Software Engineering I: Software Technology

WS 2008/09

Addressing Design Goals

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

Announcements

- Friday: No lecture
- Mid-term registration via TUMonline
  - Details on Thursday
  - Details about the tours to the airport are in the exercise portal.

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 Handlung
- Access Control List vs Capabilities
- Security

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

8. Boundary Conditions
- Initialization
- Termination
- Failure

Concurrency

- System Design Activities:
  - Identify concurrent objects and address design issues

- Nonfunctional Requirements to be addressed: Performance, Response time, latency, availability.

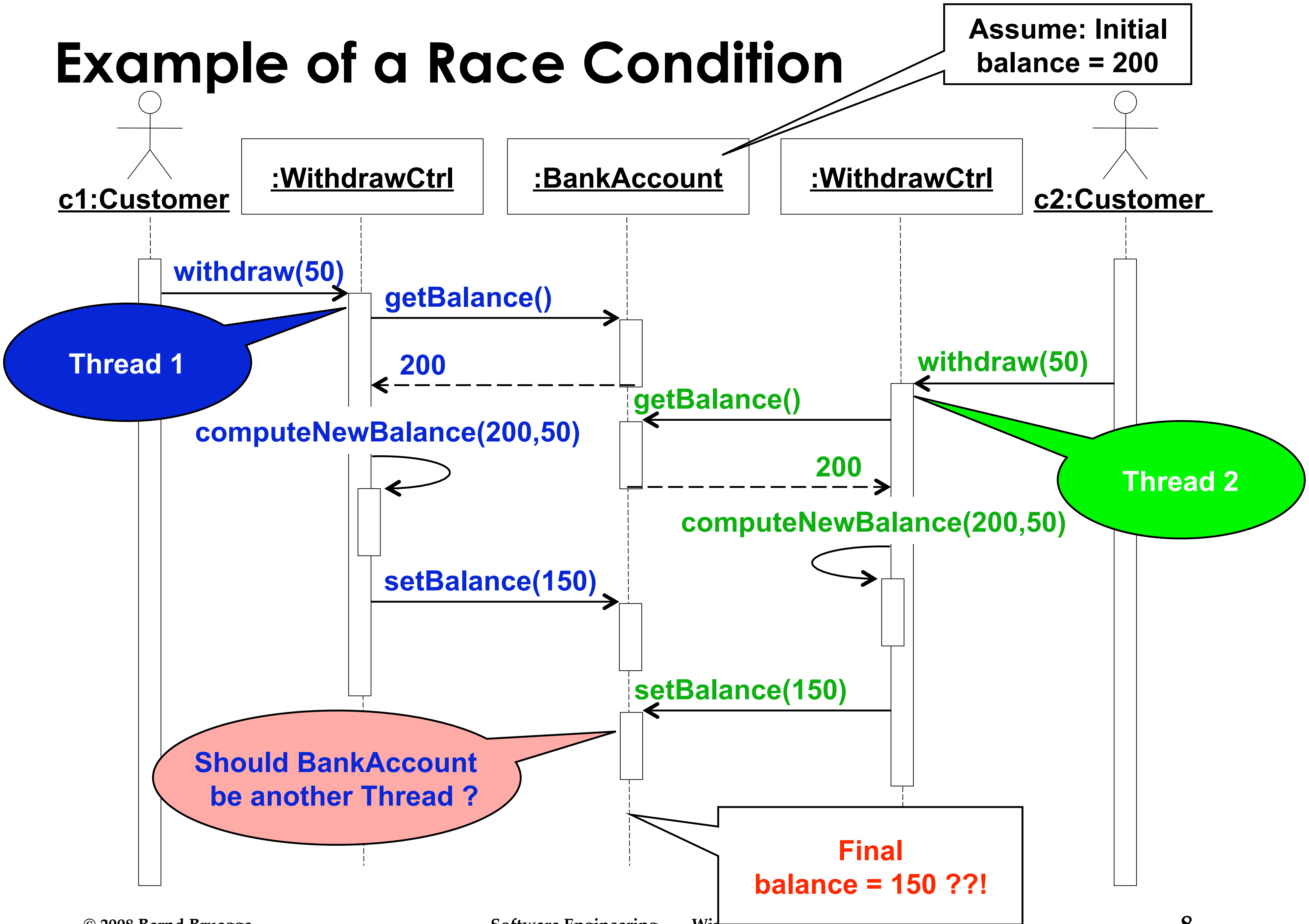
Concurrency (continued)

- 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
    - 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

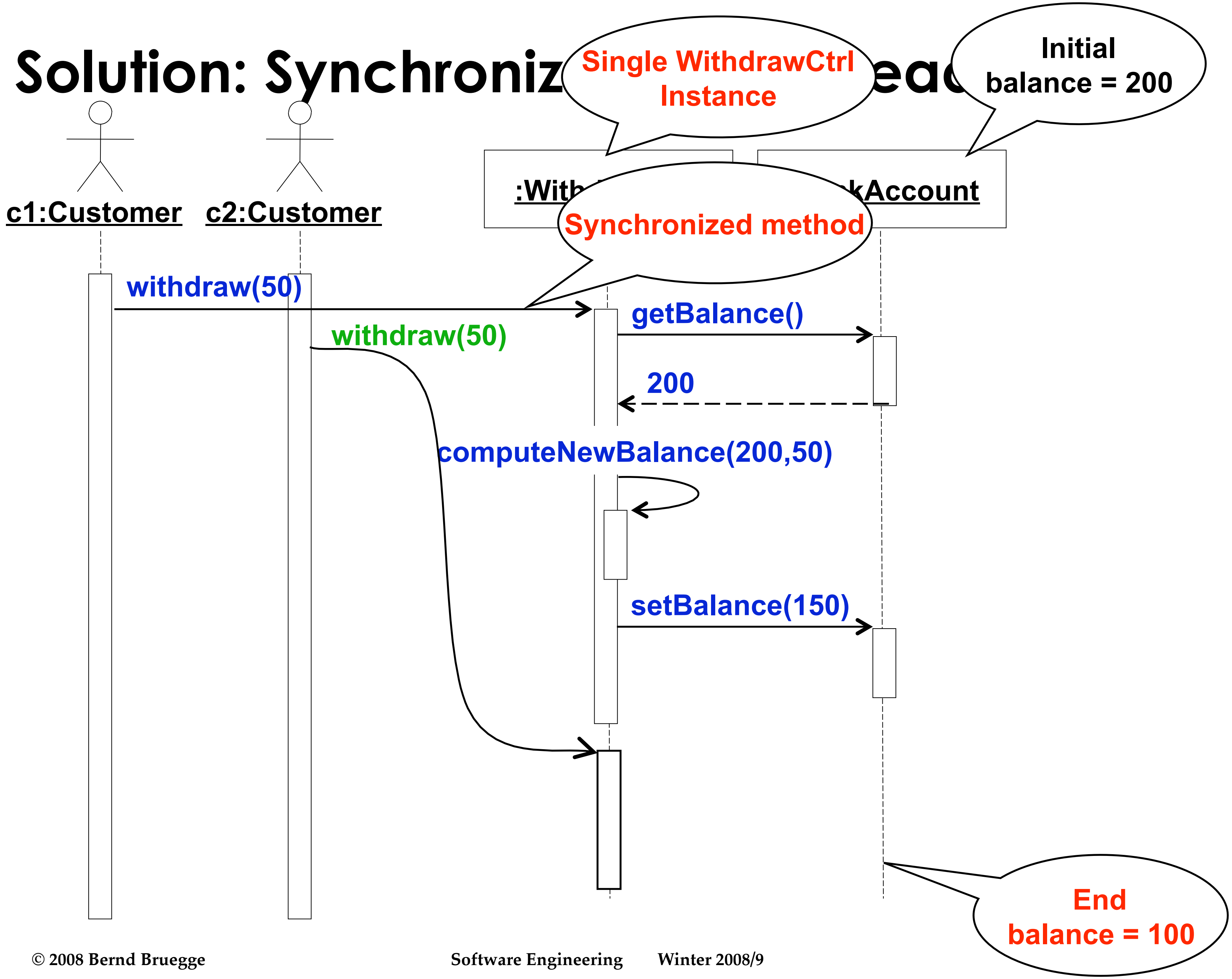
Thread of Control

- 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 different object and waits for another event
  - Thread splitting: Object does a *non-blocking send* of an event.
- Concurrent threads can lead to race conditions.
- A race condition (race hazard) is a system design flaw where the output of a process is depends on the sequence of other events.
  - First used in digital circuit design: Two signals racing each other to influence the output.

