DESIGN

“There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.”

- C.A.R. Hoare

Which would be more difficult?

WHY IS DESIGN SO DIFFICULT?

- *Analysis:* Focuses on the application domain

- *Design:* Focuses on the solution domain
  - Design knowledge is a moving target
  - The reasons for design decisions are changing very rapidly
    - Halftime knowledge in software engineering: About 3-5 years
    - What I teach today will be out of date in 3 years
    - Cost of hardware rapidly sinking

- "Design window":
  - Time in which design decisions have to be made

- Technique
  - Time-boxed prototyping

The "evolutionary rapid development" process focuses on the use of small artisan-based teams integrating software and systems engineering disciplines working multiple, often parallel short-duration *timeboxes* with frequent customer interaction. …*reuse of architectural components* …

OVERVIEW

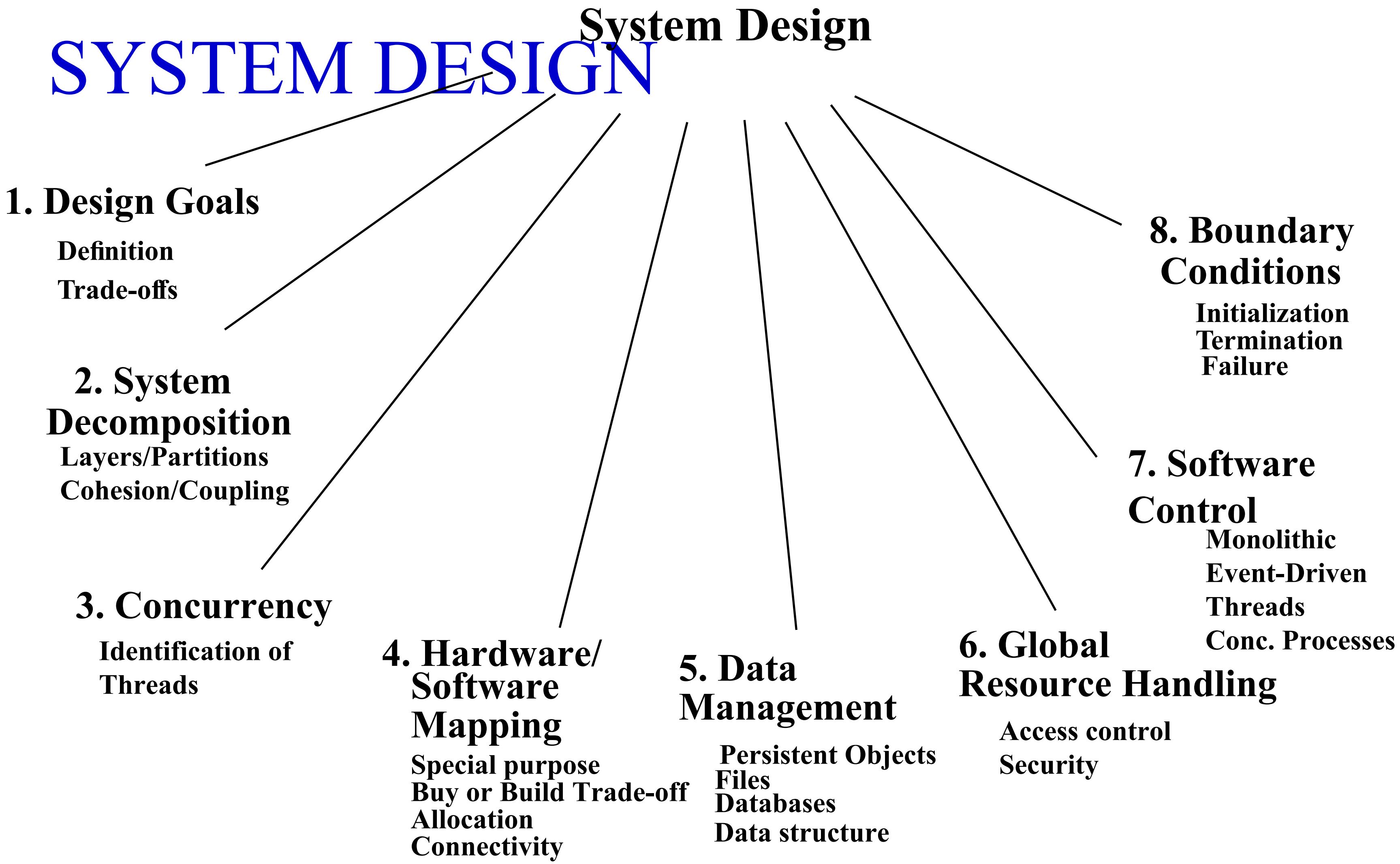
System Design I (Today)

0. Overview of System Design
1. Design Goals
2. Subsystem Decomposition

System Design II: Addressing Design Goals (next lecture)

3. Concurrency
4. Hardware/Software Mapping
5. Persistent Data Management
6. Global Resource Handling and Access Control
7. Software Control
8. Boundary Conditions

SYSTEM DESIGN



System Design

1. Design Goals
   - Definition
   - Trade-offs

2. System Decomposition
   - Layers/Partitions
   - Cohesion/Coupling

3. Concurrency
   - Identification of Threads

4. Hardware/ Software Mapping
   - Special purpose
   - Buy or Build Trade-off
   - Allocation
   - Connectivity

5. Data Management
   - Persistent Objects
   - Files
   - Databases
   - Data structure

6. Global Resource Handling
   - Access control
   - Security

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

8. Boundary Conditions
   - Initialization
   - Termination
   - Failure

HOW TO USE THE RESULTS FROM THE REQUIREMENTS ANALYSIS FOR SYSTEM DESIGN

- Nonfunctional requirements =>
  - Activity 1: Design Goals Definition
- Functional model =>
  - Activity 2: System decomposition (Selection of subsystems based on functional requirements, cohesion, and coupling)
- Object model =>
  - Activity 4: Hardware/software mapping
  - Activity 5: Persistent data management
- Dynamic model =>
  - Activity 3: Concurrency
  - Activity 6: Global resource handling
  - Activity 7: Software control
- Subsystem Decomposition
  - Activity 8: Boundary conditions

LIST OF DESIGN GOALS

- Reliability
- Modifiability
- Maintainability
- Understandability
- Adaptability
- Reusability
- Efficiency
- Portability
- Traceability of requirements
- Fault tolerance
- Backward-compatibility
- Cost-effectiveness
- Robustness
- High-performance

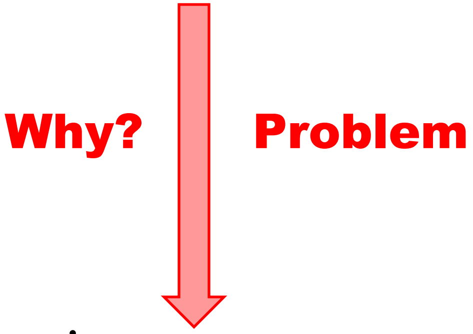
- Good documentation
- Well-defined interfaces
- User-friendliness
- Reuse of components
- Rapid development
- Minimum # of errors
- Readability
- Ease of learning
- Ease of remembering
- Ease of use
- Increased productivity
- Low-cost
- Flexibility

Are these exhaustive? Anything else?
What do we do with all these?

How do we get the Design Goals?

Let's look at a small example

- **Current Situation:**
  - Computers must be used in the office
- **What we want:**
  - A computer that can be used in mobile situations.

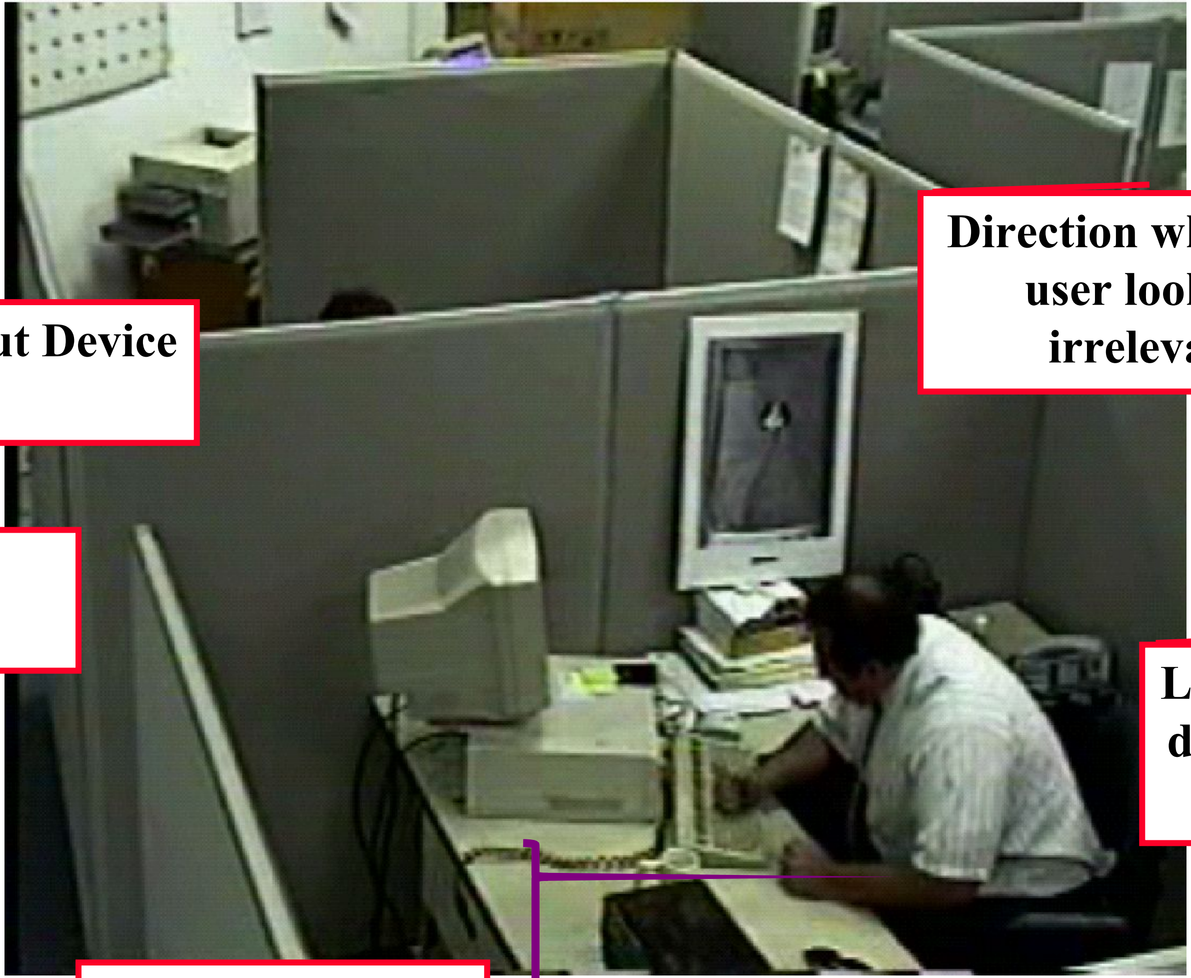


Why?

Problem

What are the technical terms describing the two?

