Modern methods in Software Engineering

System Design

Literature used

- Text Book

    Chapters 6 and 7

Content

Overview of System Design Activities:

1. Design Goals
2. Subsystem Decomposition
3. Other activities

   Hardware/Software Mapping

   Persistent Data Management

   Global Resource Handling and Access Control

   Software Control

   Boundary Conditions

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."

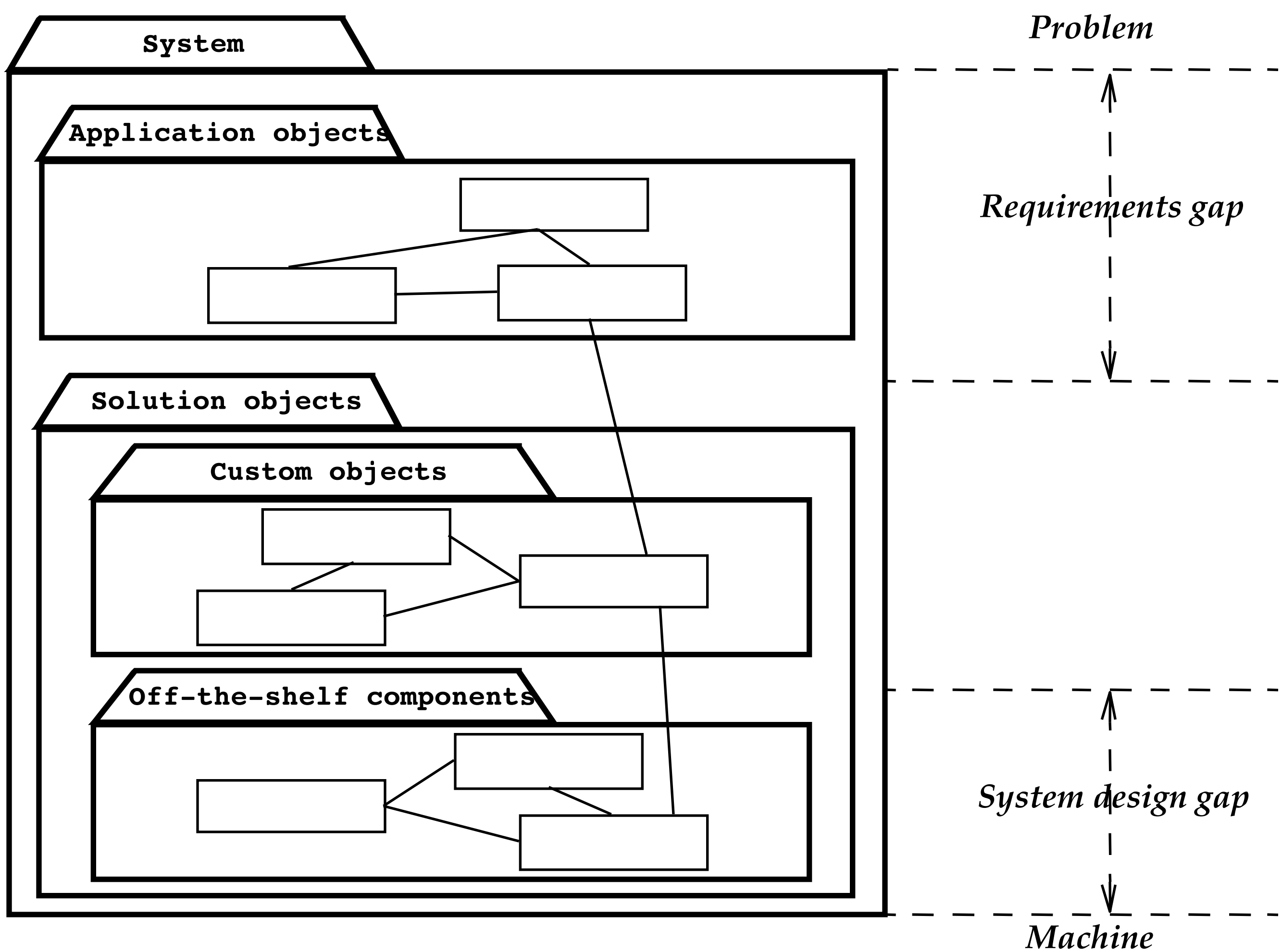
- C.A.R. Hoare

Analysis & Design

- *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 updating knowledge in solution domain: About 3-5 years
    - Cost of hardware rapidly sinking
- "Design window":
  - Time in which design decisions have to be made

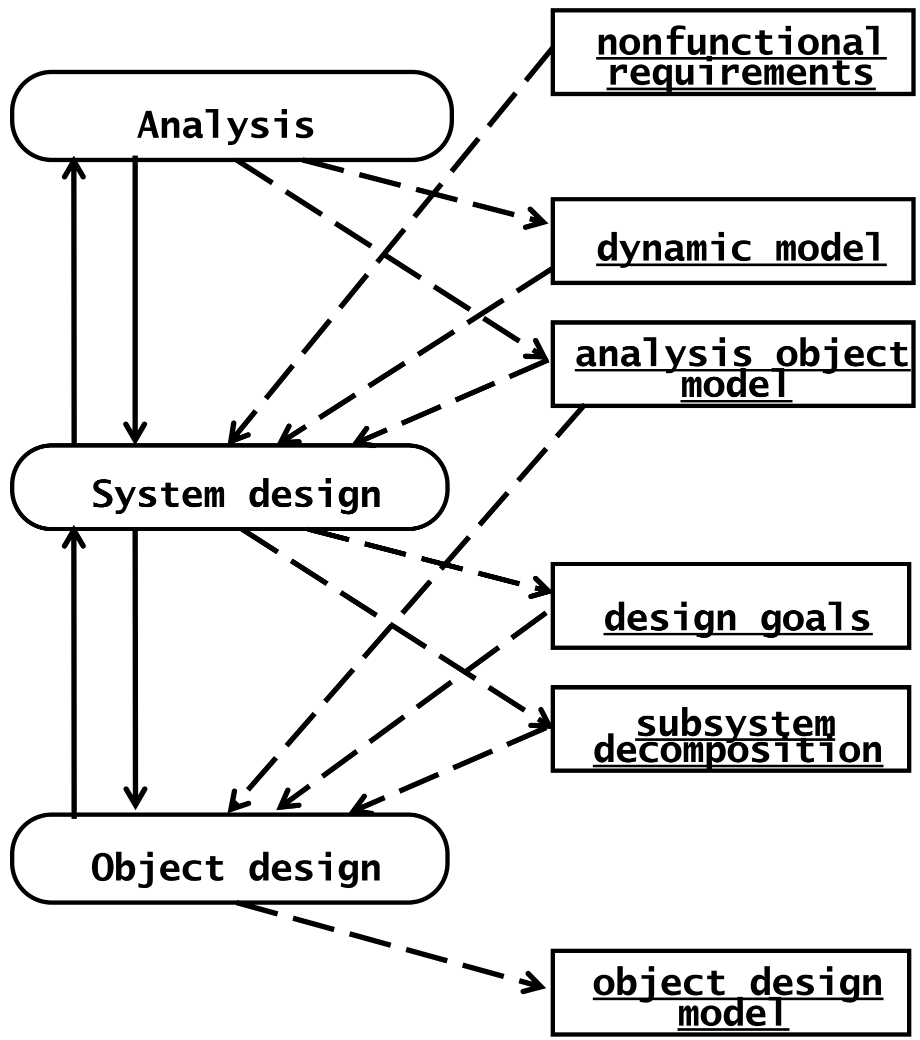
Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

System Design:



Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

System Design Activities



Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

System Design

1. Design Goals
- Definition
- Trade-offs

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

3. Concurrency
- Identification of Threads of control

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

5. Data Management
- Persistent Objects
- 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

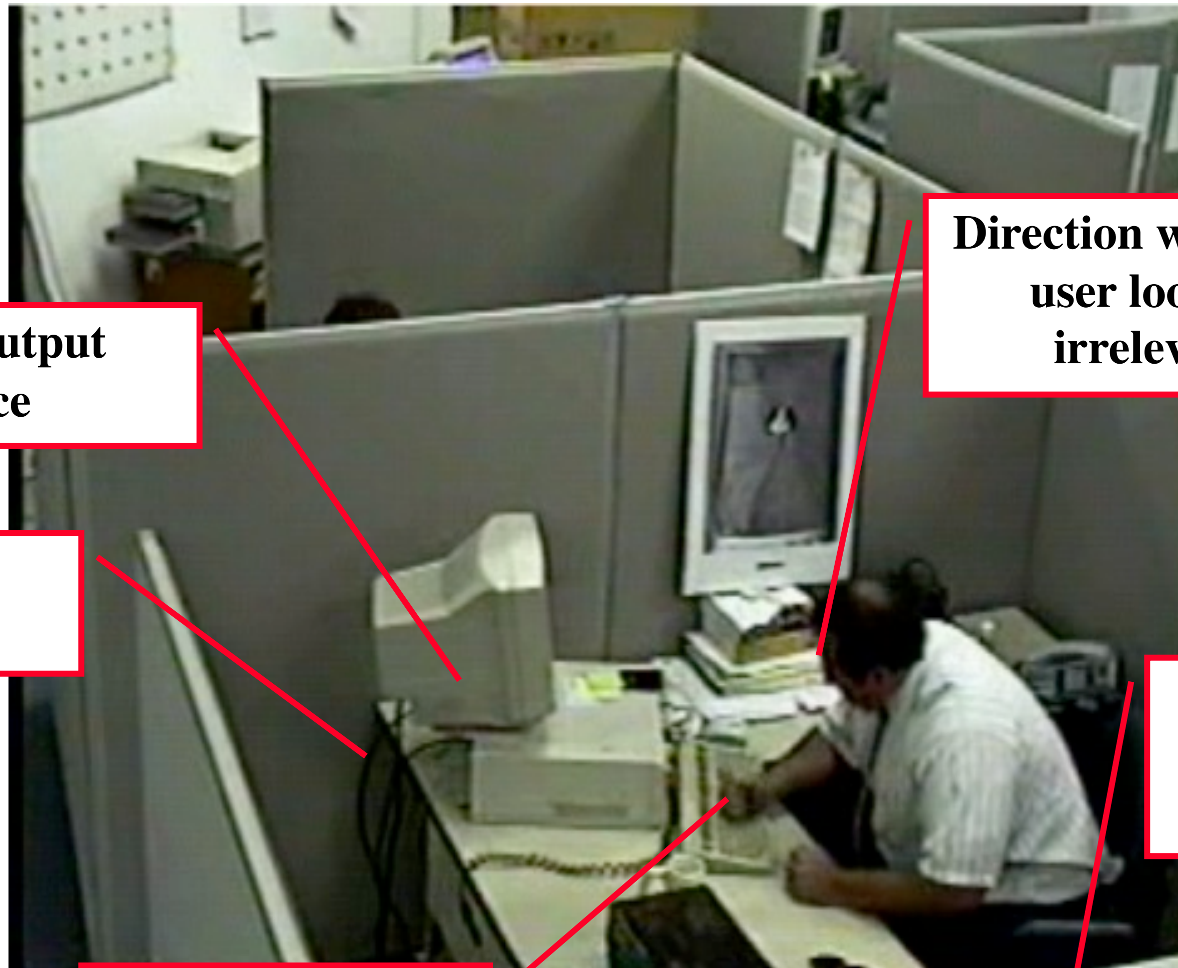
How to use the results from the Requirements Analysis for System Design

- 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
  - Activity 8: Boundary conditions

Example: Current Desktop Development

Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Identify Current Technology Constraints



