

CSCI 4050 / 6050

Software Engineering

System Design

Modified and extended slides from Bruegge and Dutoit

Overview

System Design I (This Segment)

0. Overview of System Design
1. Design Goals
2. Subsystem Decomposition, Software Architecture

System Design II (Next Segment)

3. Concurrency: Identification of parallelism
4. Hardware/Software Mapping:
Mapping subsystems to processors
5. Persistent Data Management: Storage for entity objects
6. Global Resource Handling & Access Control:
Who can access what?
7. Software Control: Who is in control?
8. Boundary Conditions: Administrative use cases.

- Software comes in all shapes and sizes. The architecture you choose will affect every part of your software, from its security and efficiency, to its modularity and maintainability. In this module we will examine the different architectures that you have to choose from to shape your software.

Design is Difficult

- There are two ways of constructing a software design (Tony Hoare):
 - One way is to make it so simple that there are obviously no deficiencies
 - The other way is to make it so complicated that there are no obvious deficiencies.”
- Corollary (Jostein Gaarder):
 - If our brain would be so simple that we could understand it, we would be too stupid to understand it.



Sir **Antony Hoare**, b. 1934

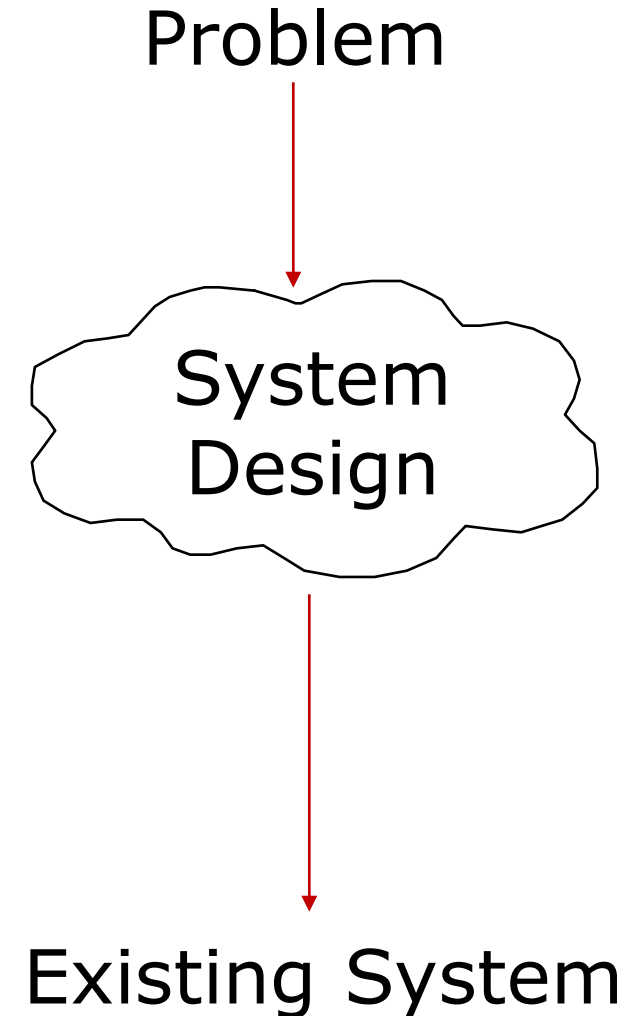
- Quicksort
- Hoare's logic for verification
- **CSP** (Communicating Sequential Processes): modeling language for concurrent processes (basis for Occam).

Why is Design so Difficult?

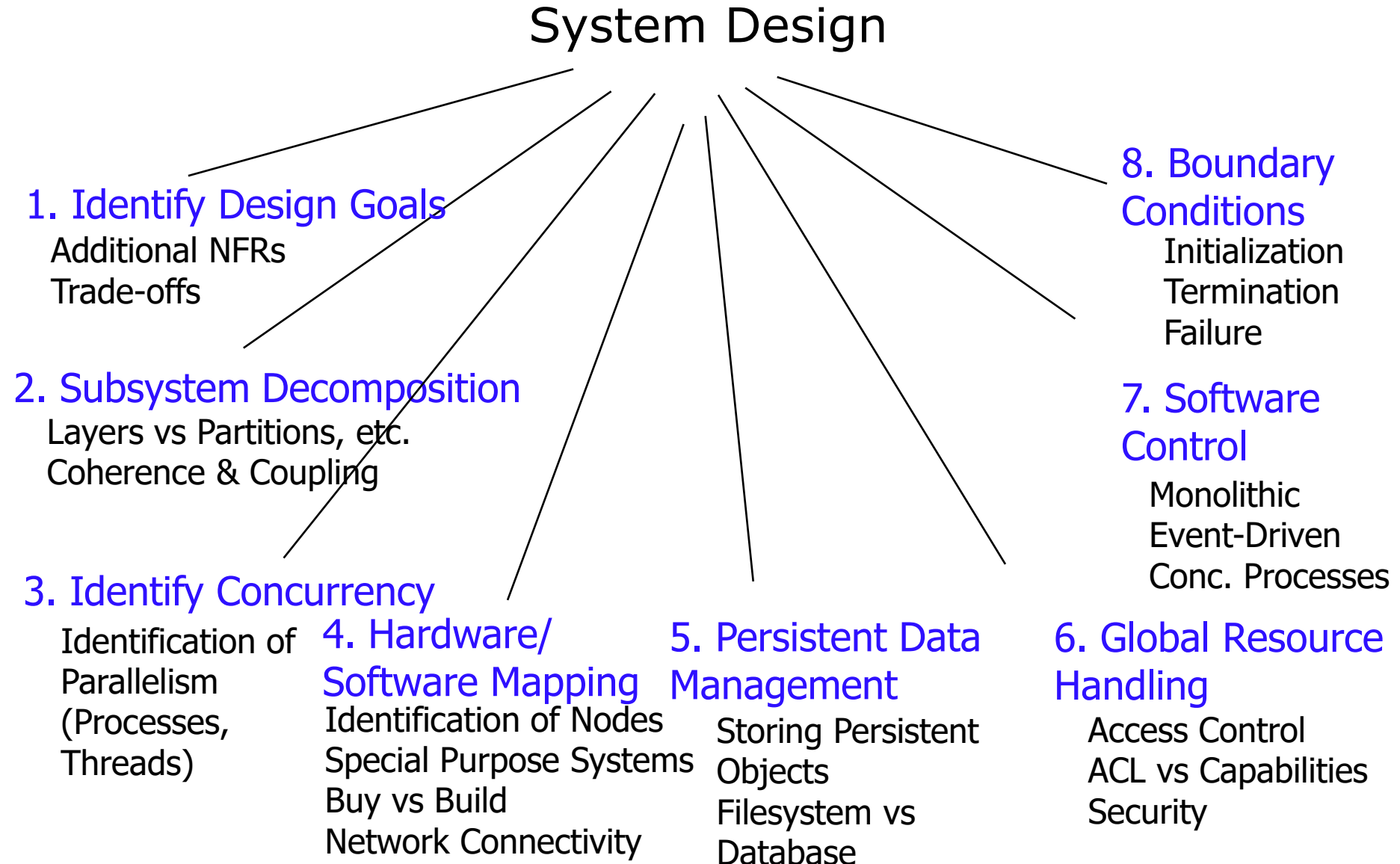
- **Analysis:** Focuses on the application domain
- **Design:** Focuses on the solution domain
 - The solution domain is changing very rapidly
 - Halftime knowledge in software engineering: About 3-5 years
 - Cost of hardware rapidly sinking
 - Design knowledge is a moving target
- **Design window:**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

The Scope of System Design

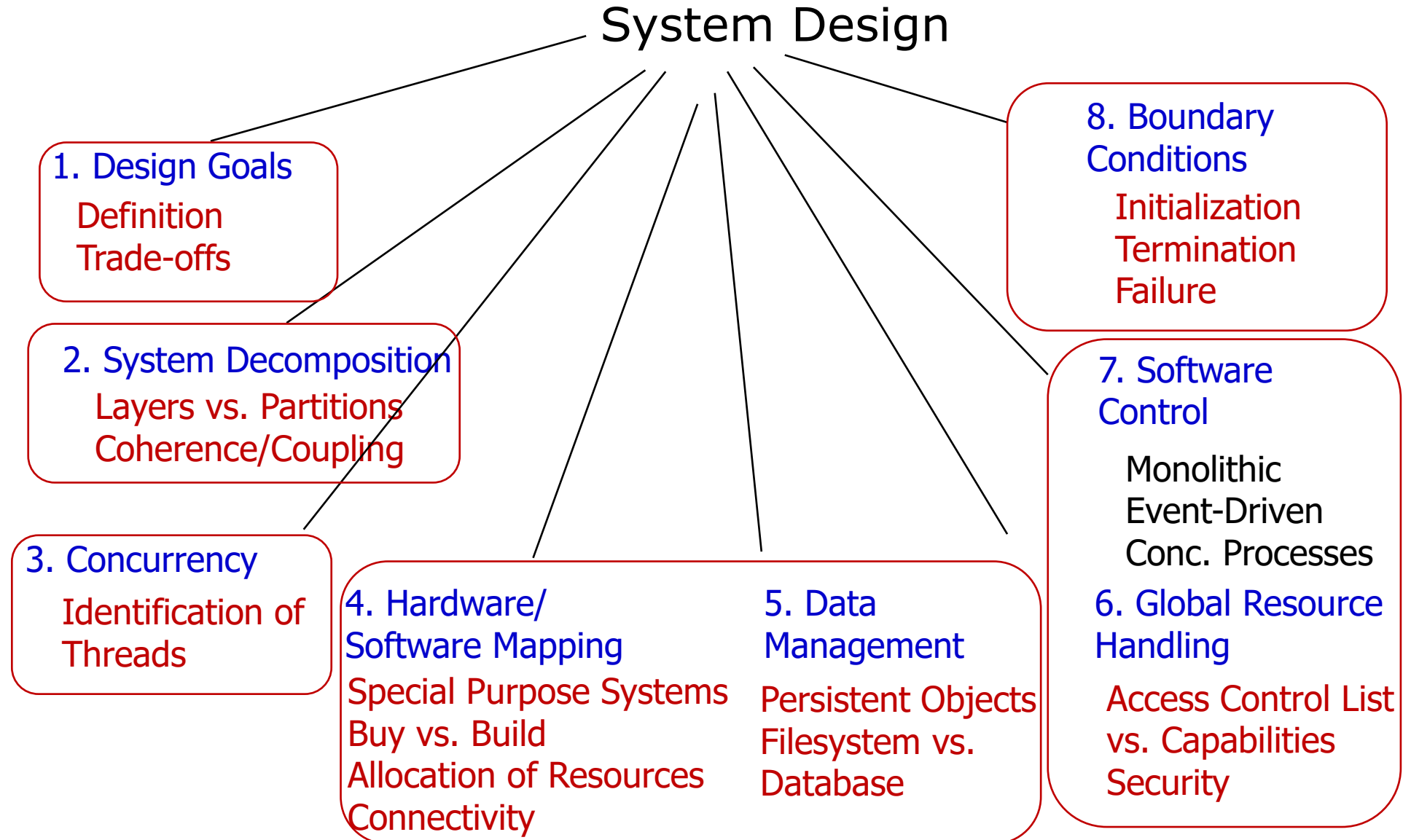
- Bridge the gap
 - between a problem and an existing system in a manageable way
 - How?
 - Use Divide & Conquer:
 - 1) Identify design goals
 - 2) Model the new system design as a set of subsystems
 - 3-8) Address the major design goals.



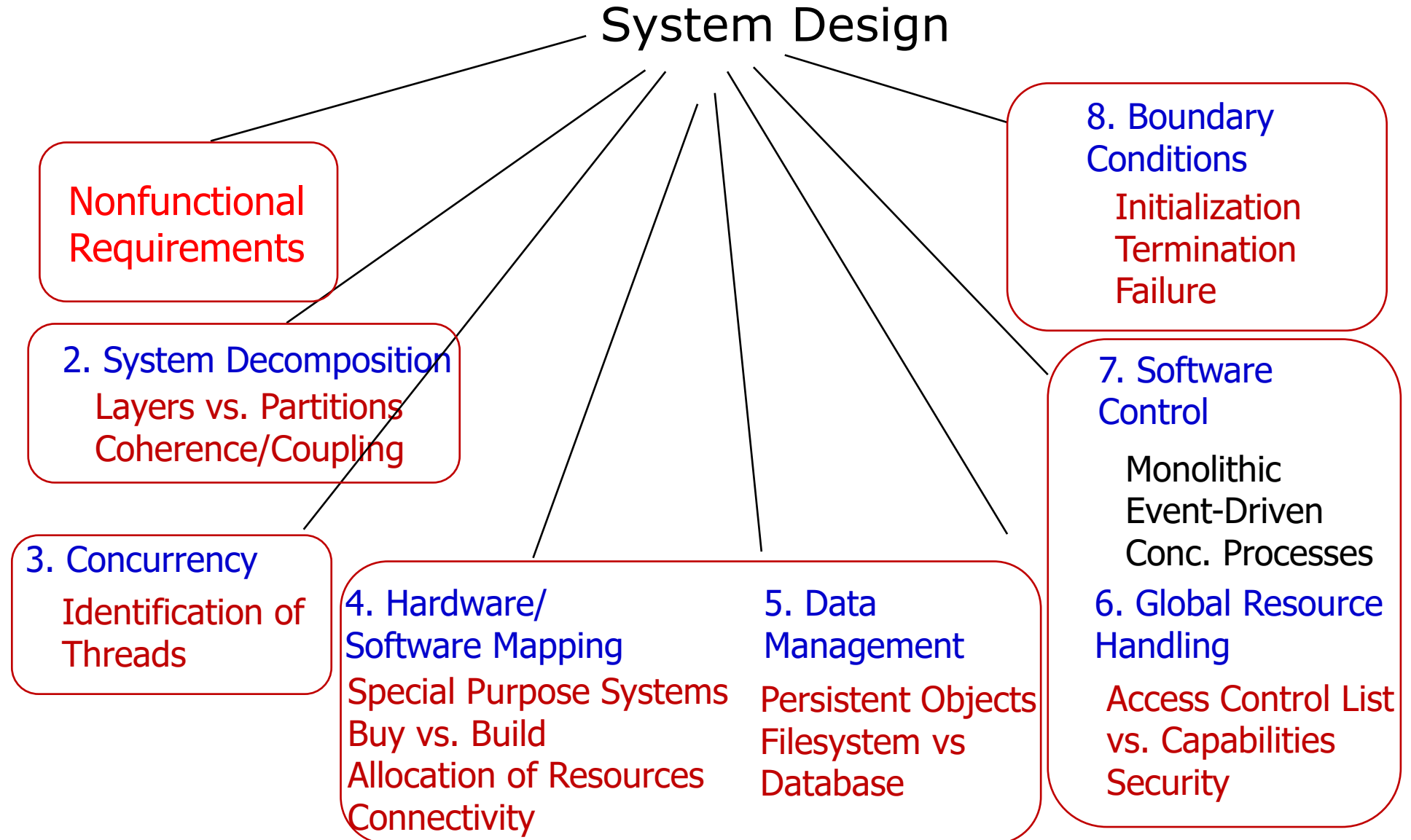
System Design: Eight Issues



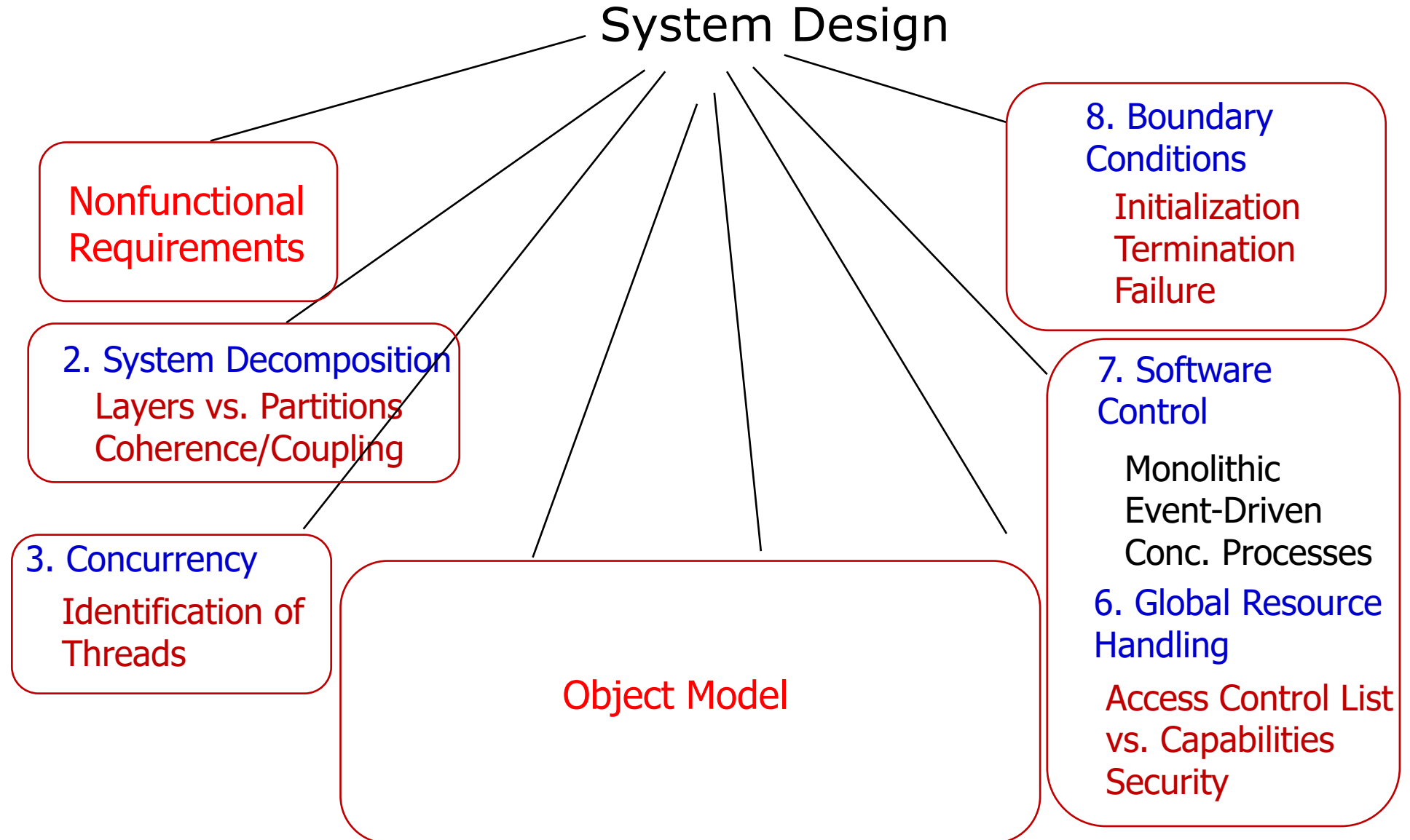
Analysis Sources: Requirements and System Model



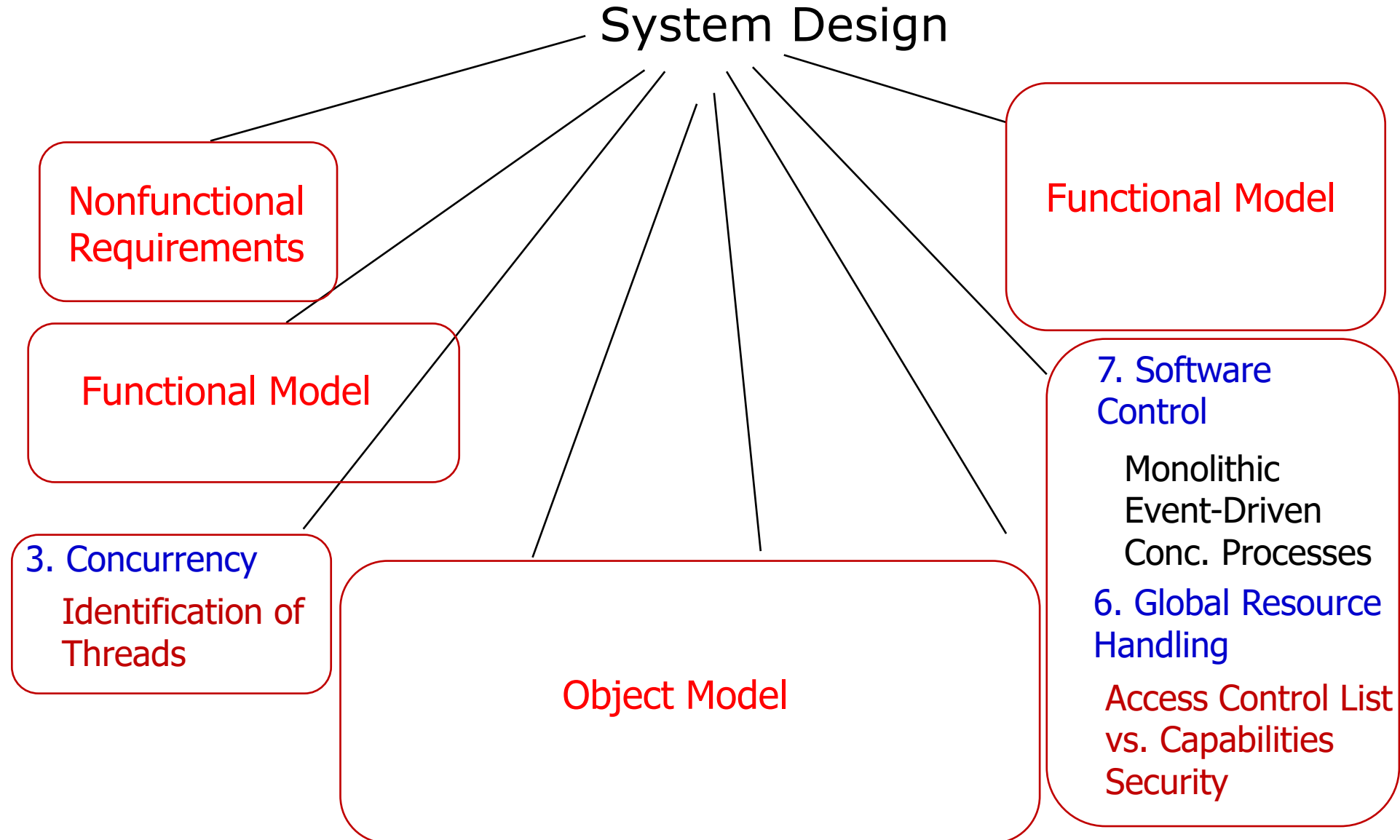
Analysis Sources: Requirements and System Model



