Object-Oriented Software Engineering

Using UML, Patterns, and Java

System Design: Decomposing the System

System Design: Decomposing the System

- **System design is the transformation of an analysis model into a system design model.**

- During system design, developers **define the design goals of the project and decompose the system into smaller subsystems** that can be realized by individual teams.
- Developers also **select strategies for building the system, such as the hardware/software strategy, the persistent data management strategy, the global control flow, the access control policy, and the handling of boundary conditions**.

System Design: Decomposing the System_2

- **System design is not algorithmic**.
- Developers must make **trade-offs among many design goals that often conflict with each other**. They also cannot anticipate all design issues that they will face because they do not yet have a clear picture of the solution domain.
- System design is decomposed into several activities, each addressing part of the overall problem of decomposing the system:
  - ***Identify design goals***. Developers identify and prioritize the qualities of the system that they should optimize.

System Design: Decomposing the System_3

- ***Design the initial subsystem decomposition***. Developers decompose the system into smaller parts based on the use case and analysis models. Developers use standard architectural styles as a starting point during this activity.

- ***Refine the subsystem decomposition to address the design goals***. The initial decomposition usually does not satisfy all design goals. Developers refine it until all goals are satisfied.

Introduction: A Floor Plan Example

- The task is to design a residential house. After agreeing with the client on the number of rooms and floors, the size of the living area, and the location of the house, the architect must design the floor plan, that is, where the walls, doors, and windows should be located.

- He must do so according to a number of functional requirements: the kitchen should be close to the dining room and the garage, the bathroom should be close to the bedrooms, and so on.

Introduction: A Floor Plan Example_2

- We set out to satisfy the following constraints:

1. This house should have two bedrooms, a study, a kitchen, and a living room area.

2. The overall distance the occupants walk every day should be minimized.

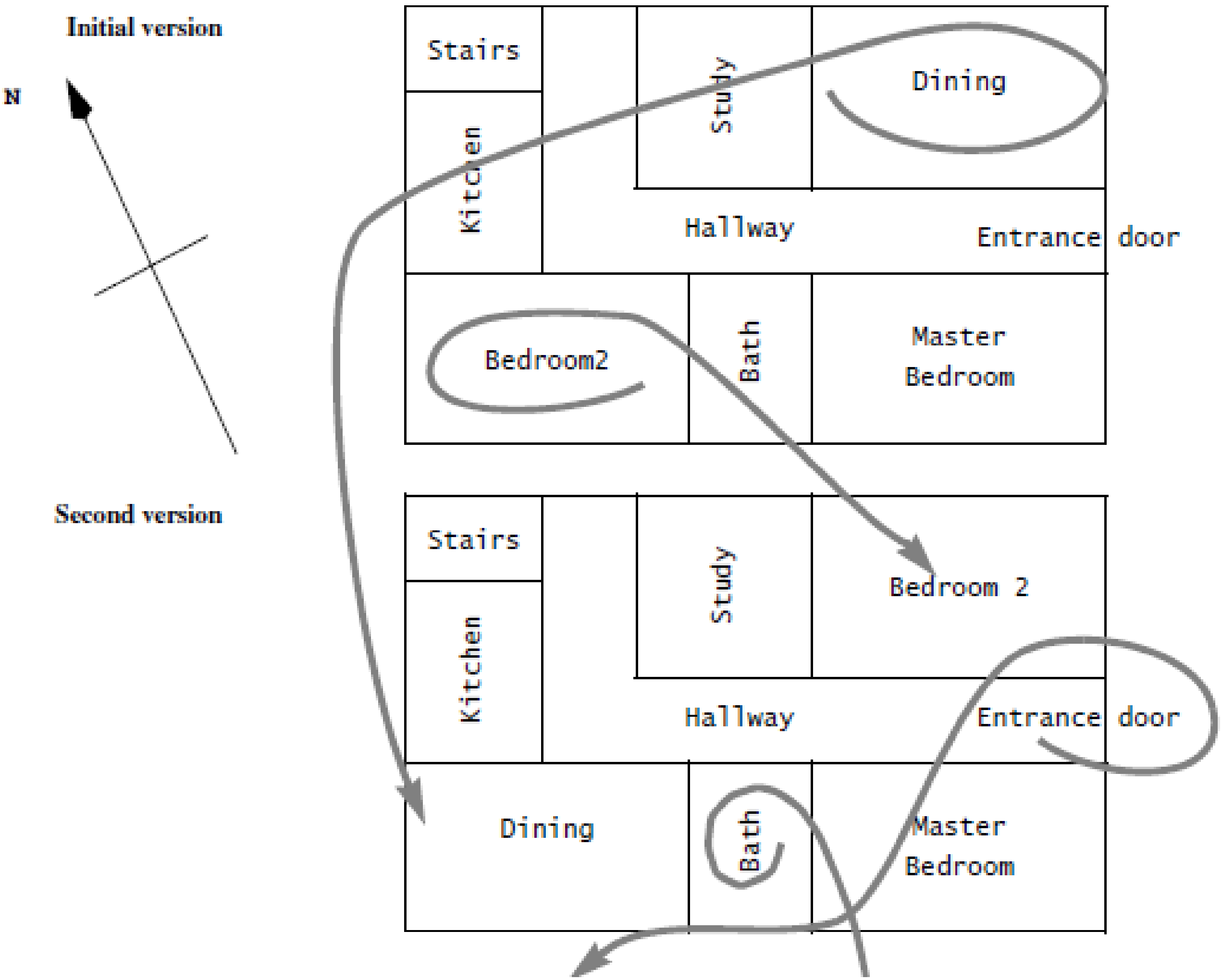
3. The use of daylight should be maximized.

Introduction: A Floor Plan Example_3

- To satisfy the above constraints, we assume that most of the walking will be done between the entrance door and the kitchen, when groceries are unloaded from the car, and between the kitchen and the living/dining area, when dishes are carried before and after the meals.

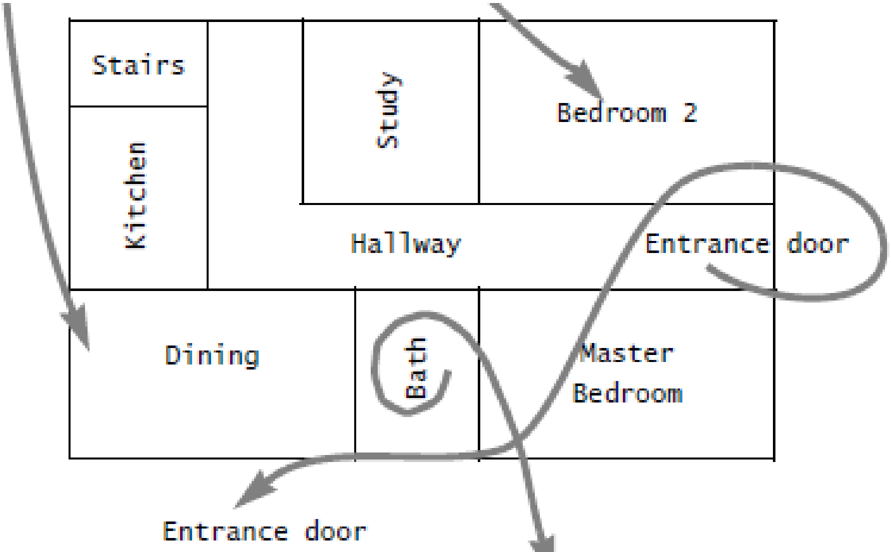
- The next walking path to minimize is the path from the bedrooms to the bathrooms. We assume that the occupants of the house will spend most of their time in the living/dining area and in the master bedroom.

A floor plan example

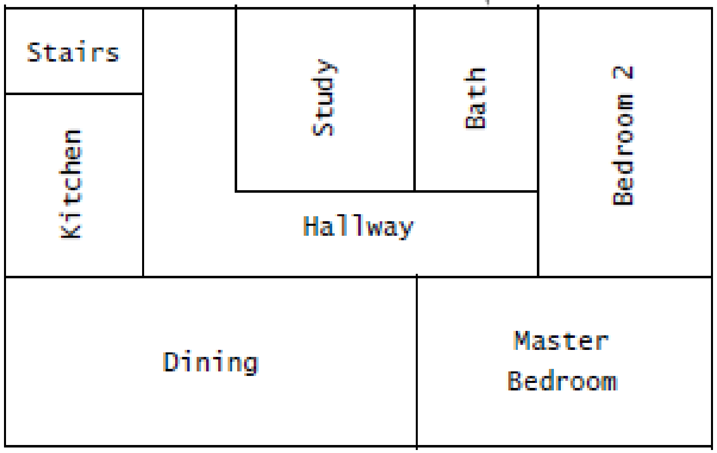


A floor plan example cont

Second version



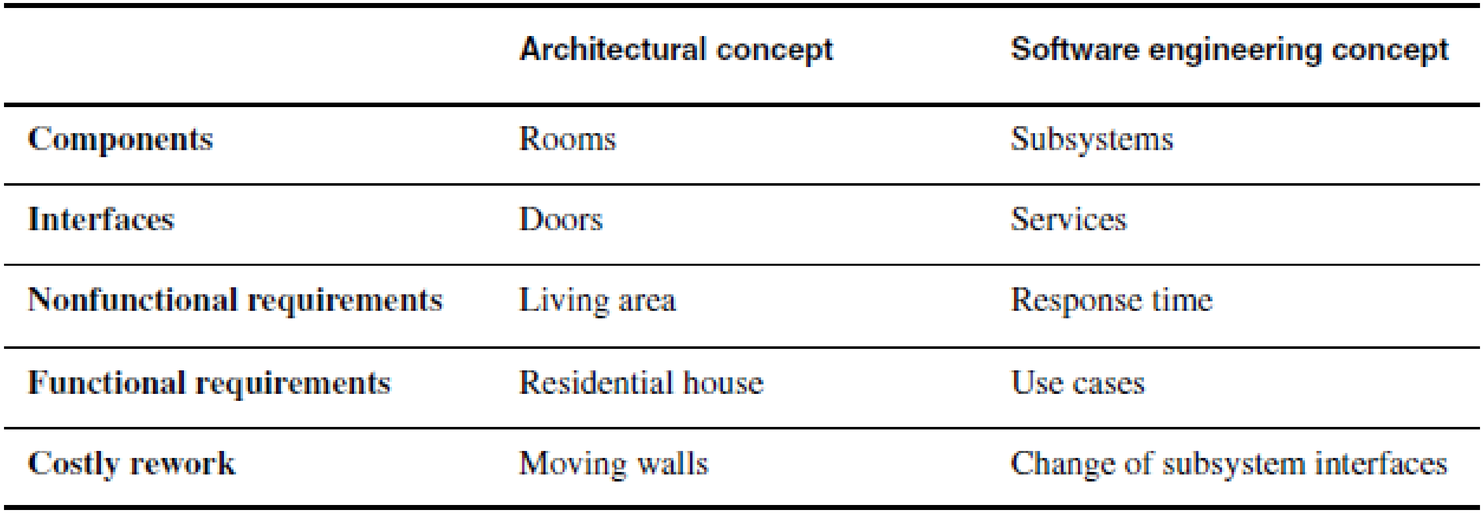
Third version



Similarities between architecture & software

- The design of a floor plan in architecture is like system design in software engineering (Table 6-1). The whole is divided into simpler components and interfaces, while considering nonfunctional and functional requirements.

Table 6-1 Mapping of architectural and software engineering concepts.



	Architectural concept	Software engineering concept
Components	Rooms	Subsystems
Interfaces	Doors	Services
Nonfunctional requirements	Living area	Response time
Functional requirements	Residential house	Use cases
Costly rework	Moving walls	Change of subsystem interfaces

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 Garder):
  - If our brain would be so simple that we can understand it, we would be too stupid to understand it.

Sir **Antony Hoare,** *1934
- Quicksort
- Hoare logic for verification
- CSP (Communicating Sequential Processes): modeling language for concurrent processes (basis for Occam).

Jostein Gardner, *1952, writer
Uses metafiction in his stories:
Fiction which uses the device of fiction
- Best known for: „Sophie‘s World“.

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

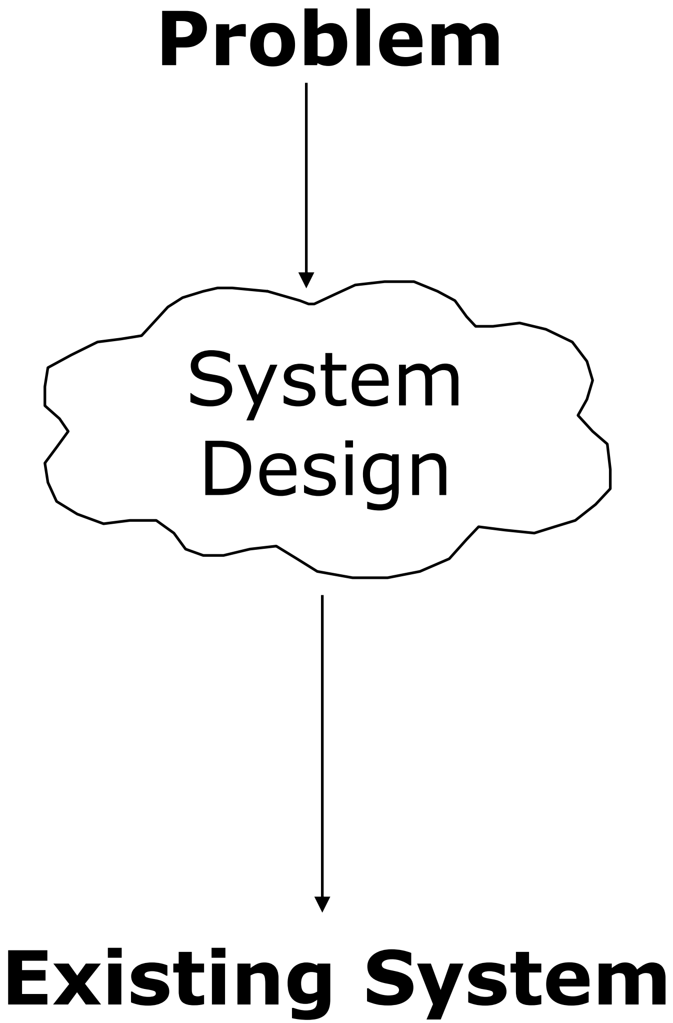
- Design knowledge is a moving target

- Design window: Time in which design decisions must 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.



From Analysis to Design

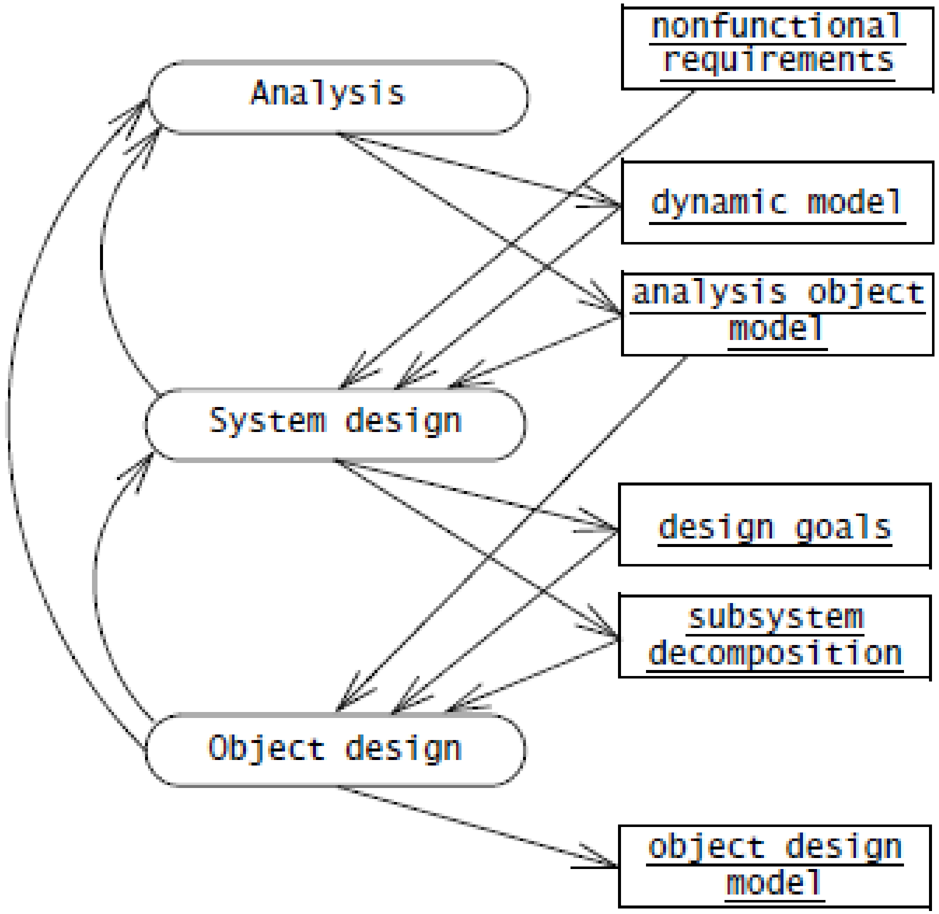


Figure 6-2 The activities of system design (UML activity diagram).

