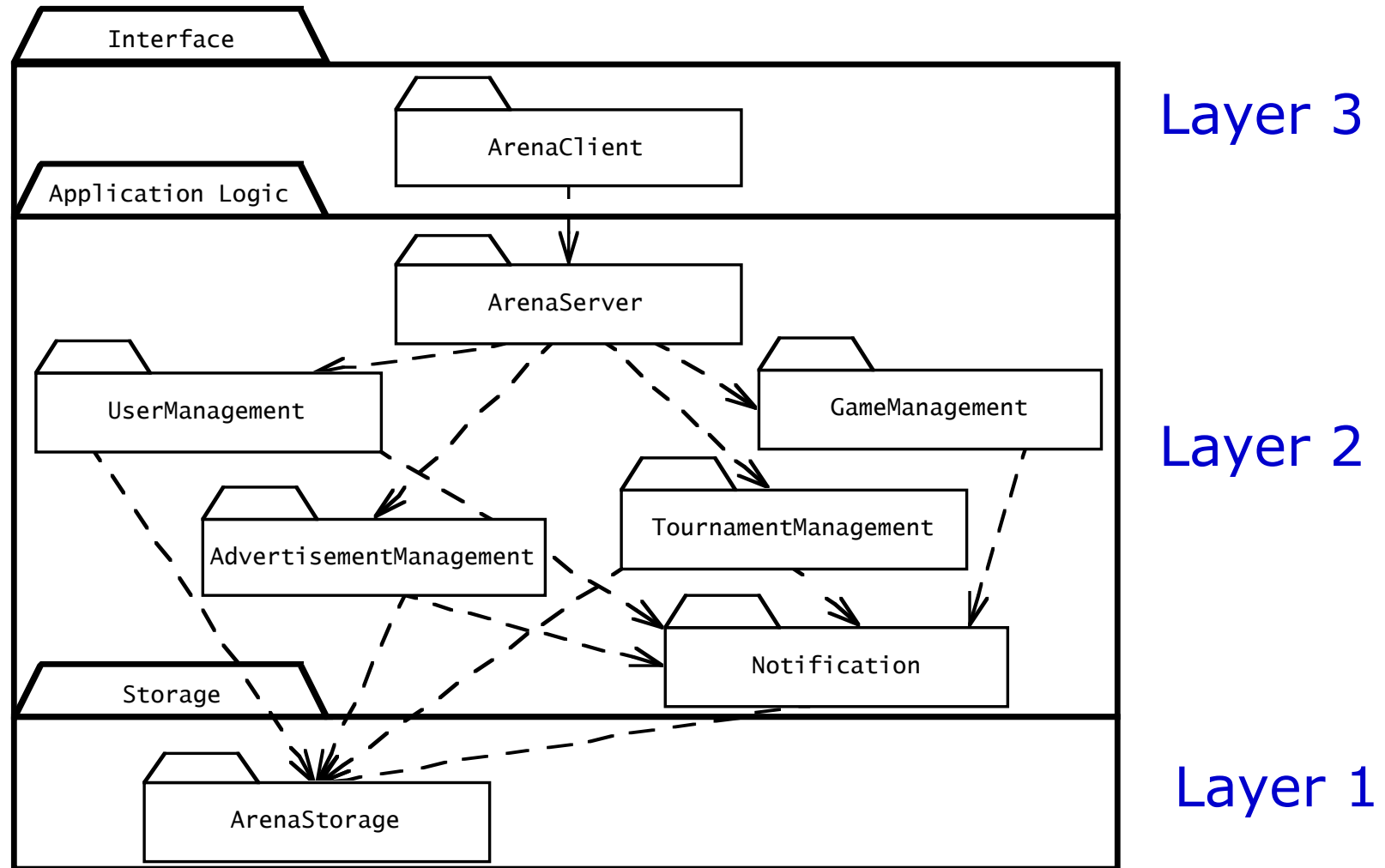
Closed Architecture (Opaque Layering)

- Each layer can only call operations from the layer below (called “direct addressing” by Buschmann et al)

Design goals:
Maintainability,
flexibility.



Opaque Layering in ARENA

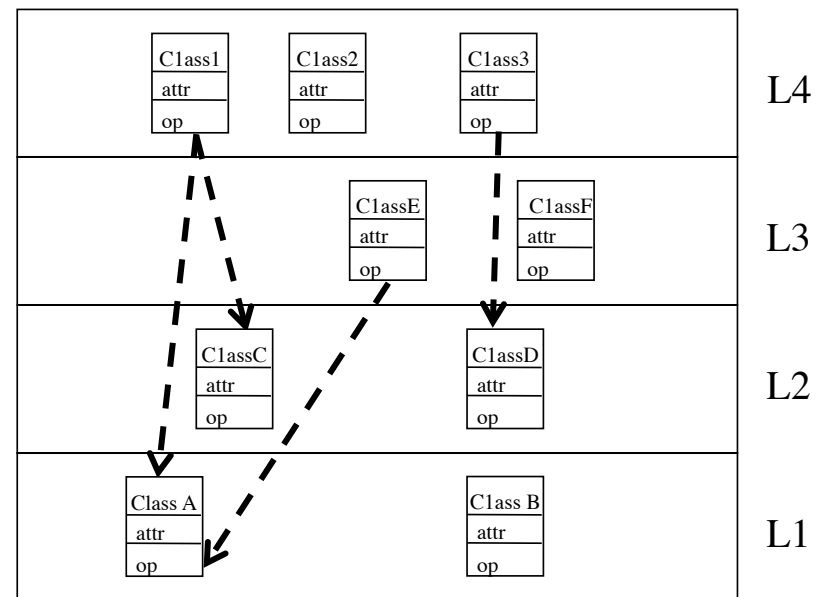


Open Architecture (Transparent Layering)

- Each layer can call operations from any layer below (“indirect addressing”)

Design goal:

Runtime efficiency.



SOA is a Layered Architectural Style

Service Oriented Architecture (SOA)

- Basic idea: A **service provider** (“business”) offers business services (“business processes”) to a **service consumer** (application, “customer”)
 - The business services are dynamically discoverable, usually offered in web-based applications
- The business services are created by composing (choreographing) them from lower-level services (basic services)
- The basic services are usually based on legacy systems
- Adapters are used to provide the “glue” between basic services and the legacy systems.

