



System Design

(chapter 7, Bruegge et al)
(chapter 6, Sommerville, 9th ed.)

Prof Adel Taweel

Objectives

To provide an overview of System Design

To appreciate issues addressed during the system design phase

To understand consequences of design goals and how to achieve them

To appreciate the value of architectural styles in formulating system designs.

Design is multi-perspective

Analysis: Focuses on the business (or problem) domain.

Design: Focuses on the solution domain.

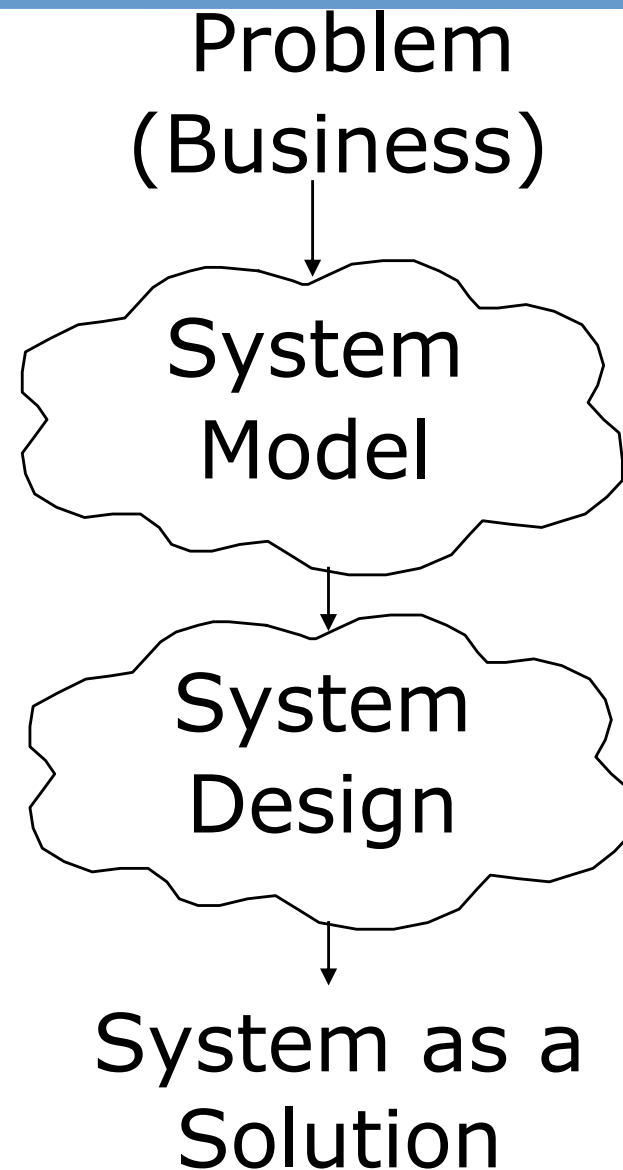
The (system) solution perspective considers three aspects:

- Software artifacts
- Associated technologies
- Hardware limitations or solutions

The Scope of System Design

It bridges the gap between a (business) problem and an system solution in a structured way

- How?
- Use Divide & Conquer (see next slide):
 - 1) Identify design goals
 - 2) Model the new system design as a set of components (subsystems) or sub-solutions
 - 3-8) Address main design goals.



System Design: Seven Key Issues to address

System Design

```
graph TD; SD[System Design] --- G1[1. Design Goals]; SD --- G2[2. Subsystem Decomposition]; SD --- G3[3. Data Management]; SD --- G4[4. Hardware/Software Mapping]; SD --- G5[5. Resource Control]; SD --- G6[6. Software Control]; SD --- G7[7. Boundary Conditions];
```

1. Design Goals

How to achieve non-functional requirements?
=> As optimally as possible

2. Subsystem Decomposition

Solution as components (sub-solutions)
->Architectural Style
->Layers vs Partitions
->Cohesion/Coupling

3. Data Management

How to persist Objects?
e.g. File system vs Database

4. Hardware/ Software Mapping

Identification of hardware needs,
nodes and configurations
Special Purpose Systems (Buy vs Build)
Network Connectivity

5. Resource Control

Security vs Capabilities
e.g. Access Control

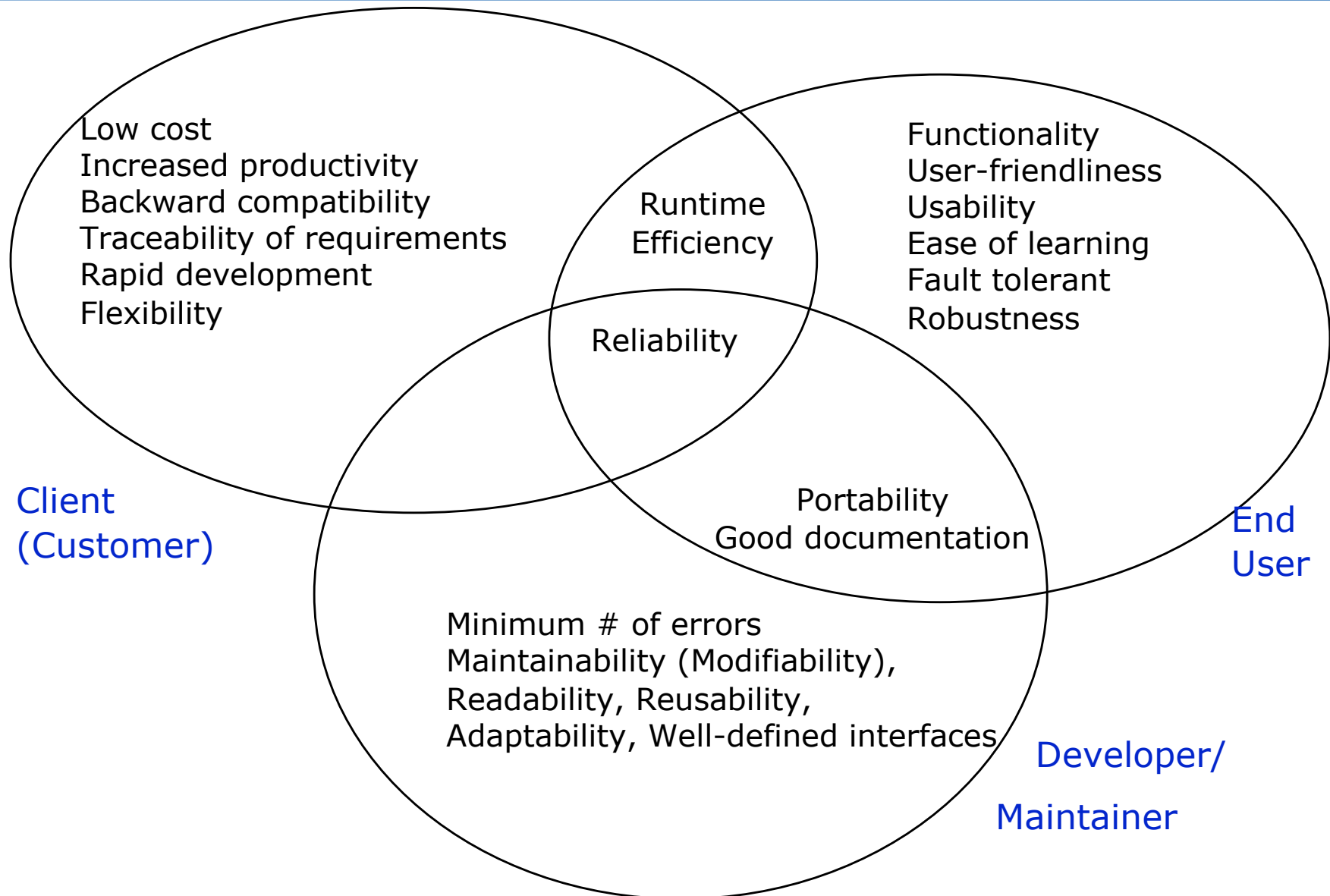
7. Boundary Conditions

Under what assumptions
the system starts?
How and when the system
terminates?
How to handle failure