System Design Concepts

- A **subsystem** is a **replaceable part of the system with well-defined interfaces** that encapsulates the state and behavior of its contained components. A subsystem typically corresponds to the amount of work that a single developer or a single development team can tackle.

- By decomposing the system into relatively independent subsystems, concurrent teams can work on individual subsystems with minimal communication overhead.

System Design Concepts_2

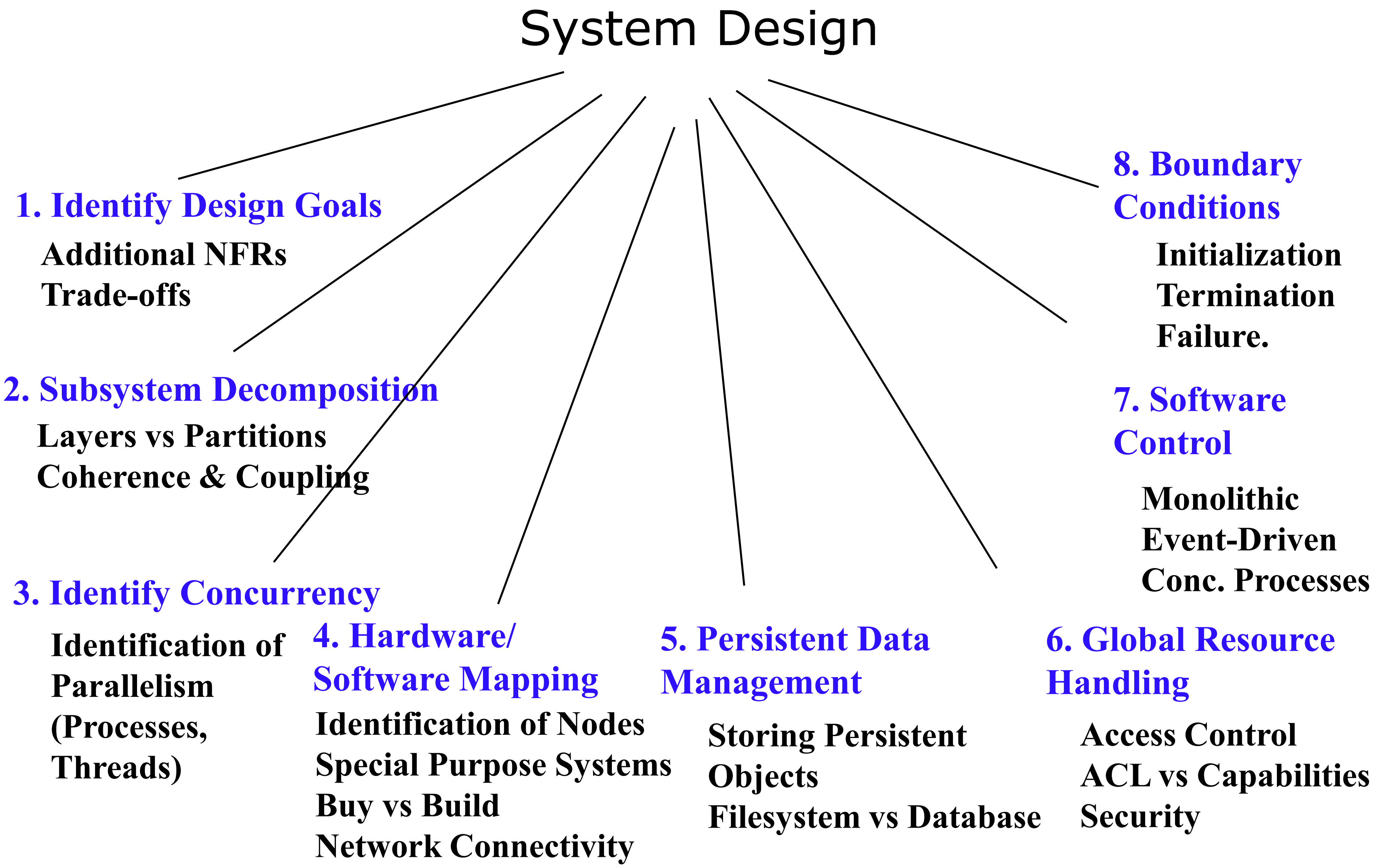
- A **service** is a set of related operations that share a common purpose. During system design, we define the subsystems in terms of the services they provide. Later, during object design, we define the subsystem interface in terms of the operations it provides.
- **Coupling,** measure the dependencies between two subsystems, whereas **cohesion** measures the dependencies among classes within a subsystem. Ideal subsystem decomposition should minimize coupling and maximize cohesion.

System Design Concepts_3

- **Layering** allows **a system** to be **organized as a hierarchy of subsystems**, each providing higher-level services to the subsystem above it by using lower-level services from the subsystems below it.

- **Partitioning organizes subsystems as peers that mutually provide different services to each other**.

System Design: Eight Issues



System Design

1. Identify Design Goals

Additional NFRs
Trade-offs

2. Subsystem Decomposition

Layers vs Partitions
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

6. Global Resource Handling

Access Control
ACL vs Capabilities
Security

7. Software Control

Monolithic
Event-Driven
Conc. Processes

8. Boundary Conditions

Initialization
Termination
Failure.

Subsystem Decomposition

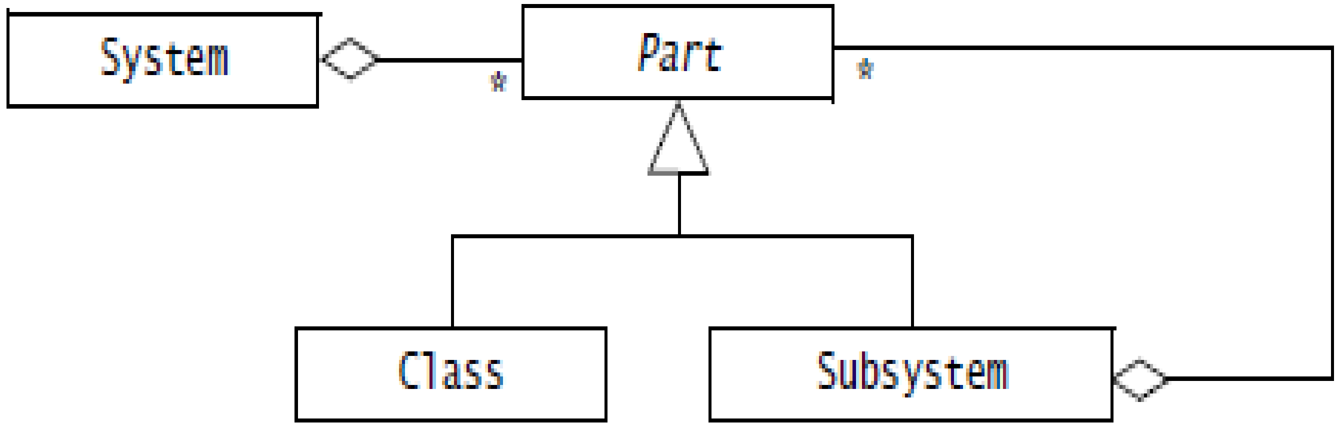
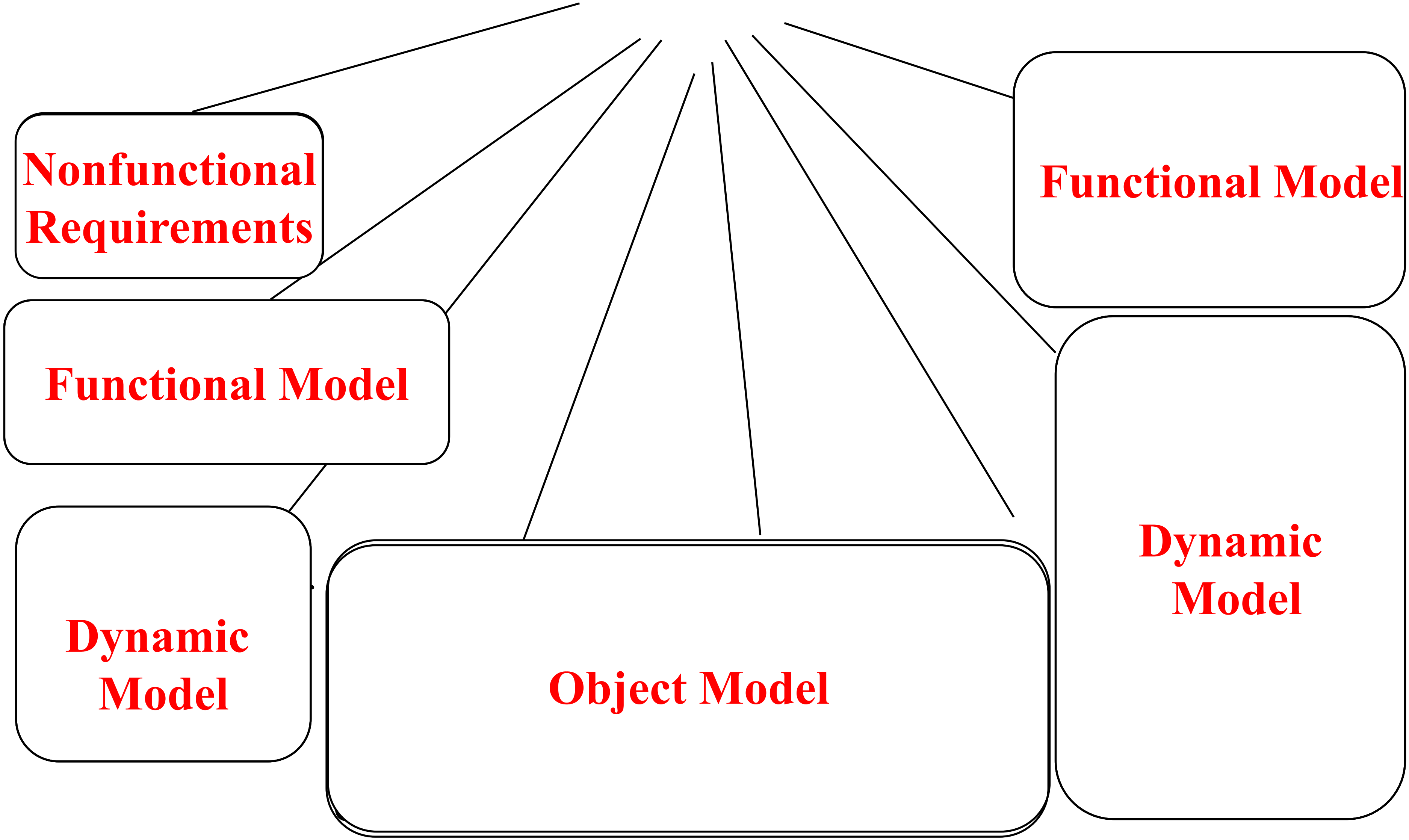


Figure 6-3 Subsystem decomposition (UML class diagram).

Composite Design Pattern

Analysis Sources: Requirements and System Model


Nonfunctional Requirements

Functional Model

Dynamic Model

Object Model

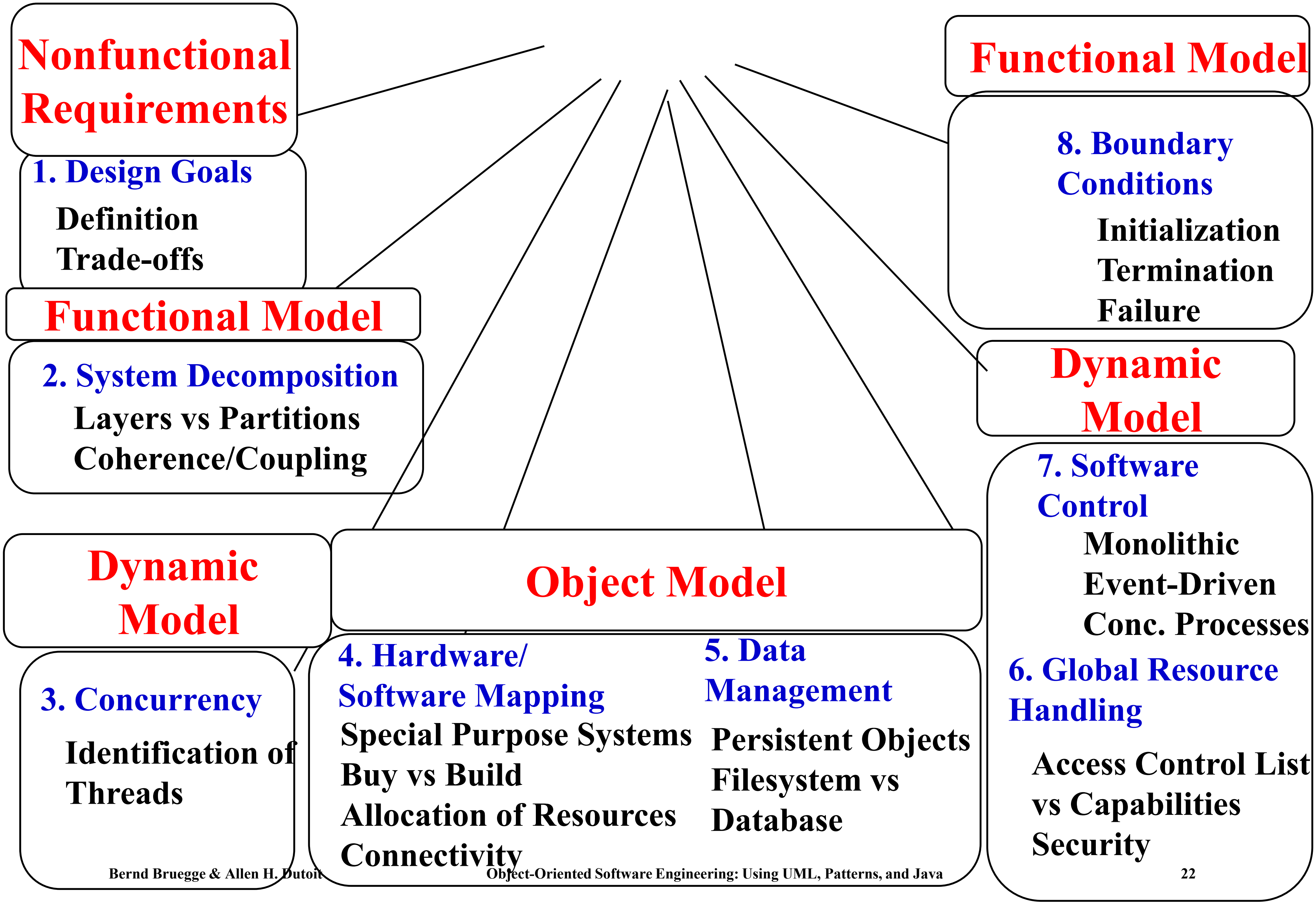
Functional Model

Dynamic 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
- Finally: Hardware/Software Mapping
  - => Boundary conditions

