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

Chapter 7, System Design: Addressing Design Goals

Overview ▷

System Design I

- ✓ 0. Overview of System Design
- ✓ 1. Design Goals
- ✓ 2. Subsystem Decomposition
  - ✓ Architectural Styles

System Design II

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

✓1. Design Goals
- Definition
- Trade-offs

✓2. Subsystem Decomposition
- Layers vs Partitions
- Coherence/Coupling

3. Concurrency
- Identification of Threads

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

5. Data Management
- Persistent Objects
- File system vs Database

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

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

8. Boundary Conditions
- Initialization
- Termination
- Failure

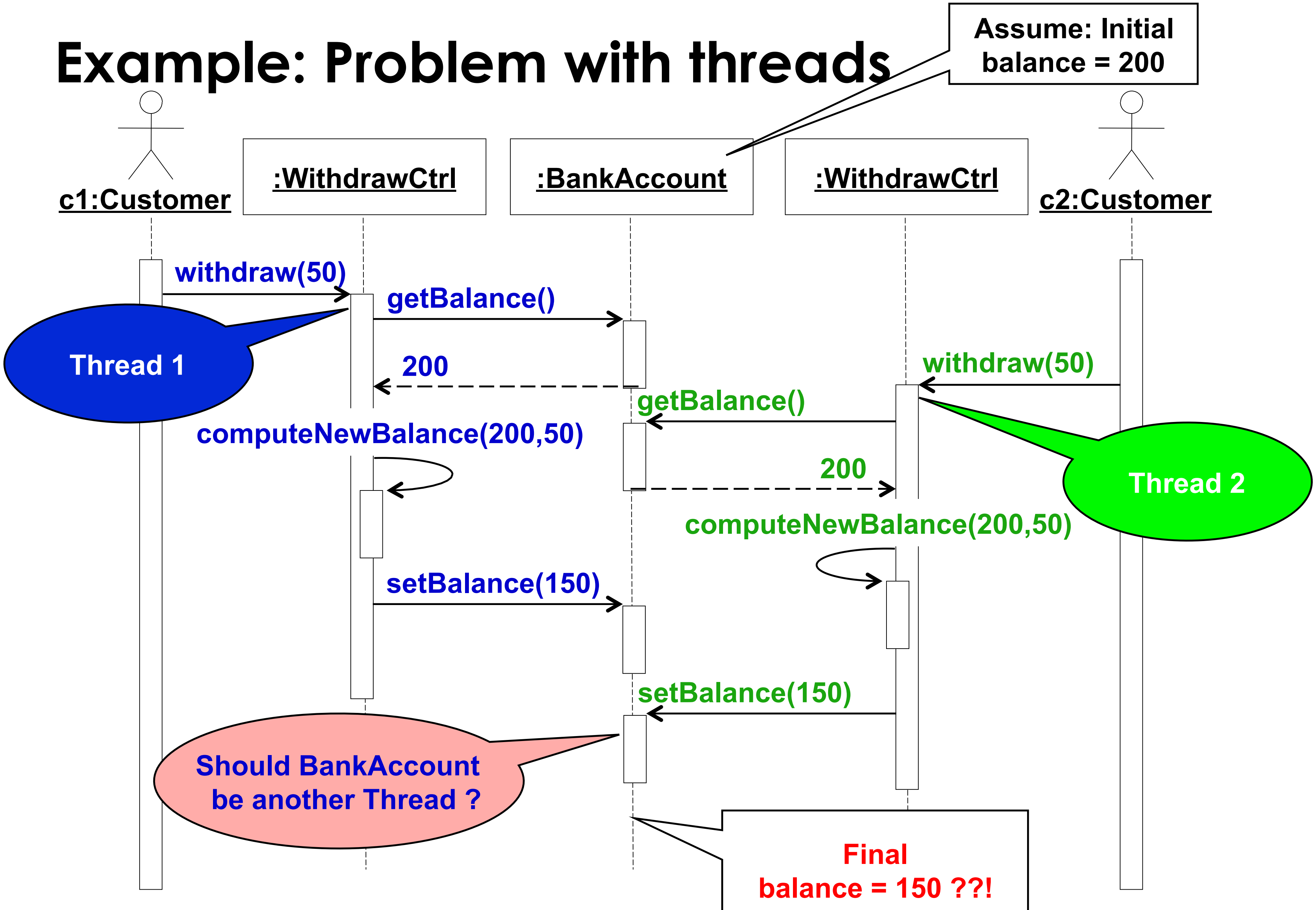
Concurrency

- Nonfunctional Requirements to be addressed: Performance, Response time, latency, availability.
- Two objects are inherently concurrent if they can receive events at the same time without interacting
    - Source for identification: Objects in a sequence diagram that can simultaneously receive events
        - Unrelated events, instances of the same event
- Inherently concurrent objects can be assigned to different **threads of control**
- Objects with mutual exclusive activity could be folded into a single **thread of control**

