

Addressing Design Goals

Software Engineering I Lecture 9

Bernd Bruegge, Ph.D.
Applied Software Engineering
Technische Universitaet Muenchen

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
Filesystem vs Database

8. Boundary Conditions

Initialization
Termination
Failure

7. Software Control

Monolithic
Event-Driven
Conc. Processes

6. Global Resource Handlung

Access Control List
vs Capabilities
Security

Concurrency

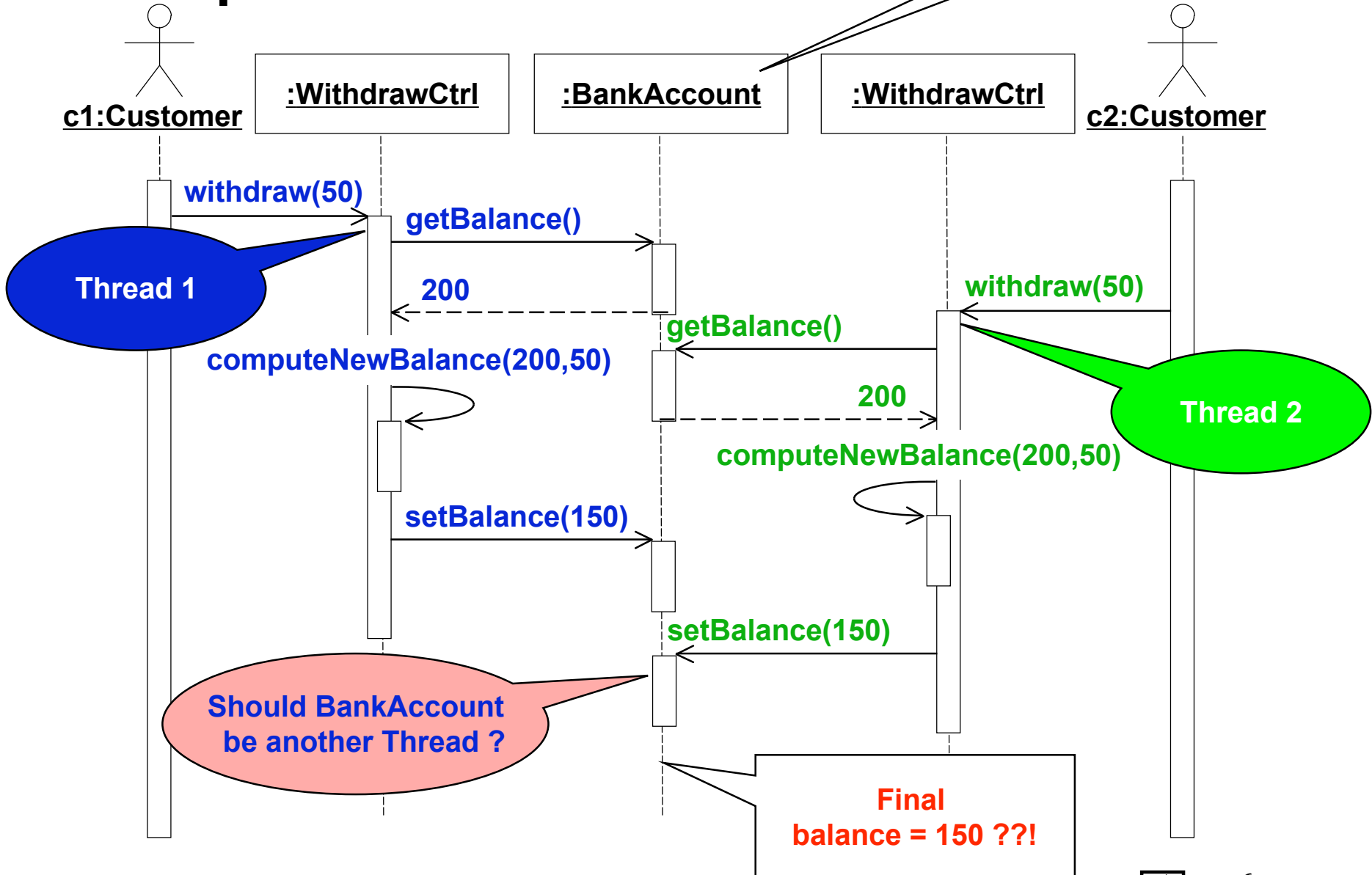
- Definition **Thread**
 - A *thread* of control is a path through a set of state diagrams on which a single object is active at a time
 - A thread remains within a state diagram until an object sends an event to another object and waits for another event
 - Thread splitting: Object does a nonblocking send of an event
- System Design Activities:
 - Identify concurrent threads and address design issues
- Design goals to be addressed: Performance (Response time, latency, availability).

Concurrency (continued)