From Analysis to System Design



Nonfunctional Requirements

1. Design Goals
- Definition
- Trade-offs

Functional Model

2. System Decomposition
- Layers vs Partitions
- Coherence/Coupling

Dynamic Model

3. Concurrency
- Identification of Threads

Object Model

4. Hardware/ Software Mapping
- Special Purpose Systems
- Buy vs Build
- Allocation of Resources
- Connectivity

5. Data Management
- Persistent Objects
- Filesystem vs Database

Functional Model

8. Boundary Conditions
- Initialization
- Termination
- Failure

Dynamic Model

7. Software Control
- Monolithic
- Event-Driven
- Conc. Processes

6. Global Resource Handling
- Access Control List vs Capabilities
- Security

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

Subsystem Decomposition

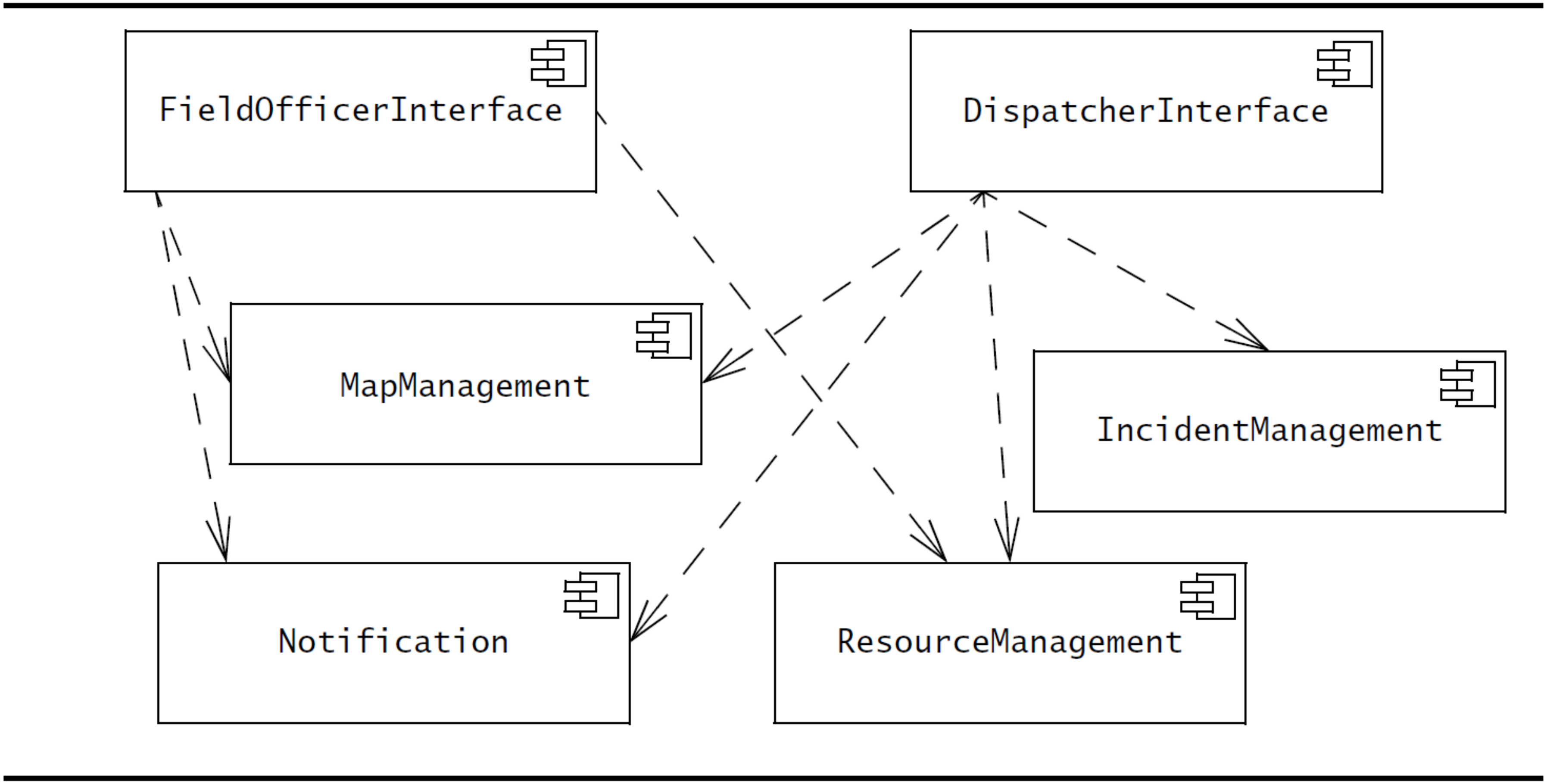


Figure 6-4 Subsystem decomposition for an accident management system (UML component diagram). Subsystems are shown as UML components. Dashed arrows indicate dependencies between subsystems.

Services and Subsystem Interfaces

- A subsystem is characterized by the services it provides to other subsystems. A **service** is a set of related operations that share a common purpose. A subsystem provides a notification service, for example, defines operations to send notices, looks up for notification channels, subscribes and unsubscribes to a channel.

- The set of operations of a subsystem that are available to other subsystems form the **subsystem interface**.

Services and Subsystem Interfaces_2

- System design focuses on defining the services provided by each subsystem, that is, enumerating the operations, their parameters, and their high-level behavior.

- Object design will focus on the **application programmer interface** (API), which refines and extends the subsystem interfaces. The API also includes the type of the parameters and the return value of each operation.

Services and Subsystem Interfaces_3

- Provided and required interfaces can be depicted in UML with **assembly connectors**, also called **ball-and-socket connectors**.

  - The provided interface is shown as a **ball icon** (also called **lollipop**) with its name next to it.

  - A **required interface** is shown as a **socket icon**.

  - The dependency between two subsystems is shown by connecting the corresponding ball and socket in the component diagram.

Ball & Socket Icons

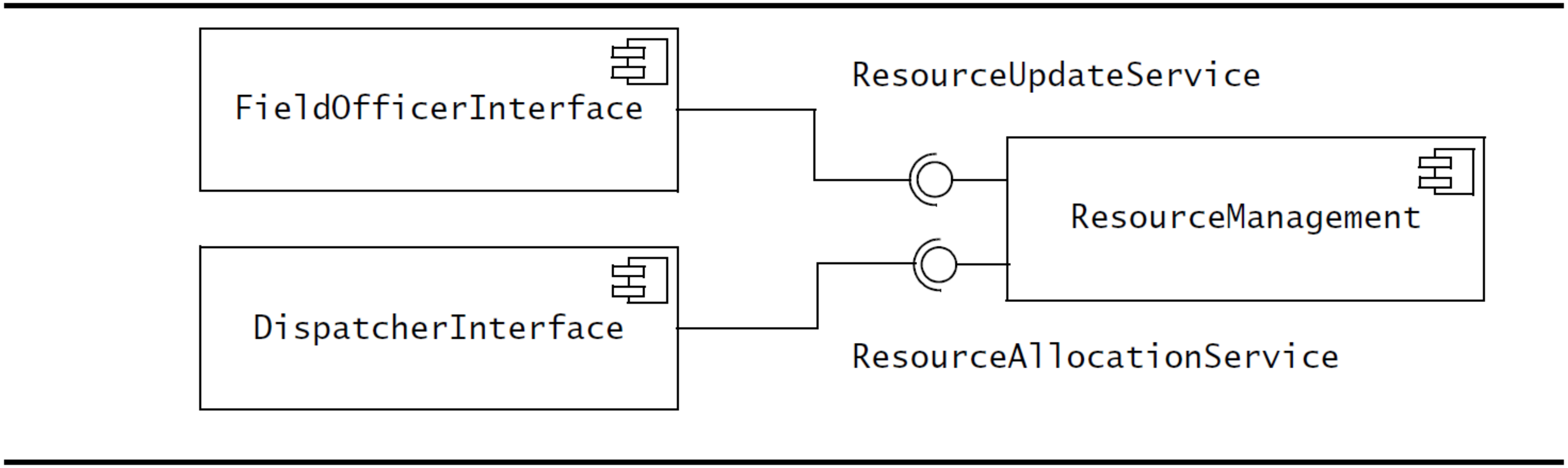
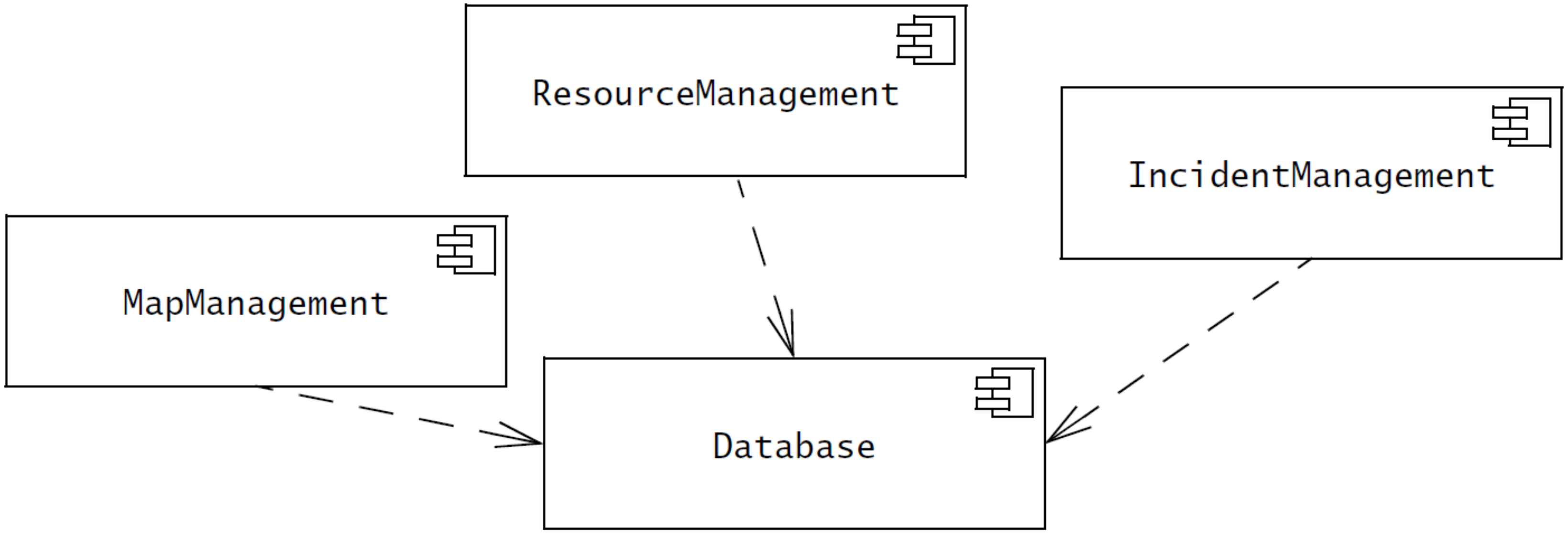


Figure 6-5 Services provided by the ResourceManagement subsystem (UML component diagram, ball-and-socket notation depicting provided and required interfaces).

Accessing the Database subsystem_1

Alternative 1: Direct access to the Database subsystem



Accessing the Database subsystem_2

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

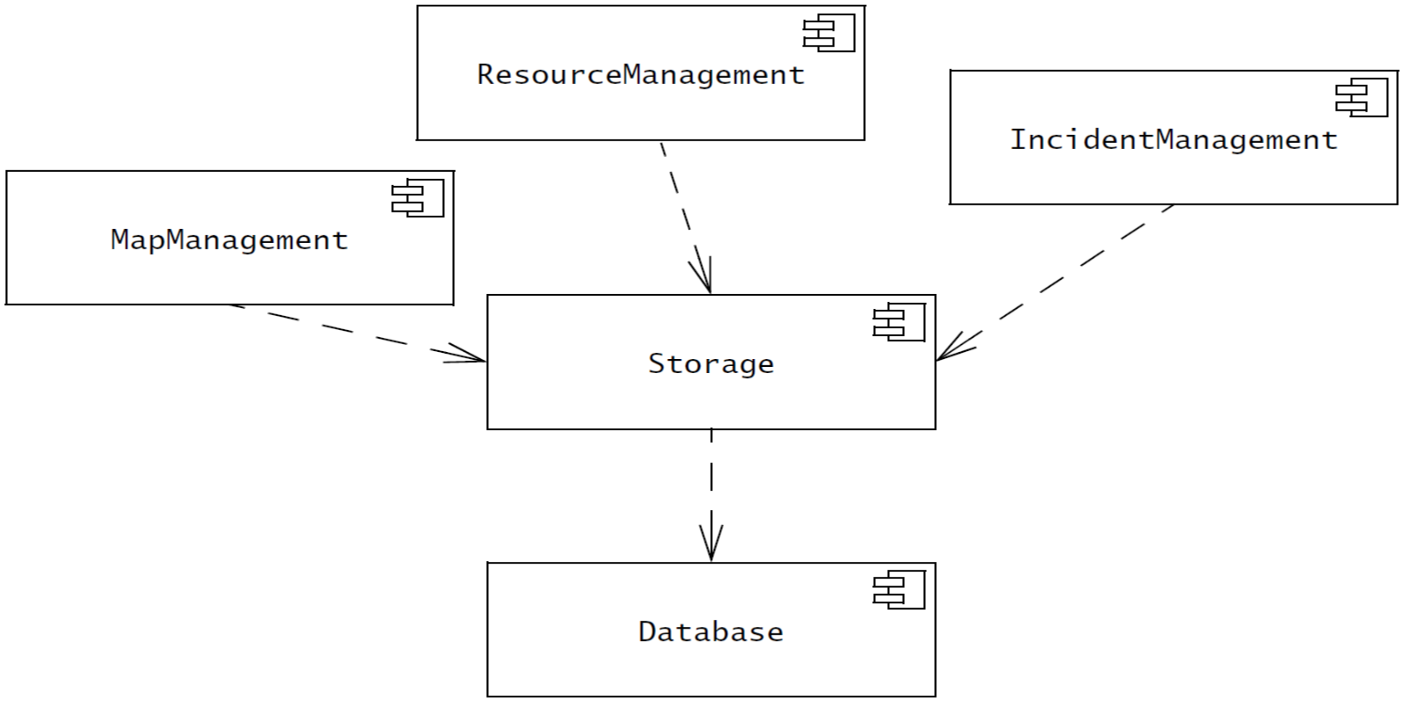


Figure 6-6 Example of reducing the coupling of subsystems (UML component diagram, subsystems FieldOfficerInterface, DispatcherInterface, and Notification omitted for clarity). Alternative 1 depicts a situation where all subsystems access the database directly, making them vulnerable to changes in the interface of the Database 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.

Coupling and Cohesion

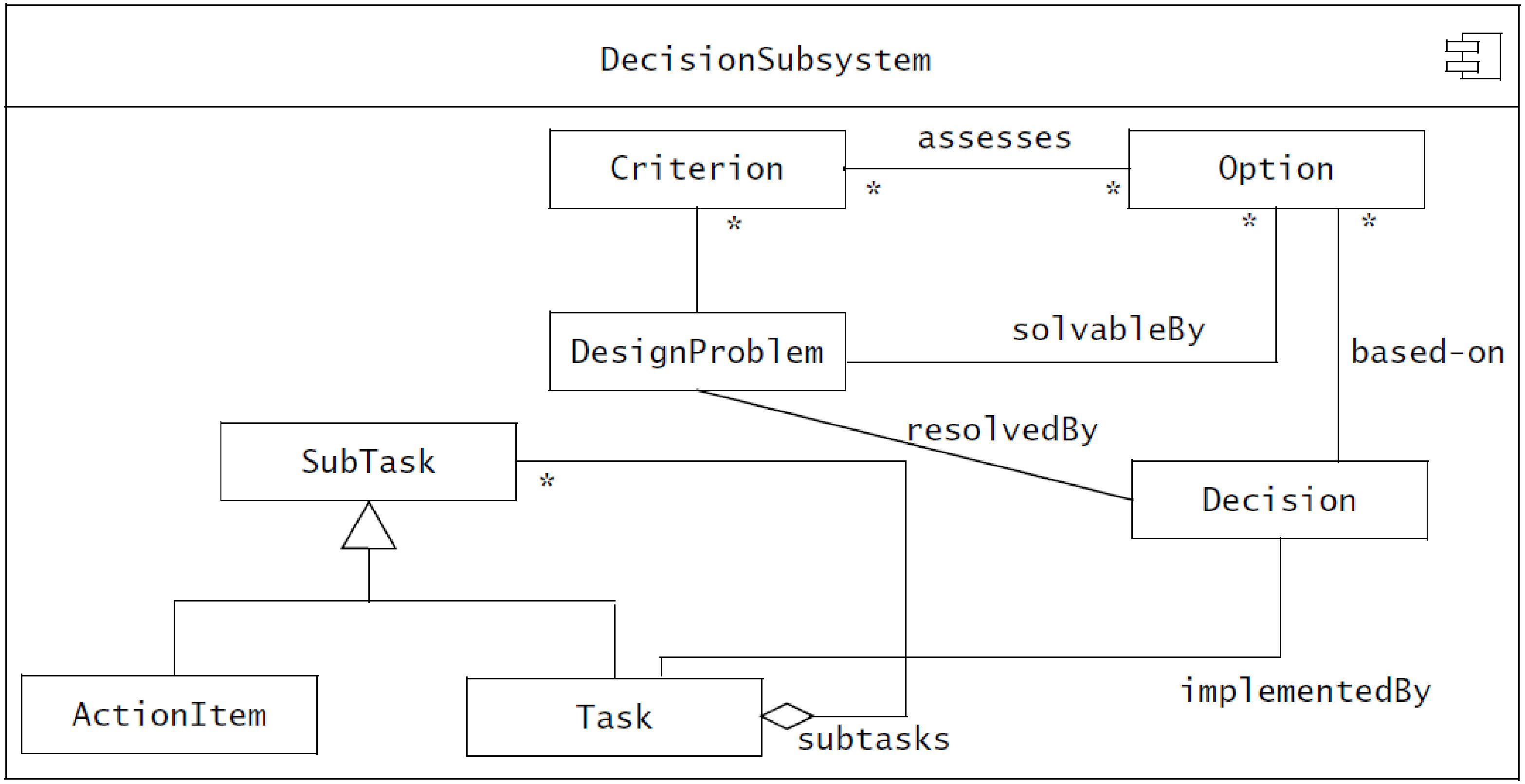


Figure 6-7 Decision tracking system (UML component diagram). The DecisionSubsystem has a low cohesion: The classes Criterion, Option, and DesignProblem have no relationships with Subtask, ActionItem, and Task.

