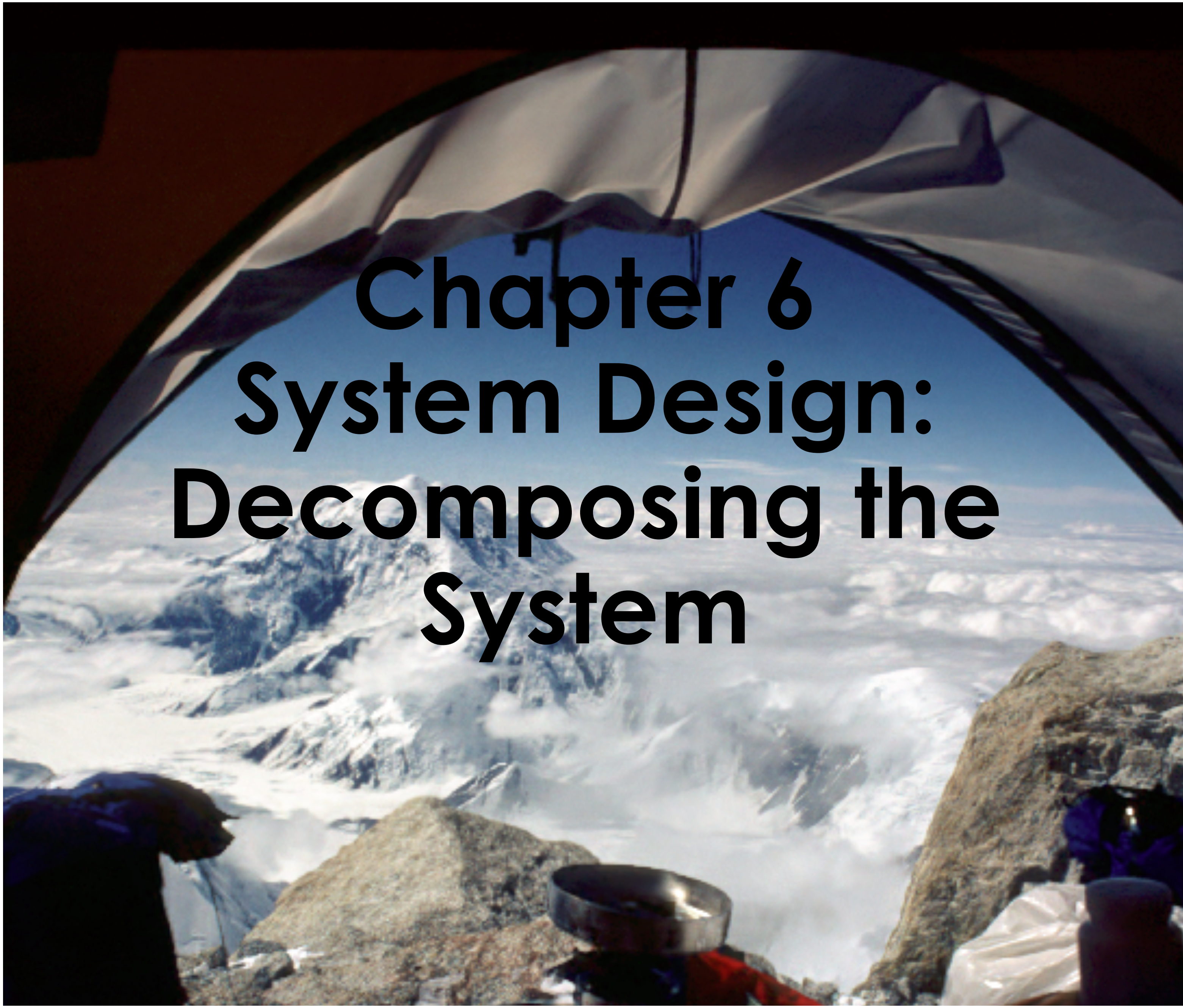
Object-Oriented Software Engineering
Using UML, Patterns, and Java