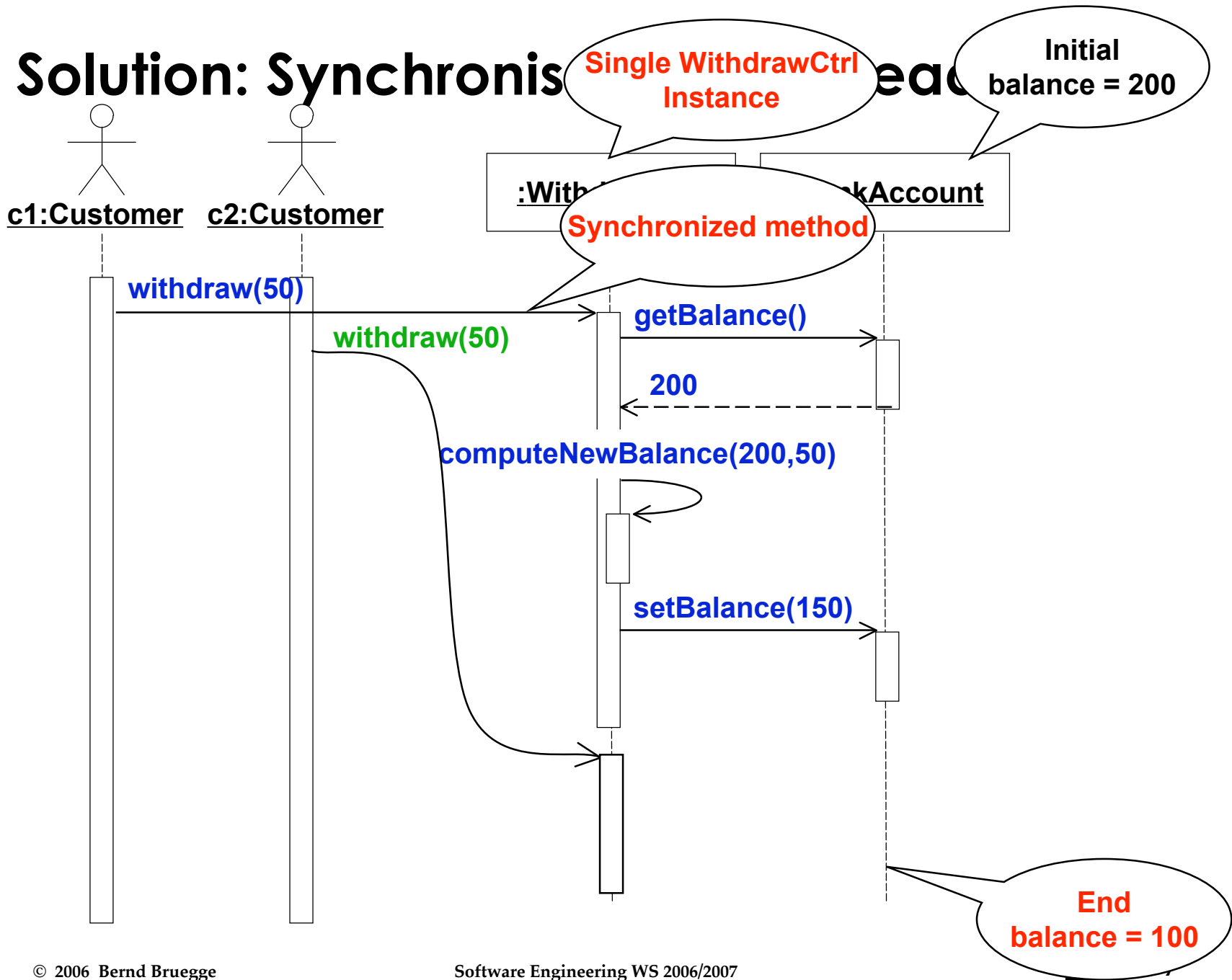
- Two objects are **inherently concurrent** if they can receive events at the same time without interacting
 - Source for identification: Objects in a sequence diagram that can simultaneously receive events
- Inherently concurrent objects can be assigned to different threads of control
- Objects with **mutual exclusive activity** could be folded into a single thread of control

Example: Problem with threads

Assume: Initial balance = 200



Solution: Synchronis



Concurrency Questions

- To identify candidates 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 and 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

8. Boundary Conditions

Initialization
Termination
Failure

7. Software Control

Monolithic
Event-Driven
Conc. Processes

6. Global Resource Handlung

Access Control List
vs Capabilities
Security

4. Hardware Software Mapping

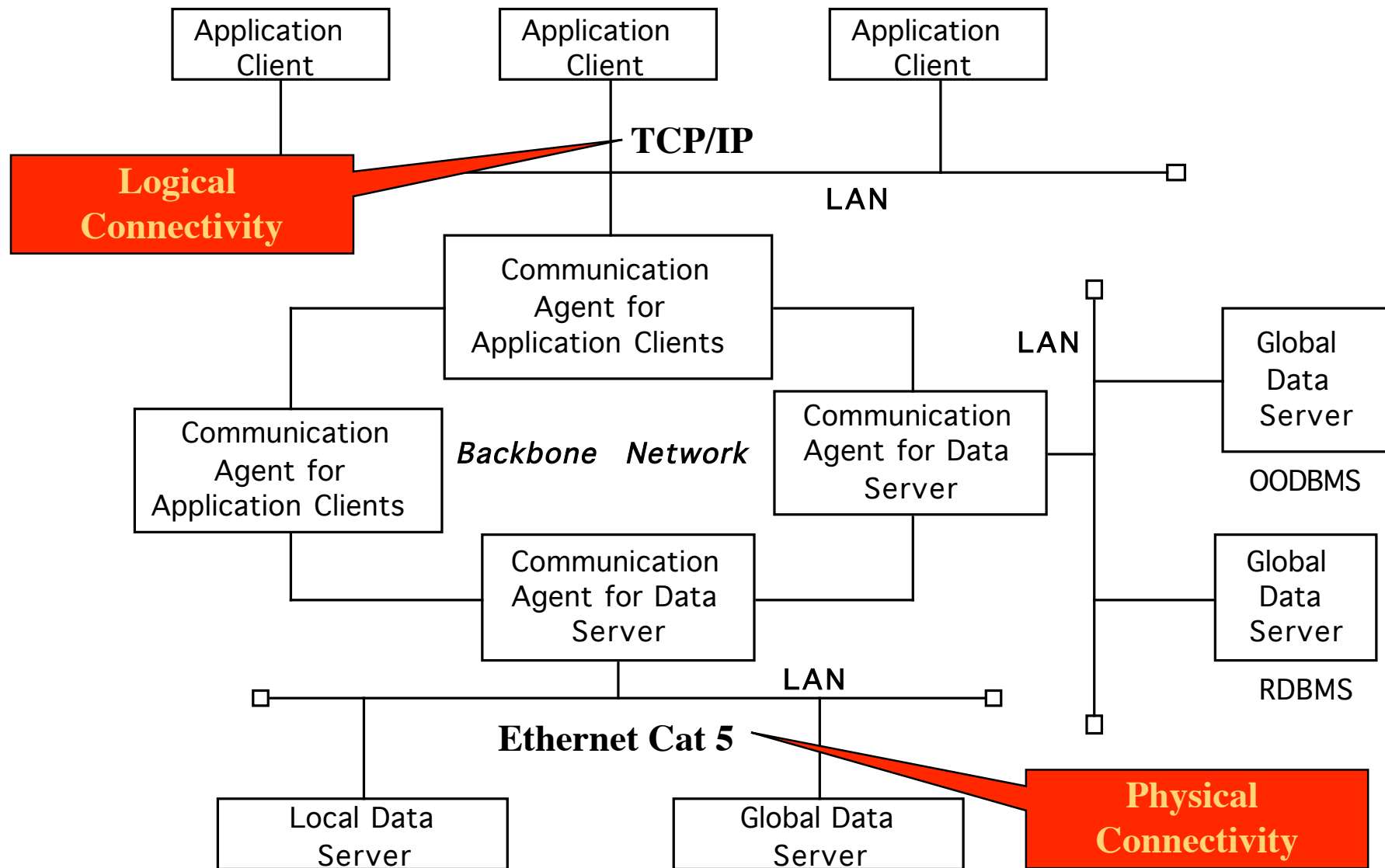
- This system design activity addresses two questions:
 - How shall we realize the subsystems: With hardware or with software?
 - How do we map the object model onto the chosen hardware and/or software?
 - Mapping the Objects:
 - Processor, Memory, Input/Output
 - Mapping the Associations:
 - Network connections