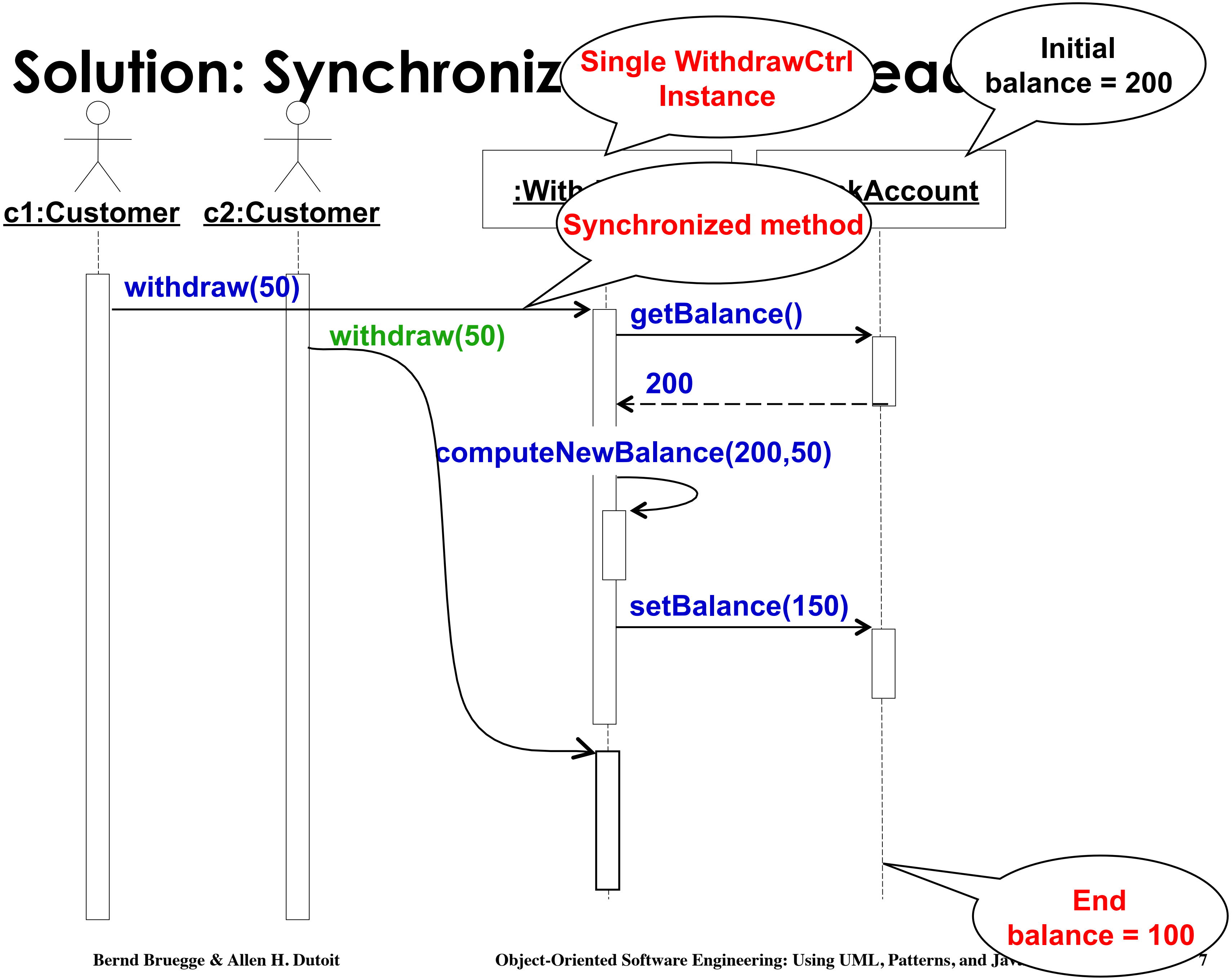
Thread of Control

- A thread of control is a path through a set of state diagrams on which a single object (different from time to time) is active at a time
  - A thread remains within a state diagram until an object sends an event to different object and waits for another event
  - Thread splitting: Object does a non-blocking send of an event to another object.
- Concurrent threads can lead to race conditions.
- A race condition (also race hazard) is a design flaw where the output of a process depends on the specific sequence of other events.
  - The name originated in digital circuit design: Two signals racing each other to influence the output.

Example: Problem with threads



Solution: Synchroniz[…]eac[…]



Concurrency Questions

- To identify threads for concurrency we ask the following questions:
  - Does the system provide access to multiple users?
  - Which entity objects of the object model can be executed independently from each other?
  - What kinds of control objects are identifiable?
  - Can a single request to the system be decomposed into multiple requests? Can these requests be handled in parallel? (Example: a distributed query)

Implementing Concurrency

- Concurrent systems can be implemented on any system that provides
  - Physical concurrency: Threads are provided by hardware

  or

  - Logical concurrency: Threads are provided by software
- Physical concurrency is provided by multiprocessors and computer networks
- Logical concurrency is provided by threads packages.

Implementing Concurrency (2)

- In both cases, - physical concurrency as well as logical concurrency - we have to solve the scheduling of these threads:
  - Which thread runs when?
- Today's operating systems provide a variety of scheduling mechanisms:
  - Round robin, time slicing, collaborating processes, interrupt handling
- General question addresses starvation, deadlocks, fairness -> Topic for researchers in operating systems
- Sometimes we have to solve the scheduling problem ourselves
  - Topic addressed by software control (system design topic 7).

System Design

✓1. Design Goals
- Definition
- Trade-offs

✓2. Subsystem Decomposition
- Layers vs Partitions
- Coherence/Coupling

✓3. Concurrency
- Identification of Threads

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

5. Data Management
- Persistent Objects
- Filesystem vs Database

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

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

8. Boundary Conditions
- Initialization
- Termination
- Failure

4. Hardware Software Mapping

- This system design activity addresses two questions:
  - How shall we realize the subsystems: With hardware or with software?
    - If hardware is chosen, how to proceed is out of the scope of the current course
  - How do we map the object model onto the chosen hardware and/or software?
    - Mapping the Objects:
      - Processor, Memory, Input/Output
    - Mapping the Associations:
      - Network connections

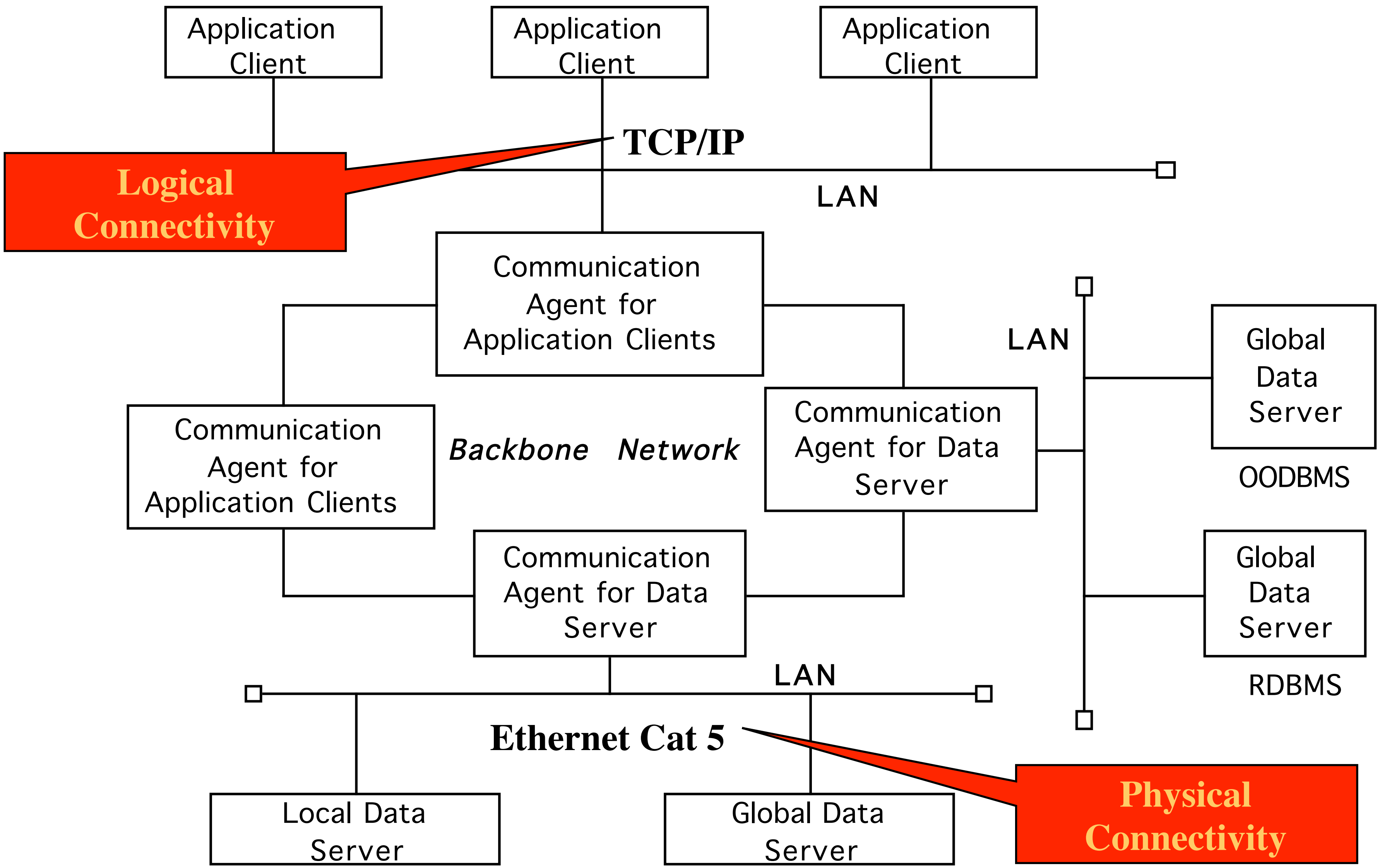
Mapping Objects onto Hardware

- Control Objects -> Processor
  - Is the computation rate too demanding for a single processor?
  - Can we get a speedup by distributing objects across several processors?
  - How many processors are required to maintain a steady state load?
- Entity Objects -> Memory
  - Is there enough memory to buffer bursts of requests?
- Boundary Objects -> Input/Output Devices
  - Do we need an extra piece of hardware to handle the data generation rates?
  - Can the desired response time be realized with the available communication bandwidth between subsystems?