Chapter 6 System Design: Decomposing the System



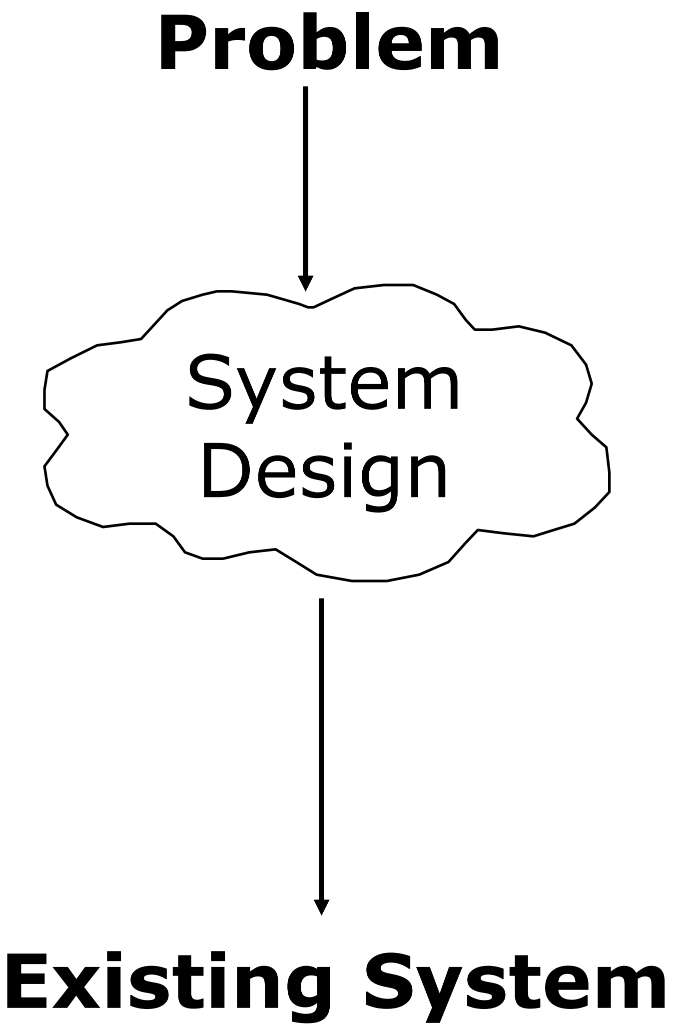
The Scope of System Design

- Bridge the gap
  - between a problem and an existing system in a manageable way

- How?
  - Use Divide & Conquer:

  1) Identify design goals
  2) Model the new system; design as a set of subsystems

  3-8) Address the major design goals.