(Web-)Application

Business Services (Composite Services)

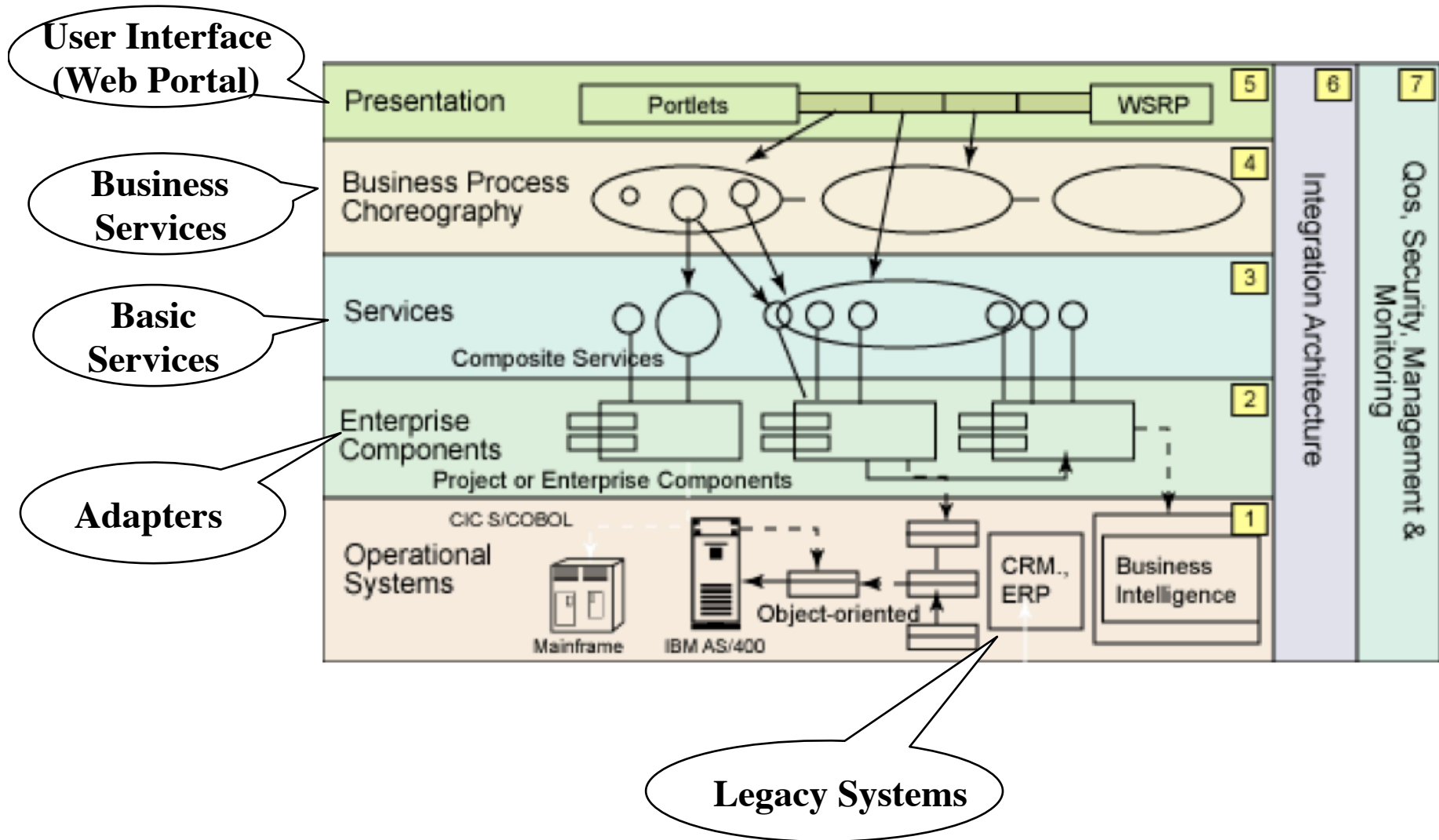
Basic Services

Adapters to Legacy Systems

Legacy Systems

IBM's View of a Service Oriented Architecture

Source <http://www.ibm.com/developerworks/webservices/library/ws-soa-design1/>

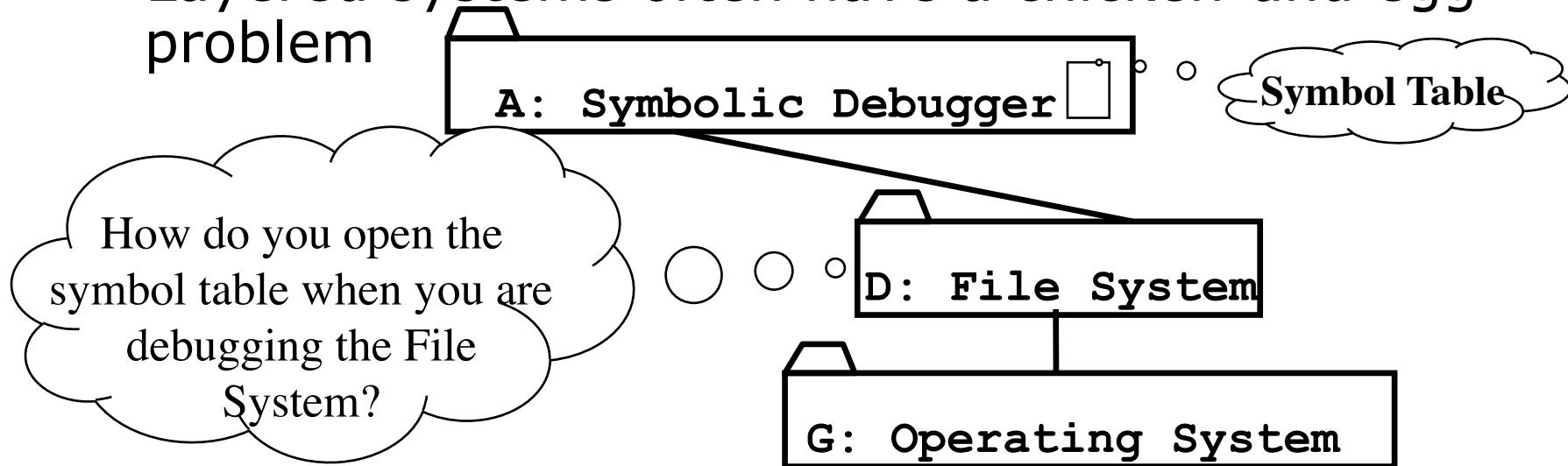


Properties of Layered Systems

- Layered systems are hierarchical. This is a desirable design
 - Hierarchy reduces complexity
- Closed architectures are more portable
 - Provide very low coupling
- Open architectures are more efficient