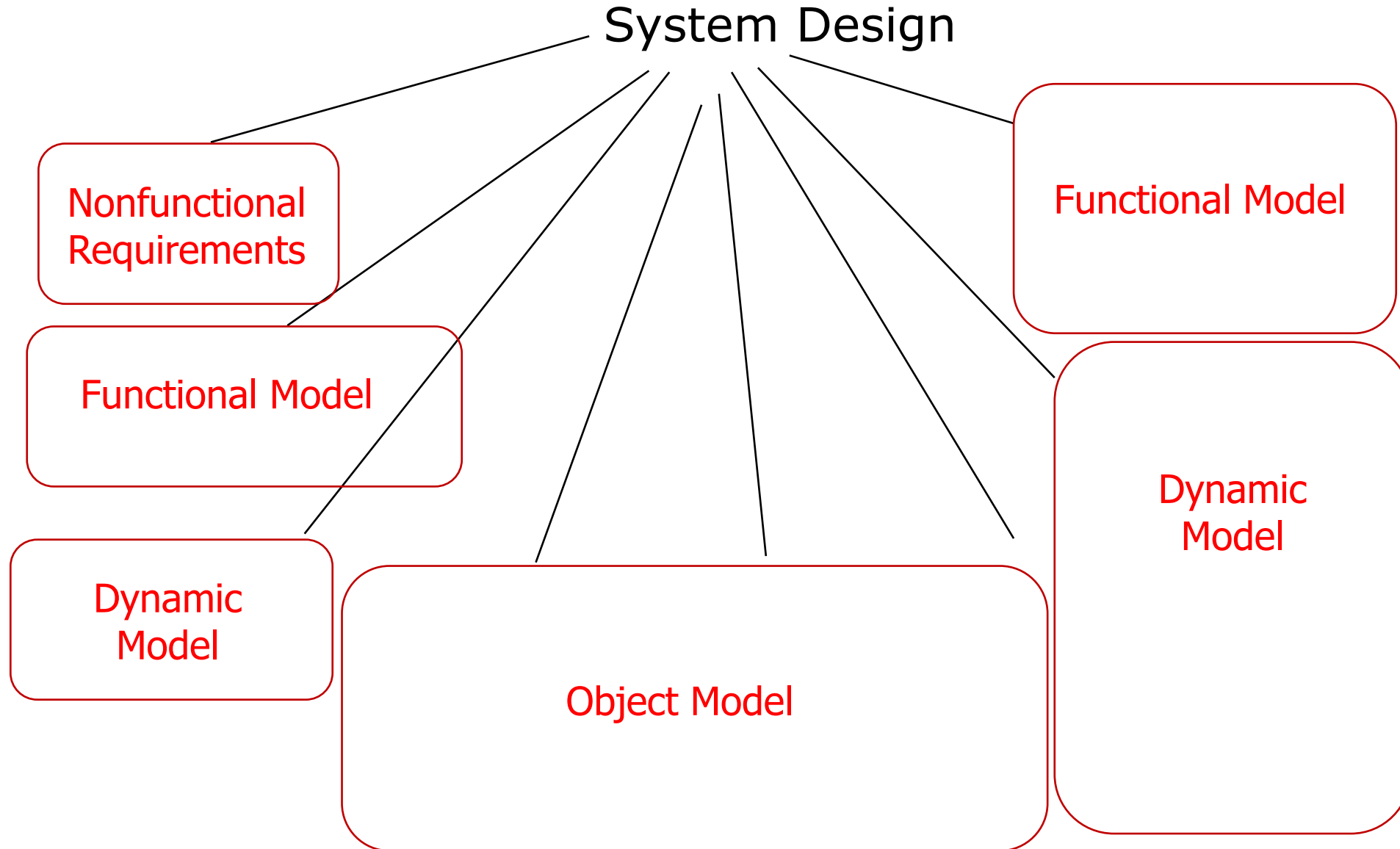
Analysis Sources: Requirements and System Model



Analysis Sources: Requirements and System Model



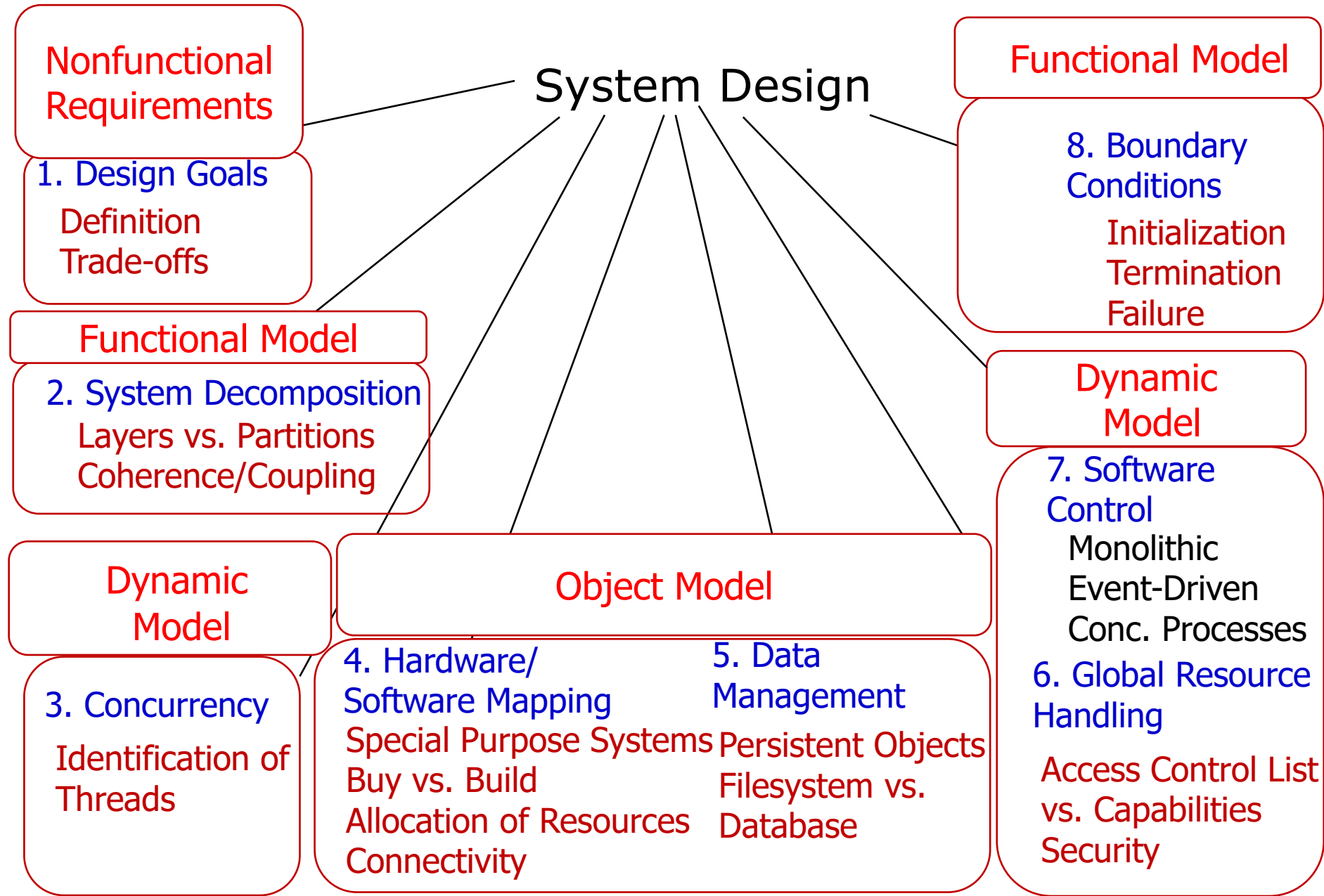
Analysis Sources: Requirements and System Model



How the Analysis Models influence System Design

- Nonfunctional Requirements
 - => Definition of Design Goals
- Functional model
 - => Subsystem Decomposition
- Object model
 - => Hardware/Software Mapping, Persistent Data Management,
- Dynamic model
 - => Identification of Concurrency, Global Resource Handling,
Software Control
- Object model
 - => Boundary conditions

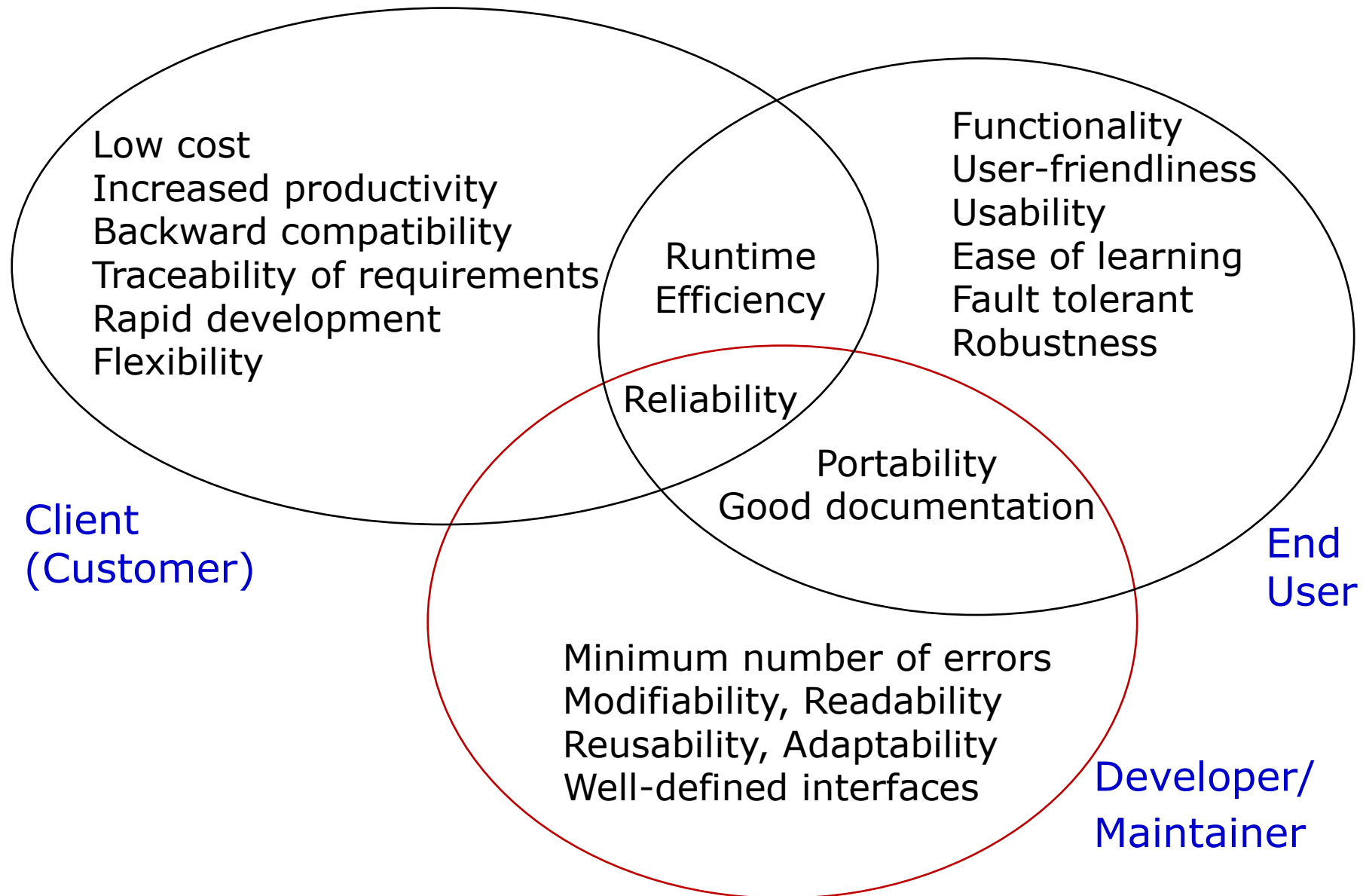
From Analysis to System Design



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

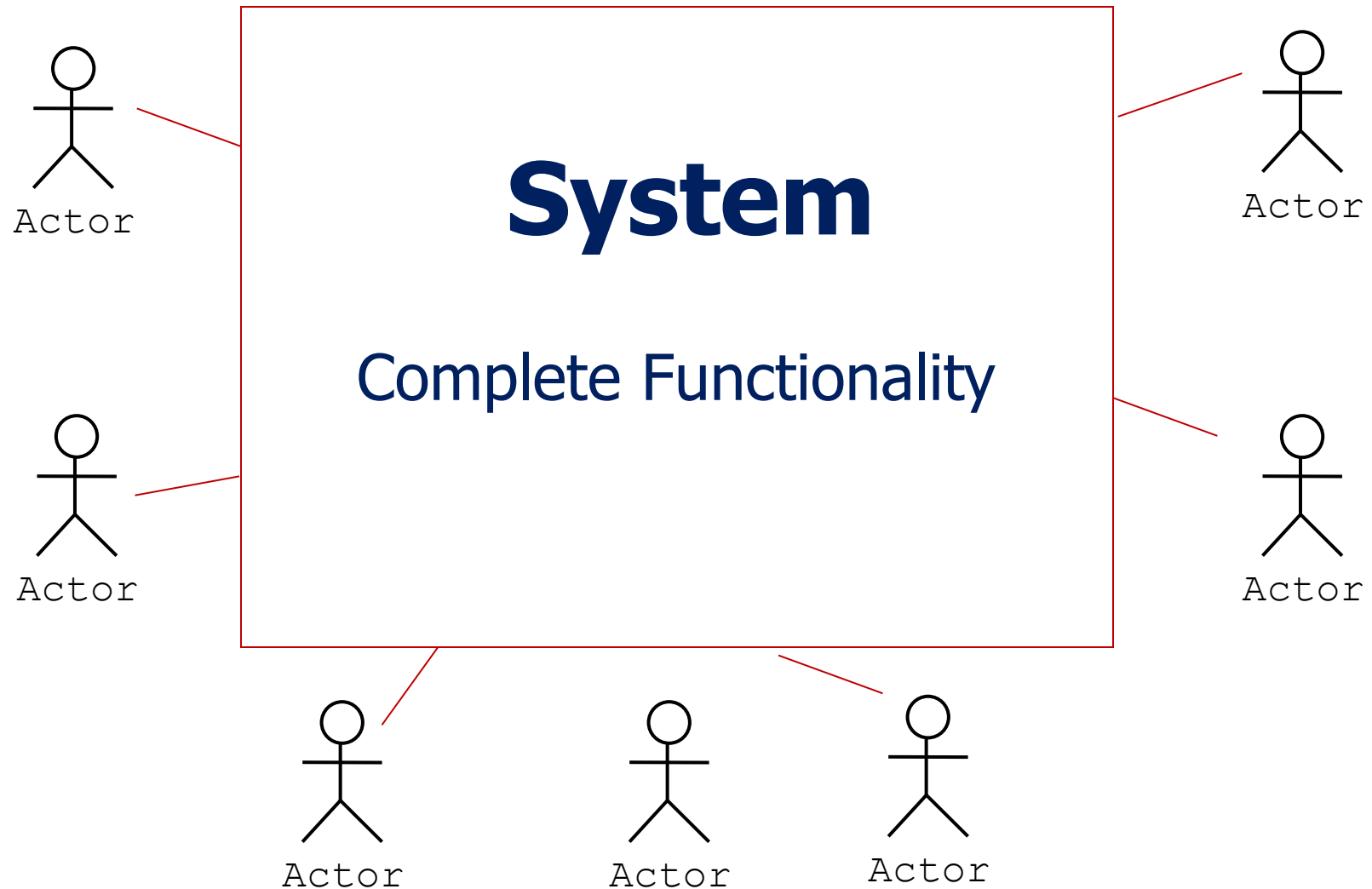
Stakeholders have different Design Goals



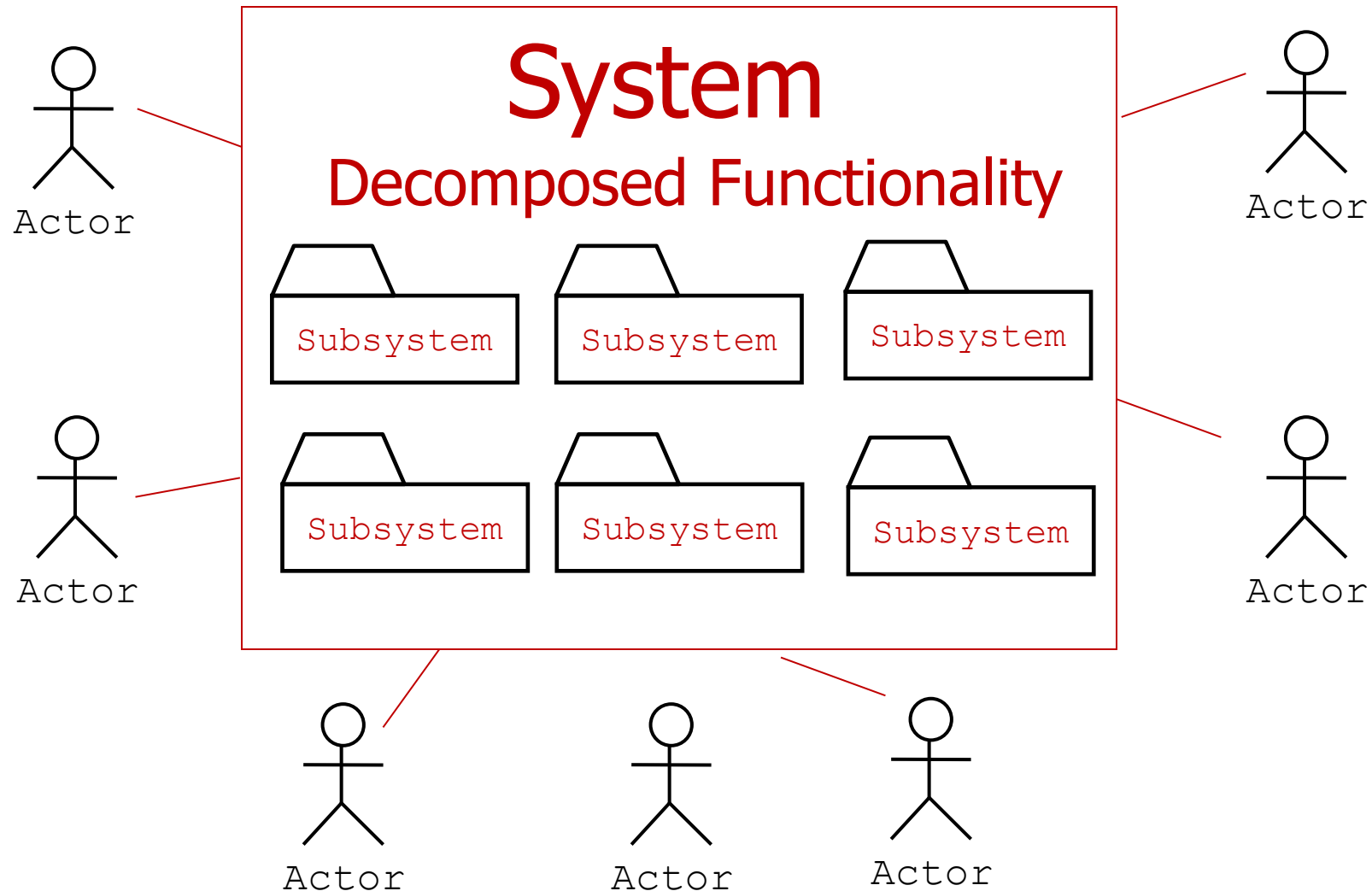
Typical Design Trade-offs

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

Software System design: complete functionality



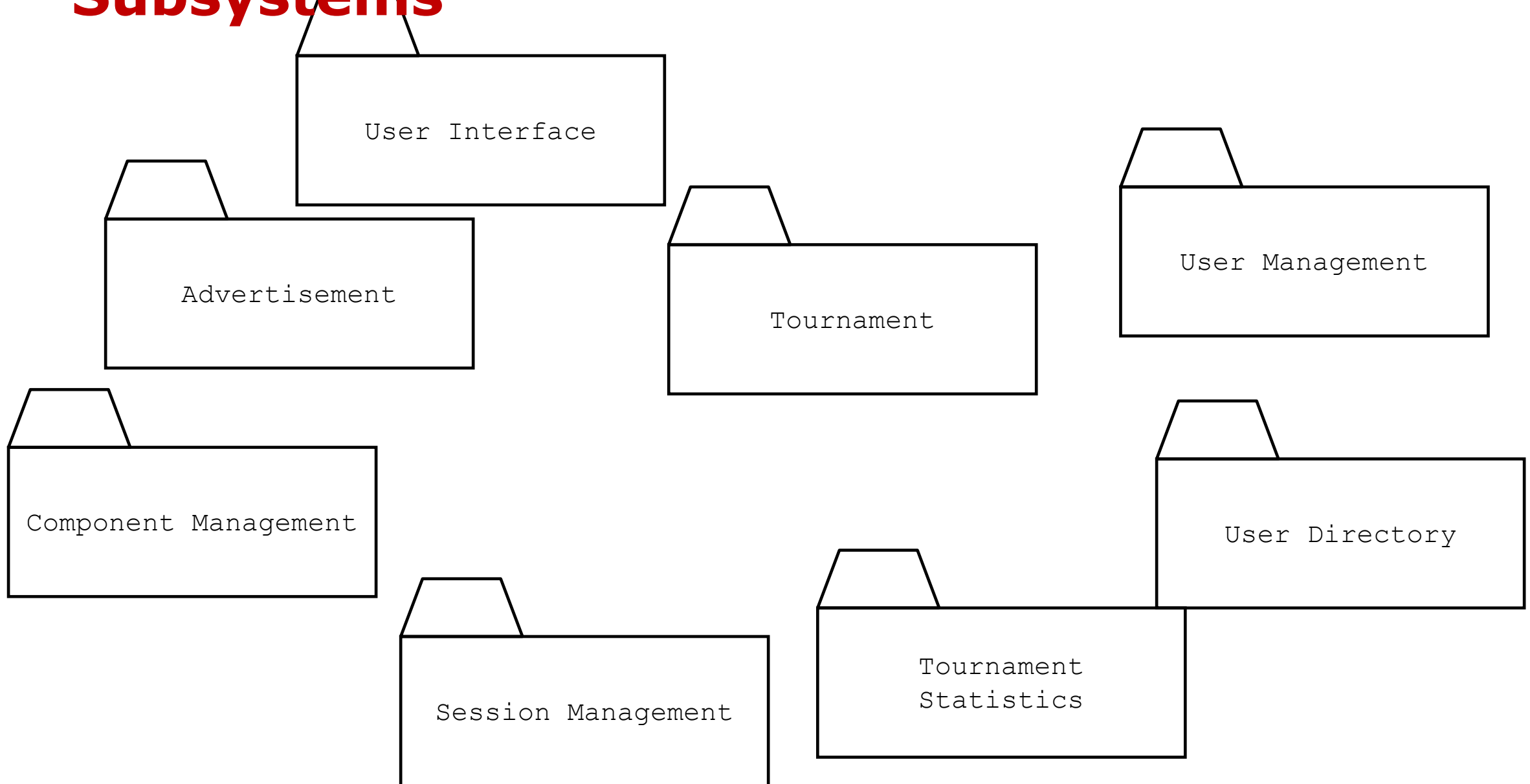
Software System design: functionality split into subsystems