System Design: Eight Issues

System Design

1. Identify Design Goals
- Additional NFRs
- Trade-offs

2. Subsystem Decomposition
- Layers vs Partitions
- 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

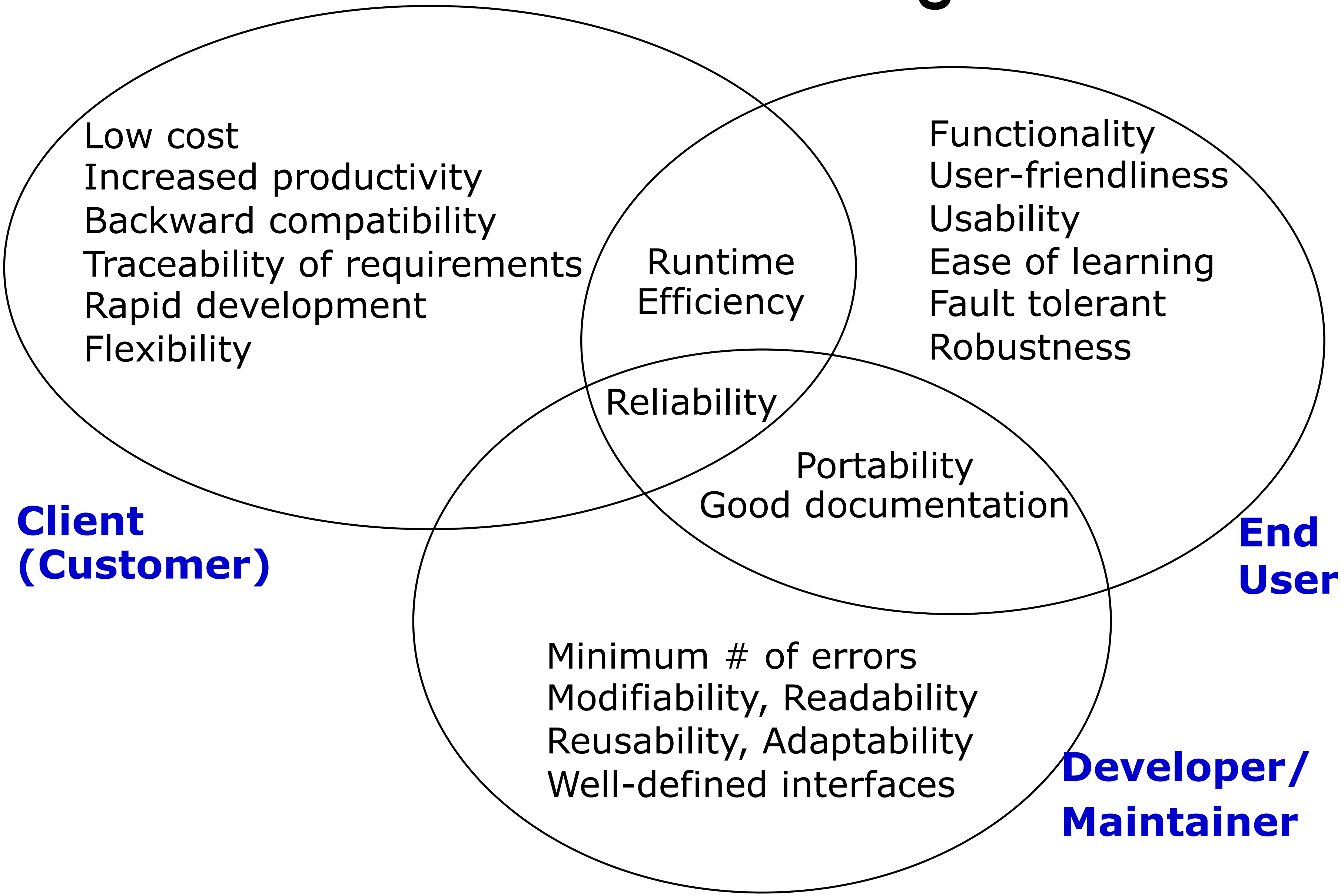
7. Software Control
- Monolithic
- Event-Driven
- Conc. Processes

8. Boundary Conditions
- Initialization
- Termination
- Failure.

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

Stakeholders have different Design Goals



Typical Design Trade-offs

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

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.

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 pattern (=> lecture on design patterns).