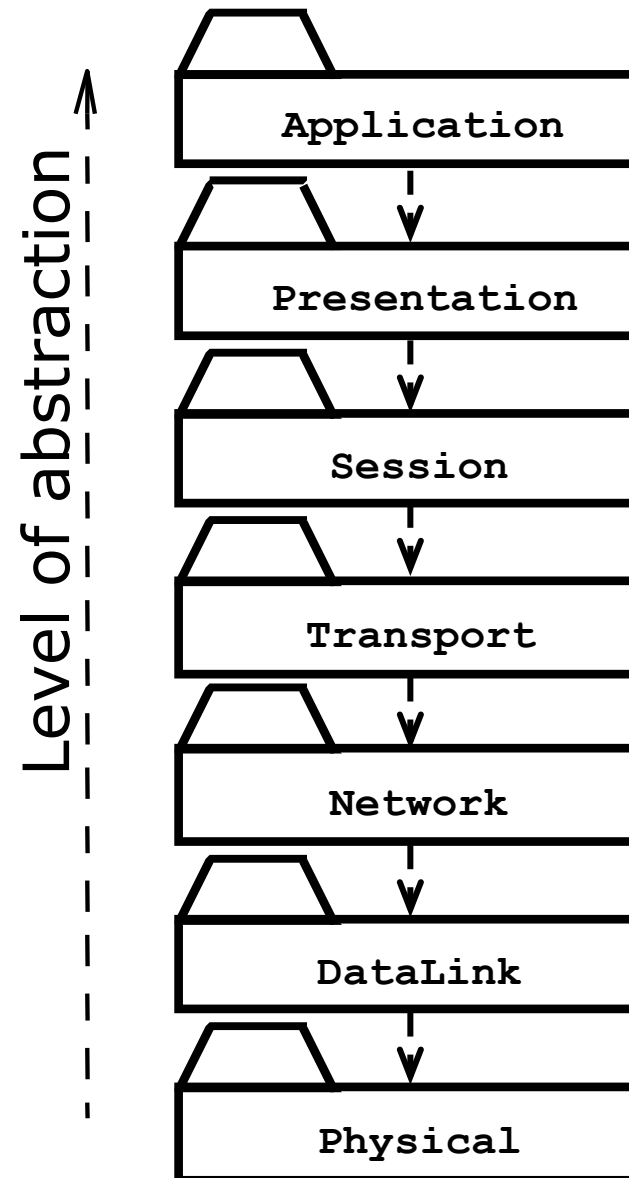
Properties of Layered Systems

- Layered systems are hierarchical. This is a desirable design
 - Hierarchy reduces complexity
- Closed architectures are more portable
 - Provide very low coupling
- Open architectures are more efficient
- Layered systems often have a chicken-and egg problem



Another Example of a Layered Architectural Style

- ISO's OSI Reference Model
 - ISO = International Standard Organization
 - OSI = Open System Interconnection
- Reference model which defines 7 layers and communication protocols between the layers



Examples of Architectural Styles

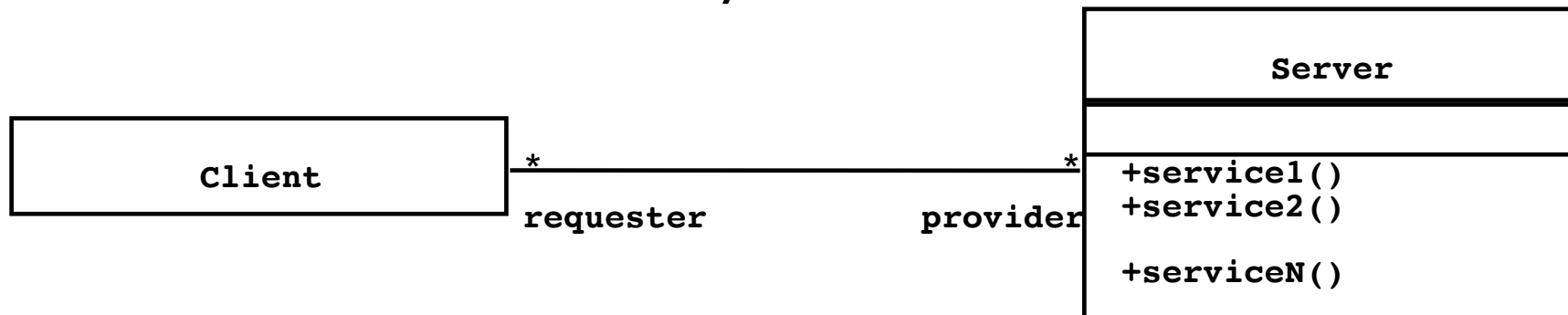
- ✓ Layered Architectural Style
 - ✓ Service-Oriented Architecture (SOA)
- Client/Server
- Peer-to-Peer
- Three-tier, Four-tier Architecture
- Repository
 - Blackboard
- Model-View-Controller
- Pipes and Filters

Client/Server Architectures

- Often used in the design of database systems
 - Front-end: User application (client)
 - Back end: Database access and manipulation (server)
- Functions performed by client:
 - Input from the user (Customized user interface)
 - Front-end processing of input data
- Functions performed by the database server:
 - Centralized data management
 - Data integrity and database consistency
 - Database security

Client/Server Architectural Style

- Special case of the Layered Architectural style
 - One or many **servers** provide services to instances of subsystems, called **clients**
- Each client calls on the server, which performs some service and returns the result
 - The clients know the *interface* of the server
 - The server does not need to know the interface of the client
- The response in general is immediate
- End users interact only with the client.

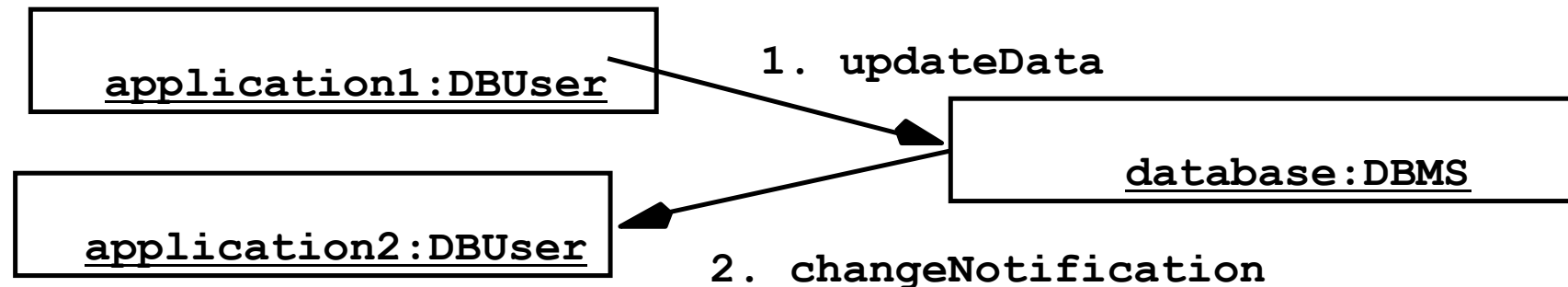


Design Goals for Client/Server Architectures

- Service Portability** Server runs on many operating systems and many networking environments
- Location-Transparency** Server might itself be distributed, but provides a single "logical" service to the user
- High Performance** Client optimized for interactive display-intensive tasks; Server optimized for CPU-intensive operations
- Scalability** Server can handle large # of clients
- Flexibility** User interface of client supports a variety of end devices (PDA, Handy, laptop, wearable computer)
- Reliability** Server should be able to survive client and communication problems.