Coupling and Cohesion_2

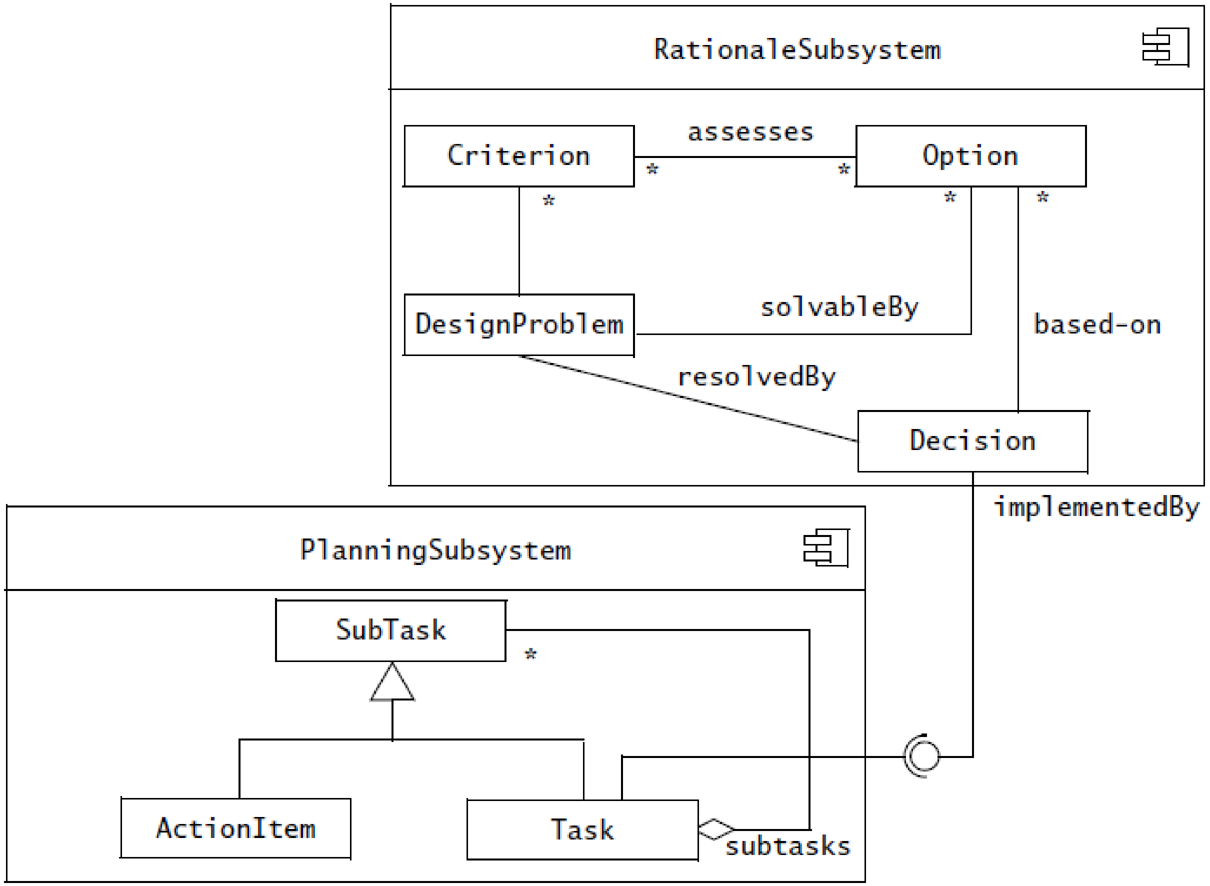


Figure 6-8 Alternative subsystem decomposition for the decision tracking system of Figure 6-7 (UML component diagram, ball-and-socket notation). The cohesion of the RationaleSubsystem and the PlanningSubsystem is higher than the cohesion of the original DecisionSubsystem. The RationaleSubsystem and PlanningSubsystem subsystems are also simpler. However, we introduced an interface for realizing the relationship between Task and Decision.

Typical Design Trade-offs

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

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 origin ("seed") for services are the use cases from the functional model
- Services are defined during system design.

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.*

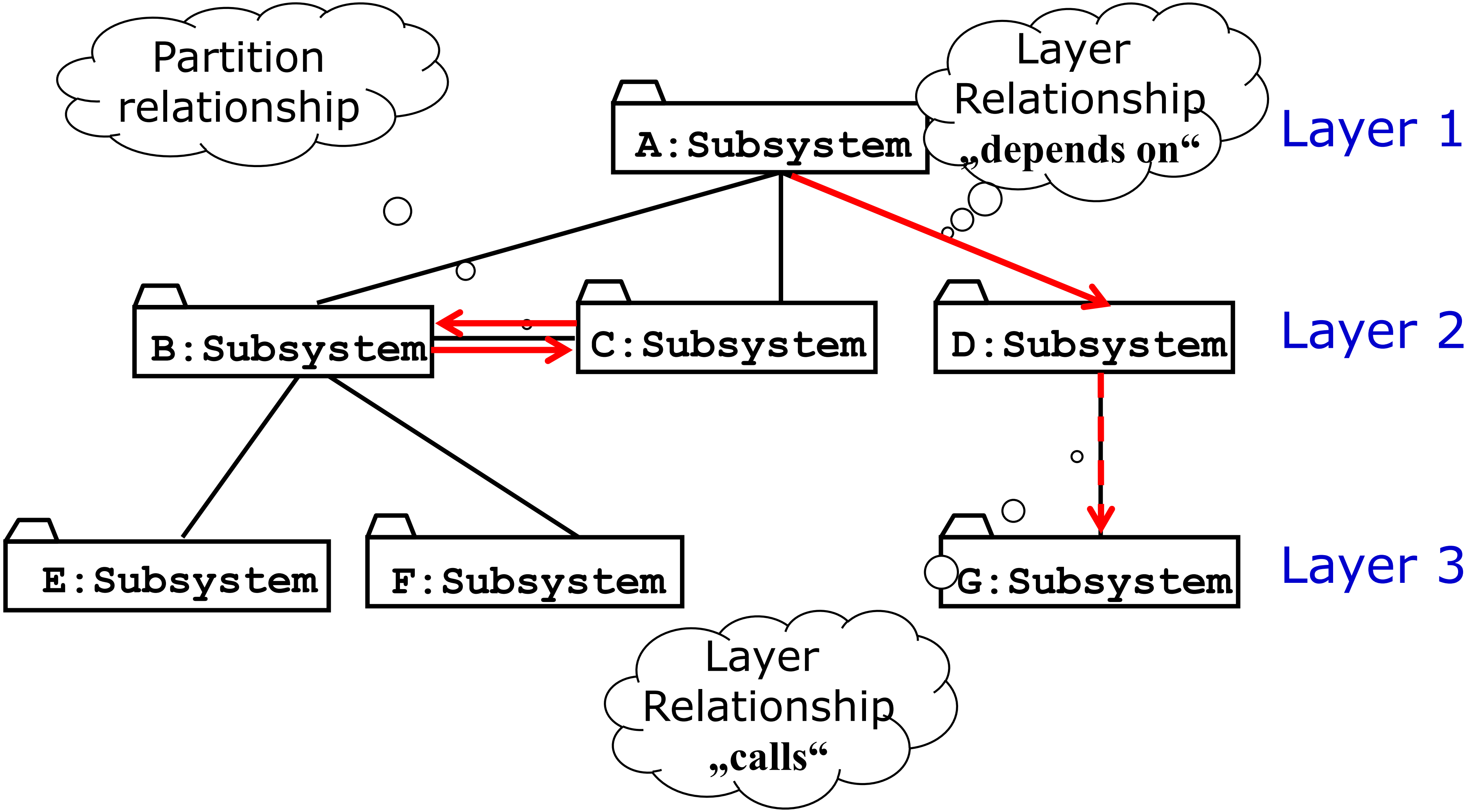
Properties of Subsystems: Layers and Partitions

- 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.

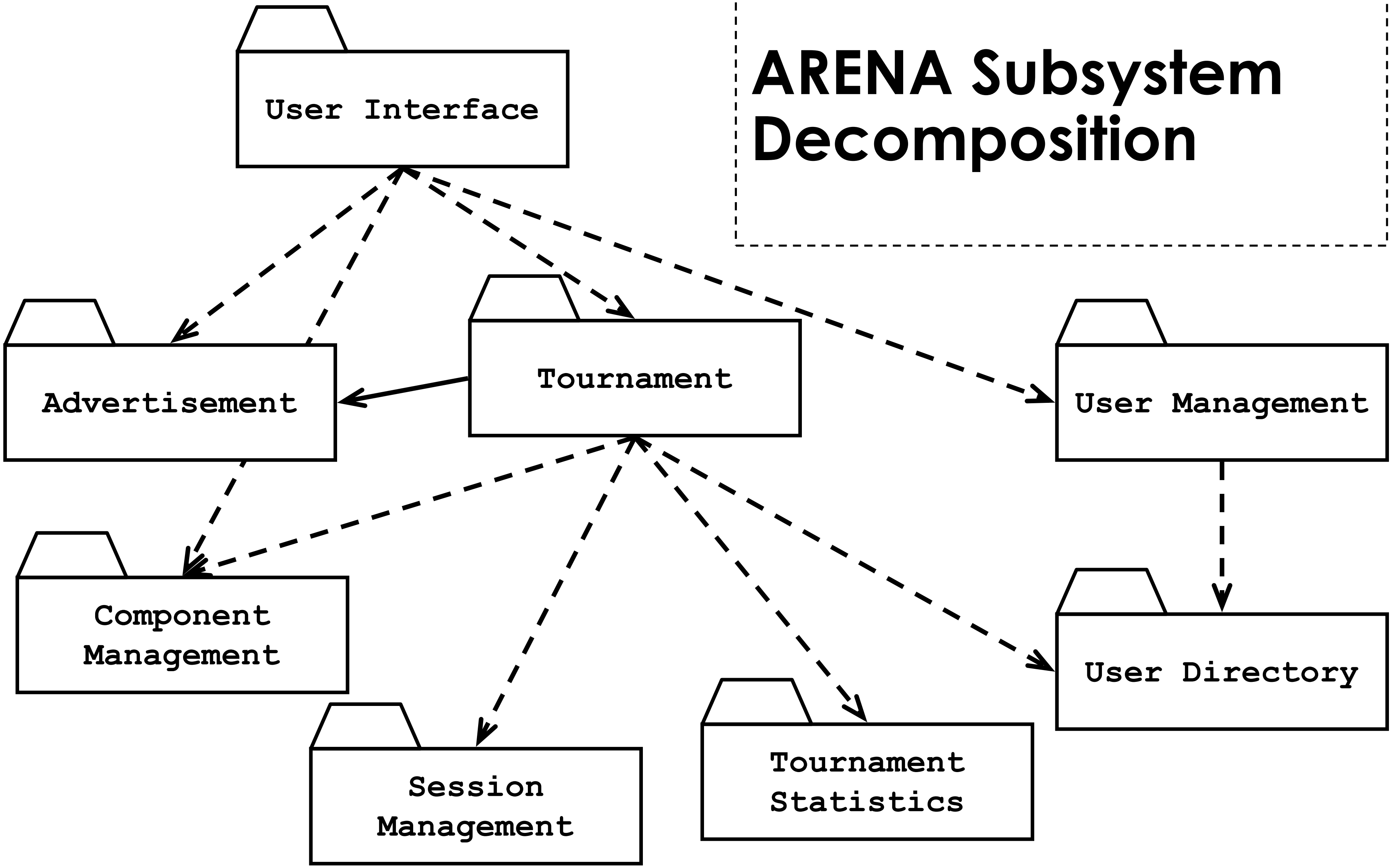
Relationships between Subsystems

- 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.