6. Software Control

Monolithic/centralised/
decentralised/distributed
e.g. Event-Driven

Stakeholders have different Design Goals (OR Issues of Concerns)



Some Design Trade-offs for a developer

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

Design Goals: Coupling and Cohesion

Cohesion measures degree of the strength of functional relatedness within classes (or among classes) in a component

High cohesion: The classes in a component (a subsystem) perform similar or related tasks/functions and are related to each other via many associations

Low cohesion: classes are grouped with no clear relatedness between them, but lots of miscellaneous and auxiliary classes, almost no associations

Coupling measures degree of interdependence between components (or subsystems)

High coupling: Changes to one component will cause significant changes to another component

Low coupling: A change in one component will cause minimal or no effect to other components

Why high Cohesion?

- increases the clarity and ease of comprehension of the design
- simplifies maintenance and future upgrades and enhancements
- often supports low coupling
- supports increased reuse
 - a highly cohesive (i.e. a highly related functionality) component can be re-used for the same specific purpose!

How to achieve high Cohesion?

High Cohesion can be achieved if most of the interaction is kept within a component (opposed to “between components”)

Indicators:

Does a component (or a subsystem) often or always call another component (or subsystem) for a specific service?

⇒ If yes: Consider moving them together into the same subsystem, preferably on the same h/w physical node.

Which of the components call each other for services?

⇒ Can this be avoided by restructuring the components (i.e. classes within) or changing their interfaces?

Can the components be hierarchically ordered (in layers)?

How to achieve Low Coupling?

Low coupling can be achieved by making (self-contained components) as independent as possible by not needing any knowledge of or relying on other components to complete its function.

Indicator: if a calling class does not need to know about the internal (knowledge, e.g. attributes) of the called class (**Principle of information hiding**, Parnas)

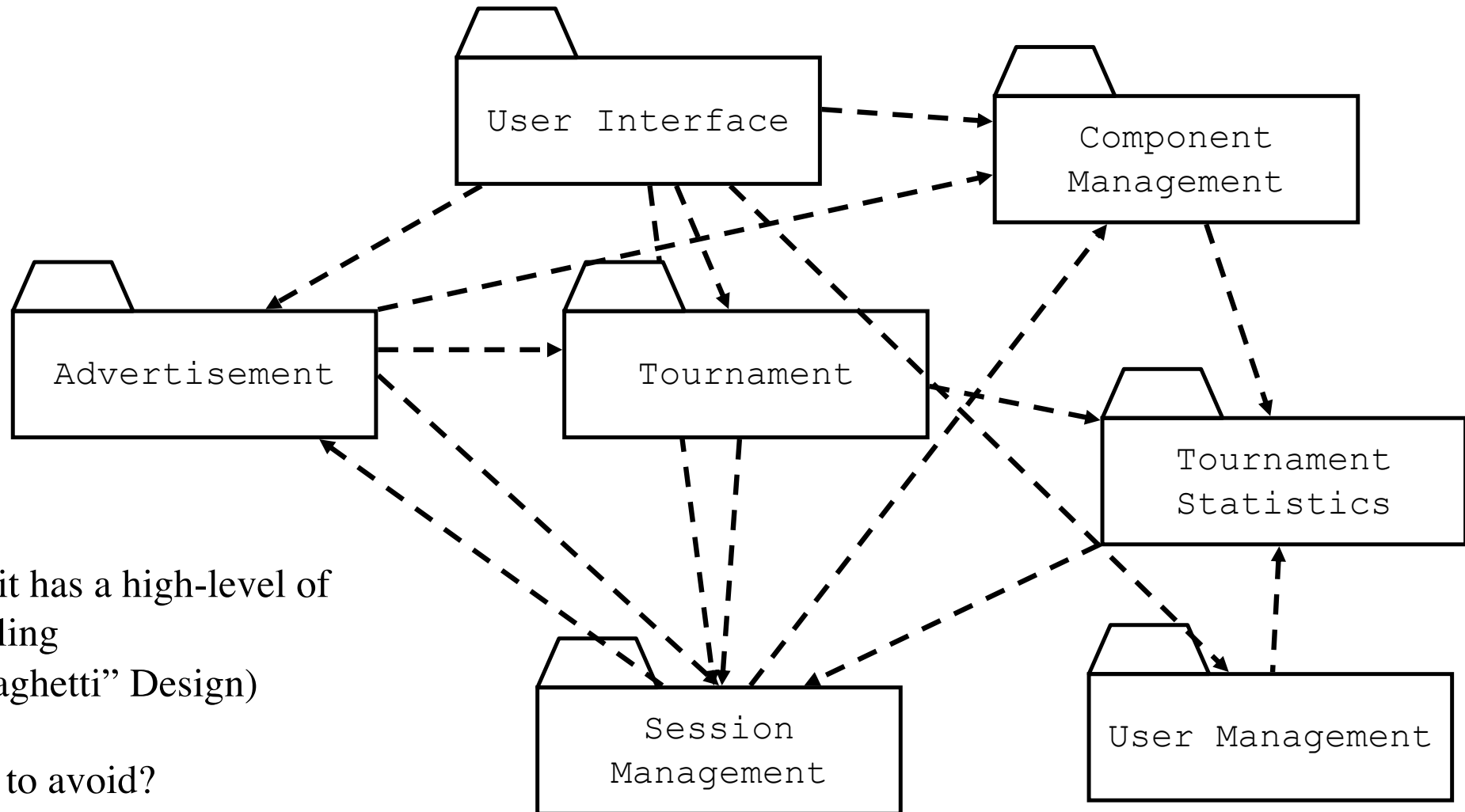
Does the calling class need to know about attributes of other called classes?
=> Define as interfaces (or public operations)

David Parnas,
Developed the concept of
Information Hiding in modular programming.



Is this a Good Design?

Package Diagram Each package contains a component or a collection of component (similar to component diagram)



No, it has a high-level of coupling
("Spaghetti" Design)

How to avoid?

Architectural Style vs Architecture

Subsystem decomposition:

Identification of subsystems, services, and their relationship to each other

Architectural Style:

A pattern for a subsystem decomposition

Software Architecture:

Instance of an architectural style.