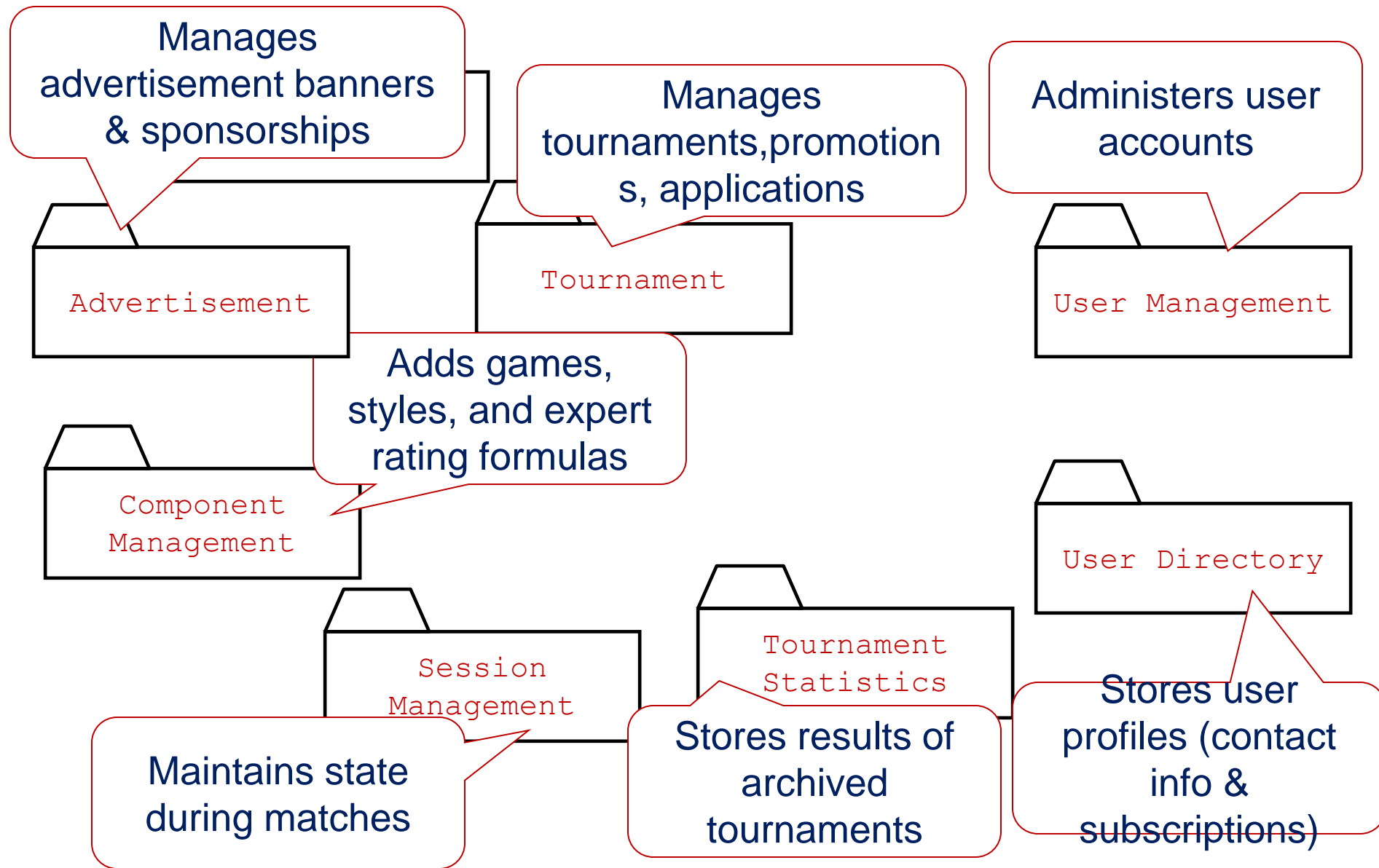
Subsystem Decomposition

- **Subsystem**
 - Collection of classes, associations, operations, events and constraints that are closely interrelated with each other
 - The objects and classes from the object model are the “seeds” for the subsystems
 - In UML subsystems are modeled as packages
- **Service**
 - A set of named operations that share a common purpose
 - The original information for services (“seed”) are the use cases from the functional model
- **Services are defined during system design.**

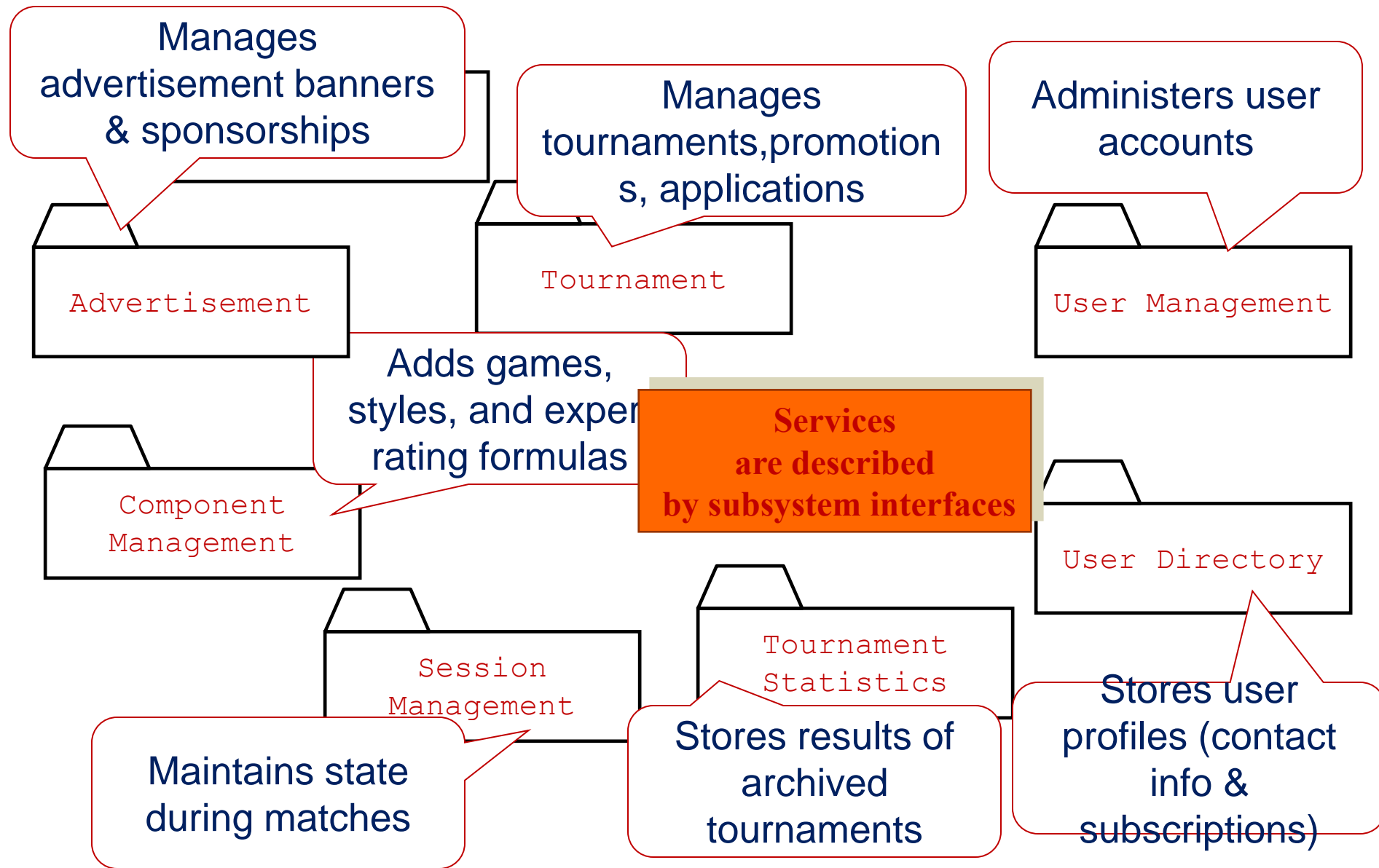
Example: Services provided by the ARENA Subsystems



Example: Services provided by the ARENA Subsystems



Example: Services provided by the ARENA Subsystems



Subsystem Interfaces vs. 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.*

Example: Notification subsystem

- **Service provided** by Notification Subsystem
 - LookupChannel()
 - SubscribeToChannel()
 - SendNotice()
 - UnscubscribeFromChannel()
- **Subsystem Interface** of Notification Subsystem
 - Set of fully typed UML operations
- **API** of Notification Subsystem
 - Implementation in Java

Subsystem Interface Object

- Good design: The subsystem interface object describes *all* the services of the subsystem interface
- Subsystem Interface Object
 - The set of public operations provided by a subsystem

Subsystem Interface Objects can be realized with the **Façade design pattern** (will be included in the lecture on design patterns).

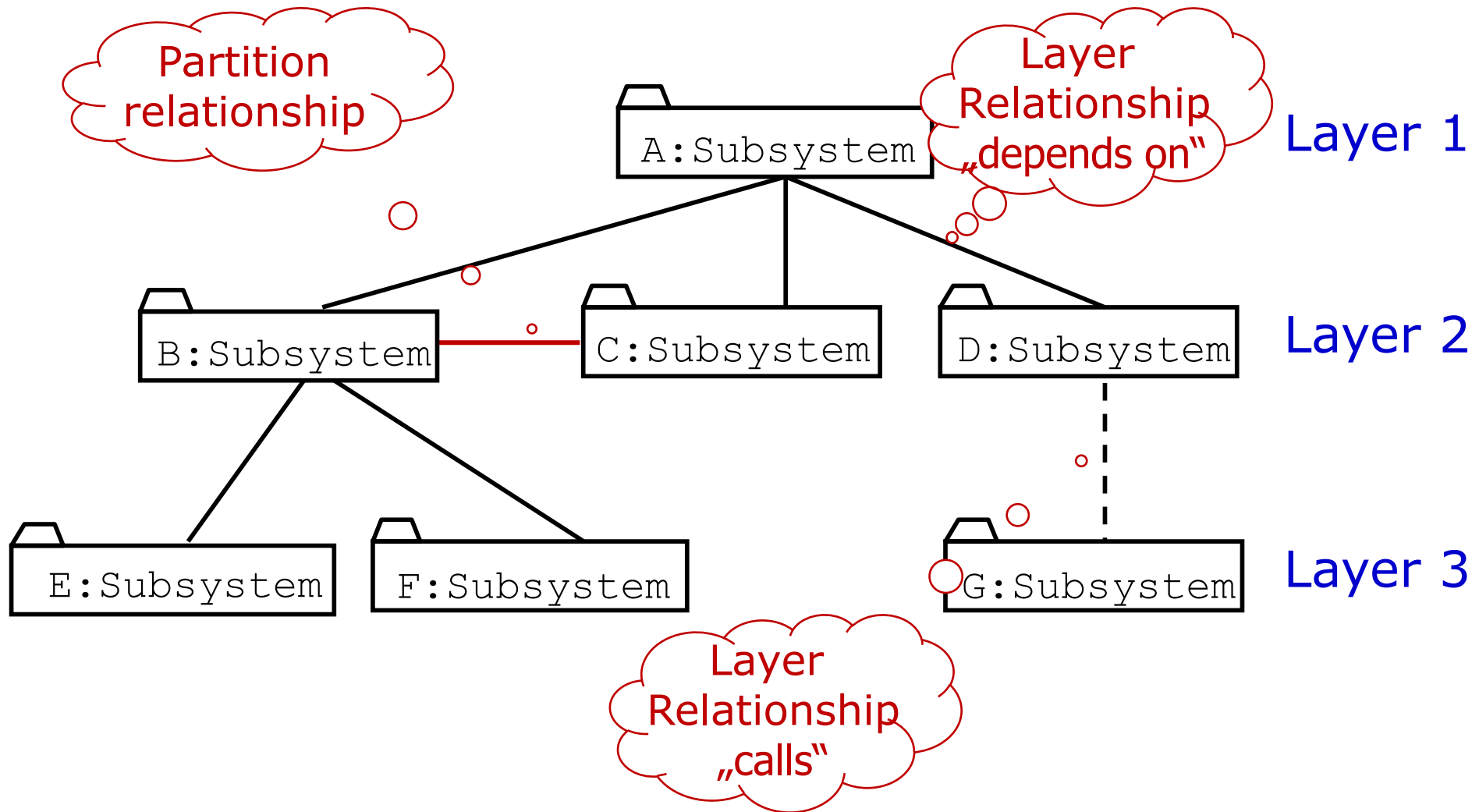
Properties of Subsystems: Layers and Partitions

- A **layer** is a subsystem that provides a service to another subsystem with the following restrictions:
 - A layer only depends on services from lower layers
 - A layer has no knowledge of higher layers
- A layer can be divided horizontally into several independent subsystems called **partitions**
 - Partitions provide services to other partitions on the same layer
 - Partitions are also called “weakly coupled” subsystems.

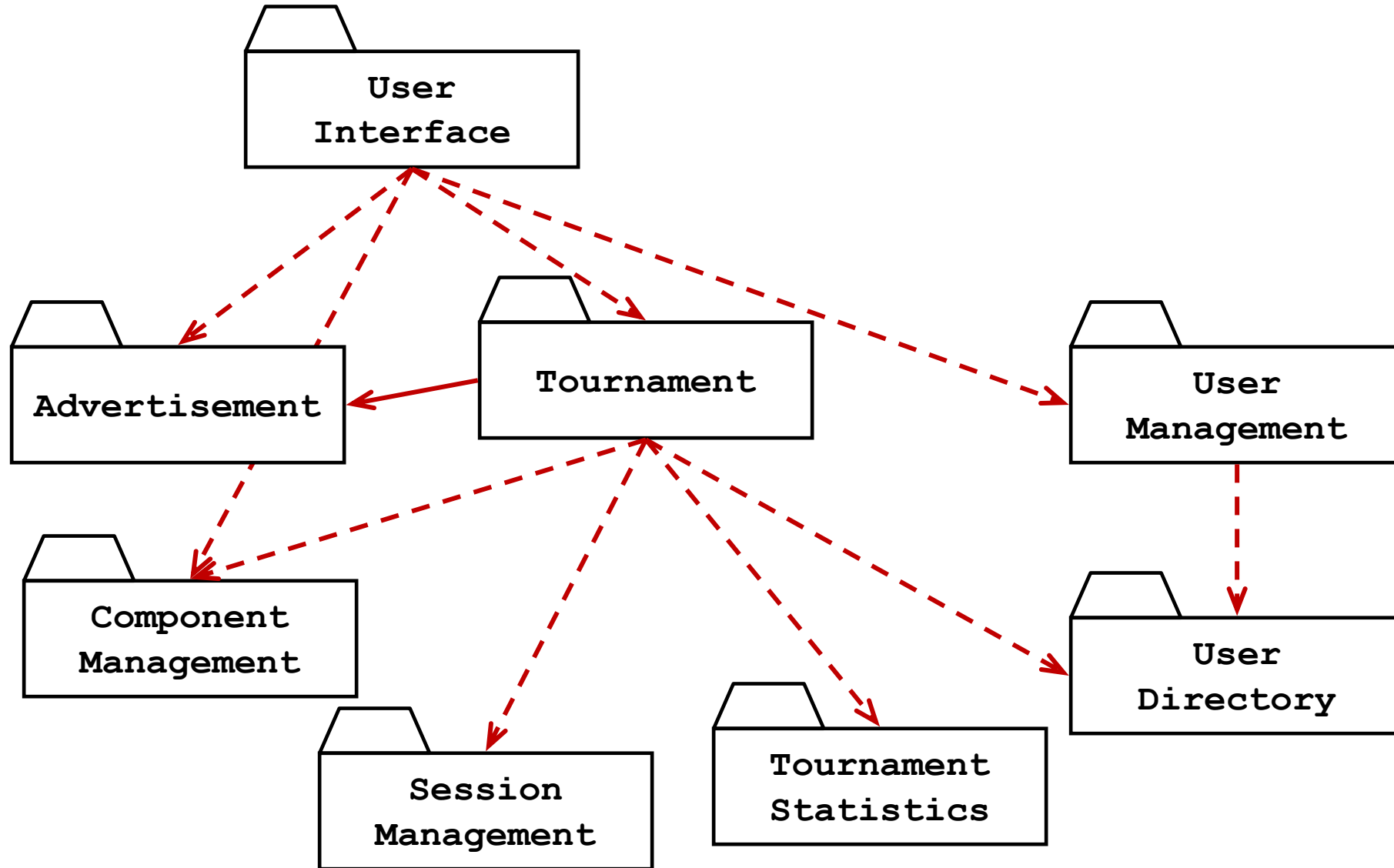
Relationships between Subsystems

- Two major types of Layer 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!
- Partition relationship
 - The subsystems have mutual knowledge about each other
 - A calls services in B; B calls services in A (Peer-to-Peer)
- UML convention:
 - Runtime dependencies are associations with dashed lines
 - Compile time dependencies are associations with solid lines.

