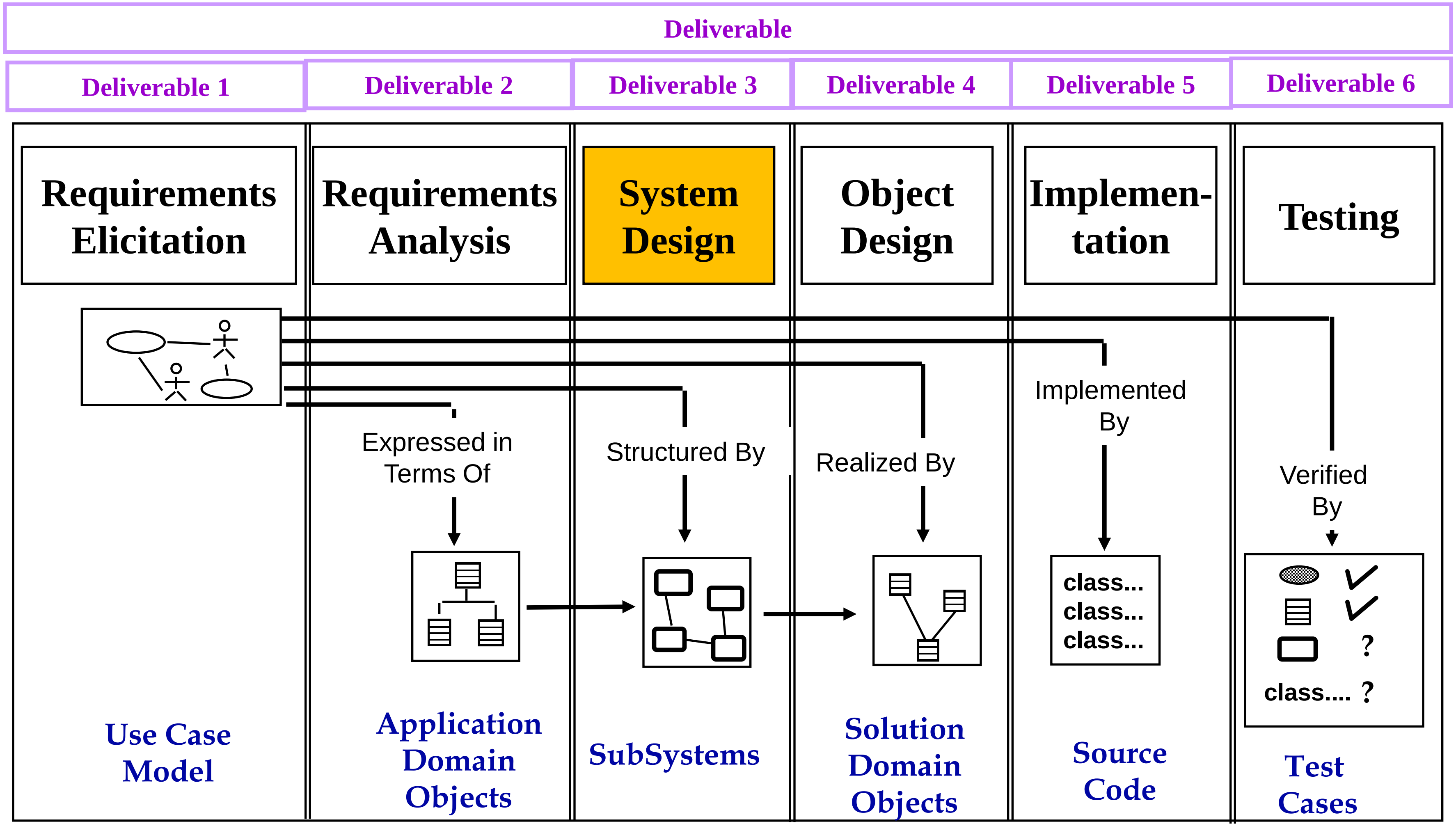
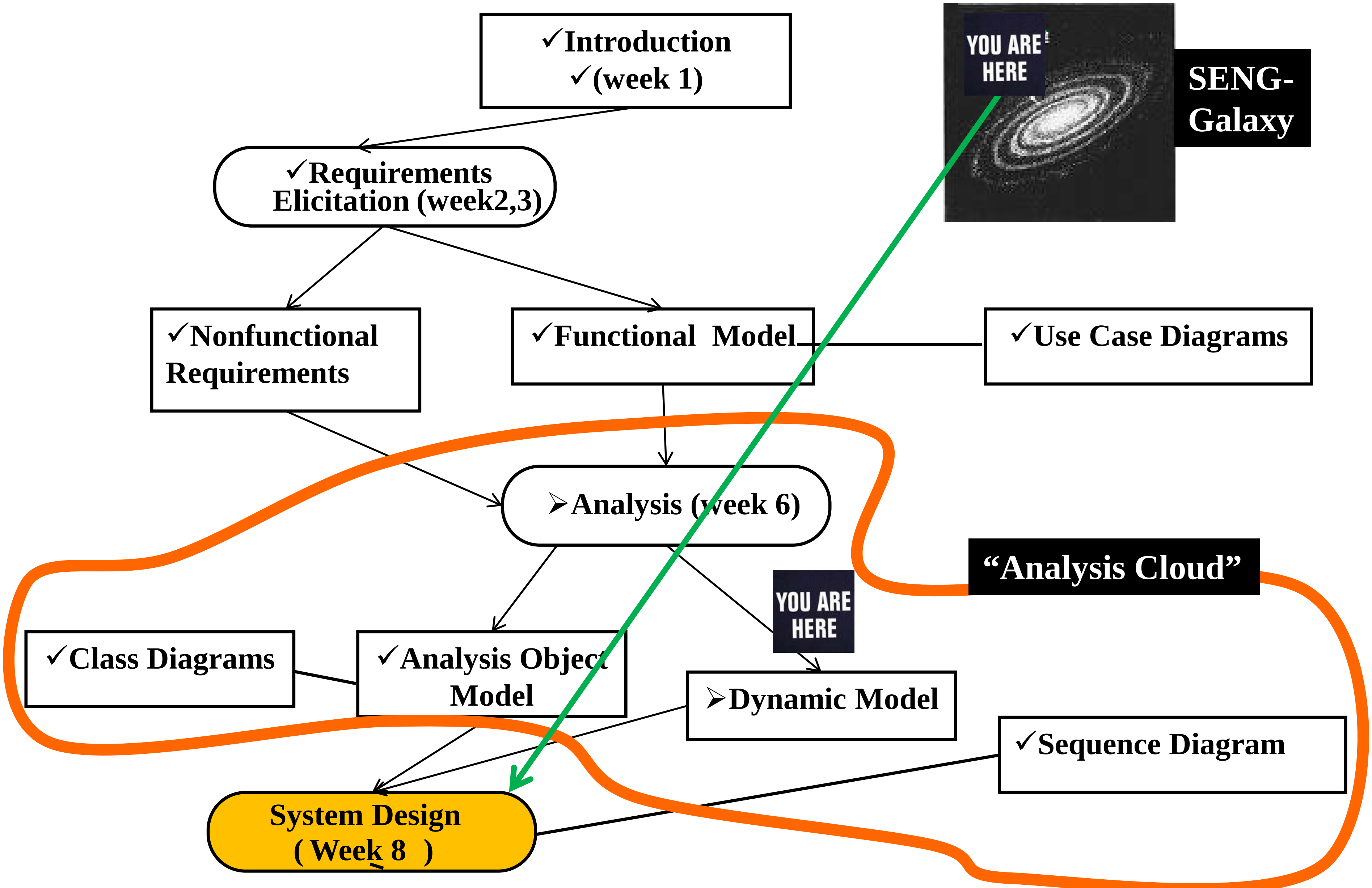
SEENG2130 – Week 8
System Design

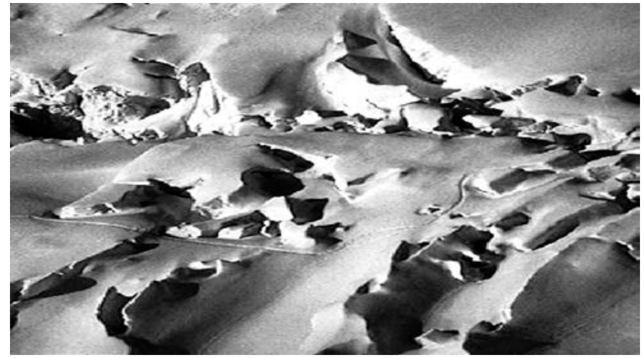
Dr. Joe Ryan
SENG2130 – Systems Analysis and Design
University of Newcastle


Deliverable
Deliverable 1
Deliverable 2
Deliverable 3
Deliverable 4
Deliverable 5
Deliverable 6
Requirements Elicitation
Requirements Analysis
System Design
Object Design
Implemen-tation
Testing
Expressed in Terms Of
Structured By
Realized By
Implemented By
Verified By
class...
class...
class...
class.... ?
?
Use Case Model
Application Domain Objects
SubSystems
Solution Domain Objects
Source Code
Test Cases



THE UNIVERSITY OF NEWCASTLE AUSTRALIA

Ways to Go



System Design
(This week)

8 issues

Design Goals

Subsystem Decomposition

Object design
(Week 9)

Class Diagram

Object Design Model

Source Co...

...entation
(Week ...)

Tes... 10)

Deliverable System

Why is Design so Difficult?

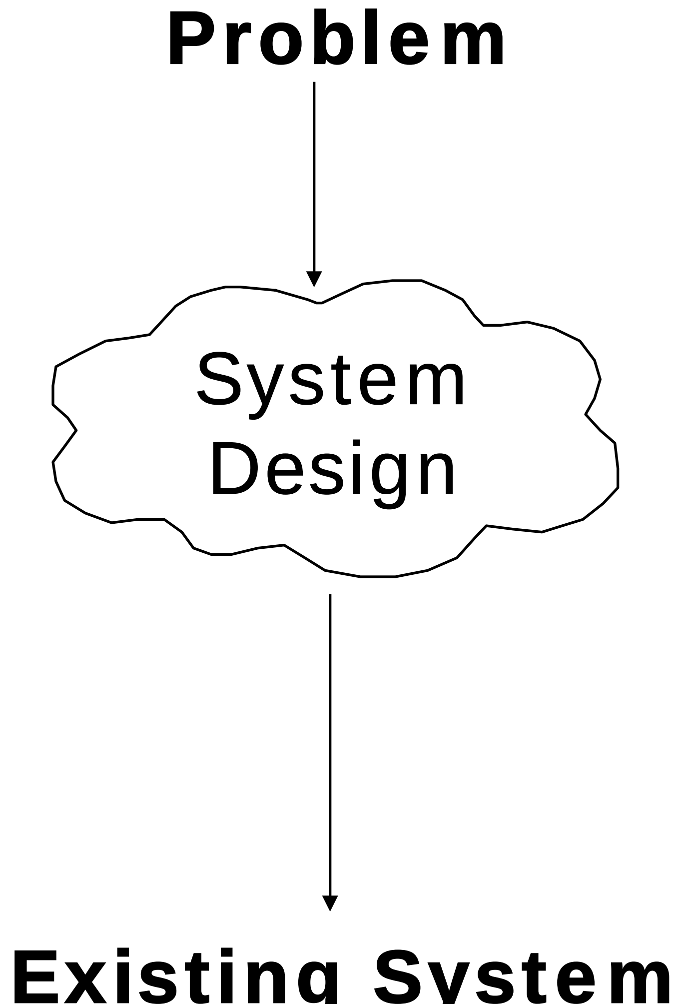
- Analysis: Focuses on the application domain
- Design: Focuses on the solution domain
  - The solution domain is changing very rapidly
    - Halftime knowledge in software engineering
    - Cost of hardware rapidly sinking
  - ➢ Design knowledge is a moving target

- Design window: Time in which design decisions have to be made.

The Scope of System Design

- Bridge the gap
  - between a problem and an existing system in a manageable way

- How?
- Use **Divide & Conquer (8 issues)**:
  1) Identify design goals
  2) Model the new system design as a set of subsystems
  3-8) Address the major design goals.