Mapping the Objects

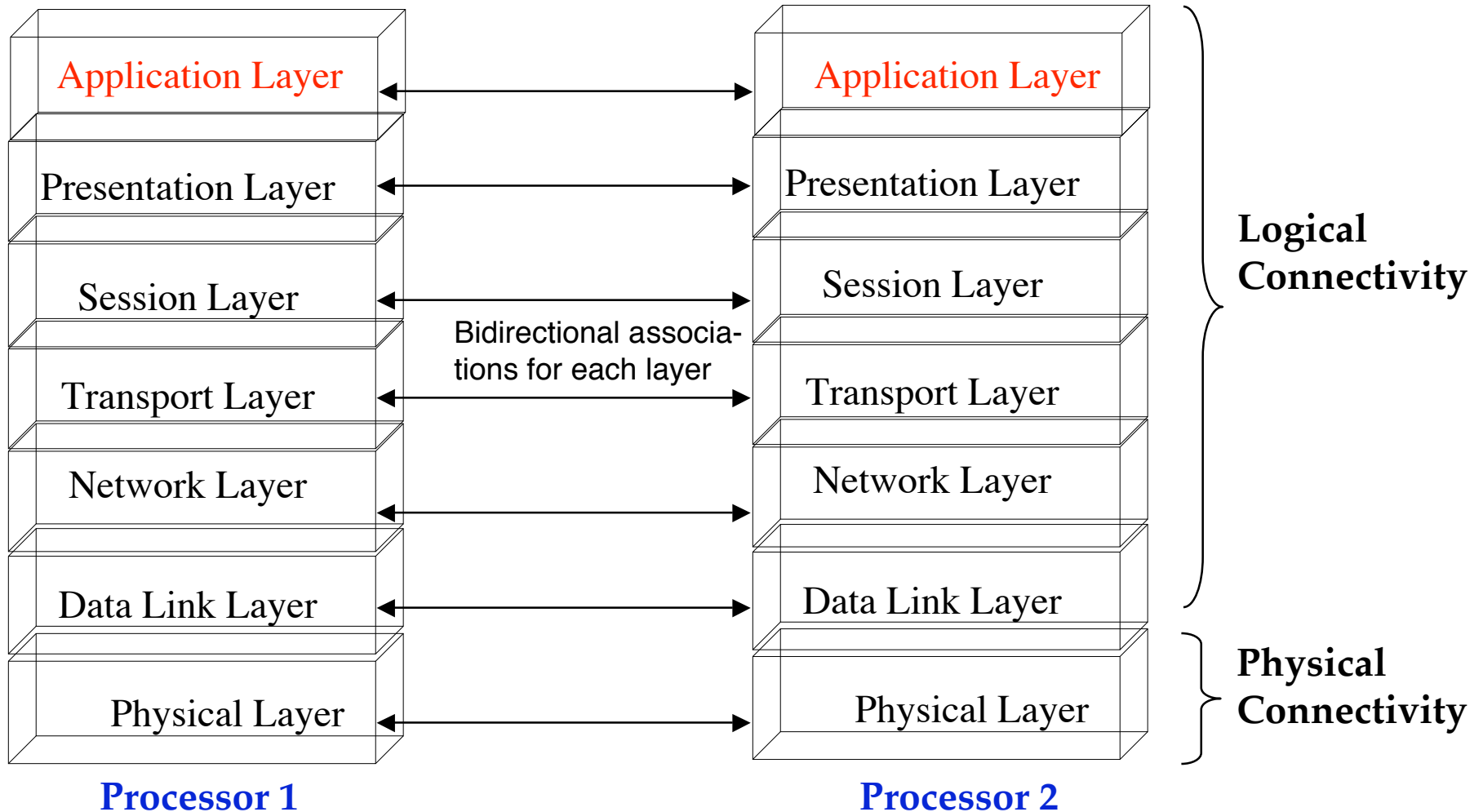
- Processor issues:
 - Is the computation rate too demanding for a single processor?
 - Can we get a speedup by distributing objects across several processors?
 - How many processors are required to maintain steady state load?
- Memory issues:
 - Is there enough memory to buffer bursts of requests?
- Input/Output issues:
 - Do we need an extra piece of hardware to handle the data generation rate?
 - Does the response time exceed 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.



Logical vs Physical Connectivity and the relationship to Subsystem Layering

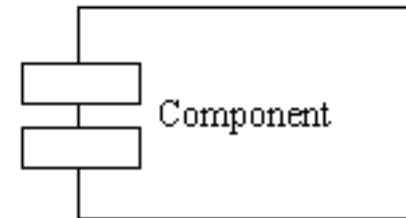


Hardware-Software Mapping Difficulties

- 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
 - Example: To send DVB-T signals, the system has to use components from a company that provides DVB-T transmitters.

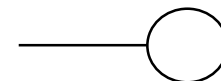
Hardware/Software Mappings in UML

- A **UML component** is a building block of the system. It is represented as a rectangle with tabs
- Components have different lifetimes:
 - Some exist only at design time
 - Classes, associations
 - Others exist until compile time
 - Source code, pointers
 - Some exist at link or only at runtime
 - Linkable libraries, executables, addresses
- The Hardware/Software Mapping addresses dependencies and distribution issues of UML components during system design.