Identify Current Technology Constraints



Single Output Device

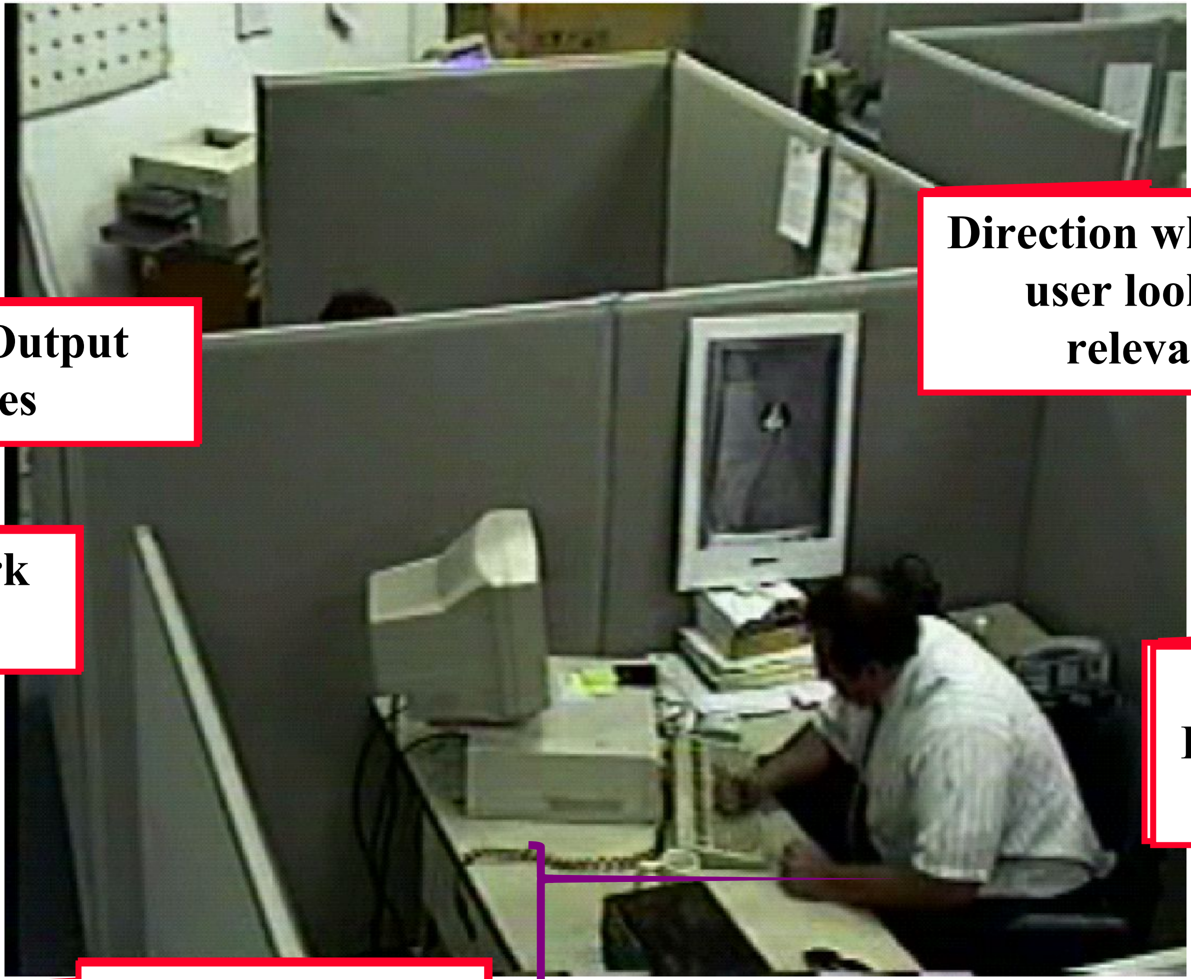
Direction where the user looks is irrelevant

Fixed Network Connection

Location of user does not matter

Precise Input

Generalize Constraints using Technology Enablers



Multiple Output Devices

Dynamic Network Connection

Vague Input

Direction where the user looks is relevant

Location-based

Any concrete scenarios?

Establish New Design Goals

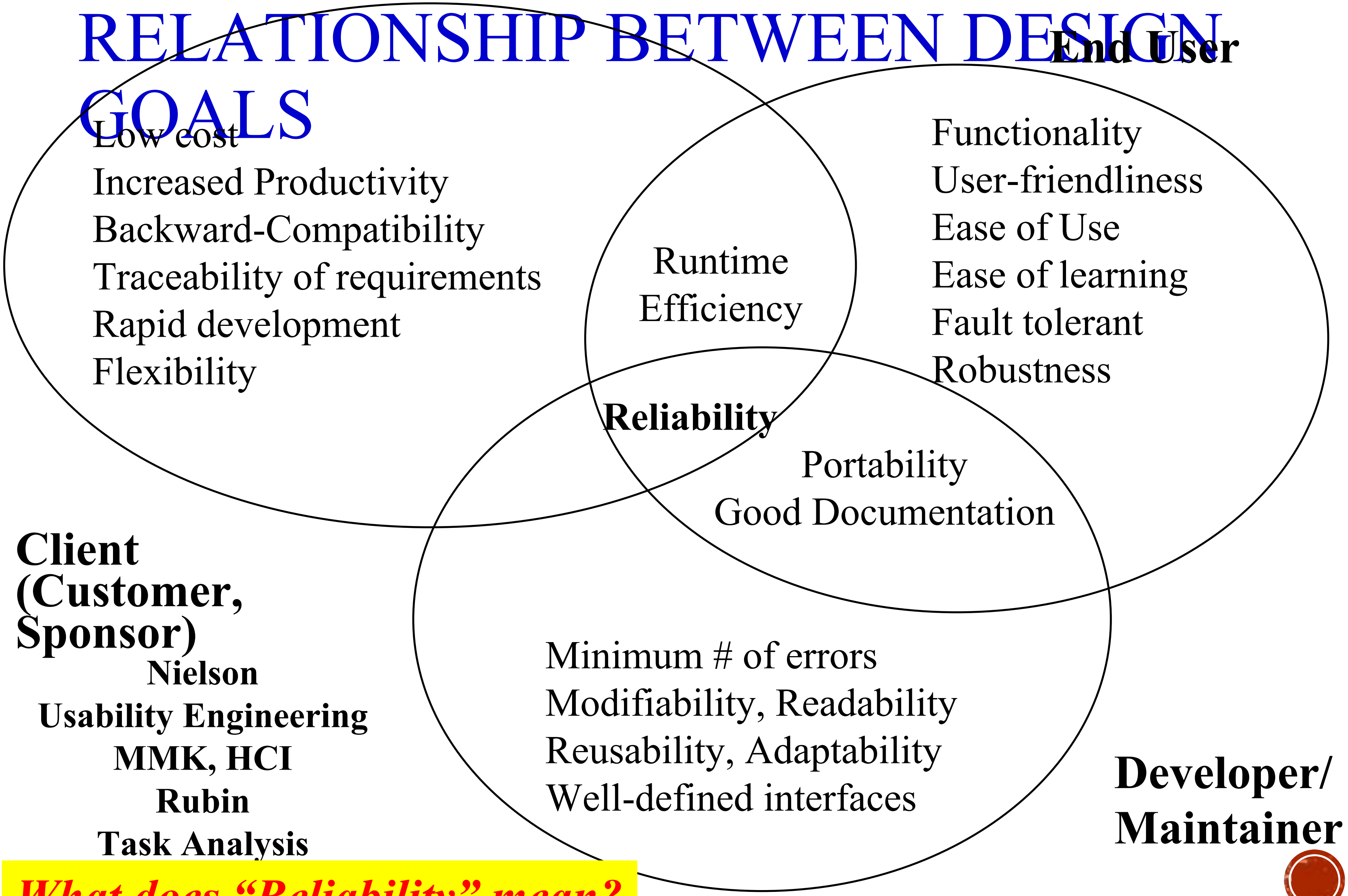
- **Mobile Network Connection**
- **Multiple Output Devices**
- **Location-Based**
- **Multimodal Input (Users Gaze, Users Location, …)**
- **Vague input**

Are these Requirements or Design?

Sharpen the Design Goals

- **Location-based input**
  - Input depends on user location
  - Input depends on the direction where the user looks ("egocentric systems")
- **Multi-modal input**
  - The input comes from more than one input device
- **Dynamic connection**
  - Contracts are only valid for a limited time
- **Is there a possibility of further generalizations?**
- **Example: location can be seen as a special case of *context***
  - User preference is part of the context
  - Interpretation of commands depends on context

RELATIONSHIP BETWEEN DESIGN GOALS



End User

Functionality
User-friendliness
Ease of Use
Ease of learning
Fault tolerant
Robustness

Low cost
Increased Productivity
Backward-Compatibility
Traceability of requirements
Rapid development
Flexibility

Runtime
Efficiency

Reliability

Portability
Good Documentation

Client (Customer, Sponsor)

Nielson
Usability Engineering
MMK, HCI
Rubin
Task Analysis

Minimum # of errors
Modifiability, Readability
Reusability, Adaptability
Well-defined interfaces

Developer/ Maintainer

What does "Reliability" mean?

TYPICAL DESIGN TRADE-OFFS

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

SECTION 2. SYSTEM DECOMPOSITION

- Subsystem (***UML: Package***)
  - Collection of classes, associations, operations, events and constraints that are interrelated
  - *Seed for subsystems: UML Objects and Classes*.
- (Subsystem) Service:
  - Group of operations provided by the subsystem
  - *Seed for services: Subsystem use cases*

From what spec.?

- Service is specified by Subsystem interface:
  - Specifies interaction and information flow from/to subsystem boundaries, but *not inside* the subsystem.
  - Should be well-defined and small.
  - Often called API: Application programmer's interface, but this term should used during implementation, not during System Design

COUPLING AND COHESION

- Goal: Reduction of ***complexity while change occurs***

- **Cohesion** measures the dependence among classes
  - High cohesion: The classes in the subsystem perform similar tasks and are related to each other (via associations)
  - Low cohesion: Lots of miscellaneous and auxiliary classes, no associations
- **Coupling** measures dependencies between subsystems
  - High coupling: Changes to one subsystem will have high impact on the other subsystem (change of model, massive recompilation, etc.)
  - Low coupling: A change in one subsystem does not affect any other subsystem
- Subsystems should have as **maximum** cohesion and **minimum** coupling as possible:

Can you illustrate these using UML conventions?

… and ???

PARTITIONS AND LAYERS

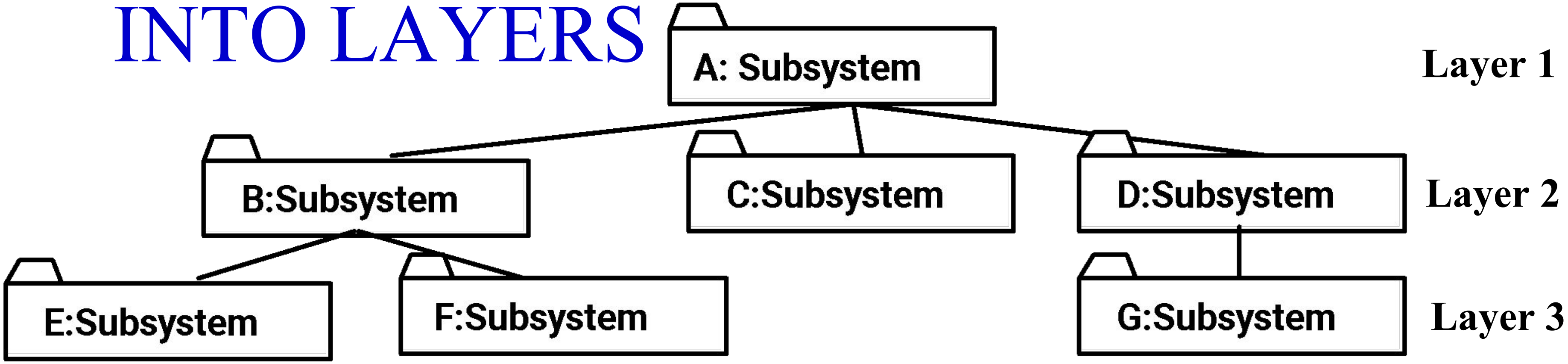
Partitioning and layering are techniques to achieve low coupling.