Single Output Device

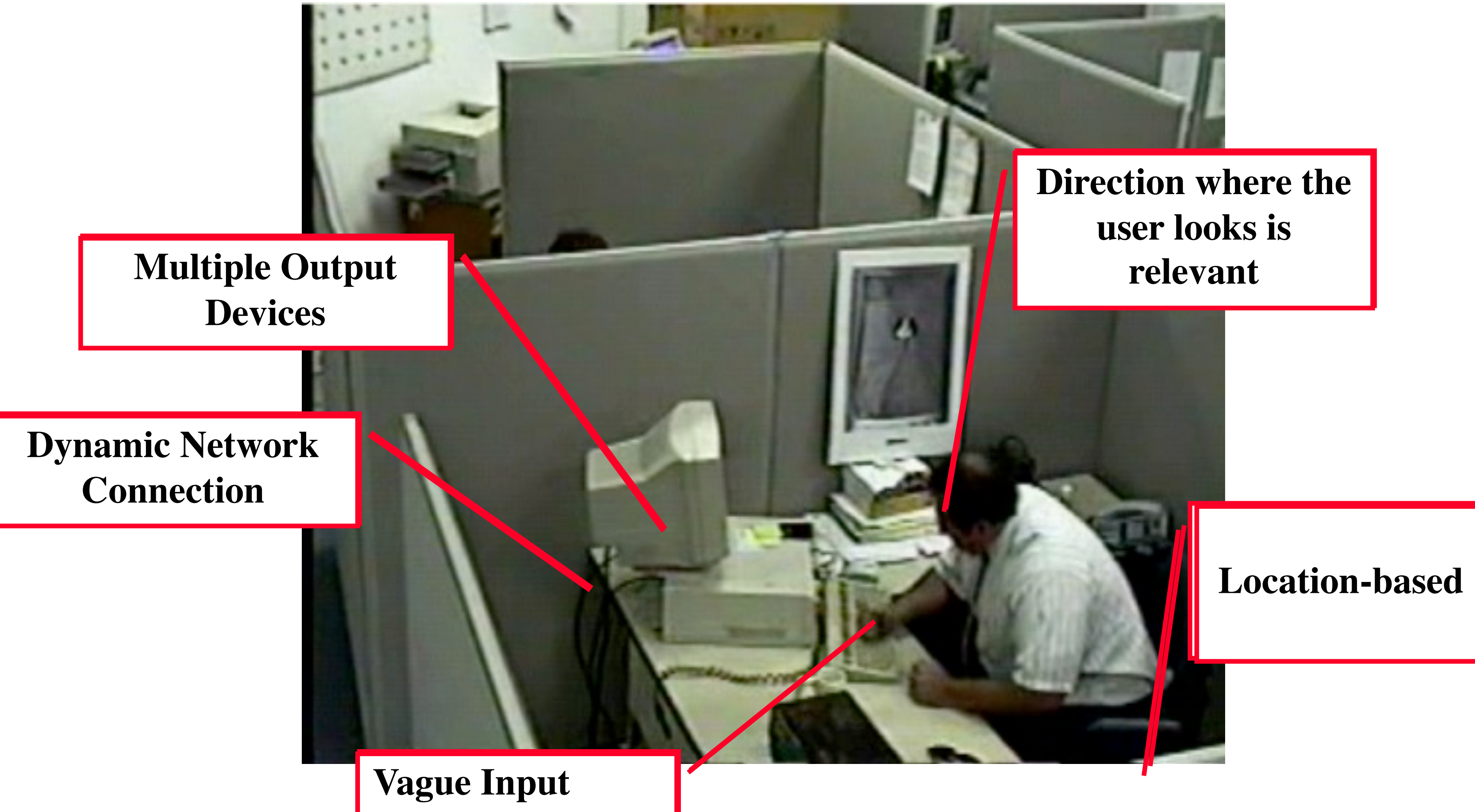
Fixed Network Connection

Precise Input

Direction where the user looks is irrelevant

Location of user does not matter

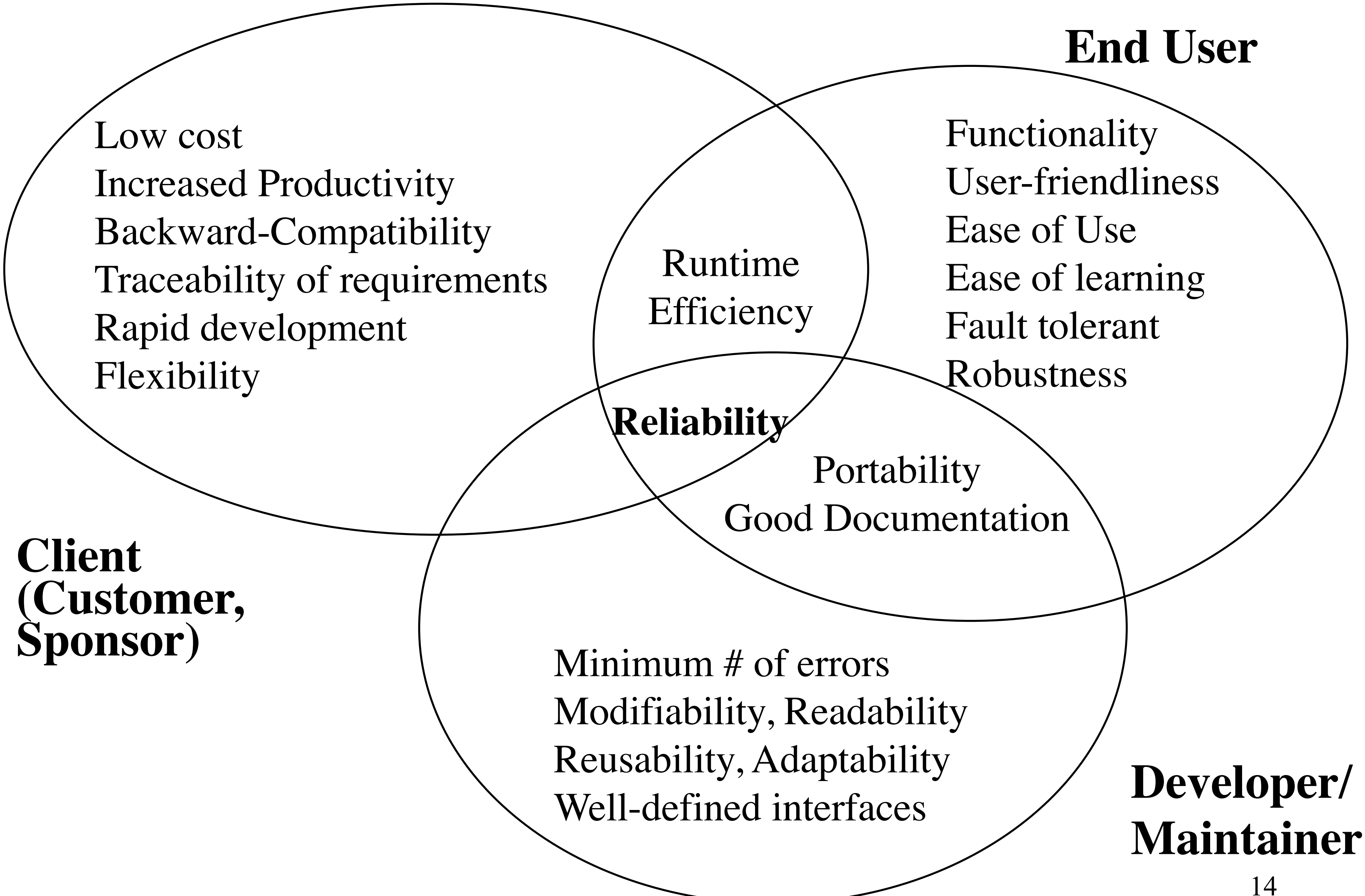
Generalize Constraints using Technology Enablers



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

Relationships among Design Goals



Design goals grouping

- Performance
  - Response time
  - Throughput
  - Memory
- Dependability
  - Robustness
  - Reliability
  - Availability
  - Fault tolerance
  - Security
  - Safety
- End-user criteria
  - Utility
  - Usability
- Cost
  - Development cost
  - Deployment cost
  - Upgrade cost
  - Maintenance cost
  - Administration cost
- Maintenance
  - Extensibility
  - Modifiability
  - Adaptability
  - Portability
  - Readability
  - Traceability of requirements

Design Trade-offs

- Space vs. Speed
- Efficiency vs. Portability
- Rapid development vs. Functionality
- Cost vs. Reusability
- Backward Compatibility vs. Readability
- Delivery time vs. Quality
- Delivery time vs. Functionality
- …

System Decomposition

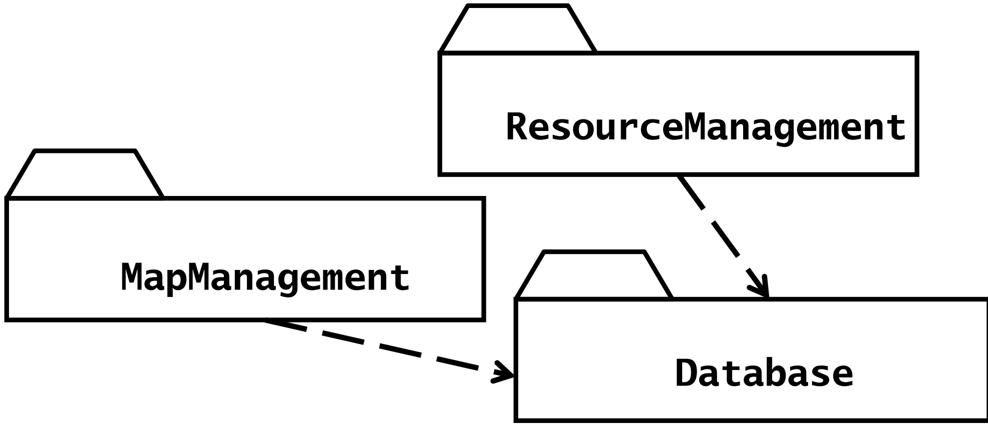
- Subsystem
  - Collection of classes, associations, operations, events and constraints that are interrelated
  - Seed for subsystems: UML Use cases, Objects and Classes.
- Service:
  - Group of operations provided by the subsystem (a set of related operations that share a common purpose)
  - Seed for services: Subsystem use cases
- Service is specified by Subsystem interface:
  - Specifies interaction and information flow from/to subsystem boundaries, but not inside the subsystem.