Example of a Race Condition



Solution: Synchroniz… ead



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
- Implementation needs to address starvation, deadlocks, fairness -> Topic 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 Handlung
- Access Control List vs Capabilities
- Security

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

8. Boundary Conditions
- Initialization
- Termination
- Failure

4. Hardware Software Mapping

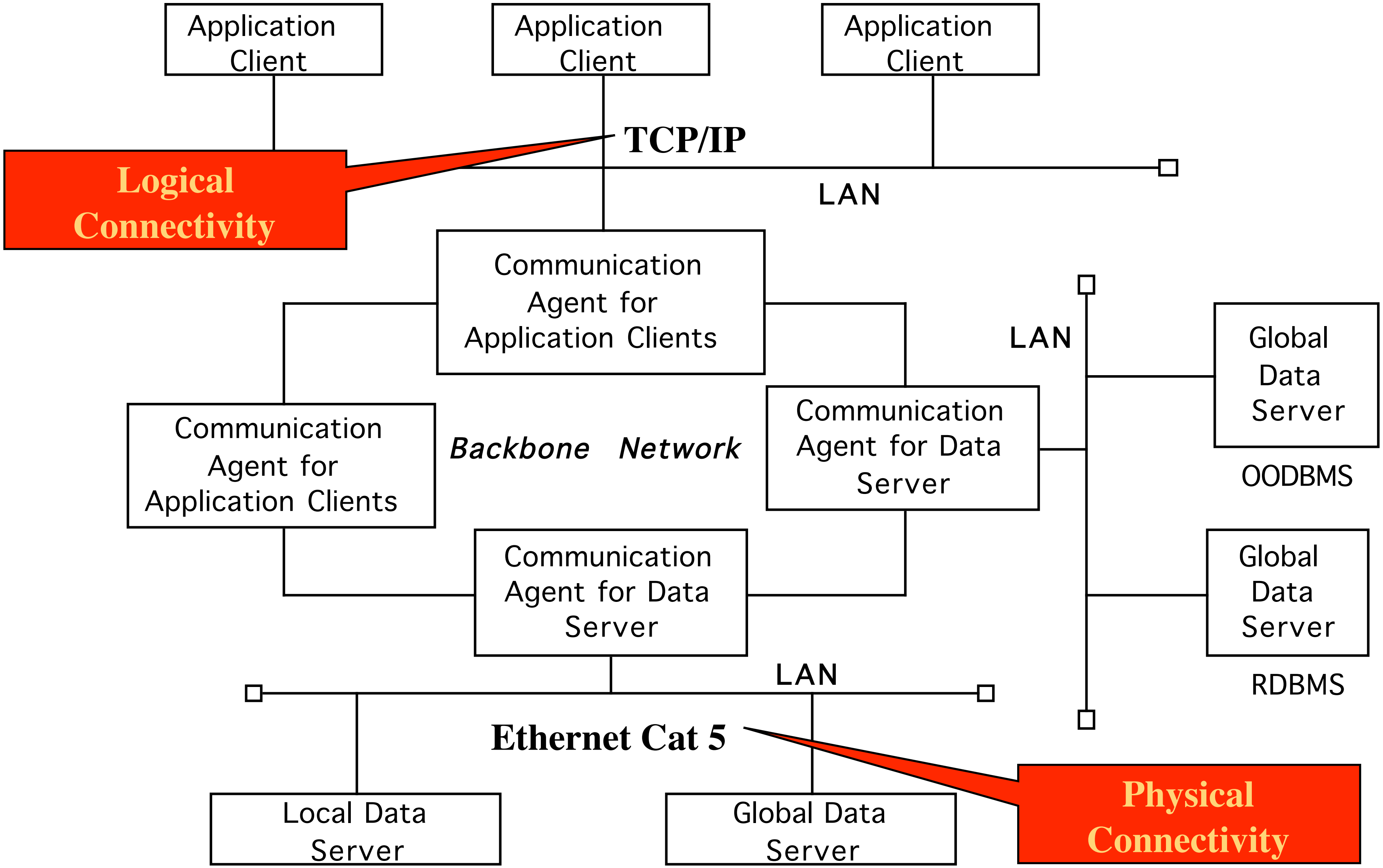
- This system design activity addresses two questions:
  - How do we realize the subsystems: With hardware or with software?
  - How do we map the object model onto the chosen hardware and/or software?
    - Mapping the Objects:
      - Processor, Memory, Input/Output
    - Mapping the Associations: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

Mapping Objects onto Hardware

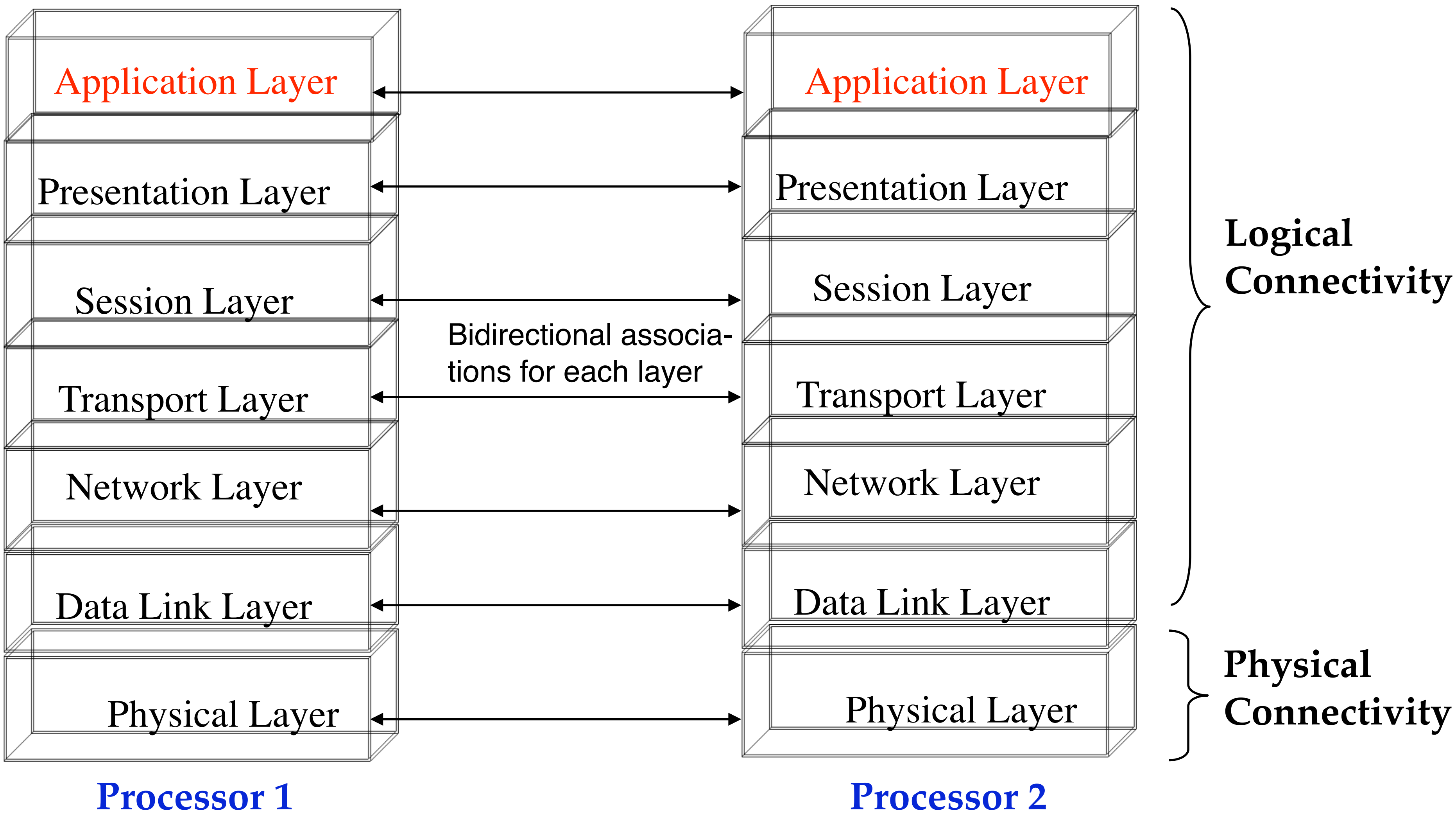
- Control Objects -> Processor
  - Is the computation rate too demanding for a single processor?
  - Can we get a speedup by distributing objects across several processors?
  - How many processors are required to maintain a steady state load?
- Entity Objects -> Memory:
  - Is there enough memory to 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.