From Analysis to System Design

Nonfunctional Requirements

1. Design Goals
- Definition
- Trade-offs

Functional Model

2. System Decomposition
- Coherence/Coupling
- Architectural Style

Dynamic Model

3. Concurrency
- Identification of Threads

Object Model

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

5. Data Management
- Persistent Objects
- Filesystem vs Database

Functional Model

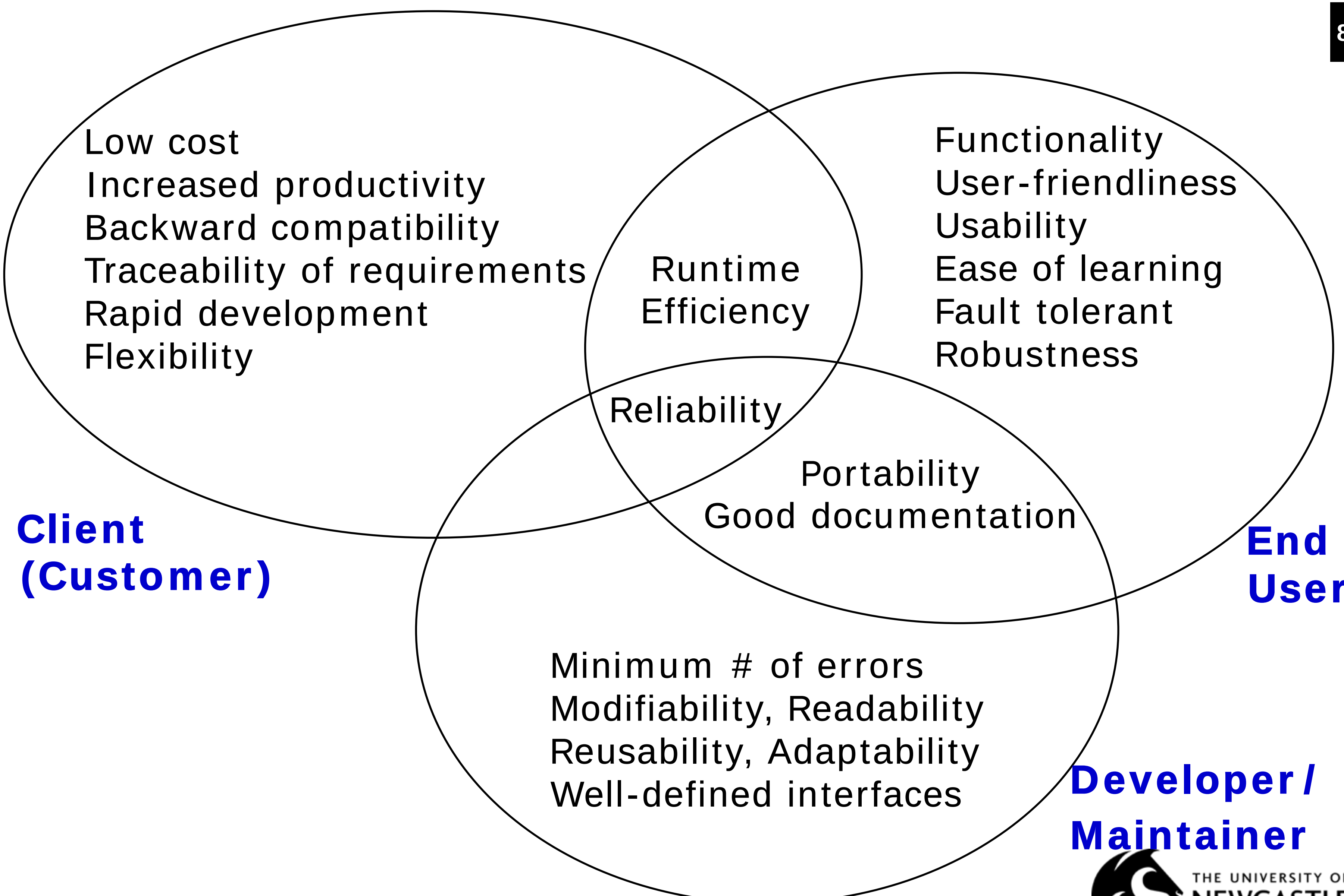
8. Boundary Conditions
- Initialization
- Termination
- Failure

Dynamic Model

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

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

Stakeholders have different Design Goals



1. Design Goals

- Typical Design Trade-offs
  - Functionality v. Usability
  - Cost v. Robustness
  - Efficiency v. Portability
  - Rapid development v. Functionality
  - Cost v. Reusability
  - Backward Compatibility v. Readability

2. System Decomposition

- Coupling and Coherence of Subsystems
  - Goal: Reduce system complexity while allowing change
- Coherence measures dependency among classes
  - High coherence: The classes in the subsystem perform similar tasks and are related to each other via many associations
  - Low coherence: Lots of miscellaneous and auxiliary classes, almost no associations
- Coupling measures dependency among subsystems
  - High coupling: Changes to one subsystem will have high impact on the other subsystem
  - Low coupling: A change in one subsystem does not affect any other subsystem.

2.1 Coupling and Coherence of Subsystems

Good System Design

- Goal: Reduce system complexity while allowing change
- Coherence measures dependency among classes
  - High coherence: The classes in the subsystem perform similar tasks and are related to each other via many associations
  - Low coherence: Lots of miscellaneous and auxiliary classes, almost no associations
- Coupling measures dependency among subsystems
  - High coupling: Changes to one subsystem will have high impact on the other subsystem
  - Low coupling: A change in one subsystem does not affect any other subsystem

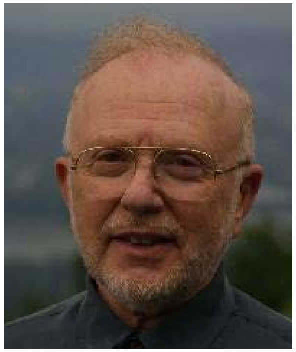
How to achieve high Coherence

- High coherence can be achieved if most of the interaction is within subsystems, rather than across subsystem boundaries
- Questions to ask:
  - Does one subsystem always call another one for a specific service?
    - Yes: Consider moving them together into the same subystem.
  - Which of the subsystems call each other for services?
    - Can this be avoided by restructuring the subsystems or changing the subsystem interface?
  - Can the subsystems even be hierarchically ordered (in layers)?

How to achieve Low Coupling

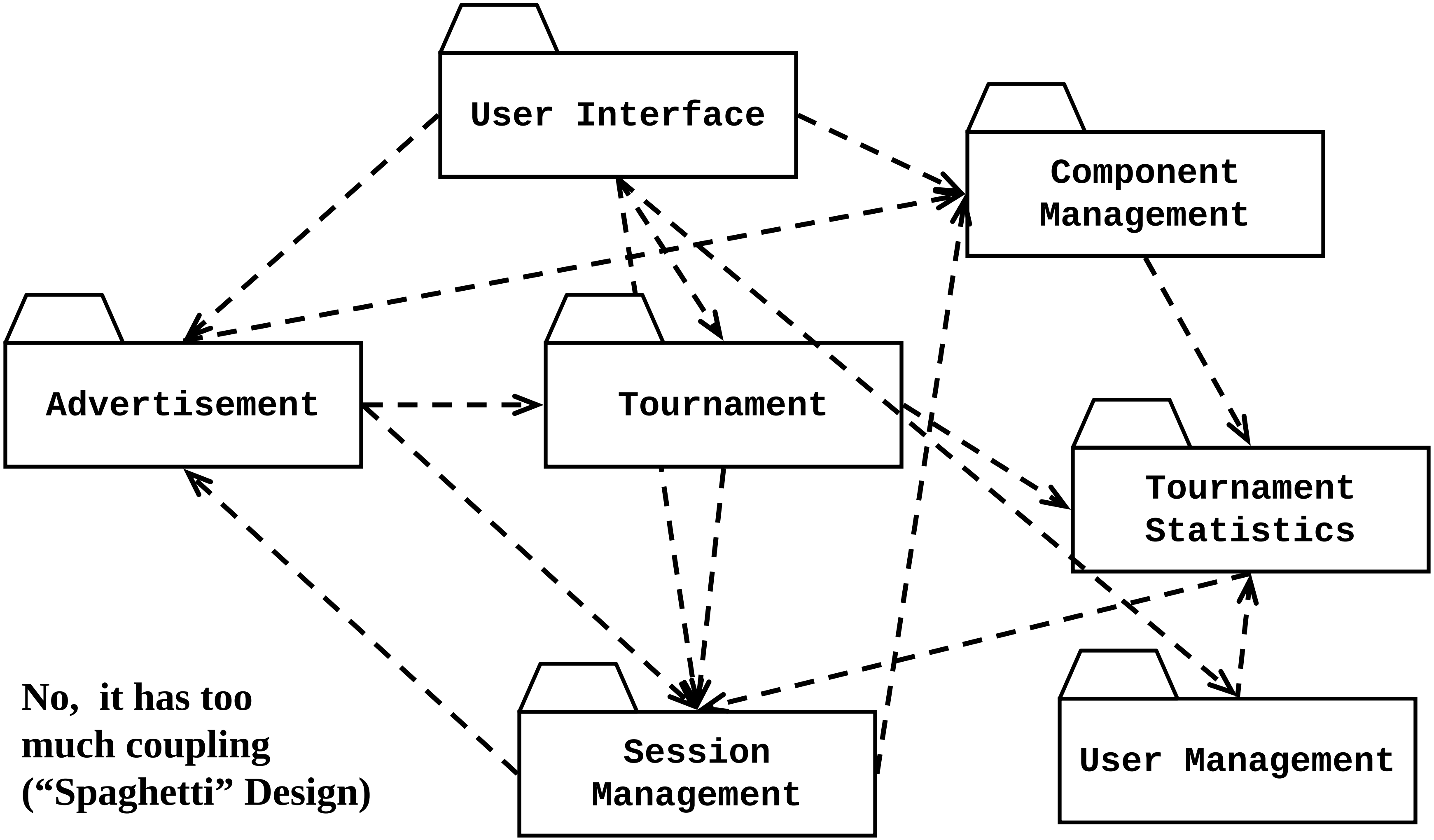
- Low coupling can be achieved if a calling class does not need to know anything about the internals of the called class (Principle of information hiding, Parnas)
- Questions to ask:
  - Does the calling class really have to know any attributes of classes in the lower layers?
  - Is it possible that the calling class calls only operations of the lower level classes?

David Parnas, *1941,
Developed the concept of
modularity in design.



THE UNIVERSITY OF NEWCASTLE AUSTRALIA

Is this a Good Design?



No, it has too much coupling ("Spaghetti" Design)

2.2 Architectural Style vs Architecture

- Subsystem decomposition: Identification of subsystems, services, and their relationship to each other
- Architectural Style: A pattern for a subsystem decomposition
- Software Architecture: Instance of an architectural style.

