General Design Issues

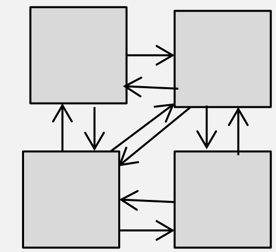
Lecture 13

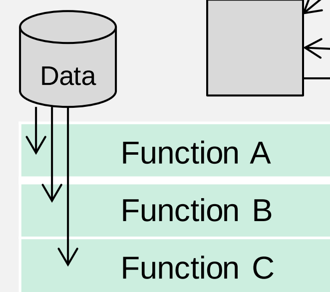
Overview

System Design I

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

System Design Concepts



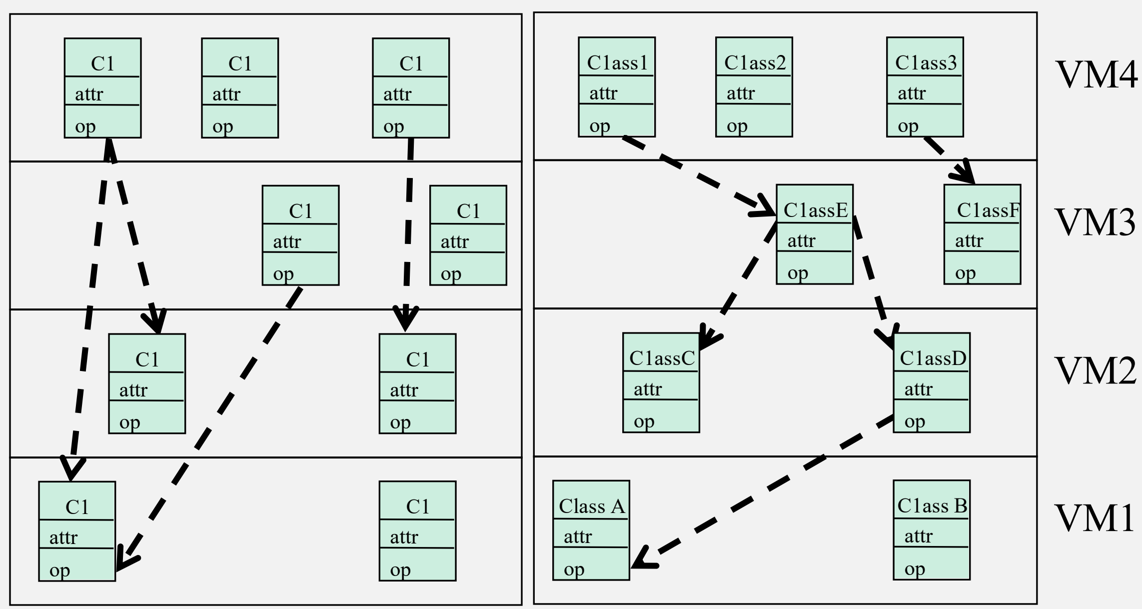


Subsystems

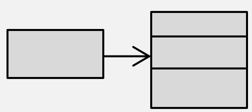
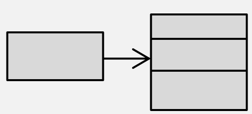
- **Coupling:** dependency between two subsystems
- **Cohesion:** dependencies within a subsystem
- Desire LOW coupling and HIGH cohesion

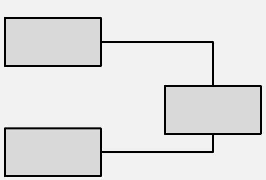
Refinement

- Layering
- Partitions



Software Architecture Patterns

- Repository 
- Model/View/Controller
- Client/Server 



Overview

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: Eight Issues

System Design

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

2. Subsystem Decomposition
   - Layers vs Partitions
   - Coherence & Coupling

3. Identify Concurrency
   - Identification of Parallelism (Processes, Threads)

4. Hardware/ Software Mapping
   - Identification of Nodes
   - Special Purpose Systems
   - Buy vs Build
   - Network Connectivity