Application Client
Application Client
Application Client
TCP/IP
LAN
Logical Connectivity
Communication Agent for Application Clients
LAN
Global Data Server
Communication Agent for Application Clients
Backbone Network
Communication Agent for Data Server
OODBMS
Communication Agent for Data Server
Global Data Server
RDBMS
LAN
Ethernet Cat 5
Local Data Server
Global Data Server
Physical Connectivity

Logical vs Physical Connectivity and the relationship to Subsystem Layering



Hardware-Software Mapping Difficulties

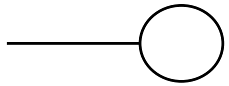
- Much of the difficulty of designing a system comes from addressing externally-imposed hardware and software constraints.
  - Certain tasks have to be at specific locations
    - Example: Withdrawing money from an ATM machine
  - Some hardware components have to be used from a specific manufacturer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

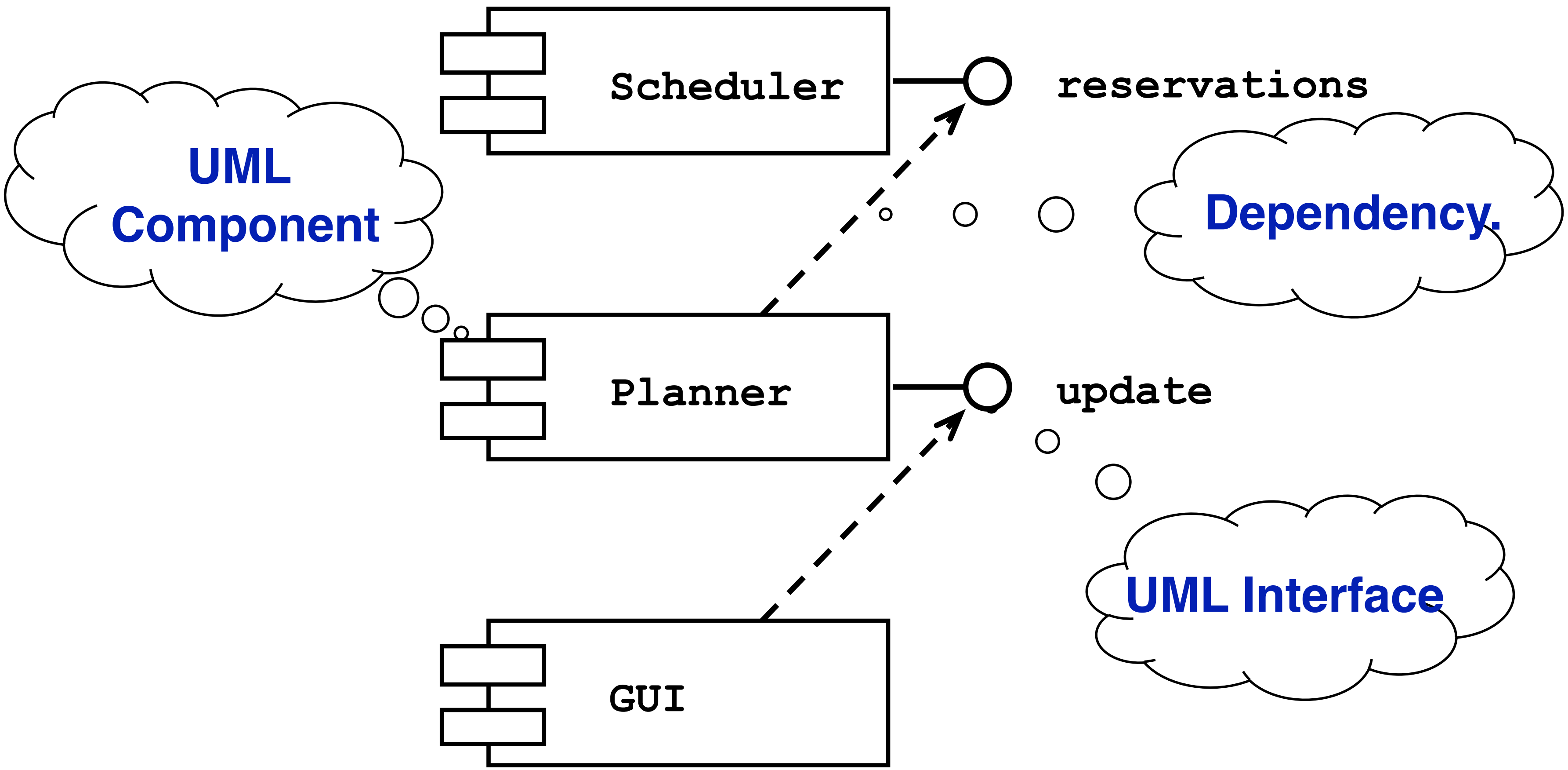
- UML Component Diagram:
  - Illustrates dependencies between components at design time, compilation time and runtime
- UML Deployment Diagram:
  - Illustrates the distribution of components at run-time.
  - Deployment diagrams use nodes and connections to depict the physical resources in the system.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

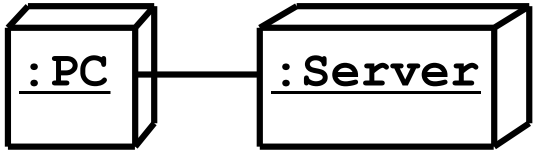
- Component Diagram
  - A graph of components connected by dependency relationships
  - Shows the dependencies among software components
    - source code, linkable libraries, executables
- Dependencies are shown as dashed arrows from the client component to the supplier component
  - The types of dependencies 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