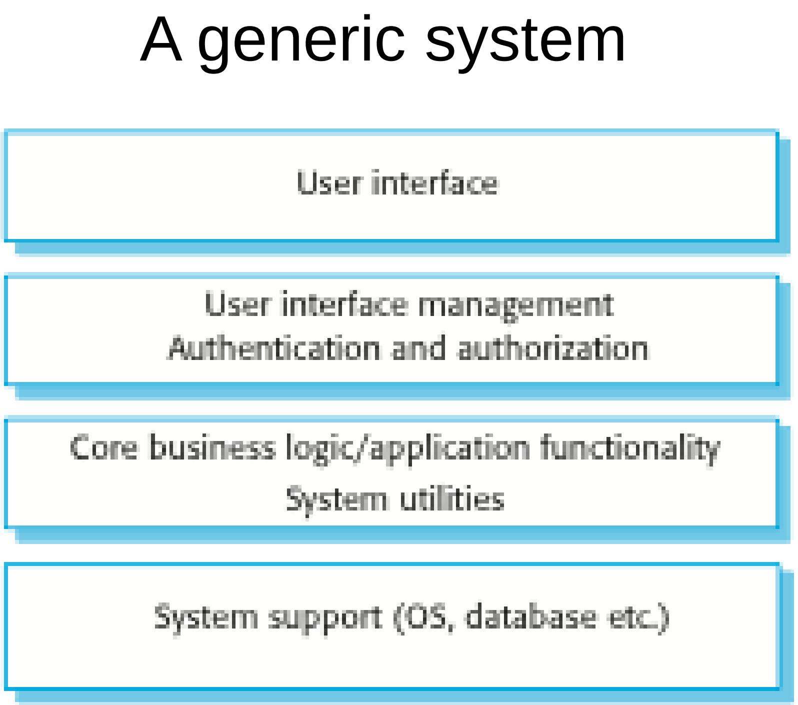
Examples of Architectural Styles

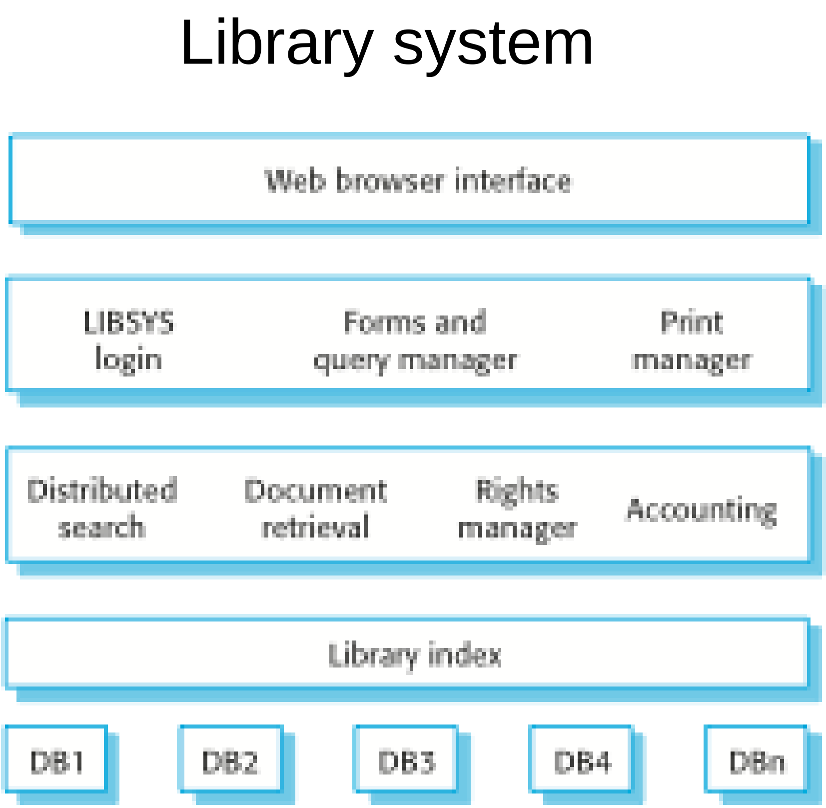
- Layered Architectural style
- Client/Server
- Peer-To-Peer
- Repository
- Blackboard
- Model-View-Controller
- Pipes and Filters

Layered Architectural Style

- A layer is a subsystem that provides a service to another subsystem with the following restrictions:
  – A layer only depends on services from lower layers
  – A layer has no knowledge of higher layers
- A layer can be divided horizontally into several independent subsystems called partitions
  – Partitions provide services to other partitions on the same layer
  – Partitions are also called "weakly coupled" subsystems.

Layered Architectural Style

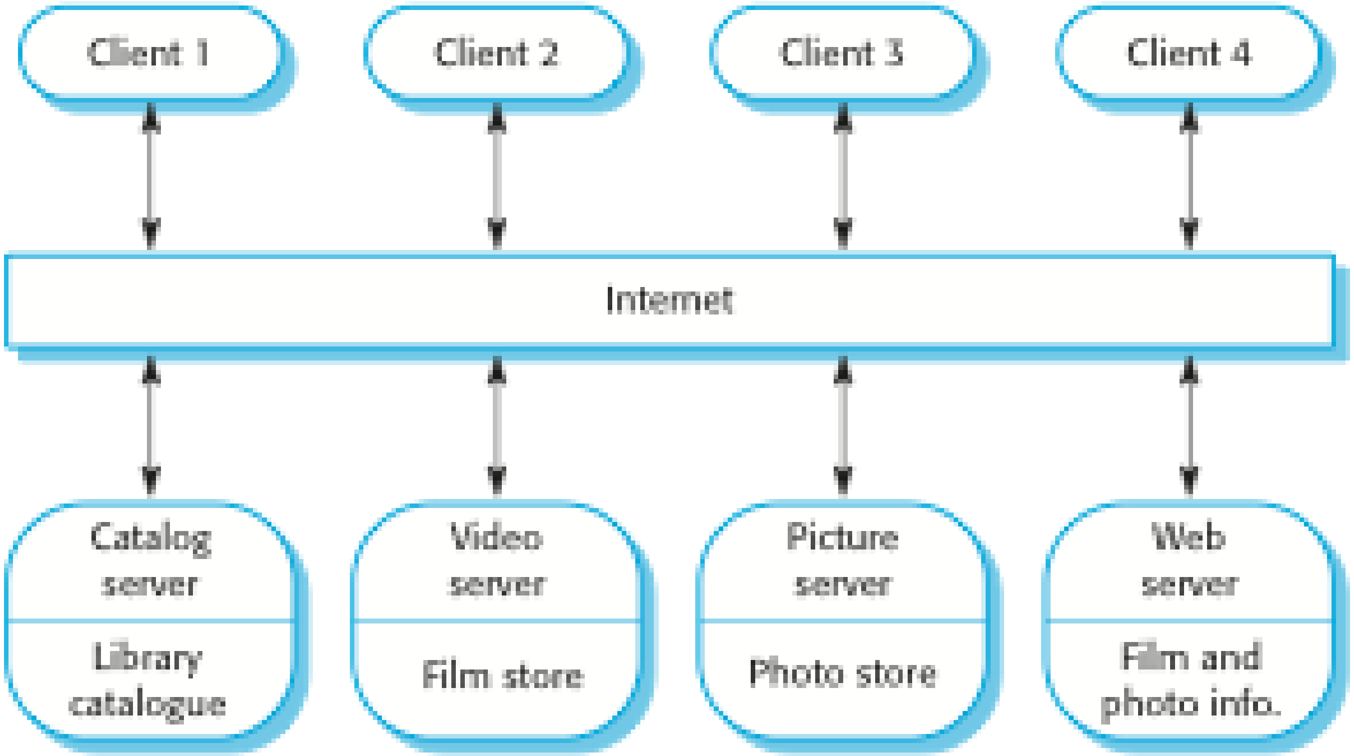




Client/Server Architectures

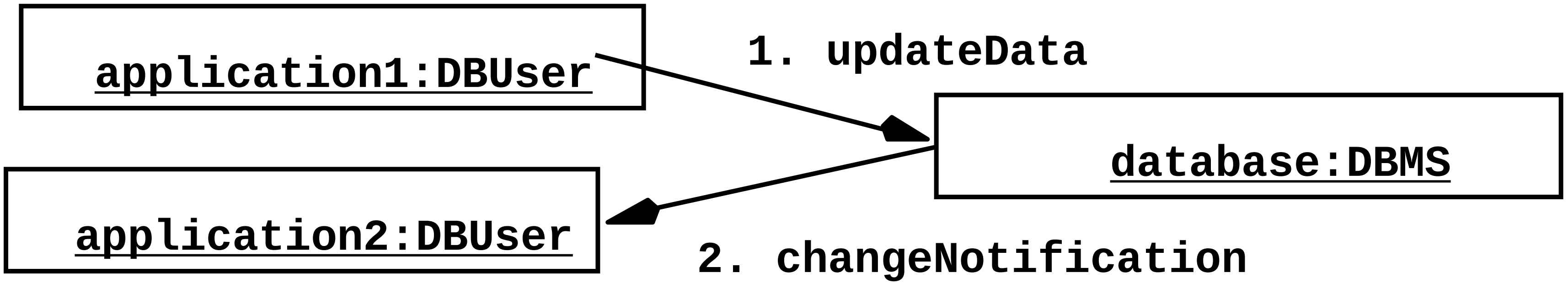
- Often used in the design of database systems
  - Front-end: User application (client)
  - Back end: Database access and manipulation (server)
- Functions performed by client:
  - Input from the user (Customized user interface)
  - Front-end processing of input data
- Functions performed by the database server:
  - Centralized data management
  - Data integrity and database consistency
  - Database security

Client/Server Architectures for a film library



Client/Server Architectures - problems

- Client/Server systems do not provide peer-to-peer communication
- Peer-to-peer communication is often needed
- Example:
  - Database must process queries from application and should be able to send notifications to the application when data have changed



Peer-to-Peer Architectural Style

Generalization of Client/Server Architectural Style

“Clients can be servers and servers can be clients”

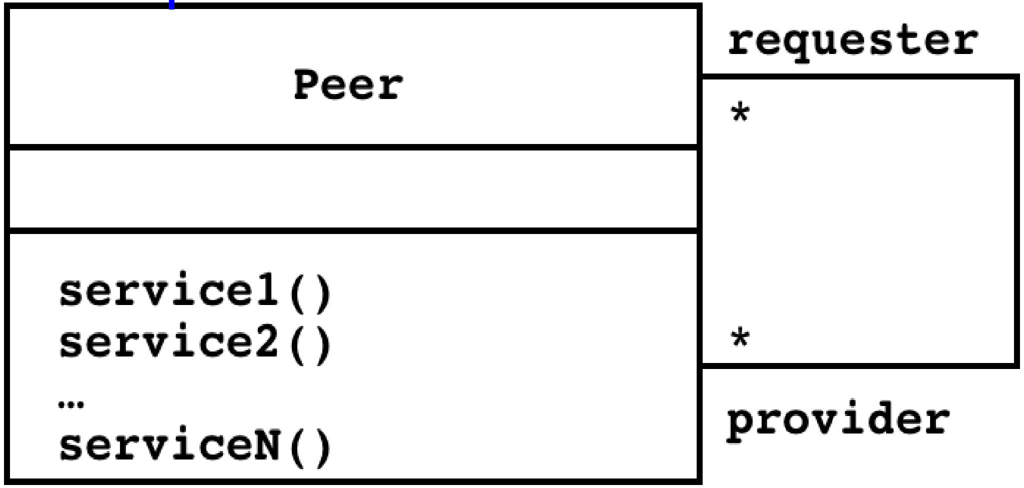
Introduction a new abstraction: Peer

“Clients and servers can be both peers”

How do we model this statement? With Inheritance?

Proposal 1: “A peer can be either a client or a server”

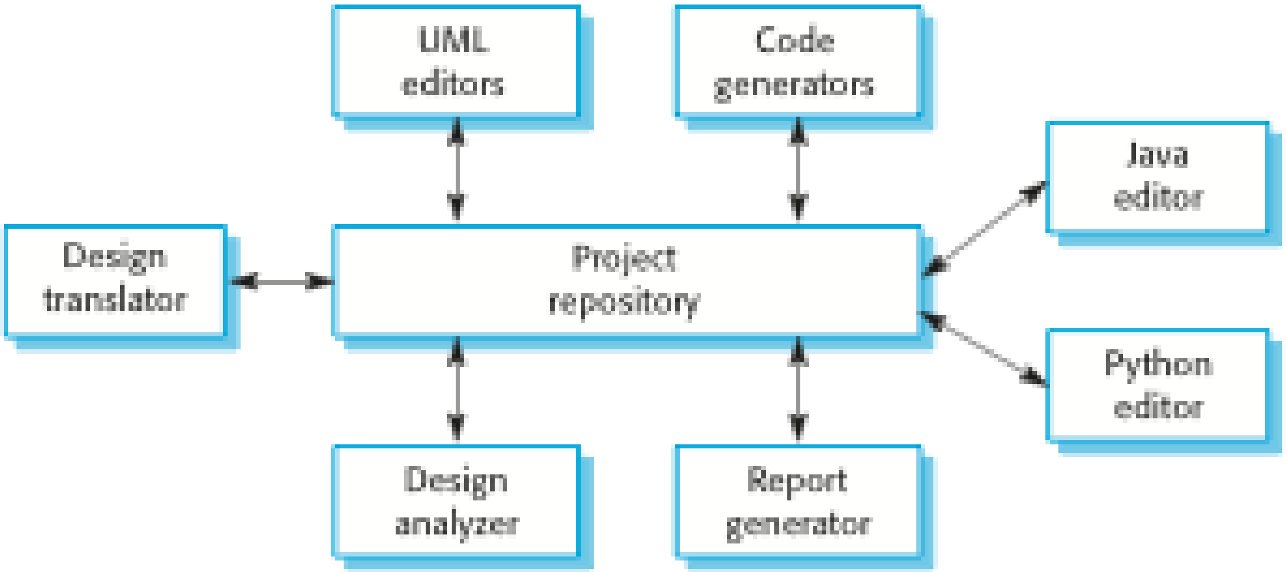
Proposal 2: “A peer can be a client as well as a server”.