Coupling and Cohesion

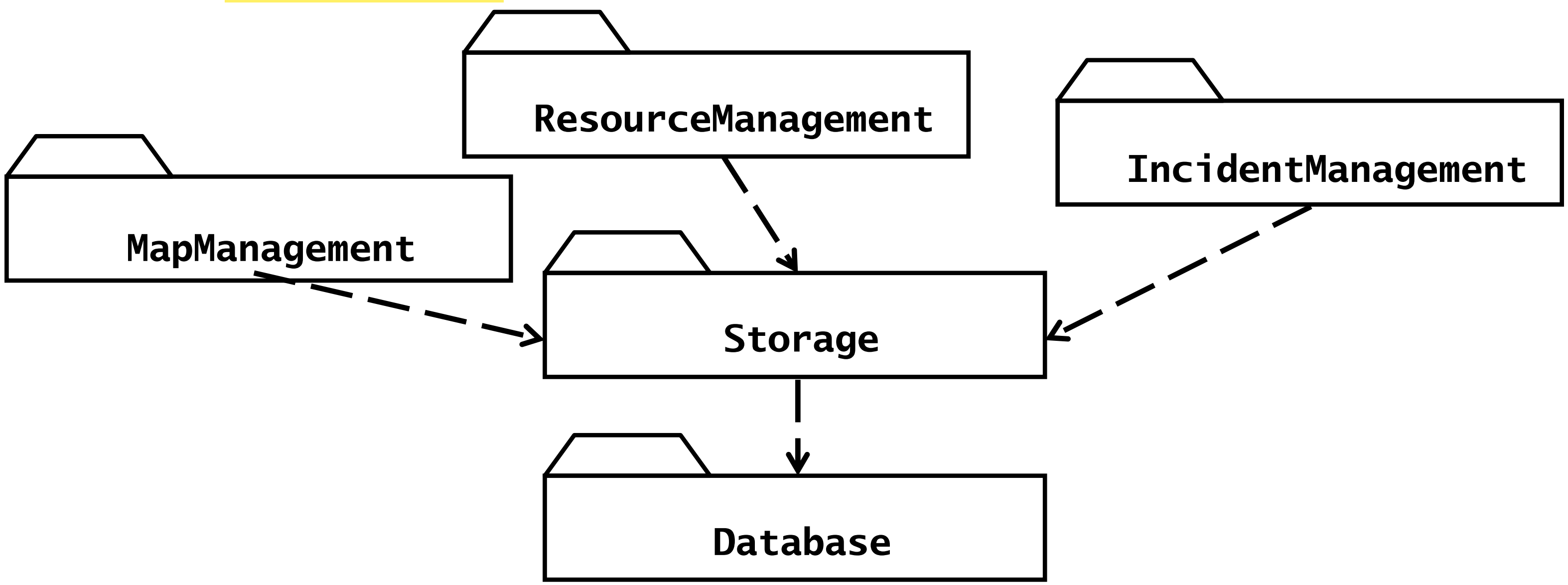
- Goal: Reduction of ***complexity while change occurs***
- Coupling measures dependencies between subsystems
  - High coupling: Changes to one subsystem will have high impact on the other subsystem (change of model, massive recompilation, etc.)
  - Low coupling: A change in one subsystem does not affect any other subsystem
- Cohesion measures the dependence within the system
  - 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
- Subsystems should have as maximum cohesion and minimum coupling as possible:
  - How can we achieve high cohesion?
  - How can we achieve loose coupling?

Reducing coupling

Alternative 1: Direct access to the Database subsystem

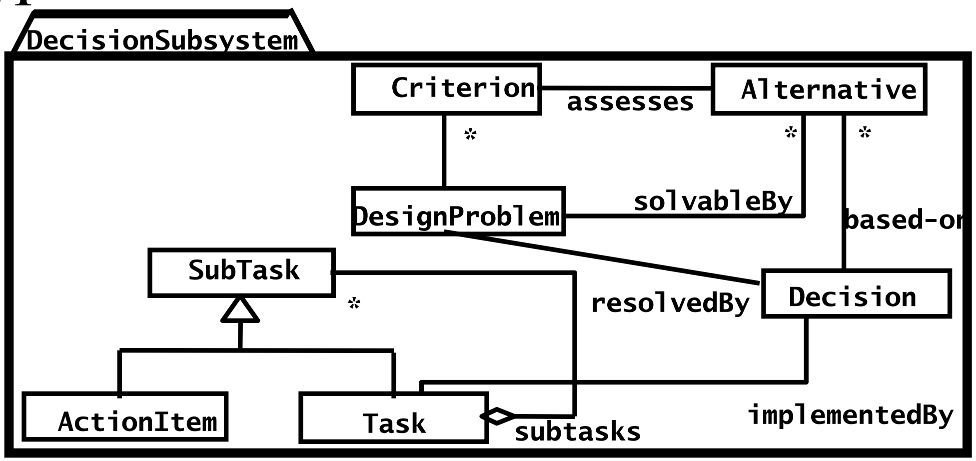


Alternative 2: Indirect access to the Database through a Storage subsystem

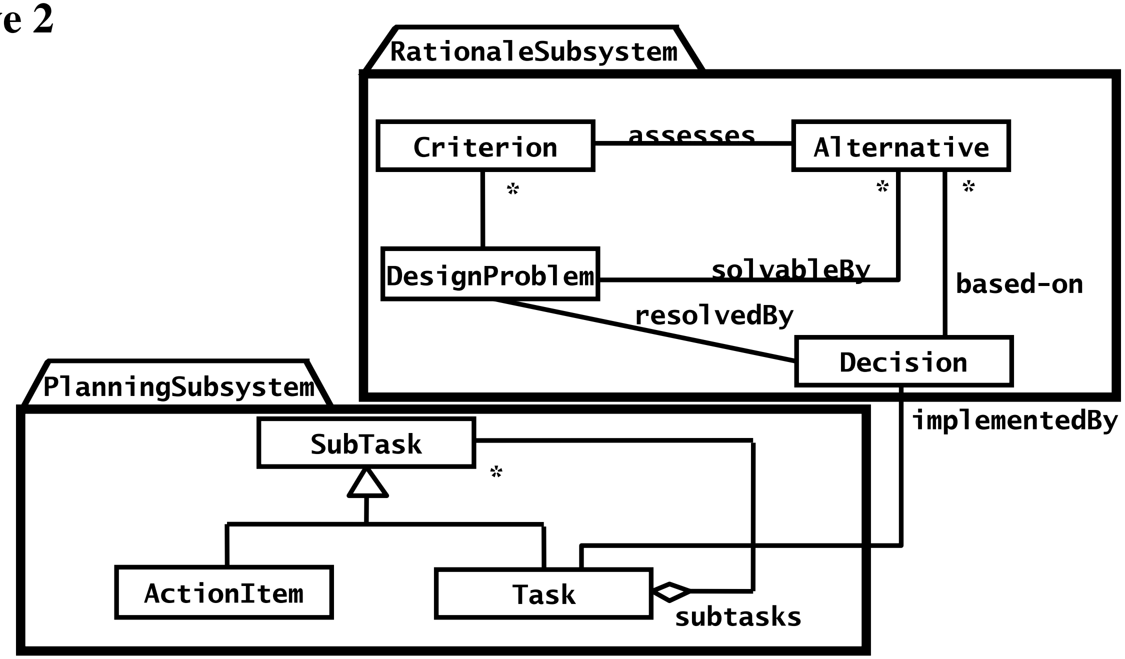


Increasing cohesion

Alternative 1



Alternative 2



Partitions and Layers

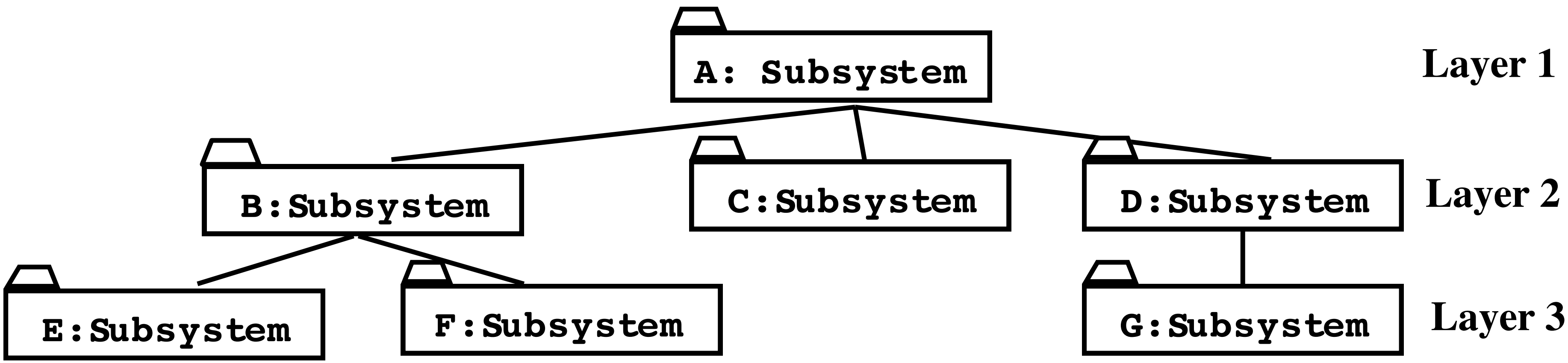
Partitioning and layering are techniques to achieve low coupling.

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

Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Subsystem Decomposition into Layers

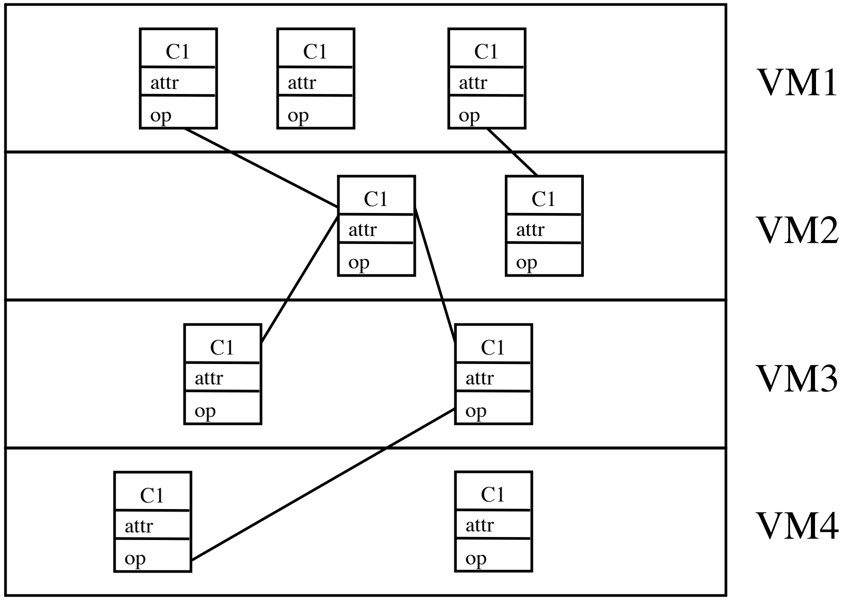


- Subsystem Decomposition Heuristics:
  - No more than 7+/-2 subsystems at any layer
    - More subsystems increase cohesion but also complexity (more services)
  - No more than 7+/-2 layers, use 3 layers (good)

Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Closed Architecture (Opaque Layering)

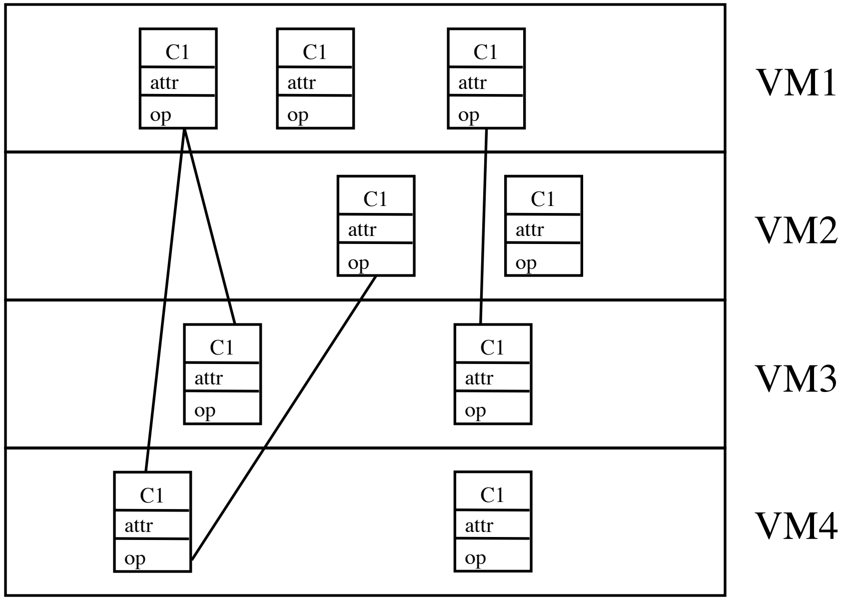
- Any layer can only invoke operations from the immediate layer below
- Design goal: **High maintainability, flexibility**



Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Open Architecture (Transparent Layering)

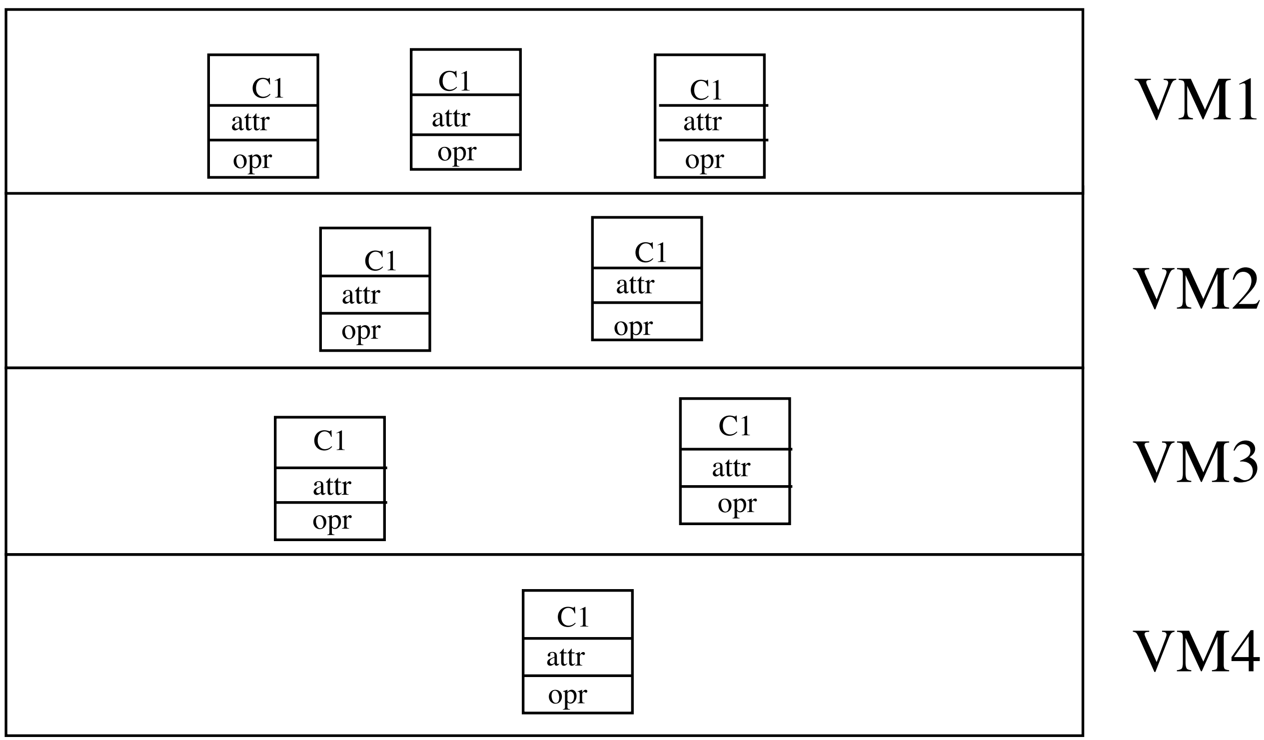
- Any layer can invoke operations from any layers below
- Design goal: **Runtime efficiency**



Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Virtual Machine

Problem



Existing System

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.
- Layered systems often have a chicken-and egg problem
  - Example: Debugger

Choosing Subsystems

- Criteria for subsystem selection: Most of the interaction should be within subsystems (High cohesion), rather than across subsystem boundaries (Low coupling).
- Primary Question:
  - What kind of service is provided by the subsystems (subsystem interface)?
- Secondary Question:
  - Can the subsystems be hierarchically ordered (layers)?

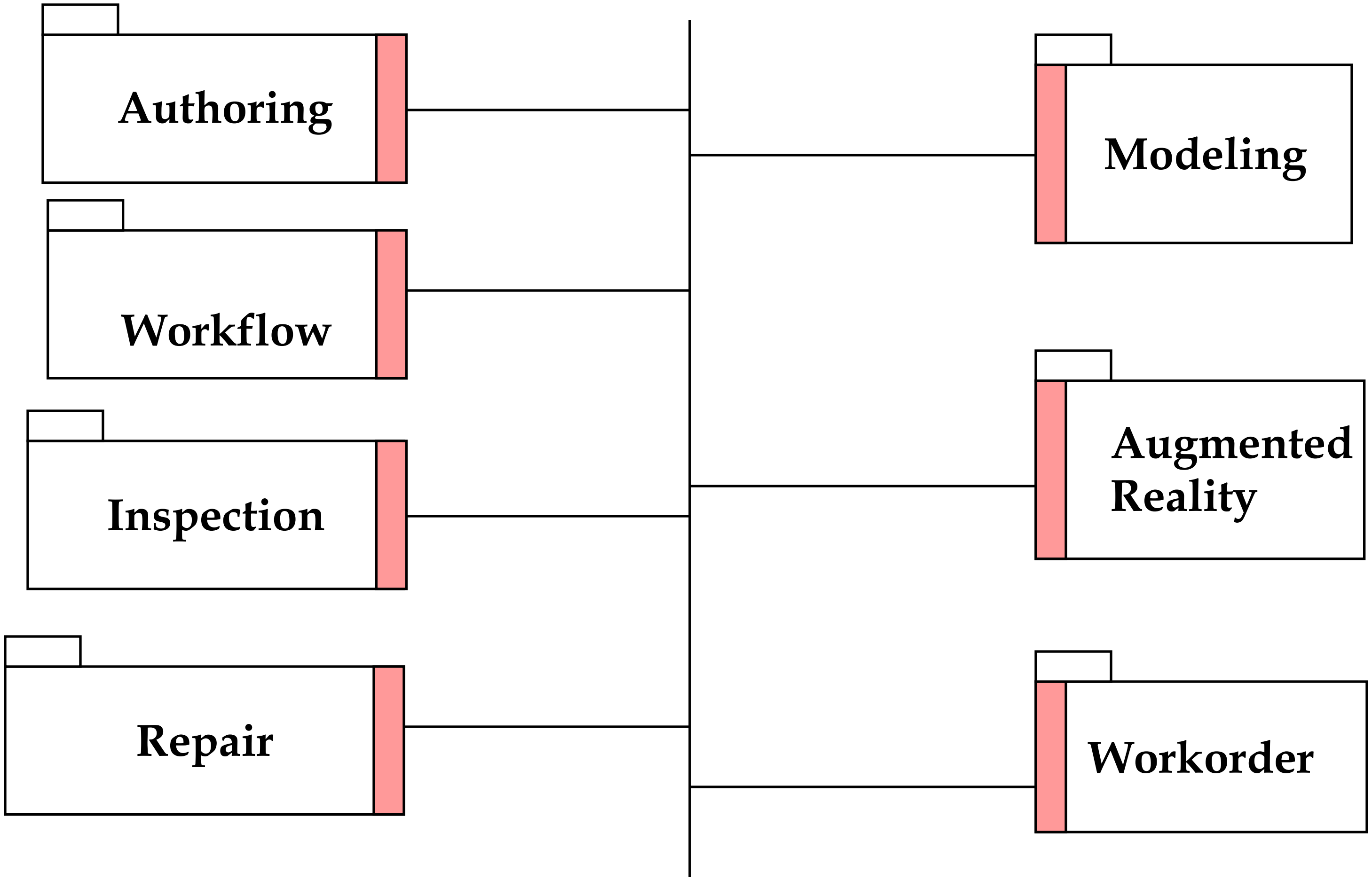
Heuristics for grouping objects into subsystems

- Assign objects identified in one use case into the same subsystem
- Minimize the number of associations crossing subsystem boundaries
- All objects in the same subsystem should be functionally related
- …

Software Architectural Styles

- Subsystem decomposition
  - Identification of subsystems, services, and their relationship to each other.
- Specification of the system decomposition is critical.
- Patterns for software architecture
  - Client/Server
  - Peer-To-Peer
  - Repository
  - Model/View/Controller
  - Pipes and Filters
  - …

System as a set of subsystems communicating via a software bus

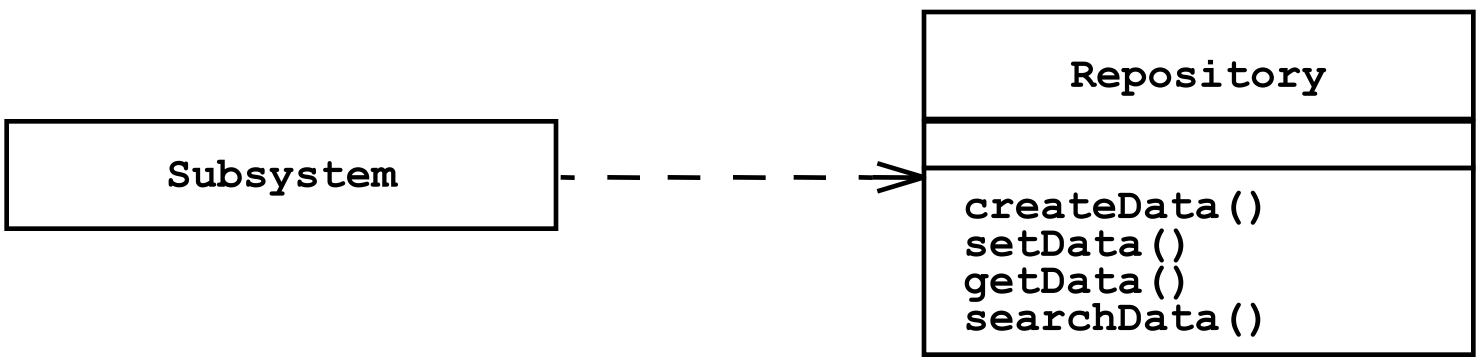


A Subsystem Interface Object publishes the service (= Set of public methods) provided by the subsystem