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 subsystem services to a higher layers (level of abstraction)
  - A layer can only depend on lower layers
  - A layer has no knowledge of higher layers

What are other architectural styles?

SUBSYSTEM DECOMPOSITION INTO LAYERS



Ideally use one package for each subsystem

- Subsystem Decomposition Heuristics:
- No more than 7+/-2 subsystems *Why?*
  - More subsystems increase cohesion but also complexity (more services)
- No more than 4+/-2 layers, use 3 layers (good)

Why?

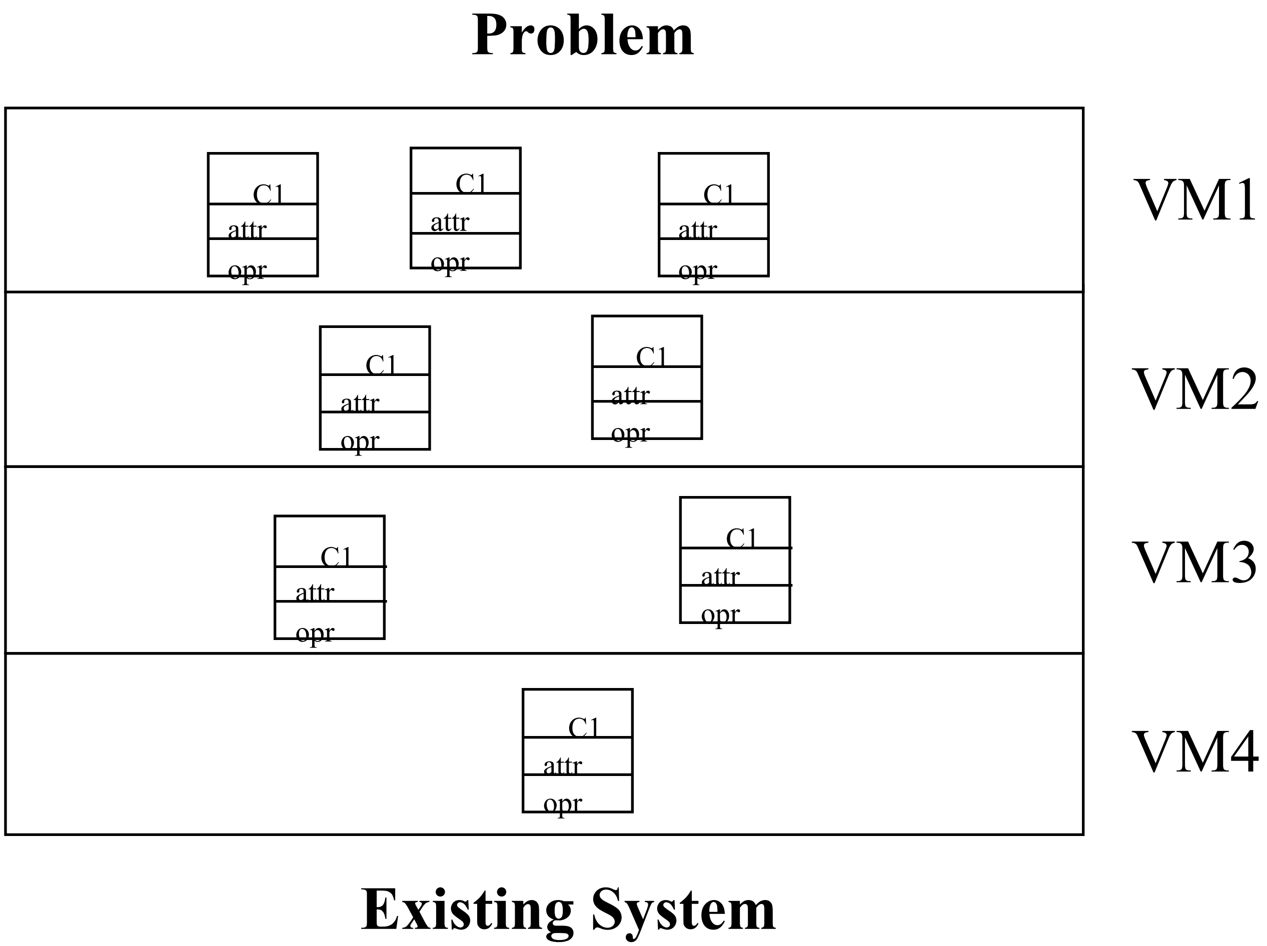
RELATIONSHIPS BETWEEN SUBSYSTEMS

- Layer relationship
  - Layer A “Calls” Layer B (runtime)
  - Layer A “Depends on” Layer B (“make” dependency, compile time)
- Partition relationship
  - The subsystem have mutual but not deep knowledge about each other
  - Partition A “Calls” partition B and partition B “Calls” partition A

Actually, this will depend on the directionality?

VIRTUAL MACHINE

- Dijkstra: T.H.E. operating system (1965)
  - A system should be developed by an ordered set of virtual machines, each built in terms of the ones below it.



Problem

C1
attr
opr

C1
attr
opr

C1
attr
opr

VM1

C1
attr
opr

C1
attr
opr

VM2

C1
attr
opr

C1
attr
opr

VM3

C1
attr
opr

VM4

Existing System

VIRTUAL MACHINE

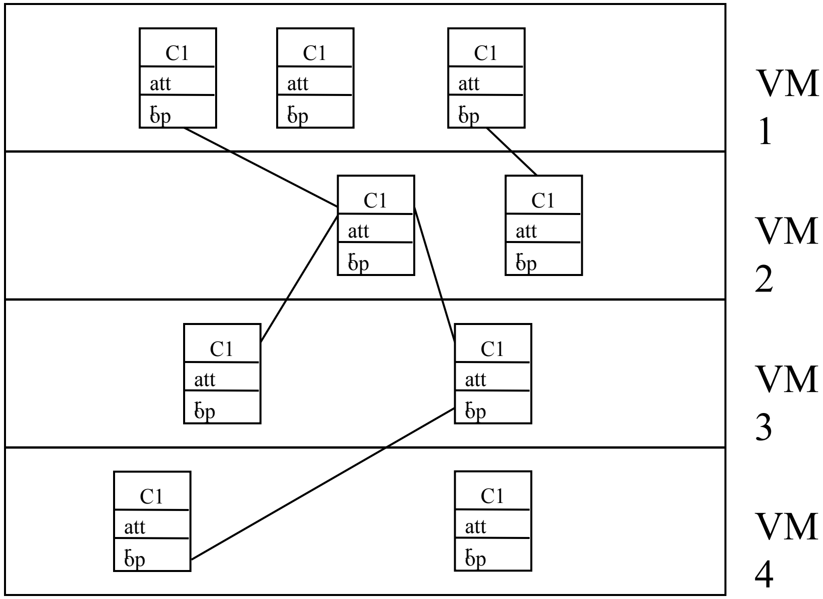
- A virtual machine is an abstraction
  - It provides a set of attributes and operations.
- A virtual machine is a subsystem
  - It is connected to higher and lower level virtual machines by "provides services for" associations.

How do we represent this in UML?

- Virtual machines can implement two types of software architecture
  - Open and closed architectures.

CLOSED ARCHITECTURE (OPAQUE LAYERING)

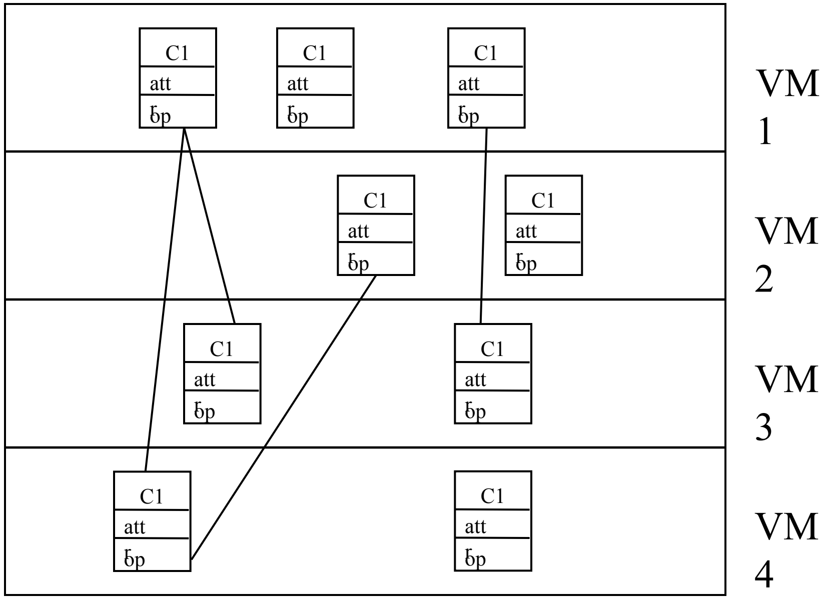
- Any layer can only invoke operations from the immediate layer below
- Design goal: **High maintainability, flexibility**



Only vertical communications?

OPEN ARCHITECTURE (TRANSPARENT LAYERING)

- Any layer can invoke operations from any layers below
- Design goal: **Runtime efficiency**



PROPERTIES OF LAYERED SYSTEMS

- Layered systems are ***hierarchical***. They are desirable because hierarchy reduces complexity (by low coupling).

- Closed architectures are more portable.
- Open architectures are more efficient.

Why?

and what else?

So, which is better?

- If a subsystem is a layer, it is often called a virtual machine.

What are examples of systems using a layered architectural style?

SOFTWARE ARCHITECTURAL STYLES

- Subsystem decomposition
  - Identification of subsystems, services, and their relationship to each other.
- ***Specification*** of the system decomposition is critical.

Patterns = styles?