Examples of Architectural Styles

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

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

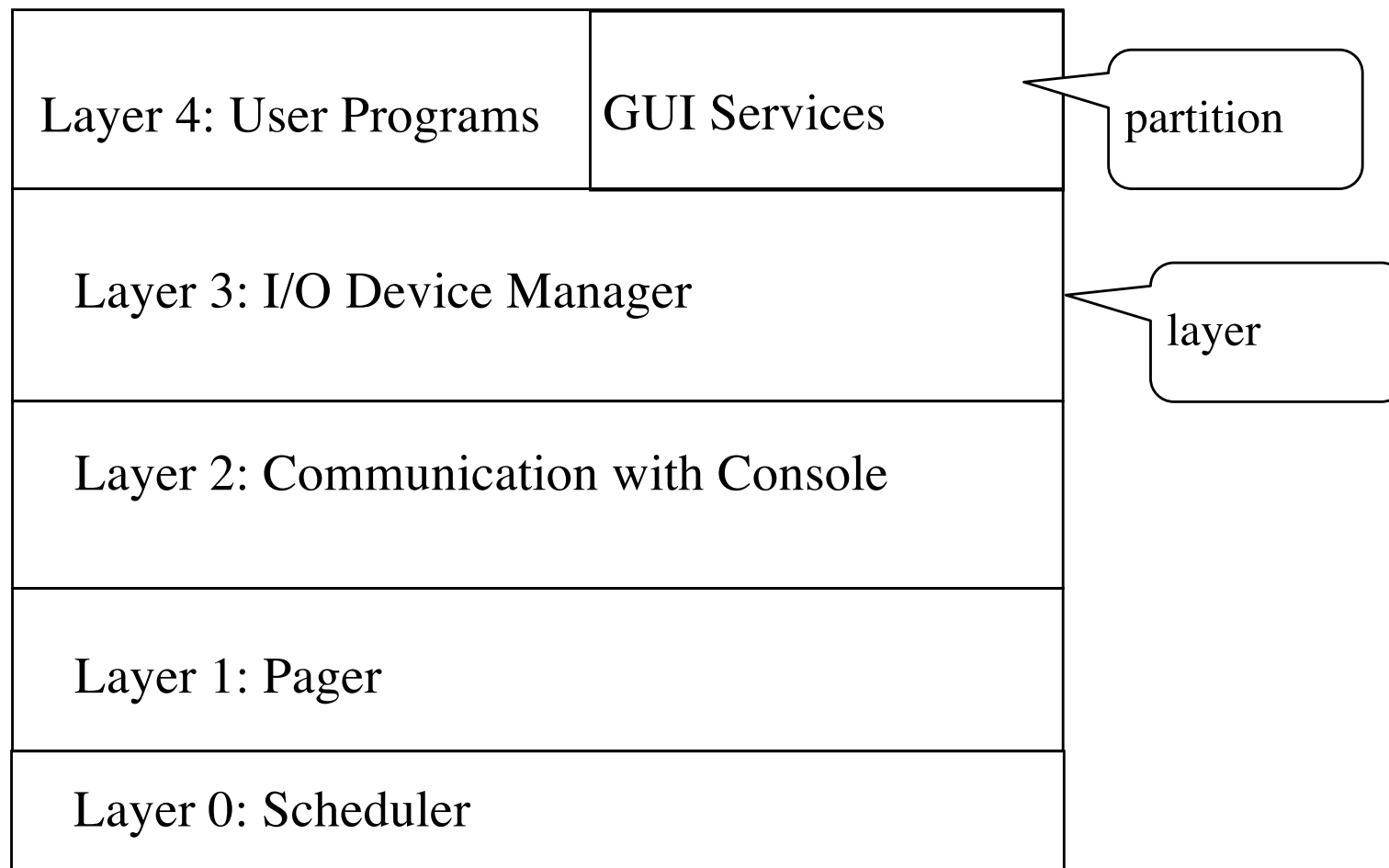
A layer can be divided horizontally into several independent subsystems called **partitions**

- Partitions provide services to other partitions on the same layer

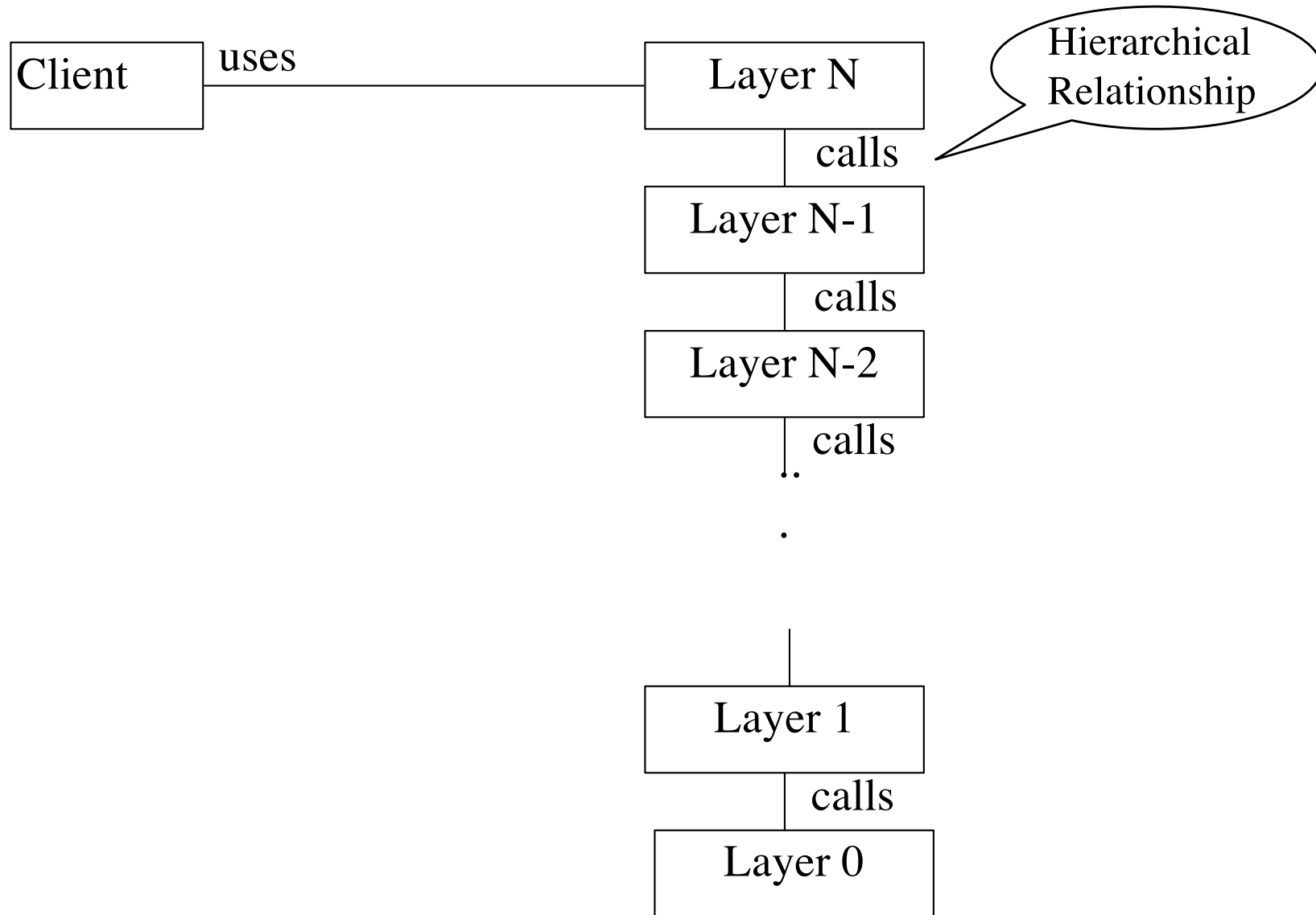
- Partitions are also called “weakly coupled” subsystems.