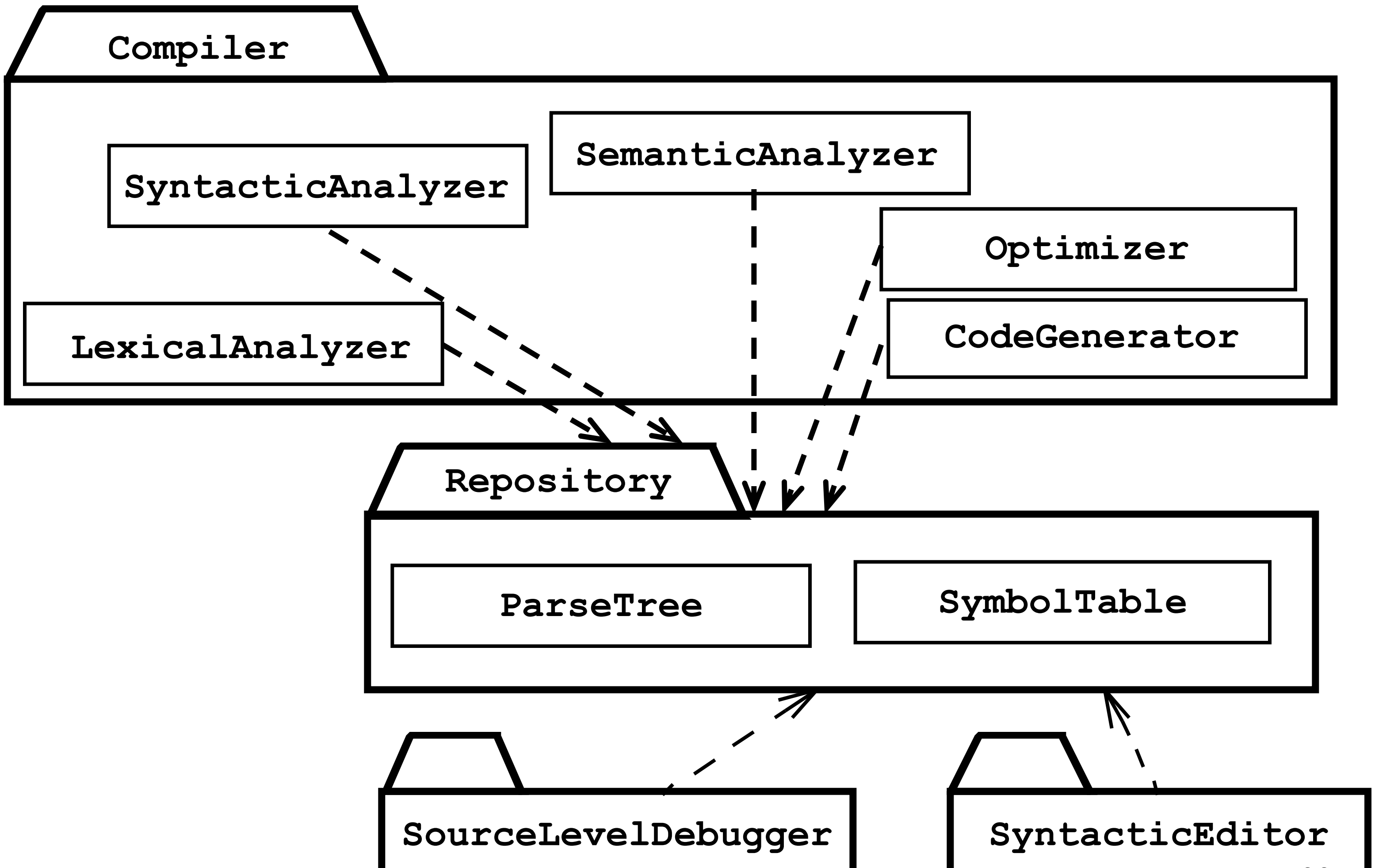
Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Repository Architectural Style

- 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)



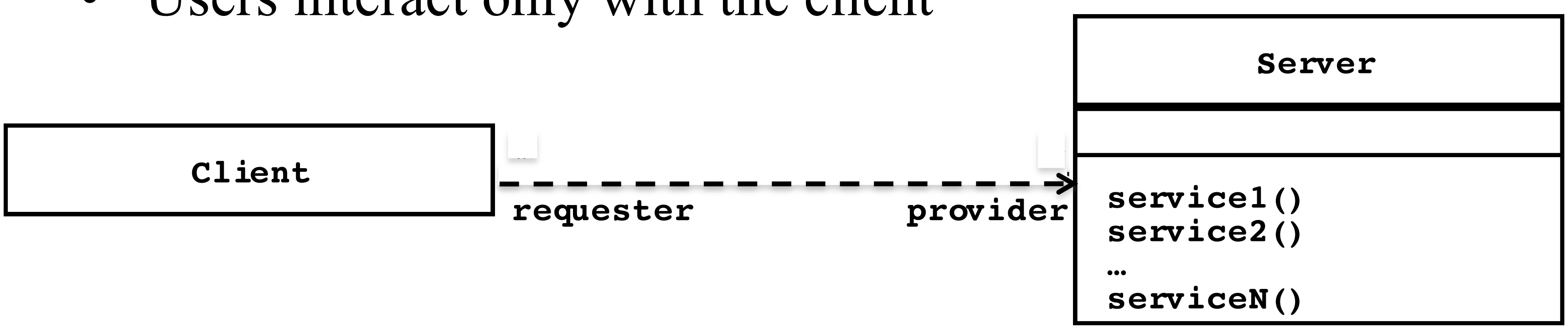
Repository Architectural Style



Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Client/Server Architectural Style

- One or many servers provide 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 immediate
- Users interact only with the client



Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Client/Server Architectural Style

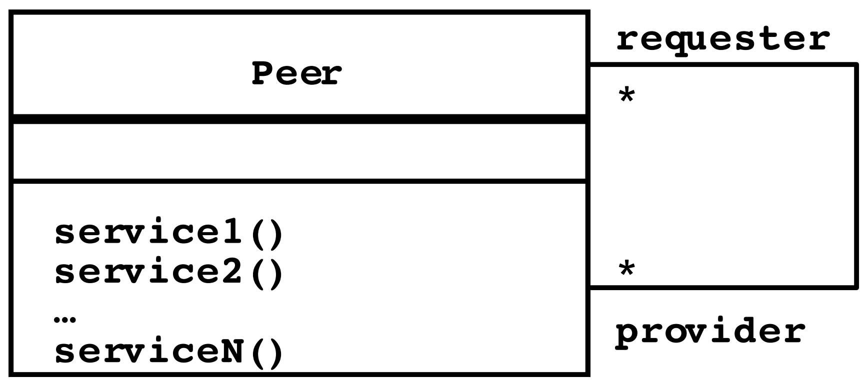
- Often used in database systems:
  - Front-end: User application (client)
  - Back end: Database access and manipulation (server)
- Functions performed by client:
  - Customized user interface
  - Front-end processing of data
  - Initiation of server remote procedure calls
  - Access to database server 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)

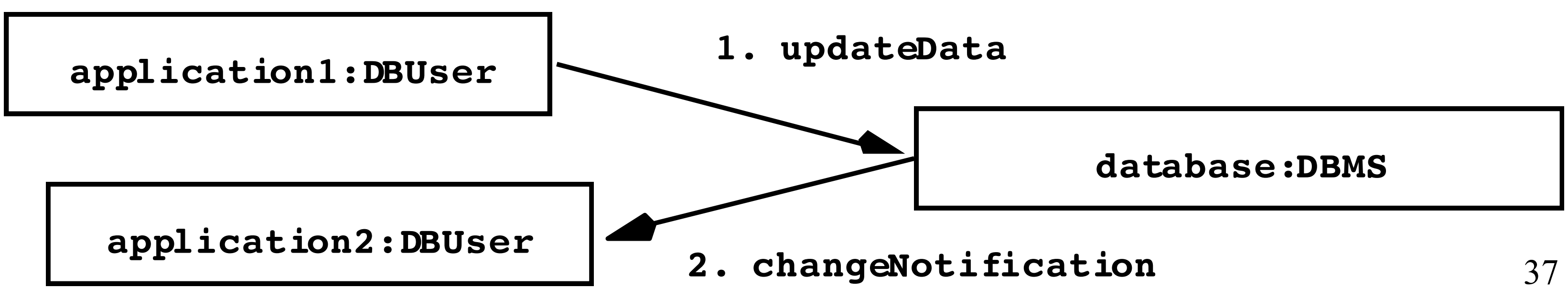
Design Goals for Client/Server Architectural Style

- *Service Portability*
  - Service 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, but should provide a single "logical" service to the user
- *Performance*
  - 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. wearable computer, desktop)
- *Reliability*
  - System should survive node or communication link problems

Peer-to-Peer Architectural Style

- Generalization of Client/Server Architecture
- Clients can be servers and servers can be clients
- More difficult because of possibility of deadlocks