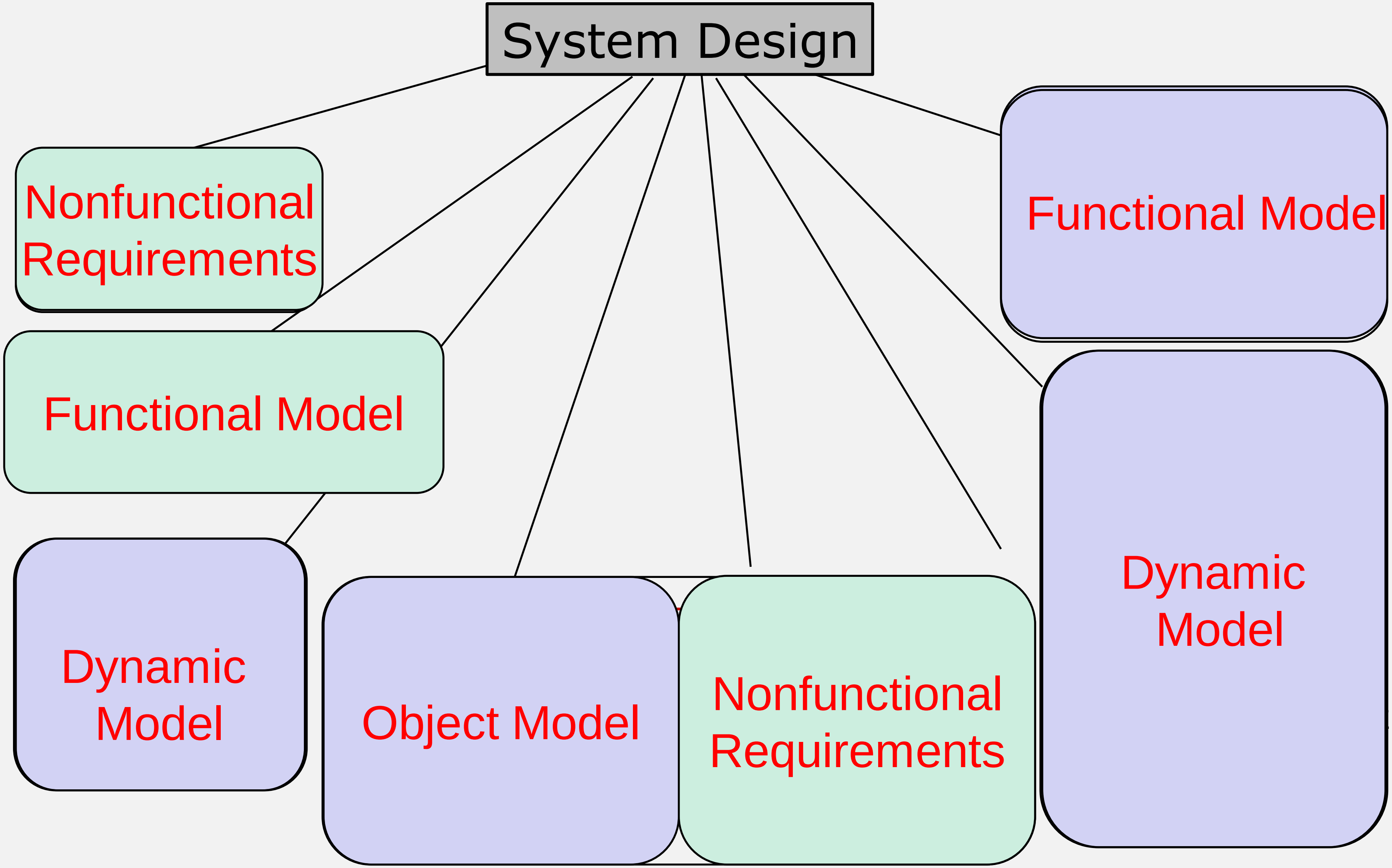
5. Persistent Data Management
   - Storing Persistent Objects
   - Filesystem vs Database

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

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

8. Boundary Conditions
   - Initialization
   - Termination
   - Failure.

Analysis Sources: Requirements and System Model



Concurrency

- 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 is active at a time

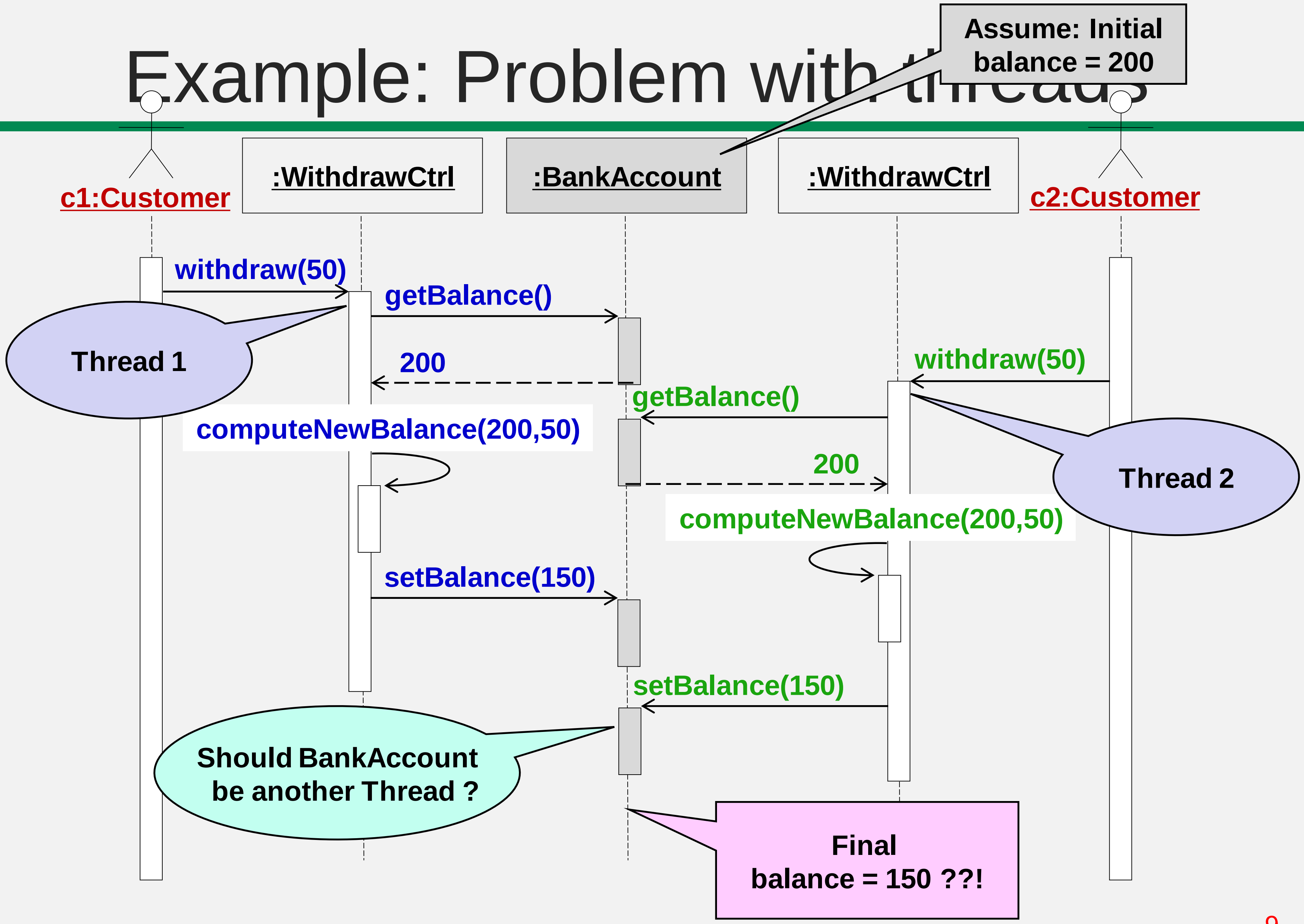
A thread remains within a state diagram until an object sends an event to different object and waits for another event