The Layers of an O.S. System

“An operating system is a hierarchy of layers, each layers using services offered by the lower layers”



The Layered Architectural Style



Hierarchical Relationships between Subsystems

There are two major types of hierarchical relationships

Layer A “depends on” layer B (compile time dependency)

Example: Build dependencies (e.g. make, ant, maven)

Layer A “calls” layer B (runtime dependency)

Example: A web browser calls a web server

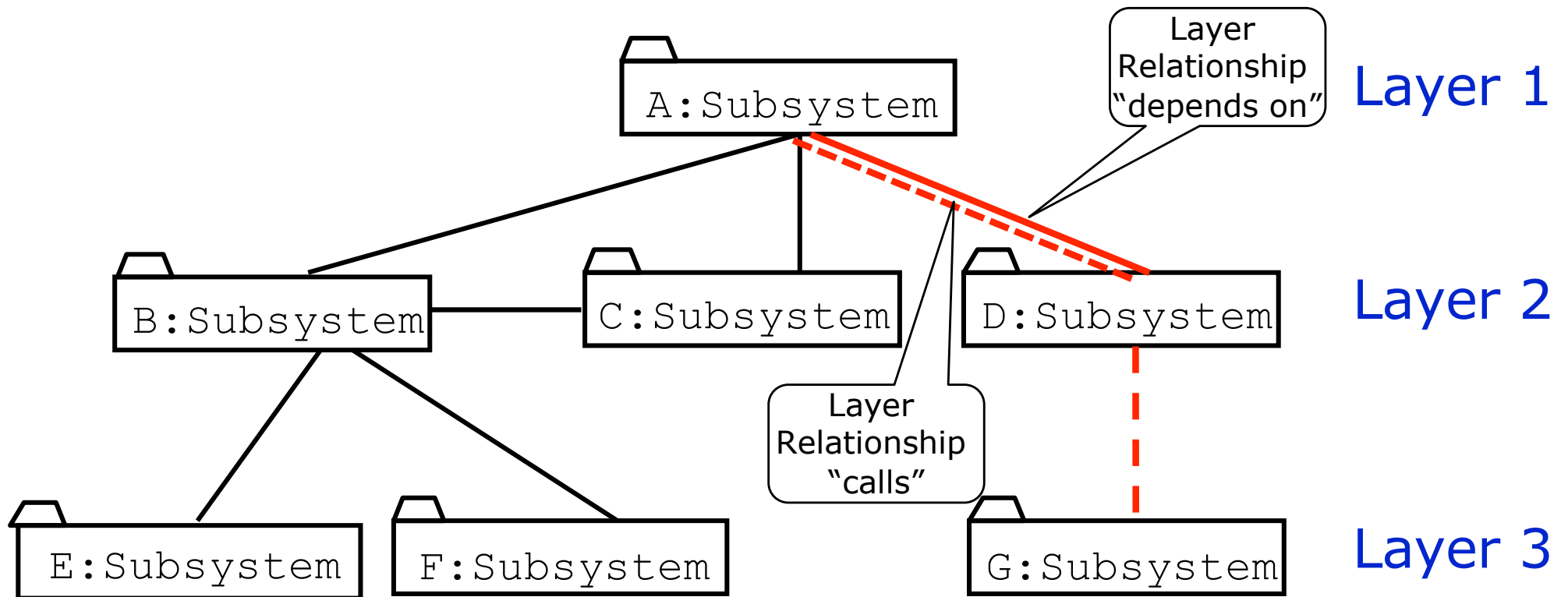
UML convention:

⇒ Runtime relationships are represented with dashed lines

⇒ Compile time relationships are represented with solid lines

Example of a System with more than one Hierarchical Relationship

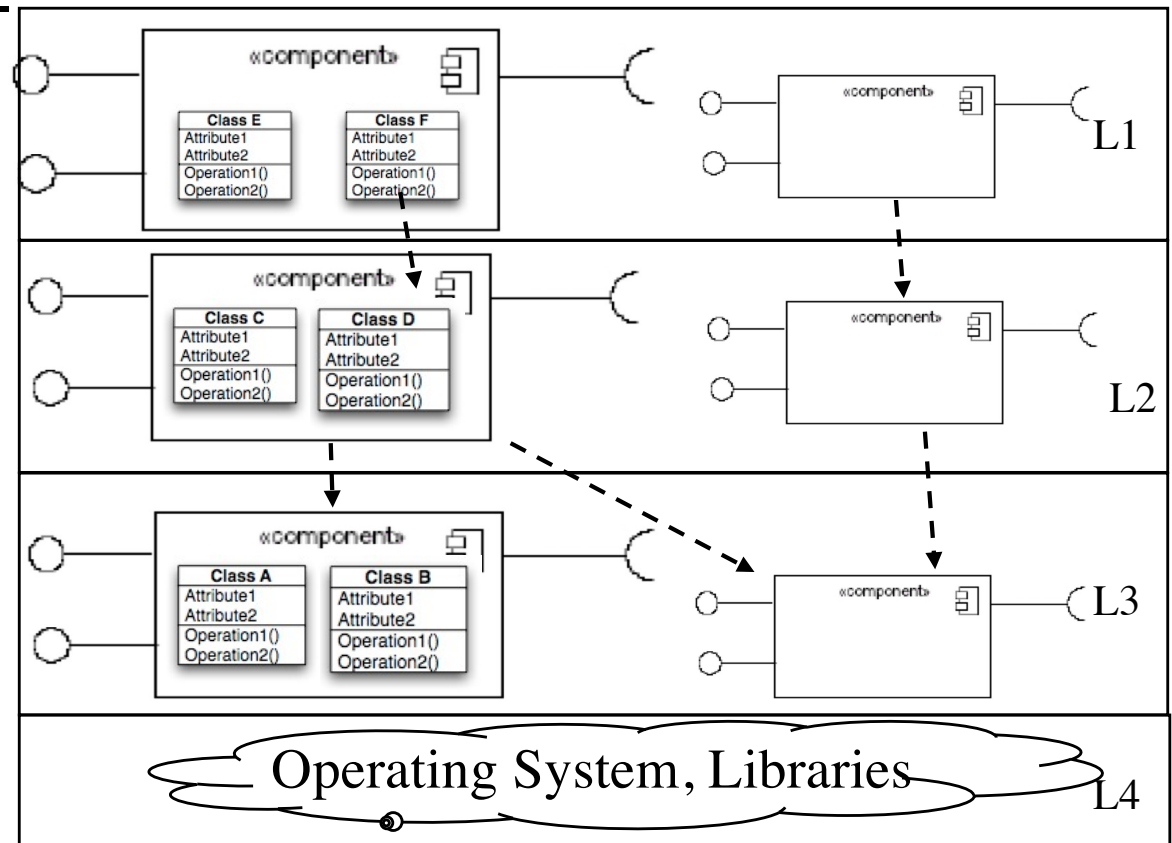
- compile time dependency
- - - Run time dependency



Closed Architecture (Opaque Layering)

Each layer can only call operations from the layer below (called “direct addressing”)