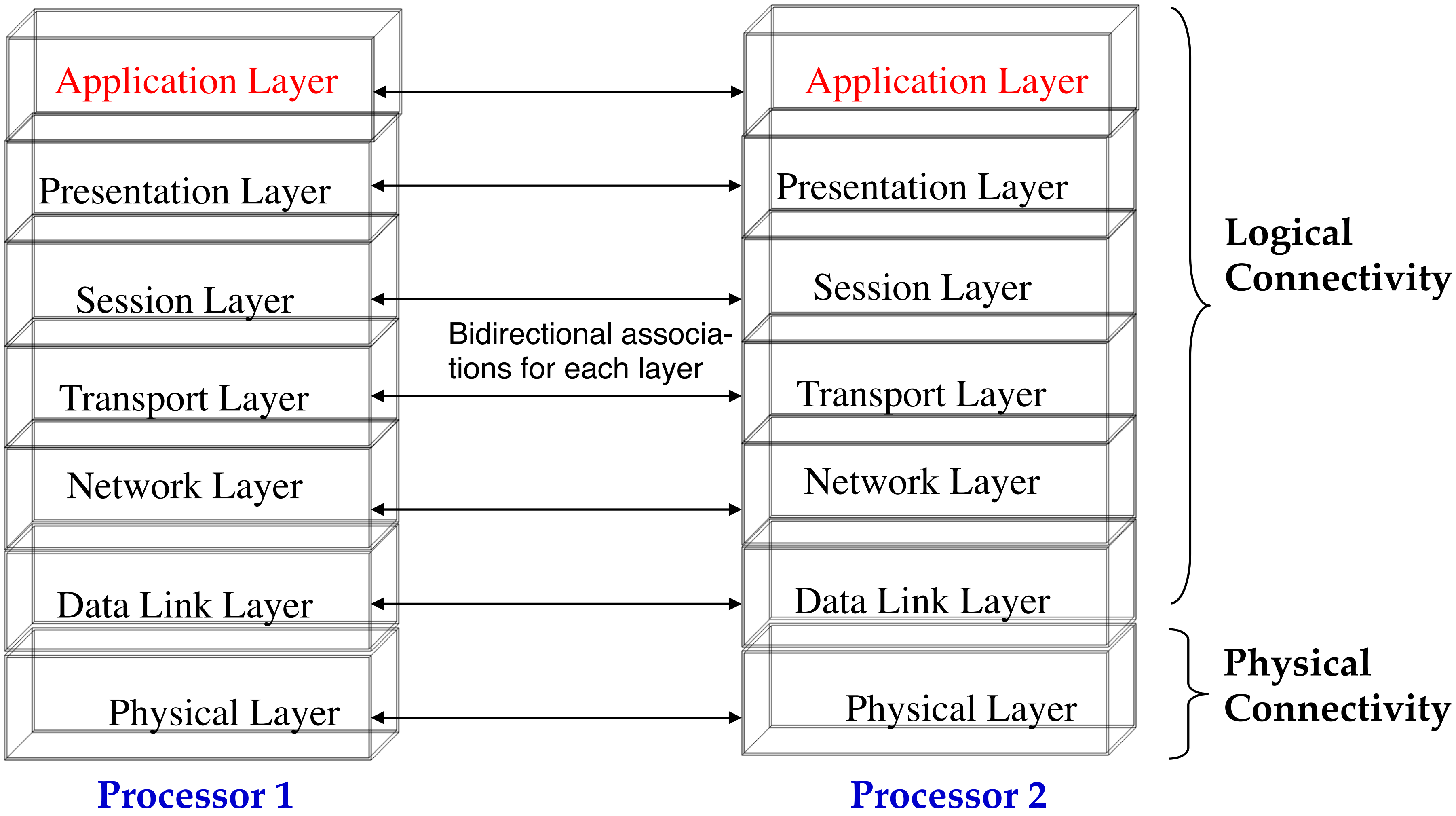
Mapping the Associations: Connectivity

- Describe the physical connectivity
  - ("Physical layer in the OSI reference model")
    - Describes **which associations** in the object model **are mapped to physical connections**
- Describe the logical connectivity (subsystem associations)
  - Associations that do not directly map into physical connections
  - In which layer should these associations be implemented?
- Informal connectivity drawings often contain both types of connectivity
  - Practiced by many developers, sometimes confusing.

Example: Informal Connectivity Drawing



Logical vs Physical Connectivity and the relationship to Subsystem Layering



Hardware-Software Mapping Difficulties

- Much of the difficulty of designing a system comes from addressing externally-imposed hardware and software constraints
  - Certain tasks have to be at specific locations
    - Example: Withdrawing money from an ATM machine
  - Some hardware components have to be used from a specific manufacturer

Hardware/Software Mappings in UML

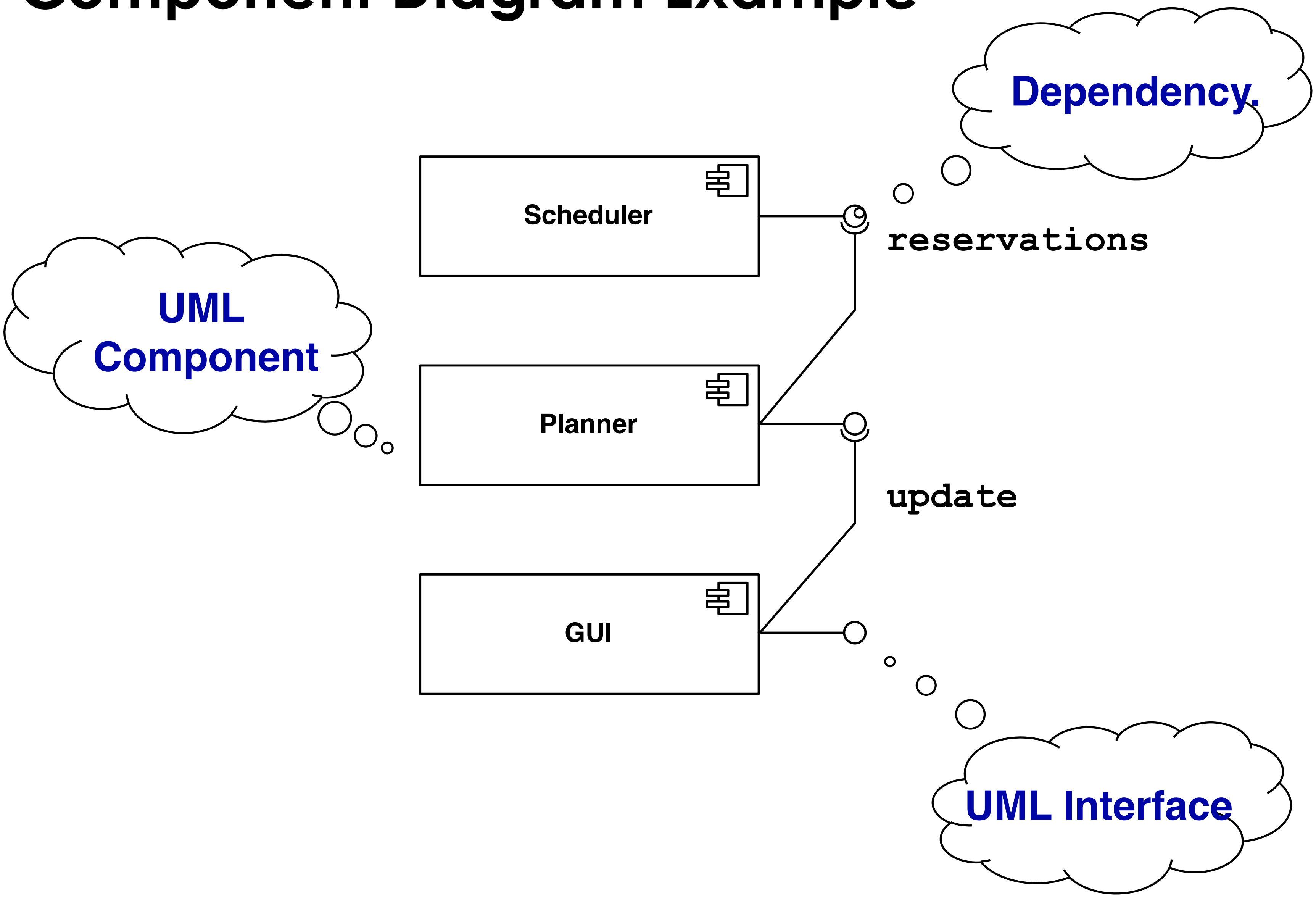
- A UML component is a building block of the system. It is represented as a rectangle with a tabbed rectangle symbol inside