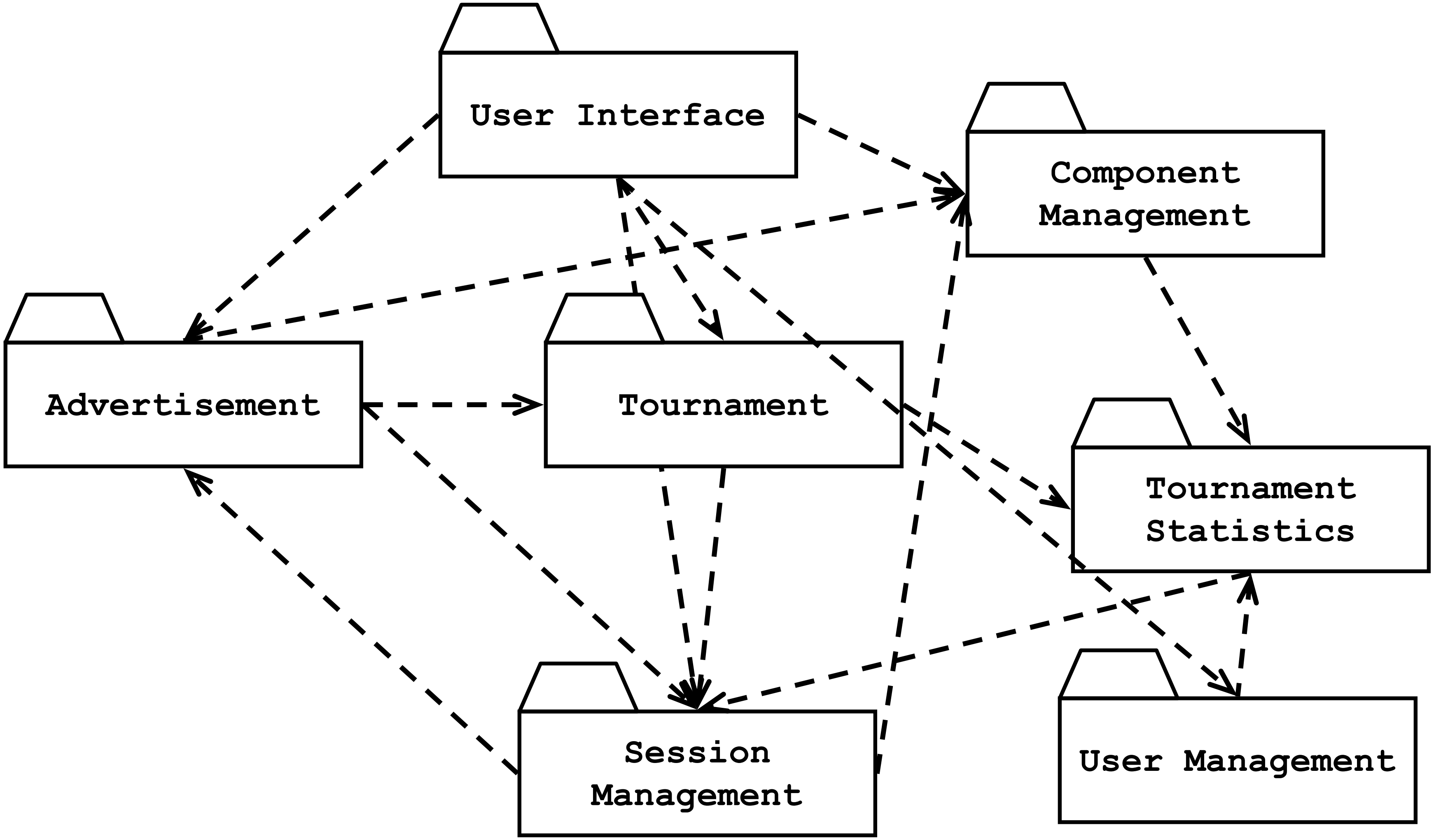
Example of a Subsystem Decomposition



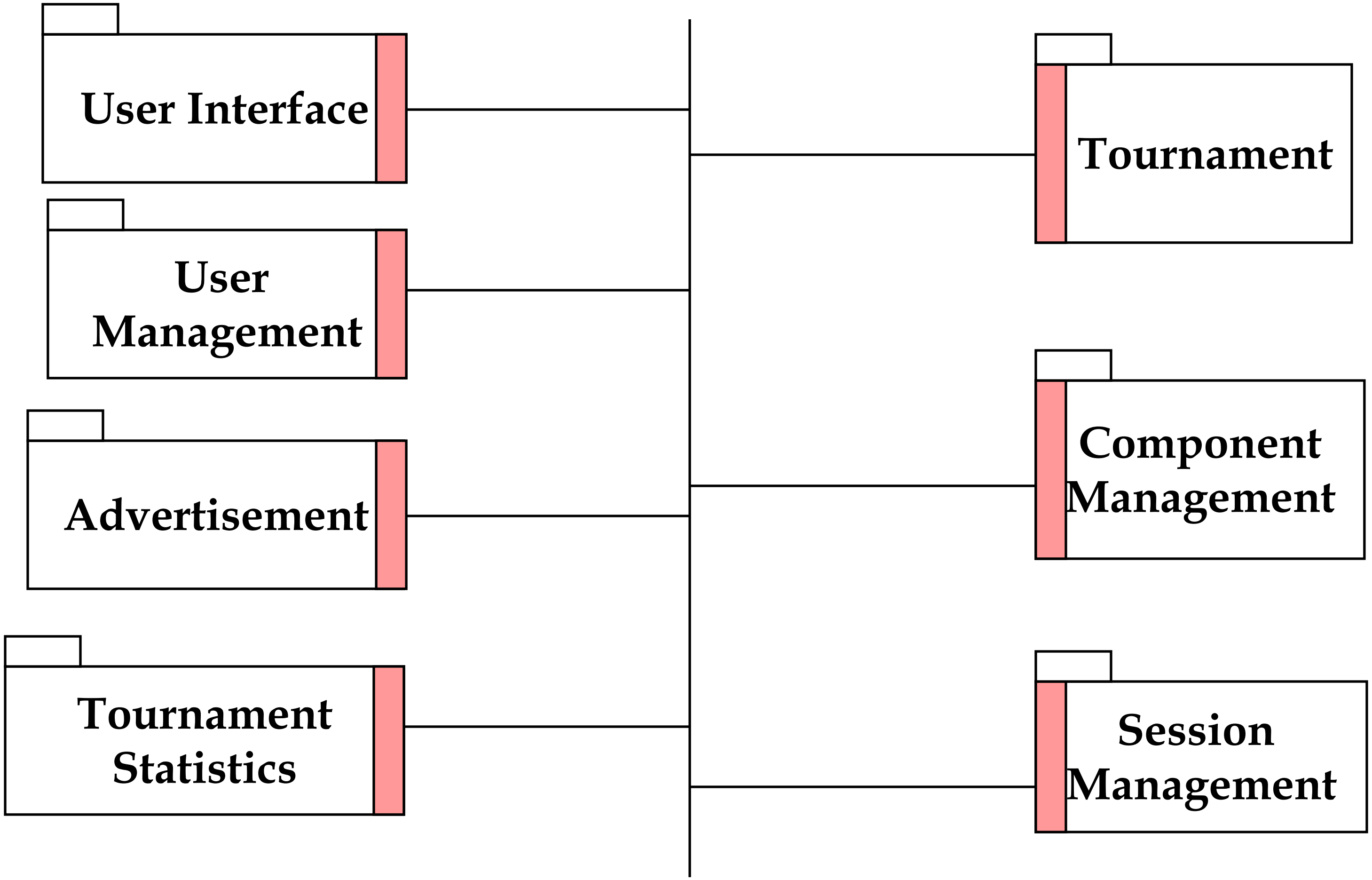
ARENA Subsystem Decomposition



Example of a Bad Subsystem Decomposition



Good Design: The System as set of Interface Objects



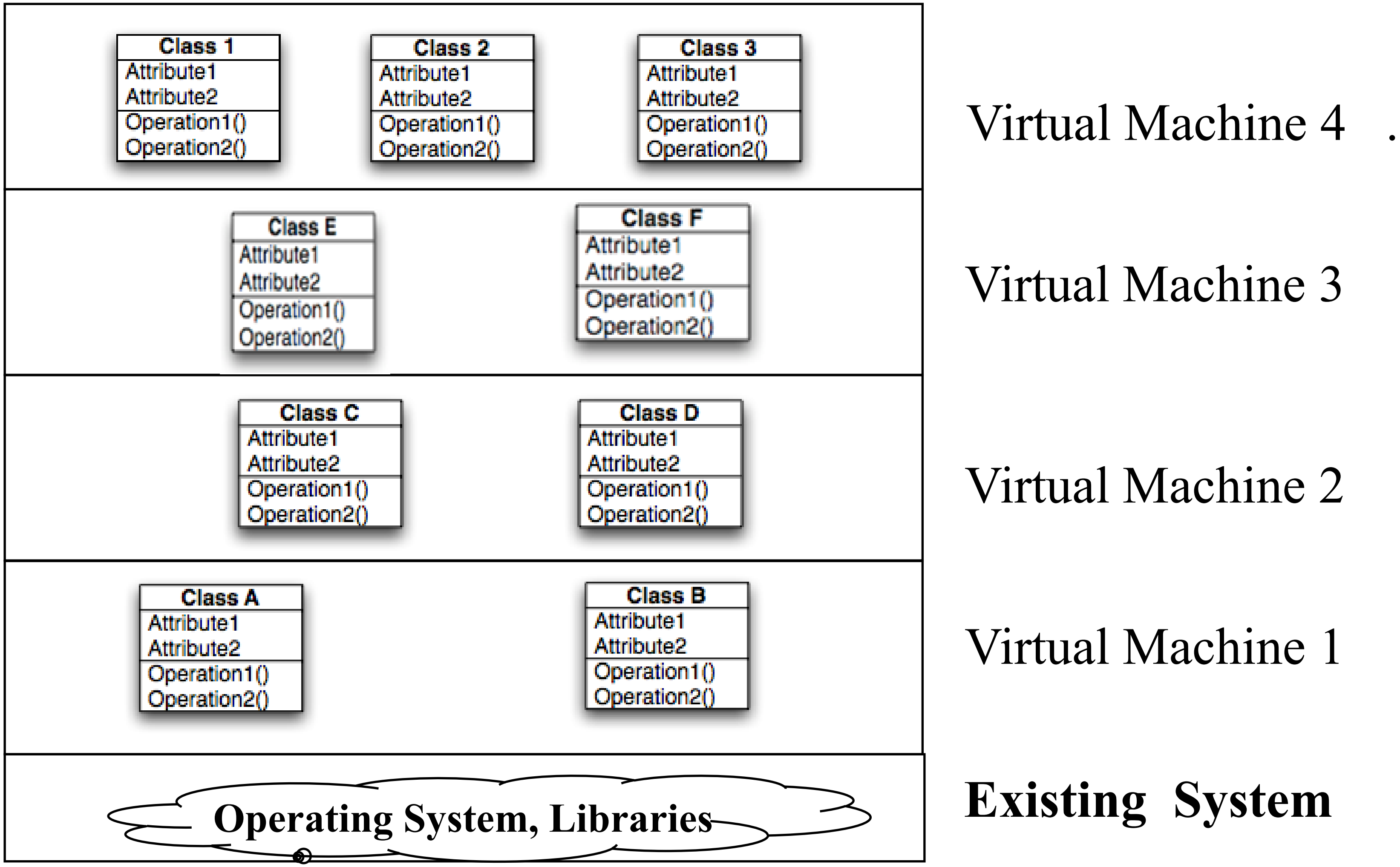
🟥 Subsystem Interface Objects

Virtual Machine

- A virtual machine is a subsystem connected to higher and lower-level virtual machines by "providing and requesting services"
- 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.



Class 1
Attribute1
Attribute2
Operation1()
Operation2()

Class 2
Attribute1
Attribute2
Operation1()
Operation2()

Class 3
Attribute1
Attribute2
Operation1()
Operation2()

Virtual Machine 4 .

Class E
Attribute1
Attribute2
Operation1()
Operation2()

Class F
Attribute1
Attribute2
Operation1()
Operation2()

Virtual Machine 3

Class C
Attribute1
Attribute2
Operation1()
Operation2()

Class D
Attribute1
Attribute2
Operation1()
Operation2()

Virtual Machine 2

Class A
Attribute1
Attribute2
Operation1()
Operation2()

Class B
Attribute1
Attribute2
Operation1()
Operation2()

Virtual Machine 1

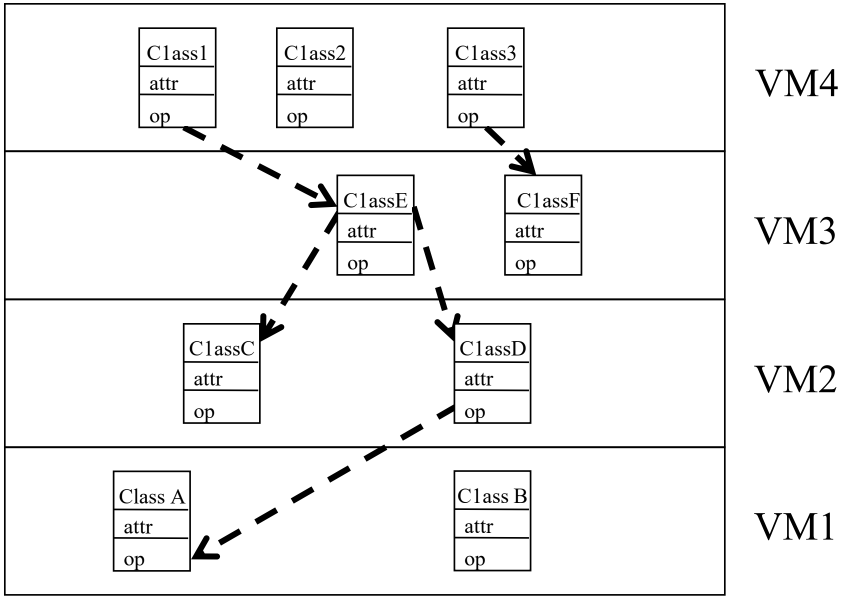
Operating System, Libraries

Existing System

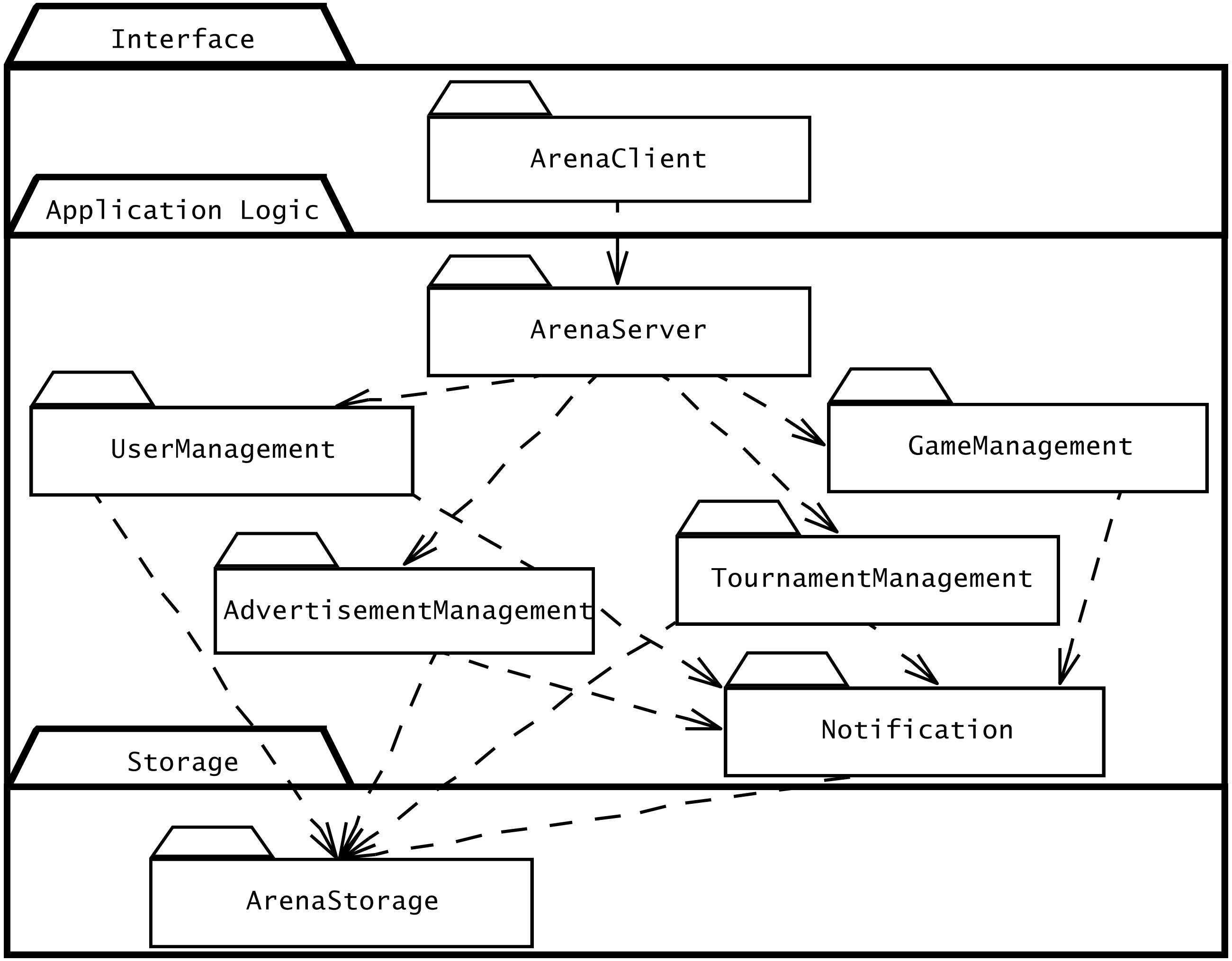
Closed Architecture (Opaque Layering)

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

Design goals: Maintainability, flexibility.



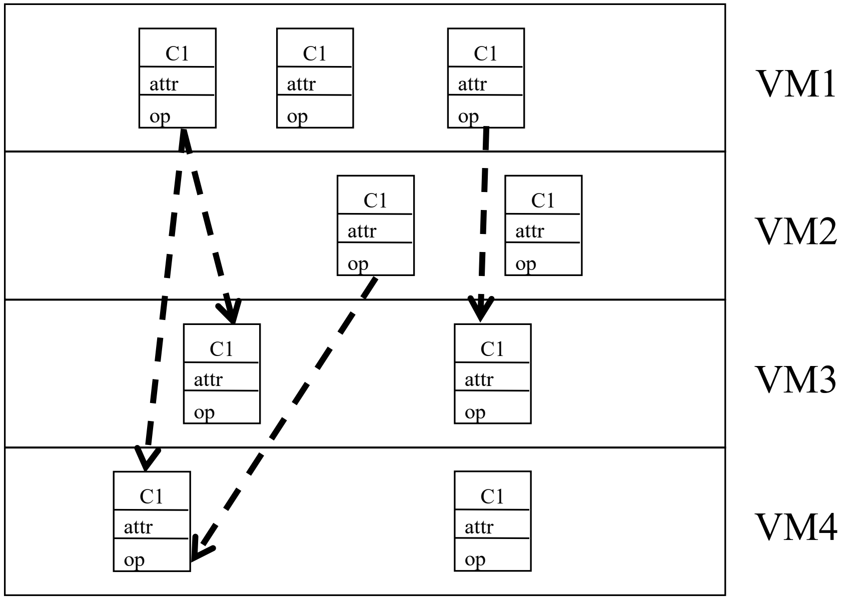
Opaque Layering in ARENA



Open Architecture (Transparent Layering)

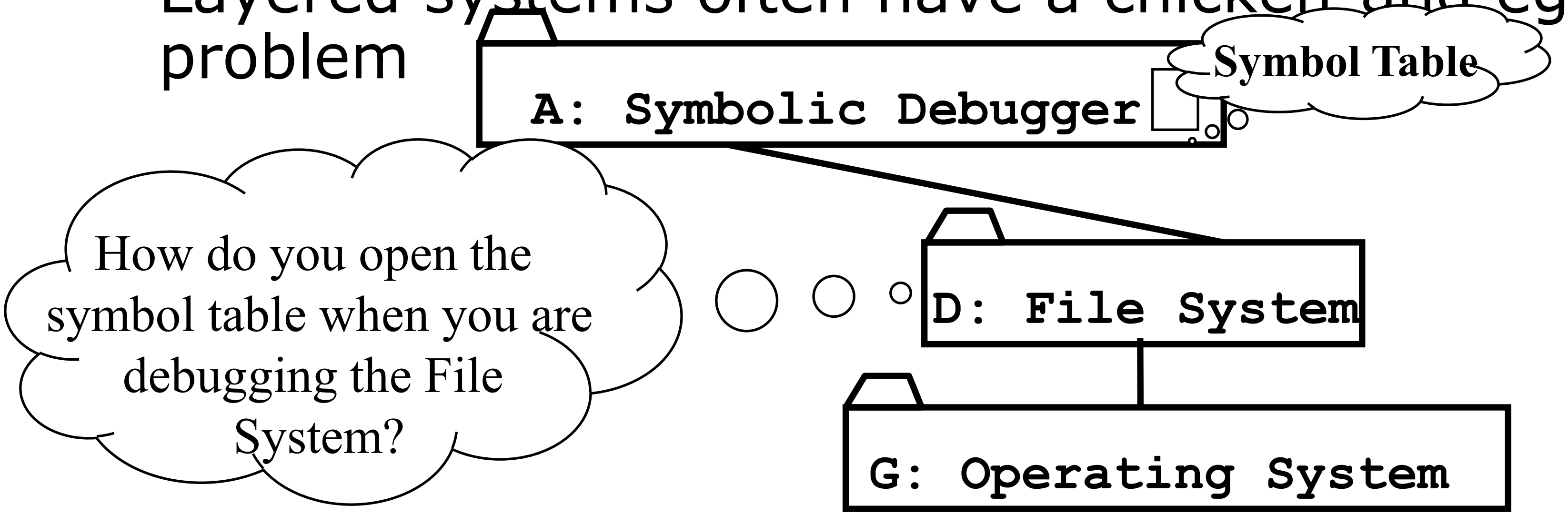
- Each virtual machine can call operations from any layer below

Design goal:
Runtime efficiency



Properties of Layered Systems

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


Symbol Table
A: Symbolic Debugger
How do you open the symbol table when you are debugging the File System?
D: File System
G: Operating System

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 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 subystem.
  - 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, *1941,
Developed the concept of
modularity in design.

