CptS 487 Software Design

Lecture #22

System Design – Part 2

Persistent Data Management, Access Control

Some of the slides have been adopted from the supplementary notes provided for Object-Oriented Software Engineering textbook

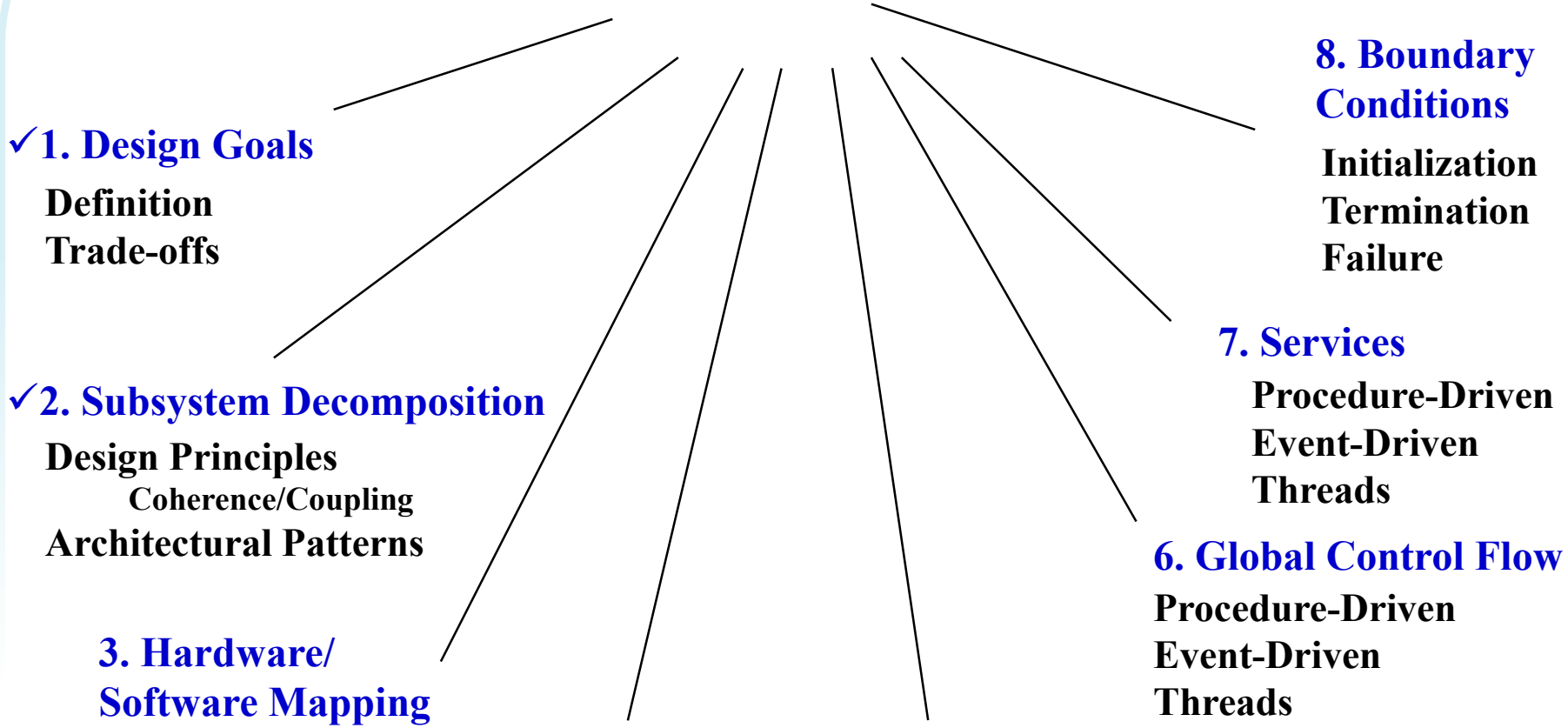
Instructor: Bolong Zeng

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Overview

- Persistent Data Storage
  - File vs. Database system
  - Be able to map simple class diagrams with associations into relational database tables
- Access control
  - 3 ways to implement Access Matrix
  - Understand the differences, and be able to write correct representations of access rights in:
    - Access control list
    - Capability