- The Hardware/Software Mapping addresses dependencies and distribution issues of UML components during system design.

Two New UML Diagram Types

- Deployment Diagram:
  - Illustrates the distribution of components at run-time.
  - Deployment diagrams use nodes and connections to depict the physical resources in the system.
- Component Diagram:
  - Illustrates dependencies between components at design time, compilation time and runtime

Component Diagram Example

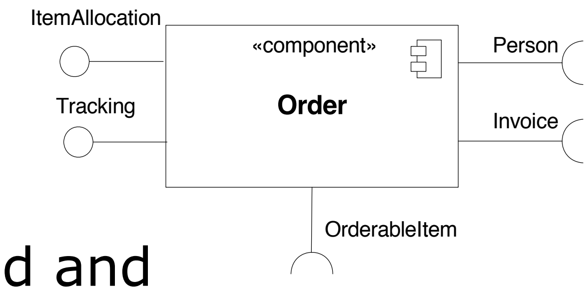


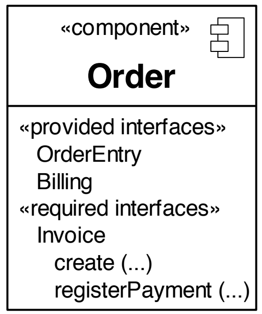
UML Component Diagram

- Used to model the top-level view of the system design in terms of components and dependencies among the components. Components can be
  - source code, linkable libraries, executables
- The dependencies (edges in the graph) are shown as dashed lines with arrows from the client component to the supplier component:
  - The lines are often also called connectors
  - The types of dependencies are implementation language specific

UML Interfaces: Lollipops and Sockets

- A UML interface describes a group of operations used or created by UML components.
  - There are two types of interfaces: provided and required interfaces.
    - A provided interface is modeled using the lollipop notation
    - A required interface is modeled using the socket notation.
-





Component diagram – details from UML 2.4.1

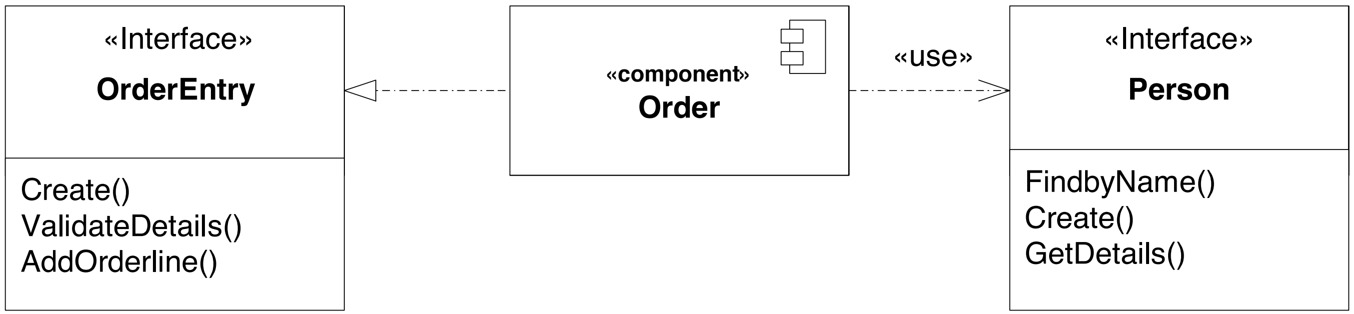
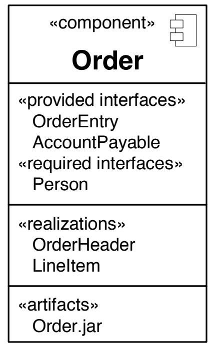


Figure 8.8 - Explicit representation of the provided and required interfaces, allowing interface details such as operation to be displayed (when desired).