Two New UML Diagram Types

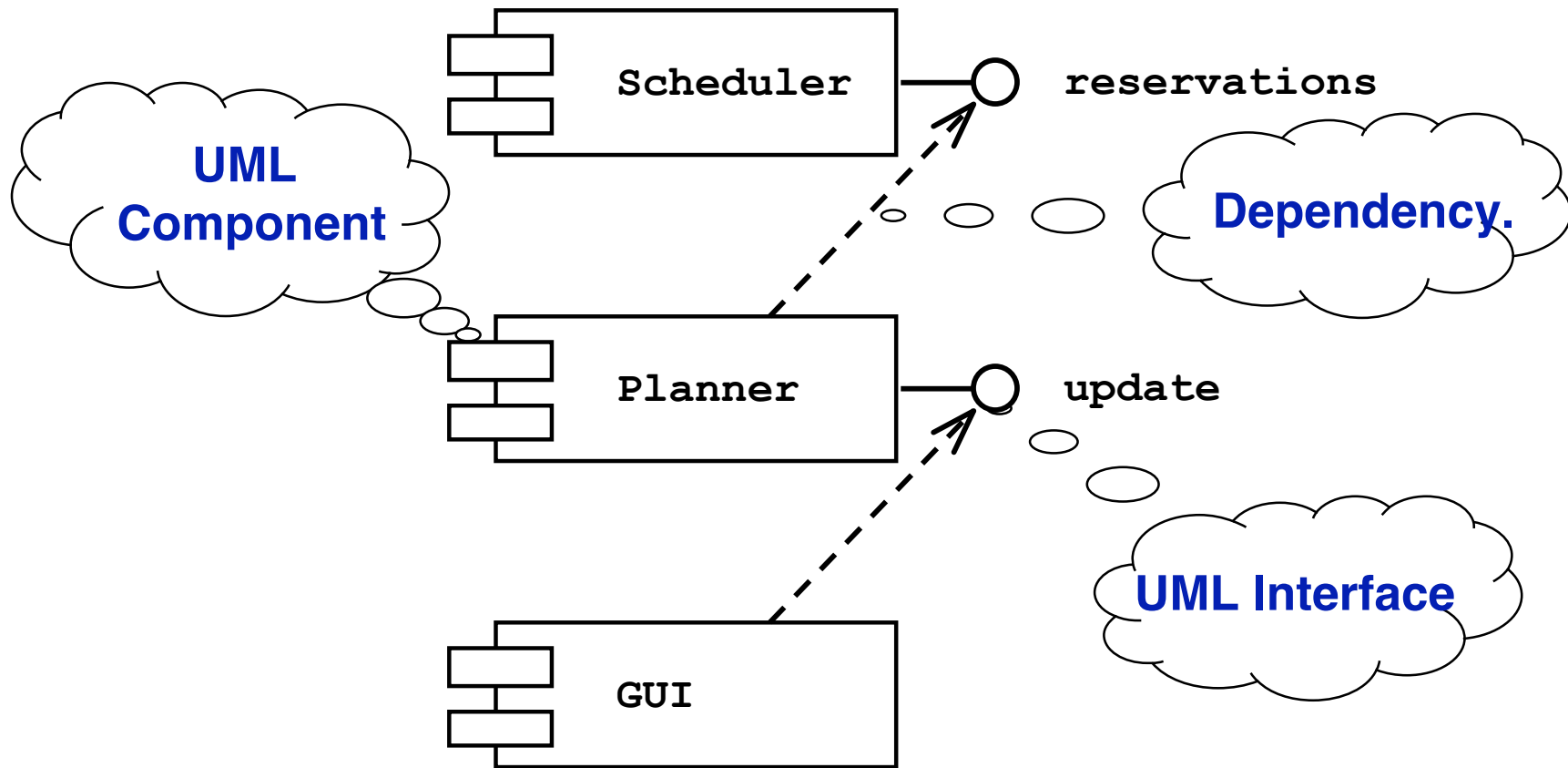
- **UML Component Diagram:**
 - Illustrates dependencies between components at design time, compilation time and runtime
- **UML Deployment Diagram:**
 - Illustrates the distribution of components at run-time.
 - Deployment diagrams use nodes and connections to depict the physical resources in the system.
- **UML Interface:**
 - A UML interface describes a group of operations used or created by UML components.
 - It is represented as a line with a circle.



Component Diagram

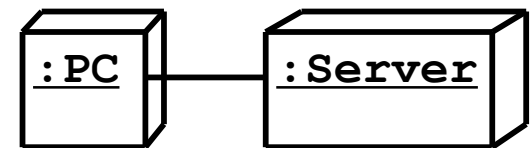
- **Component Diagram**
 - A graph of components connected by dependency relationships
 - Shows the dependencies among software components
 - source code, linkable libraries, executables
- Dependencies are shown as dashed arrows from the client component to the supplier component
 - The types of dependencies are implementation language specific.
- A component diagram may also be used to show dependencies on a subsystem interface:
 - Use a dashed arrow between the component and the UML interface it depends on.