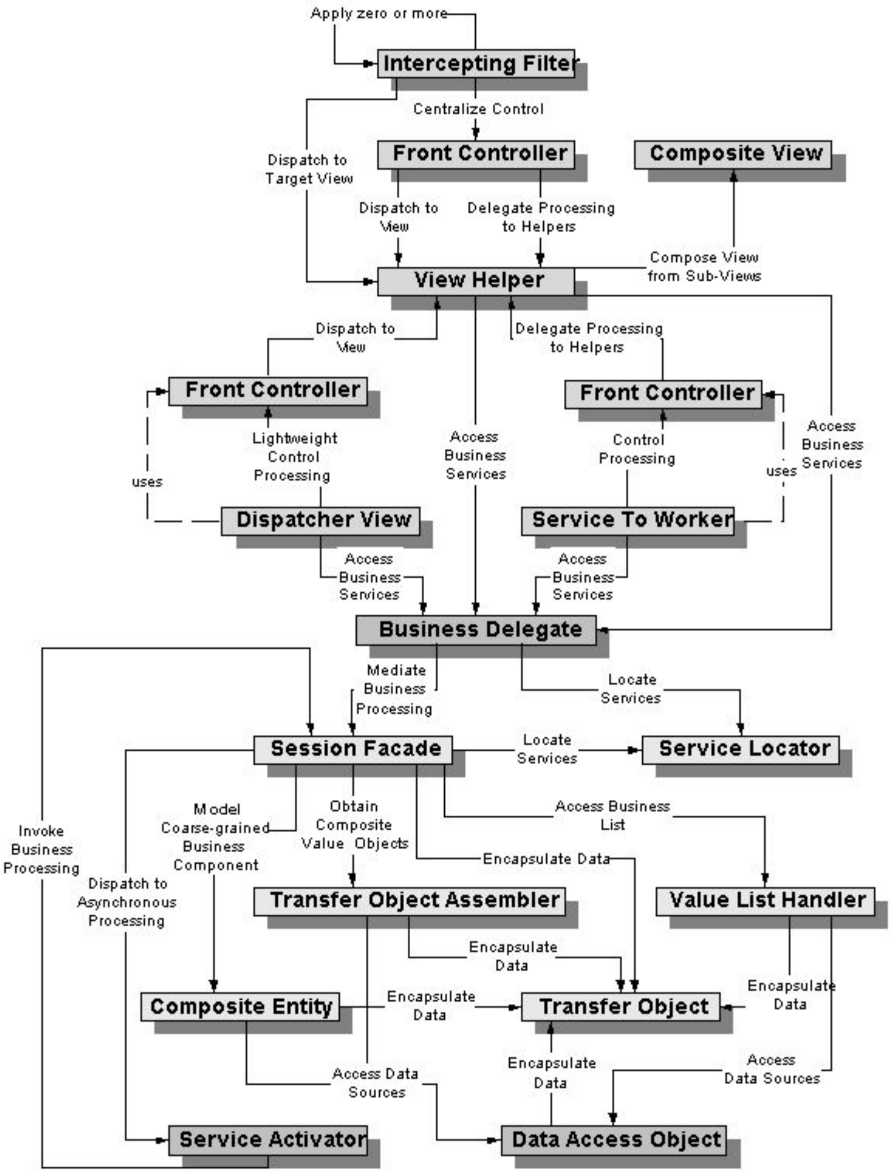
- Patterns for software architecture
  - Client/Server
  - Peer-To-Peer
  - Repository
  - Model/View/Controller
  - Pipes and Filters

Is this a J2EE pattern?

What are other architectural styles?

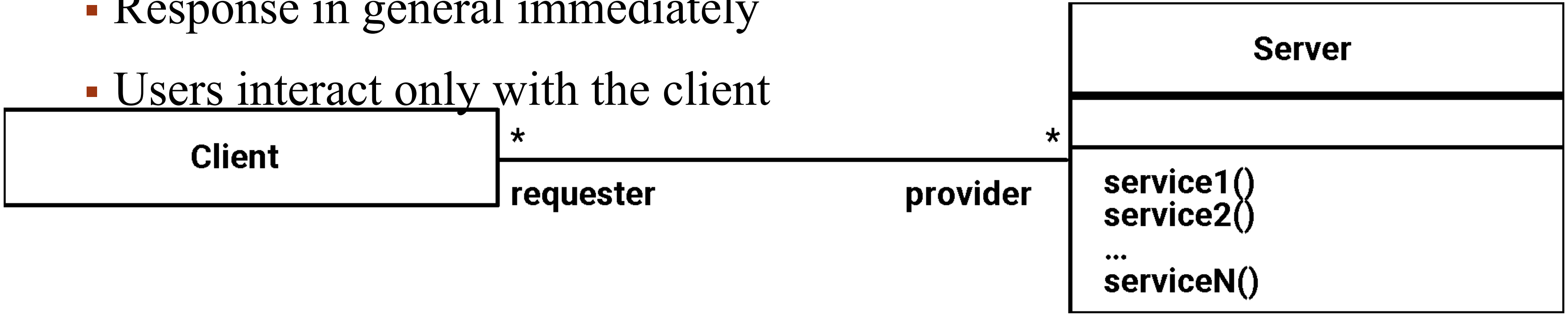
CORE J2EE PATTERNS: PATTERNS INDEX PAGE

http://java.sun.com/blueprints/corej2eepatterns/Patterns/index.html



CLIENT/SERVER ARCHITECTURAL STYLE

- One or many **servers** provides services to instances of subsystems, called **clients**.
- Client calls on the server, which performs some service and returns the result
  - Client knows the *interface* of the server *(its service)*
  - Server does not need to know the interface of the client
- Response in general immediately
- Users interact only with the client



Client

*

requester

*

provider

Server

service1()
service2()
...
serviceN()

Is "interface" the same as "interface of the server" in UML?

CLIENT/SERVER ARCHITECTURAL STYLE

- Often used in database systems:
  - Front-end: User application (client)
  - Back end: Database access and manipulation (server)
- Functions performed by client:
  - Customized user interface
  - Front-end processing of data
  - Initiation of server remote procedure calls
  - Access to database server across the network
- Functions performed by the database server:
  - Centralized data management
  - Data integrity and database consistency
  - Database security
  - Concurrent operations (multiple user access)
  - Centralized processing (for example archiving)

?

Cf. J2EE and its evolution:
-motivation behind J2EE?
-architecture?

Does a system use a single style or multiple styles?

glance through

DESIGN GOALS FOR CLIENT/SERVER SYSTEMS

- *Service Portability*
  - Server can be installed on a variety of machines and operating systems and functions in a variety of networking environments
- *Transparency, Location-Transparency*
  - The server might itself be distributed (why?), but should provide a single "logical" service to the user
- *Performance*

Is this what performance means to you?

  - Client should be customized for interactive display-intensive tasks
  - Server should provide CPU-intensive operations
- *Scalability*
  - Server should have spare capacity to handle larger number of clients
- *Flexibility*
  - The system should be usable for a variety of user interfaces and end devices (eg. WAP Handy, wearable computer, desktop)
- ***Reliability***
  - System should survive node or communication link problems

Is this what realiability means to you?

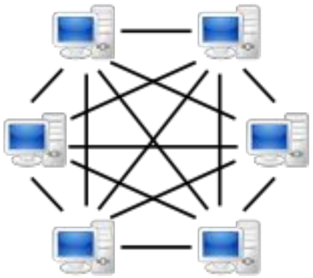
PROBLEMS WITH CLIENT/SERVER ARCHITECTURAL STYLES

- do not provide peer-to-peer communication
- Peer-to-peer communication is often needed
- Example: Database receives queries from application but also sends notifications to application when data have changed

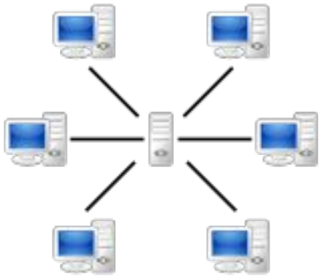
What does this mean?

PEER-TO-PEER COMMUNICATION [WIKIPEDIA]

Peer-to-peer (**P2P**) networking is a method of delivering computer network services in which the participants share a portion of their own resources, such as processing power, disk storage, network bandwidth, printing facilities. Such resources are provided directly to other participants without intermediary network hosts or servers.[1] Peer-to-peer network participants are providers and consumers of network services simultaneously, which contrasts with other service models, such as traditional client-server computing.



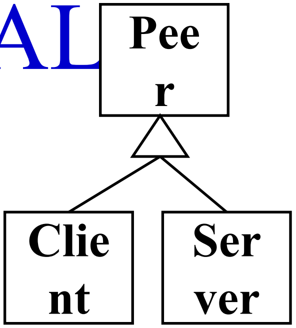
A peer-to-peer based network

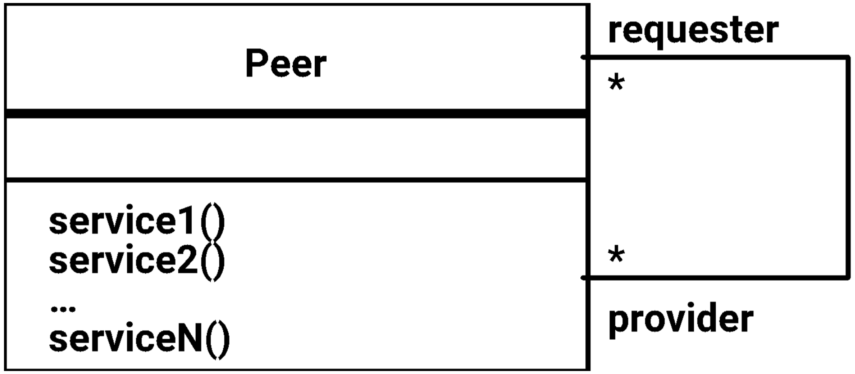


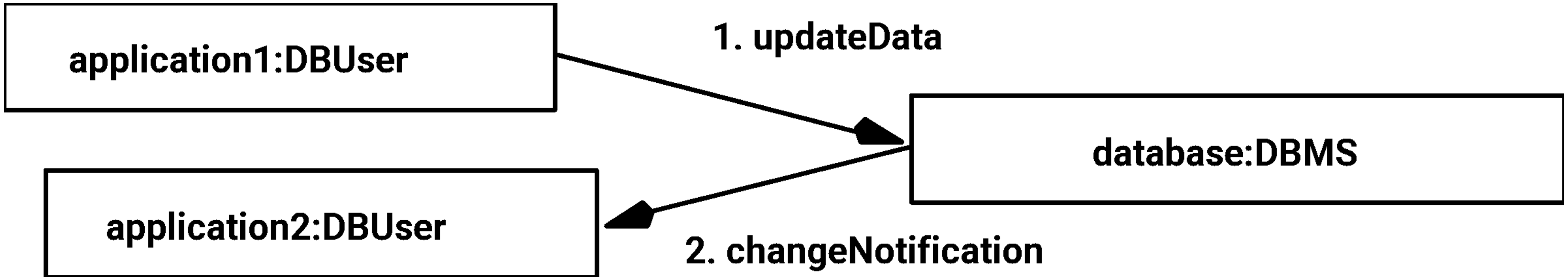
A server based network (i.e: not peer-to-peer).

PEER-TO-PEER ARCHITECTURAL STYLE

- Generalization of Client/Server Architecture
- Clients can be servers and servers can be clients



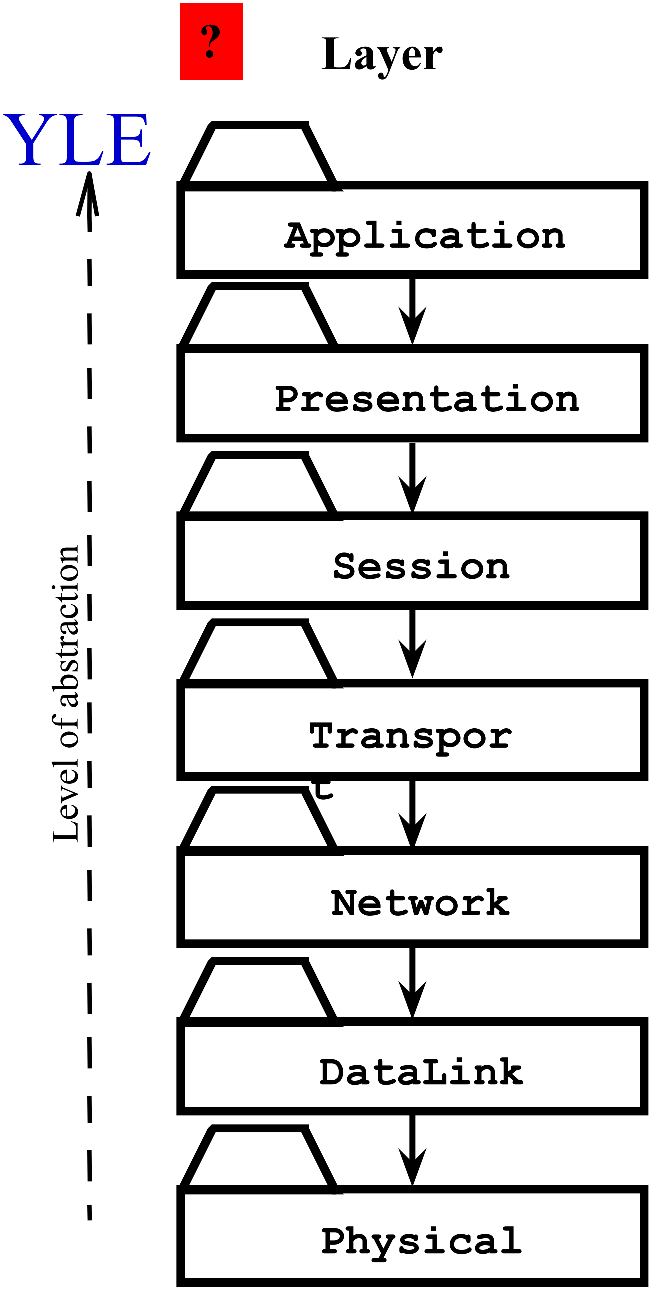




This is where the chicken-and-egg problem exists!