A white-box representation of a component

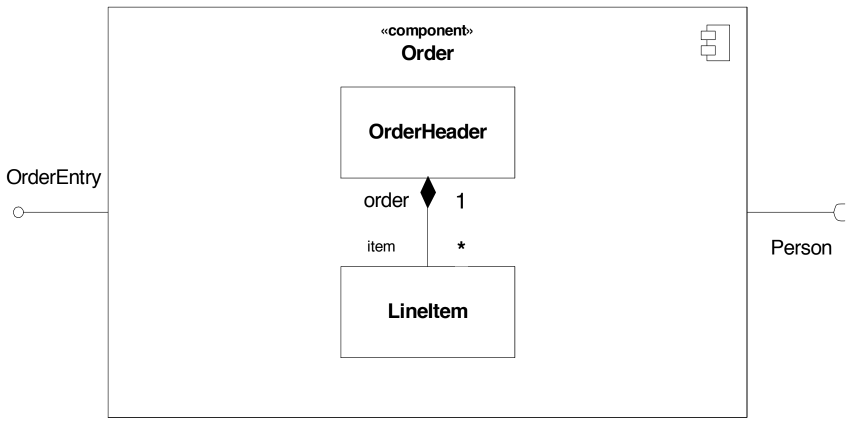


Figure 8.11 - An alternative nested representation of a complex component

Component diagram – details from UML 2.4.1 / 2

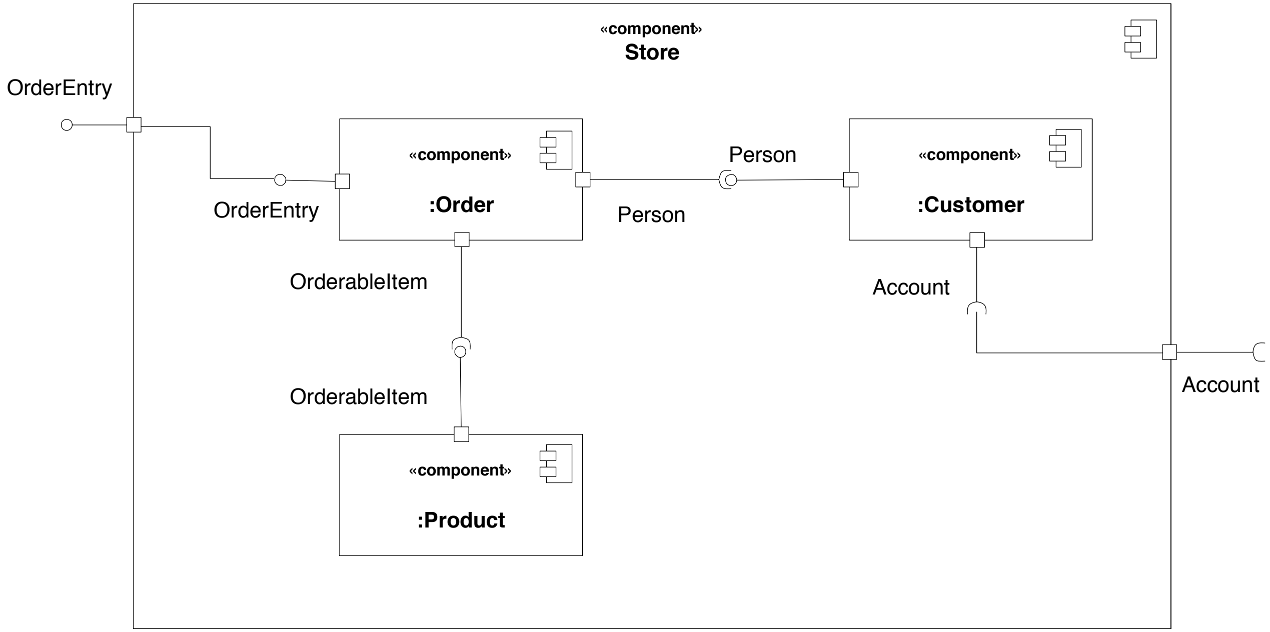


Figure 8.12 - An internal or white-box view of the internal structure of a component that contains other components with simple ports as parts of its internal assembly

- A port specifies a distinct interaction point between the component and its environment.
  - Ports are depicted as small squares on the sides of classifiers.
  - The interfaces associated with a port specify the nature of the interactions that may occur over a port.

Component diagram – details from UML 2.4.1 / 3

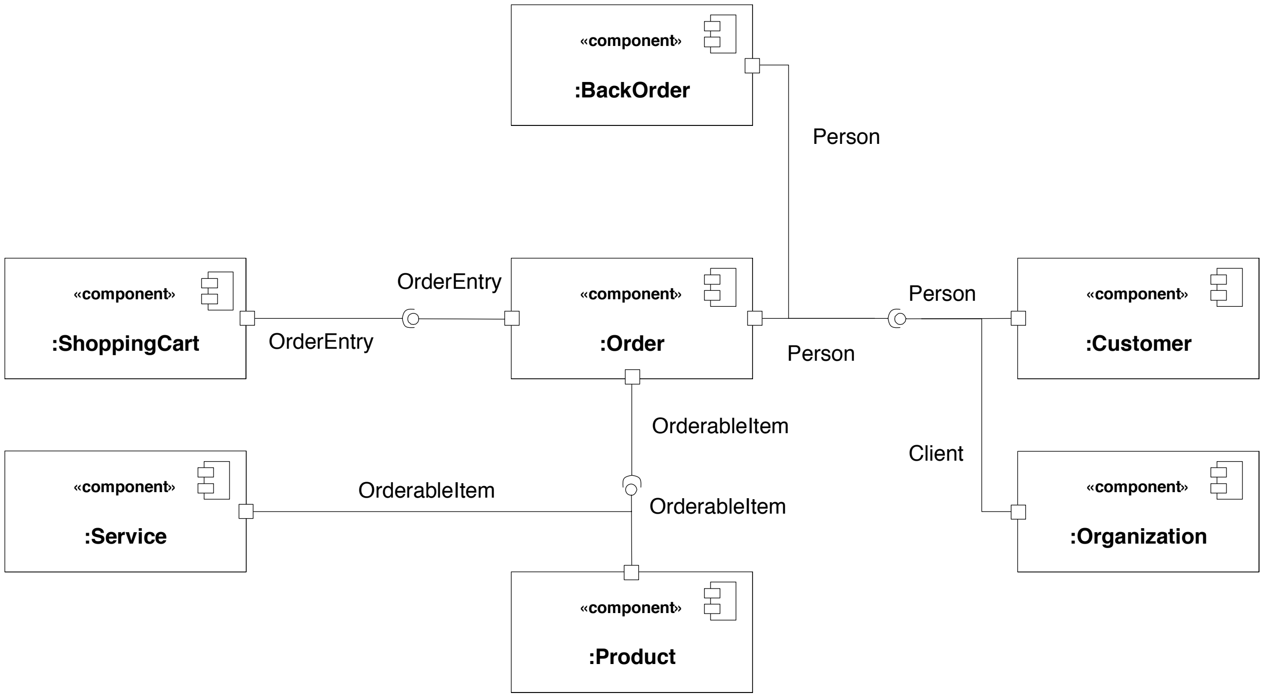
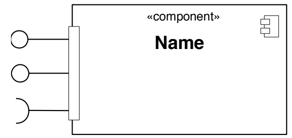