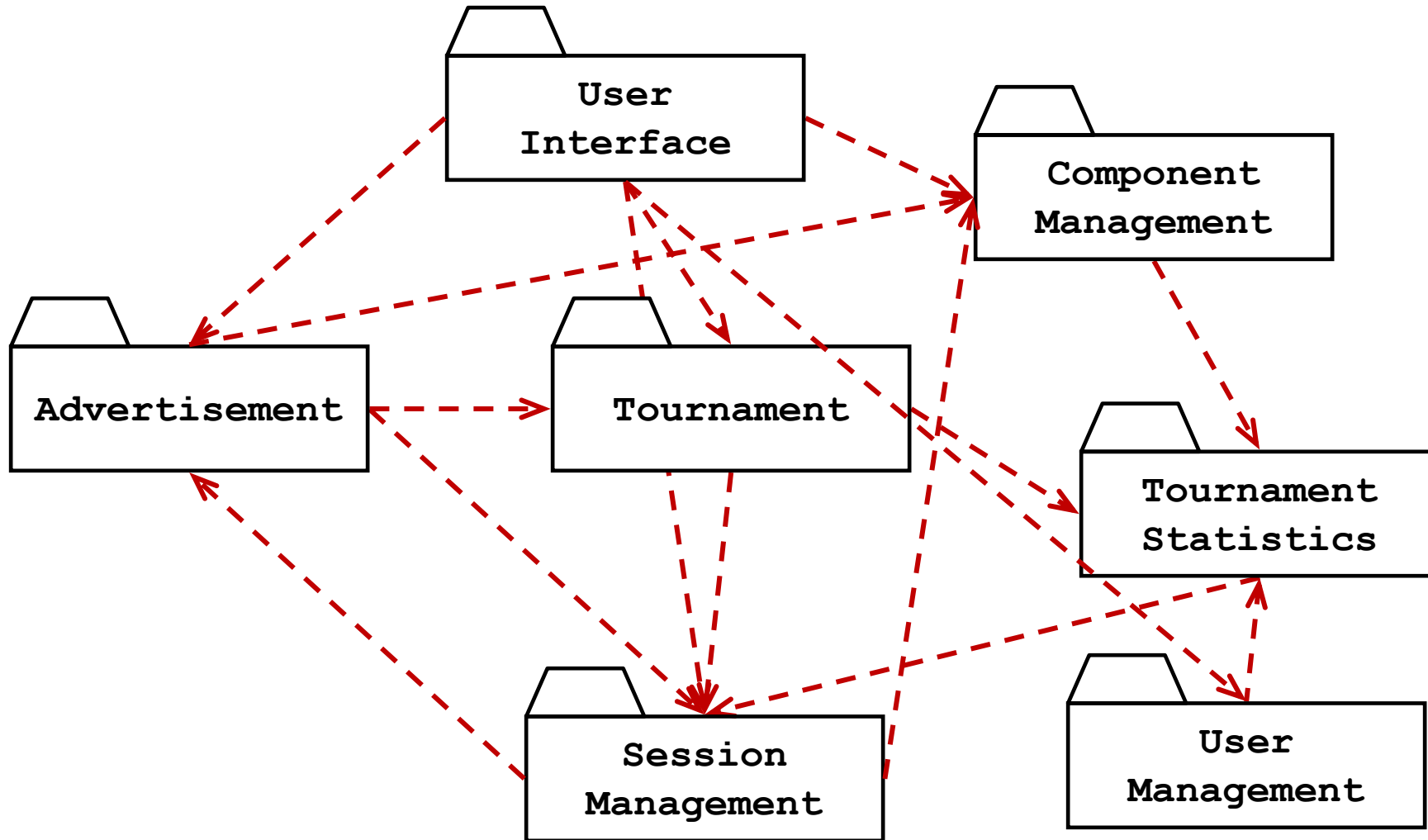
Example of a Subsystem Decomposition



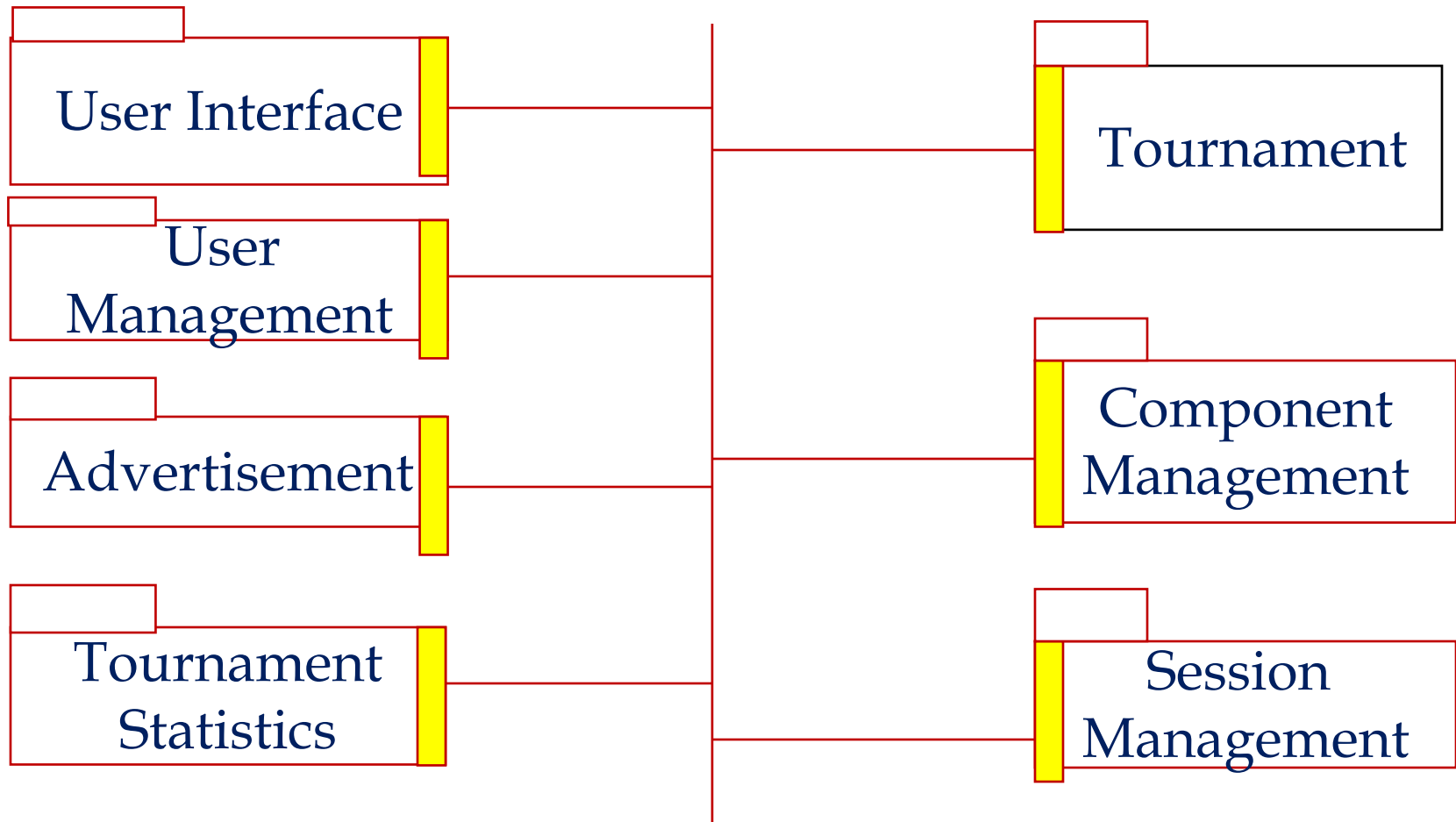
ARENA Subsystem Decomposition



Example of a Bad Subsystem Decomposition



Good Design: The System as set of Interface Objects



Subsystem Interface Objects



Clear subsystem interconnectivity

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 associations
 - **Low coherence**: Lots of miscellaneous and auxiliary classes, 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 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 associations
 - **Low coherence**: Lots of miscellaneous and auxiliary classes, 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

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

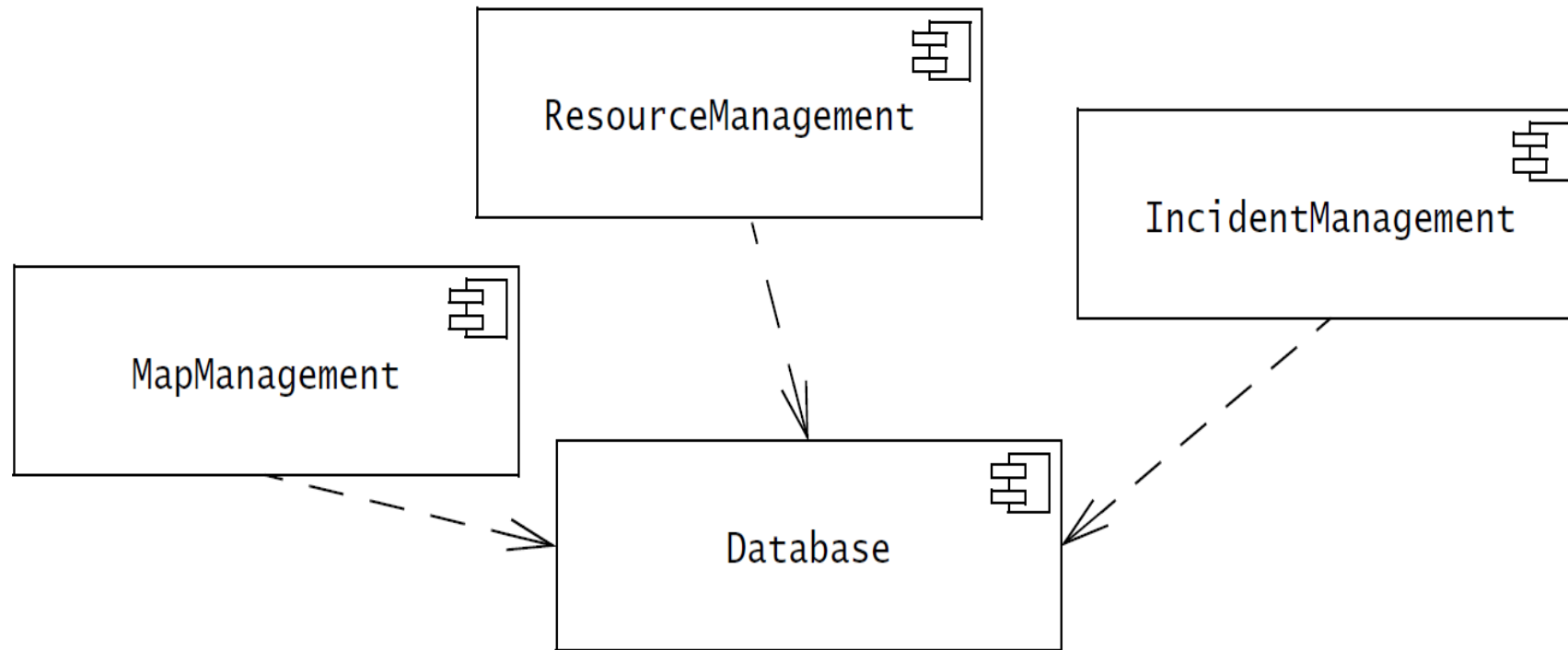
- Low coupling can be achieved if a calling class does not need to know anything about the internals of the called class (Principle of information hiding, Parnas)
- Questions to ask:
 - Does the calling class really have to know any attributes of classes in the lower layers?
 - Is it possible that the calling class calls only operations of the lower level classes?

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



Coupling: Example

Alternative 1: Direct access to the Database subsystem

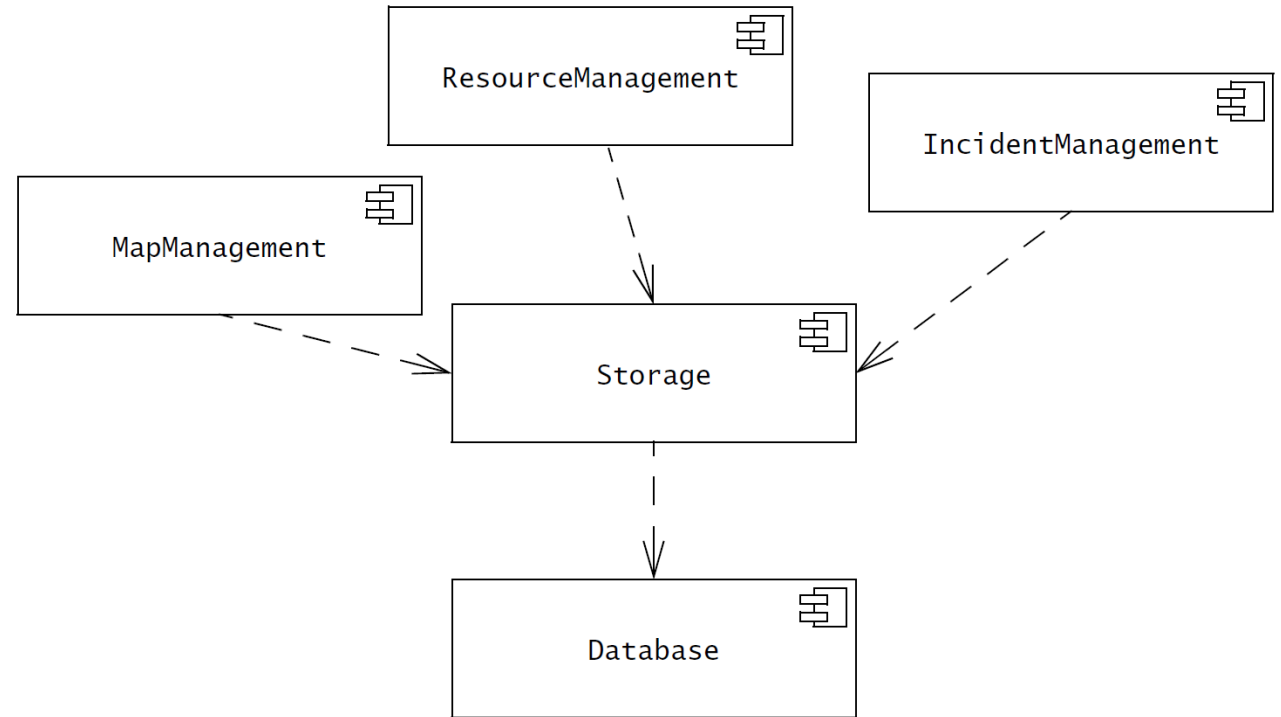


all subsystems access the database directly, making them vulnerable to changes in the interface of the Database subsystem.

Example of reducing the coupling of subsystems

Alternative 2: Indirect access to the Database through a Storage subsystem

Alternative 2 shields the database with an additional subsystem (Storage). In this situation, only one subsystem will need to change if there are changes in the interface of the Database subsystem. The assumption behind this design change is that the Storage subsystem has a more stable interface than the Database subsystem.



Low cohesion. Example:

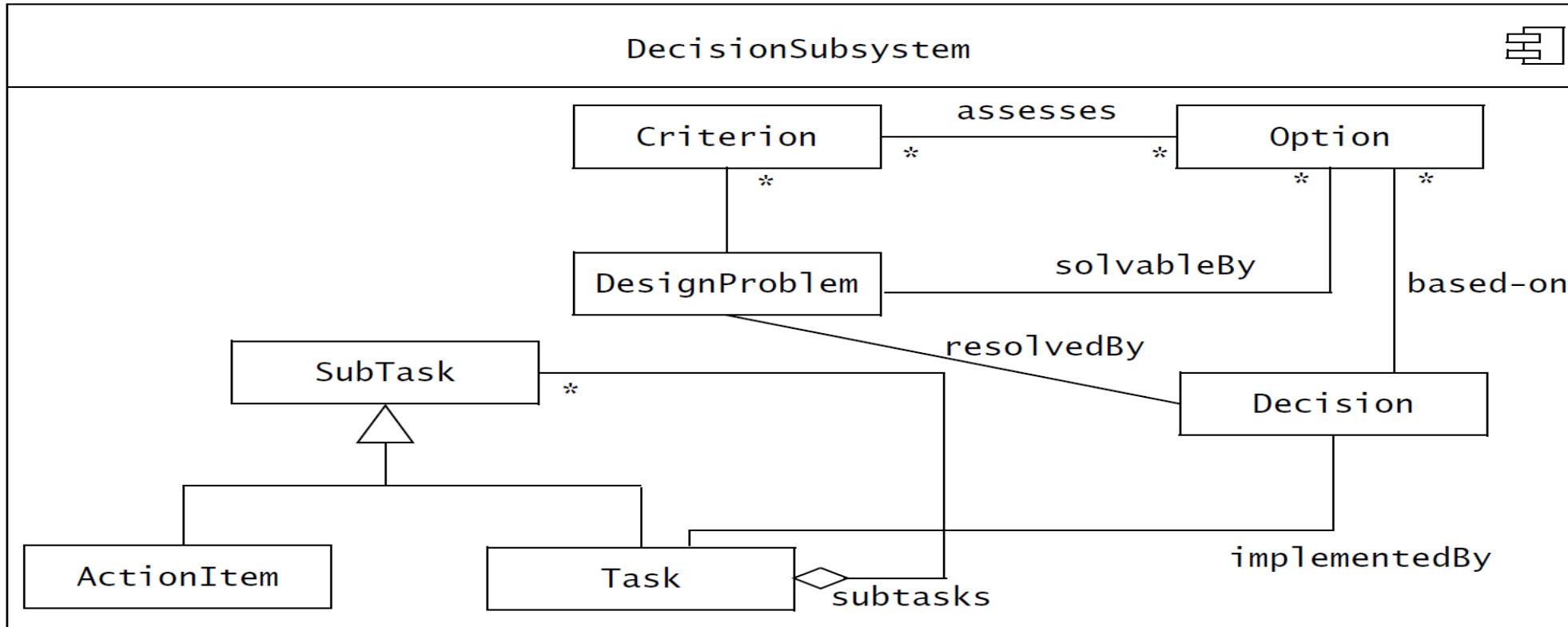
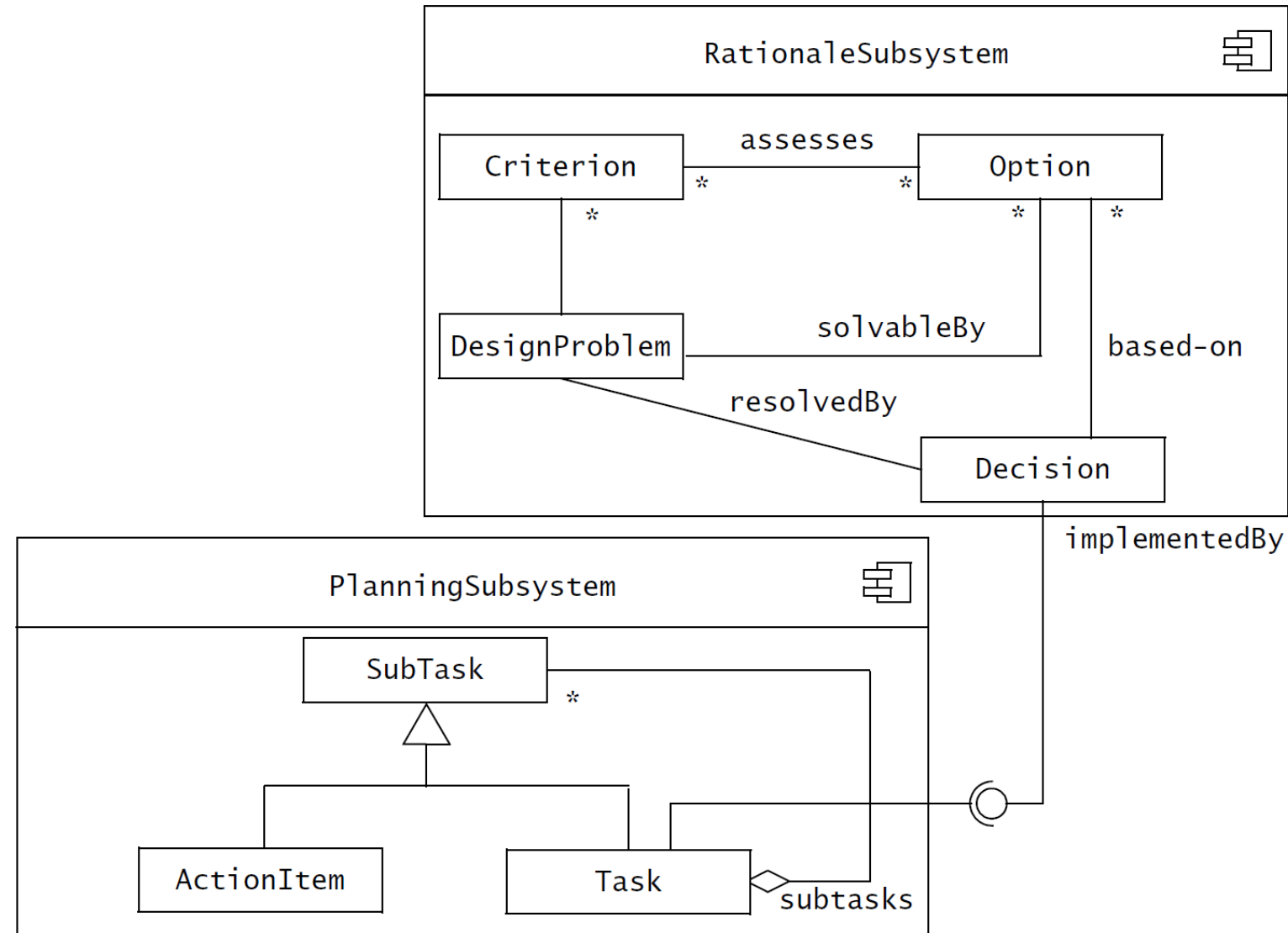


Figure 6-7 Decision tracking system (UML component diagram). The DecisionSubsystem has a low cohesion: The classes Criterion, Option, and DesignProblem have no relationships with Subtask, ActionItem, and Task.

Decompose into two subsystems



Architectural Style vs. Architecture

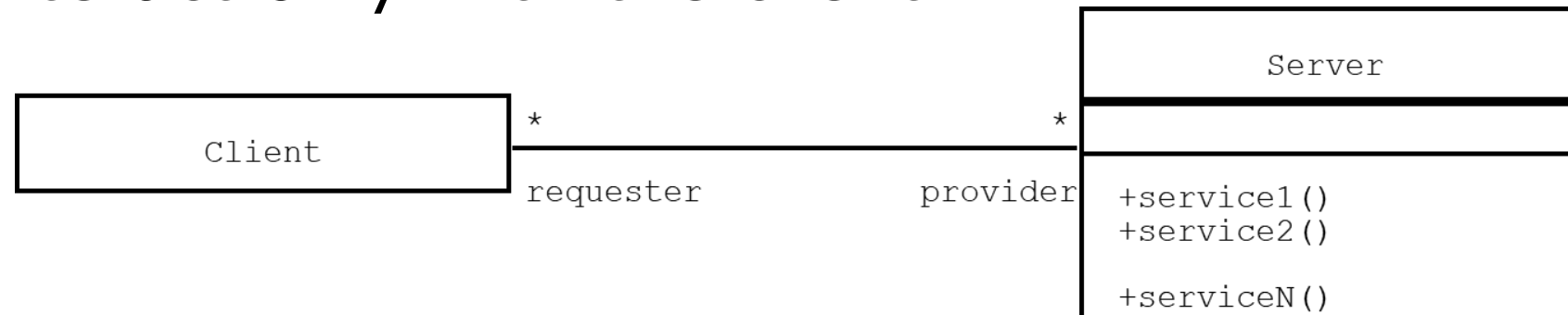
- **Subsystem decomposition:** Identification of subsystems, services, and their association to each other (hierarchical, peer-to-peer, etc)
- **Architectural Style:** A pattern for a subsystem decomposition
- **Software Architecture:** Instance of an architectural style

Examples of Architectural Styles

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

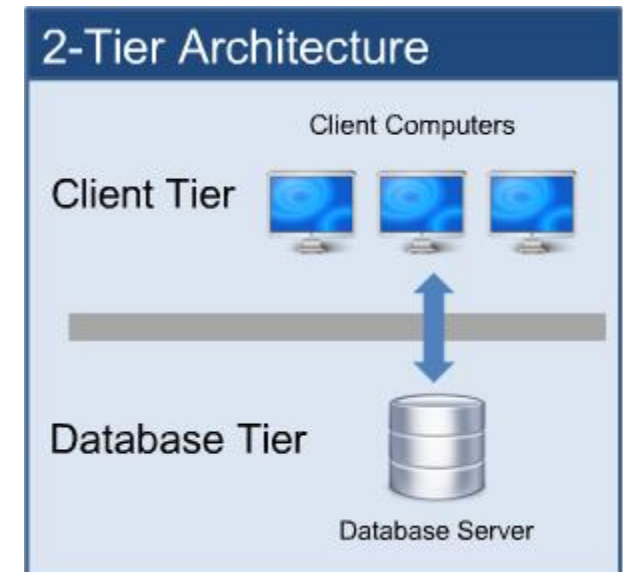
Client/Server Architectural Style

- 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.