Repository Architectural Style

- The basic idea behind this architectural style is to support a collection of independent programs that work cooperatively on a common data structure called the repository
- Subsystems access and modify data from the repository. The subsystems are loosely coupled (they interact only through the repository).
- When large amounts of data are to be shared, the repository model of sharing is most commonly used as this is an efficient data sharing mechanism.

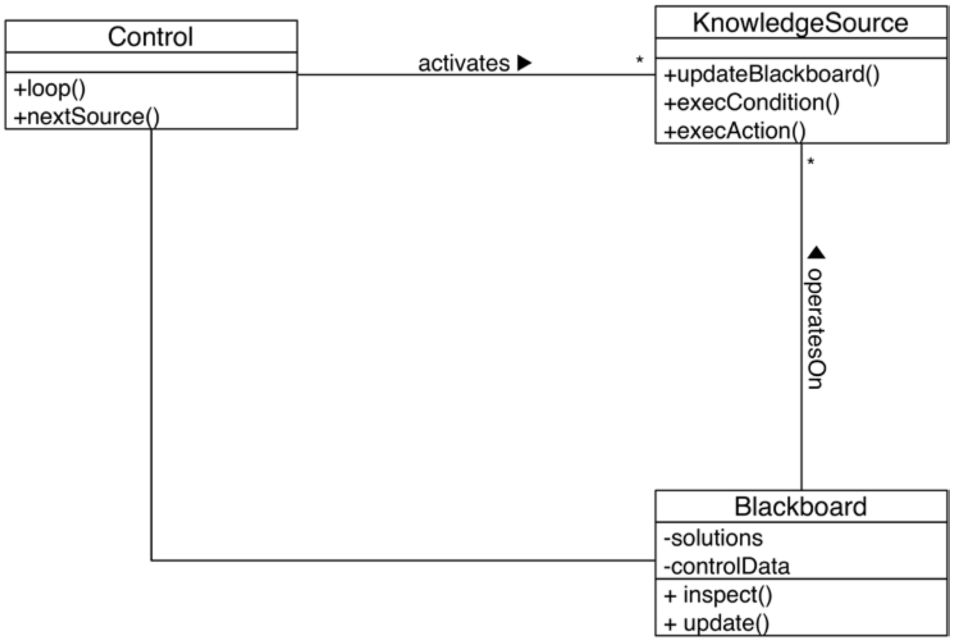
Repository Architectural Style for an IDE (Incremental Development Environment)



From Repository to Blackboard

- The repository architectural style does not specify any control
  - The control flow is dictated by the repository through triggers or by the subsystems through locks and synchronization primitives
- In the blackboard architectural style, we can model the controller more explicitly
  - This style is used for solving problems for which an algorithmic solution does not (yet) exist

Blackboard Architectural Style



Synonyms:

Control: Supervisor
Knowledge Source: Specialist, Expert
Blackboard: Knowledge Sharing Area.

- The blackboard is the repository for the problem to be solved, partial solutions and new information
- The knowledge sources read anything that is placed on the black-board and place newly generated information on it
- Control governs the flow of problemsolving activity in the system, in particular how the knowledge sources get notified of any new information put on the blackboard.

Model-View-Controller Architectural Style

- Problem: In systems with high coupling changes to the user interface (boundary objects) often force changes to the entity objects (data)
  - The user interface cannot be reimplemented without changing the representation of the entity objects
  - The entity objects cannot be reorganized without changing the user interface
- Solution: Decoupling! The model-view-controller (MVC) style decouples data access (entity objects) and data presentation (boundary objects)
    - Views: Subsystems containing boundary objects
    - Model: Subsystem with entity objects
    - Controller: Subsystem mediating between Views (data presentation) and Models (data access).

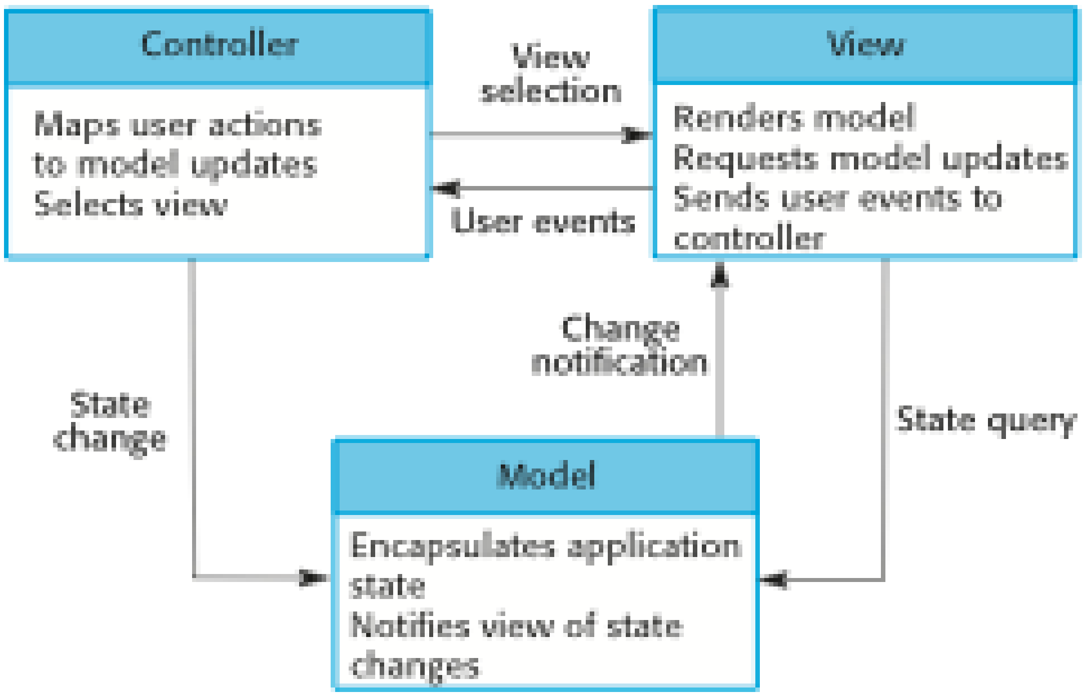
Model-View-Controller Architectural Style

- Subsystems are classified into 3 different types

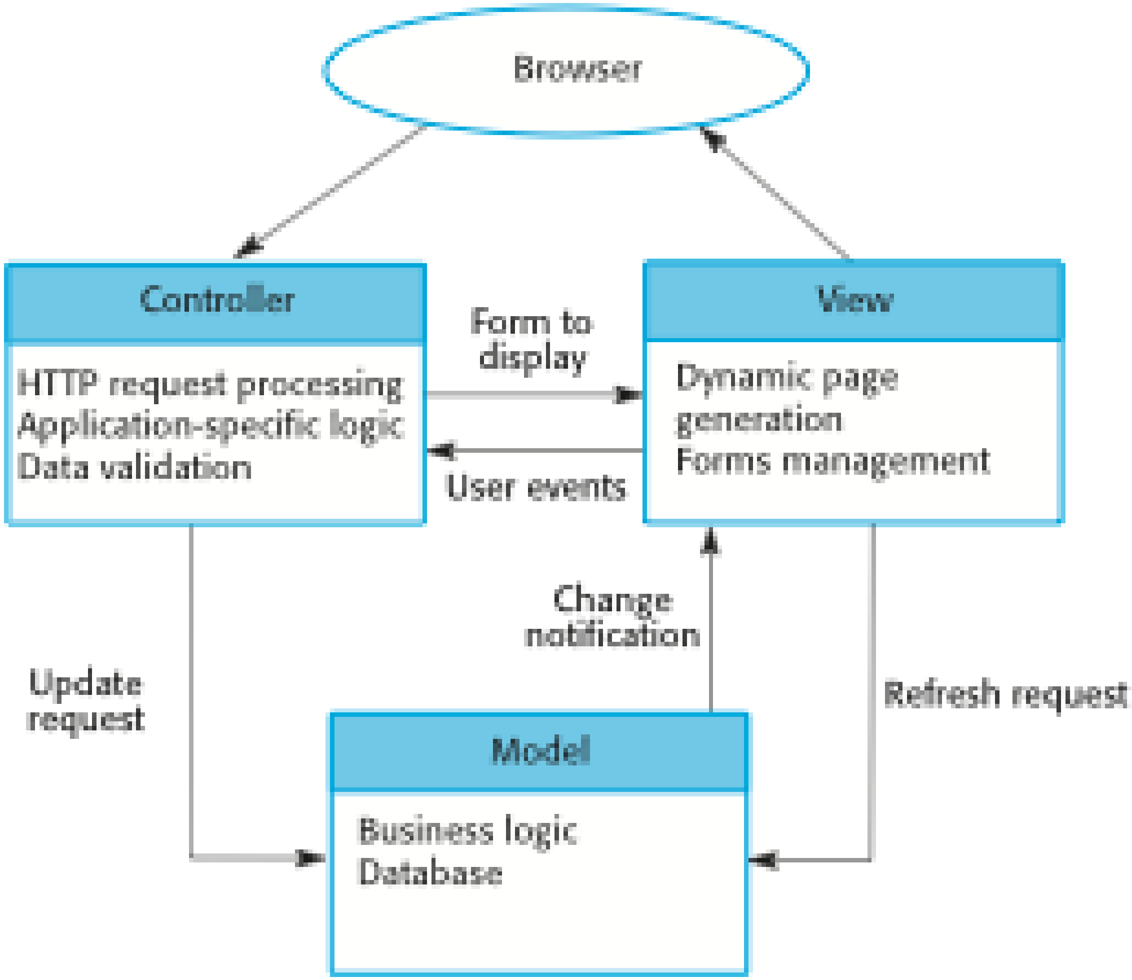
  Model subsystem: Responsible for application domain knowledge

  View subsystem: Responsible for displaying information to the user

  Controller subsystem: Responsible for interacting with the user and notifying views of changes in the model