Problems with Client/Server Architectures

- Client/Server systems do not provide peer-to-peer communication
- Peer-to-peer communication is often needed
- Example:
 - Database must process queries from application and should be able to send notifications to the application when data have changed



Peer-to-Peer Architectural Style

Generalization of Client/Server Architectural Style

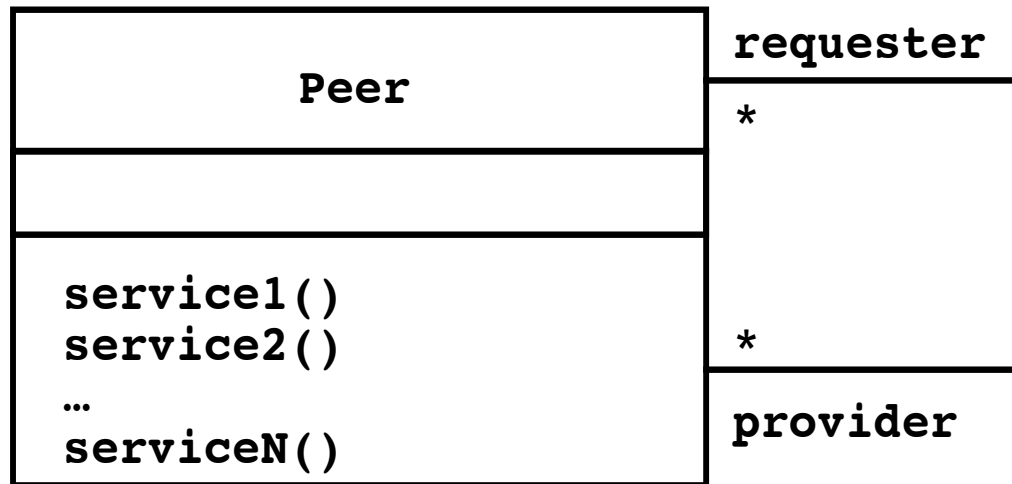
“Clients can be servers and servers can be clients”

Introduction a new abstraction: Peer

“Clients and servers can be both peers”

How do we model this statement? With Inheritance?

“A peer can be a client as well as a server”.



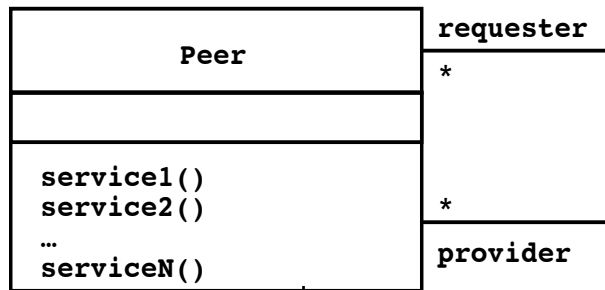
Relationship Client/Server & Peer-to-Peer

Problem statement “Clients can be servers and servers can be clients”

Which model is correct?

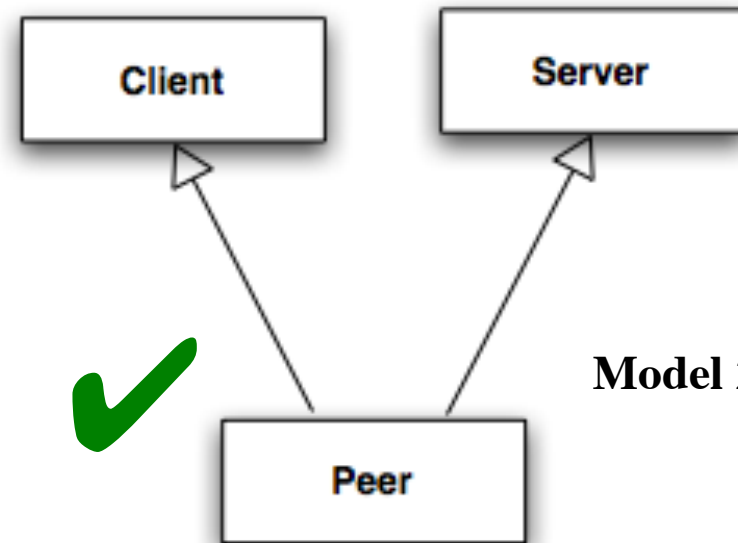
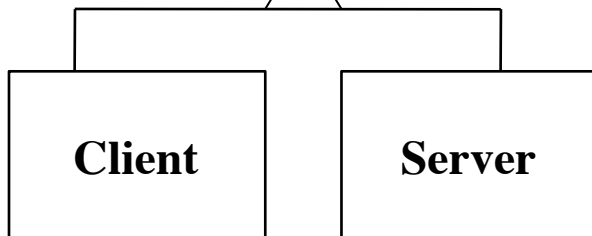
Model 1: “A peer can be either a client or a server”

Model 2: “A peer can be a client as well as a server”



Model 1

NO



Model 2

3-Layer-Architectural Style

3-Tier Architecture

Definition: 3-Layered Architectural Style

- An architectural style, where an application consists of 3 hierarchically ordered subsystems
 - A user interface, middleware and a database system
 - The middleware subsystem services data requests between the user interface and the database subsystem