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

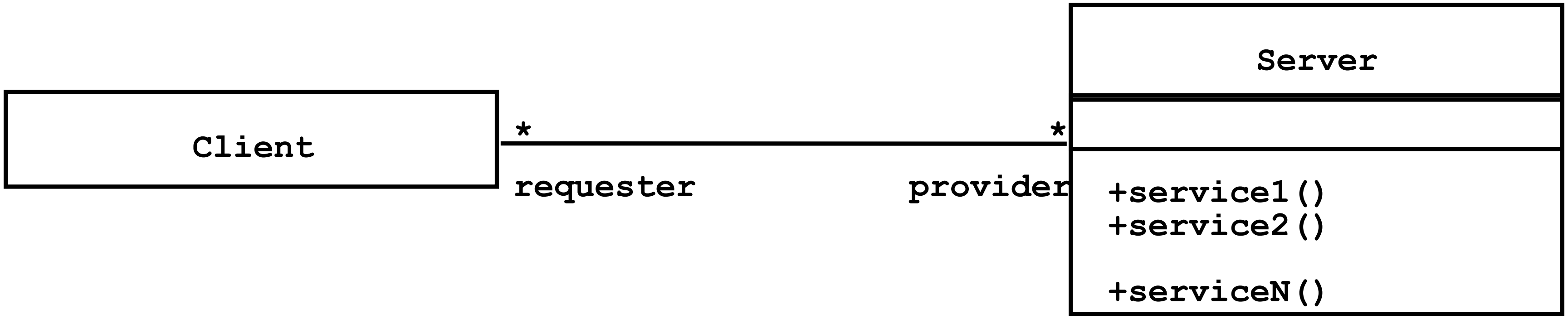
- Client/Server
- Peer-To-Peer
- Repository
- Model/View/Controller
- Three-tier, Four-tier Architecture
- Service-Oriented Architecture (SOA)
- Pipes and Filters

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-intensive operations

Scalability

Server can handle large # of clients

Flexibility

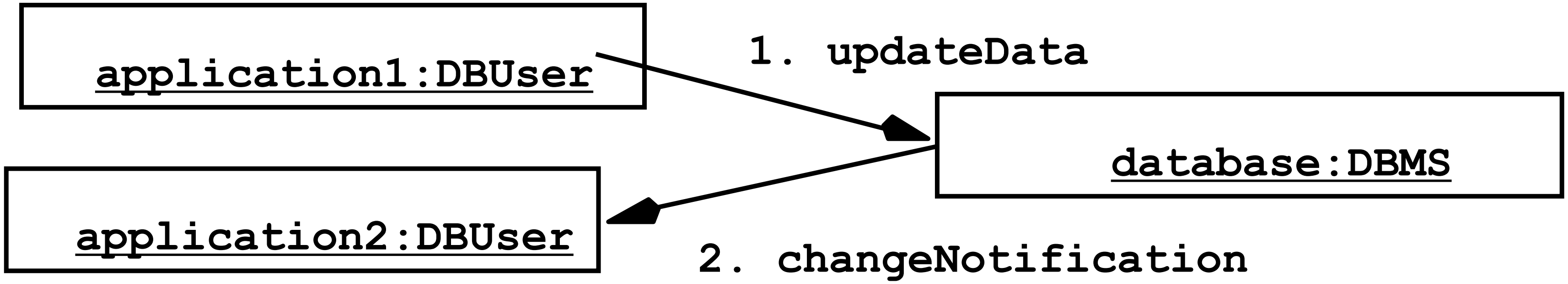
User interface of client supports a variety of end devices (PDA, Handy, laptop, wearable computer)

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 application and should be able to send notifications to the application when data have changed

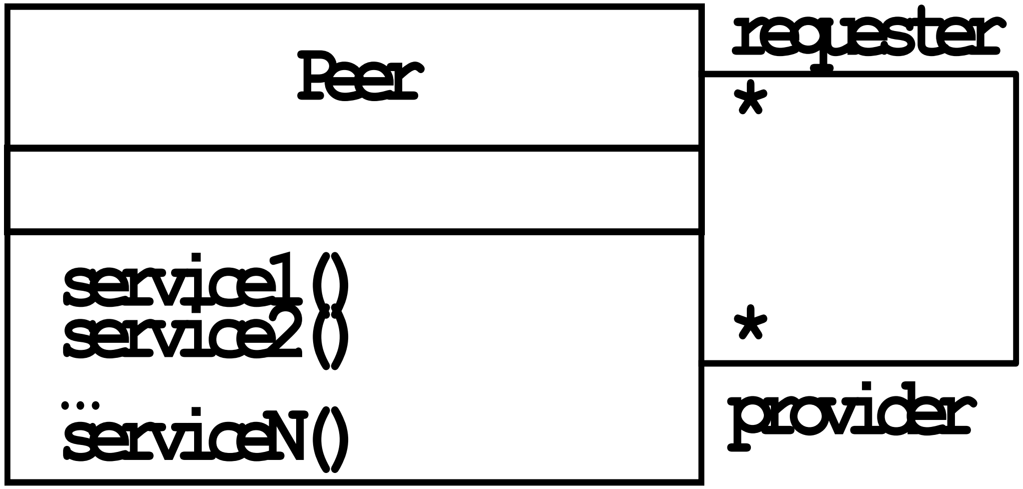


Peer-to-Peer Architectural Style

- **Specilization** of Client/Server Architectural Style
  "Clients can be servers and servers can be clients"
  - Introduction a new abstraction: Peer can be both "Clients and servers"
  - 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 a client as well as a server".

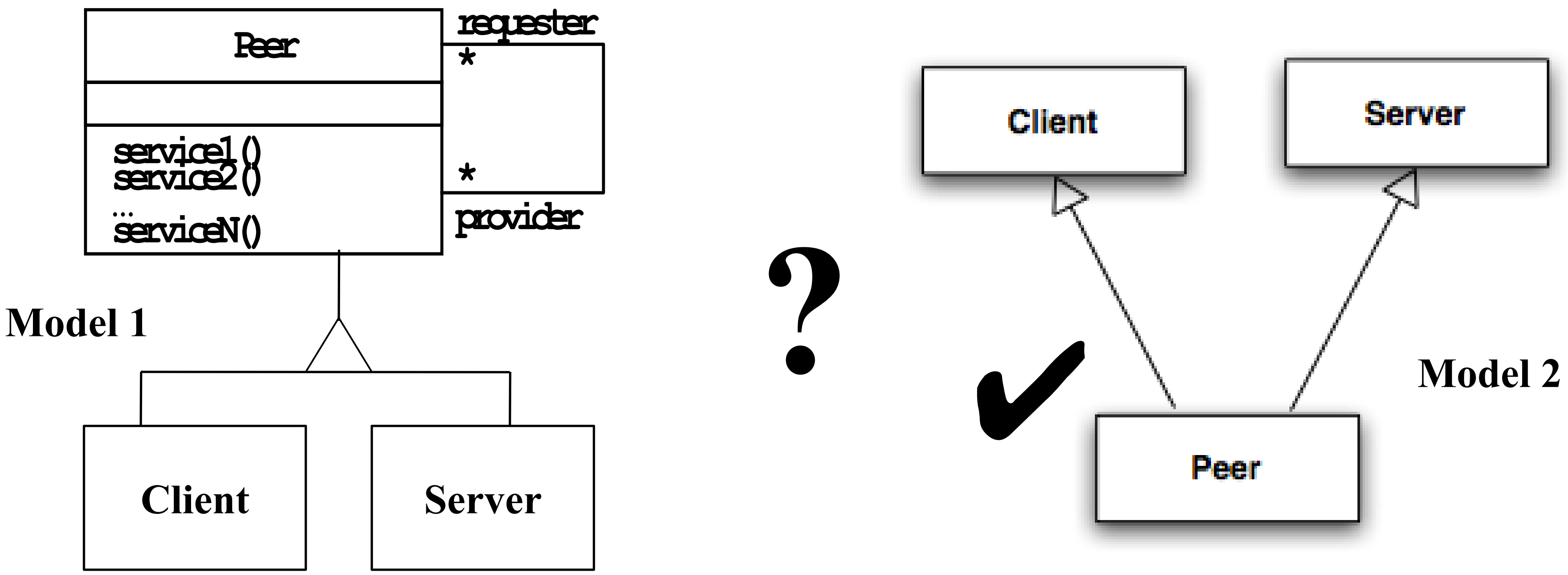


Relationship Client/Server & Peer-to-Peer

- Problem statement "Clients can be servers and servers can be clients"
- Which model is correct?

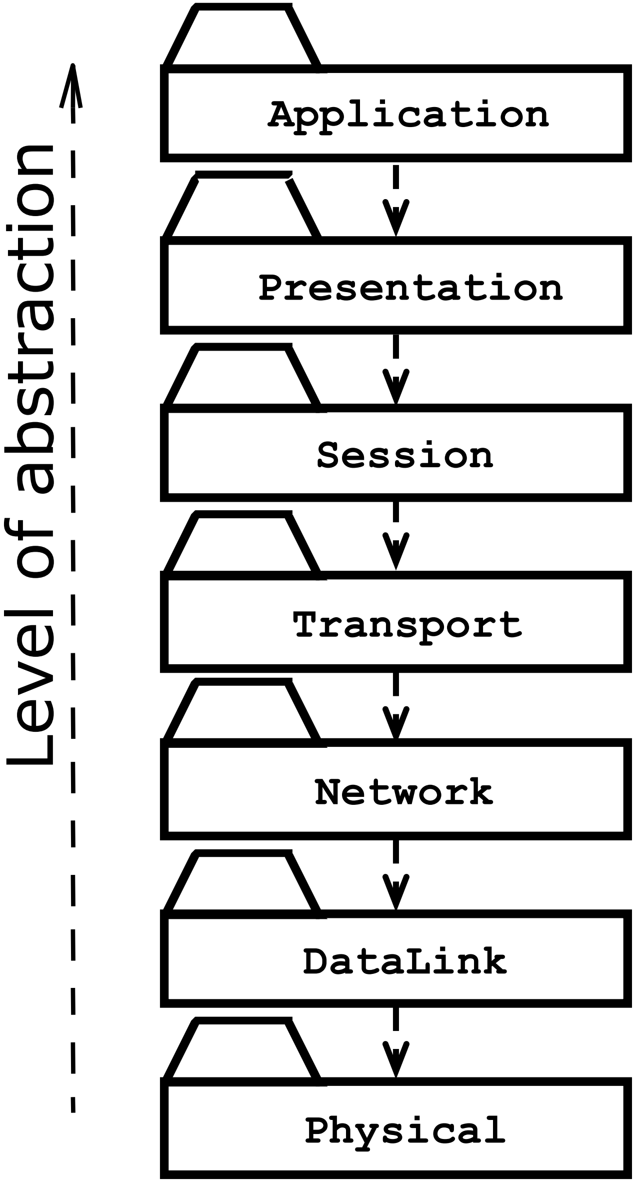
Model 1: "A peer can be either a client or a server"

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



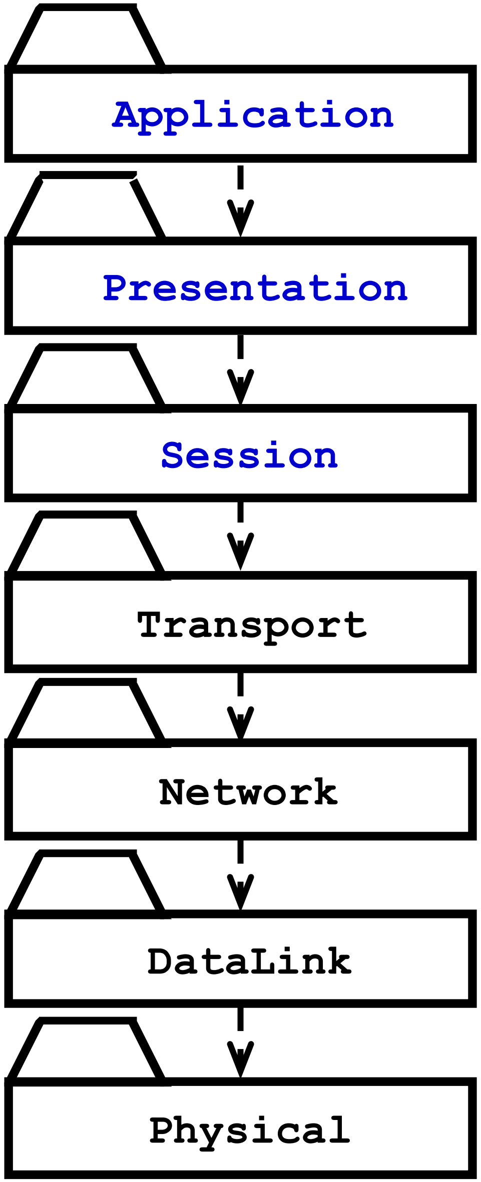
Example: Peer-to-Peer Architectural Style