EXAMPLE OF A PEER-TO-PEER ARCHITECTURAL STYLE

- ISO’s OSI Reference Model
  - ISO = International Standard Organization
  - OSI = Open System Interconnection
- Reference model defines 7 layers of network protocols and strict methods of communication between the layers.
- Closed software architecture



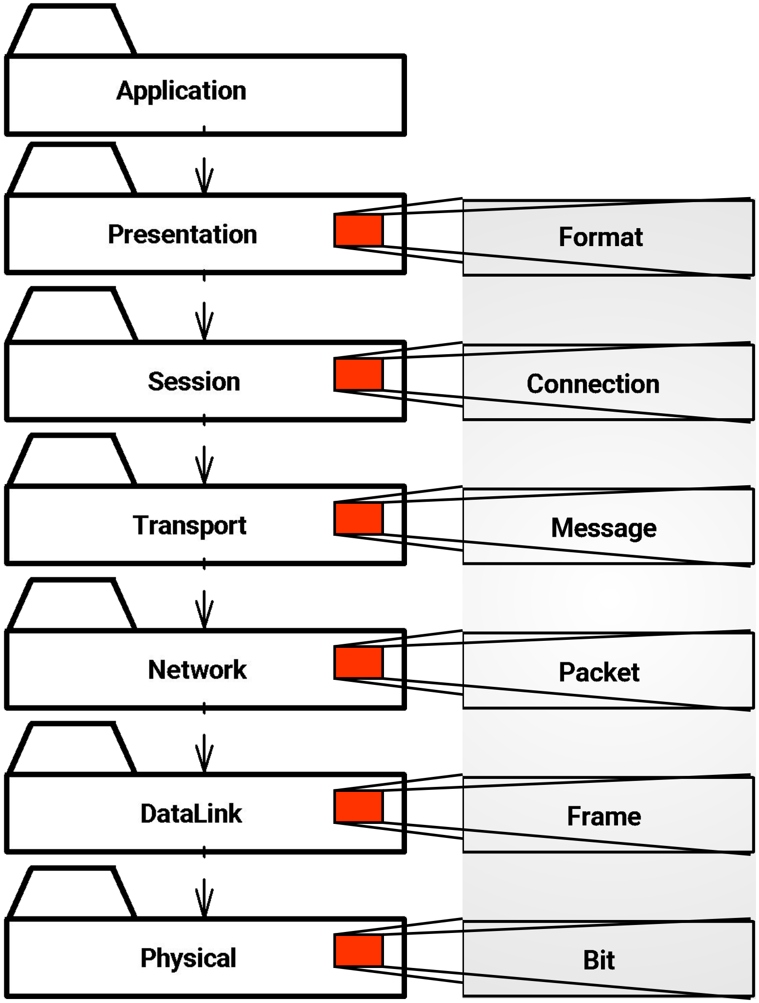
OSI MODEL PACKAGES AND THEIR RESPONSIBILITY

glance through

- The **Physical** layer represents the hardware interface to the net-work. It allows to **send()** and **receive bits** over a **channel**.
- The **Datalink** layer allows to send and receive **frames** without error using the services from the Physical layer.
- The **Network** layer is responsible for that the data are reliably **transmitted** and **routed** within a network.
- The **Transport** layer is responsible for reliably transmitting from end to end. (This is the interface seen by Unix programmers when transmitting over TCP/IP sockets)
- The **Session** layer is responsible for initializing a connection, including authentication.
- The **Presentation** layer performs data transformation services, such as byte swapping and encryption
- The **Application** layer is the system you are designing (unless you build a protocol stack). The application layer is often layered itself.

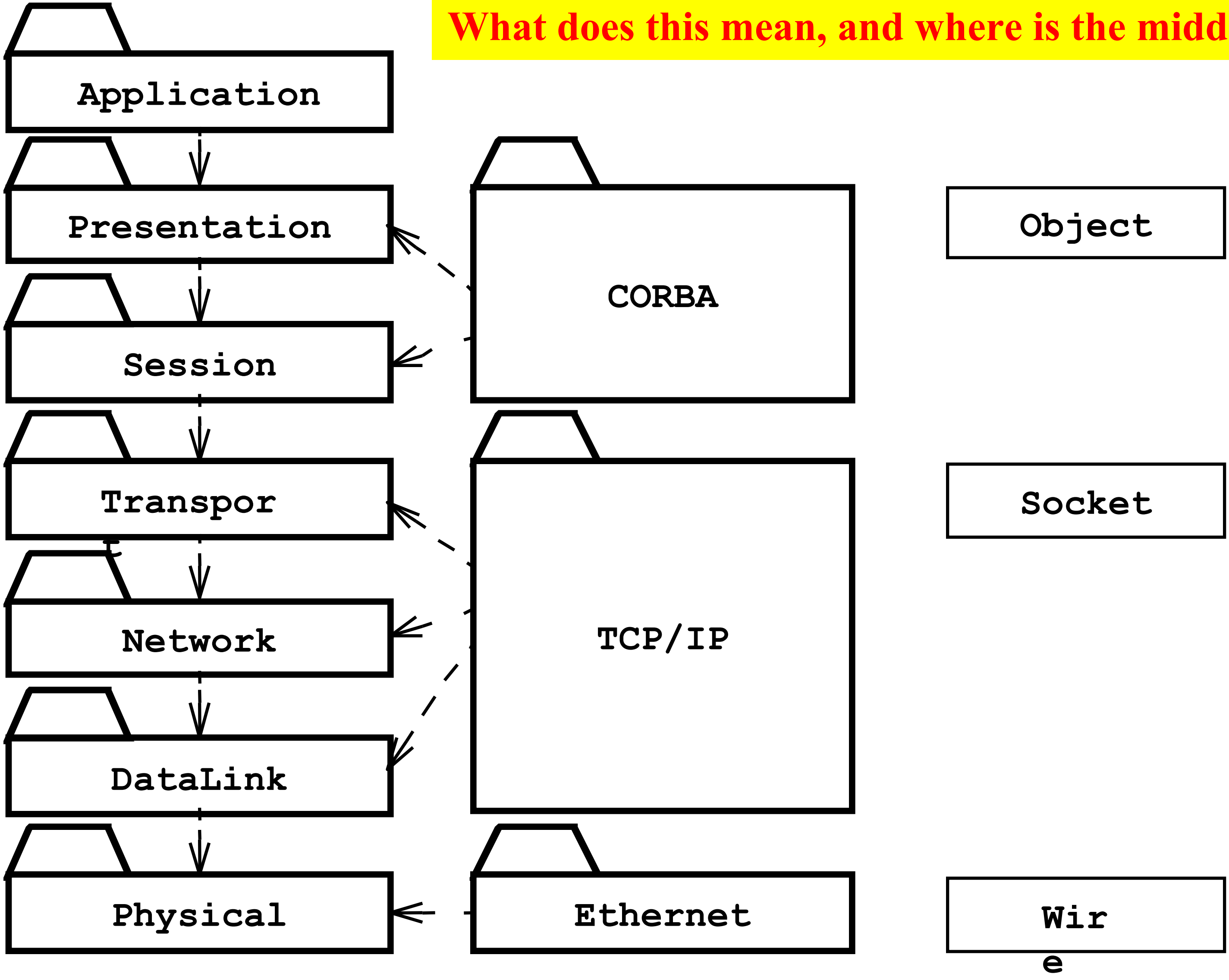
ANOTHER VIEW AT THE ISO MODEL

- A closed software architecture
- Each layer is a UML package containing a set of objects



MIDDLEWARE ALLOWS FOCUS ON THE APPLICATION LAYER

What does this mean, and where is the middleware?

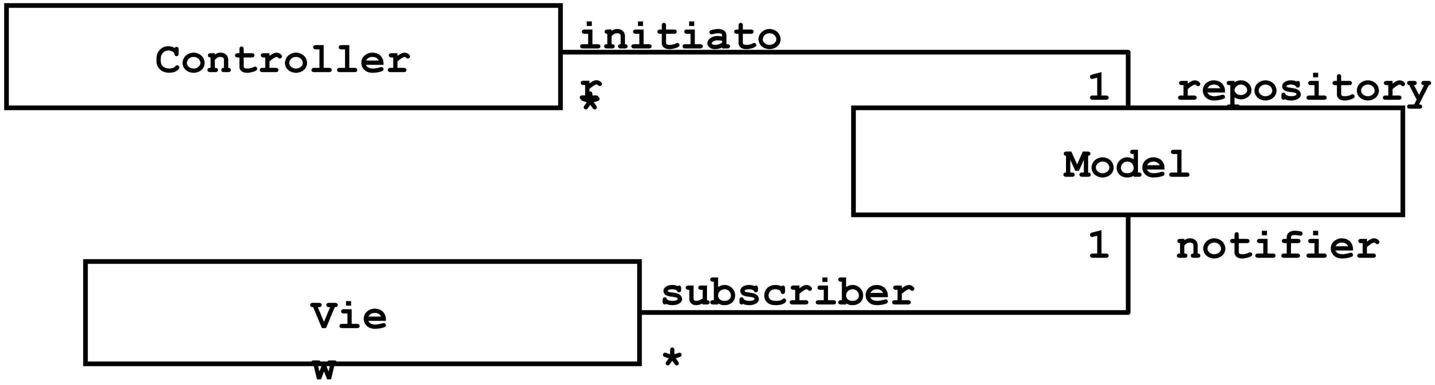


MODEL/VIEW/CONTROLLER

Cf. boundary/control/entity

- 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.
- MVC is a special case of a repository architecture:
  - Model subsystem implements the central datastructure, the Controller subsystem explicitly dictate the control flow