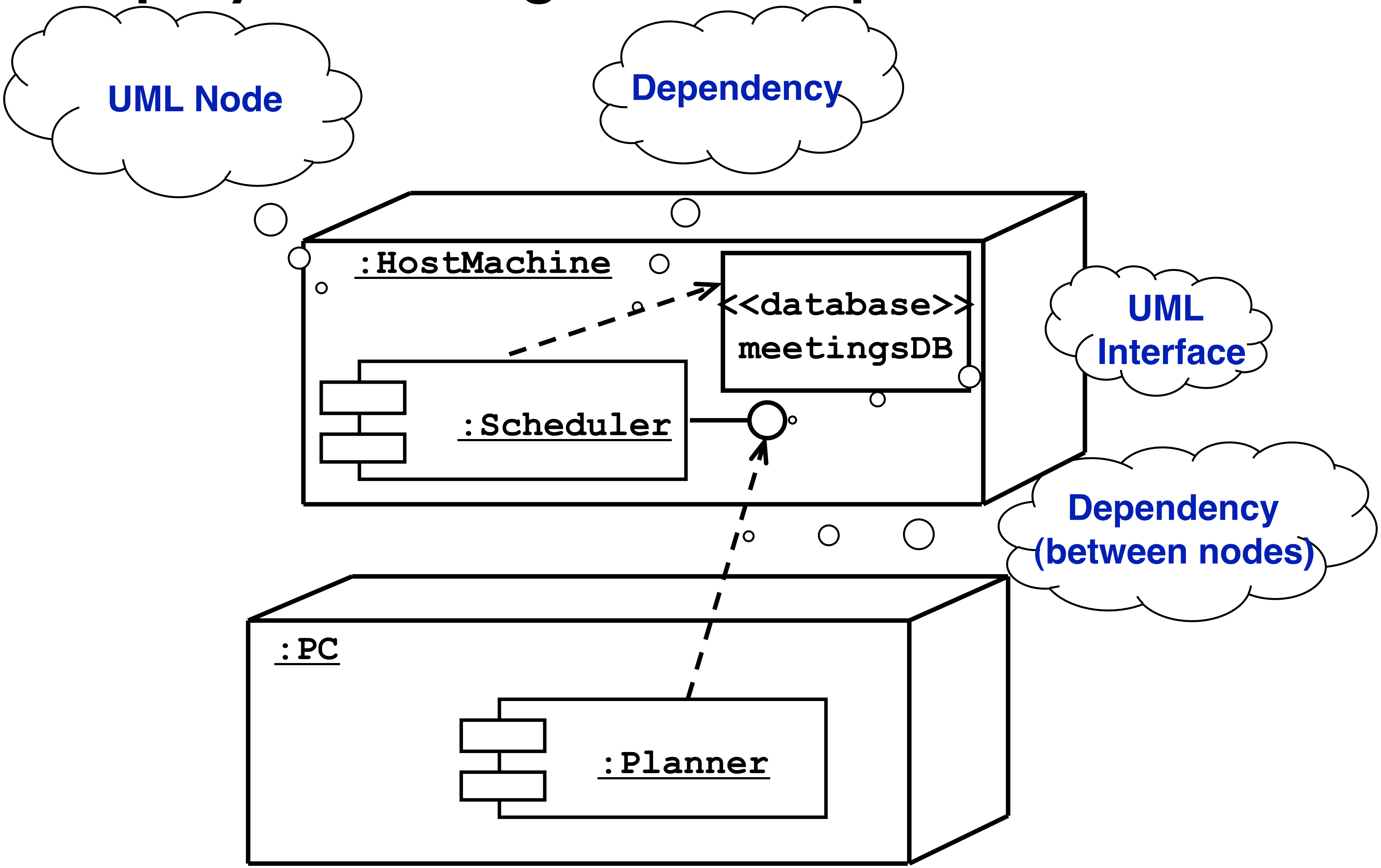
Deployment Diagram

- Deployment diagrams are useful for showing a system design after the following system design decisions have been made:
  1. Subsystem decomposition
  2. Concurrency
  3. Hardware/Software Mapping

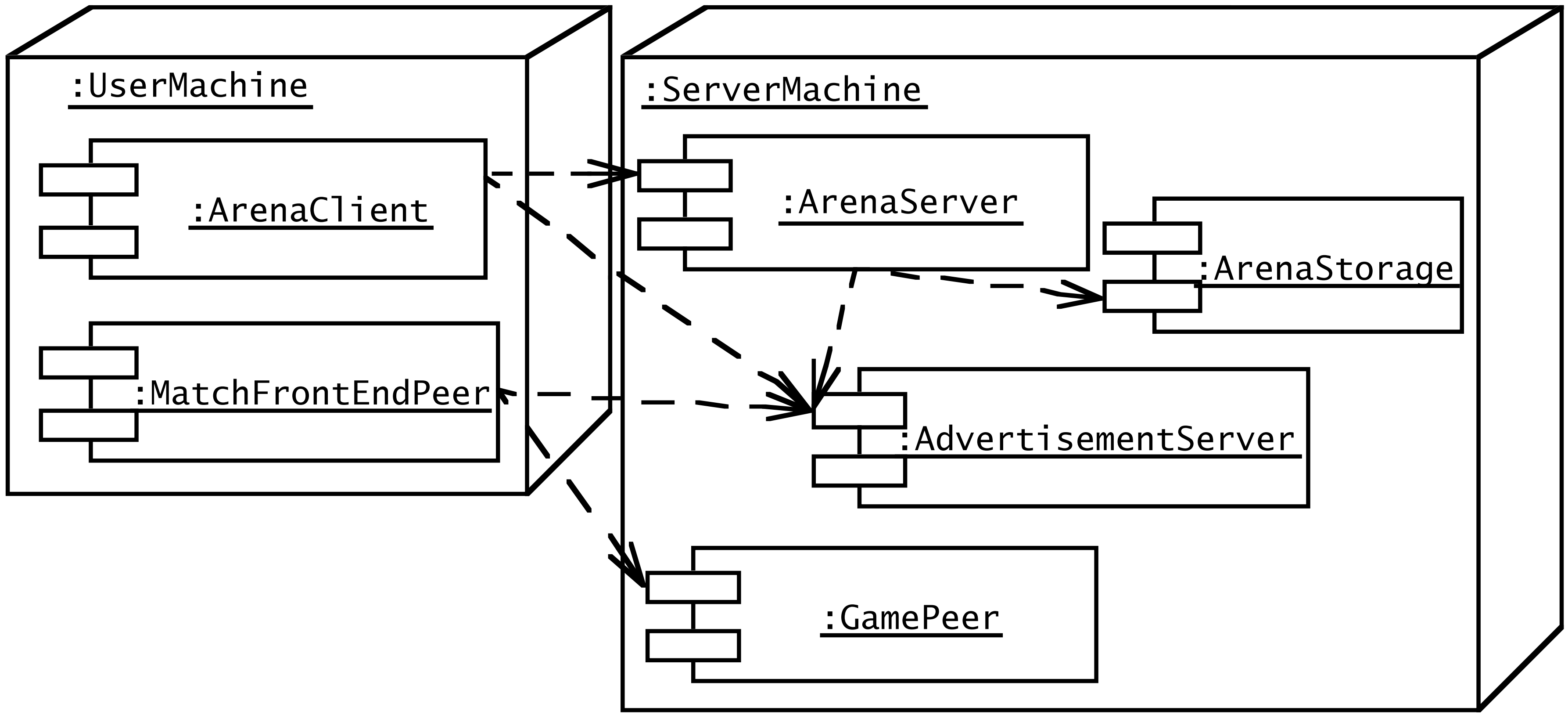


- A deployment diagram is a graph of nodes and connections ("communication associations").
  - Nodes are shown as 3-D boxes
  - Connections are shown as solid lines
  - Nodes may contain components
  - Components may contain objects (indicating that the object is part of the component).

Deployment Diagram Example



ARENA Hardware/Software Mapping



5. Data Management

- Some objects – most of the entity objects - in the system model need to be persistent:
  - Values of the attributes of a persistent object 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.

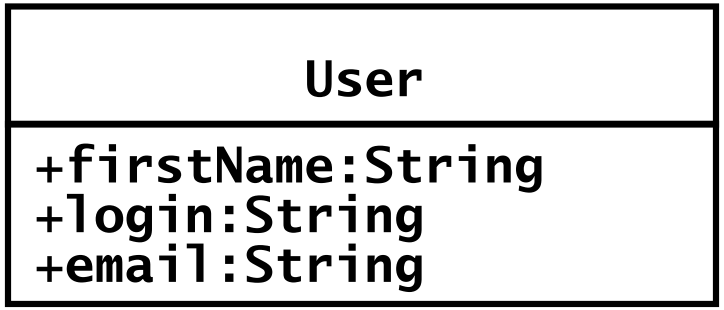
Data Management Questions

- 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 to Relational Databases

- UML object models can be mapped to relational databases
- The basic idea of 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

Mapping a Class to a Table



User table

id:long	firstName:text[25]	login:text[8]	email:text[32]

Primary and Foreign Keys

- Any set of attributes that could be used to uniquely identify any data record in a relational table is called a **candidate key**
- The actual candidate key that is used in the application to identify the records is called the **primary key**
  - The primary key of a table is a set of attributes whose values uniquely identify the data records in the table
- A **foreign key** is an attribute (or a set of attributes) that references the primary key of another table.