Component Diagram Example



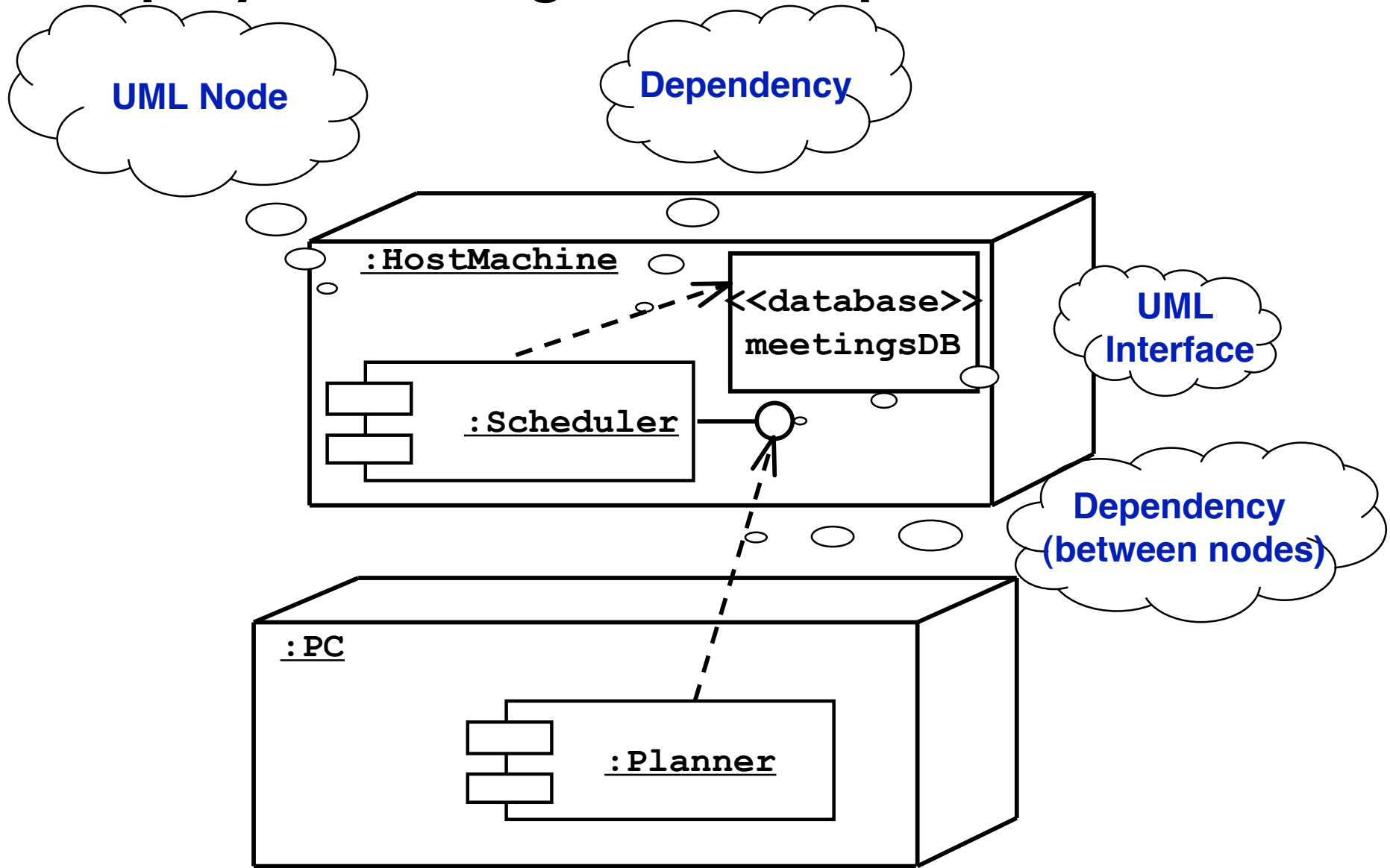
Deployment Diagram

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

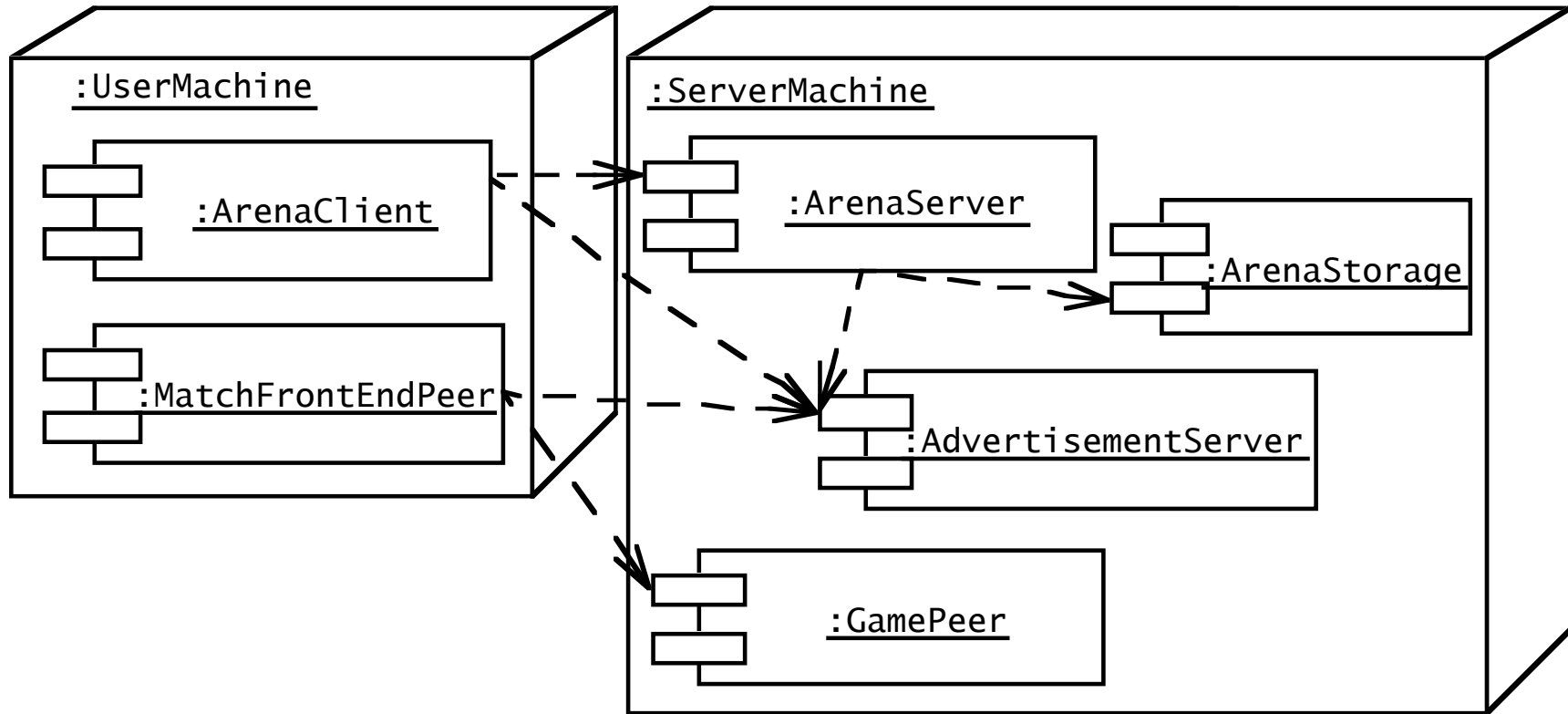


- A **deployment diagram** is a graph of nodes and connections (“communication associations”).
 - Nodes are shown as 3-D boxes
 - Connections are shown as solid lines
 - Nodes may contain components
 - Components may contain objects (indicating that the object is part of the component).