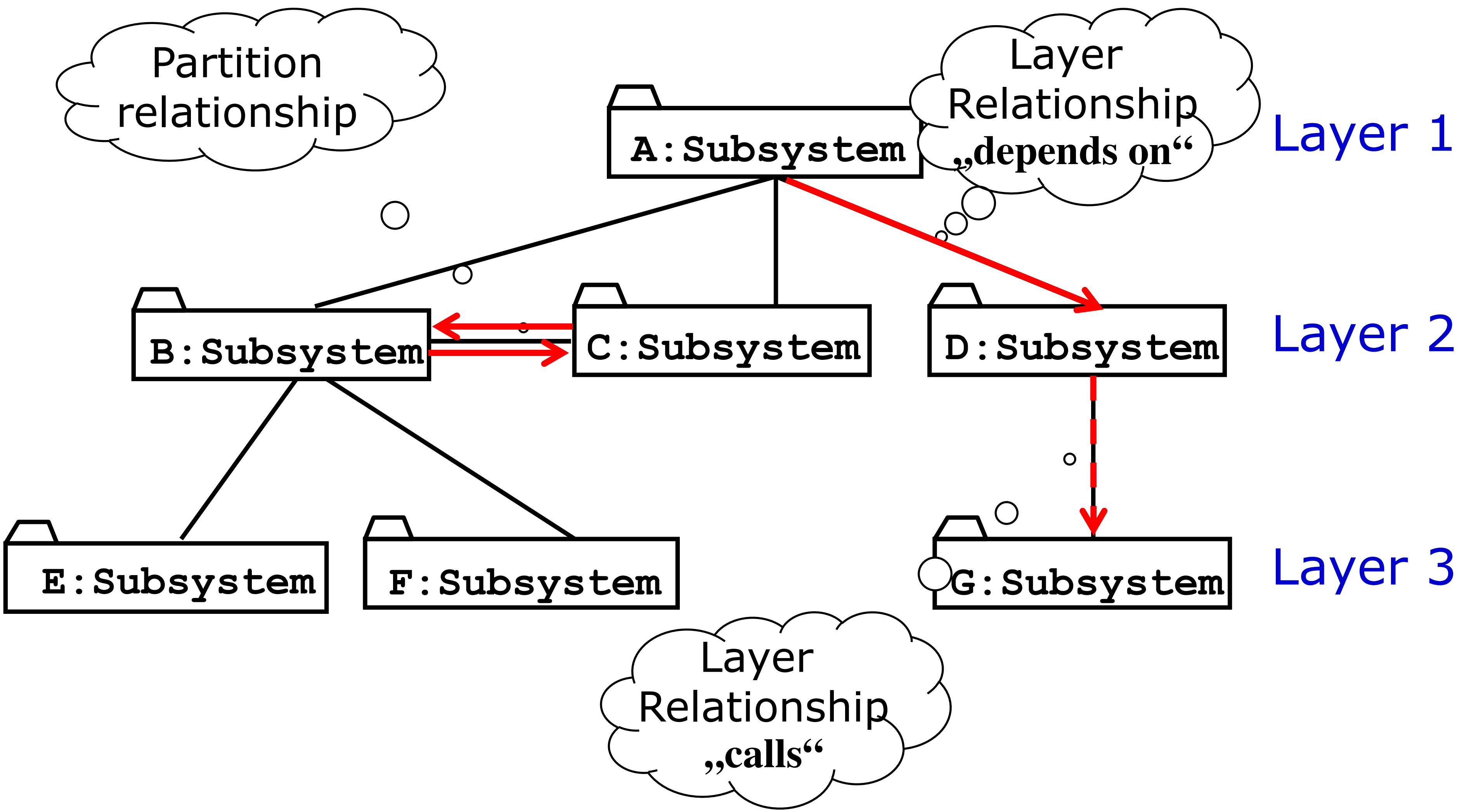
Properties of Subsystems: Layers and Partitions

- 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



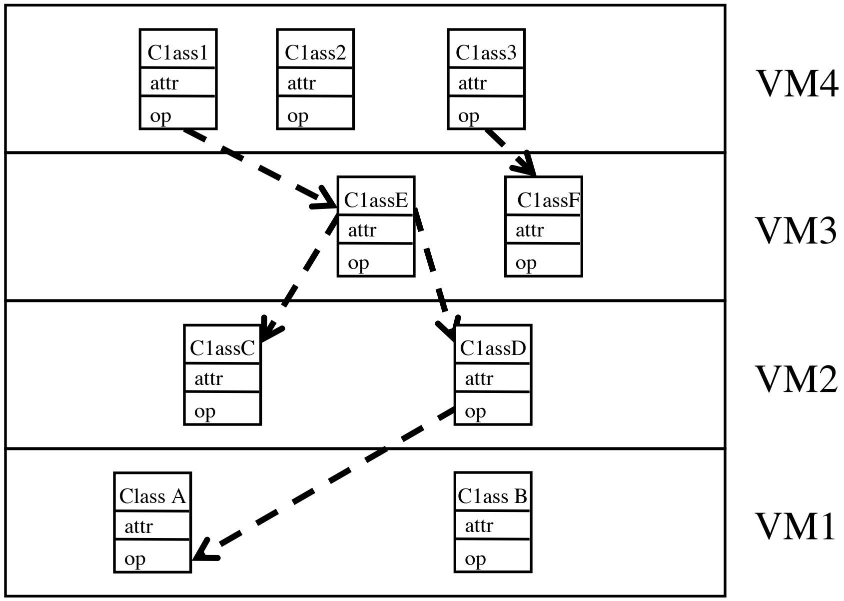
Virtual Machine

- 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

Closed Architecture (Opaque Layering)