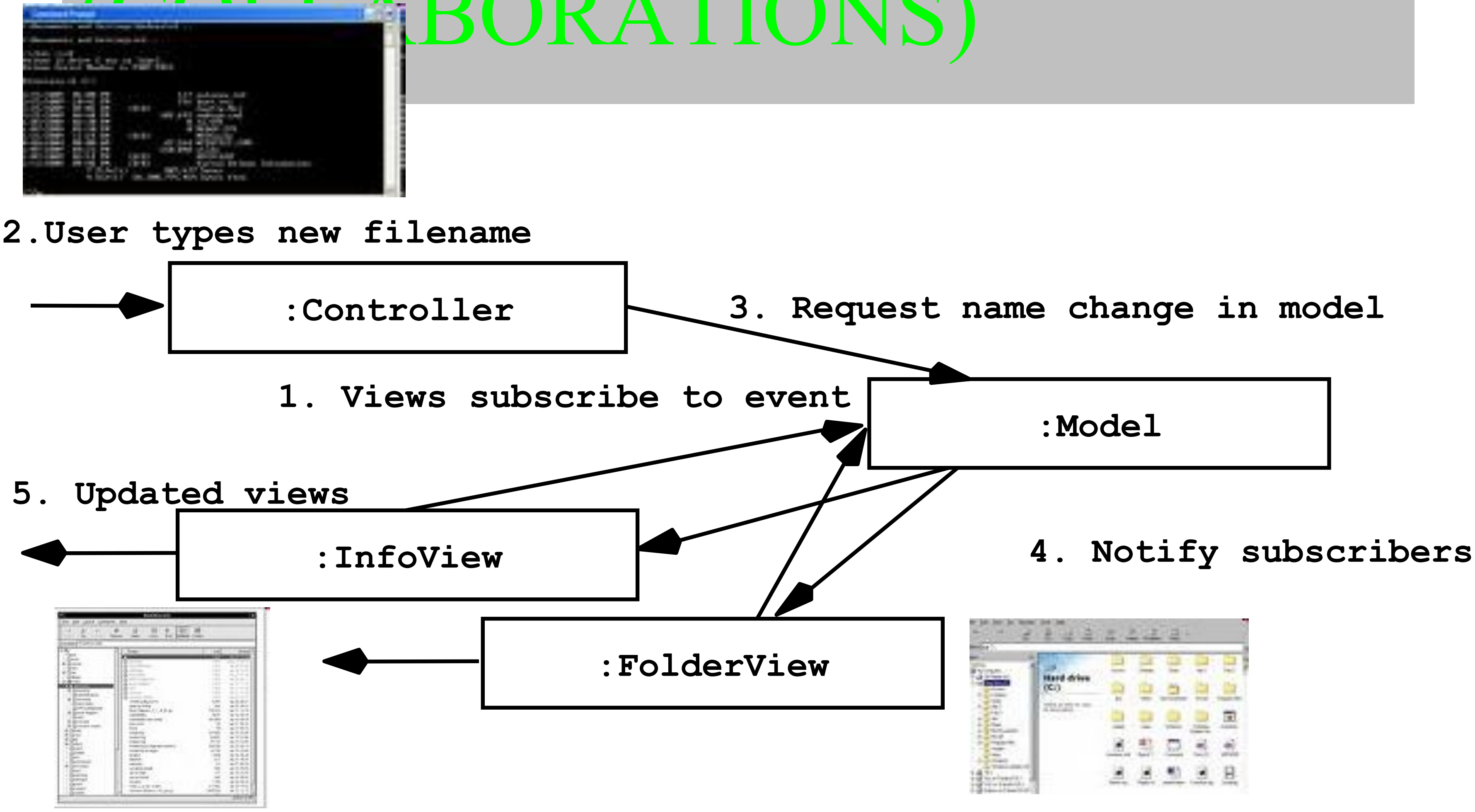
What is this?



Which interacts with the user?

What creates boundary objects?

SEQUENCE OF EVENTS (COLLABORATIONS)



2.User types new filename

:Controller

3. Request name change in model

1. Views subscribe to event

:Model

5. Updated views

:InfoView

4. Notify subscribers

:FolderView

Which interacts with the user?

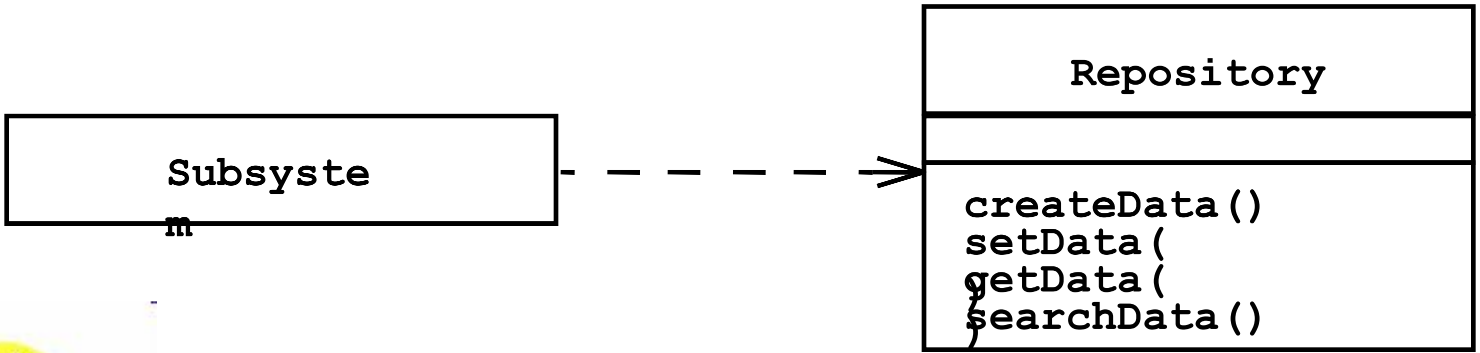
What creates boundary objects?

REPOSITORY ARCHITECTURAL STYLE (BLACKBOARD ARCHITECTURE, HEARSAY II SPEECH RECOGNITION SYSTEM)

What's blackboard? Are all repository architectural styles mean blackboard?

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

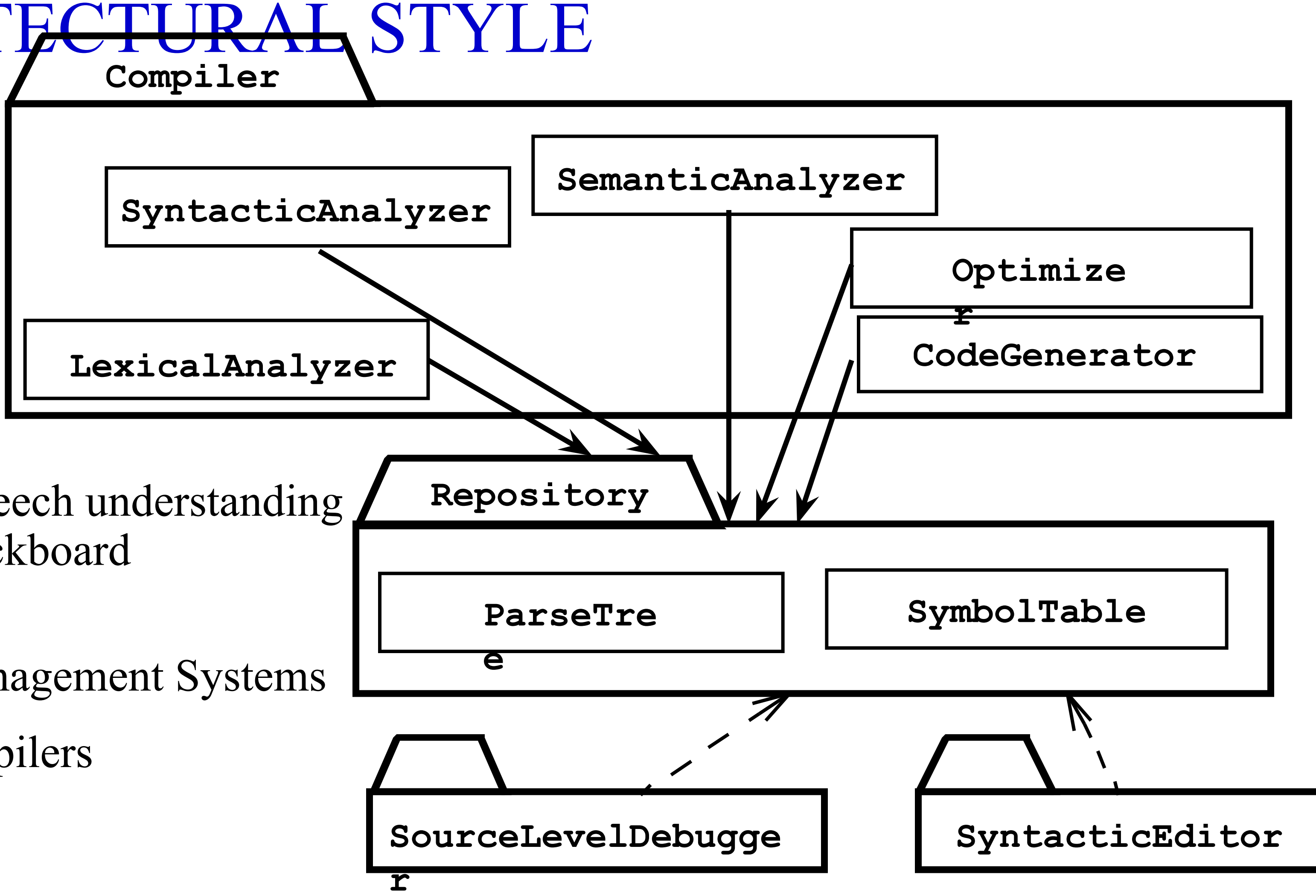
Two kinds



Wreck the nice beach

EXAMPLES OF REPOSITORY ARCHITECTURAL STYLE

- Hearsay II speech understanding system (“Blackboard architecture”)
- Database Management Systems
- Modern Compilers