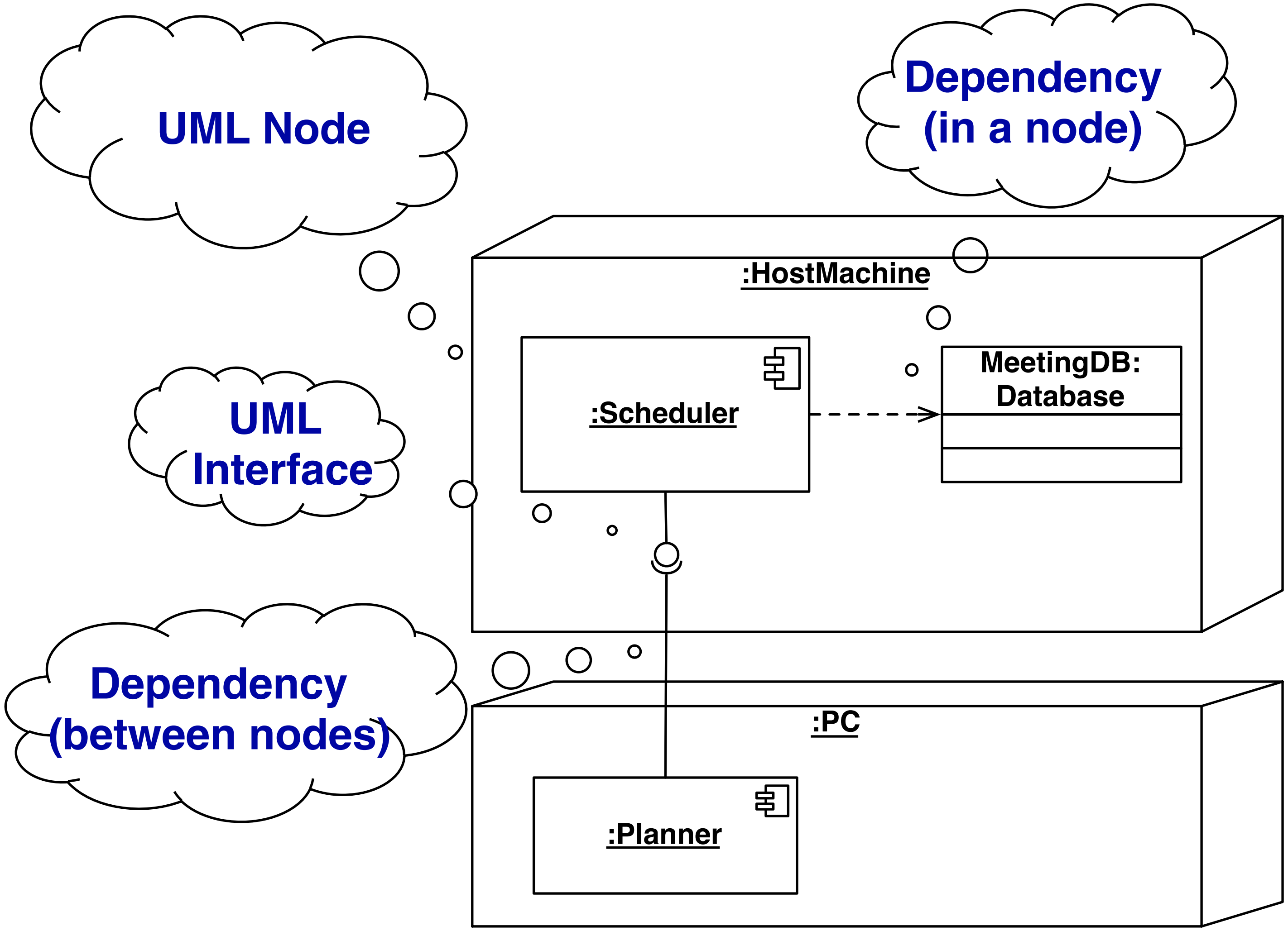
Figure 8.15 -Example of a composite structure of components, with connector wiring between simple ports on parts (Note: "Client" interface is a subtype of "Person").



Component with a complex port

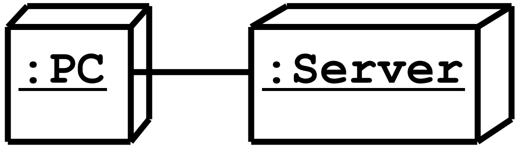
Where multiple components have simple ports that provide or require the same interface, a single symbol representing the interface can be shown, and lines from the components can be drawn to that symbol

Deployment Diagram Example



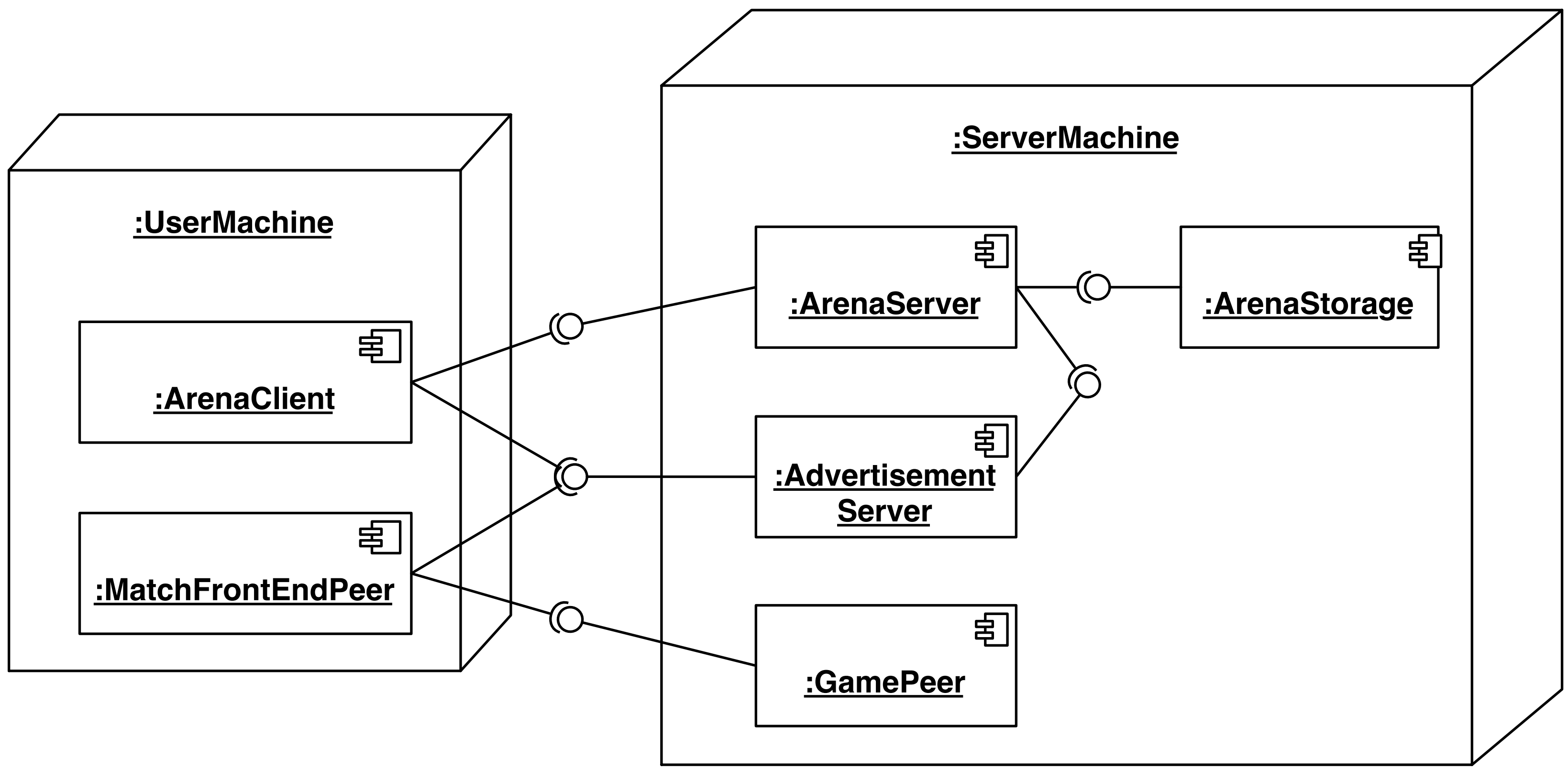
Deployment Diagram

- Deployment diagrams are useful for showing a system design after these system design decisions have been made:
  - Subsystem decomposition
  - Concurrency
  - Hardware/Software Mapping



- A deployment diagram is a graph of nodes and connections ("communication associations")
  - Nodes are shown as 3-D boxes
  - Connections between nodes are shown as solid lines
  - Nodes may contain components
    - Components can be connected by "lollipops" and "grabbers"
    - Components may contain objects (indicating that the object is part of the component).