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

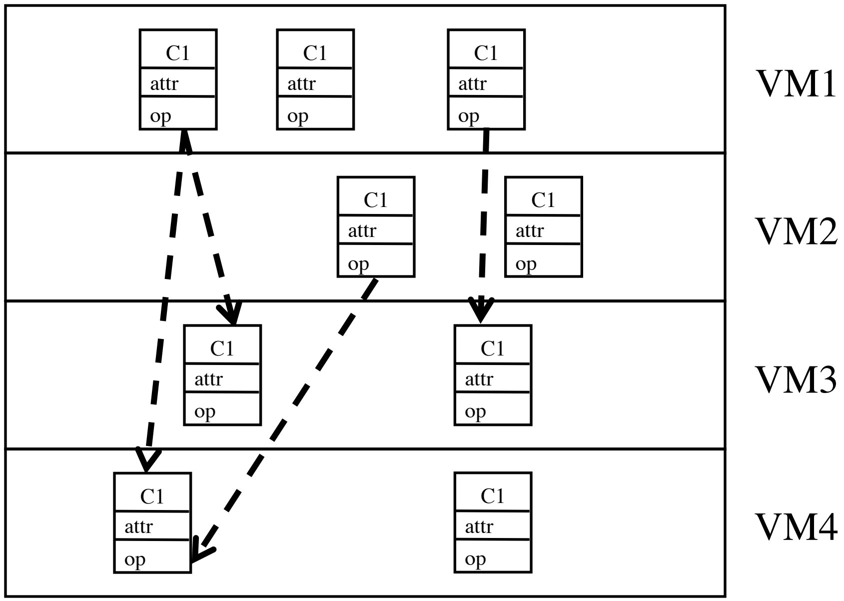
Design goals: Maintainability, flexibility.



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

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)
- 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?

Architectural Style vs Architecture

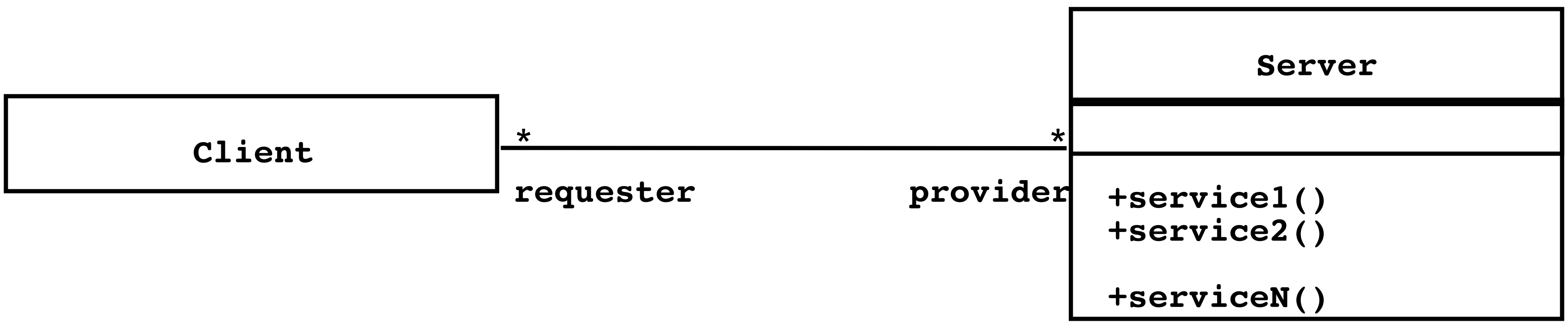
- Subsystem decomposition: Identification of subsystems, services, and their association to each other (hierarchical, peer-to-peer, etc)
- Architectural Style: A pattern for a subsystem decomposition
- Software Architecture: Instance of an architectural style.

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.