Web application architecture using the MVC pattern



THE UNIVERSITY OF NEWCASTLE AUSTRALIA

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

3. Concurrency

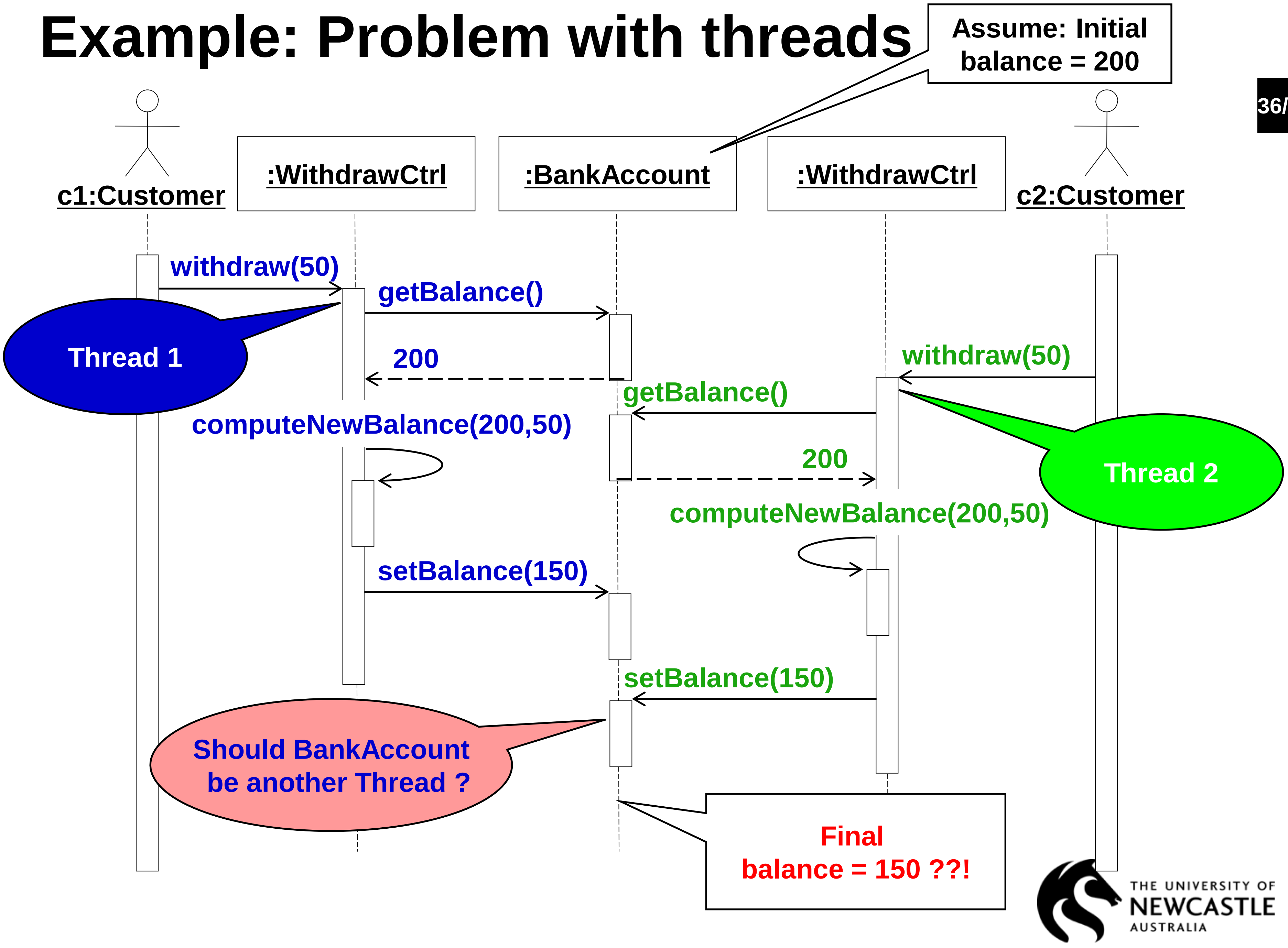
- Nonfunctional Requirements to be addressed: Performance, Response time, latency, availability.
- Two objects are inherently concurrent if they can receive events at the same time without interacting
  - Source for identification: Objects in a sequence diagram that can simultaneously receive events
    - 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

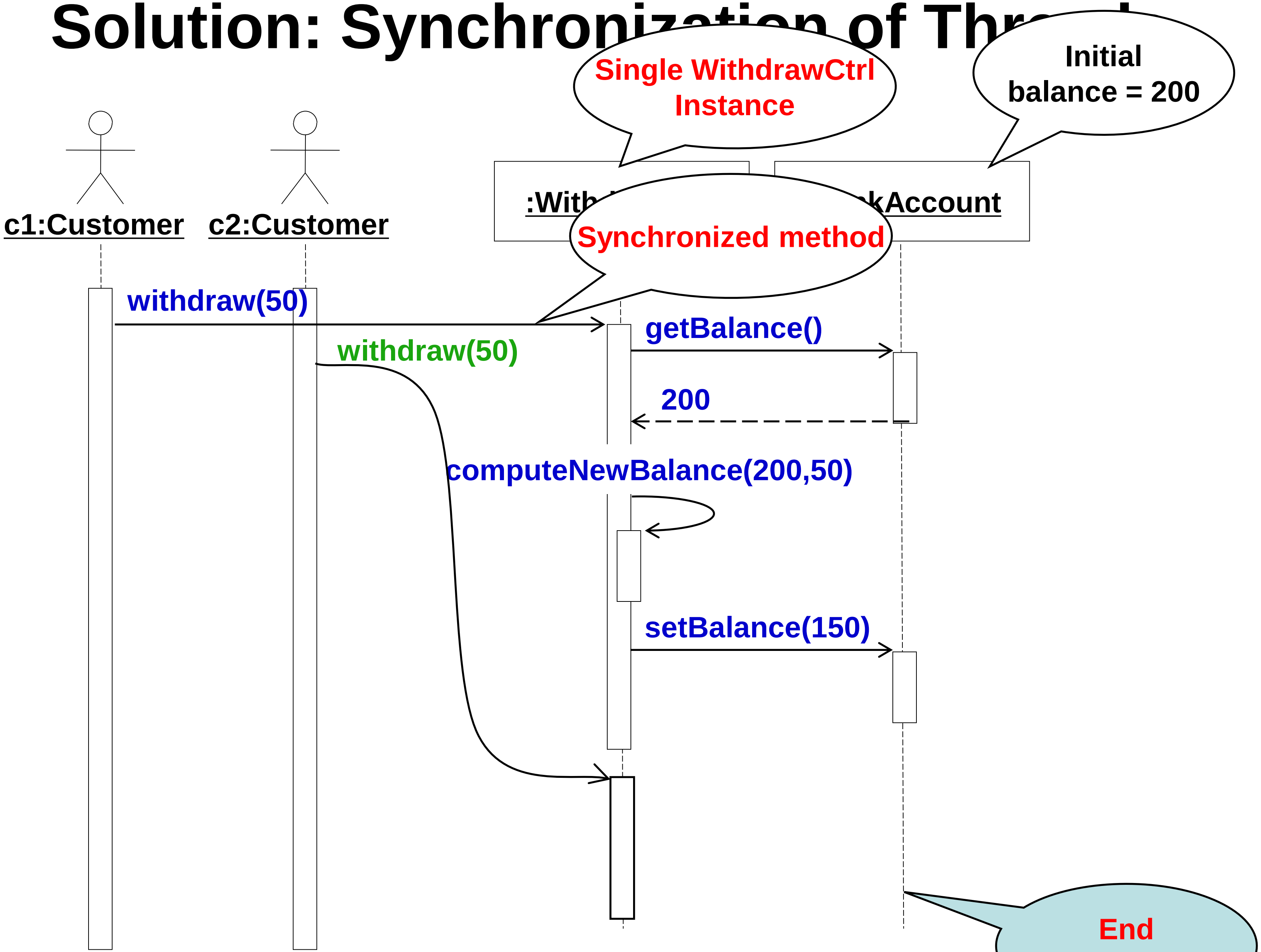
- A thread of control is a path through a set of sequence diagrams on which a single object is active at a time
  - A thread remains within a sequence diagram until an object sends an event to different object and waits for another event
  - Thread splitting: Object does a non-blocking send of an event 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.
  - The name originated in digital circuit design: Two signals racing each other to influence the output.

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

Example: Problem with threads



Solution: Synchronization of Threads



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? (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

- 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 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 shall we realize the subsystems: With hardware or with software?
  - How do we map the object model onto the chosen hardware and/or software?

    4.1 Mapping the Objects:
      - Processor, Memory, Input/Output

    4.2 Mapping the Associations:
      - Network connections

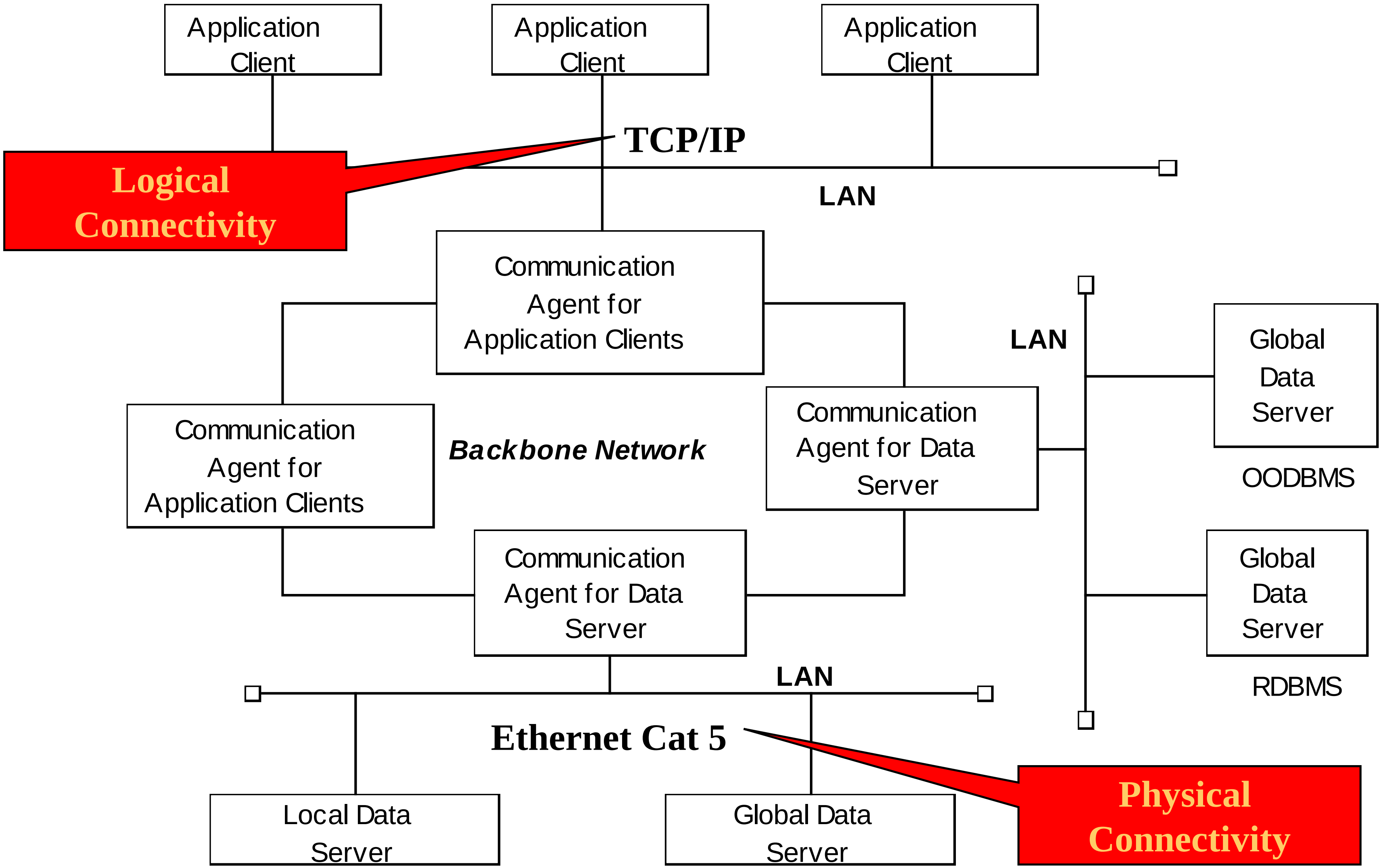
4.1 Mapping Objects onto Hardware

- 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?