Deployment Diagram Example



ARENA Hardware/Software Mapping



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?
- Should the data format be extensible?

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
- More details in Lecture: *Mapping Models to Relational Schema*

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.

Access Matrix Example

The diagram illustrates an Access Matrix. A callout labeled 'Actors' points to the first column. A callout labeled 'Classes' points to the first row. A callout labeled 'Access Rights' points to the first cell of the matrix.

	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

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

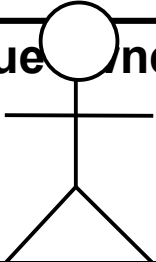
LeagueOwner, Arena, view()
LeagueOwner, League, edit()
LeagueOwner, Tournament, <<create>>
LeagueOwner, Tournament, view()
LeagueOwner, Tournament, schedule()
LeagueOwner, Tournament, archive()
LeagueOwner, Match, <<create>>
LeagueOwner, Match, end()

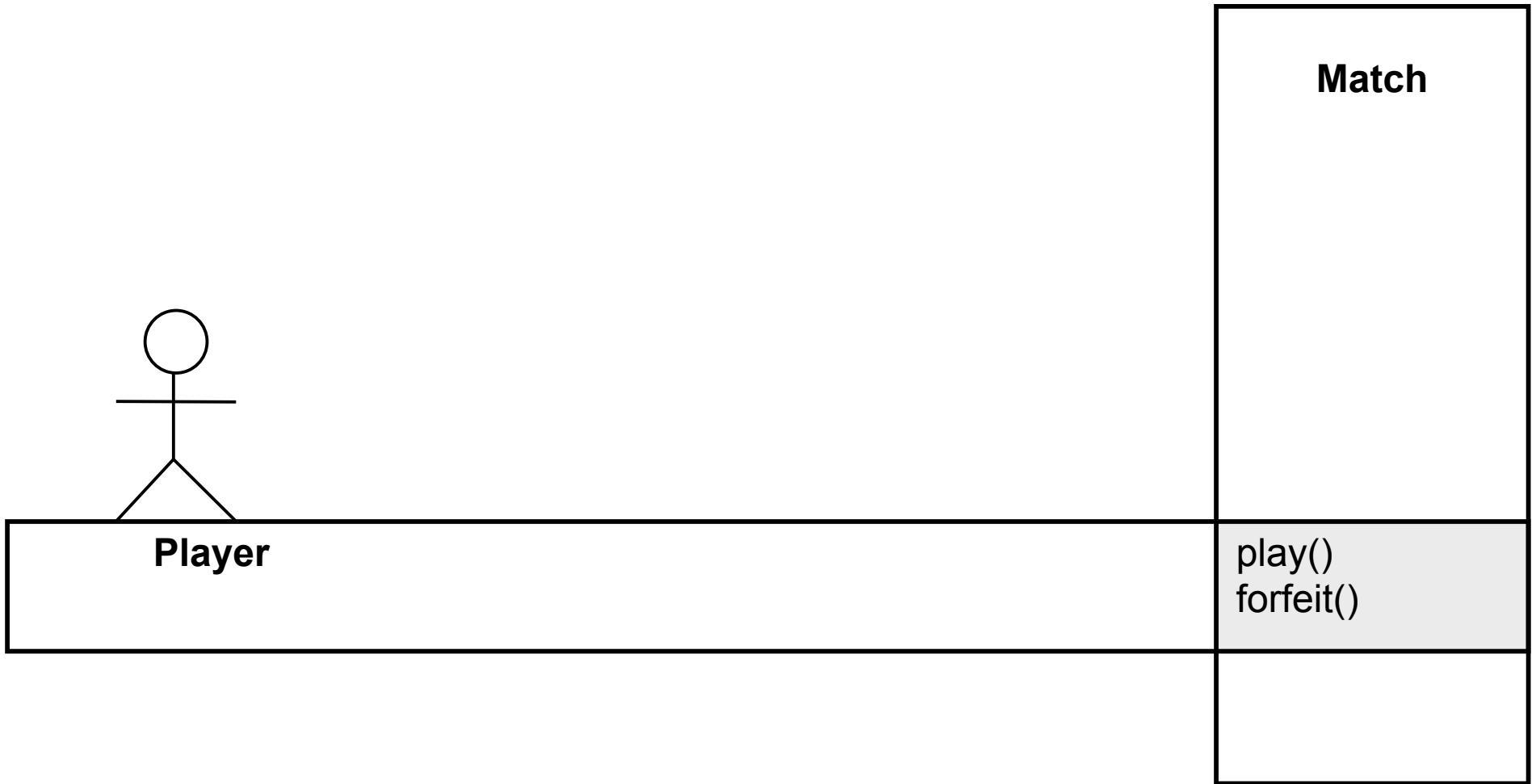
.