Example for Primary and Foreign Keys

User table

Primary key: login

firstName	login	email
"alice"	"am384"	"am384@mail.org"
"john"	"js289"	"john@mail.de"
"bob"	"bd"	"bobd@mail.ch"

Candidate key: login — Candidate key: email

League table

name	login
"tictactoeNovice"	"am384"
"tictactoeExpert"	"bd"
"chessNovice"	"js289"

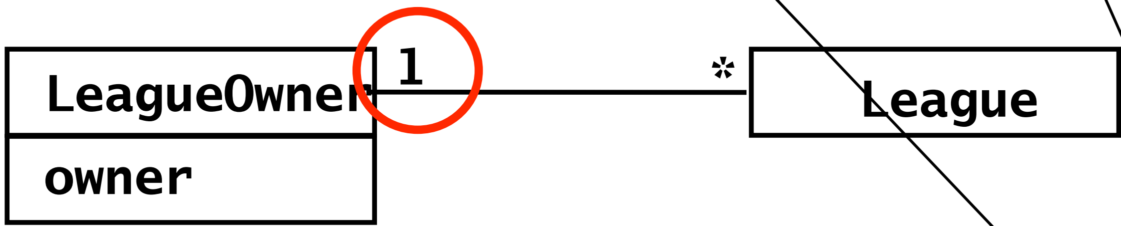
Foreign key referencing User table: login

Buried Association

- Associations with multiplicity "one" can be implemented using a foreign key

For one-to-many associations we add the foreign key to the table representing the class on the "many" end

For all other associations we can select either class at the end of the association.



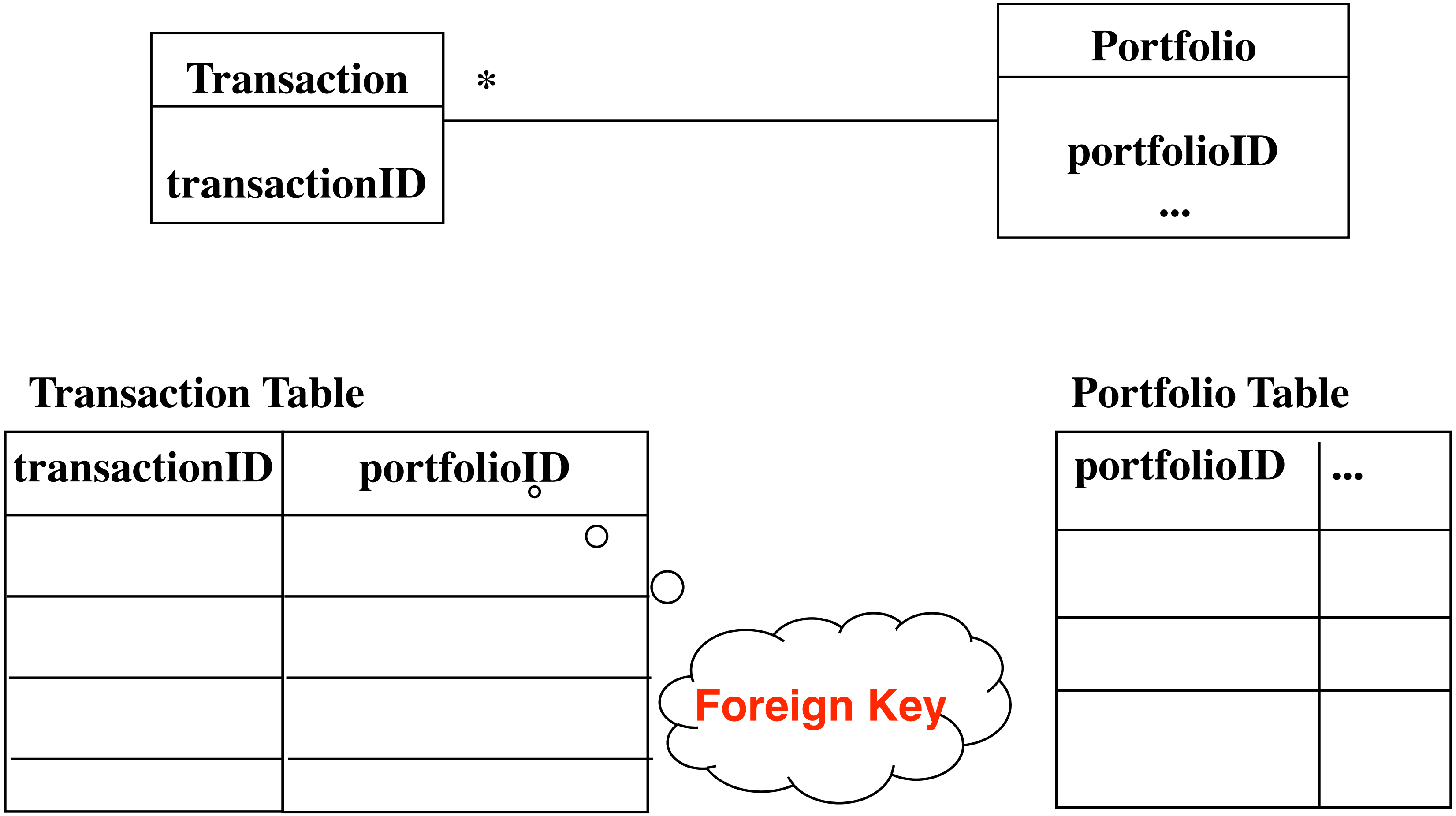
LeagueOwner table

id:long	...

League table

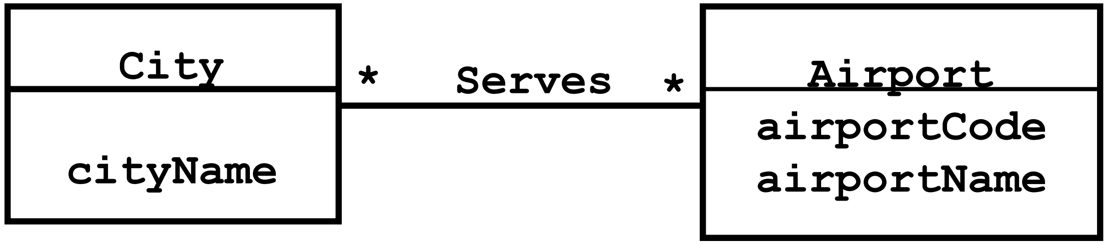
id:long	...	owner:long

Another Example for Buried Association



Mapping Many-To-Many Associations

In this case we need a separate table for the association



Separate table for the association “Serves”

Primary Key

City Table

cityName
Houston
Albany
Munich
Hamburg

Airport Table

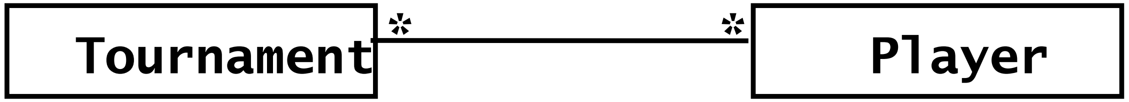
airportCode	airportName
IAH	Intercontinental
HOU	Hobby
ALB	Albany County
MUC	Munich Airport
HAM	Hamburg Airport

Serves Table

cityName	airportCode
Houston	IAH
Houston	HOU
Albany	ALB
Munich	MUC
Hamburg	HAM

Another Many-to-Many Association Mapping

We realize the Tournament/Player association as a separate table "TournamentPlayerAssociation "