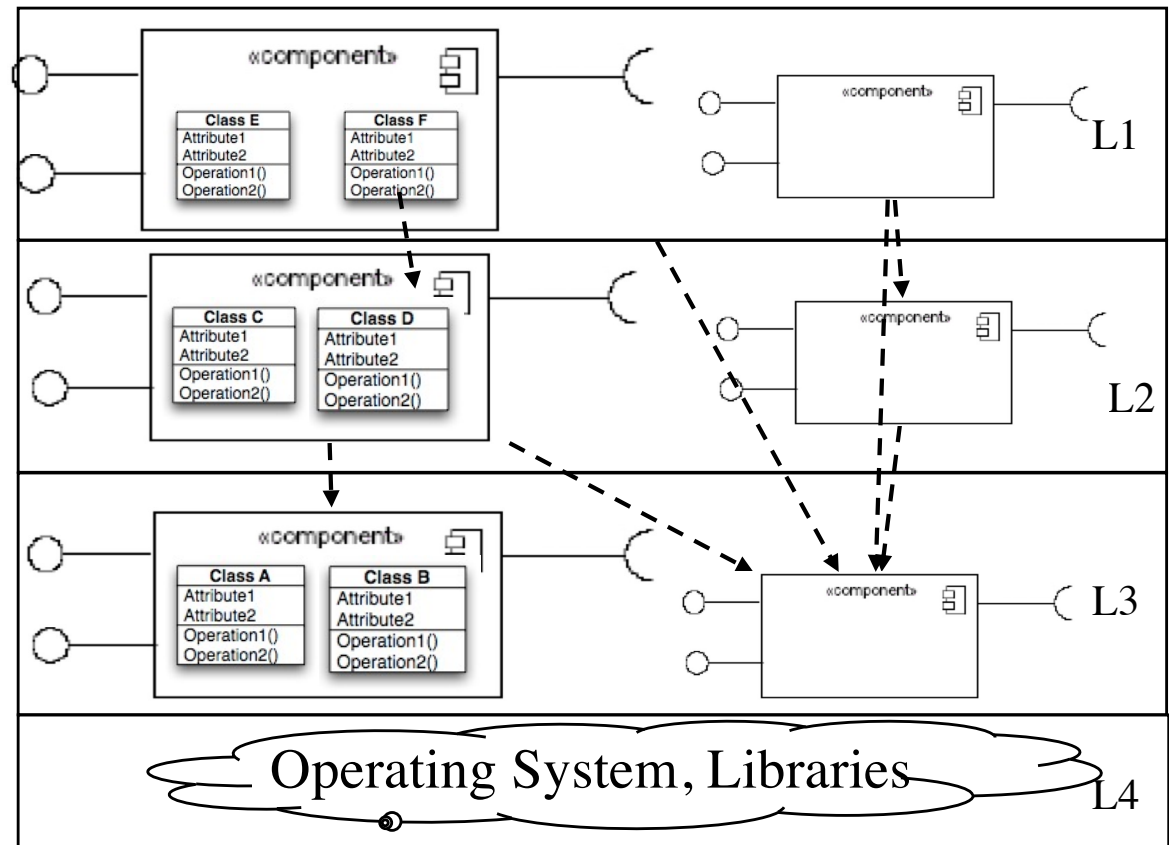
Design goals:
Maintainability,
flexibility.



Open Architecture (Transparent Layering)

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

Design goal:
Runtime efficiency.



Client/Server Architectures

Often used in the design of many applications, e.g. database systems

Front-end: User application (client)

Back end: Database access and manipulation (server)

Functions performed by client:

Input from the user (Customized user interface)

Front-end processing of input data

Functions performed by the database server:

Centralized data management

Data integrity and database consistency

Database security

Client/Server Architectural Style

Special case of the Layered Architectural style

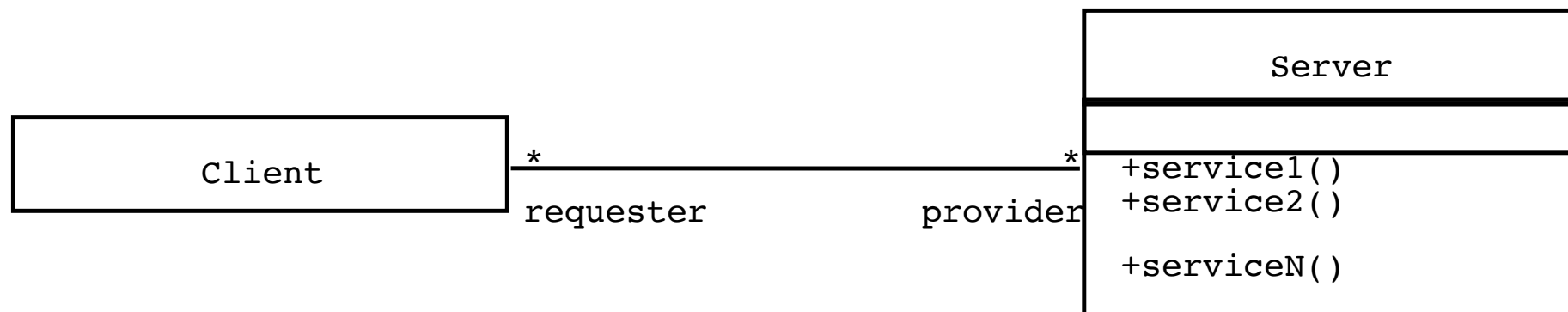
One or many **servers** provide services to instances of subsystems, called **clients**

- Each client calls on the server, which performs some service and returns the result

The clients know the *interface* of the server

The server does not need to know the interface of the client

- The response is, in general, reasonably fast
- End users interact only with the client



Client/Server Architectures: Design Features/goals

Service Portability

High=>Server runs on many operating systems and many networking environments

Location-Transparency

Reasonable=> Server might itself be distributed, but provides a single "logical" service to the user

High Performance

Reasonable=>Client optimized for interactive display-intensive tasks; Server optimized for CPU-intensive operations

Scalability

High=>Server can handle large # of clients

Client Portability

Reasonable=>User interface of client can support a variety of end devices (PDA, Handy, laptop, wearable computer)

Reliability

Reasonable=>Server would survive some client and communication problems. e.g. crash of a client does not cause all the system to fail

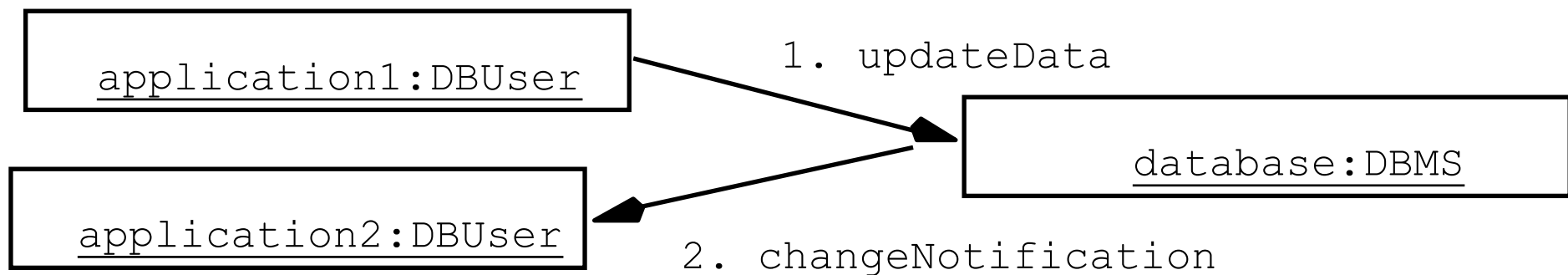
Client/Server Architectures: Limitations

Client/Server systems do not provide peer-to-peer communication, e.g. calls can only be triggered from a client NOT vice versa

Peer-to-peer communication is often needed for improved robustness.

Example:

In some situations, it would be useful if a Database could process queries from an application/client and be able to send notifications to the application when data have changed – this is not possible in client-server!