Definition: 3-Tier Architecture

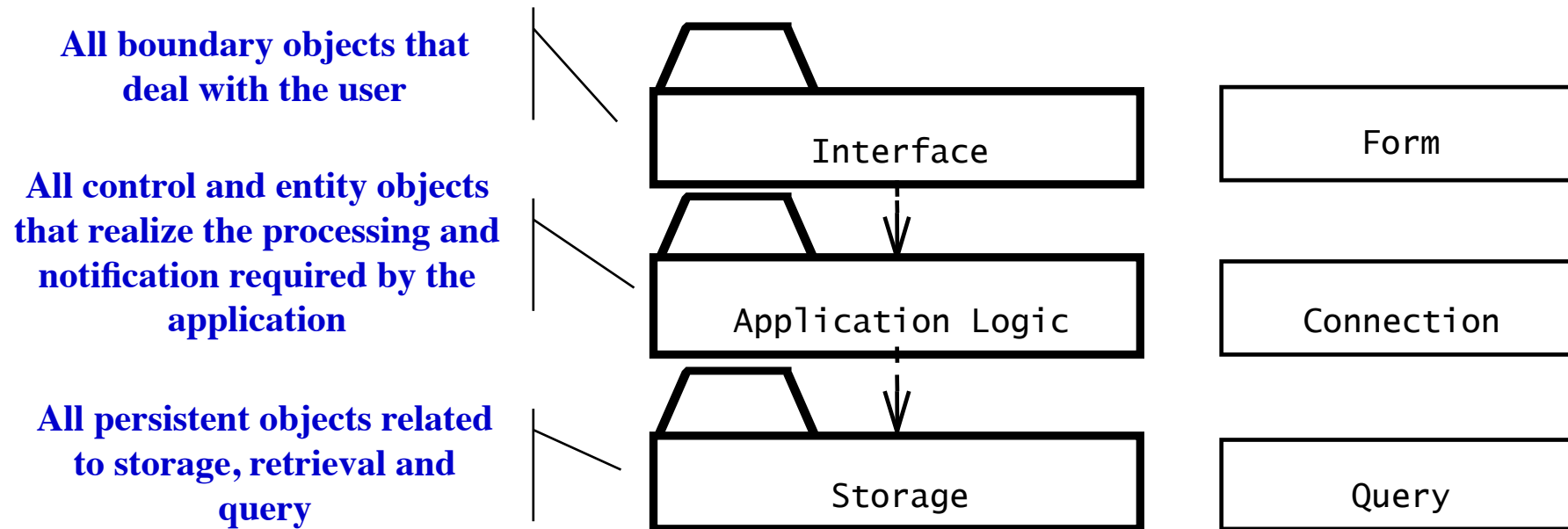
- A software architecture where the 3 layers are allocated on 3 separate hardware nodes
- Note: **Layer** is a type (e.g. class, subsystem) and **Tier** is an instance (e.g. object, hardware node)
- Layer and Tier are often used interchangeably.

Example of a 3-Layered Architectural Style

- Three-Layered Architectural style are often used for the development of Websites:
 1. The **Web Browser** implements the user interface
 2. The **Web Server** serves requests from the web browser
 3. The **Database** manages and provides access to the persistent data.

Three-tier architectural style.

- Subsystems are organized into three layers



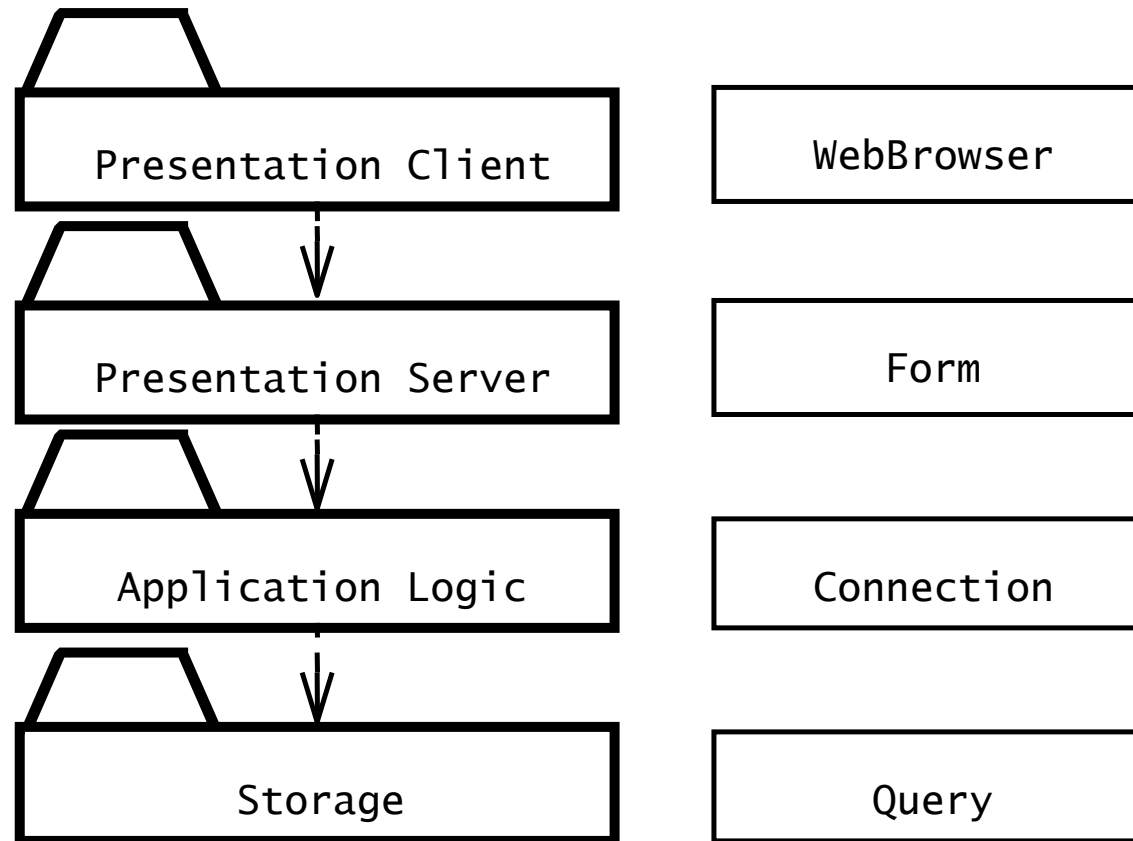
Example of a 4-Layered Architectural Style

4-Layer-architectural styles are usually used for the development of electronic commerce sites. The layers are

1. The **Web Browser**, providing the user interface
2. A **Web Server**, serving static HTML requests
3. An **Application Server**, providing session management (for example the contents of an electronic shopping cart) and processing of dynamic HTML requests
4. A back end **Database**, that manages and provides access to the persistent data
 - In commercially available 4-tier architectures, this is usually a relational database management system (RDBMS).