Compiler

SyntacticAnalyzer

SemanticAnalyzer

Optimize
r

CodeGenerator

LexicalAnalyzer

Repository

ParseTre
e

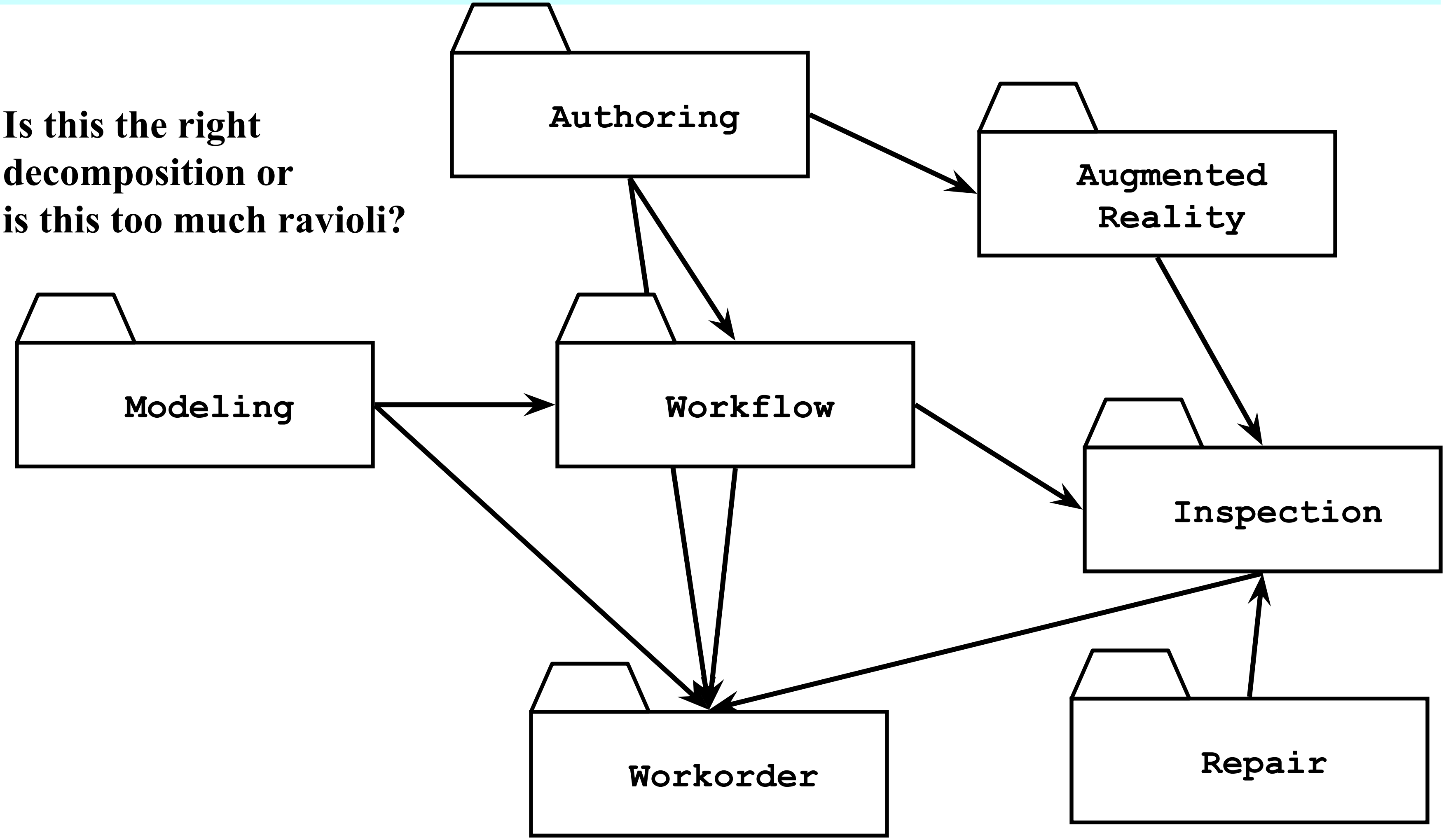
SymbolTable

SourceLevelDebugge
r

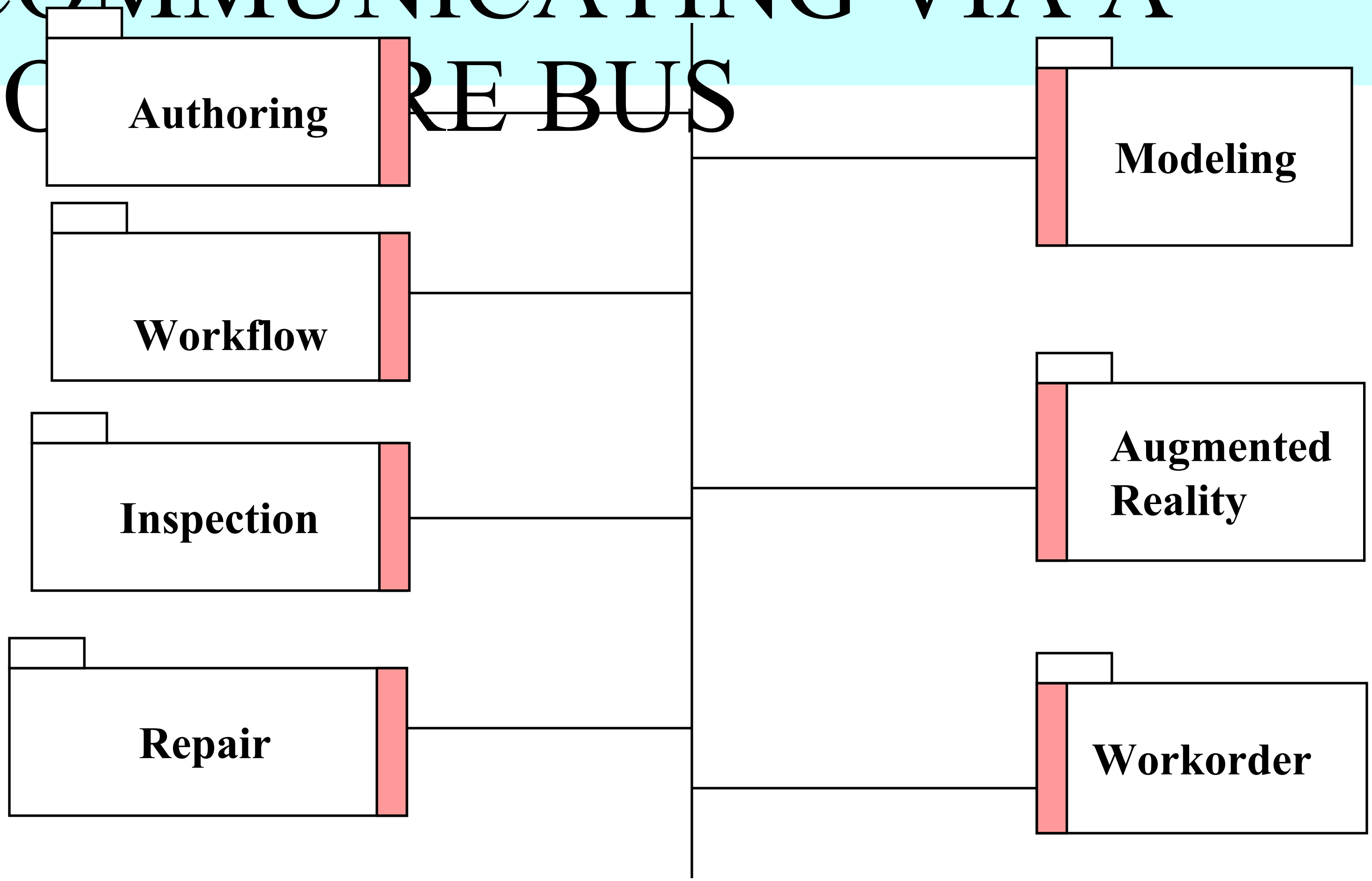
SyntacticEditor

SUBSYSTEM DECOMPOSITION EXAMPLE

Is this the right
decomposition or
is this too much ravioli?



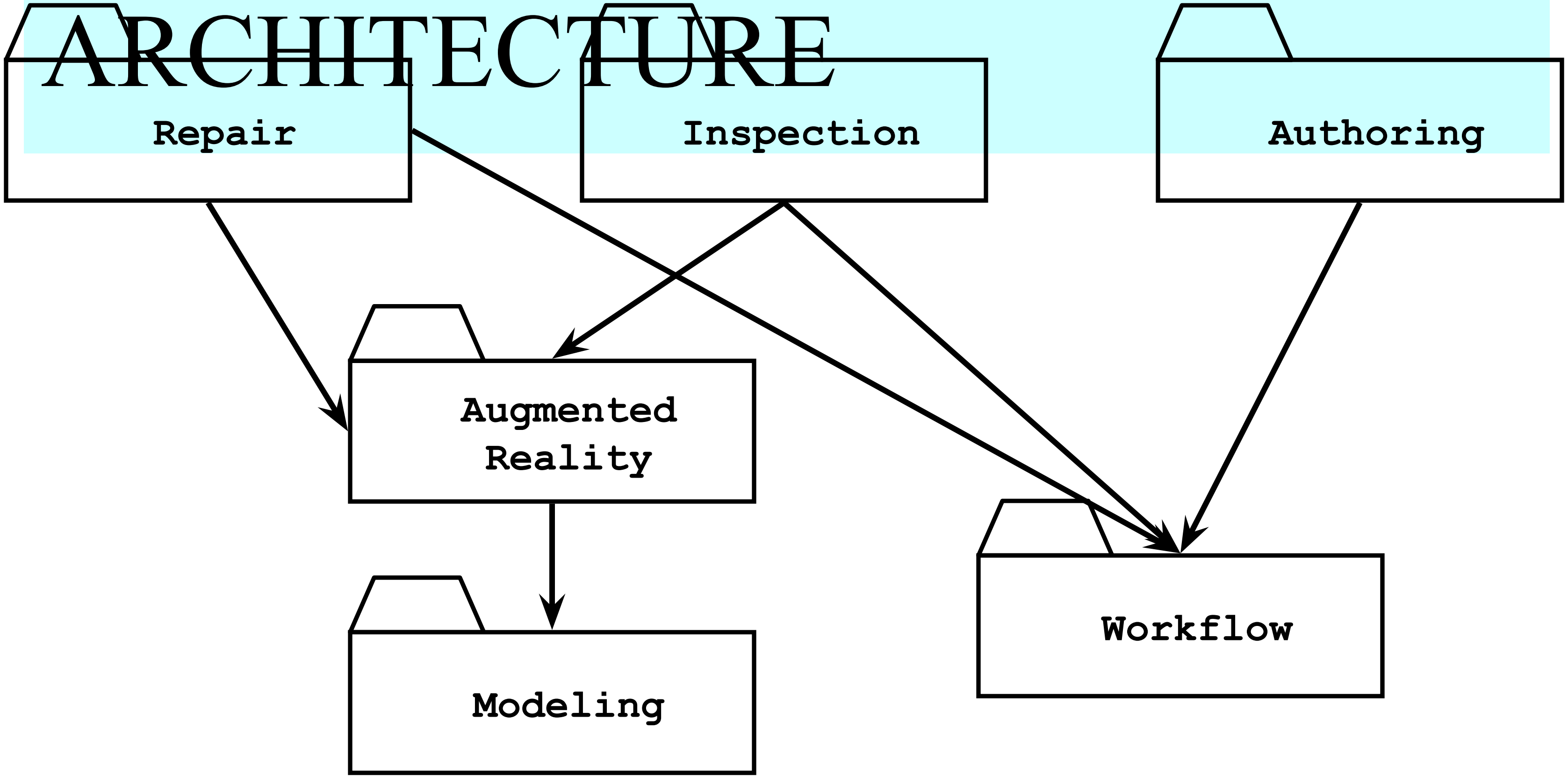
SYSTEM AS A SET OF SUBSYSTEMS COMMUNICATING VIA A SOFTWARE BUS



🟥 **A Subsystem Interface Object publishes the service (= Set of public methods) provided by the subsystem**

What is this architectural style called?

A 3-LAYERED ARCHITECTURE



What is the relationship between Modeling and Authoring?
Are other subsystems needed?