ARENA Deployment Diagram



System Design

✓1. Design Goals
Definition
Trade-offs

✓2. Subsystem Decomposition
Layers vs Partitions
Coherence/Coupling

✓3. Concurrency
Identification of Threads

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

5. Data Management
Persistent Objects
Filesystem vs Database

6. Global Resource Handling
Access Control List vs Capabilities
Security

7. Software Control
Monolithic
Event-Driven
Conc. Processes

8. Boundary Conditions
Initialization
Termination
Failure

5. Data Management

- Some objects in the system model need to be persistent:
  - Values for their attributes have a lifetime longer than a single execution
- A persistent object can be realized with one of the following mechanisms:
  - Filesystem:
    - If the data are used by multiple readers but a single writer
  - Database:
    - If the data are used by concurrent writers and readers.

Data Management Questions

- How often is the database accessed?
  - What is the expected request (query) rate? The worst case?
  - What is the size of typical and worst case requests?
- Do the data need to be archived?
- Should the data be distributed?
  - Does the system design try to hide the location of the databases (location transparency)?
- Is there a need for a single interface to access the data?
  - What is the query format?

Mapping Object Models

- UML object models can be mapped to relational databases
- The mapping:
  - Each class is mapped to its own table
  - Each class attribute is mapped to a column in the table
  - An instance of a class represents a row in the table
  - One-to-many associations are implemented with a buried foreign key
  - Many-to-many associations are mapped to their own tables
- Methods are not mapped

System Design

✓1. Design Goals
Definition
Trade-offs

✓2. Subsystem Decomposition
Layers vs Partitions
Coherence/Coupling

✓3. Concurrency
Identification of Threads

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

5. Data Management
Persistent Objects
Filesystem vs Database

6. Global Resource Handling
Access Control List vs Capabilities
Security

7. Software Control
Monolithic
Event-Driven
Conc. Processes

8. Boundary Conditions
Initialization
Termination
Failure

6. Global Resource Handling

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

Defining Access Control

- In multi-user systems different actors usually have different access rights to different functionality and data
- How do we model these accesses?
  - During analysis we model them by associating different use cases with different actors
  - During system design we model them determining which objects are shared among actors.

Access Matrix

- We model access on classes with an access matrix:
  - The rows of the matrix represents the actors of the system
  - The column represent classes whose access we want to control
- Access Right: An entry in the access matrix. It lists the operations that can be executed on instances of the class by the actor.

	Class 1	Class 2	Class 3
Actor 1	methodX() methodZ()	methodW()	
Actor 2	MethodY()		methodV()

Access Matrix Example

Classes

Access Rights

Actors

	Arena	League	Tournament	Match
Operator	<<create>> createUser() view ()	<<create>> archive()		
LeagueOwner	view ()	edit ()	<<create>> archive() schedule() view()	<<create>> end()
Player	view() applyForOwner()	view() subscribe()	applyFor() view()	play() forfeit()
Spectator	view() applyForPlayer()	view() subscribe()	view()	view() replay()

Access Matrix Implementations (1 of 2)

- Global access table: Represents explicitly every cell in the matrix as a triple (actor,class, operation)

LeagueOwner, **Arena**, view()
LeagueOwner, League, edit()
LeagueOwner, **Tournament**, <<create>>
LeagueOwner, **Tournament**, view()
LeagueOwner, **Tournament**, schedule()
LeagueOwner, **Tournament**, archive()
LeagueOwner, **Match**, <<create>>
LeagueOwner, **Match**, end()

.

Access Matrix Implementations (2 of 2)

- Access control list
  - Associates a list of (actor,operation) pairs with each class to be accessed.
  - Every time an instance of this class is accessed, the access list is checked for the corresponding actor and operation.
- 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.

Global Resource Questions

- Does the system need authentication?
- If yes, what is the authentication scheme?
  - User name and password? Access control list
  - Tickets? Capability-based
- What is the user interface for authentication?
- Does the system need a network-wide name server?
- How is a service known to the rest of the system?
  - At runtime? At compile time?
  - By Port?
  - By Name?

System Design

✓1. Design Goals
- Definition
- Trade-offs

✓2. Subsystem Decomposition
- Layers vs Partitions
- Coherence/Coupling

✓3. Concurrency
- Identification of Threads

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

5. Data Management
- Persistent Objects
- Filesystem vs Database

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

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

8. Boundary Conditions
- Initialization
- Termination
- Failure

7. Decide on Software Control

Two major design choices:

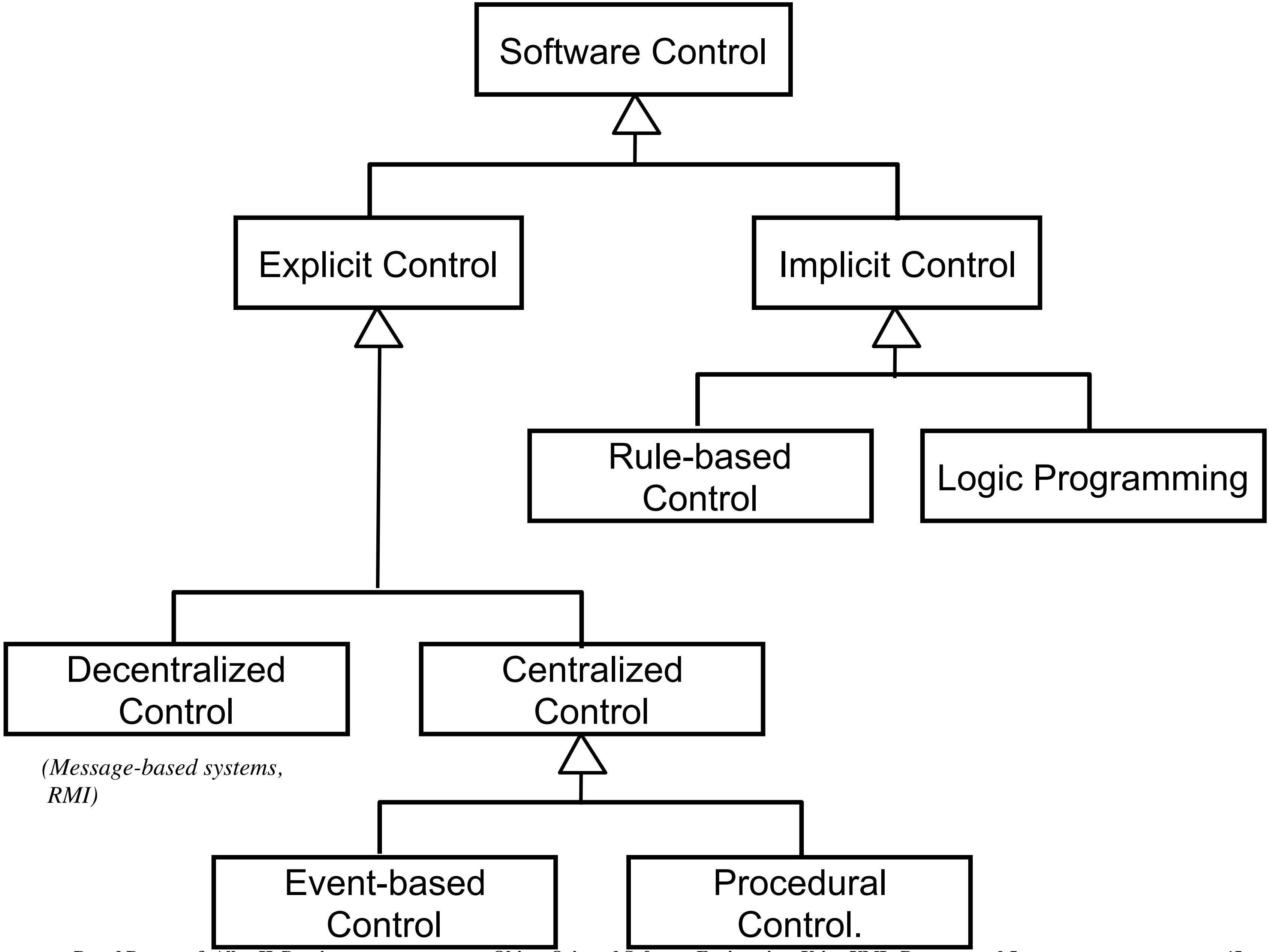
1. Choose implicit control
   - Rule-based systems, Logic programming
2. Choose explicit control
   - Procedural languages: Centralized or decentralized