- 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



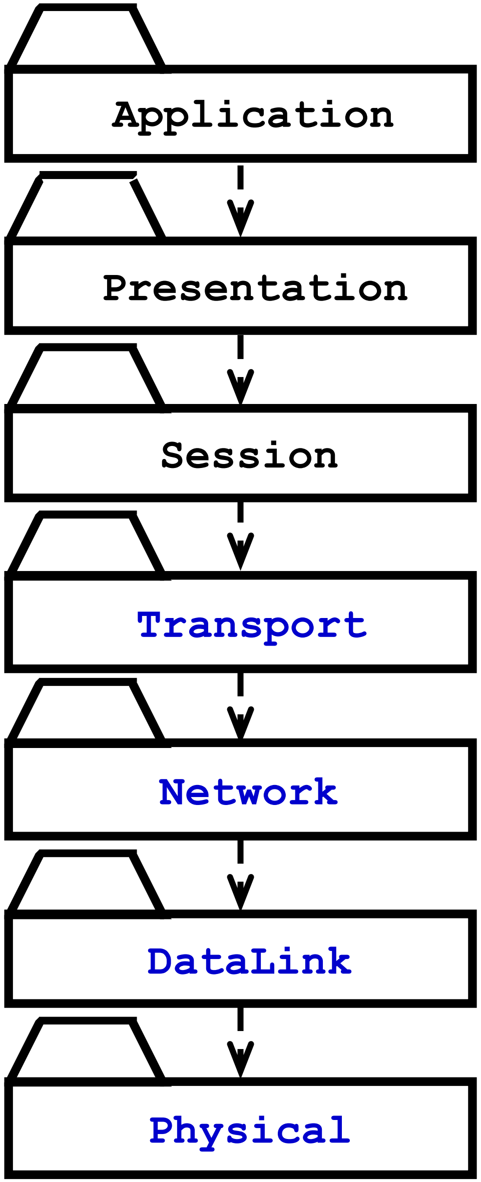
OSI Model Layers and Services

- 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

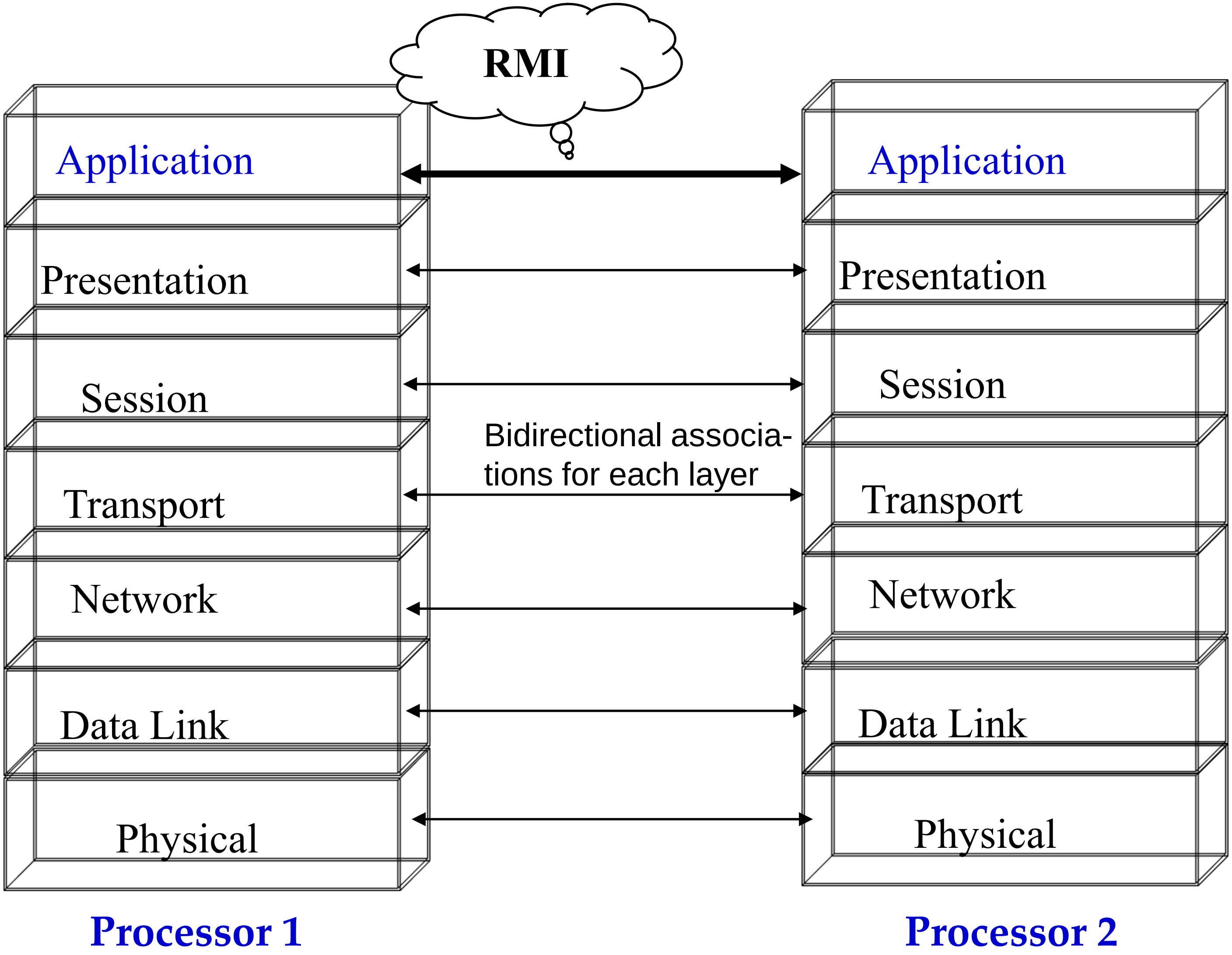


OSI Model Layers and their Services

- 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()

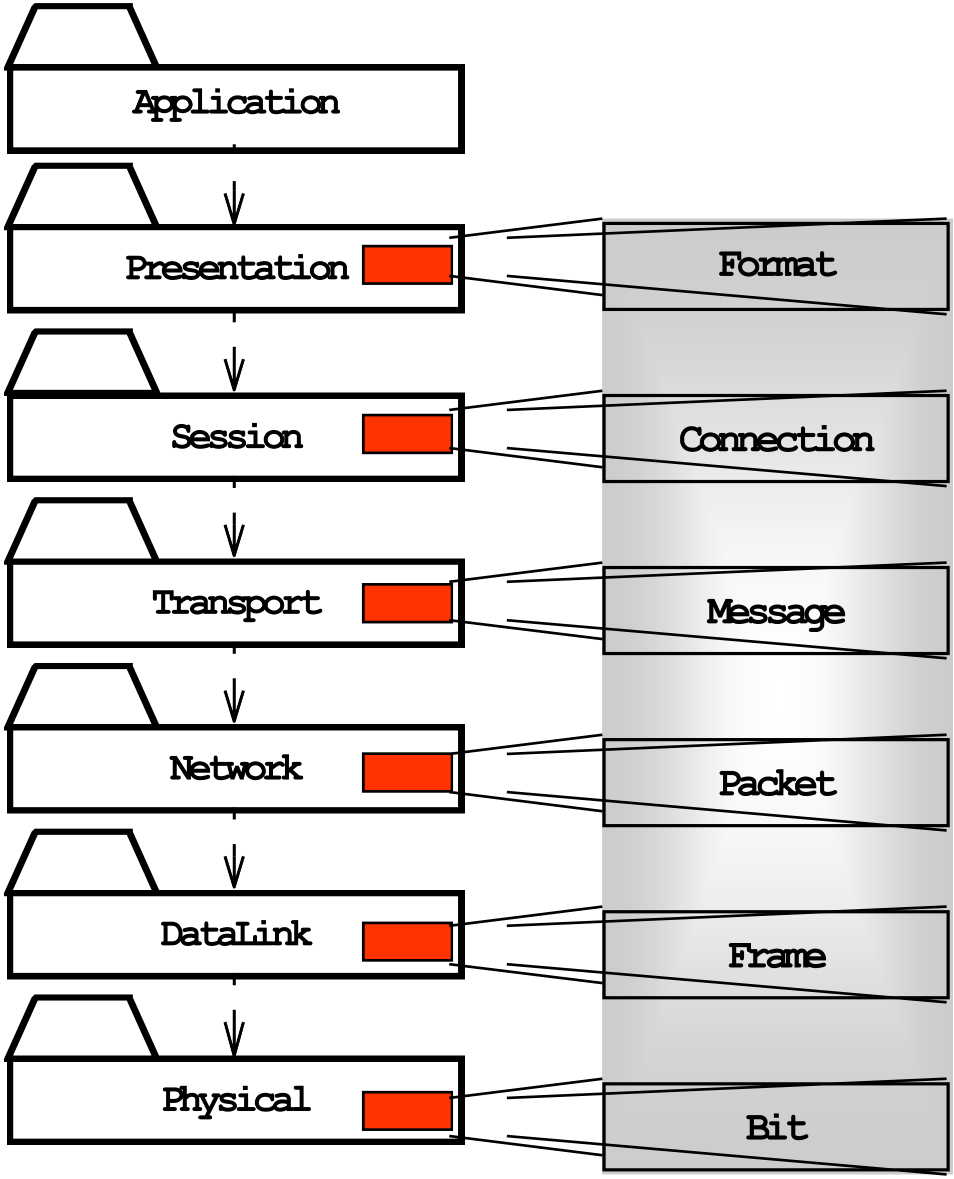


The Application Layer Provides the Abstractions of the “New System”

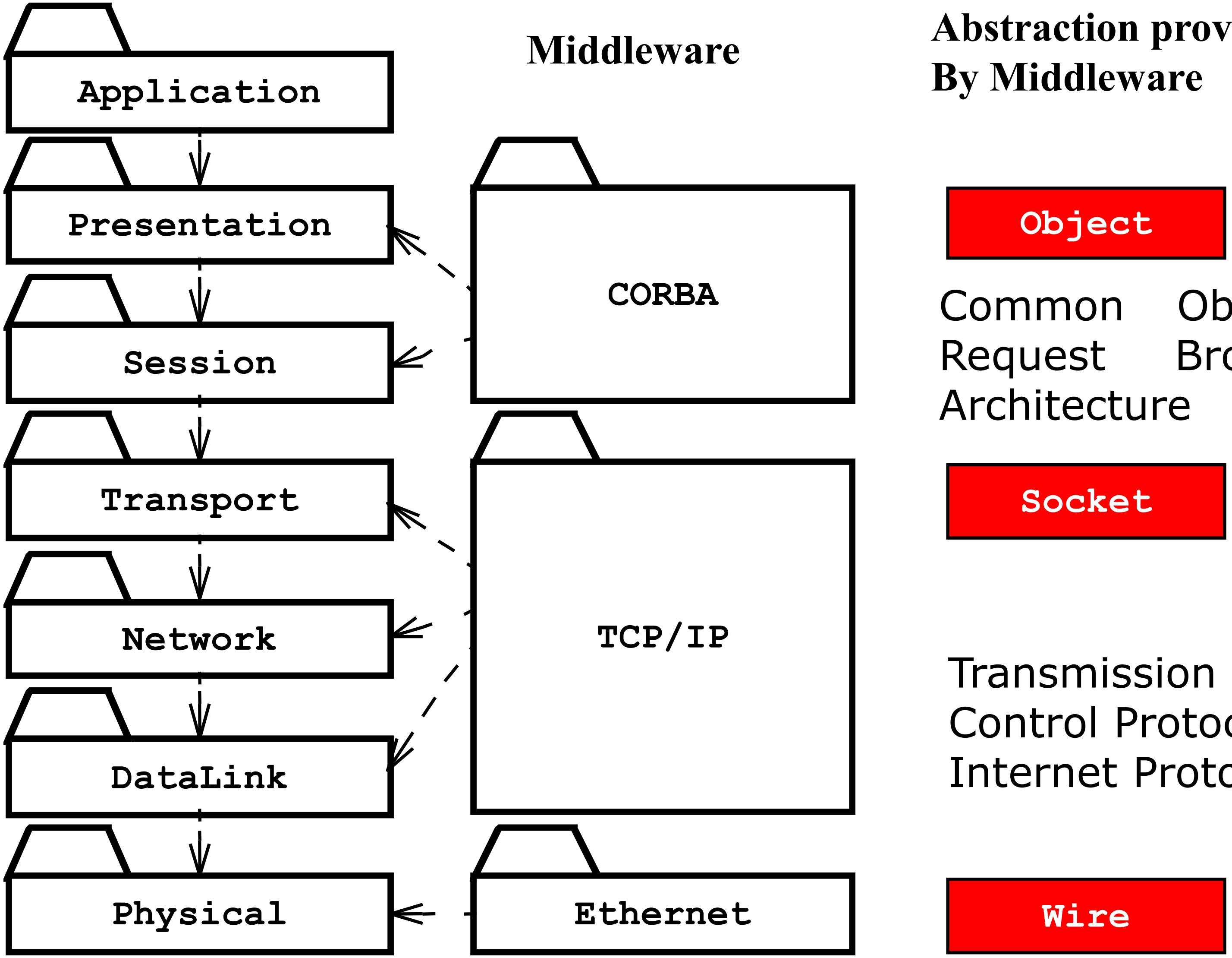


An Object-Oriented View of the OSI Model

- The OSI Model is a closed software architecture (i.e., it uses opaque layering)
- Each layer can be modeled as a UML package containing a set of classes available for the layer above



Middleware Allows Focus On Higher Layers

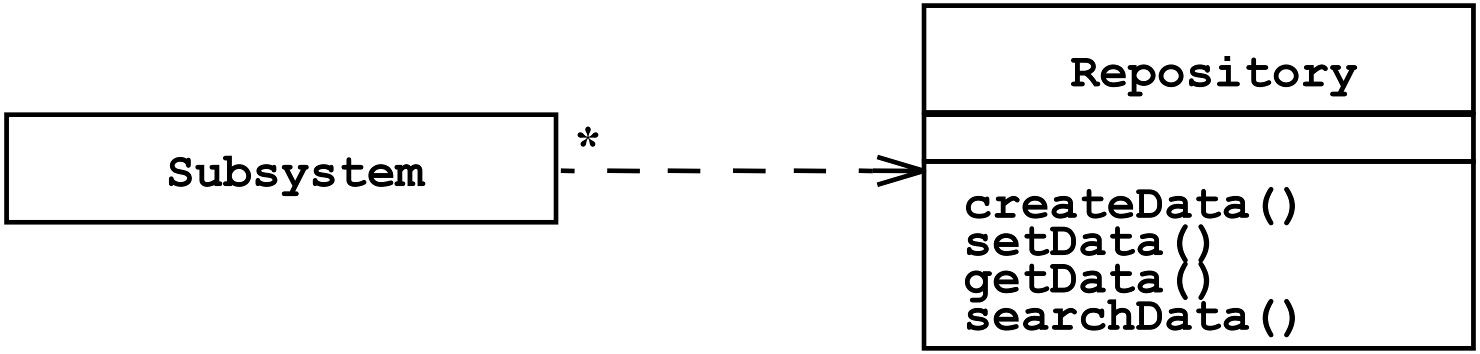


Common Object Request Broker Architecture

Transmission Control Protocol / Internet Protocol

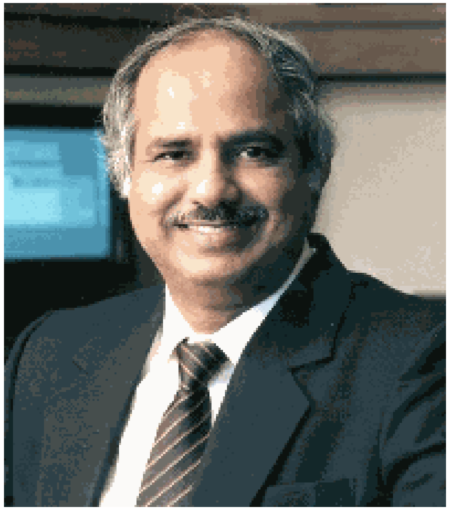
Repository Architectural Style

- Subsystems access and modify data from a single data structure called the repository
- Historically called blackboard architecture (Erman, Hayes-Roth and Reddy 1980)
- 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



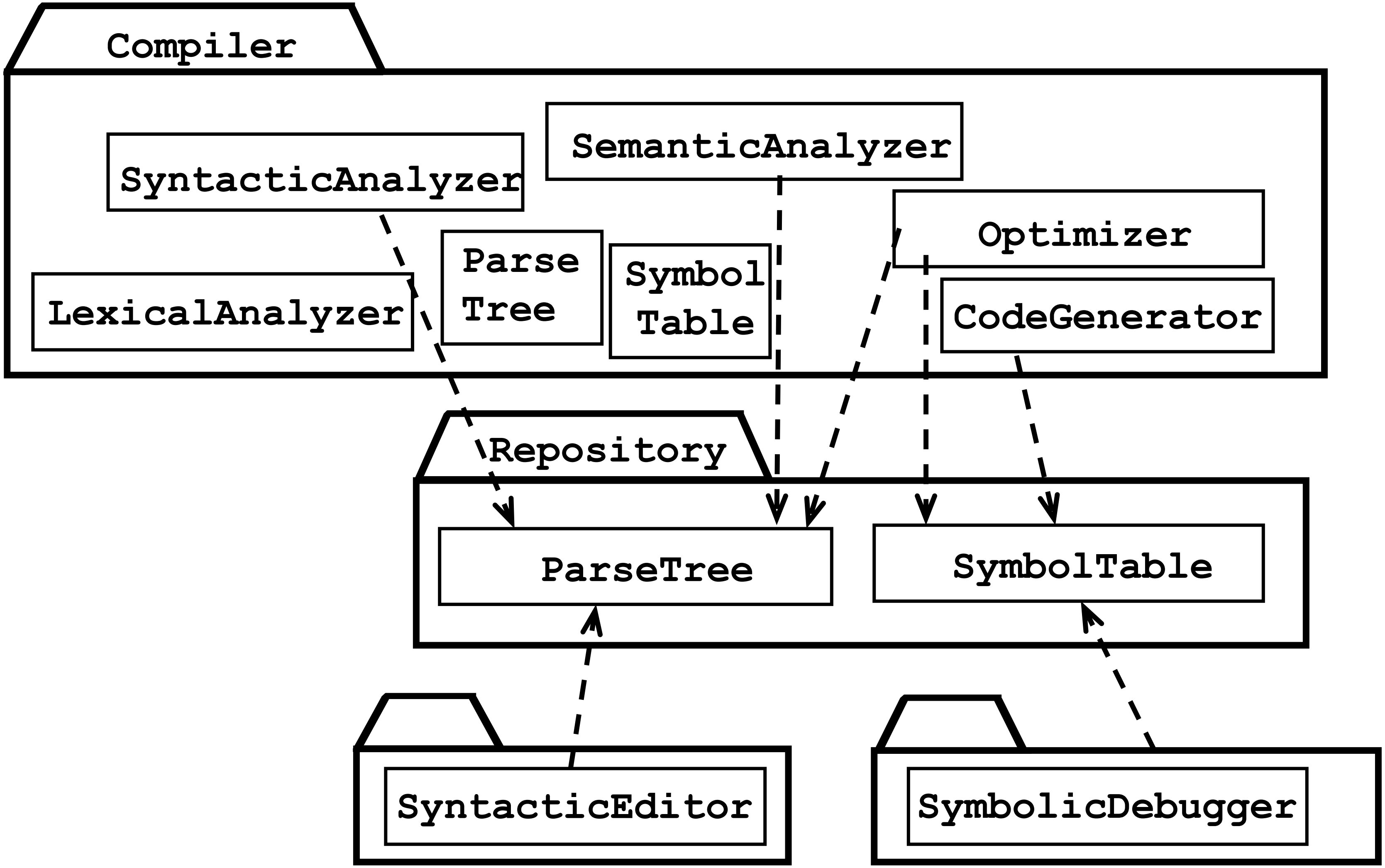
Blackboard Subsystem Decomposition

- 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, *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: Incremental Development Environment (IDE)



Providing Consistent Views

- 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 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 Controller subsystem mediates between View (data presentation) and Model (data access)
- Often called MVC.