4.2 Mapping the Associations: Connectivity

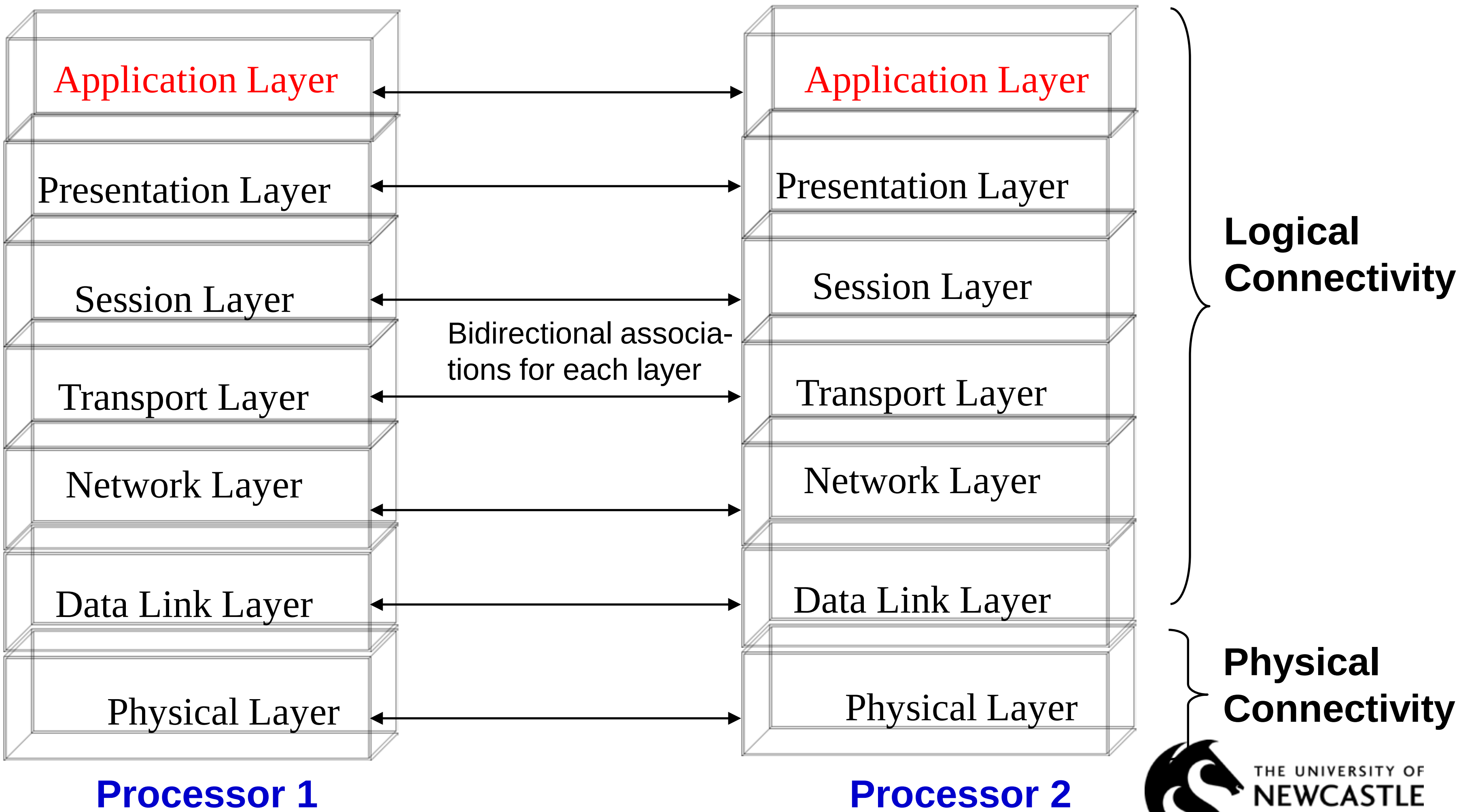
- Describe the physical connectivity
  - (“Physical layer in the OSI reference model”)
    - Describes which associations in the object model are mapped to physical connections
- Describe the logical connectivity (subsystem associations)
  - Associations that do not directly map into physical connections
  - 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



THE UNIVERSITY OF NEWCASTLE AUSTRALIA

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 a tabbed rectangle symbol inside
- 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

- 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

UML Diagram

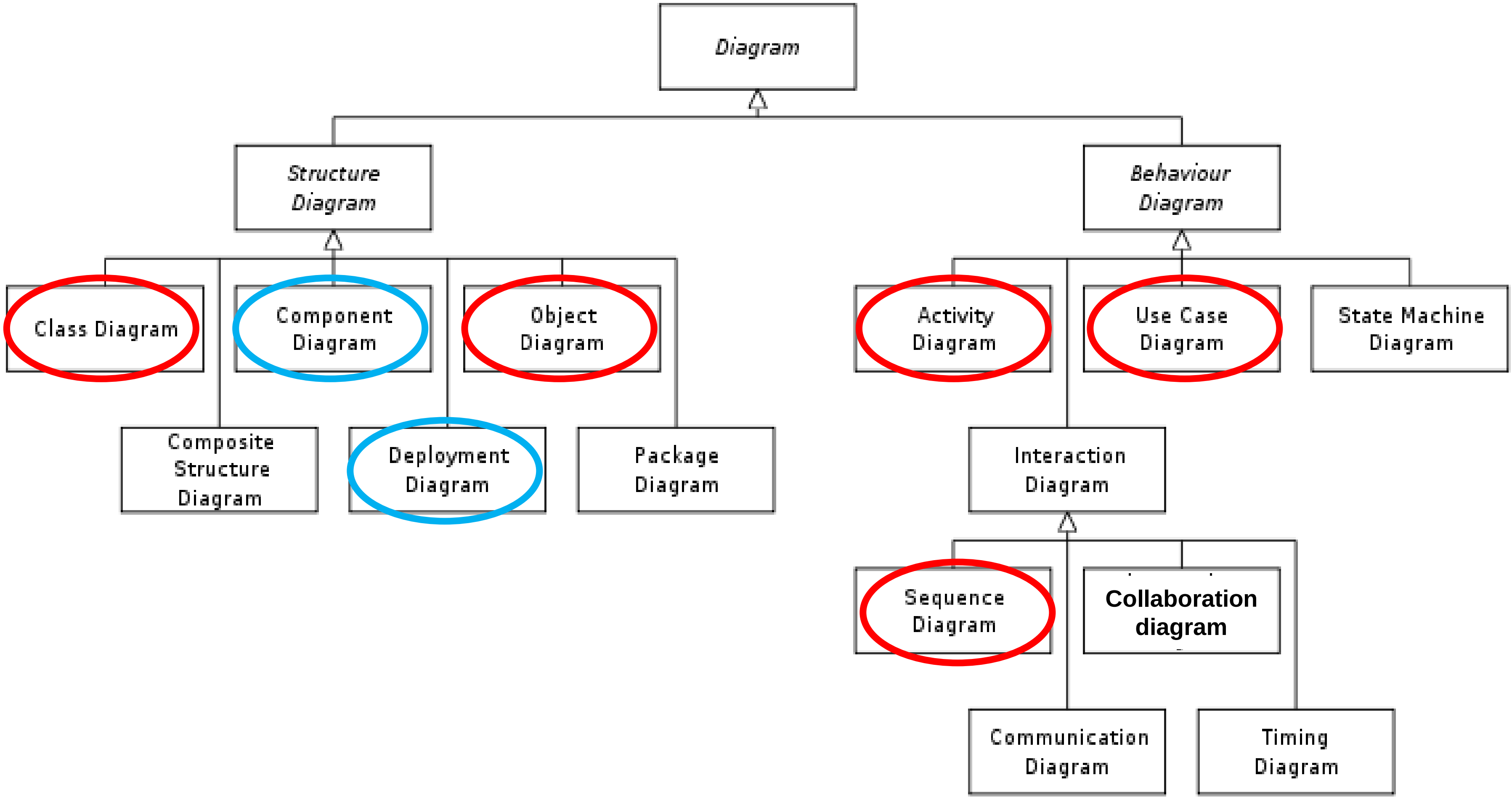
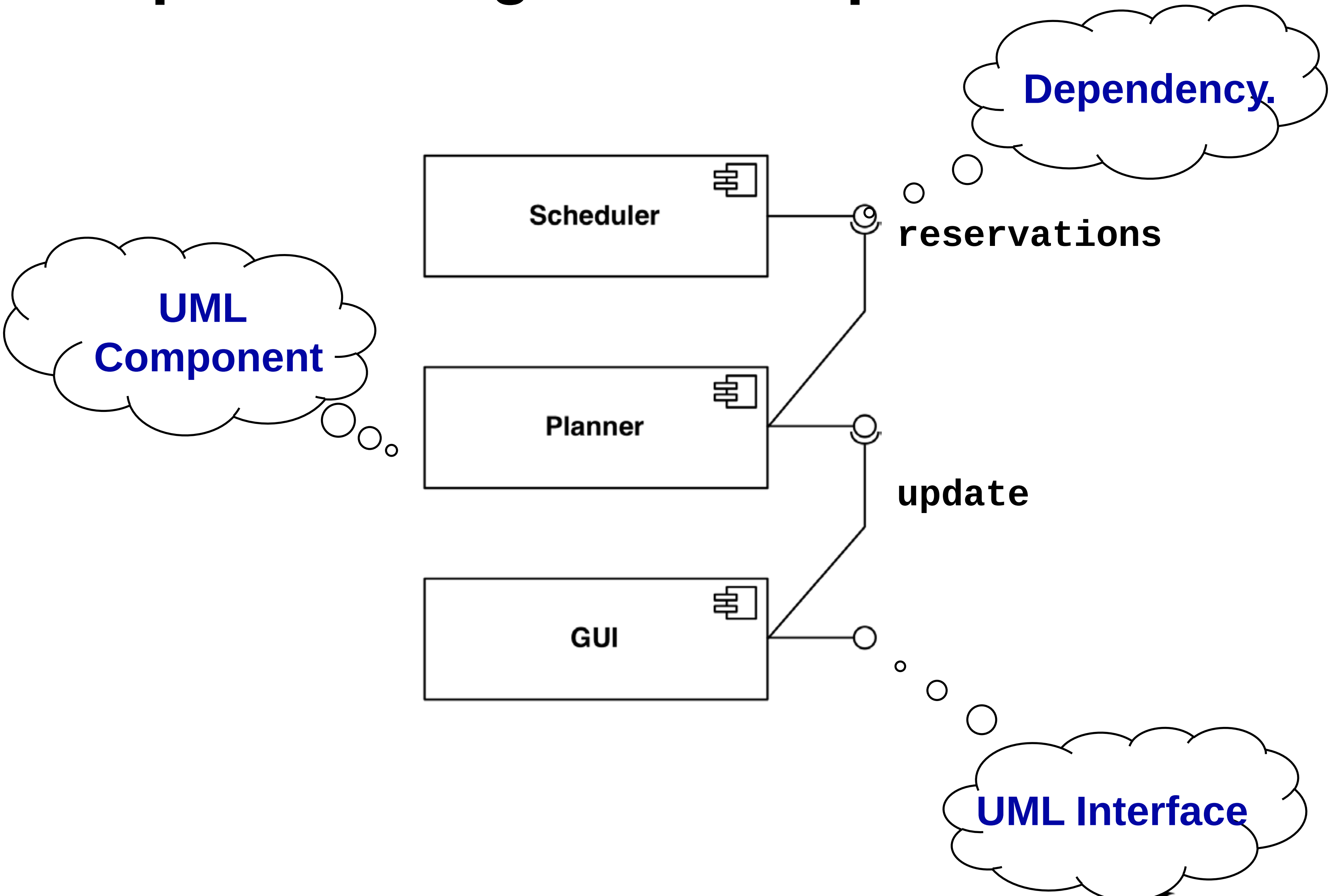
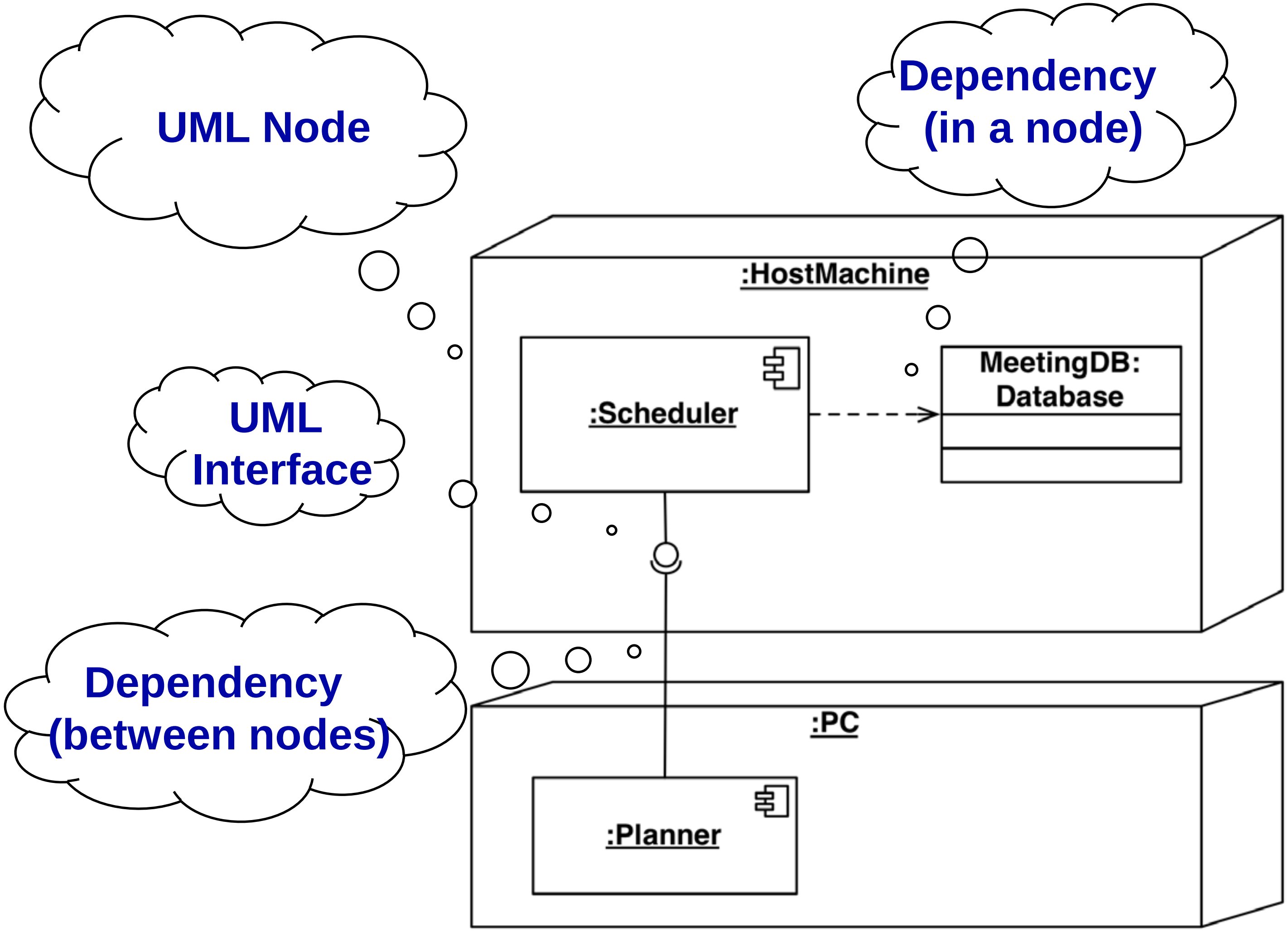