Tournament table

id	name	...
23	novice	
24	expert	

TournamentPlayerAssociation table

tournament	player
23	56
23	79

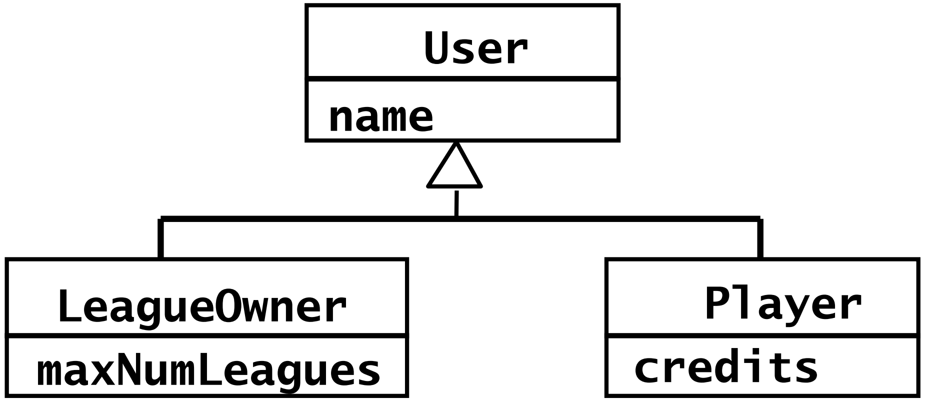
Player table

id	name	...
56	alice	
79	john	

Realizing Inheritance

- Relational databases do not support inheritance
- Two possibilities to map an inheritance association to a database schema
  - With a separate table ("vertical mapping")
    - The attributes of the superclass and the subclasses are mapped to different tables
  - By duplicating columns ("horizontal mapping")
    - There is no table for the superclass
    - Each subclass is mapped to a table containing the attributes of the superclass and the attributes of the subclass

Realizing inheritance with a separate table (Vertical mapping)



User table

id	name	...
56	zoe	
79	john	

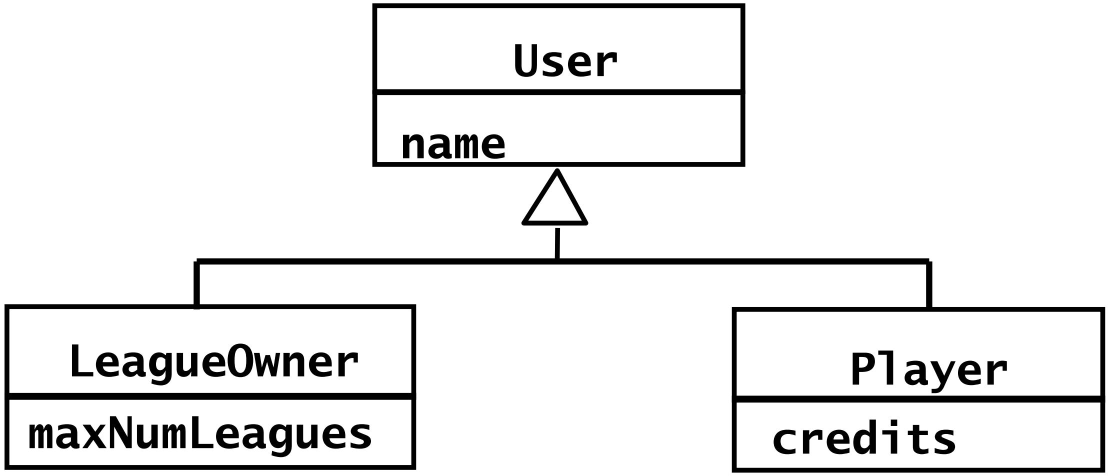
LeagueOwner table

id	maxNumLeagues	...
56	12	

Player table

id	credits	...
79	126	

Realizing inheritance by duplicating columns (Horizontal Mapping)



LeagueOwner table

id	name	maxNumLeagues	...
56	zoe	12	

Player table

id	name	credits	...
79	john	126	

Comparison: Separate Tables vs Duplicated Columns

- The trade-off is between modifiability and response time
  - How likely is a change of the superclass?
  - What are the performance requirements for queries?
- Separate table mapping (Vertical mapping)
  - ☺ We can add attributes to the superclass easily by adding a column to the superclass table
  - ☹ Searching for the attributes of an object requires a join operation
- Duplicated columns (Horizontal Mapping)
  - ☹ Modifying the database schema is more complex and error-prone
  - ☺ Individual objects are not fragmented across a number of tables, resulting in faster queries.

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

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

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

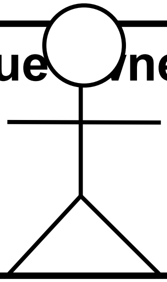
  LeagueOwner, **Arena**, view()
  LeagueOwner, League, edit()
  LeagueOwner, **Tournament**, <<create>>
  LeagueOwner, **Tournament**, view()
  LeagueOwner, **Tournament**, schedule()
  LeagueOwner, **Tournament**, archive()
  LeagueOwner, **Match**, <<create>>
  LeagueOwner, **Match**, end()

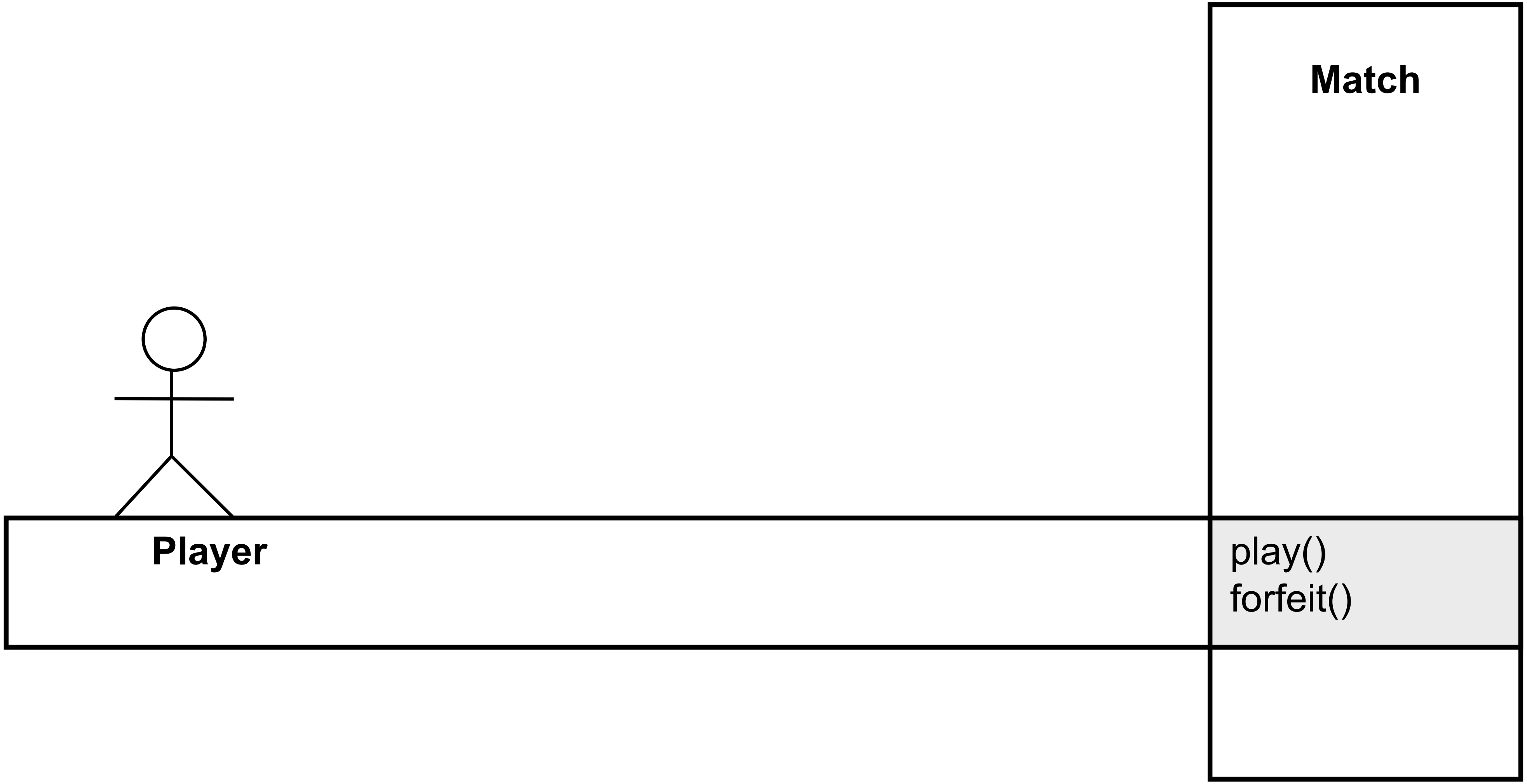
.