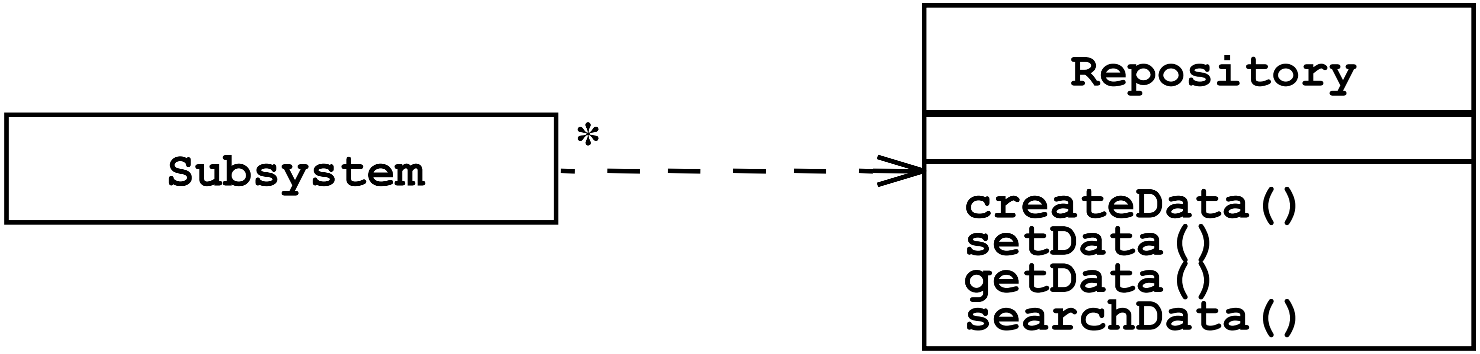


Client/Server Architectures

- 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

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



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

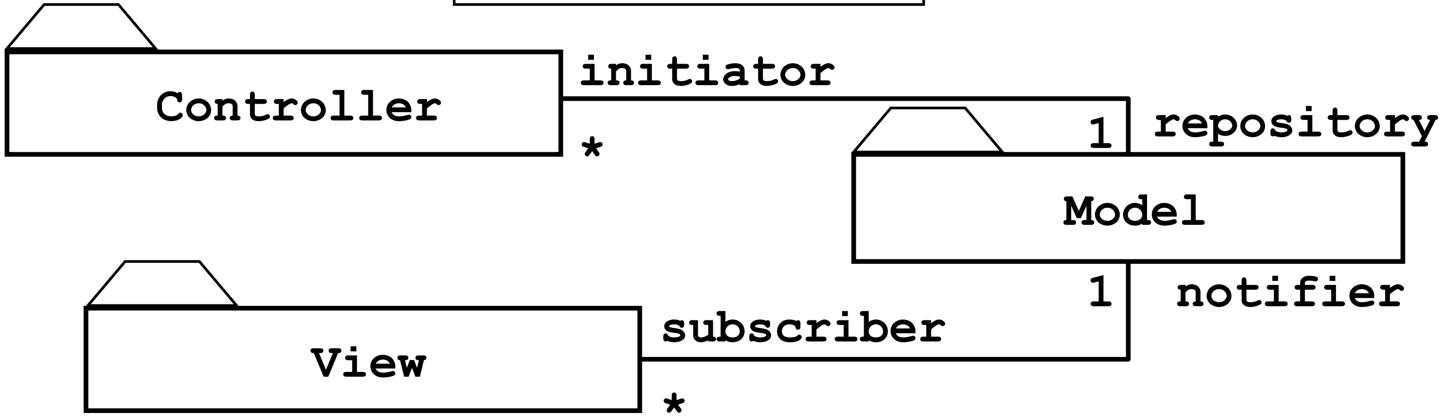
- Subsystems are classified into 3 different types