Peer-to-Peer Architectural Style

Generalization of Client/Server Architectural Style

“Clients can be servers and servers can be clients”

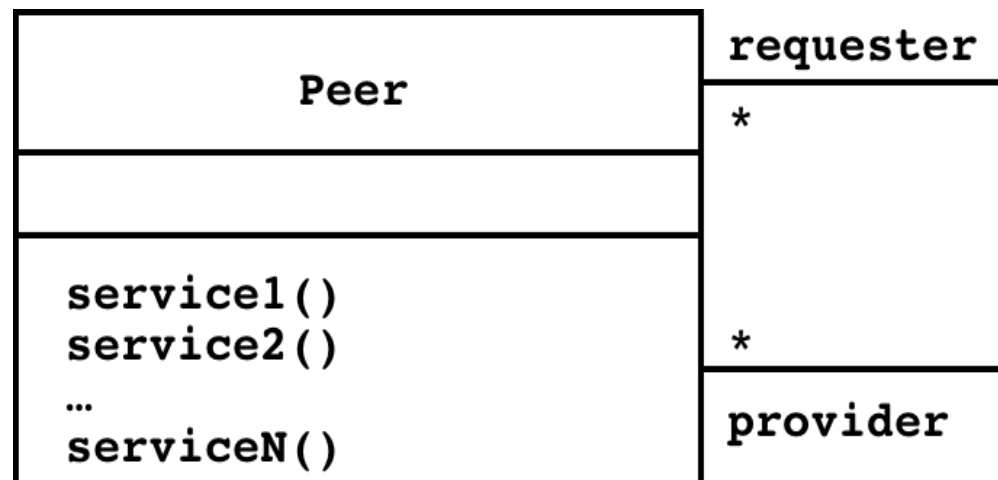
Introduces a new abstraction: **Peer**

“Clients and servers can both be 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 a client as well as a server”.



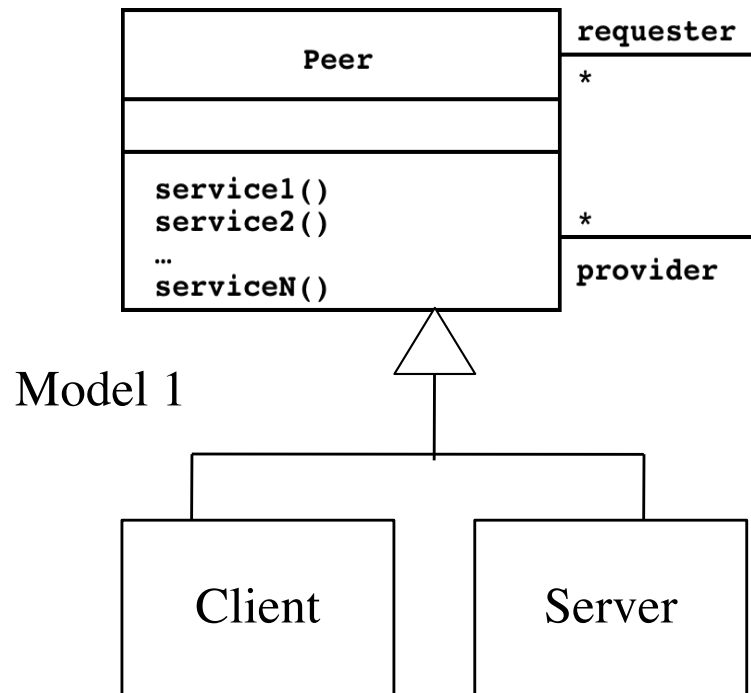
Client/Server Vs Peer-to-Peer

“Clients can be servers and servers can be clients”
How to Model?

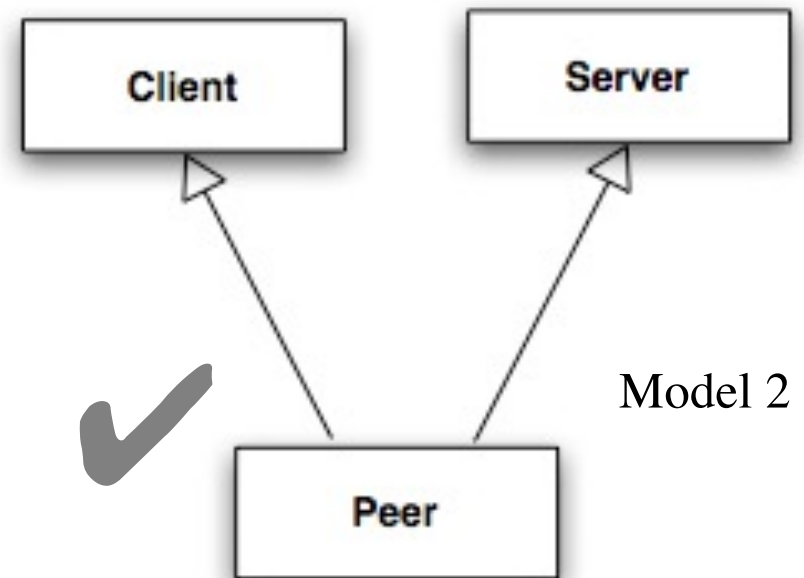
Which model is correct?

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”



?



3-Layer-Architectural Style (3-Tier Architecture)

Definition: 3-Layered Architectural Style

An architectural style, where an application consists of 3 hierarchically ordered subsystems

A user interface, middleware and a database system

The middleware subsystem services data requests between the user interface and the database subsystem

Definition: 3-Tier Architecture

A software architecture where the 3 layers are allocated on 3 separate hardware nodes

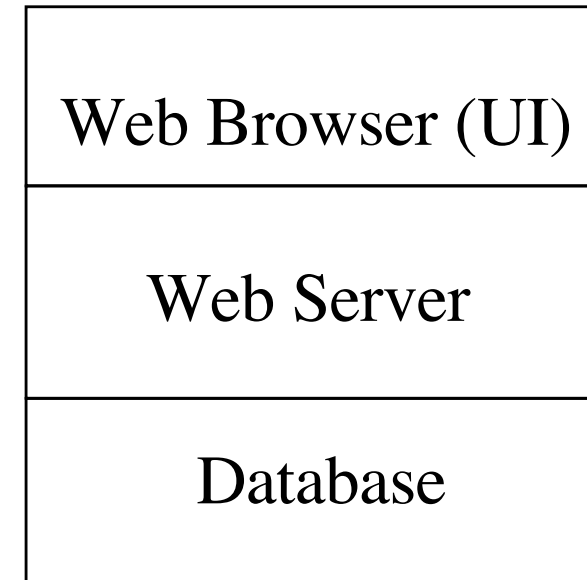
Note: **Layer** is a type (e.g. class, subsystem) and **Tier** is an instance (e.g. object, hardware node)

Layer and Tier are often used interchangeably.

Example of a 3-Layered Architectural Style

Three-Layered Architectural style are often used for the development of Websites:

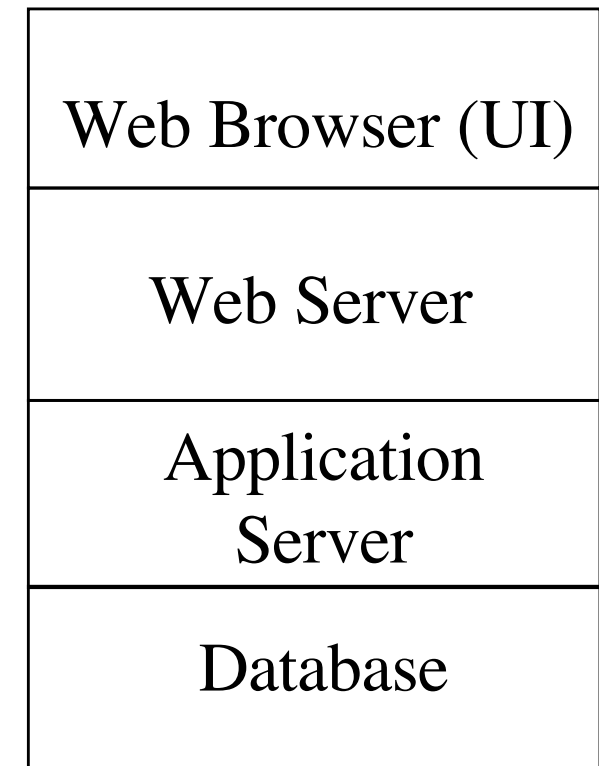
1. The **Web Browser** implements the user interface
2. The **Web Server** serves requests from the web browser
3. The **Database** manages and provides access to the persistent data.