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



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- or disk-intensive operations

Scalability

Server can handle large # of clients

Flexibility

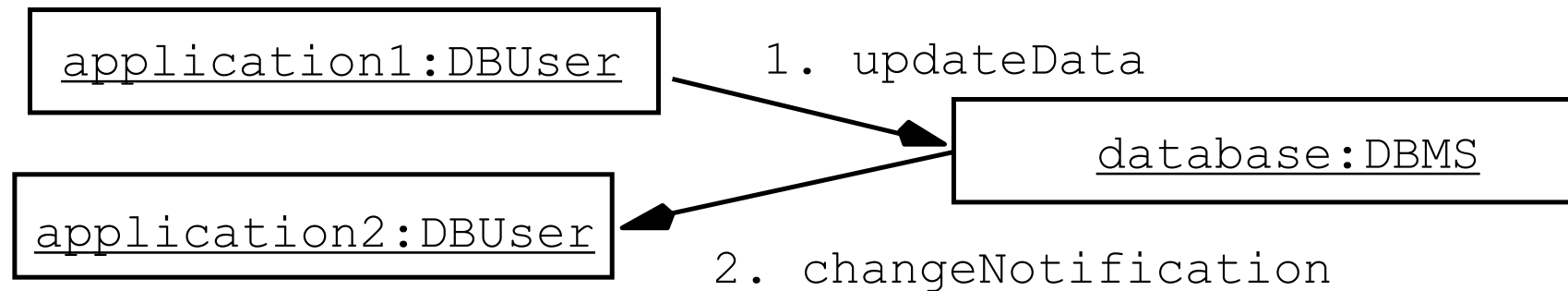
User interface of client supports a variety of end devices (PDA, Handy, laptop, wearable computer)

Reliability

A measure of success with which the observed behavior of a system confirms to the specification of its behavior (Chapter 11: Testing)

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 an 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"

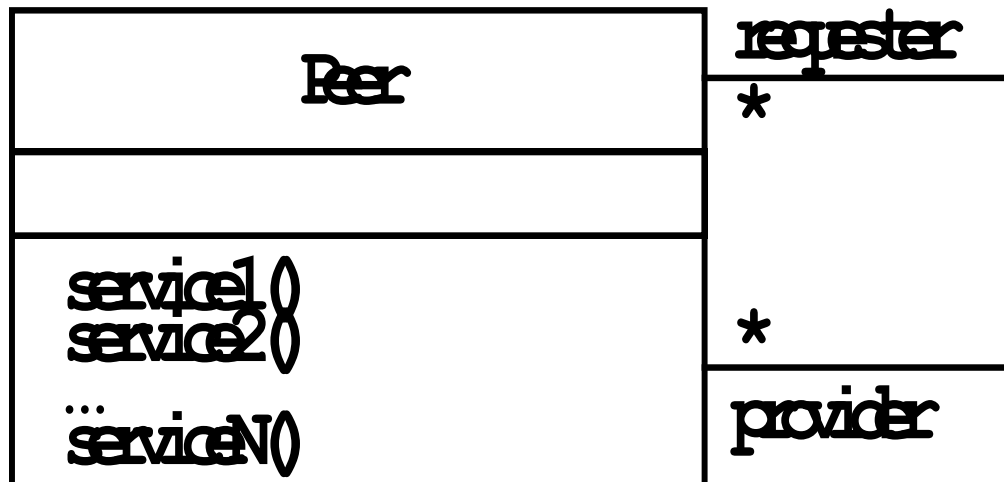
Introduction a new abstraction: Peer

"Clients and servers can be both peers"

How do we model this statement? With Inheritance?

Proposal 1: "A peer can be *either* a client or a server"

Proposal 2: "A peer can be *both* a client and a server"



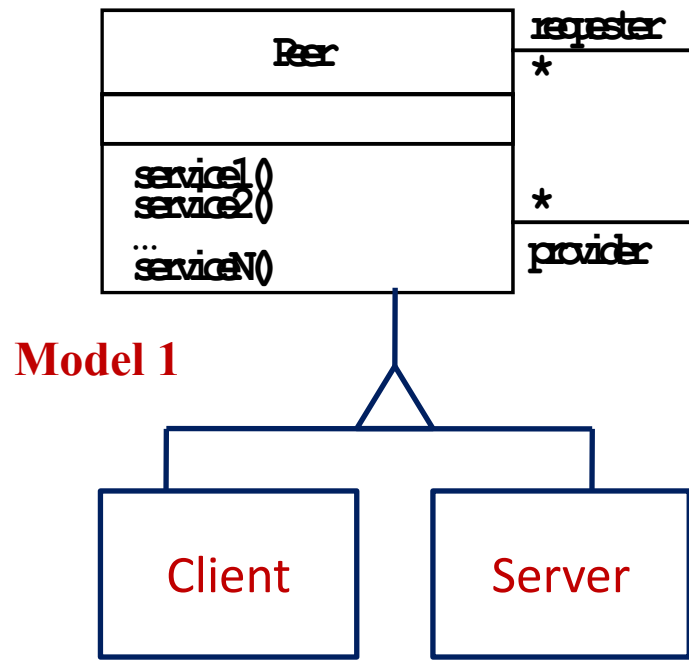
Relationship Client/Server & Peer-to-Peer

Problem statement "Clients can be servers and servers can be clients"

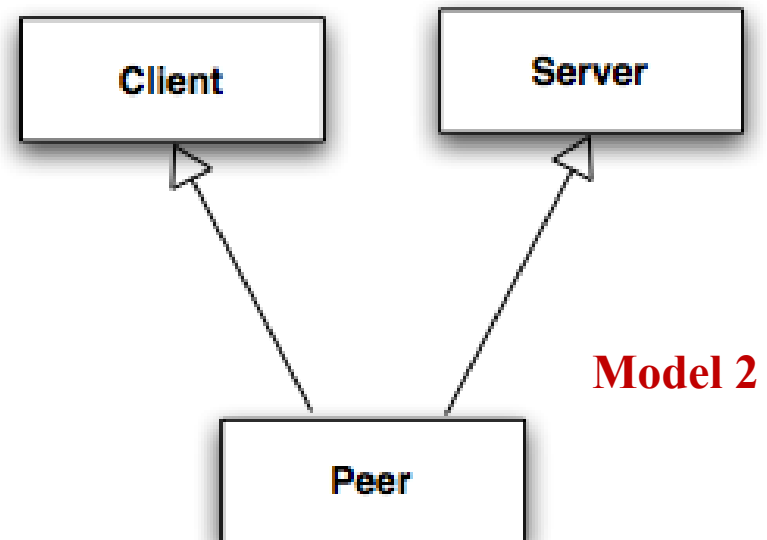
Which model is correct?

Model 1: "A peer can be *either* a client or a server"

Model 2: "A peer can be *both* a client and a server"



?



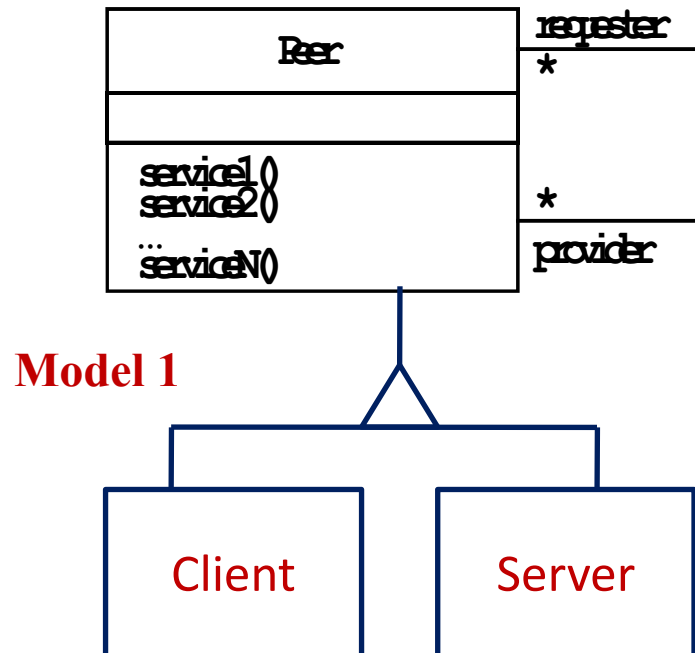
Relationship Client/Server & Peer-to-Peer

Problem statement "Clients can be servers and servers can be clients"

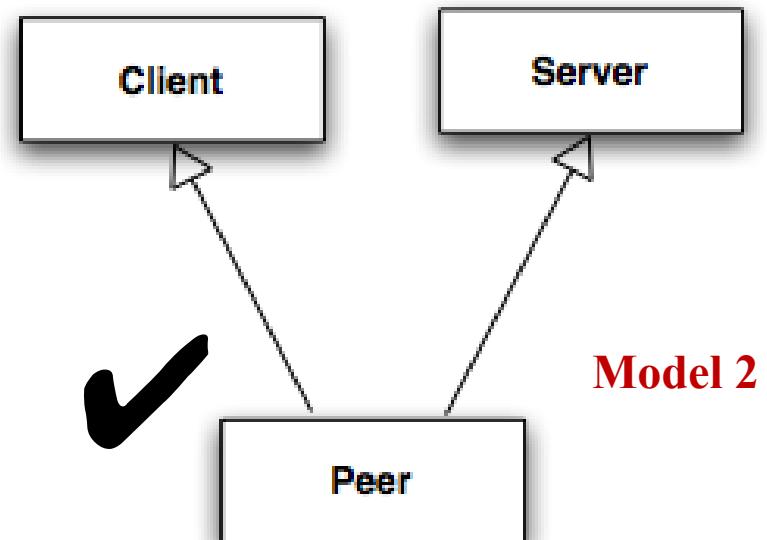
Which model is correct?

Model 1: "A peer can be *either* a client or a server"

Model 2: "A peer can be *both* a client and a server"



?



Virtual Machine

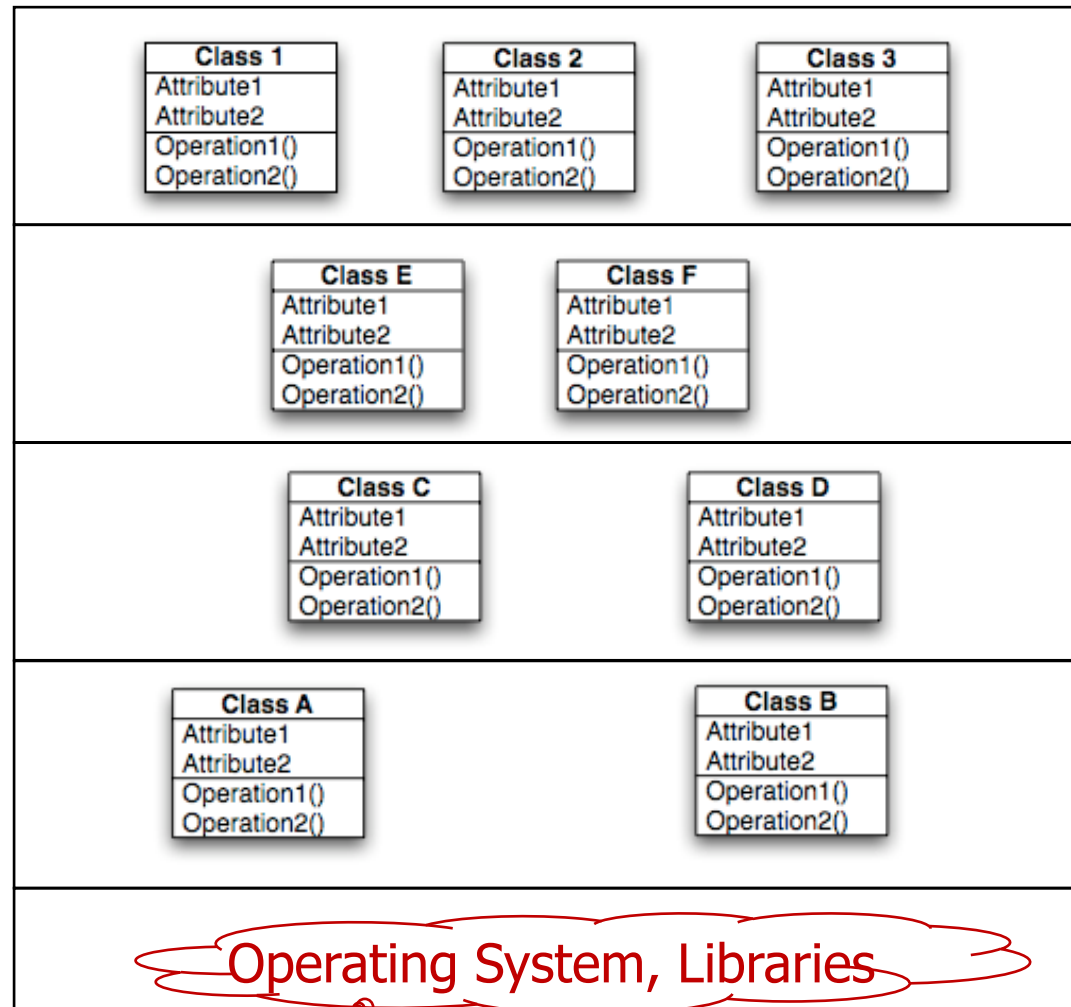
- A **virtual machine** is a subsystem connected to higher and lower level virtual machines by "provides services for" associations
- A virtual machine is an abstraction that provides a set of attributes and operations
- The terms layer and virtual machine can be used interchangeably
 - Also sometimes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

Virtual Machine: A Bit of History

- Virtual Machine (VM) is an emulation of an existing or hypothetical computer hardware system.
- A virtual machine can emulate the entire existing computer system.
- Examples:
 - CP/CMS (Control Program/Cambridge Monitor System), IBM, late 60's – native emulation of the whole system
 - VM/CMS, IBM, 1972 for IBM System/360; latest release is zVM, 2013, for the zEnterprise System emulation
 - VMware ESXi, VMware, Inc., emulation of the x86-64 system
 - Many other systems of this type exist
 - Recently, Cloud Computing

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.



Virtual Machine 4 .

Virtual Machine 3

Virtual Machine 2

Virtual Machine 1

Existing System

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.



Existing System

Building Systems as a Set of Virtual Machines

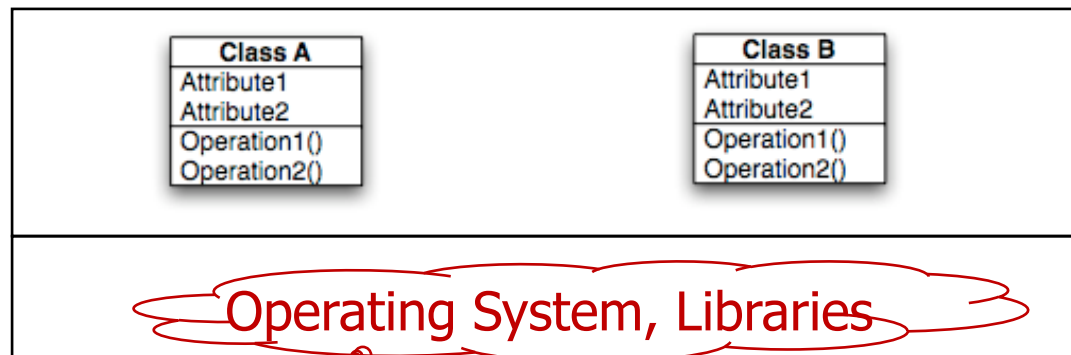
A system is a hierarchy of virtual machines, each using language primitives offered by the lower machines.



Existing System

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.

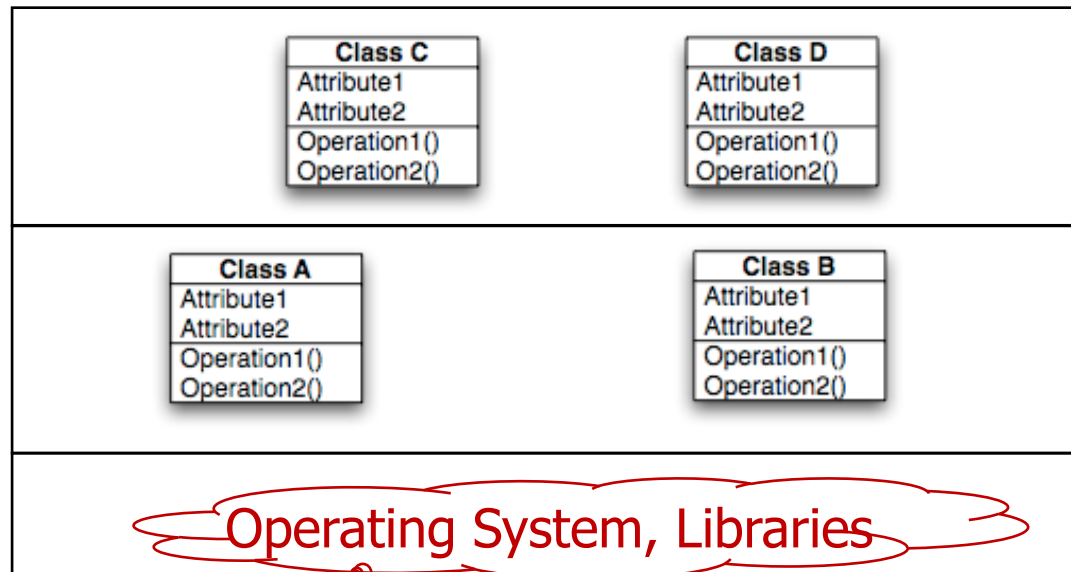


Virtual Machine 1

Existing System

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.



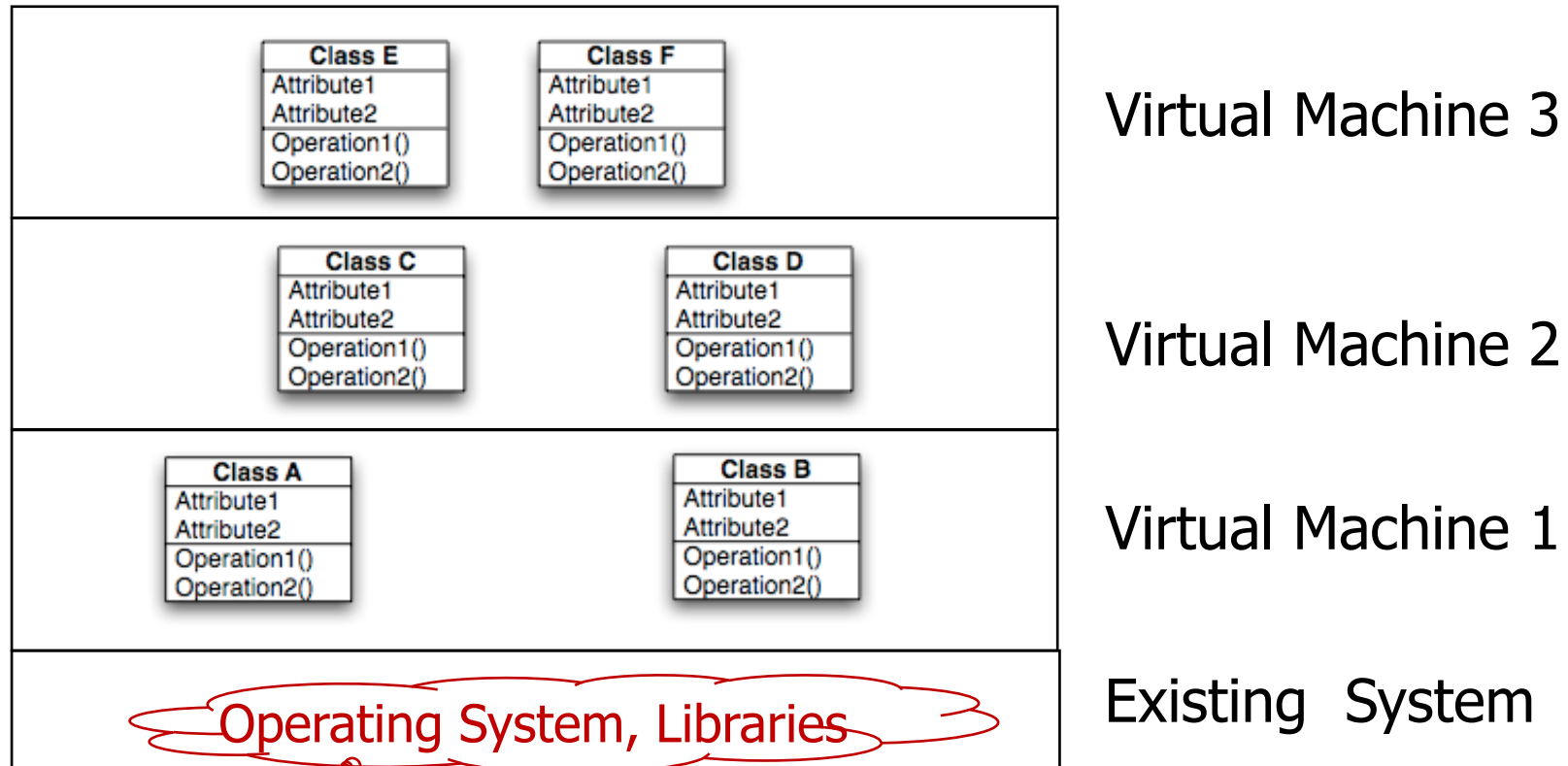
Virtual Machine 2

Virtual Machine 1

Existing System

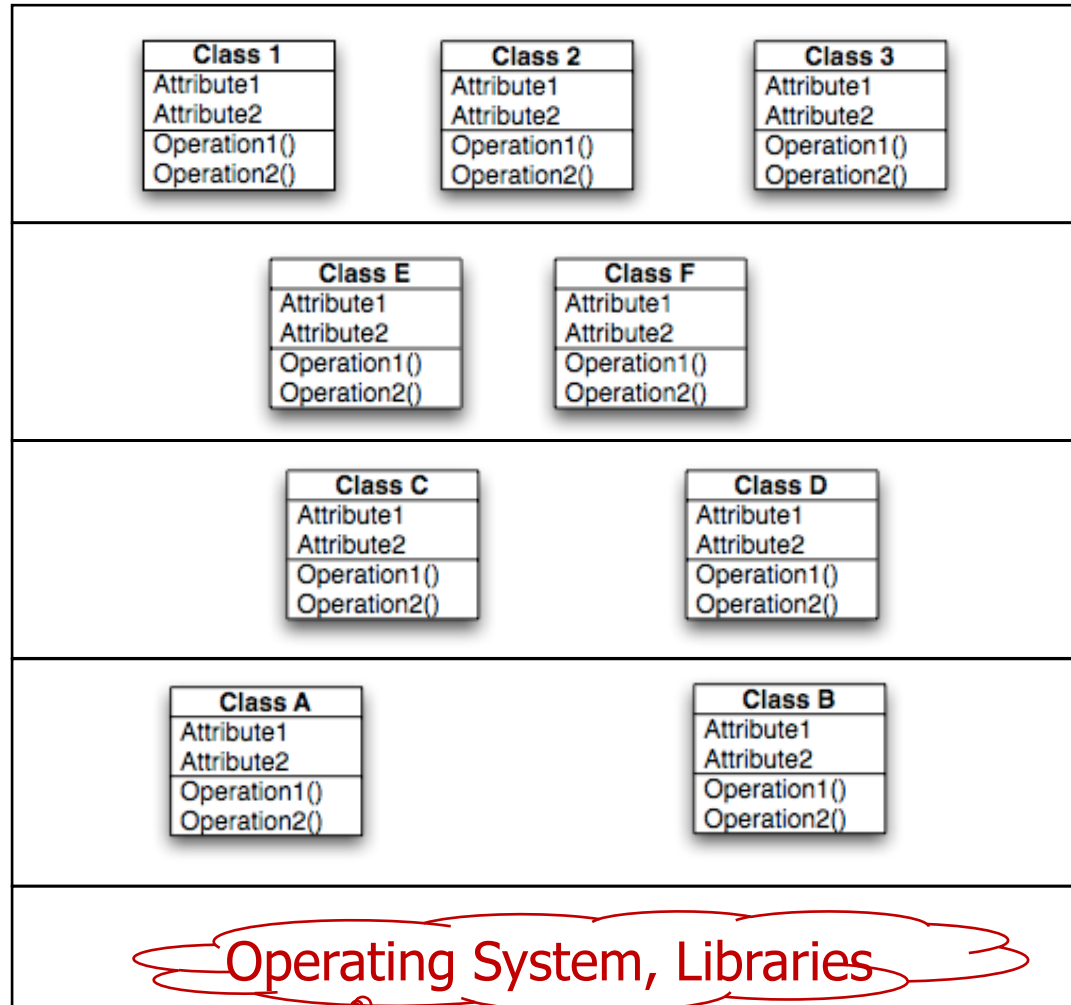
Building Systems as a Set of Virtual Machines

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



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.