Better Access Matrix Implementations

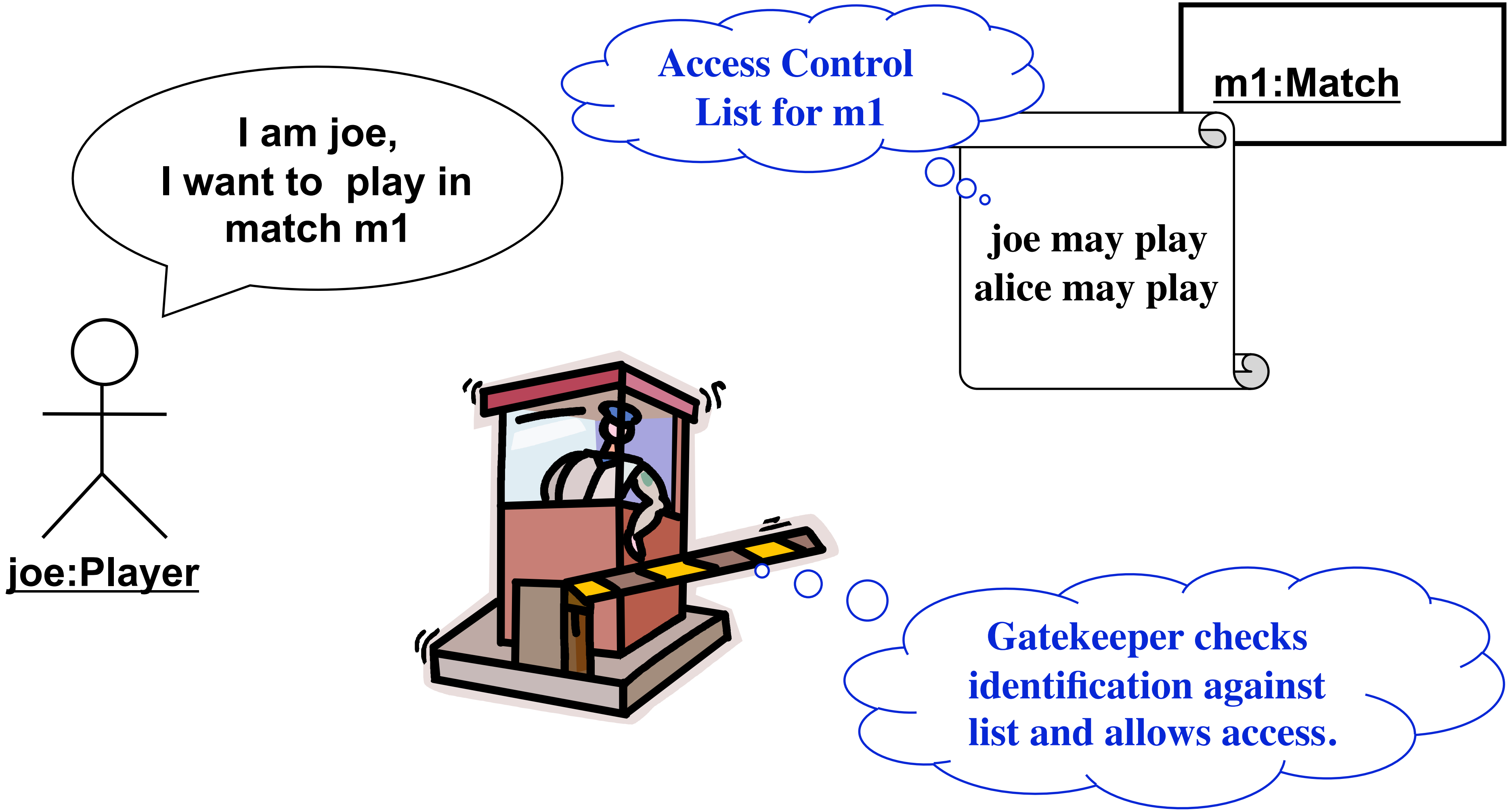
- Access control list
  - Associates a list of (actor,operation) pairs with each class to be accessed.
  - Every time an instance of this class is accessed, the access list is checked for the corresponding actor and operation.
- Capability
  - Associates a (class,operation) pair with an actor.
  - A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

Access Matrix Example

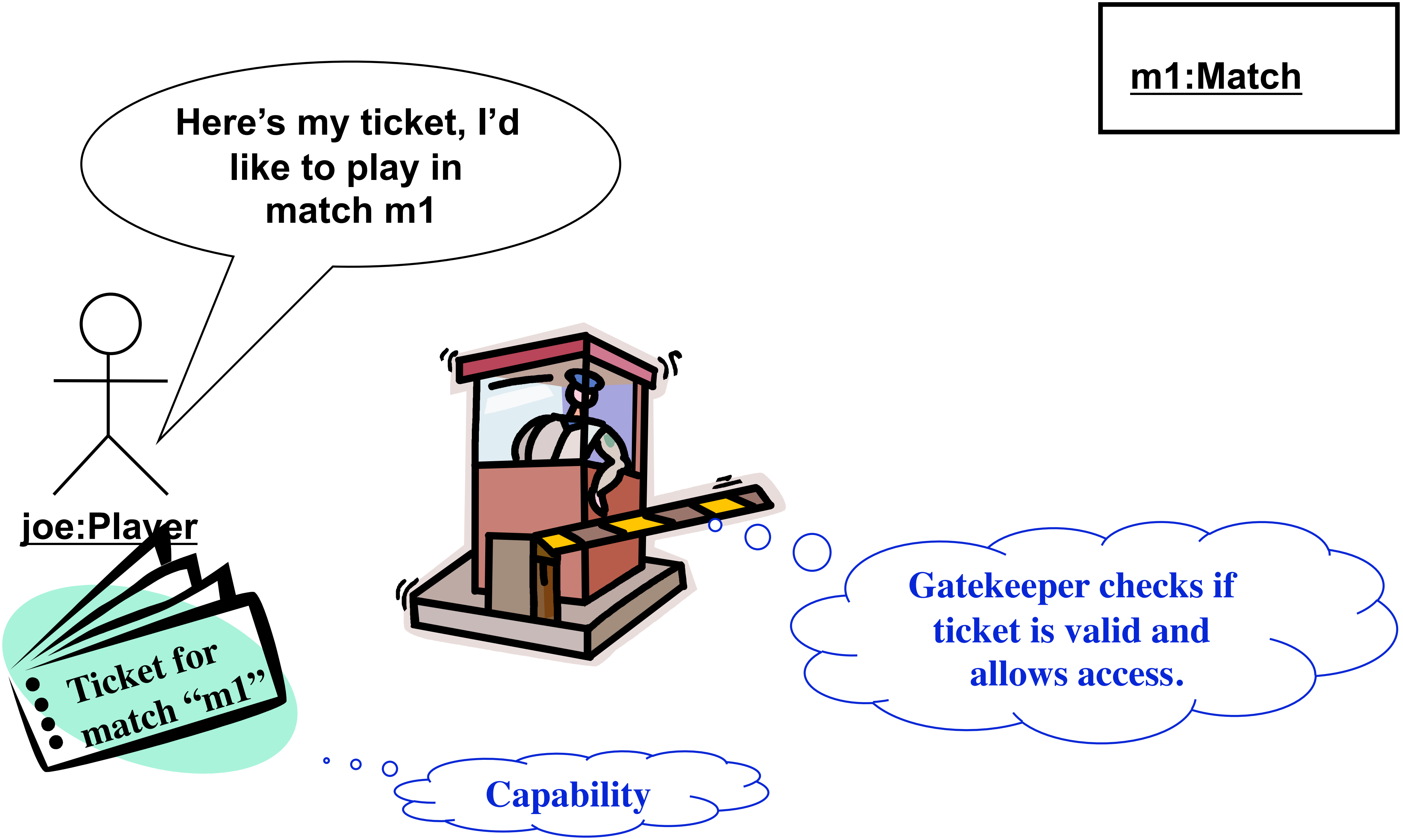
	Arena	League	Tournament	Match
Operator	<<create>> createUser() view ()	<<create>> archive()		
LeagueOwner 	view ()	edit ()	<<create>> archive() schedule() view()	<<create>> end()
Player	view() applyForOwner()	view() subscribe()	applyFor() view()	play() forfeit()
Spectator	view() applyForPlayer()	view()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