Thread splitting: Object does a non-blocking send of an event to another object.

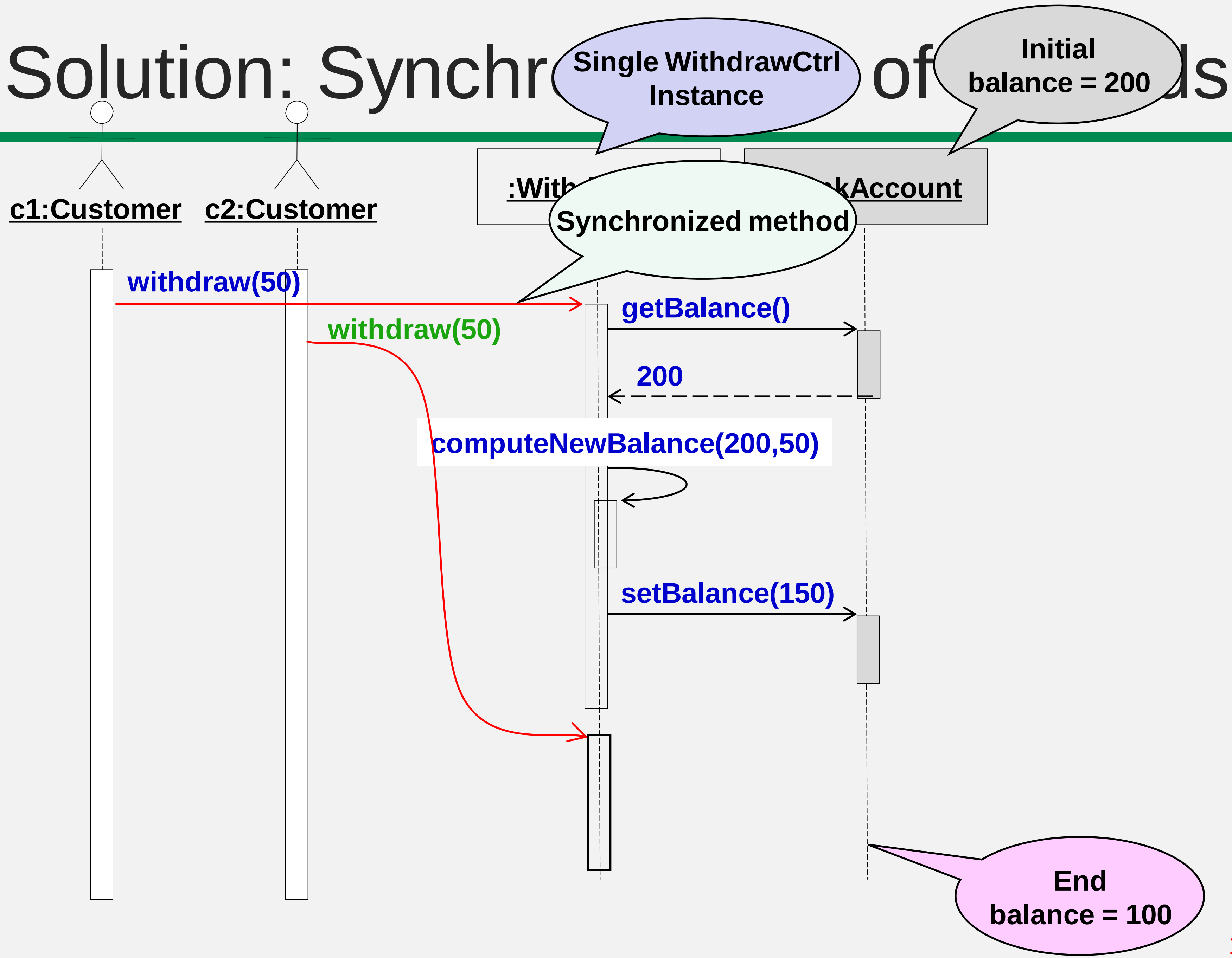
Concurrent threads can lead to race conditions.