Diagram
Structure Diagram
Behaviour Diagram
Class Diagram
Component Diagram
Object Diagram
Composite Structure Diagram
Deployment Diagram
Package Diagram
Activity Diagram
Use Case Diagram
State Machine Diagram
Interaction Diagram
Sequence Diagram
Collaboration diagram
Communication Diagram
Timing Diagram

Component Diagram Example



Deployment Diagram Example



THE UNIVERSITY OF NEWCASTLE AUSTRALIA

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

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()

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

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()

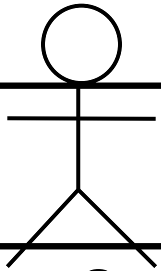
.

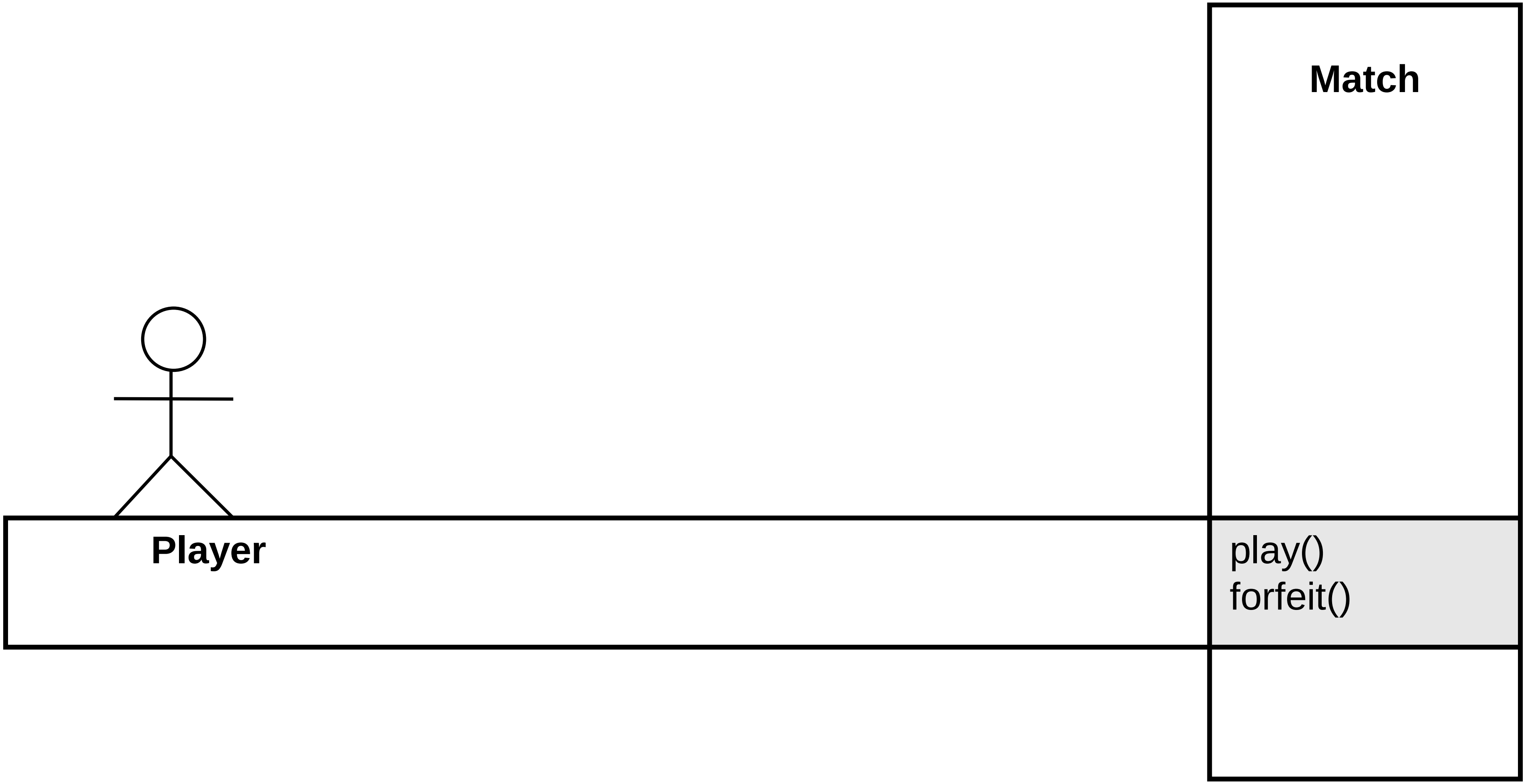
Better Access Matrix Implementations

- 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

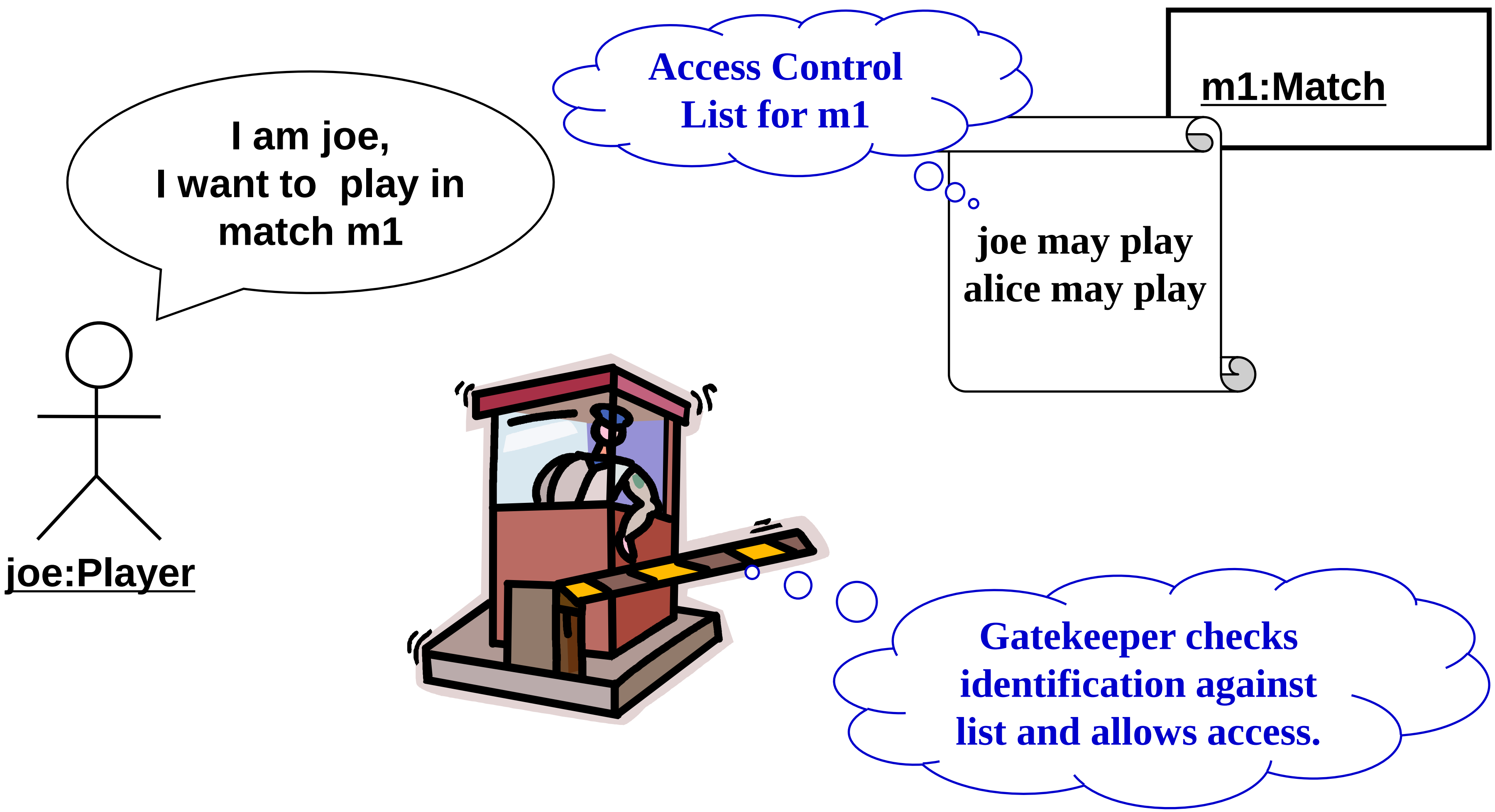
	Arena	**League**	**Tournament**	**Match**
Operator	<<create>> createUser() view ()	<<create>> archive()		
LeagueOwner	view ()	edit ()	<<create>> archive() schedule() view()	<<create>> end()
Player	view() applyForOwner()	view() subscribe()	applyFor() view()	play() forfeit()
Spectator	view() applyForPlayer()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