Better Access Matrix Implementations

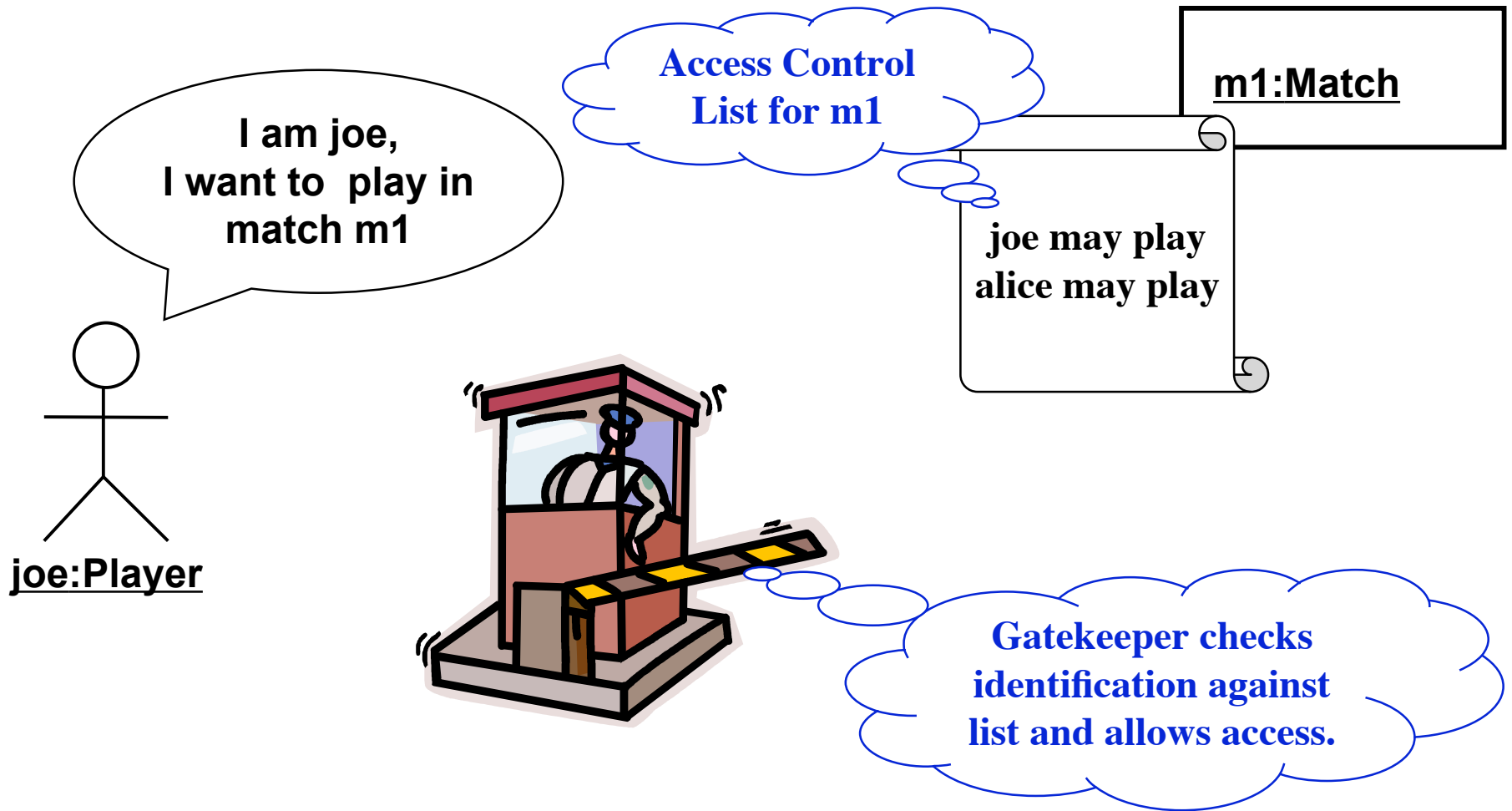
- **Access control list**
 - Associates a list of (actor,operation) pairs with each class to be accessed.
 - Every time an instance of this class is accessed, the access list is checked for the corresponding actor and operation.
- **Capability**
 - Associates a (class,operation) pair with an actor.
 - A capability provides an actor to gain control access to an object of the class described in the capability.

Access Matrix Example

	Arena	League	Tournament	Match
Operator	<<create>> createUser() view ()	<<create>> archive()		
League Owner 	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()



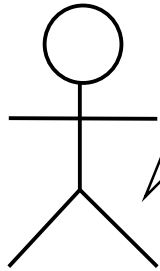
Access Control List Realization



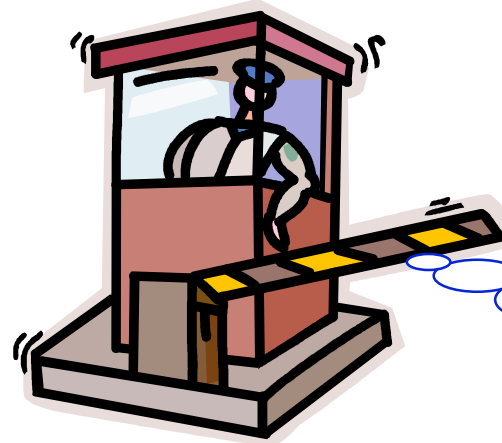
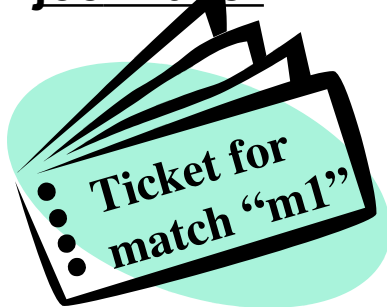
Capability Realization

m1:Match

Here's my ticket, I'd like to play in match m1



joe:Player



Gatekeeper checks if ticket is valid and allows access.