A race condition (also race hazard) is a design flaw where the output of a process is depends on the specific sequence of other events.

Example: Problem with threads



Solution: Synchr… of …ds



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 and handled in parallel?

Implementing Concurrency

Concurrent systems can be implemented on any system that provides

1. Physical concurrency: Threads are provided by hardware
2. 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 related to operating systems

4. Hardware Software Mapping

This system design activity addresses two questions:

1. How shall we realize the subsystems: With hardware or with software?
2. 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

- Load too demanding for a single processor?
- distributing objects across several processors possible?
- How many processors are required for steady state load?

Entity Objects -> Memory

- Is there enough memory to buffer bursts of requests?

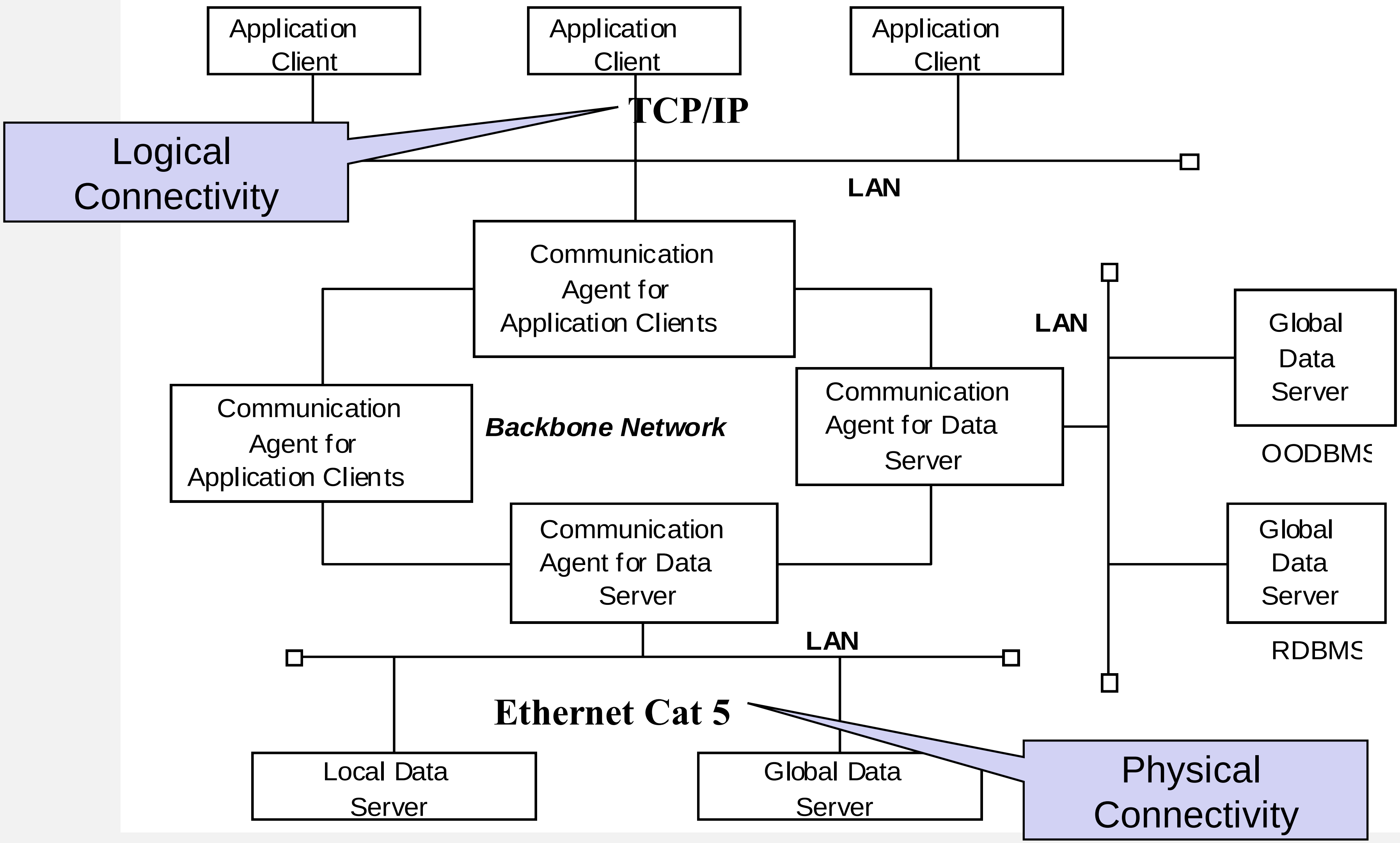
Boundary Objects -> Input/Output Devices

- 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?