Centralized vs. Decentralized Designs

- (Explicit) Centralized Design
  - One control object or subsystem ("spider") controls everything
    - Pro: Change in the control structure is very easy
    - Con: The single control object is a possible performance bottleneck
- (Explicit) Decentralized Design
  - Not a single object is in control, control is distributed; That means, there is more than one control object
    - Con: The responsibility is spread out
    - Pro: Fits nicely into object-oriented development

Centralized vs Decentralized Design/2

- (Explicit) Centralized control:
  - Procedure-driven: Control resides within program code.
  - Event-driven: Control resides within a dispatcher calling functions via callbacks.
- (Explicit) Decentralized control
  - Control resides in several independent objects.
    - Examples: Message based system, RMI
  - Possible speedup by mapping the objects on different processors, increased communication overhead.


Software Control
Explicit Control
Implicit Control
Rule-based Control
Logic Programming
Decentralized Control
Centralized Control
(Message-based systems, RMI)
Event-based Control
Procedural Control.

Centralized vs. Decentralized Designs (2)

- 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 the sequence diagram looks like a fork => Centralized design
  - If the sequence diagram looks like a stair => Decentralized design.

System Design

✓1. Design Goals
- Definition
- Trade-offs

✓2. Subsystem Decomposition
- Layers vs Partitions
- Coherence/Coupling

✓3. Concurrency
- Identification of Threads

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

5. Data Management
- Persistent Objects
- Filesystem vs Database

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

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

8. Boundary Conditions
- Initialization
- Termination
- Failure

8. Boundary Conditions

- Initialization
  - The system is brought from a non-initialized state to steady-state
- Termination
  - Resources are cleaned up and other systems are notified upon termination
- Failure
  - Possible failures: Bugs, errors, external problems

- Good system design foresees fatal failures and provides mechanisms to deal with them.

Boundary Condition Questions

- Initialization
  - 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?
- Termination
  - Are single subsystems allowed to terminate?
  - Are subsystems notified if a single subsystem terminates?
  - How are updates communicated to the database?
- Failure
  - How does the system behave when a node or communication link fails?
  - How does the system recover from failure?.

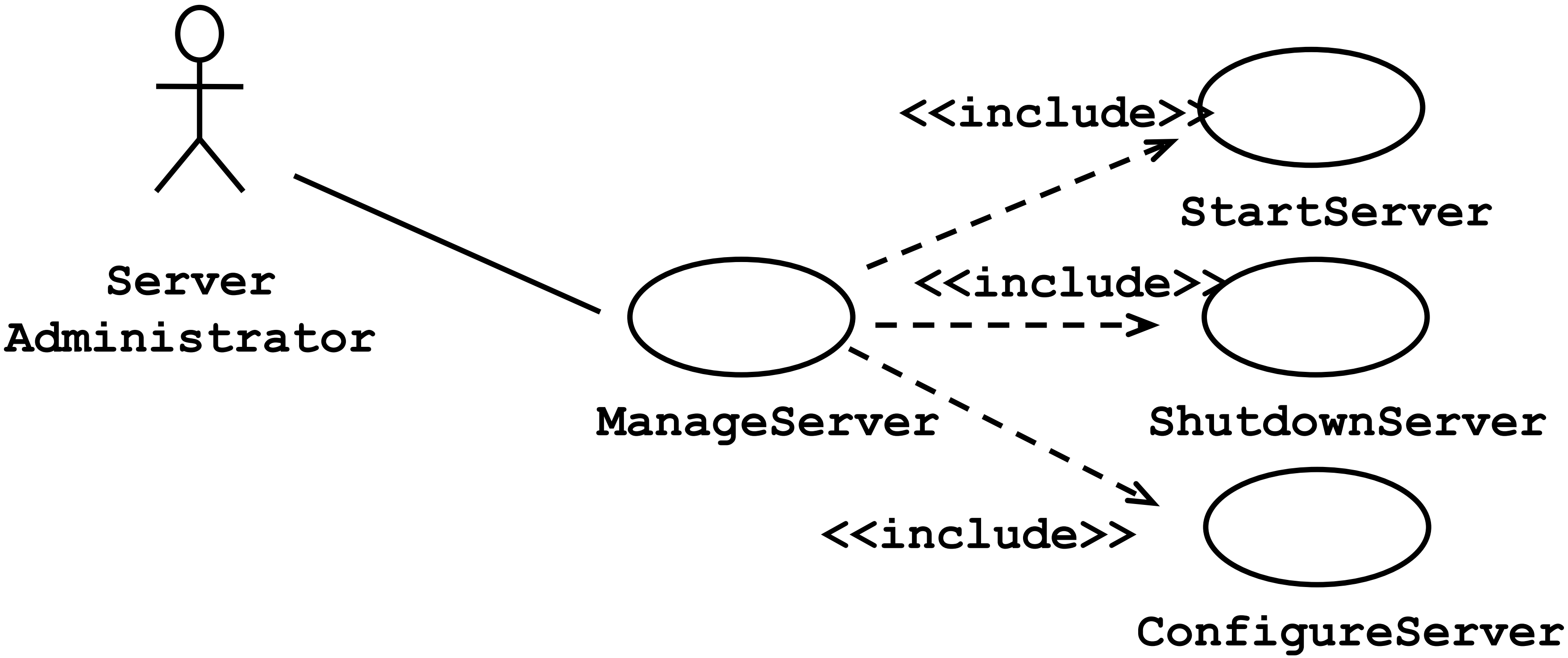
Modeling Boundary Conditions as use cases

- Boundary conditions are best modeled as use cases with actors and objects
- We call them boundary use cases or administrative use cases
- 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 subsystem or component, failure of a subsystem or component.

Example: Boundary Use Case for ARENA

- Let us assume, we identified the subsystem `AdvertisementServer` during system design
- This server takes a big load during the holiday season
- During hardware software mapping we decide to dedicate a special node for this server
- For this node we define a new boundary use case `ManageServer`
- `ManageServer` includes all the functions necessary to start up and shutdown the `AdvertisementServer`.

ManageServer Boundary Use Case



Summary

- System design activities:
  - Concurrency identification
  - Hardware/Software mapping
  - Persistent data management
  - Global resource handling
  - Software control selection
  - Boundary conditions
- Each of these activities may affect the subsystem decomposition
- Two new UML Notations
  - UML Component Diagram: Showing compile time and runtime dependencies between subsystems
  - UML Deployment Diagram: Drawing the runtime configuration of the system.