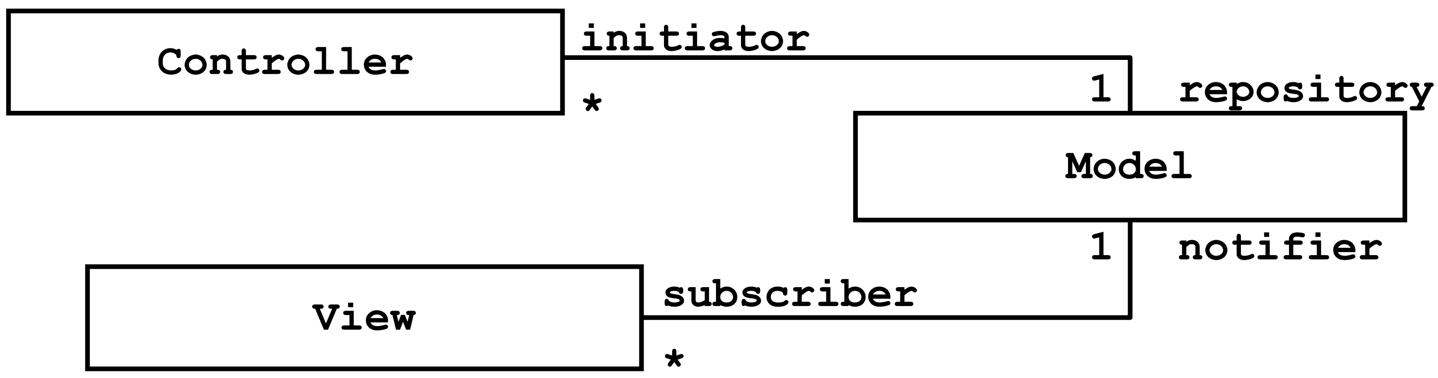




Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

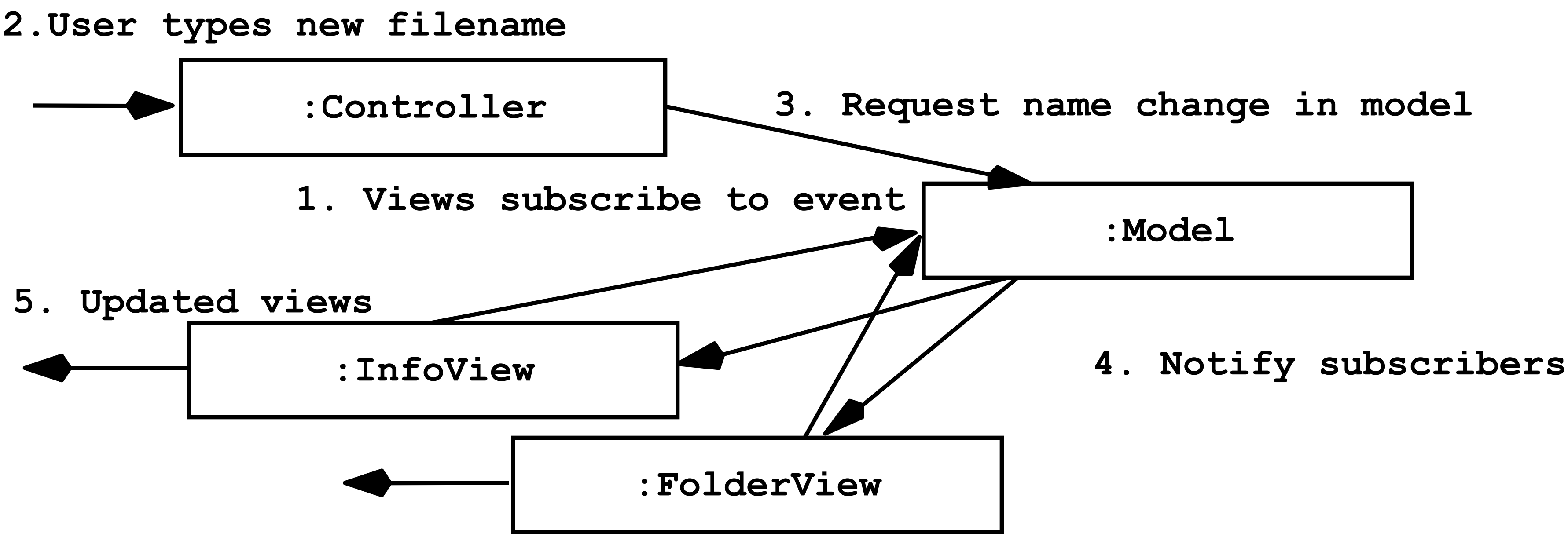
Model/View/Controller

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



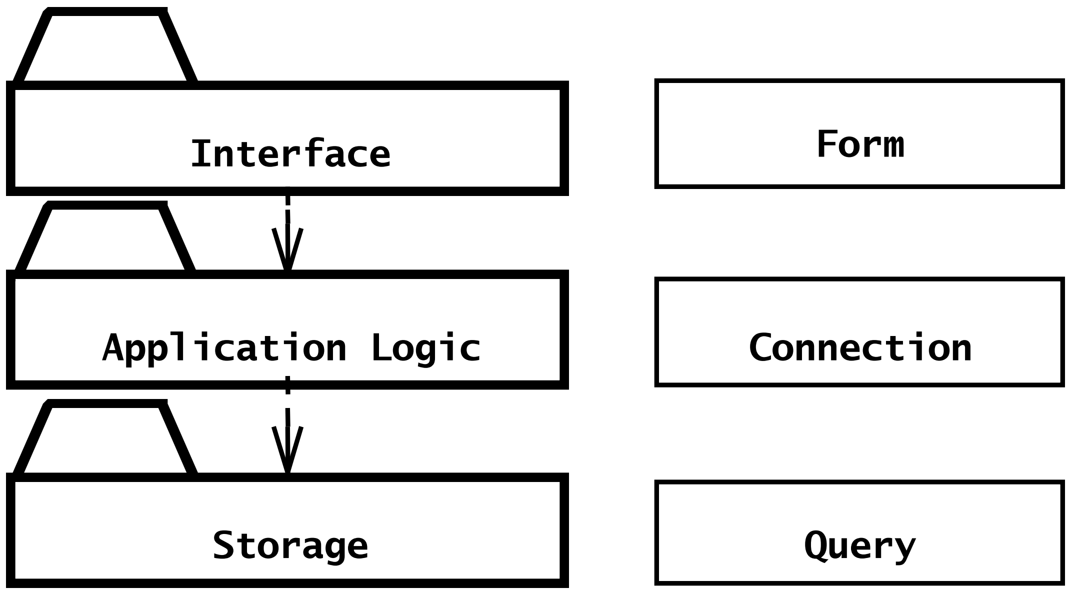
Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Sequence of Events (Collaborations)



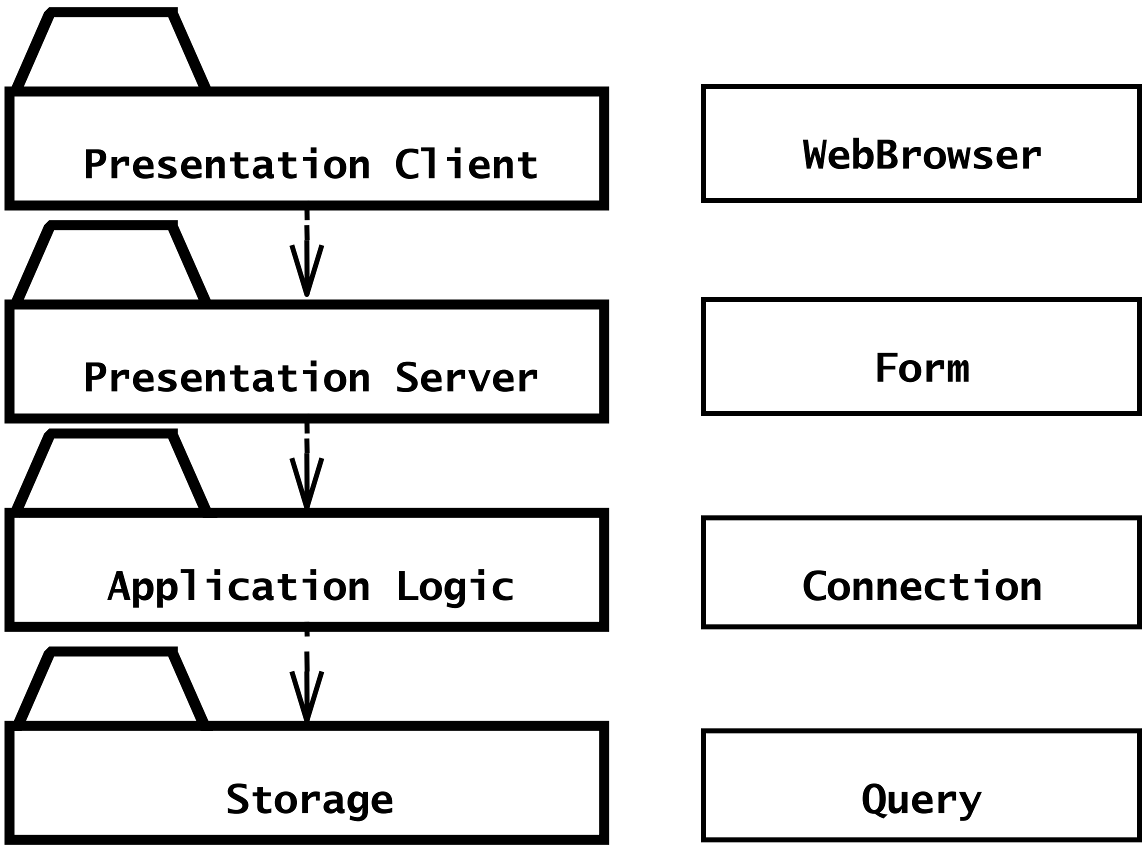
Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Three-tier architectural style.



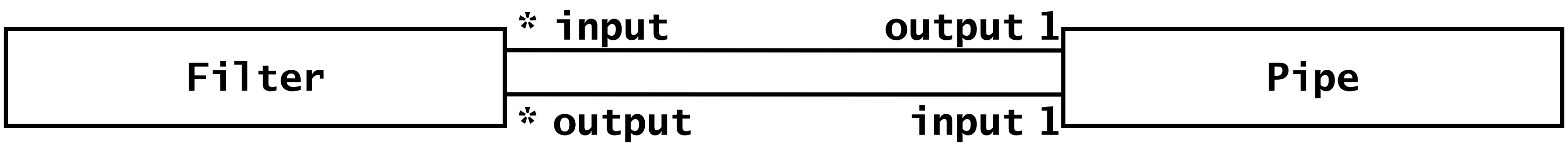
Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Four-tier architectural style.



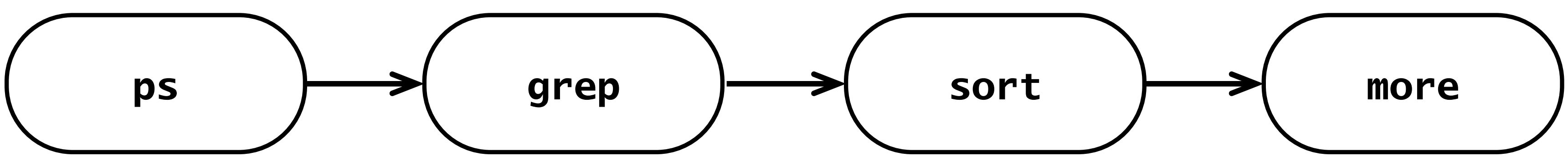
Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Pipe and filter architectural style.




```
% 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
```



Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Intermediate 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

System Design

1. **Design Goals**
   - Definition
   - Trade-offs
2. **System Decomposition**
   - Layers/Partitions
   - Cohesion/Coupling
3. **Concurrency**
   - Identification of Threads of control
4. **Hardware/ Software Mapping**
   - Special purpose
   - Buy or Build Trade-off
   - Allocation
   - Connectivity
5. **Data Management**
   - Persistent Objects
   - 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

Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

System Design Activities (continue)

- – Hardware/Software Mapping
- – Persistent Data Management
- – Global Resource Handling and Access Control
- – Software Control
- – Boundary Conditions

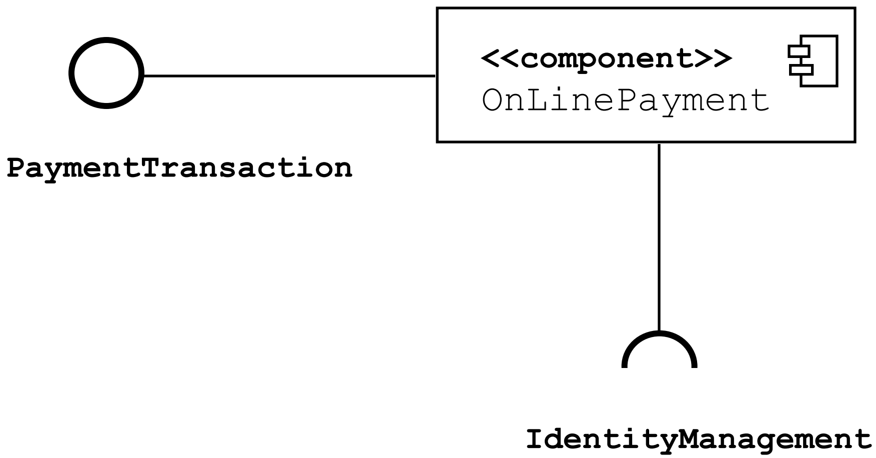
Hardware/Software Mapping

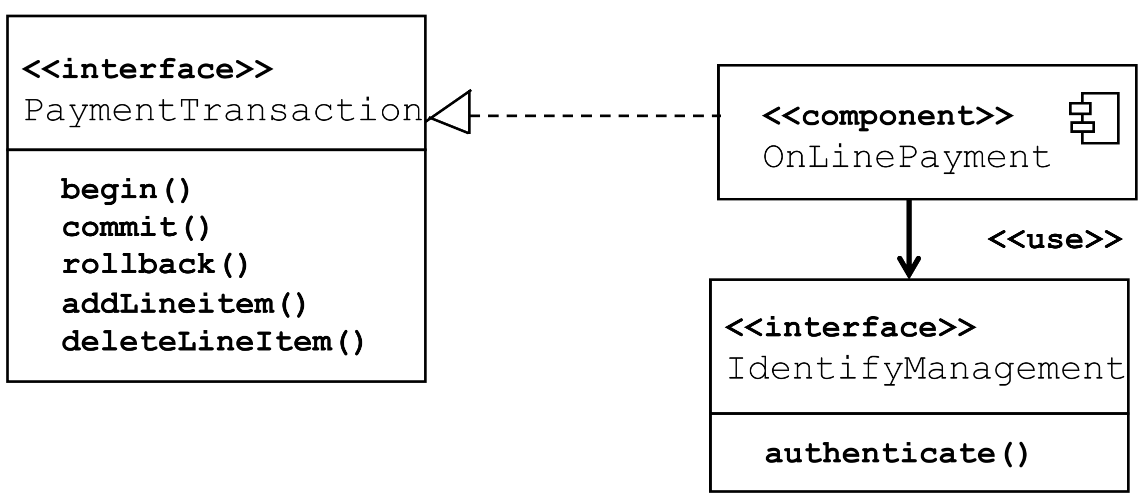
- This activity addresses two questions:
  - How shall we realize the subsystems: Hardware or Software?
  - How is the object model mapped on the chosen hardware & software?
    - Mapping Objects onto Reality: Processor, Memory, Input/Output
    - Mapping Associations onto Reality: Connectivity
- Much of the difficulty of designing a system comes from meeting externally-imposed hardware and software constraints.
  - Certain tasks have to be at specific locations