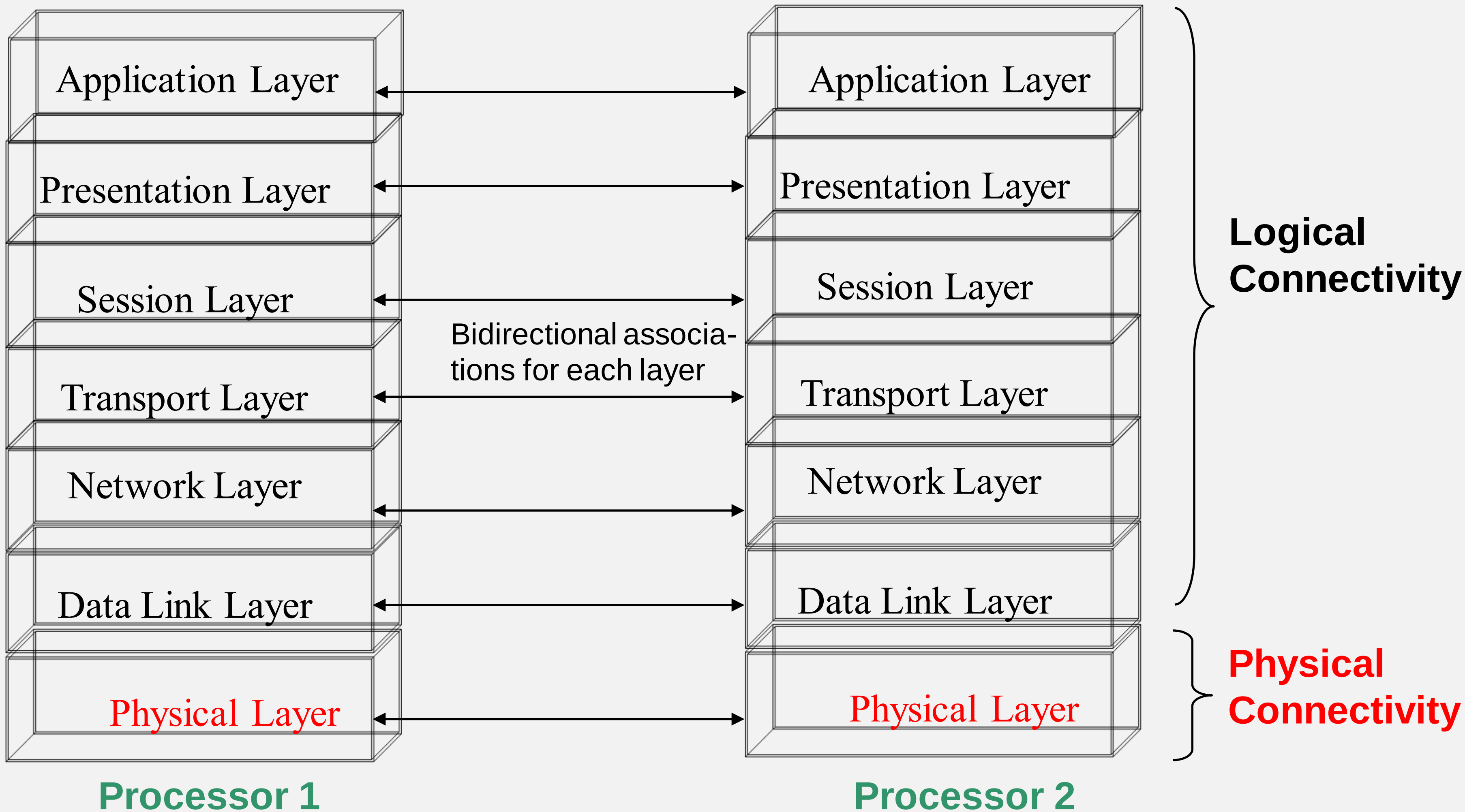
Mapping the Associations: Connectivity

- Describe the physical connectivity
    - 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?

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

    Example: cisco etc

Hardware/Software Mappings in UML

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

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

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.

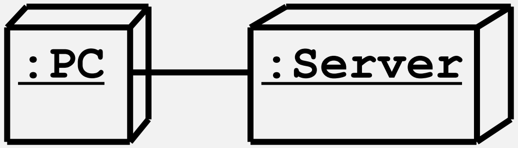
Component Diagram:

- Illustrates dependencies between components at design time, compilation time and runtime

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).