Example of a 4-Layered Architectural Style

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

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



Model-View-Controller Architectural Style

Problem: In systems with high coupling changes to the user interface (boundary objects) often force changes to the entity objects (data)

The user interface cannot be 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 (MVC) style decouples data access (entity objects) and data presentation (boundary objects)

Views: Subsystems containing boundary objects

Model: Subsystem with entity objects

Controller: Subsystem mediating between Views (data presentation) and Models (data access).

Model-View-Controller Architectural Style

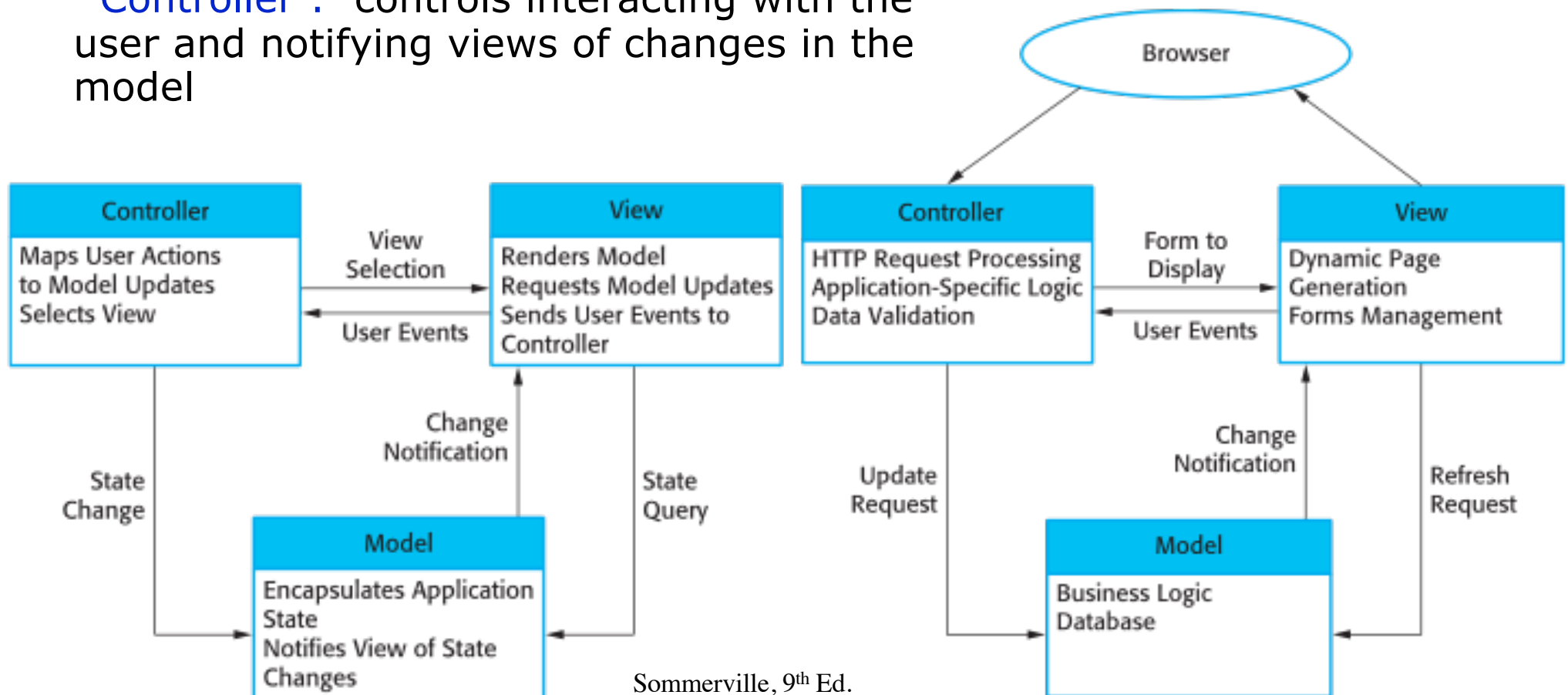
Subsystems are classified into 3 different types

Model : Provides application domain knowledge

View : Responsible for displaying information to the user

Controller : controls interacting with the user and notifying views of changes in the model

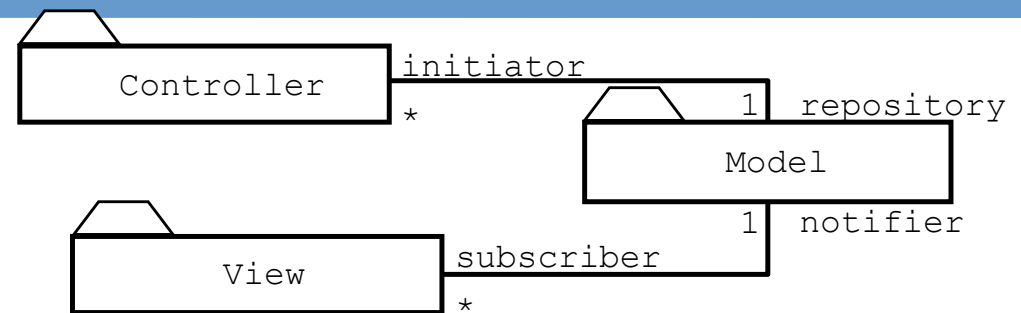
Example: Web Application in MVC



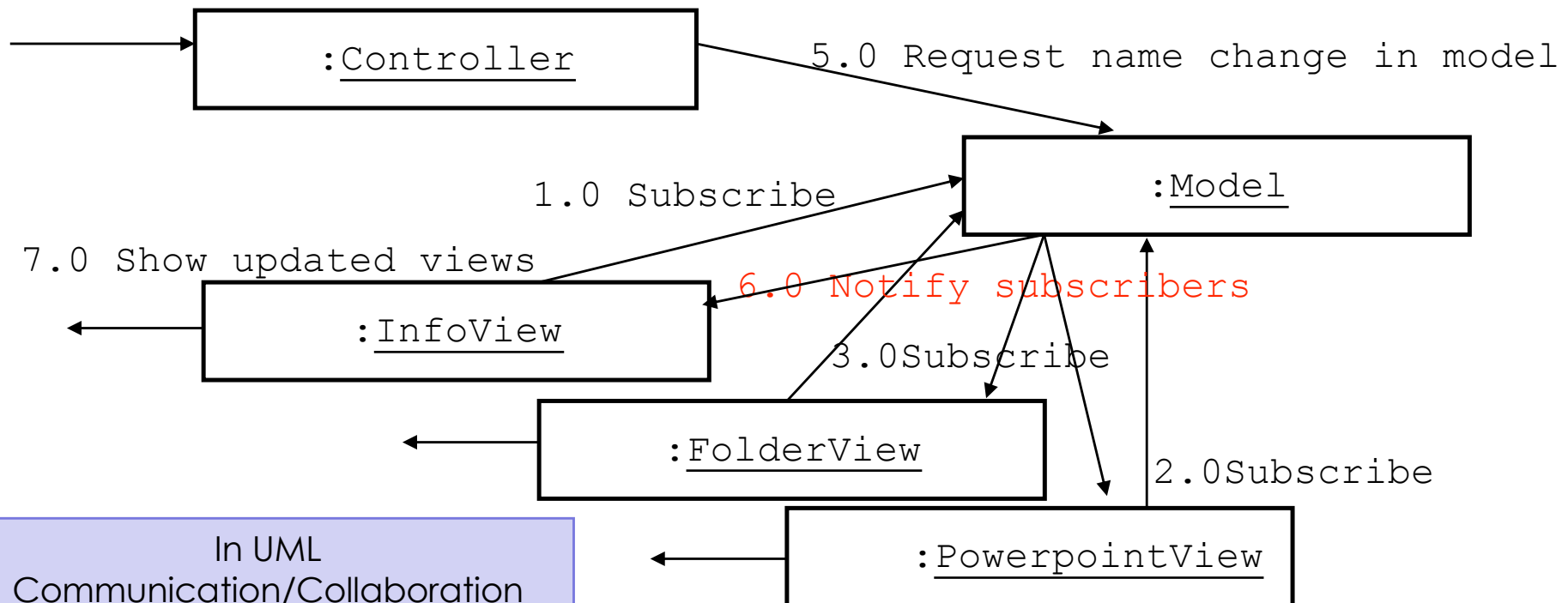
Sommerville, 9th Ed.