UML: Component and Deployment diagrams

- Models for physical implementation of the system
- Show system components, their structure and dependencies and how they are deployed on computer nodes
- Two kinds of diagrams:
  - component diagrams
  - deployment diagrams
- Component diagrams show structure of components, including their interface and implementation dependencies
- Deployment diagrams show the runtime deployment of the system on computer nodes

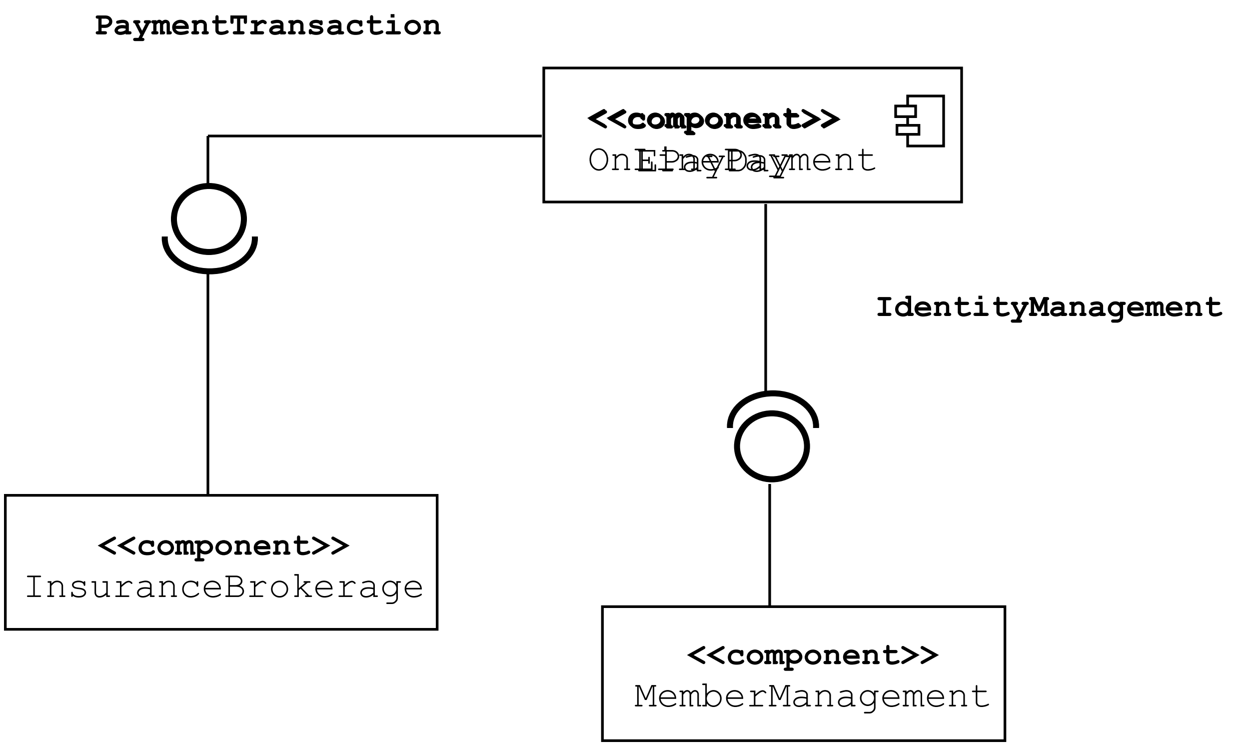
Component Diagrams (component interface)





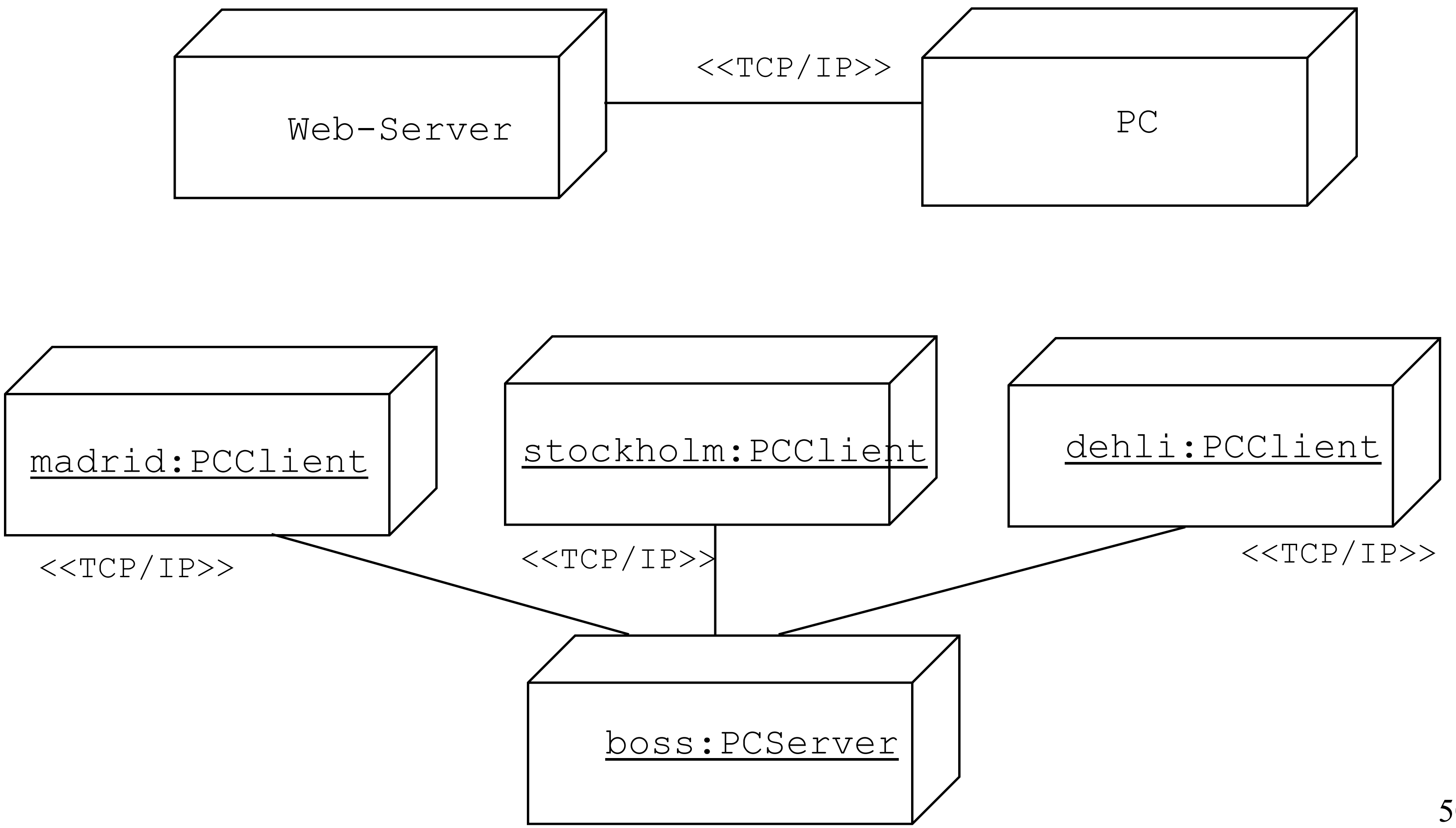
Adopted from Shaum

Component Diagram



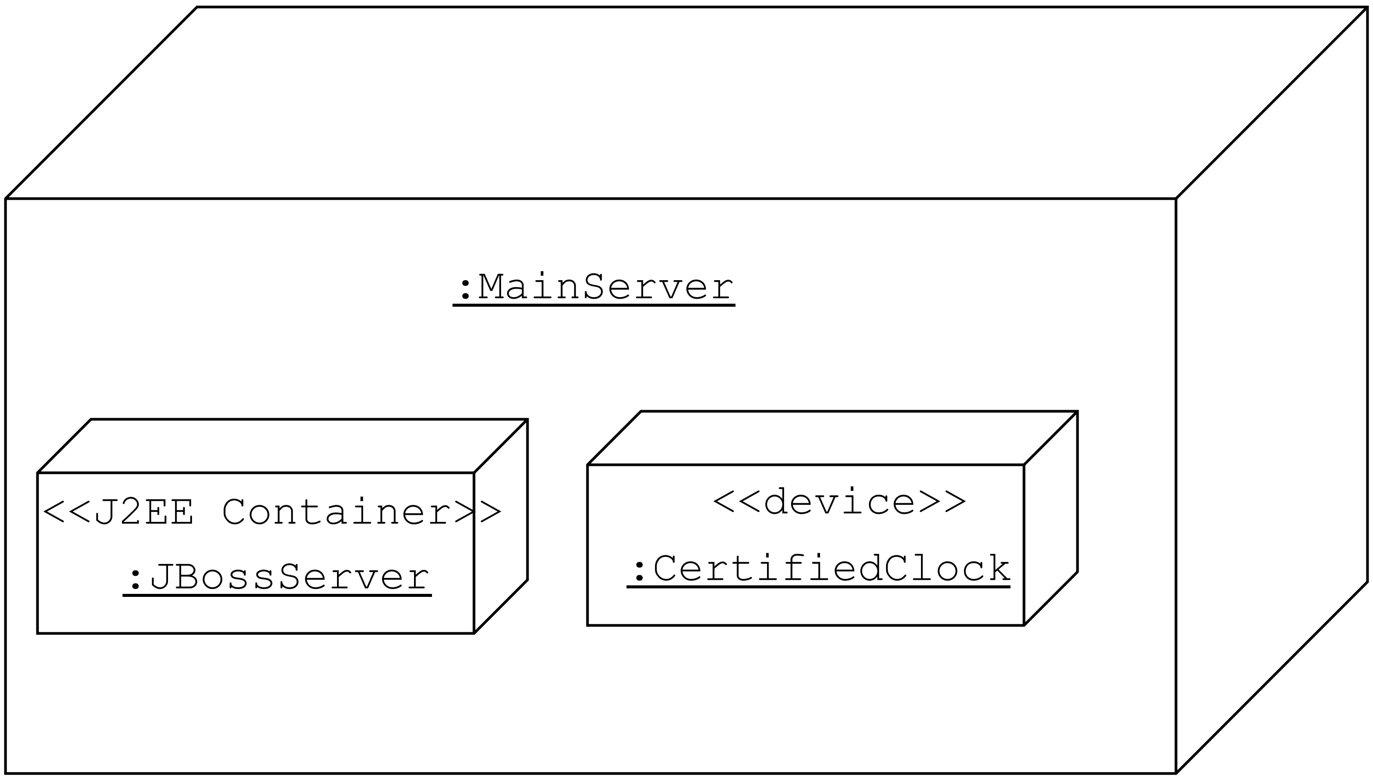
Adopted from

Deployment diagrams



Adopted from Shaum

Deployment diagrams



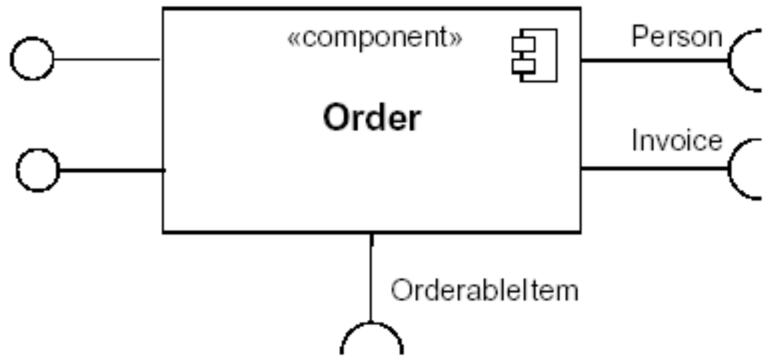
Adopted from Shaum

Drawing Hardware/Software Mappings in UML

- System design must model static and dynamic structures:
  - Component Diagrams for static structures
    - show the structure at design time or compilation time
  - Deployment Diagram for dynamic structures
    - show the structure of the run-time system