UML Component Diagram

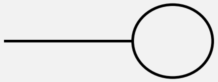
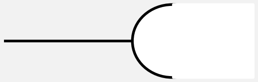
Used to model the top-level view of the system design in terms of components and dependencies among the components. Components can be

- source code, linkable libraries, executables

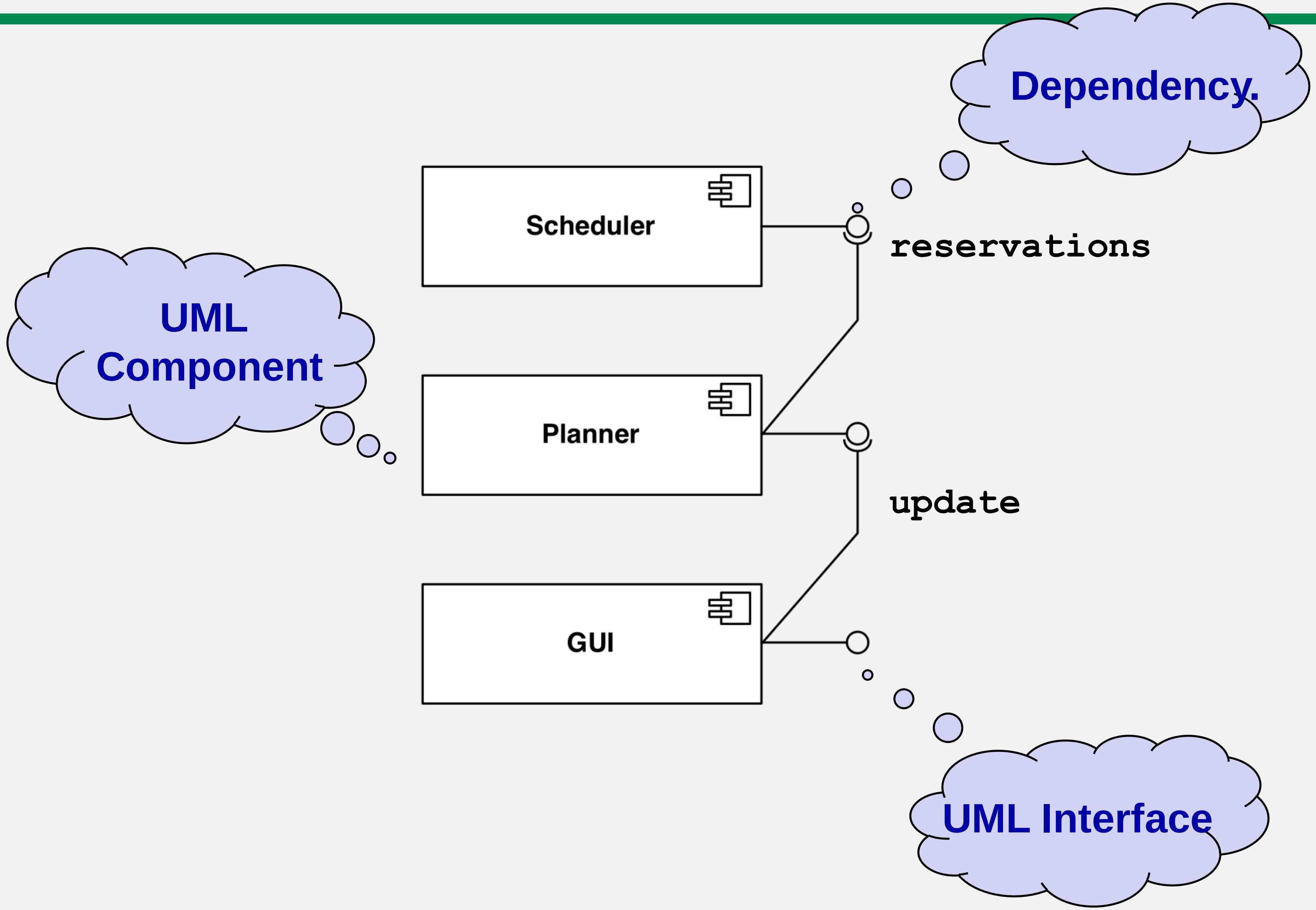
The dependencies (edges in the graph) are shown as dashed lines with arrows from the client component to the supplier component:

- The lines are often also called connectors
- The types of dependencies are implementation language specific