SUMMARY

- System Design
  - Reduces the gap between requirements and the (virtual) machine
  - Decomposes the overall system into manageable parts
- Design Goals Definition
  - Describes and *prioritizes* the qualities that are important for the system
  - Defines the value system against which options are evaluated
- Subsystem Decomposition
  - Results into a set of loosely dependent parts which make up the system

NONFUNCTIONAL REQUIREMENTS MAY GIVE A CLUE FOR THE USE OF DESIGN PATTERNS

- Read the problem statement again
- Use textual clues (similar to Abbot’s technique in Analysis) to identify design patterns

- *Text:* “manufacturer independent”, “device independent”, “must support a family of products”
  - Abstract Factory Pattern
- *Text:* “must interface with an existing object”
  - Adapter Pattern
- *Text:* “must deal with the interface to several systems, some of them to be developed in the future”, “ an early prototype must be demonstrated”
  - Bridge Pattern

TEXTUAL CLUES IN NONFUNCTIONAL REQUIREMENTS

- *Text:* “complex structure”, “must have variable depth and width”
  - Composite Pattern
- *Text:* “must interface to an set of existing objects”
  - Façade Pattern
- *Text:* “must be location transparent”
  - Proxy Pattern
- *Text:* “must be extensible”, “must be scalable”
  - Observer Pattern
- *Text:* “must provide a policy independent from the mechanism”
  - Strategy Pattern

DEFINITION: SUBSYSTEM INTERFACE OBJECT

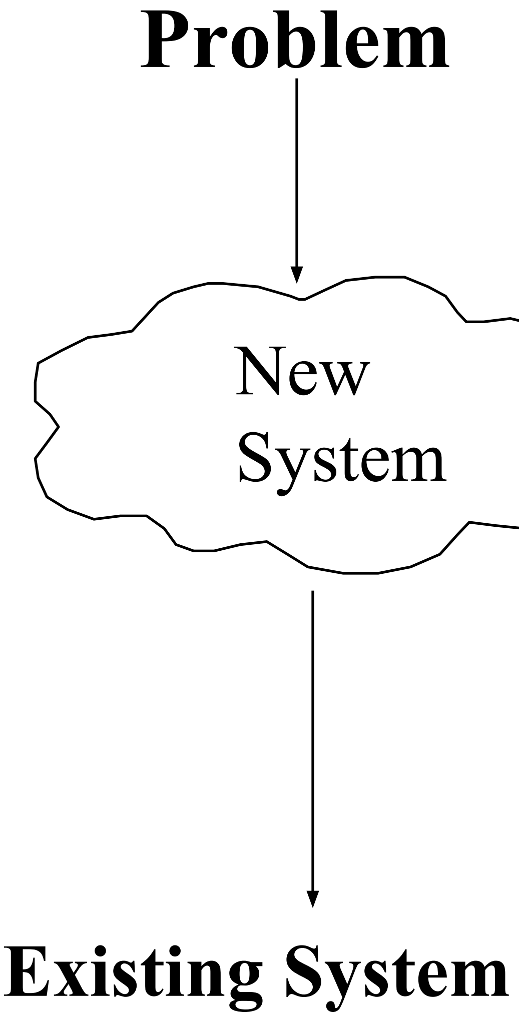
- A *Subsystem Interface Object* provides a service
  - This is the set of public methods provided by the subsystem
  - The Subsystem interface describes all the methods of the subsystem interface object
- Use a Facade pattern for the subsystem interface object

CHOOSING SUBSYSTEMS

- Criteria for subsystem selection: Most of the interaction should be within subsystems, rather than across subsystem boundaries (High cohesion).
  - Does one subsystem always call the other for the service?
  - Which of the subsystems call each other for service?
- Primary Question:
  - What kind of service is provided by the subsystems (subsystem interface)?
- Secondary Question:
  - Can the subsystems be hierarchically ordered (layers)?
- What kind of model is good for describing layers and partitions?

THE PURPOSE OF SYSTEM DESIGN

- Bridging the gap between desired and existing system in a manageable way
- Use Divide and Conquer
  - We model the new system to be developed as a set of subsystems

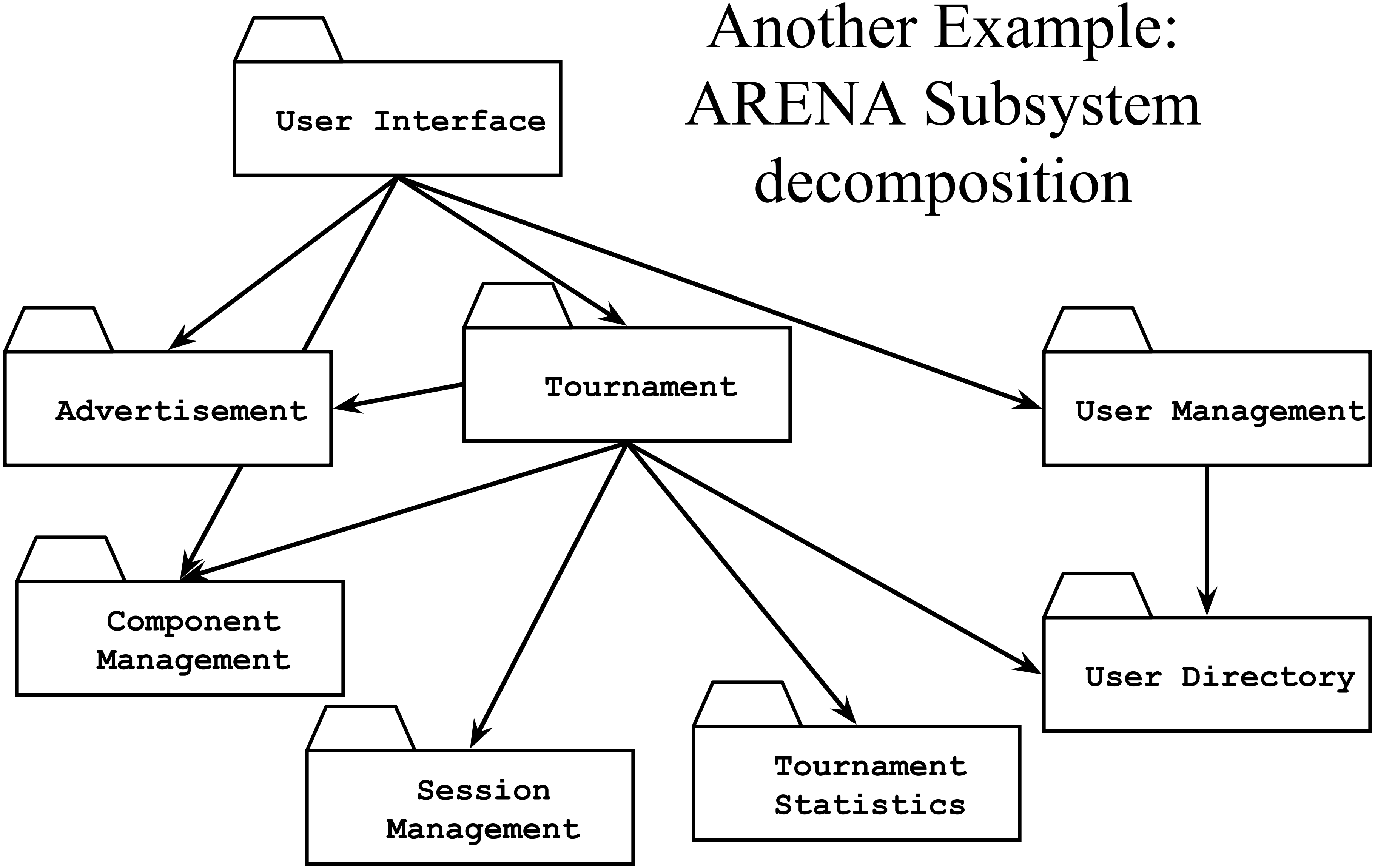


Problem

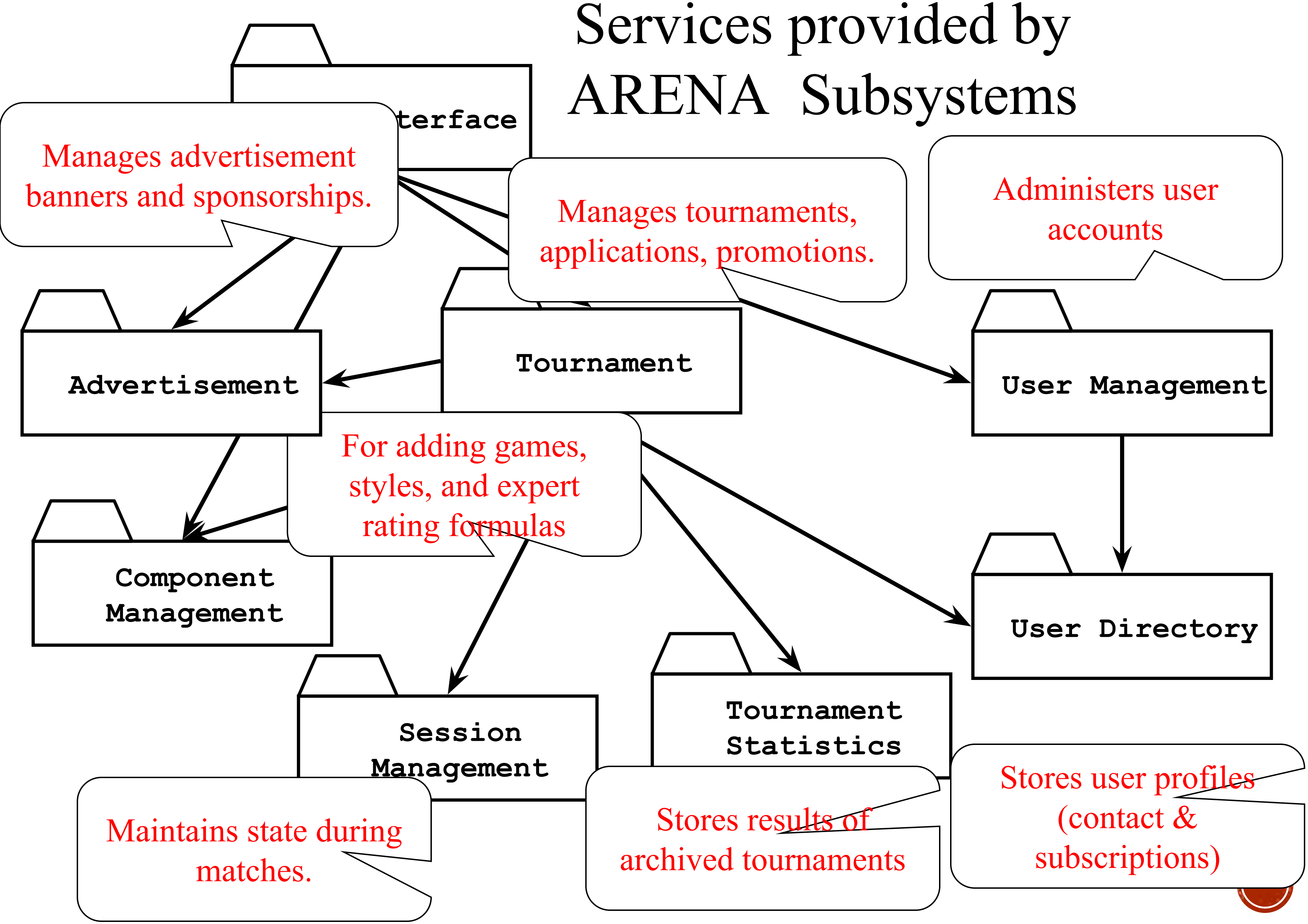
New System

Existing System

Another Example: ARENA Subsystem decomposition



Services provided by ARENA Subsystems



terface

Manages advertisement banners and sponsorships.

Manages tournaments, applications, promotions.

Administers user accounts

Advertisement

Tournament

User Management

For adding games, styles, and expert rating formulas

Component Management

User Directory

Session Management

Tournament Statistics

Maintains state during matches.

Stores results of archived tournaments

Stores user profiles (contact & subscriptions)

SERVICES AND SUBSYSTEM INTERFACES

- Service: A set of related operations that share a common purpose
  - Notification subsystem service:
    - LookupChannel()
    - SubscribeToChannel()
    - SendNotice()
    - UnscubscribeFromChannel()
  - Services are defined in System Design
- Subsystem Interface: Set of fully typed related operations.
  - Subsystem Interfaces are defined in Object Design
  - Also called application programmer interface (API)