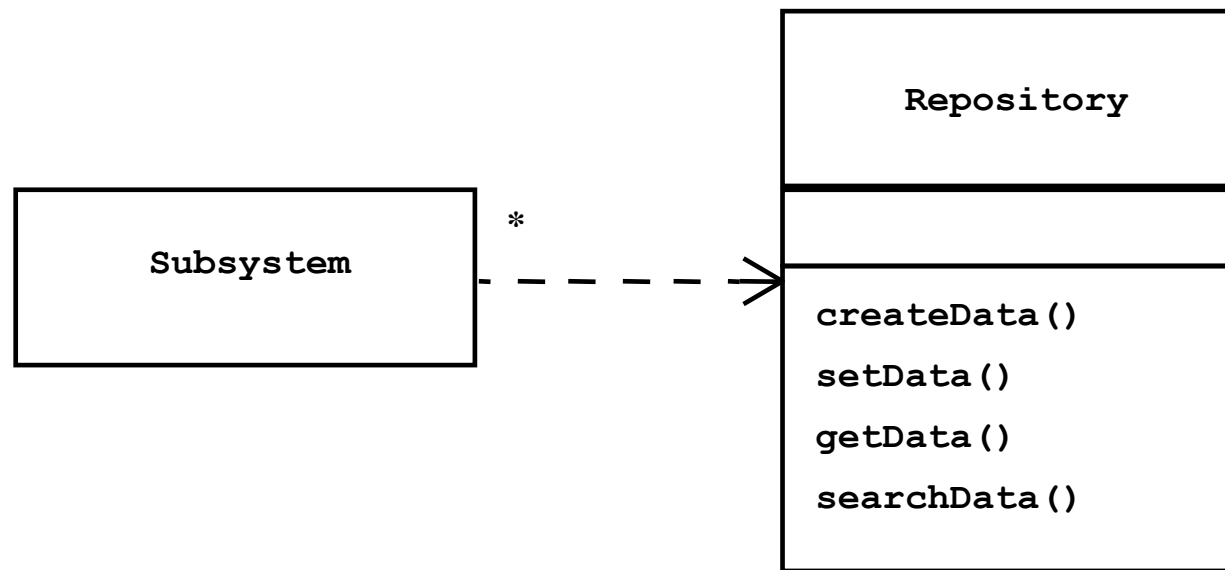
Four-tier architectural style.

- Subsystems are organized into four layers
- *Web-based applications*

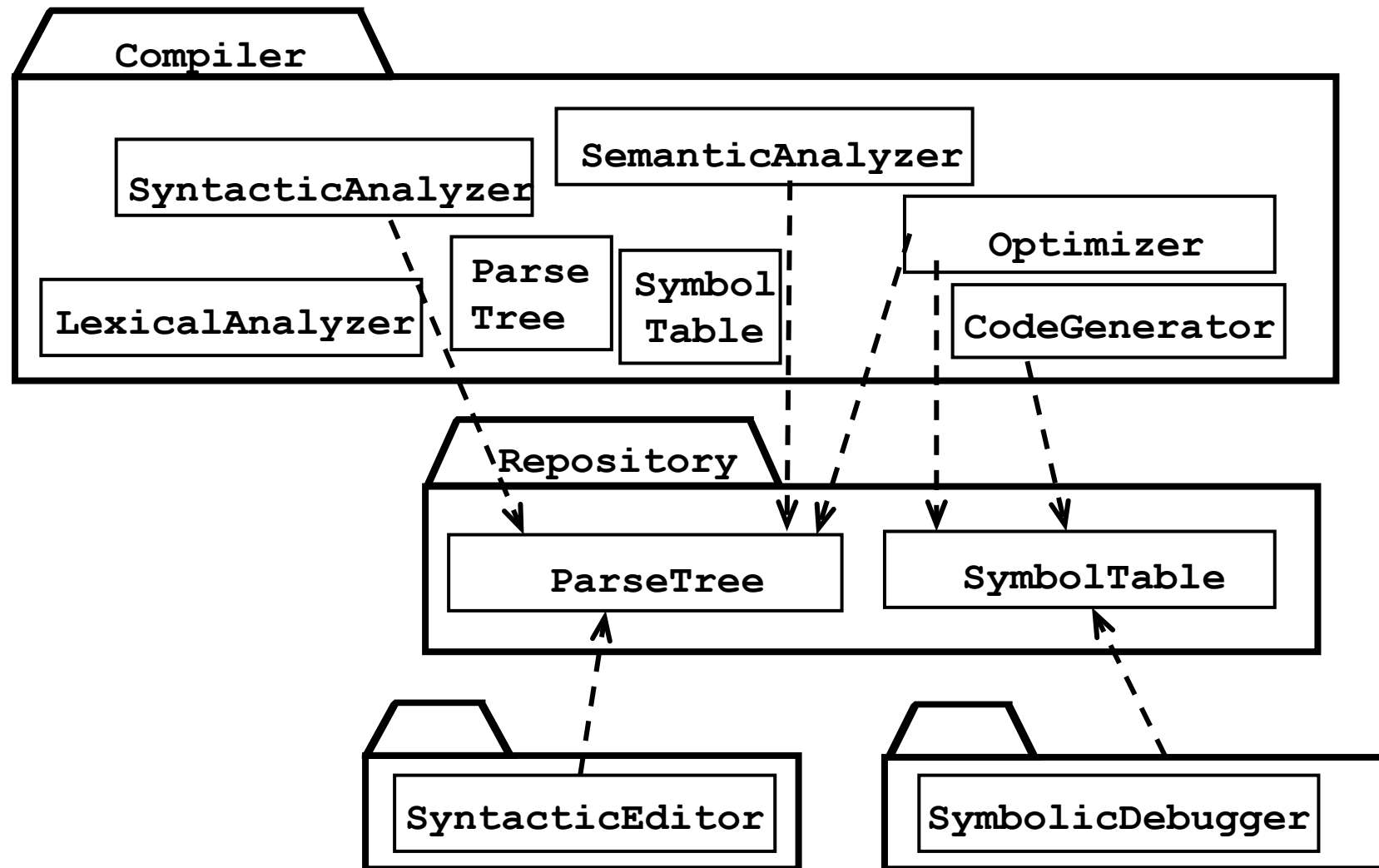


Repository Architectural Style

- The basic idea behind this architectural style is to support a collection of independent programs that work cooperatively on a common data structure called the **repository**
- **Subsystems** access and modify data from the repository. The subsystems are loosely coupled (they interact only through the repository).



Repository Architecture Example: Incremental Development Environment (IDE)



Repository architectures: when and why

- Repository architectures are **well suited for applications with constantly changing complex data processing tasks.**
- Once a central repository is well defined, we can **easily add new services** in the form of additional subsystems.
- The main **disadvantage** of repository systems is that the **central repository** can quickly **become a bottleneck**, both from a performance aspect and a modifiability aspect.
- The **coupling** between each subsystem and the repository **is high**, thus making it difficult to change the repository without having an impact on all subsystems.