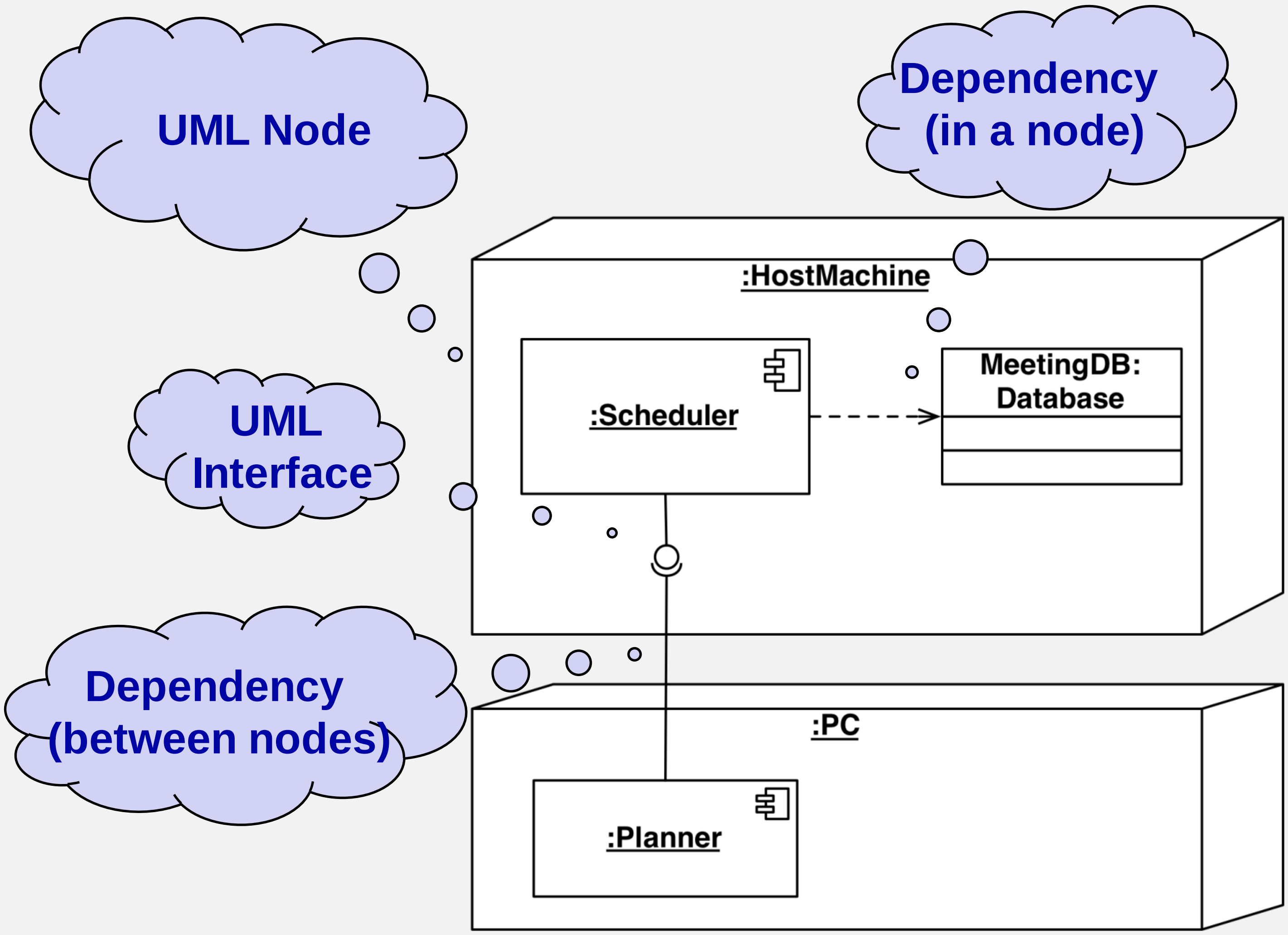
UML Interfaces: Lollipops and Sockets

- 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. 
- A port specifies a distinct interaction point between the component and its environment.
  - Ports are depicted as small squares on the sides of classifiers.

Component Diagram Example



Deployment Diagram Example



6. Global Resource Handling

1. Discusses access control
2. Describes access rights for different classes of actors
3. 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 Example

Global Resource Questions

Does the system need authentication?

If yes, what is the authentication scheme?

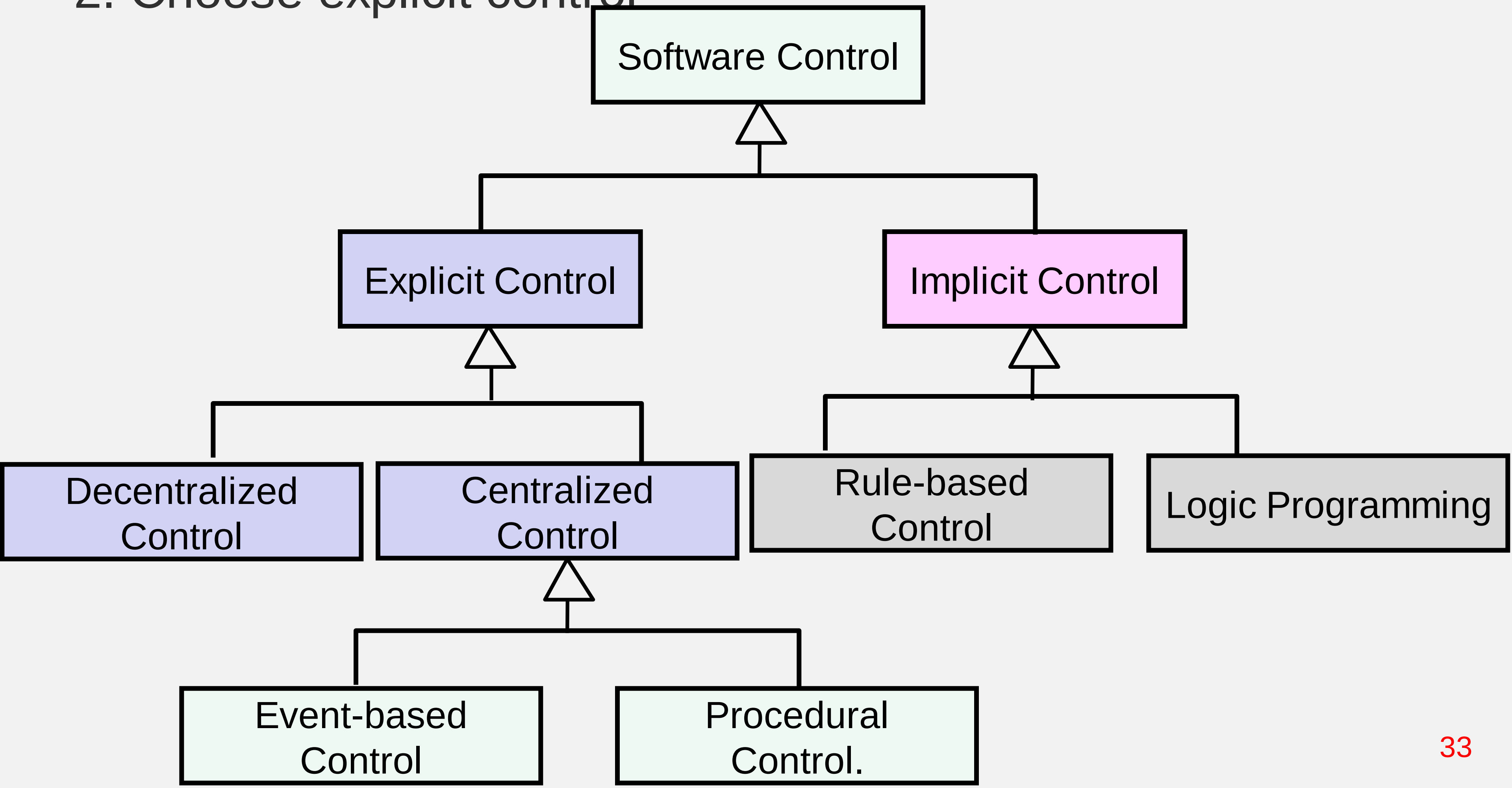
- User name and password?
- Access control list Tickets? Capability-based?