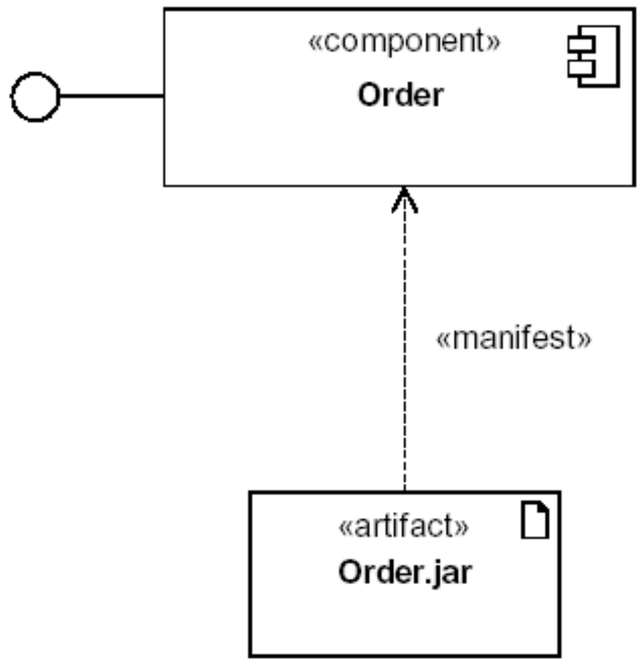
Deployment Diagrams (artifact)

- Component

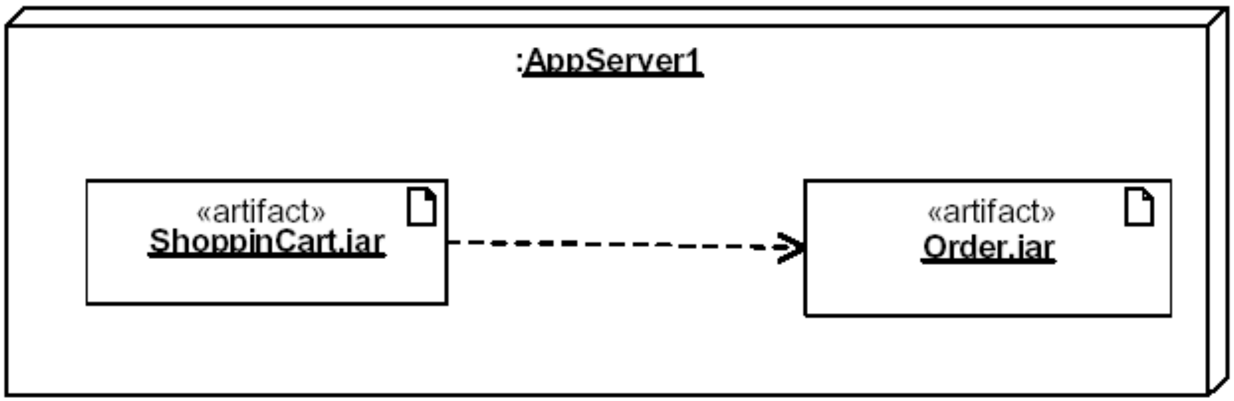


- Artifact - Artifacts represent concrete elements in the physical world that are the result of a development process.

Manifestation relationship between artifacts and components



Location of Artifacts on the node



Data Management

- Some objects in the models need to be persistent

- A persistent object can be realized with one of the following
  - Files
    - Cheap, simple, permanent storage
    - Low level (Read, Write)
    - Applications must add code to provide suitable level of abstraction
  - Database
    - Powerful, easy to port
    - Supports multiple writers and readers

File or DB?

When should you choose flat files?
- Voluminous data (e.g., images)
- Temporary data (e.g., core file)
- Low information density (e.g., archival files, history logs)

When should you choose a relational or an object-oriented database?
- Concurrent accesses
- Access at finer levels of detail
- Multiple platforms or applications for the same data

When should you choose a relational database?
- Complex queries over attributes
- Large data set

When should you choose an object-oriented database?
- Extensive use of associations to retrieve data
- Medium-sized data set
- Irregular associations among objects

Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Global Resource Handling

- Discusses access control
- Describes access rights for different classes of actors
- Describes how object guard against unauthorized access

Access Matrix Implementations

- Global access table: Represents explicitly every cell in the matrix as a (actor,class, operation) tuple.
  - Determining if an actor has access to a specific object requires looking up the corresponding tuple. If no such tuple is found, access is denied.
- Access control list associates a list of (actor, operation) pairs with each class to be accessed.
  - Every time an object is accessed, its access list is checked for the corresponding actor and operation.
  - Example: guest list for a party.
- A capability associates a (class,operation) pair with an actor.
  - A capability provides an actor to gain control access to an object of the class described in the capability.
  - Example: An invitation card for a party.
- Which is the right implementation?

Decide on Software Control

Choose implicit control (non-procedural, declarative languages)
  - Rule-based systems
  - Logic programming

Choose explicit control (procedural languages):

- Procedure-driven control
  - Control resides within program code. Operations wait for input whenever they need data from an actor. Example: Main program calling procedures of subsystems.
  - Simple, easy to build, hard to maintain (high recompilation costs)
- Event-driven control
  - Control resides within a dispatcher calling functions via callbacks.
  - Very flexible, good for the design of graphical user interfaces, easy to extend

Centralized vs. Decentralized Designs

- Should you use a centralized or decentralized design?
  - Take the sequence diagrams and control objects from the analysis model
  - Check the participation of the control objects in the sequence diagrams
    - If sequence diagram looks more like a fork: Centralized design
    - The sequence diagram looks more like a stair: Decentralized design
- Centralized Design
  - One control object or subsystem ("spider") controls everything
    - Pro: Change in the control structure is very easy
    - Cons: The single control object is a possible performance bottleneck
- Decentralized Design
  - Not a single object is in control, control is distributed, that means, there is more than one control object
    - Cons: The responsibility is spread out – changes are not obvious
    - Pro: Fits nicely into object-oriented development

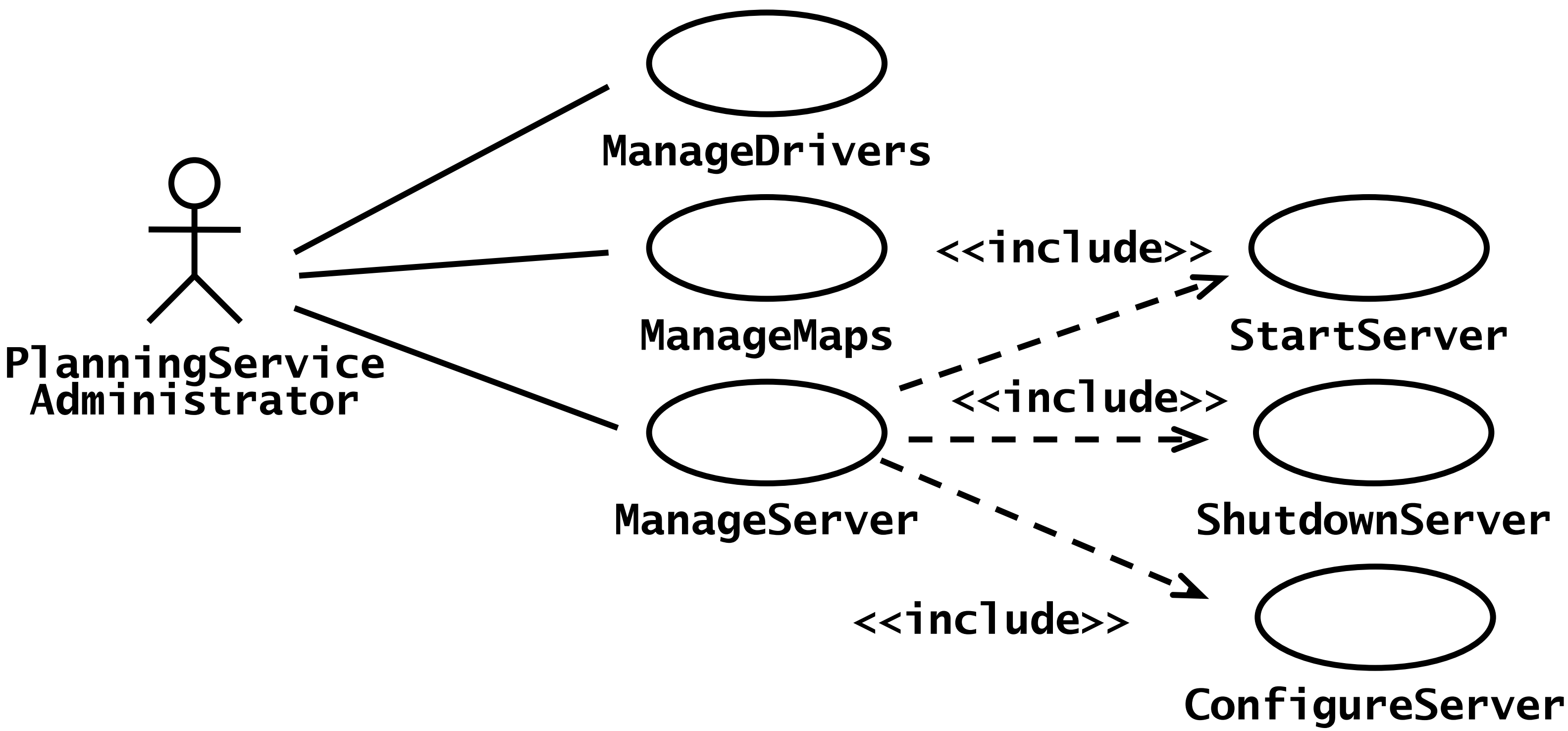
Boundary Condition Questions

- Initialization
  - How does the system start up?
    - What data need to be accessed at startup time?
    - What services have to be registered?
  - What does the user interface do at start up time?
    - How does it present itself to the user?
- Termination
  - Are single subsystems allowed to terminate?
  - Are other subsystems notified if a single subsystem terminates?
  - How are local updates communicated to the database?
- Failure
  - How does the system behave when a node or communication link fails? Are there backup communication links?
  - How does the system recover from failure? Is this different from initialization?

Modeling Boundary Conditions

- Boundary conditions are best modeled as use cases with actors and objects.
- Actor: often the system administrator
- Interesting use cases:
  - Start up of a subsystem
  - Start up of the full system
  - Termination of a subsystem
  - Error in a subystem or component, failure of a subsystem or component

ManageServer Use Case



Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Summary

In this lecture, we reviewed the activities of system design :

- Design goals identification
- System decomposition
- Concurrency identification
- Hardware/Software mapping
- Persistent data management
- Global resource handling
- Software control selection
- Boundary conditions

Each of these activities revises the subsystem decomposition to address a specific issue. Once these activities are completed, the interface of the subsystems can be defined.

Next lecture

- Object design

Chapters 8 & 9