Example: Modeling the Sequence of Events in MVC

package
Diagram



4.0 User types new filename



In UML
Communication/Collaboration
Diagram

MVC vs. 3-Tier Architectural Style

The **MVC** architectural style is **nonhierarchical** (triangular):

- View subsystem sends updates to the Controller subsystem

- Controller subsystem updates the Model subsystem

- View subsystem is updated directly from the Model

The **3-tier** architectural style is **hierarchical** (linear):

- The presentation layer never communicates directly with the data layer (opaque architecture)

- All communication must pass through the middleware layer

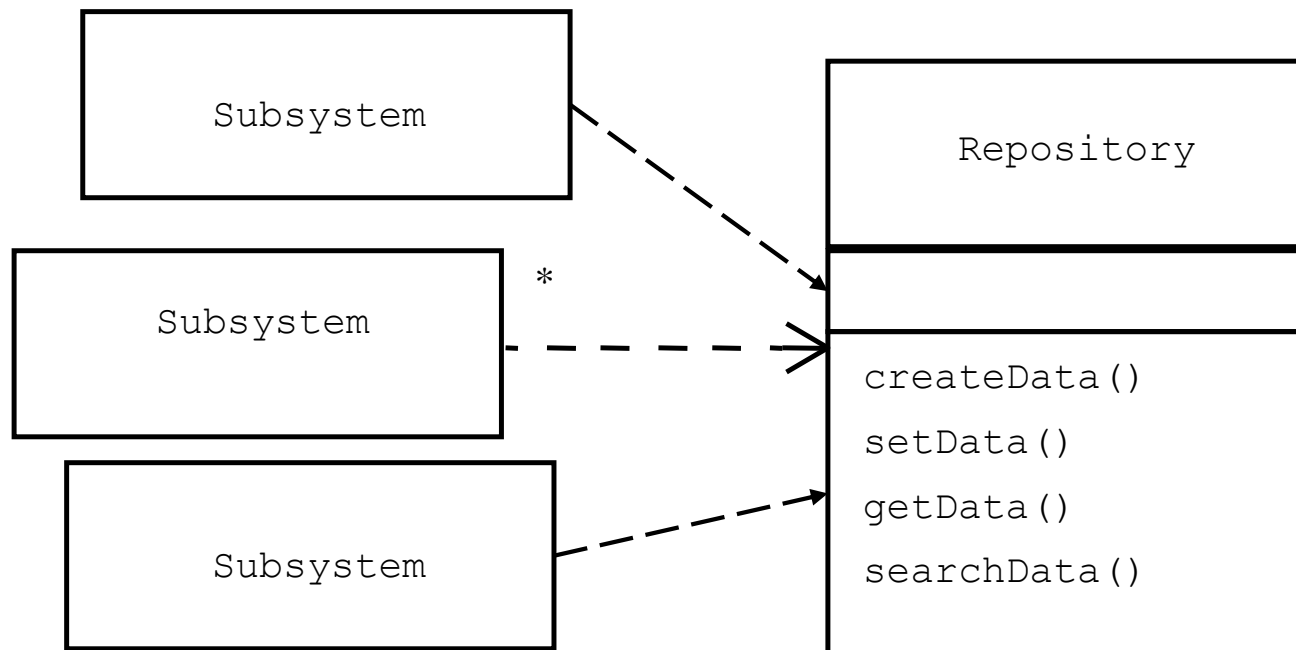
History:

- MVC (1970-1980): Originated during the development of modular graphical applications for a single graphical workstation at Xerox Parc

- 3-Tier (1990s): Originated with the appearance of Web applications, where the client, middleware and data layers ran on physically separate platforms.

Repository Architectural Style

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



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

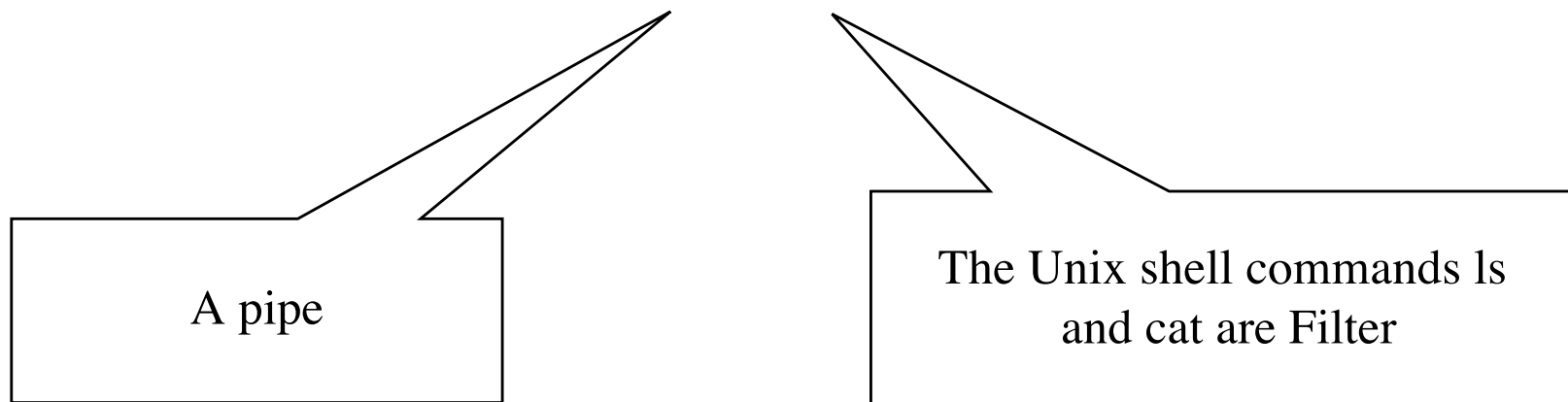
Pipe: A Pipe is a connection between two processing steps

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**



Example of Design Goals

To whom (the customer, end-user or developer) each of the following design goals is most important to?

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

Summary

System Design

Focuses on finding an optimal solution combining software and hardware approaches

Design Goals

Evaluates important system features against alternative designs (design-tradeoffs)

Considers General Design Goals and principles (cohesion and coupling) and system specific ones (from non-functional requirements) in design

System Composition

Modularising the system from parts or modules in creating an optimal design based on the general and specific design goals/principles.

Architectural Style

Choosing a suitable pattern of system layout (architectural style) that meets the needs of identified design goals; from layer styles (C/S, SOA, n-Tier), MVC, Repository, Pipes & Filters

Software architecture

An instance of an architectural style: e.g. client/server, SOA etc.