Model-View-Controller Architectural Style

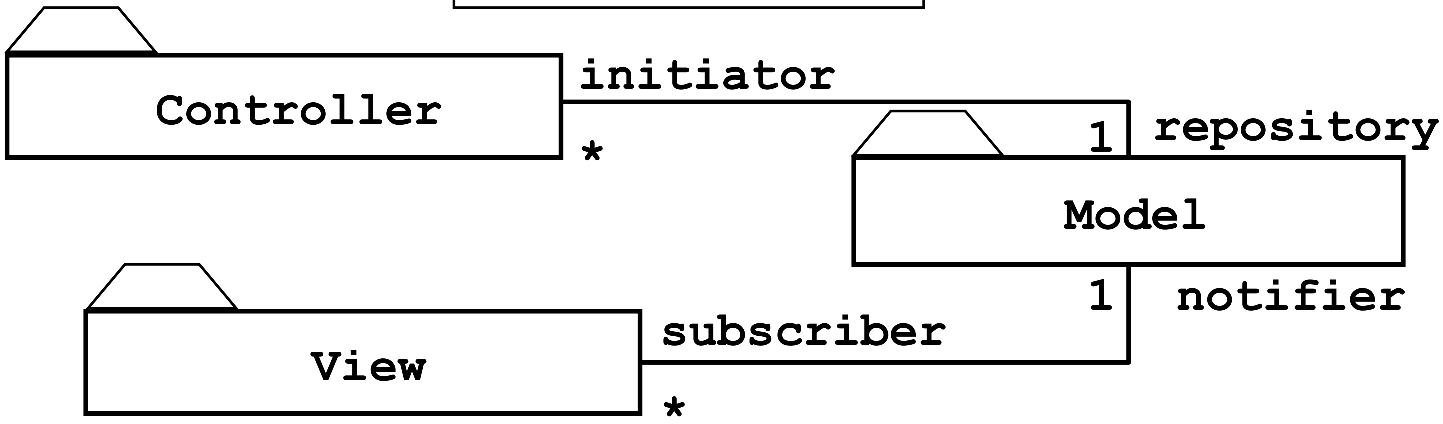
- Subsystems are classified into 3 different types

  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

Class Diagram



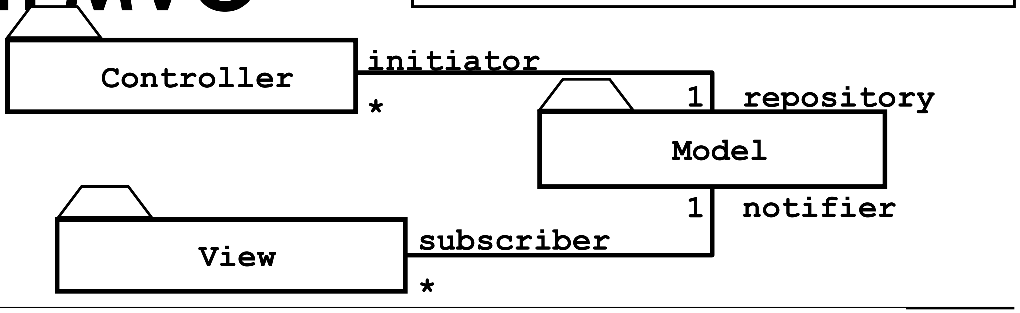
Better understanding with a Collaboration Diagram

UML Collaboration Diagram

- A Collaboration Diagram is an instance diagram that 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
    - Collaboration 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 chronological number and placed near the link the message is sent over
- Reading a collaboration diagram involves starting at message 1.0, and following the messages from object to object.

Example: Modeling the Sequence of Events in MVC

UML Class Diagram



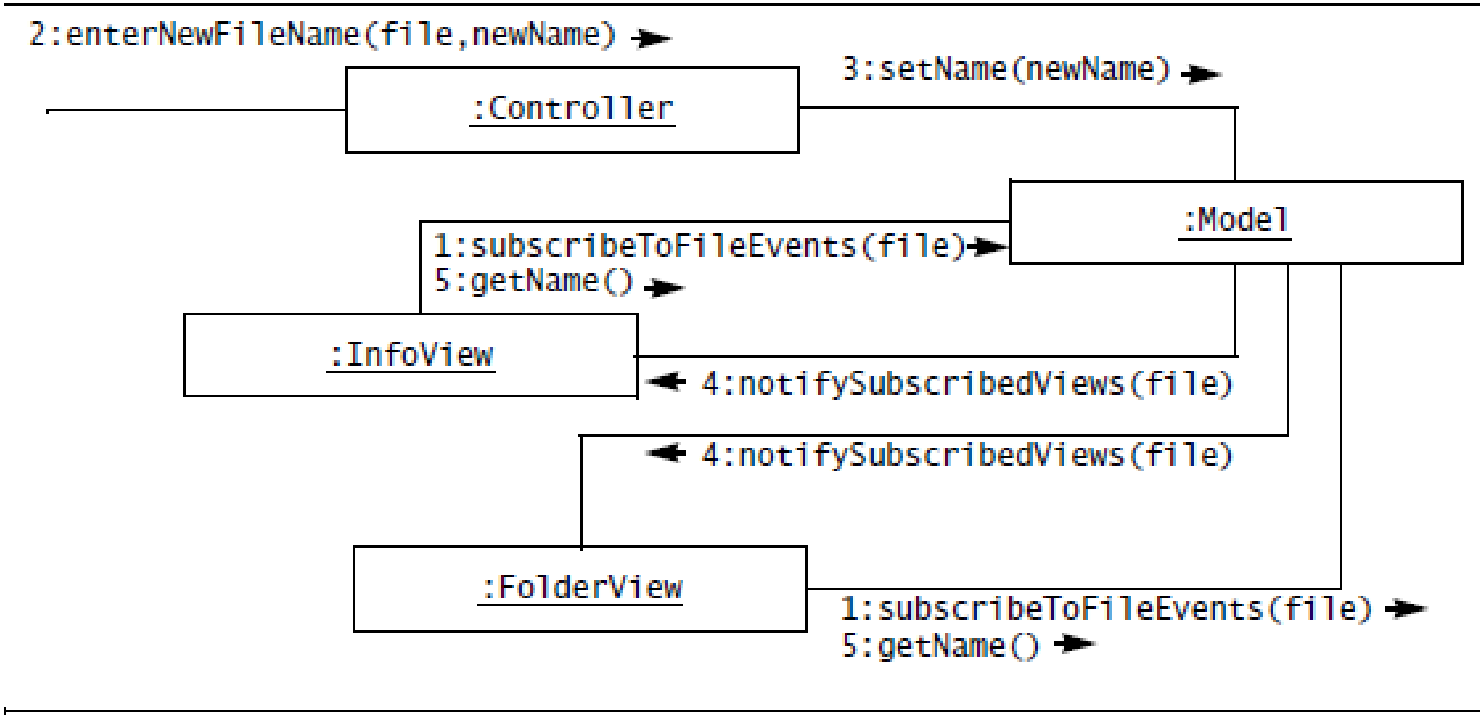

2:enterNewFileName(file,newName)
3:setName(newName)
:Controller
:Model
1:subscribeToFileEvents(file)
5:getName()
:InfoView
4:notifySubscribedViews(file)
4:notifySubscribedViews(file)
:FolderView
1:subscribeToFileEvents(file)
5:getName()

Figure 6-17 Sequence of events in the Model/View/Control architectural style (UML communication diagram).

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

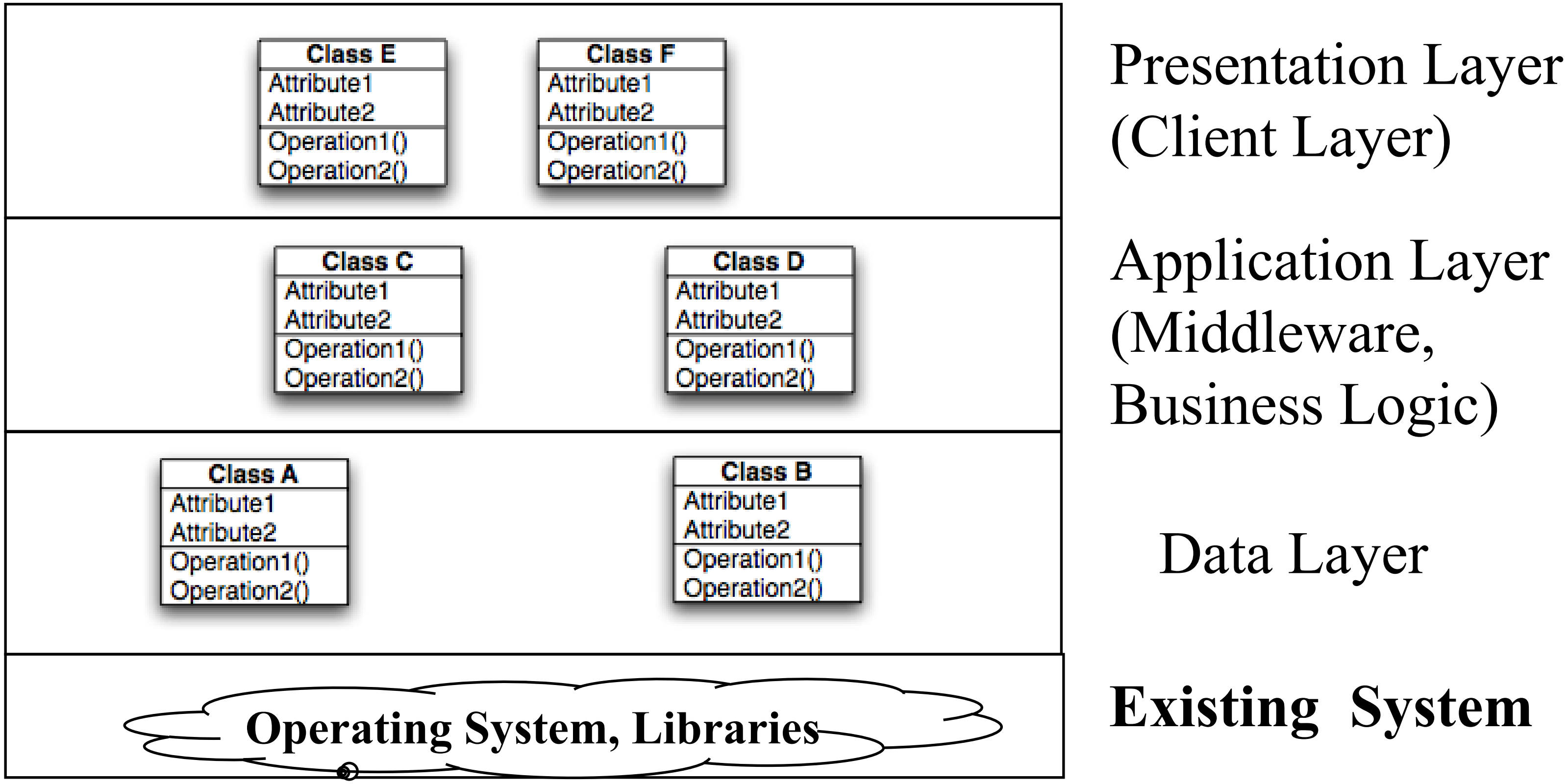
Definition: 3-Tier Architecture

- 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)
- Layer and Tier 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 manages and provides access to the persistent data.

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:

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 current 4-tier architectures, this is usually a relational Database management system (RDBMS).

MVC vs. 3-Tier Architectural Style

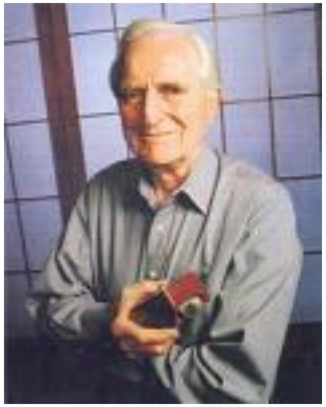
- The MVC architectural style is nonhierarchical (triangular):
  - View subsystem sends updates to the Controller subsystem
  - Controller subsystem updates the Model subsystem
  - View subsystem is updated directly from the Model subsystem
- The 3-tier architectural style is hierarchical (linear):
  - The presentation layer never communicates directly with the data layer (opaque architecture)
  - All communication must pass through the middleware layer
- History:
  - 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.

History at Xerox Parc

Xerox PARC (Palo Alto Research Center)

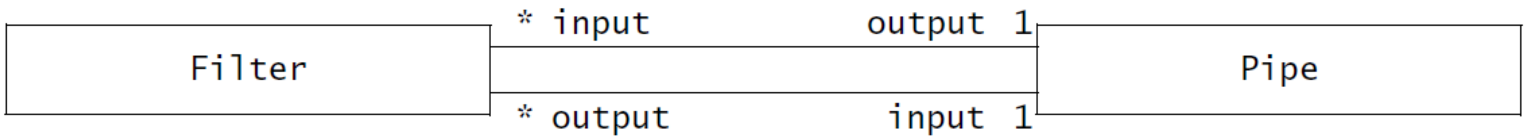
Founded in 1970 by Xerox, since 2002 a separate company PARC (wholly owned by Xerox). Best known for the invention of:

- Laser printer (1973, Gary Starkweather),
- Ethernet (1973, Bob Metcalfe),
- Modern personal computer (1973, Alto, Bravo),
- Graphical user interface (GUI) based on WIMP,
  - Windows, icons, menus and pointing device
    - Based on Doug Engelbart´s invention of the mouse in 1965,
- Object-oriented programming (Smalltalk, 1970s, Adele Goldberg),
- Ubiquitous computing (1990, Mark Weiser).



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.



Pipe and filter architectural style. A Filter can have many inputs and outputs. A Pipe connects one of the outputs of a Filter to one of the inputs of another Filter.

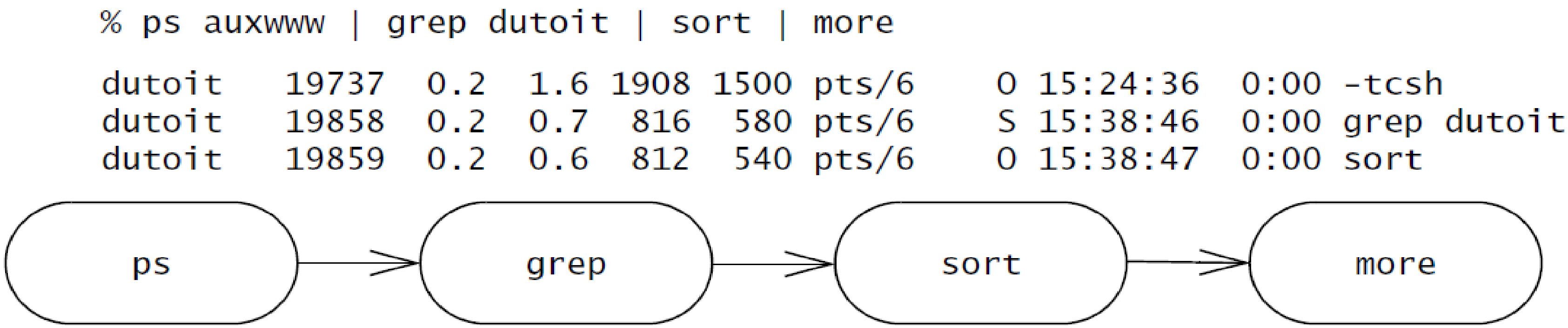
Pipes and Filters Architectural Style

- 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

Pipes and Filters example



```
% ps auxwww | grep dutoit | sort | more

dutoit   19737  0.2  1.6 1908 1500 pts/6    O 15:24:36  0:00 -tcsh
dutoit   19858  0.2  0.7  816  580 pts/6    S 15:38:46  0:00 grep dutoit
dutoit   19859  0.2  0.6  812  540 pts/6    O 15:38:47  0:00 sort
```

Unix command line as an instance of the pipe and filter style (UML activity diagram).

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.

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.