  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

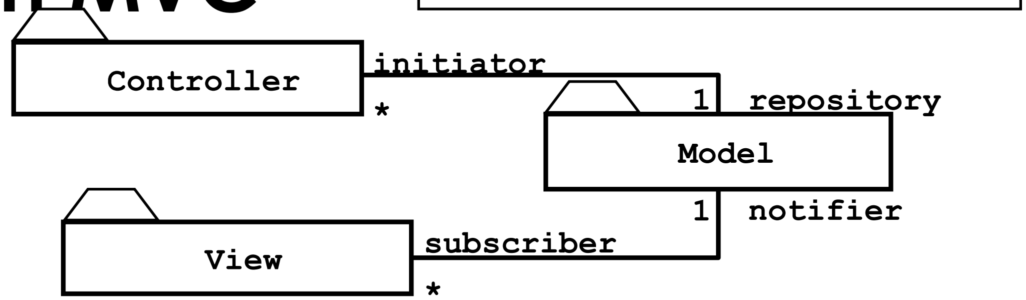
Class Diagram

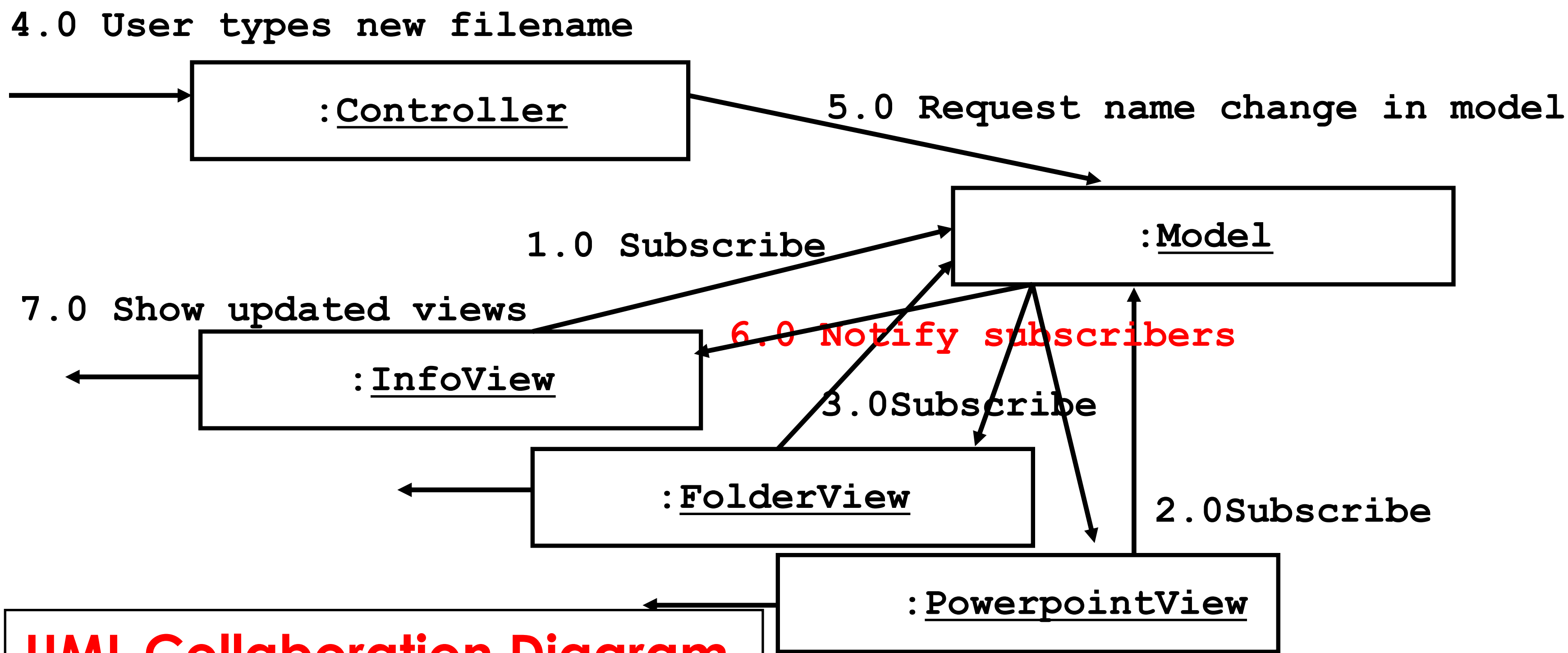


Better understanding with a Collaboration Diagram

Example: Modeling the Sequence of Events in MVC

UML Class Diagram





UML Collaboration 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.

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.

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

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

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.