Virtual Machine 4 .

Virtual Machine 3

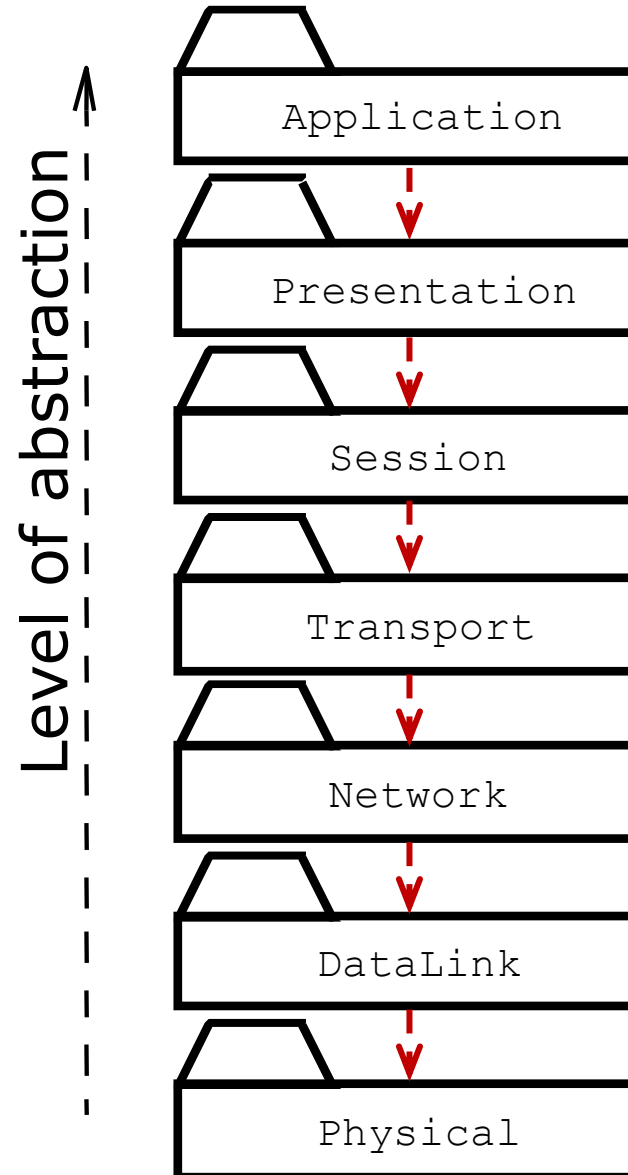
Virtual Machine 2

Virtual Machine 1

Existing System

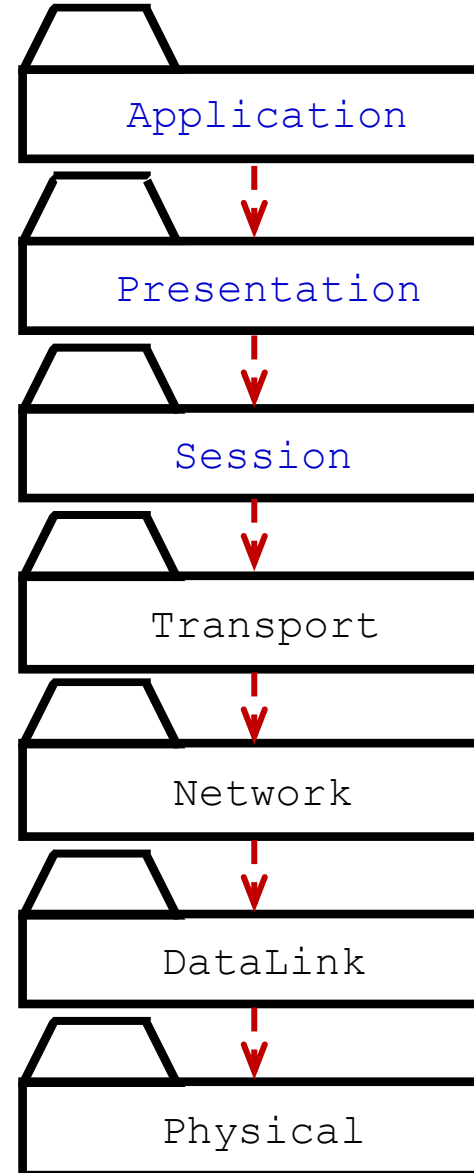
Example: Layered Architectural Style

- 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



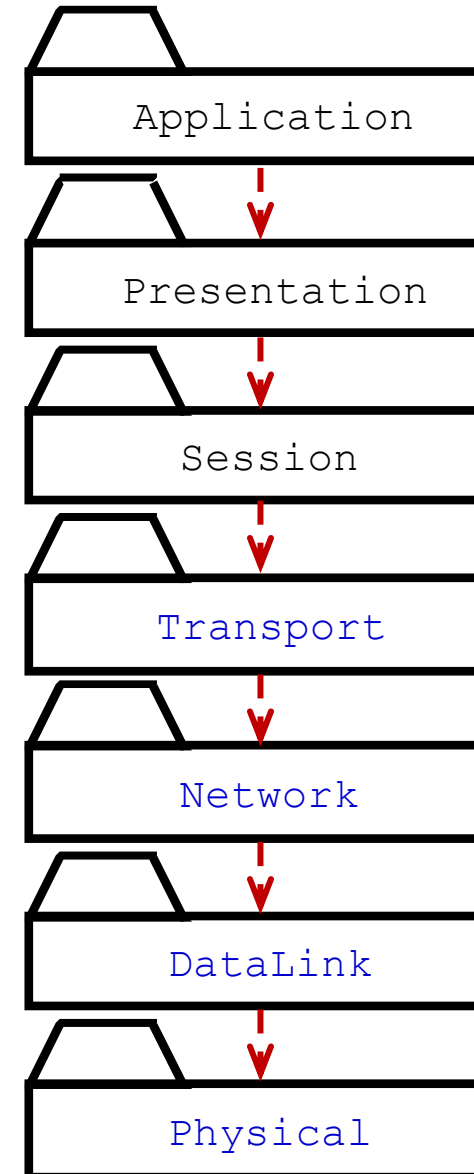
OSI Model Layers and Services

- The **Application layer** is the system you are building (unless you build a protocol stack)
- ! • The application layer is usually layered itself
- The **Presentation layer** performs data transformation services, such as byte swapping and encryption
- The **Session layer** is responsible for initializing a connection, including authentication

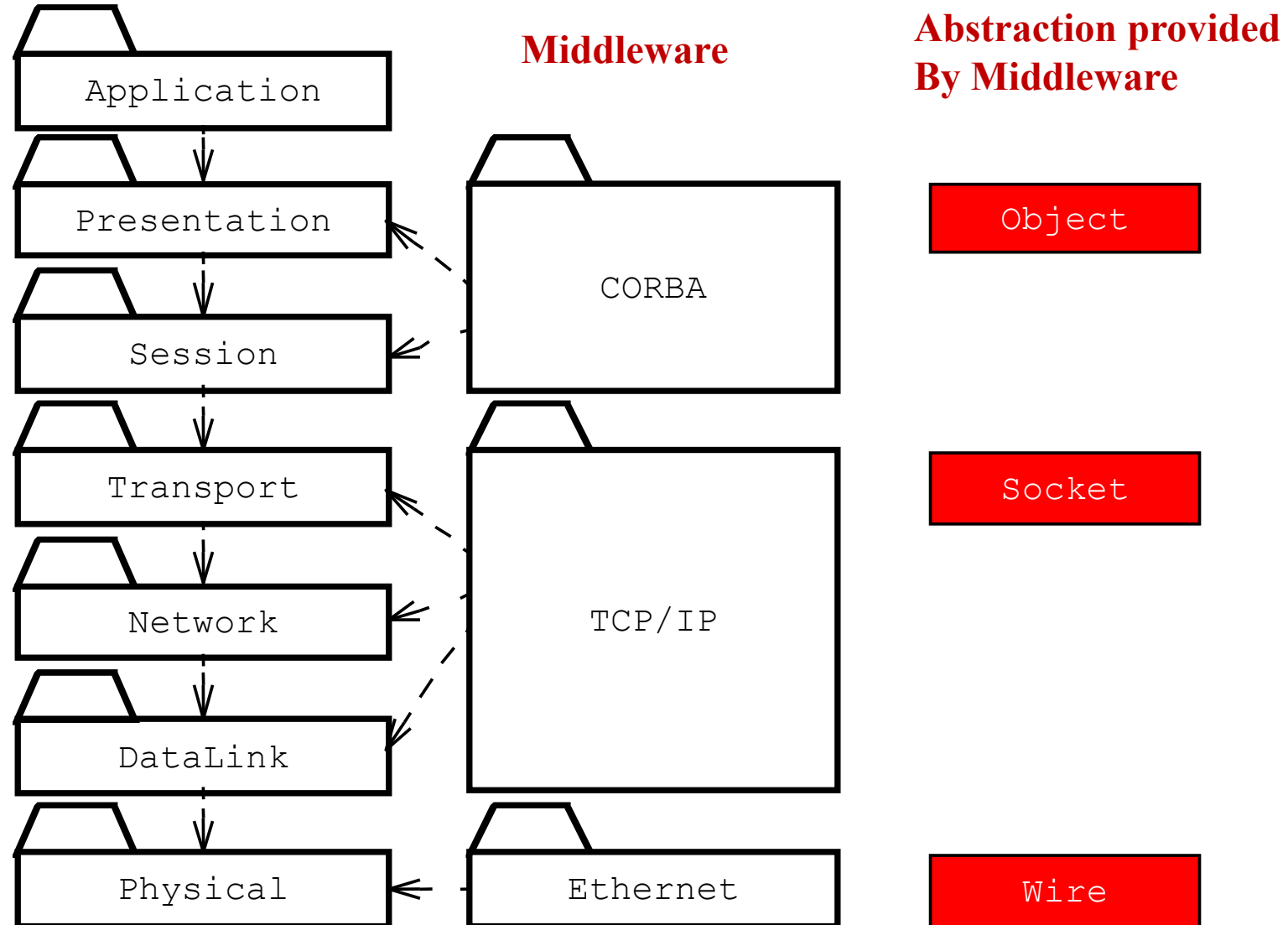


OSI Model Layers and their Services

- The **Transport layer** is responsible for reliably transmitting messages
 - Used by Unix programmers who transmit messages over TCP/IP sockets
- The **Network layer** ensures transmission and routing
 - Services: Transmit and route data within the network
- The **Datalink layer** models frames
 - Services: Transmit frames without error
- The **Physical layer** represents the hardware interface to the network
 - Services: sendBit() and receiveBit()



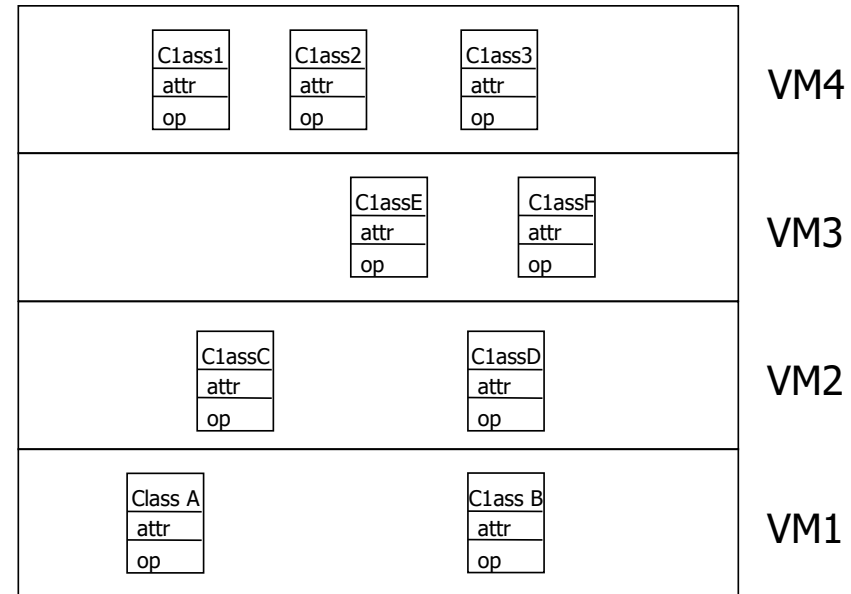
Middleware Allows Focus On Higher Layers



Closed Architecture (Opaque Layering)

- Each virtual machine can only call operations from the layer below

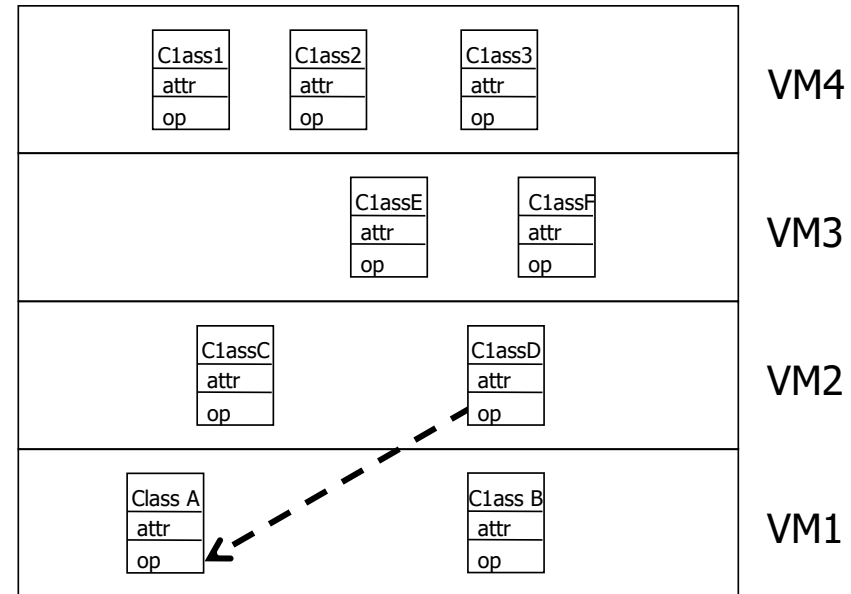
Design goals:
Maintainability,
flexibility
Security.



Closed Architecture (Opaque Layering)

- Each virtual machine can only call operations from the layer below

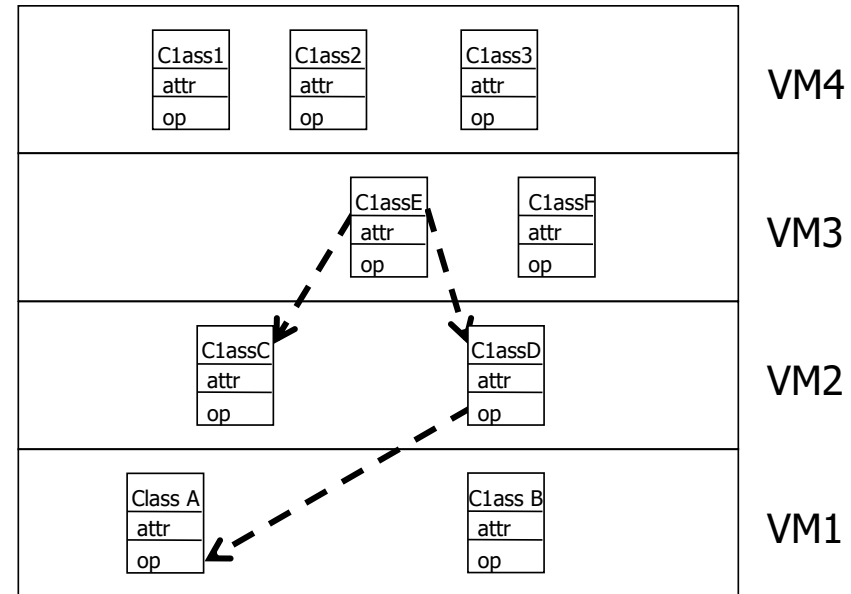
Design goals:
Maintainability,
flexibility
Security.



Closed Architecture (Opaque Layering)

- Each virtual machine can only call operations from the layer below

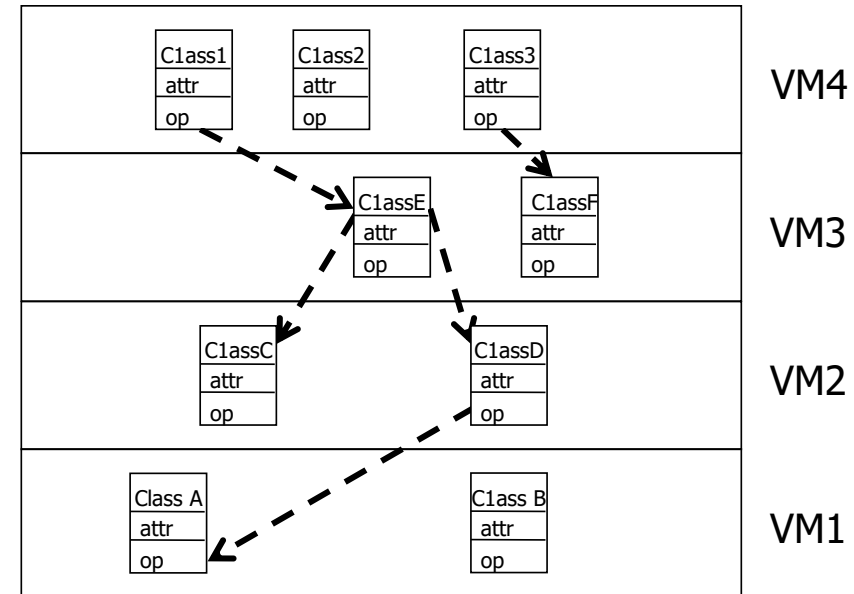
Design goals:
Maintainability,
flexibility
Security.



Closed Architecture (Opaque Layering)

- Each virtual machine can only call operations from the layer below

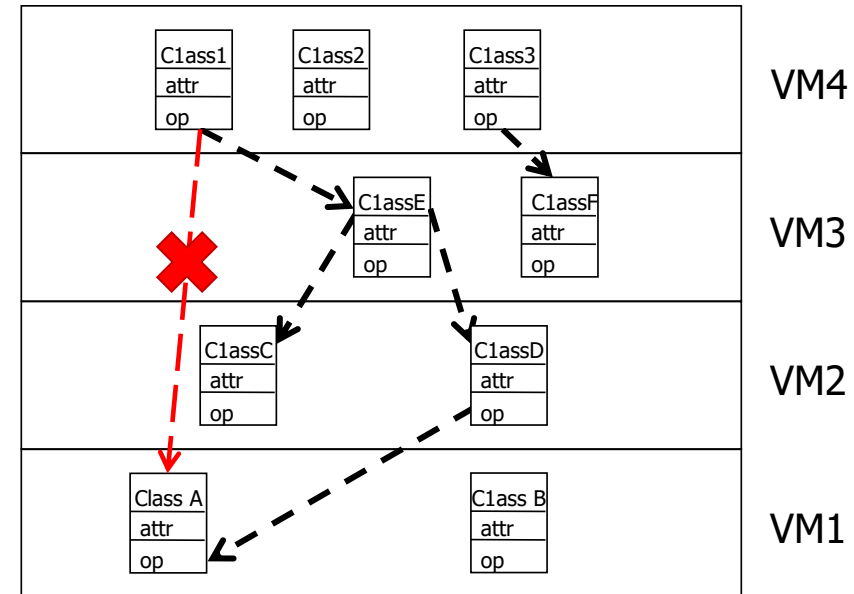
Design goals:
Maintainability,
flexibility
Security.