Capability

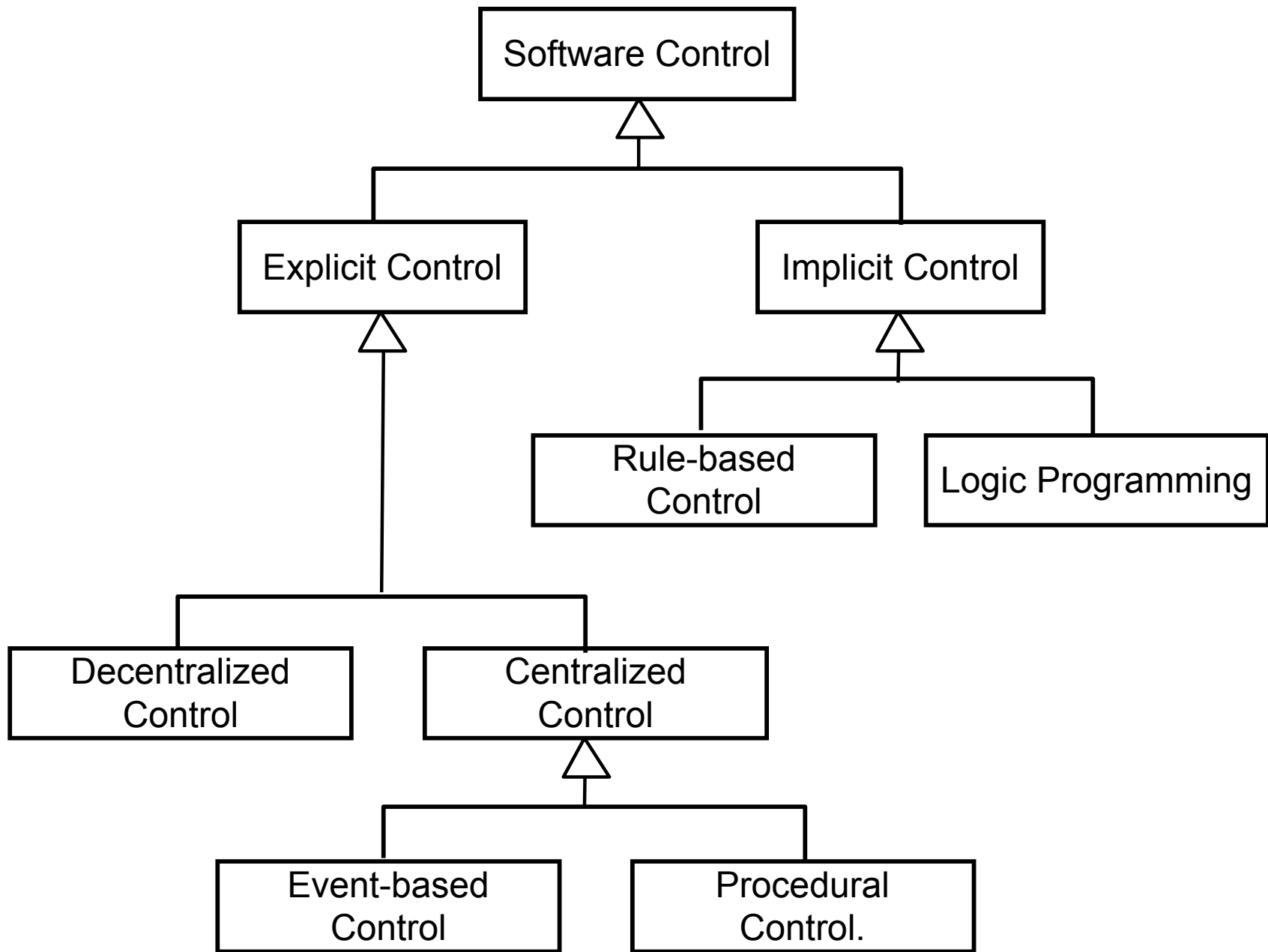
Global Resource Questions

- 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?

7. Decide on Software Control

Two major design choices:

1. Choose implicit control
 2. Choose explicit control
 - Centralized or decentralized
- **Centralized control:**
 - **Procedure-driven:** Control resides within program code.
 - **Event-driven:** Control resides within a dispatcher calling functions via callbacks.
 - **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.



Centralized vs. Decentralized Designs

- 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
- 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 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.

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 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?.

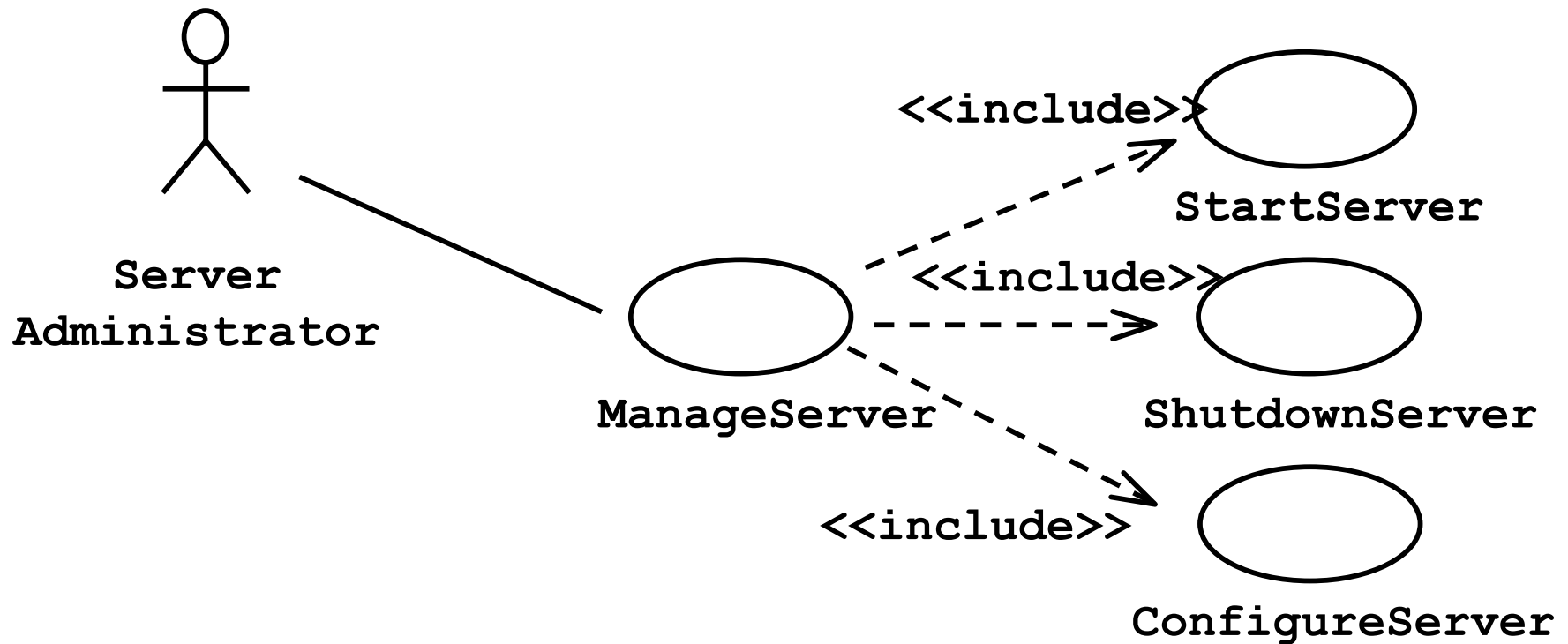
Modeling Boundary Conditions

- 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