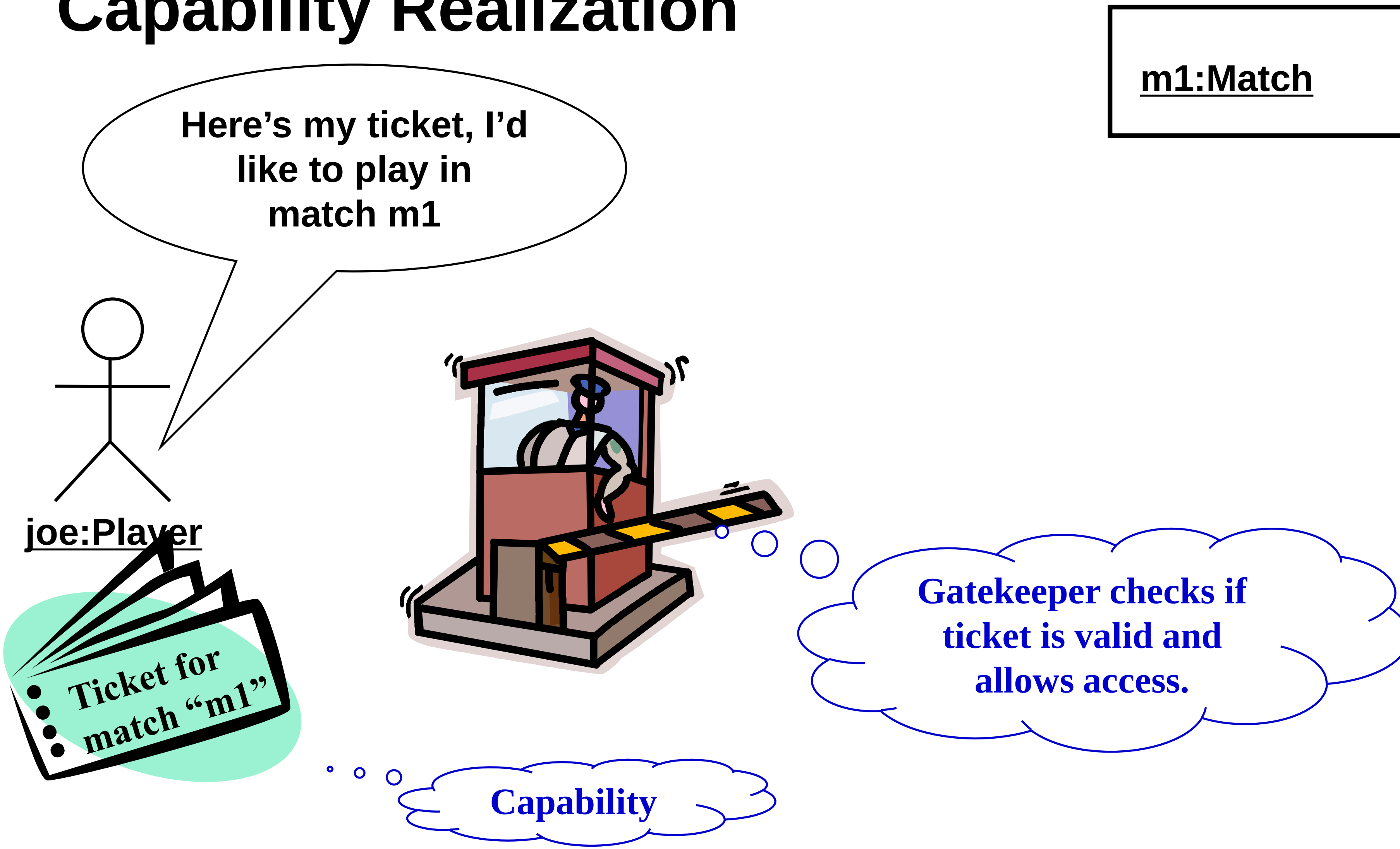


THE UNIVERSITY OF NEWCASTLE AUSTRALIA

Access Control List Realization



Capability Realization



Global Resource Questions

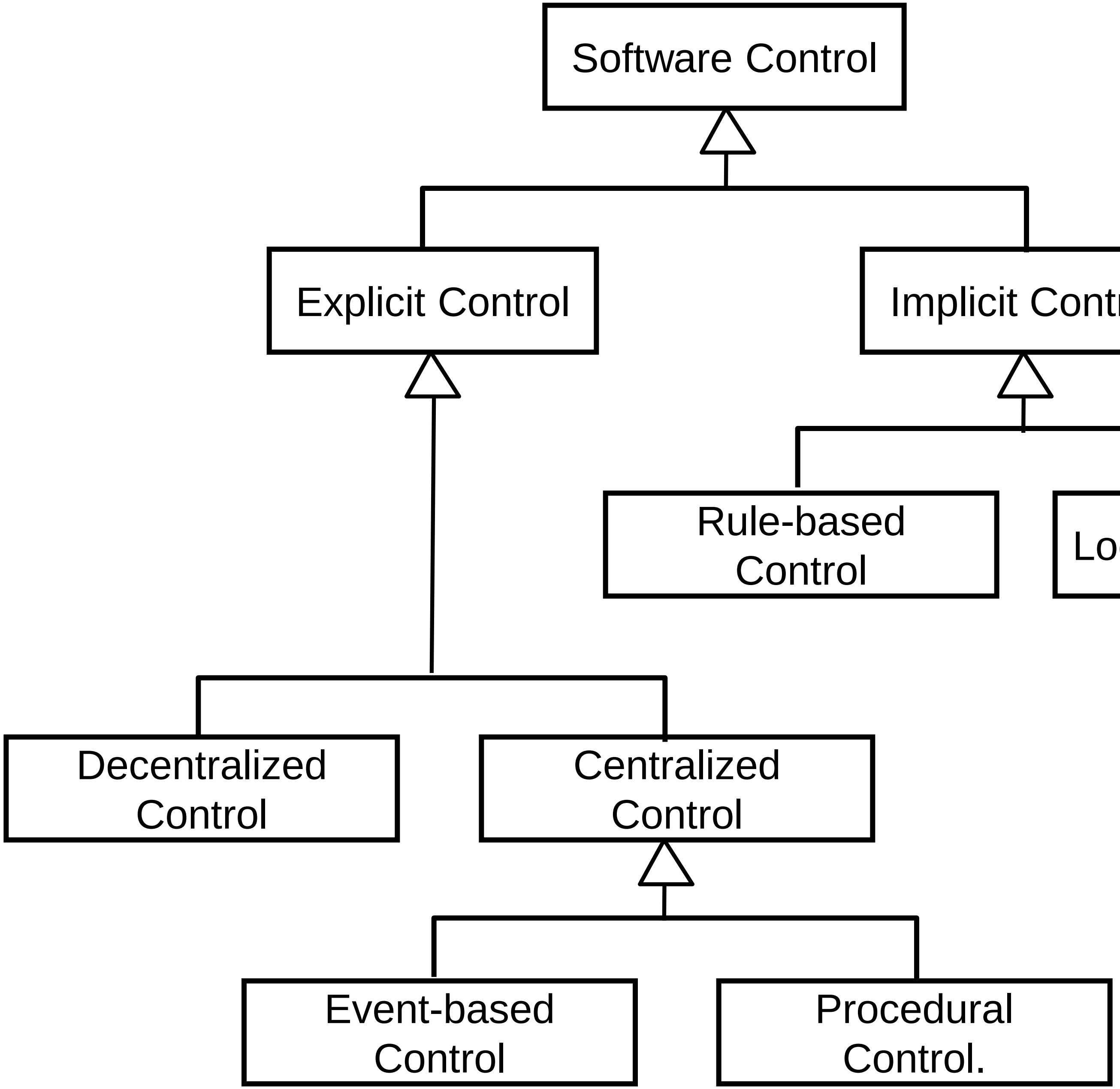
- Does the system need authentication?
- If yes, what is the authentication scheme?
  - User name and password? Access control list
  - Tickets? Capability-based
- What is the user interface for authentication?
- Does the system need a network-wide name server?
- How is a service known to the rest of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



THE UNIVERSITY OF NEWCASTLE AUSTRALIA

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

- 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?.

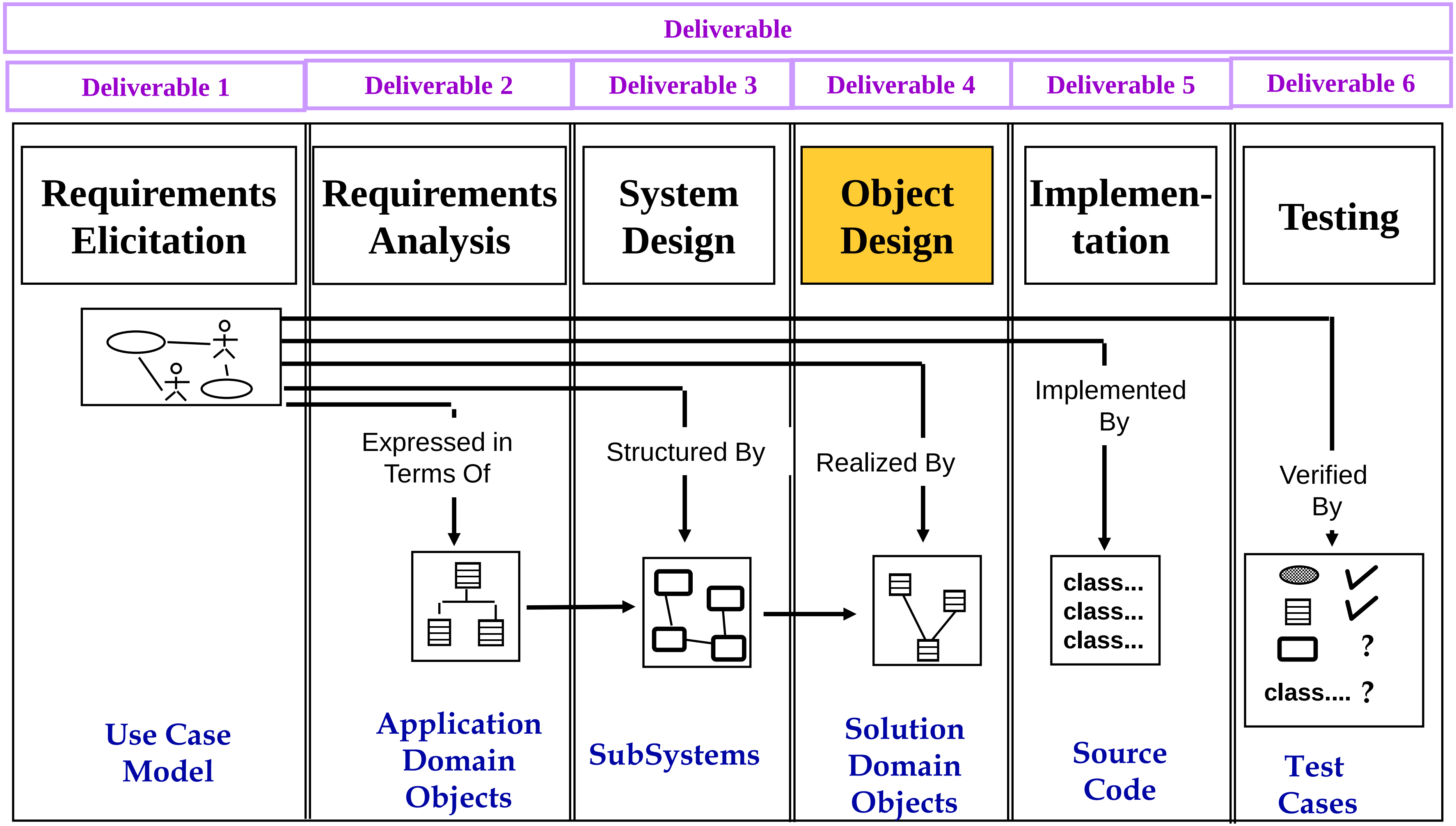
Modeling Boundary Conditions

- 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.

Summary

- System Design
  - An activity that reduces the gap between the problem and an existing (virtual) machine
- Design Goals Definition
  - Describes the important system qualities
  - Defines the values against which options are evaluated
- Subsystem Decomposition
  - Decomposes the overall system into manageable parts by using the principles of cohesion and coherence
  - Software Pattern: an instance of an architectural style
    - Layered, Repository, Blackboard, Client/Server, Peer-to-Peer, Model-View-Controller
- System design activities:
  - Concurrency identification
  - Hardware/Software mapping
  - Database management
  - Global resource handling
  - Software control selection
  - Boundary conditions

Next Week



Deliverable
Deliverable 1
Deliverable 2
Deliverable 3
Deliverable 4
Deliverable 5
Deliverable 6
Requirements Elicitation
Requirements Analysis
System Design
Object Design
Implemen-tation
Testing
Expressed in Terms Of
Structured By
Realized By
Implemented By
Verified By
class...
class...
class...
class.... ?
?
Use Case Model
Application Domain Objects
SubSystems
Solution Domain Objects
Source Code
Test Cases