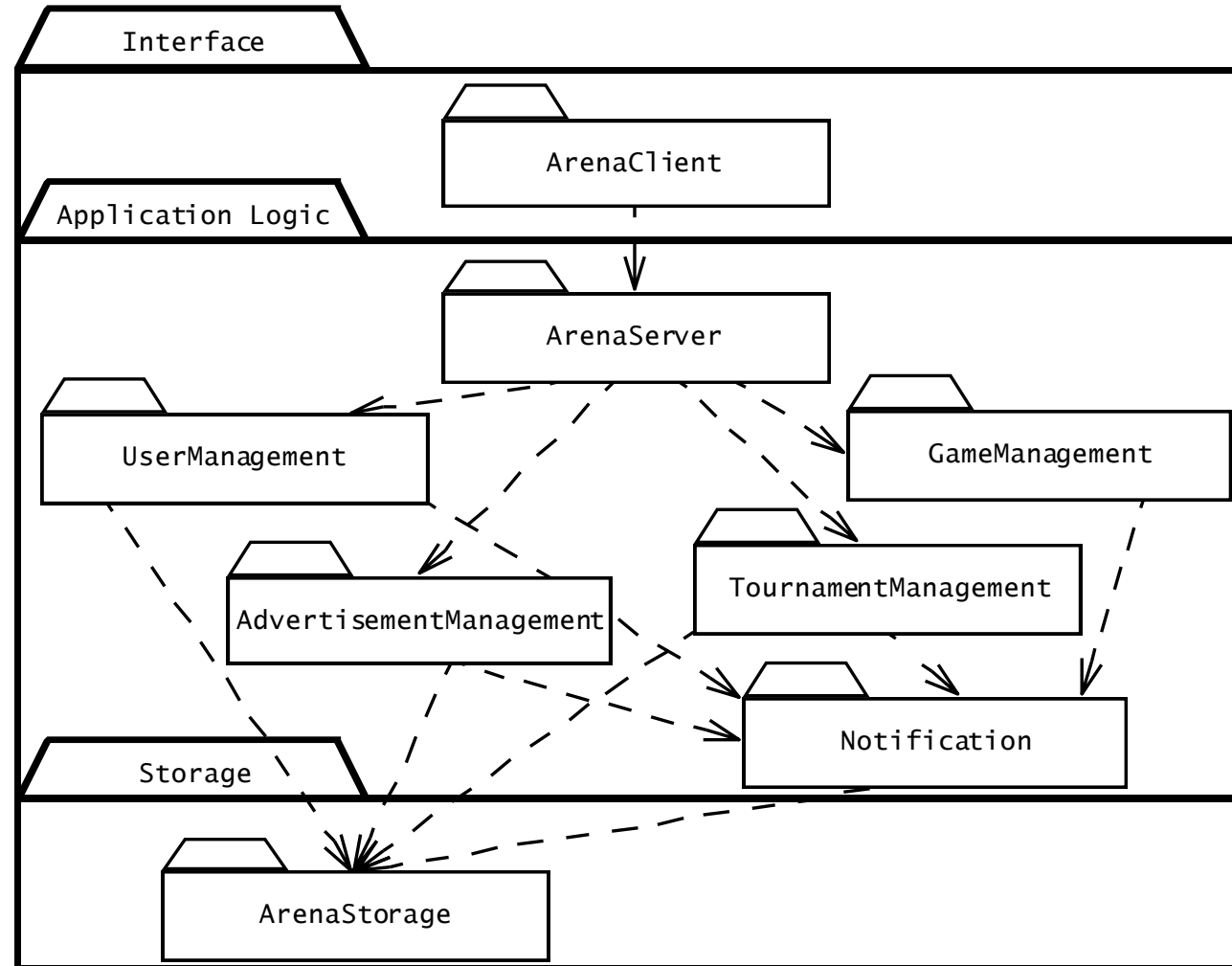
Closed Architecture (Opaque Layering)

- Each virtual machine can only call operations from the layer below

Design goals:
Maintainability,
flexibility
Security.



Opaque Layering in ARENA

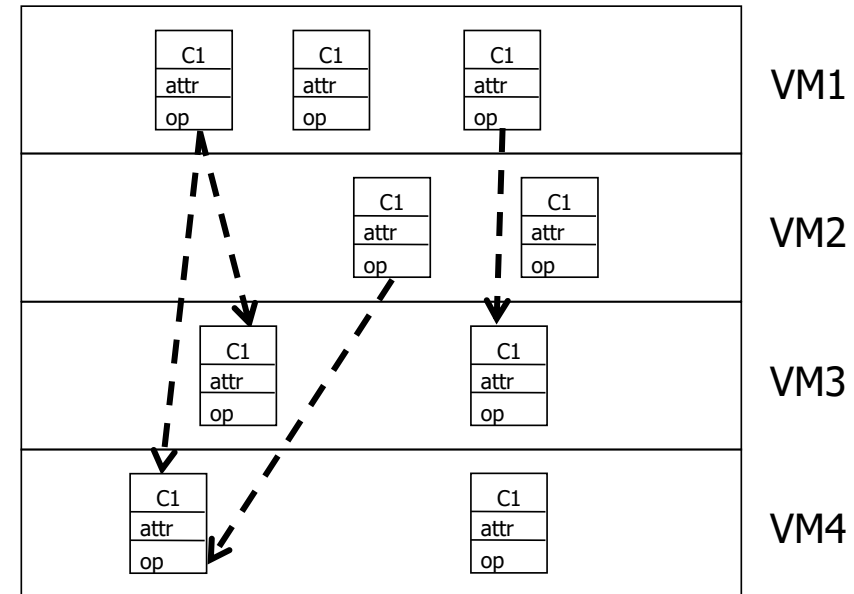


Open Architecture (Transparent Layering)

- Each virtual machine can call operations from any layer below

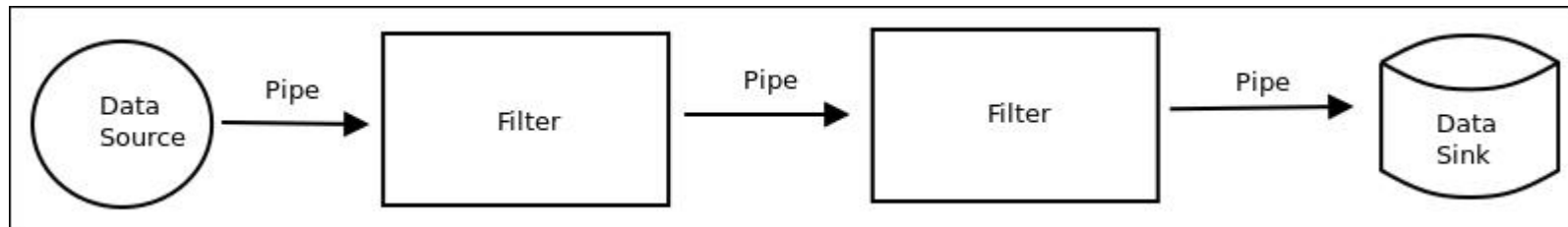
Design goal:

Runtime efficiency



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

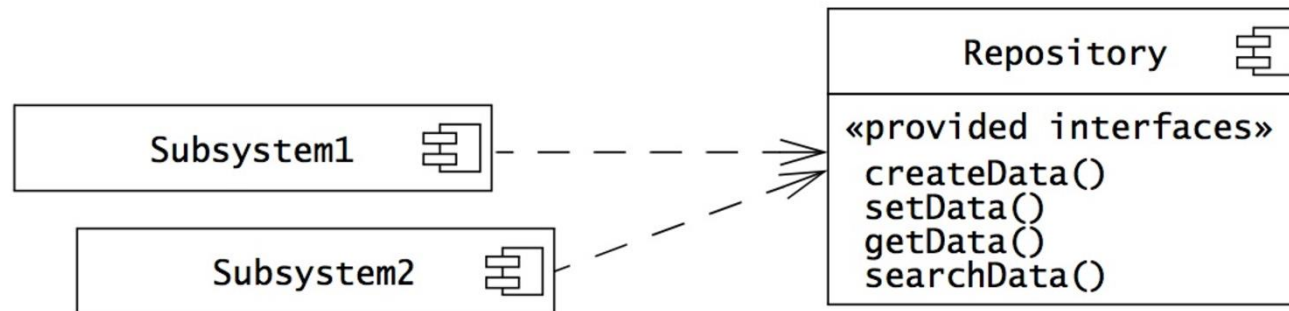
- An architectural style that consists of two subsystems called pipes and filters
 - **Filter**: A subsystem that does a processing step
 - **Pipe**: A Pipe is a connection between two processing steps
- Each filter has an input pipe and an output pipe.
 - The data from the input pipe are processed by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

A pipe

The Unix shell commands ls and cat are Filter

Repository Architectural Style

- Subsystems access and modify data from a single data structure called the **repository**
 - Subsystems are loosely coupled (interact only through the repository)
 - Control flow is dictated by the repository through triggers or by the subsystems through locks and synchronization primitives

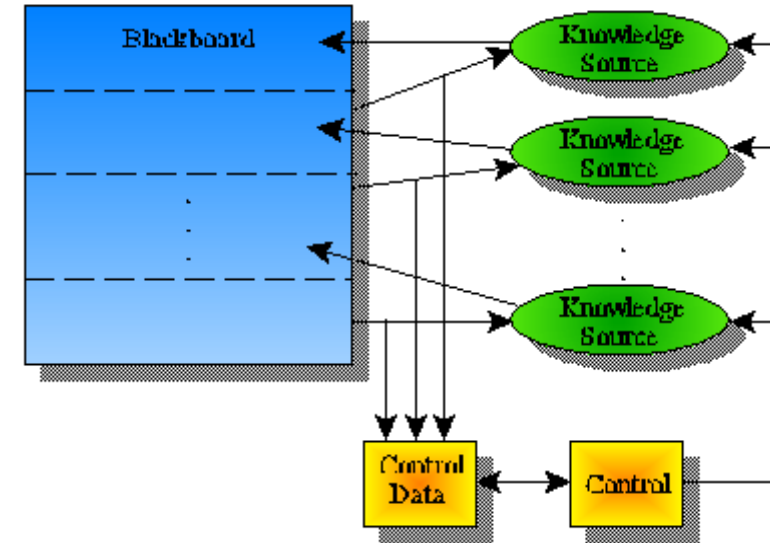


Repository Architectural Style

- The repository only ensures that concurrent accesses are serialized. Conversely, the repository can be used to invoke the subsystems based on the state of the central data structure.
- These systems are called “blackboard systems.” (Erman, Hayes-Roth and Reddy 1980)

Blackboard Subsystem Decomposition

- A blackboard-system consists of three major components
 - The **blackboard**. A shared repository of problems, partial solutions and new information.
 - The **knowledge sources** (KSs). Each knowledge source embodies specific expertise. It reads the information placed on the blackboard and places new information on the blackboard.
 - The **control shell**. It controls the flow of problem-solving activity in the system, in particular how the knowledge sources get notified of new information put into the blackboard.

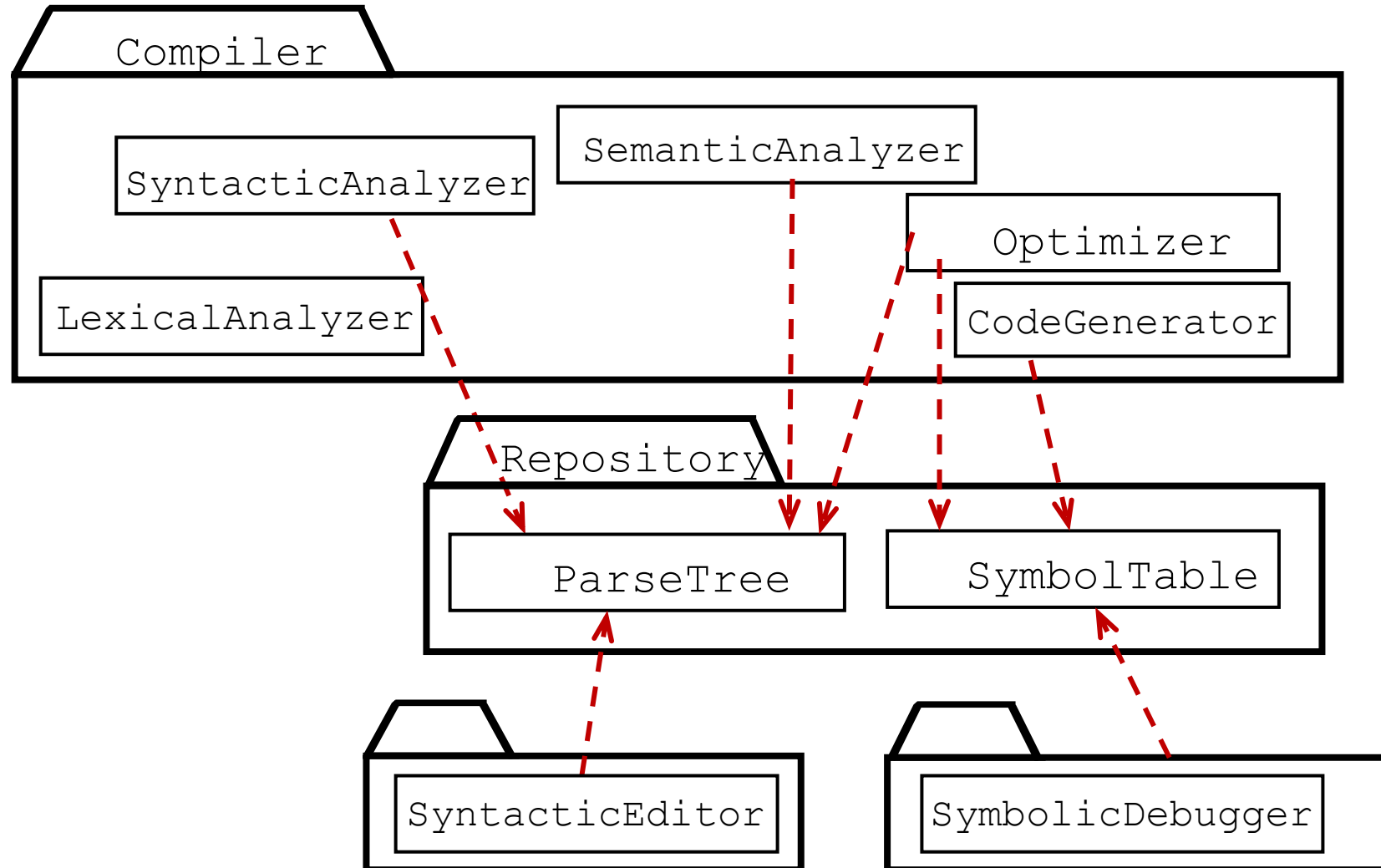


Raj Reddy, b. 1937, AI pioneer

- Major contributions to
speech, vision, robotics, e.g.,
Hearsay and Harpy

- Founding Director of
Robotics Institute, HCII,
Center for Machine Learning, etc
1994: Turing Award (with Ed
Feigenbaum).

Repository Architecture Example: Integrated Development Environment (IDE)



Repository Systems Pros and Cons

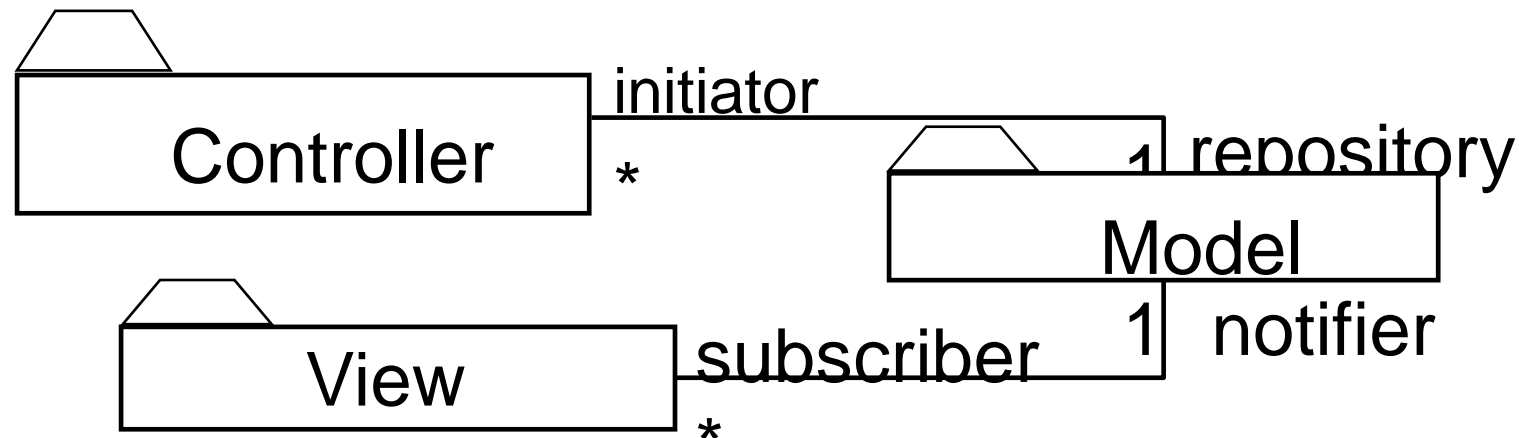
- Repositories are well suited for applications with constantly changing, complex data- processing tasks. Once a central repository is well defined, we can easily add new services in the form of additional subsystems.
- The main **disadvantage** of repository systems is that the central repository can 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.

Providing Consistent Views

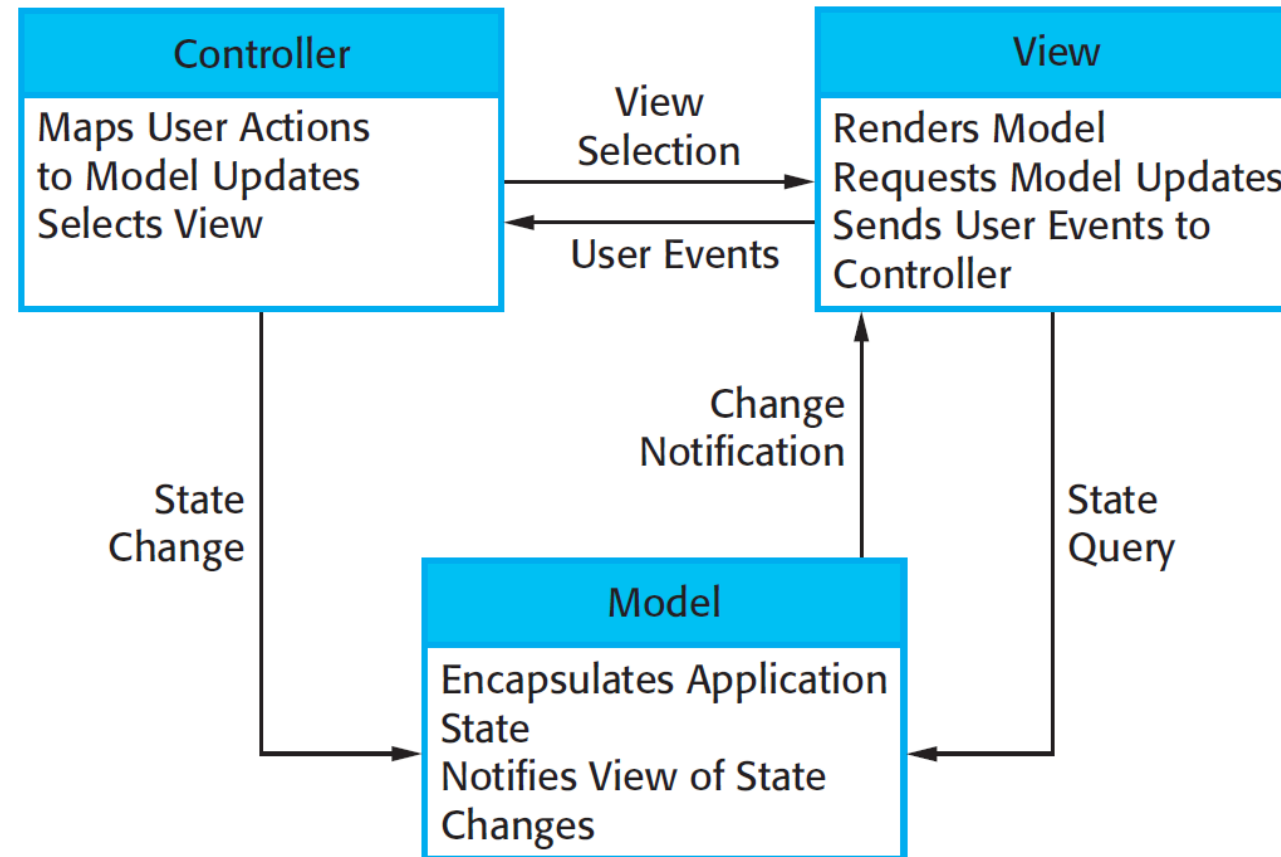
- **Problem:** In systems with high coupling changes to the user interface (boundary objects) often force changes to the entity objects (data model)
 - The user interface cannot be re-implemented without changing the representation of the entity objects
 - The entity objects cannot be reorganized without changing the user interface
- **Solution: Decoupling!** The model-view-controller architectural style decouples data access (entity objects) and data presentation (boundary objects)
 - The Data Presentation subsystem is called the **View**
 - The Data Access subsystem is called the **Model**
 - The Logic subsystem is called the **Controller**, which mediates between View (data presentation) and Model (data access)
- Often called **MVC**