Model-View-Controller Architectural Style

- **Problem:** In systems with high coupling changes to the user interface (boundary objects) often force changes to the entity objects (data)
 - The user interface cannot be reimplemented without changing the representation of the entity objects
 - The entity objects cannot be reorganized without changing the user interface
- **Solution:** Decoupling! The model-view-controller (MVC) architectural style decouples data access (entity objects) and data presentation (boundary objects)
 - Views: Subsystems containing boundary objects
 - Model: Subsystem with entity objects
 - Controller: Subsystem mediating between Views (data presentation) and Models (data access).

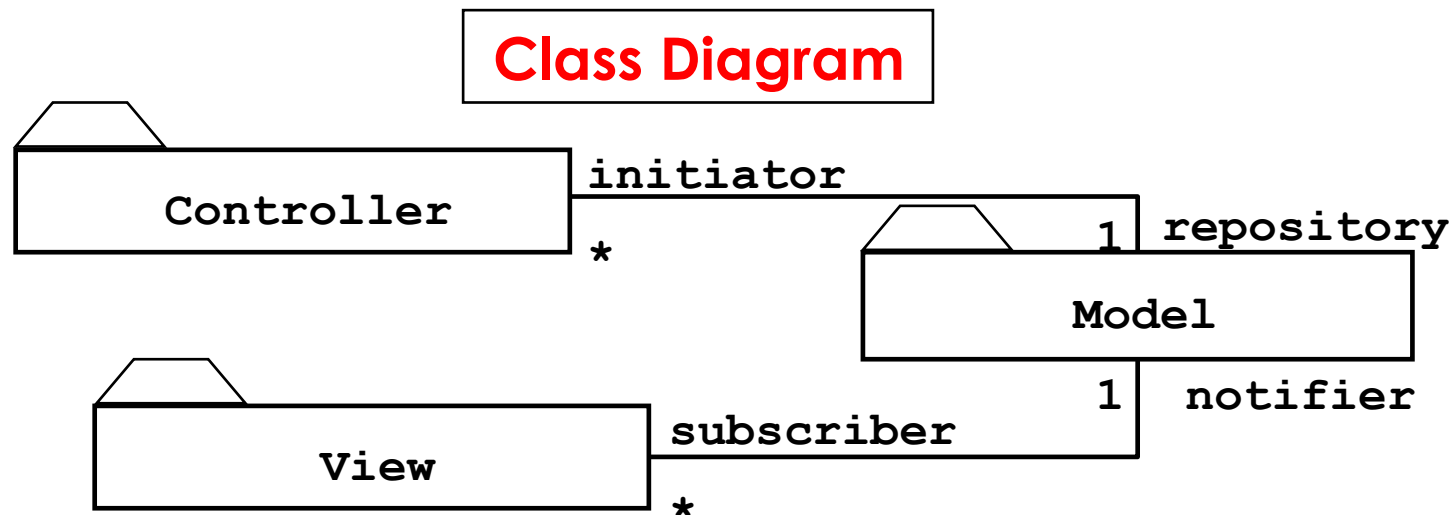
Model-View-Controller Architectural Style

- Subsystems are classified into 3 different types

Model subsystem: Responsible for application domain knowledge

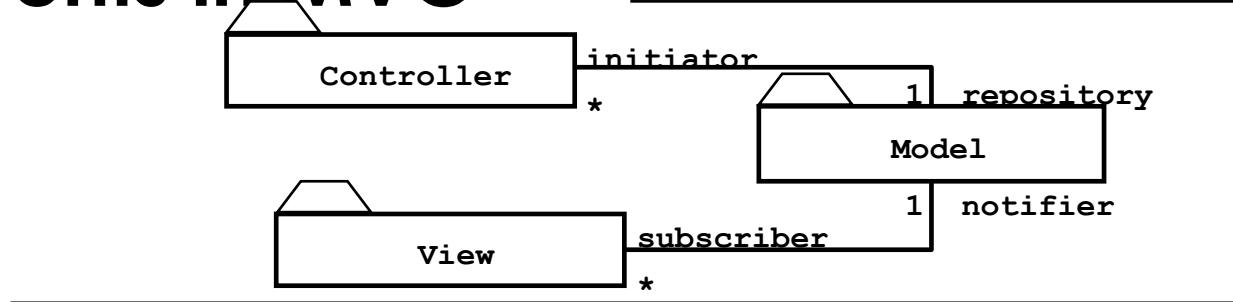
View subsystem: Responsible for displaying information to the user

Controller subsystem: Responsible for interacting with the user and notifying views of changes in the model