What is the user interface for authentication?

7. Decide on Software Control

Two major design choices:

1. Choose implicit control
2. Choose explicit control



Software 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

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 fore sees 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?

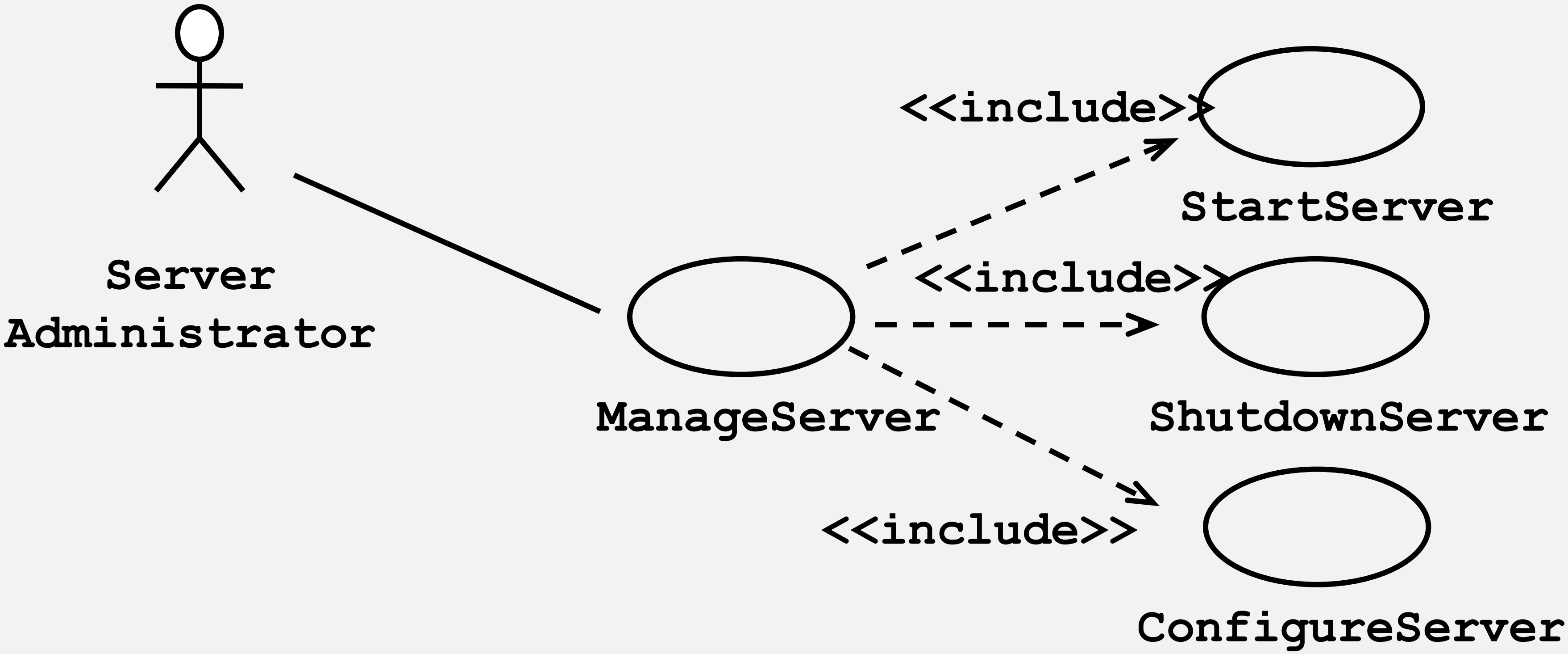
Failure

- 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

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.

Thank You