Match
Player
play()
forfeit()

Access Control List Realization



Capability Realization



Global Resource Questions

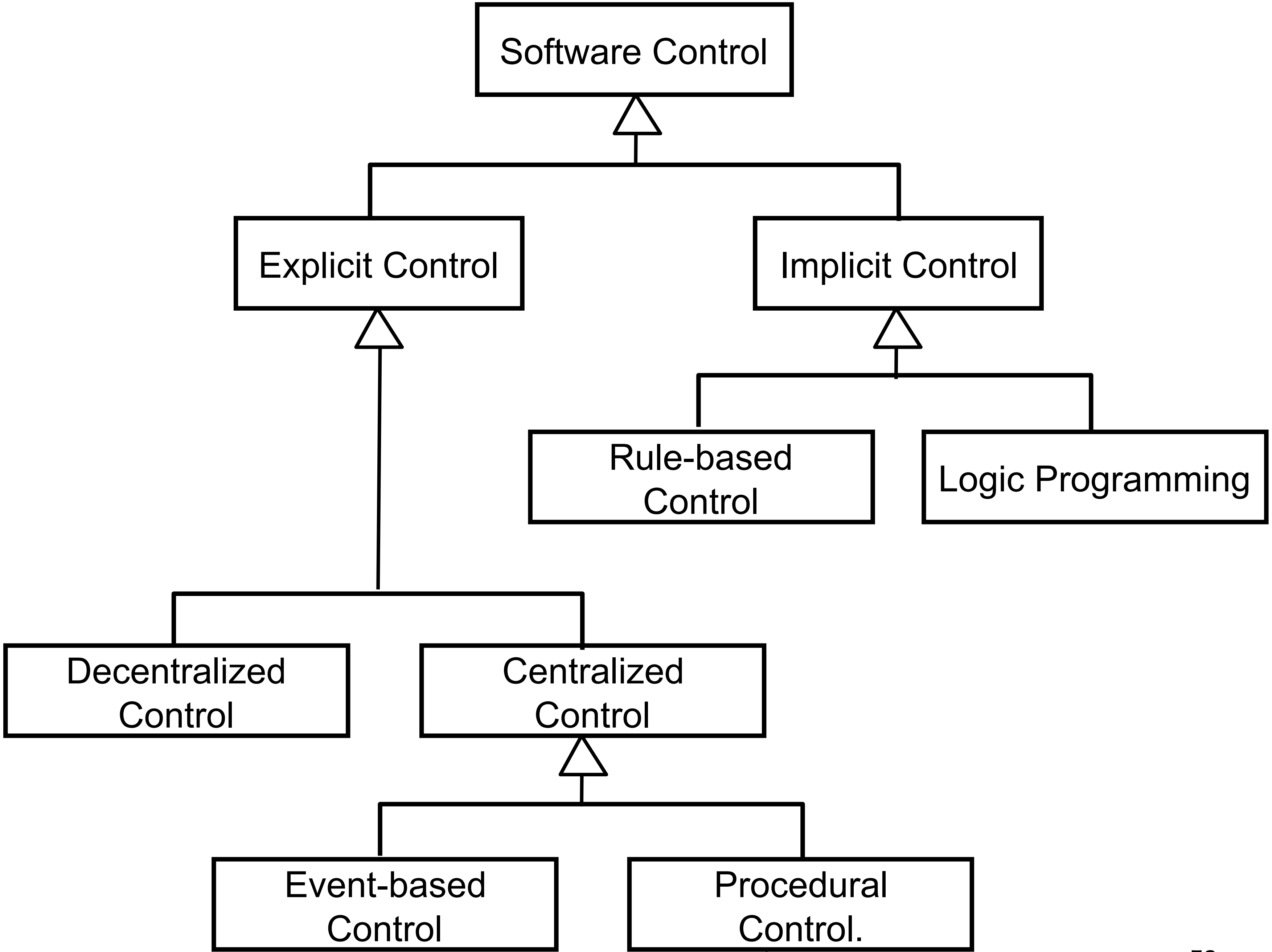
- Does the system need authentication?
- If yes, what is the authentication scheme?
  - User name and password? Access control list
  - Tickets? Capability-based
- What is the user interface for authentication?
- Does the system need a network-wide name server?
- How is a service known to the rest of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



Software Control
Explicit Control
Implicit Control
Rule-based Control
Logic Programming
Decentralized Control
Centralized Control
Event-based Control
Procedural Control.

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?.

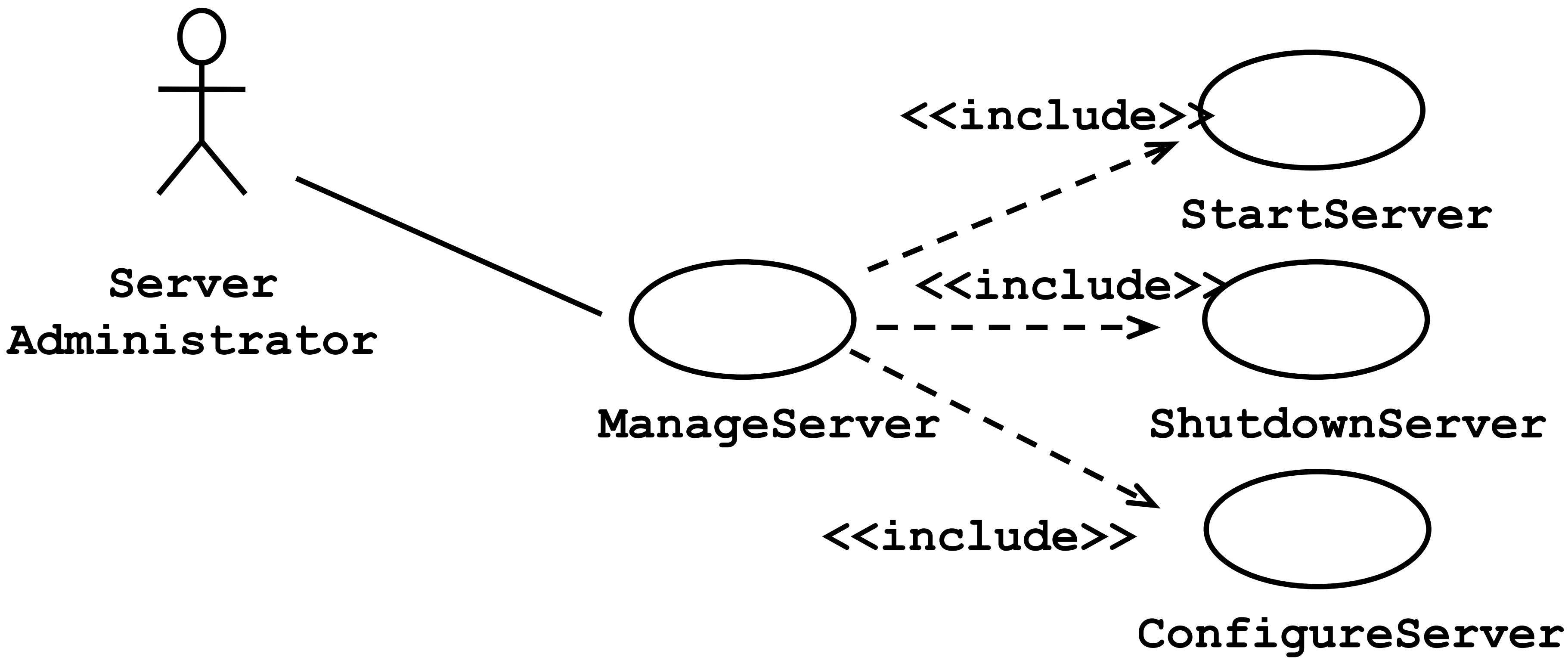
Modeling Boundary Conditions

- 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
- Mapping Object models to relational databases.