Model-View-Controller Architectural Style

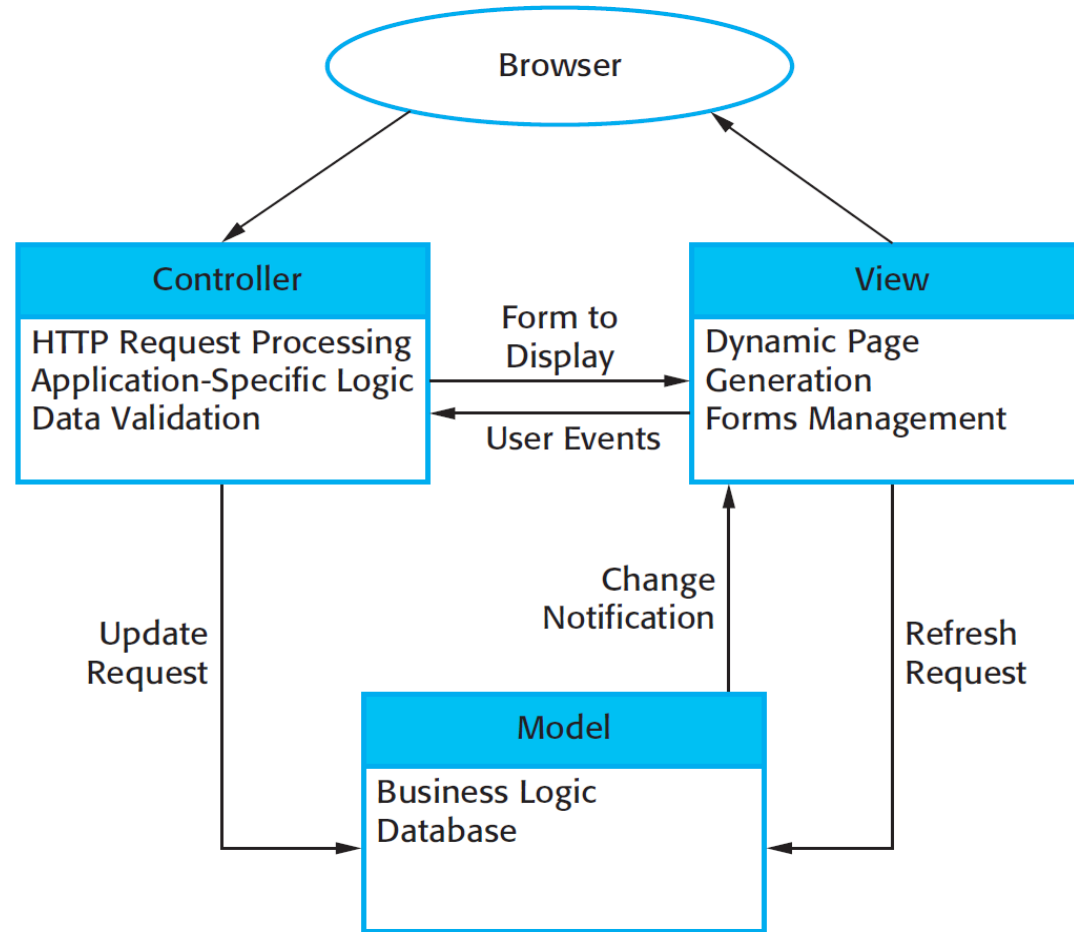
- Subsystems are classified into 3 different types
 - **Model subsystem**: Responsible for application domain knowledge.
 - **View subsystem**: Responsible for displaying application domain objects to the user
 - **Controller subsystem**: Responsible for sequence of interactions with the user and notifying views of changes in the model



Conceptual view of the MVC



Web application Architecture using the MVC Pattern



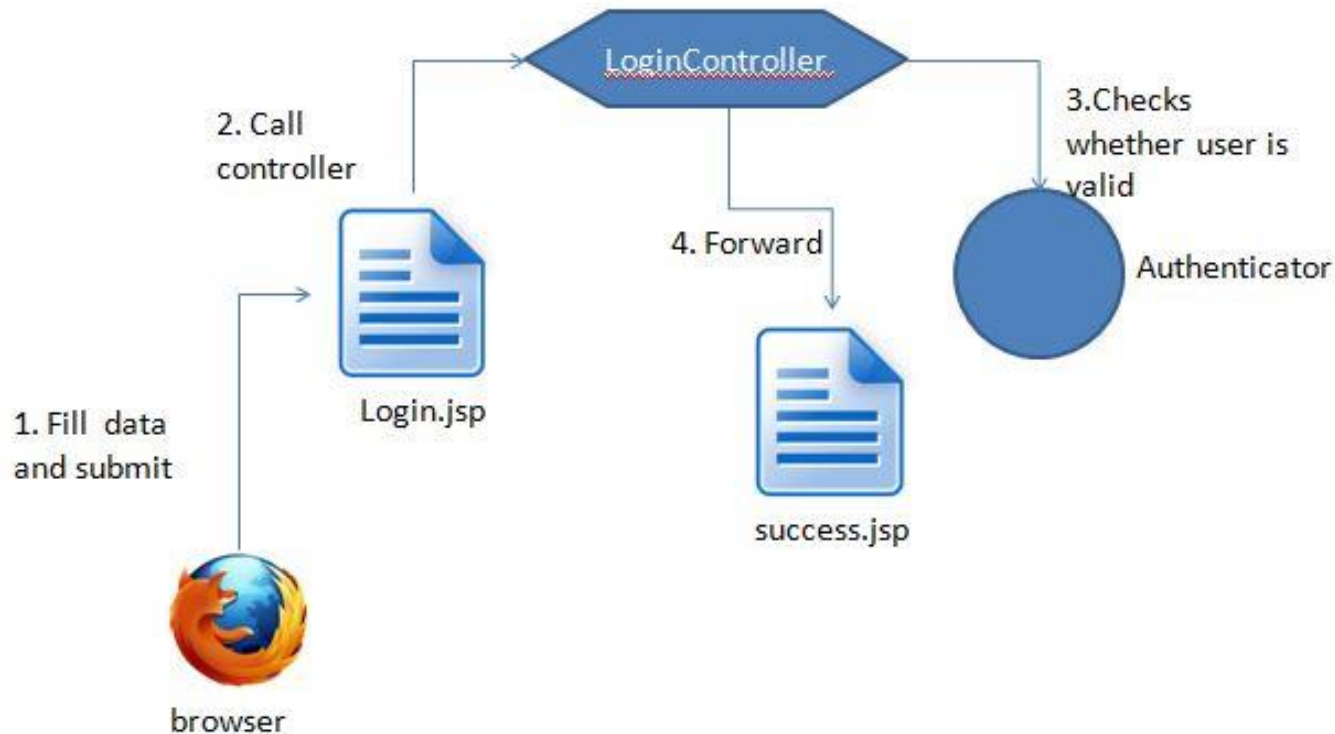
- pattern used for interaction management in a web-based system. (reference: Chapter 6, Software Engineering, by Somerville)

- Model represents a POJO object that carries data.
- View is the layer in which the data is presented in visual format.
- Controller is the component which is responsible for communication between model and view.

MVC architecture with servlets and jsp

- See this example in java from TheJavaGeek.com

<http://www.thejavageek.com/2013/08/11/mvc-architecture-with-servlets-and-jsp/>



3-Layer-Architectural Style

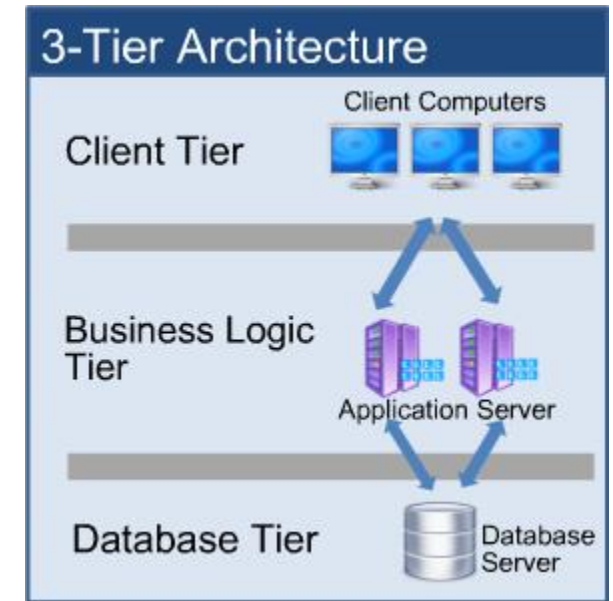
3-Tier Architecture

Definition: 3-Layer 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

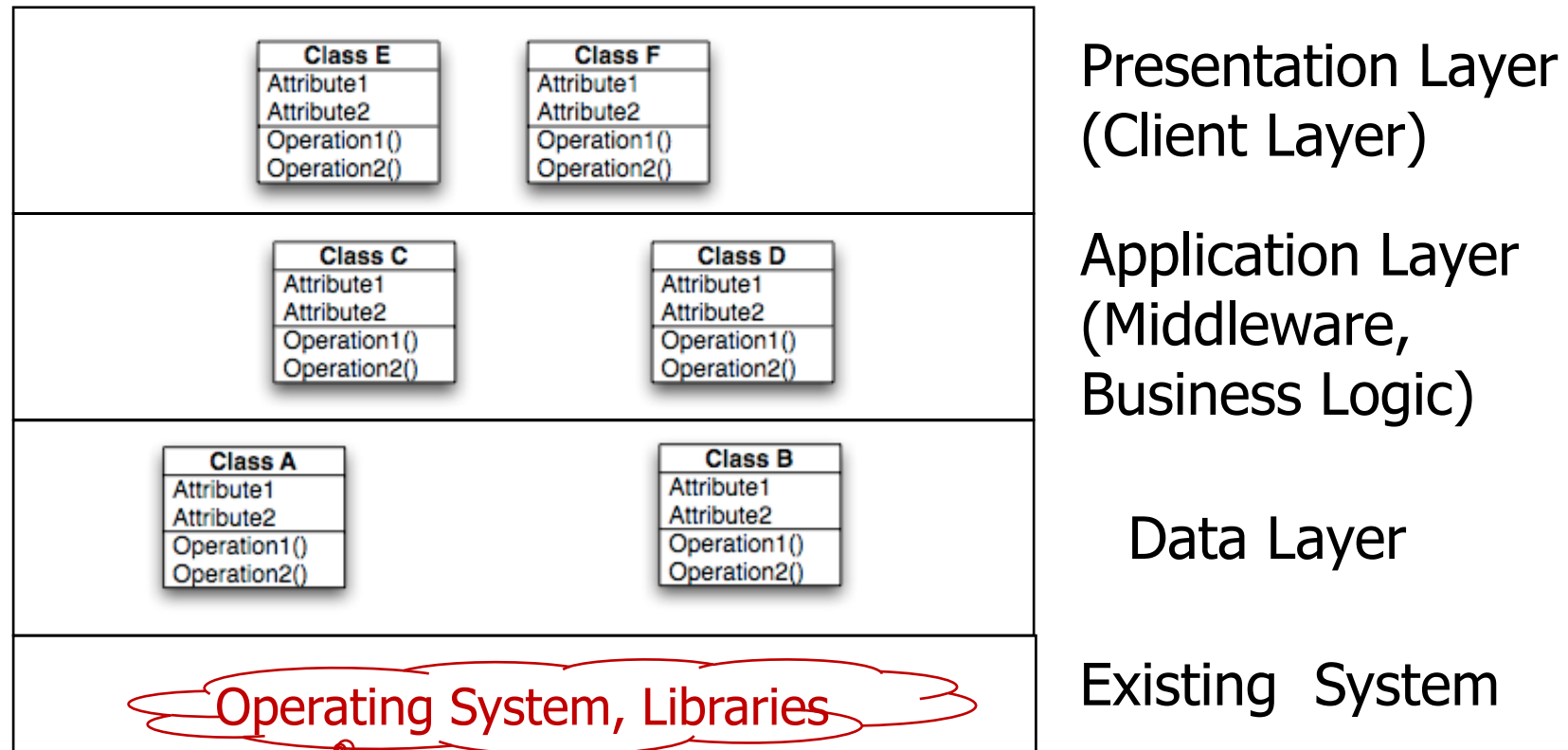
Definition: 3-Tier Architecture

- 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)
- Layers and Tiers are often used interchangeably.



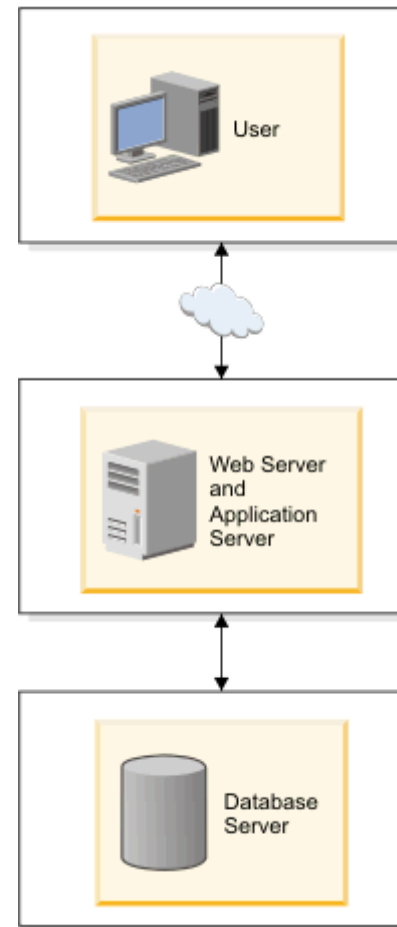
Virtual Machines in 3-Layer Architectural Style

A 3-Layer Architectural Style is a hierarchy of 3 virtual machines usually called presentation, application and data layer

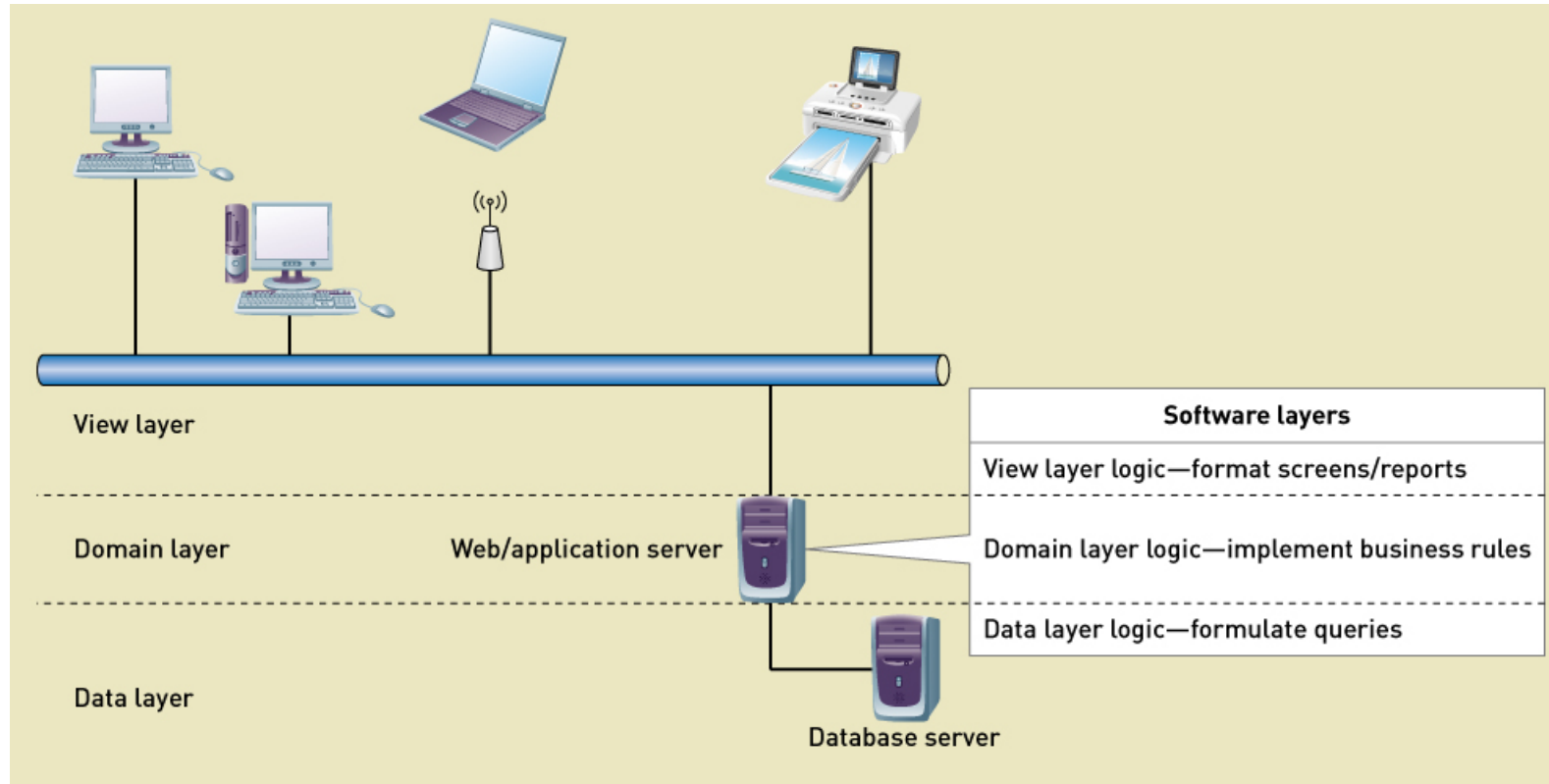


Example of a 3-Layer Architectural Style

- Three-Layer 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 Server** manages and provides access to the persistent data.



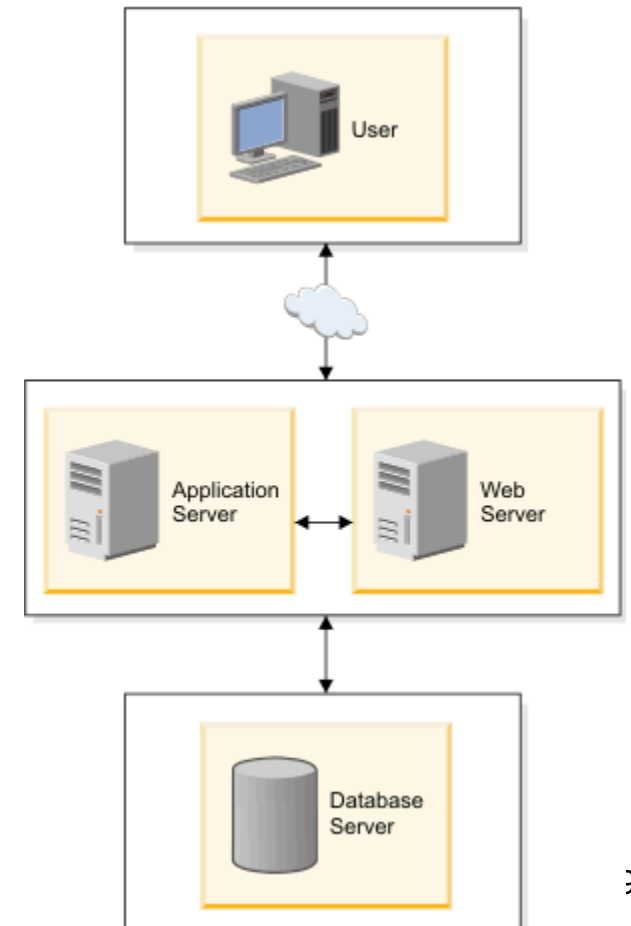
Three Layer Architecture



Example of a 4-Layer Architectural Style

- 4-Layer-architectural styles (4-Tier Architectures) are usually used for the development of electronic commerce sites. The layers are:

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



4-Layer Architectural Style: Another View

- A slightly different view (perhaps more general) may subdivide the 4 layers into the following:
 - ❑ The Presentation Layer (also referred to as the View Layer or Presentation tier) – provides the user interface
 - ❑ The Application Layer (or the Service Layer) – provides the business logic (control)
 - ❑ The Business Layer (or the Business Domain Layer) – provides the data model
 - ❑ The Data Access Layer (or the Persistence Layer) – provides access to the persistent data

Presentation Layer

Application Layer

Business Layer

Data Access Layer

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 subsystem
- 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:**
 - 3-Tier (1990s): Originated with the appearance of Web applications, where the client, middleware and data layers ran on physically separate platforms.
 - MVC (1970-1980): Originated during the development of modular graphical applications for a single graphical workstation at **Xerox PARC**.

Exercise 1:

Discuss how the MVC architecture helps or hurts the following design goals.

- Extensibility (e.g., the addition of new types of views).
- Response time (e.g., the time between a user input and the time all views have been updated)
- Modifiability (e.g., the addition of new attributes in the model)
- Access control (i.e., the ability to ensure that only legitimate users can access specific parts of the model).

Exercise 1:

- Discuss how the MVC architecture helps or hurts the following design goals.
- **Extensibility** (e.g., the addition of new types of views).

The MVC helps extensibility in terms of new types of views, since neither model objects nor existing view objects need to be modified to accommodate the new view.

Exercise 1:

- *Discuss how the MVC architecture helps or hurts the following design goals.*
- **Response time** (e.g., the time between a user input and the time all views have been updated)
- A pure MVC hurts response time, because the view the user sees is updated only after the model has been updated.

This can be an issue for commands involving a rapid series of interactions (e.g., dragging or resizing a geometrical shape).

Exercise 1:

- *Discuss how the MVC architecture helps or hurts the following design goals.*
- **Modifiability** (e.g., the addition of new attributes in the model)
- The MVC helps this goal, since only the views that need to be aware of the new attribute need to be modified. All other views can be left unchanged.

Exercise 1:

- *Discuss how the MVC architecture helps or hurts the following design goals.*
- **Access control** (i.e., the ability to ensure that only legitimate users can access specific parts of the model).

The MVC helps this goal, as the model is accessed using a clear interface (the public methods of the model objects), which can be controlled, for example, using a proxy pattern for each model class.

Exercise 2:

List design goals that would be difficult to meet when using a closed architecture with many layers, such as the OSI example.

Solution:

- A closed architecture requires method invocations across each layer boundary and often also copying data between each layer. This hurts both response time and memory foot print.

Exercise 3:

In many architectures, such as the three- and four-tier architectures, the storage of persistent objects is handled by a dedicated layer. In your opinion, which design goals have lead to this decision?

Exercise 3: Solution

In many architectures, such as the three- and four-tier architectures, the storage of persistent objects is handled by a dedicated layer. In your opinion, which design goals have lead to this decision?

- **Portability:** Separating the application from the details of storage allows the storage layer to be replaced by a different one (e.g., switching database management systems).
- **Robustness:** It is easier to guarantee the integrity of the stored data if it is handled by a single layer.

Summary

- System Design
 - An activity that reduces the gap between the problem and an existing (virtual) machine
- Design Goals Definition
 - Describes the important system qualities
 - Defines the values against which options are evaluated
- Subsystem Decomposition
 - Decomposes the overall system into manageable parts by using the principles of cohesion and coherence
- Architectural Style
 - A pattern of a typical subsystem decomposition
- Software architecture
 - An instance of an architectural style
 - Client Server, Peer-to-Peer, Model-View-Controller.

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
- L.D. Erman, F. Hayes-Roth (1980)
 - The Hearsay-II-Speech-Understanding System, ACM Computing Surveys, Vol 12. No. 2, pp 213-253
- 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.