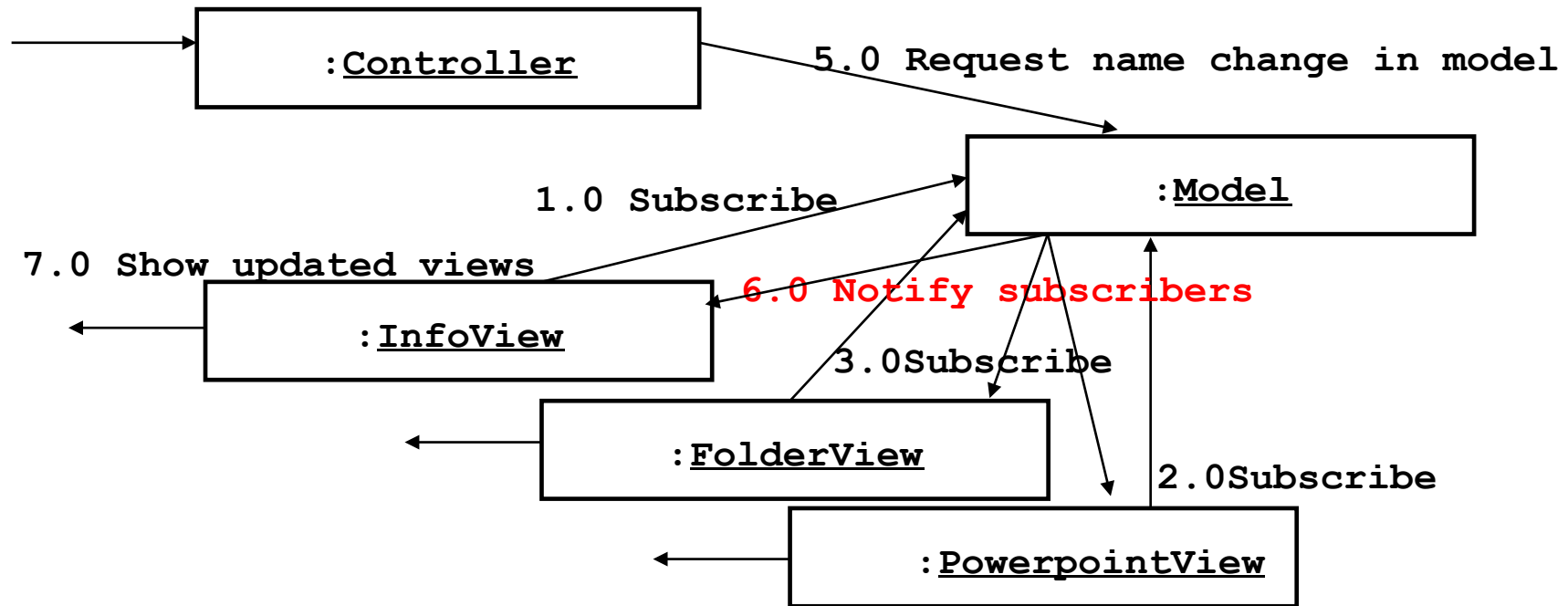


Example: Modeling the Sequence of Events in MVC

UML Class Diagram



4.0 User types new filename



UML Communication Diagram

Review: UML Communication Diagram

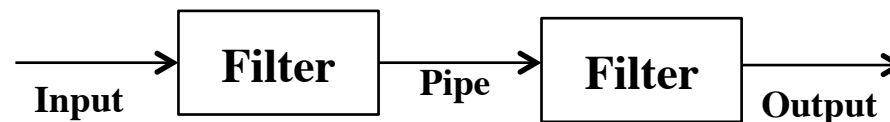
- A **Communication Diagram** visualizes the interactions between objects as a flow of messages. Messages can be events or calls to operations
- Communication diagrams **describe the static structure** as well as the **dynamic behavior of a system**:
 - The static structure is obtained from the UML class diagram
 - Communication diagrams reuse the layout of classes and associations in the class diagram
 - The dynamic behavior is obtained from the dynamic model (UML sequence diagrams and UML statechart diagrams)
 - Messages between objects are labeled with a number and placed near the link the message is sent over
- Reading a communication diagram involves starting at message 1.0, and following the messages from object to object.

MVC vs. 3-Tier Architectural Style

- The **MVC** architectural style is **nonhierarchical** (triangular):
 - View subsystem sends updates to the Controller subsystem
 - Controller subsystem updates the Model subsystem
 - View subsystem is updated directly from the Model
- The **3-tier** architectural style is **hierarchical** (linear):
 - The presentation layer never communicates directly with the data layer (opaque architecture)
 - All communication must pass through the middleware layer
- **History:**
 - MVC (1970-1980): Originated during the development of modular graphical applications for a single graphical workstation at Xerox Parc
 - 3-Tier (1990s): Originated with the appearance of Web applications, where the client, middleware and data layers ran on physically separate platforms.

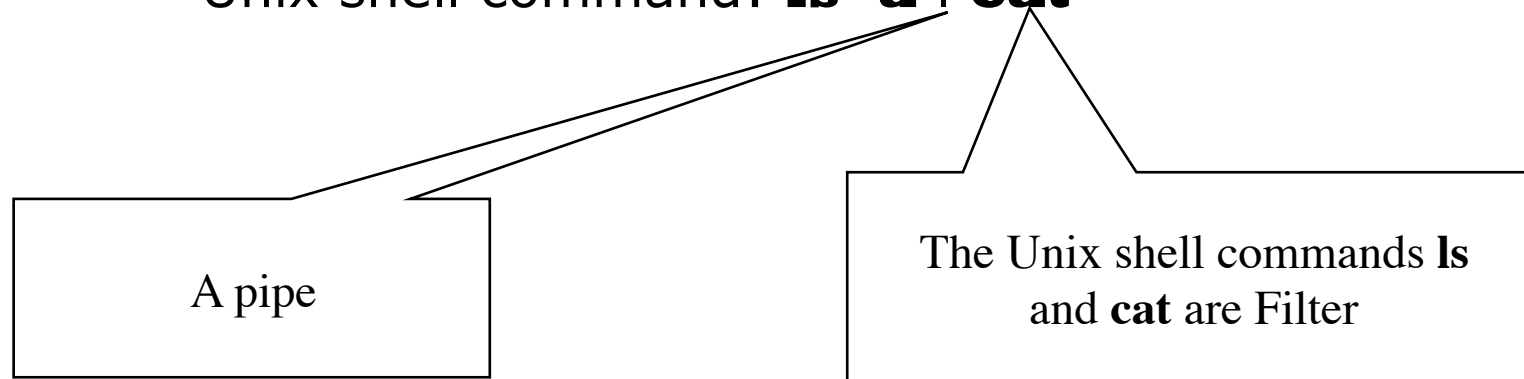
Pipes and Filters

- A **pipeline** consists of a chain of processing elements (processes, threads, etc.), arranged so that the output of one element is the input to the next element
 - Usually some amount of buffering is provided between consecutive elements
 - The information that flows in these pipelines is often a stream of records, bytes or bits.



Pipes and Filters Architectural Style

- An architectural style that consists of two subsystems called pipes and filters
 - **Filter**: A subsystem that does a processing step
 - **Pipe**: A Pipe is a connection between two processing steps
- Each filter has an input pipe and an output pipe.
 - The data from the input pipe are processed by the filter and then moved to the output pipe
- Example of a Pipes-and-Filters architecture: Unix
 - Unix shell command: **ls -a | cat**



Summary

- System Design
 - Reduces the gap between problem and existing machine
- Design Goals
 - Describe important system qualities and values against which alternative designs are evaluated (design-tradeoffs)
 - Additional nonfunctional requirements found at design time
- Subsystem Decomposition
 - Decomposes the overall system into manageable part by using the principles of cohesion and coherence
- Architectural Style
 - A pattern for a subsystem decomposition: All kind of layer styles (C/S, SOA, n-Tier), Repository, MVC, Pipes&Filters
- Software architecture
 - An instance of an architectural style.

Additional Readings

- E.W. Dijkstra (1968)
 - The structure of the T.H.E Multiprogramming system, Communications of the ACM, 18(8), pp. 453-457
- D. Parnas (1972)
 - On the criteria to be used in decomposing systems into modules, CACM, 15(12), pp. 1053-1058
- J.D. Day and H. Zimmermann (1983)
 - The OSI Reference Model, Proc. IEEE, Vol.71, 1334-1340
- Jostein Gaarder (1991)
 - Sophie 's World: A Novel about the History of Philosophy
- Frank Buschmann et al:
 - Pattern-Oriented Software Architecture, Vol 1: A System of Patterns, Wiley, 1996.