System Design



✓1. Design Goals

Definition
Trade-offs

✓2. Subsystem Decomposition

Design Principles
Coherence/Coupling
Architectural Patterns

3. Hardware/ Software Mapping

Special Purpose
Buy vs Build
Allocation of Resources
Connectivity

4. Persistent Data Management

Persistent Objects
File system vs Database

5. Access Control

Global Access Table vs Access Control List vs Capabilities
Security

6. Global Control Flow

Procedure-Driven
Event-Driven
Threads

7. Services

Procedure-Driven
Event-Driven
Threads

8. Boundary Conditions

Initialization
Termination
Failure

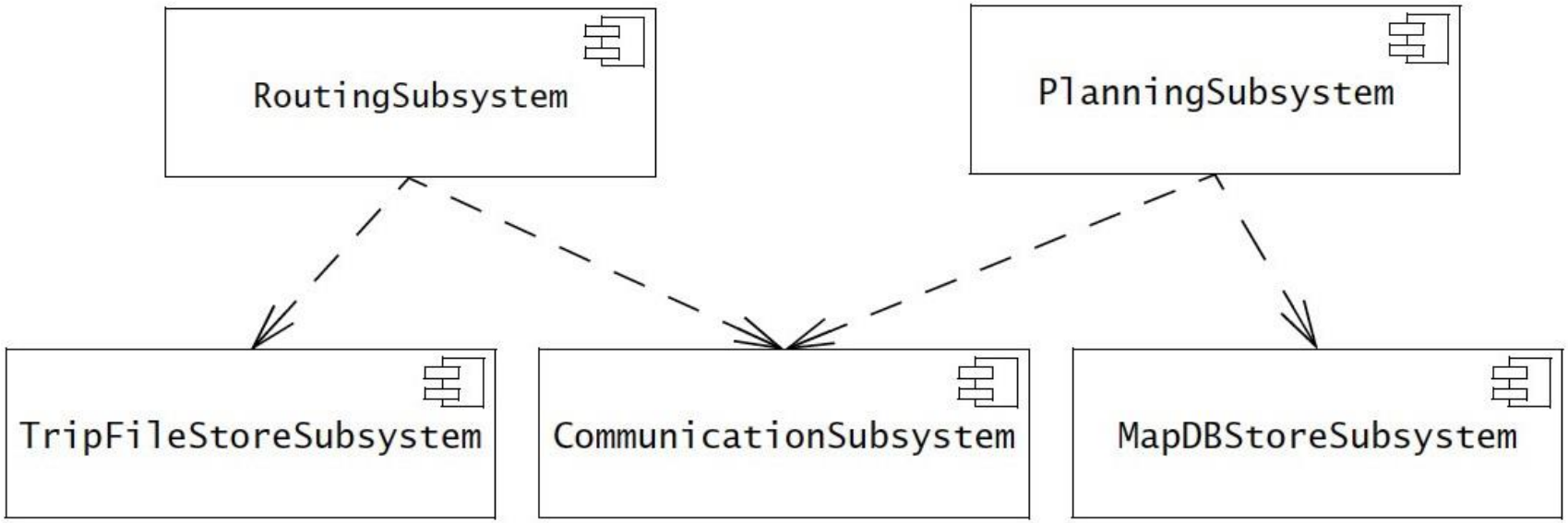
4. Persistent Data Management

- Some objects in the system model need to be persistent:
  - Values for their attributes have a lifetime longer than a single execution
    - Example: Information related to employees (employment status, paychecks, etc.)
- A persistent object can be realized with one of the following mechanisms:
  - File system:
    - If the data are used by multiple readers but a single writer
  - Database:
    - If the data are used by concurrent writers and readers.

Identifying Persistent Objects

- Identify all the objects that must survive the system shutdowns (both controlled shutdown and unexpected crash)
- Candidate persistent objects:
  - Most entity objects
  - Some properties of boundary and control objects (example: user interface preferences)

MyTrip – Revised UML Component Diagram



- The **TripFi leStoreSubsystem** is responsible for storing trips in files on the onboard computer
- The **MapDBStoreSubsystem** is responsible for storing maps and trips in a database for the Planni ngSubsystem.

Data Management Questions

- Should the objects be retrieved quickly?
- How often are they accessed?
  - What is the expected request (query) rate? The worst case?
- How complex are the queries on the object data?
- Do objects require a lot of memory or disk space?
- 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?

Selecting a Storage Management Strategy

- Flat Files: Object data are stored as a sequence of bytes
  - Storage abstraction provided by operating system
- Relational Databases: Object data are stored in tables (with a predefined type called a **schema**)
- Object Oriented Database: Data are stored as objects and associations
  - Reduces the need for translation between objects and storage entities
  - Supports inheritance relations and abstract data types
  - Slower than relational databases

Trade-off between Storage Management Strategies

When should you choose:

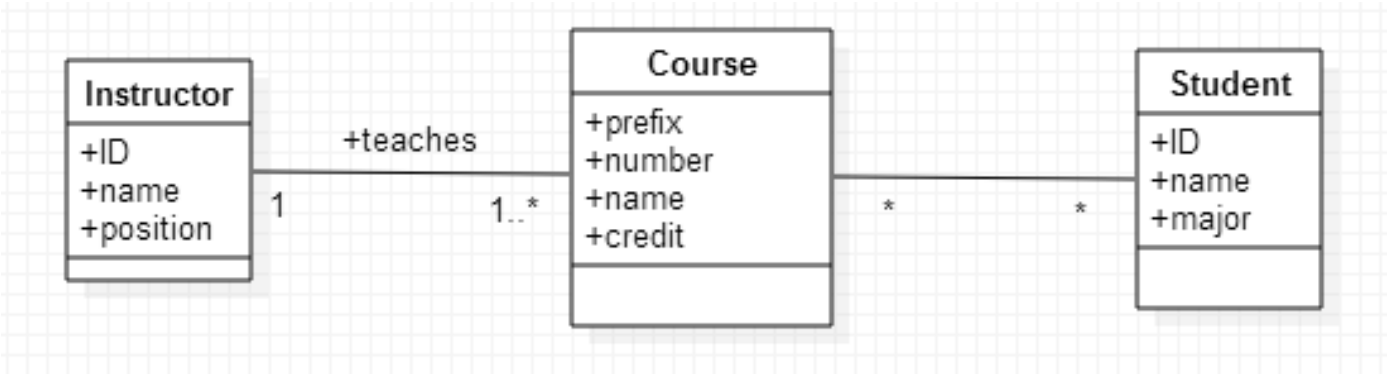
- Flat files:
  - Voluminous data (e.g., images)
  - Temporary data (e.g., core file)
  - Low information density (e.g., archival files, history logs)
- Relational or object-oriented database:
  - Concurrent accesses
  - Access at finer levels of detail
  - Multiple platforms or applications for the same data
- Relational database:
  - Complex queries over attributes
  - Large data set
- Object-oriented database:
  - Extensive use of associations to retrieve data
  - Medium-sized data set
  - Irregular associations among objects

Object Model to Relational Database

- Mapping the UML object model to a relational database
  - 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
  - Some degradation occurs since all UML constructs has to be mapped to one relational database construct: a table.

Mapping Classes to Tables

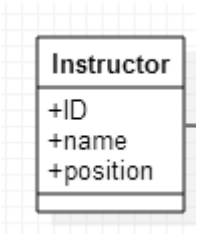
- One-to-many and many-to-many associations
  - How would you save data of these classes into a database, using tables?
  - First off, discuss what the multiplicities in this diagram mean.



Primary and Foreign Keys

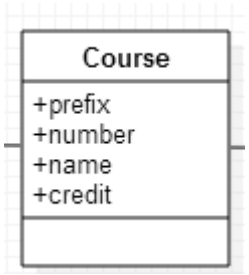
- Candidate key
  - Any set of attributes that could be used to uniquely identify any data record in a relational table
- Primary key
  - The actual candidate key that is used in the application to identify the records.
  - The primary key of a table is a set of attributes whose values uniquely identify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

Mapping a Class to a Table



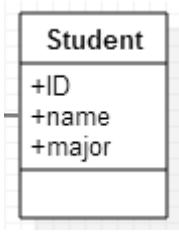
ID	Name	position
159	Alice	Professor
357	Bob	Lecturer

Primary Key



Primary Key

Prefix	Number	Name	Credit
CptS	323	Software Design	3
EE	311	Electronics	3
CptS	421	Senior Design	3

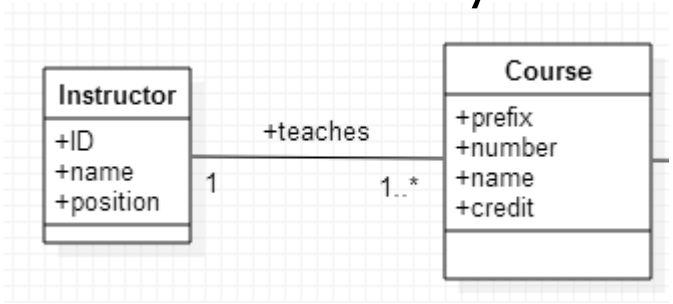


Primary Key

ID	Name	major
001	John	CS
002	Mary	EE
003	Andy	CE
004	Bill	CS
005	Tina	EE

Mapping one-to-many associations

- 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.



ID	Name	position
159	Alice	Professor
357	Bob	Lecturer

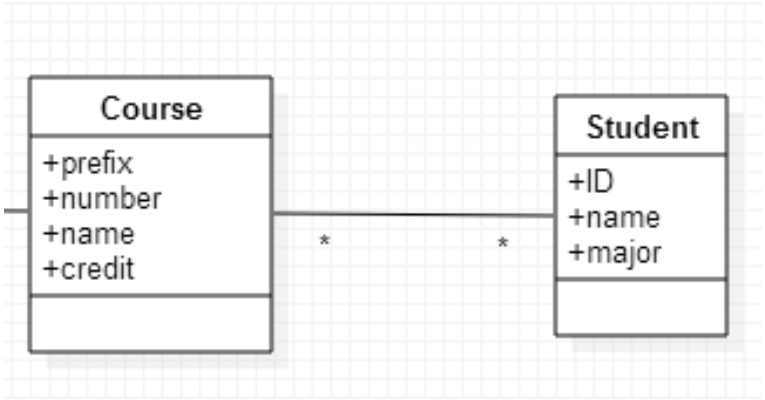
Primary Key

Primary Key

Prefix	Number	Name	Credit	Instructor
CptS	323	Software Design	3	357
EE	311	Electronics	3	159
CptS	421	Senior Design	3	357

Mapping many-to-many associations

- In this case we need a separate table for the association



Student	Course
001	CptS323
001	CptS421
002	EE311
003	CptS323
003	EE311
005	EE311

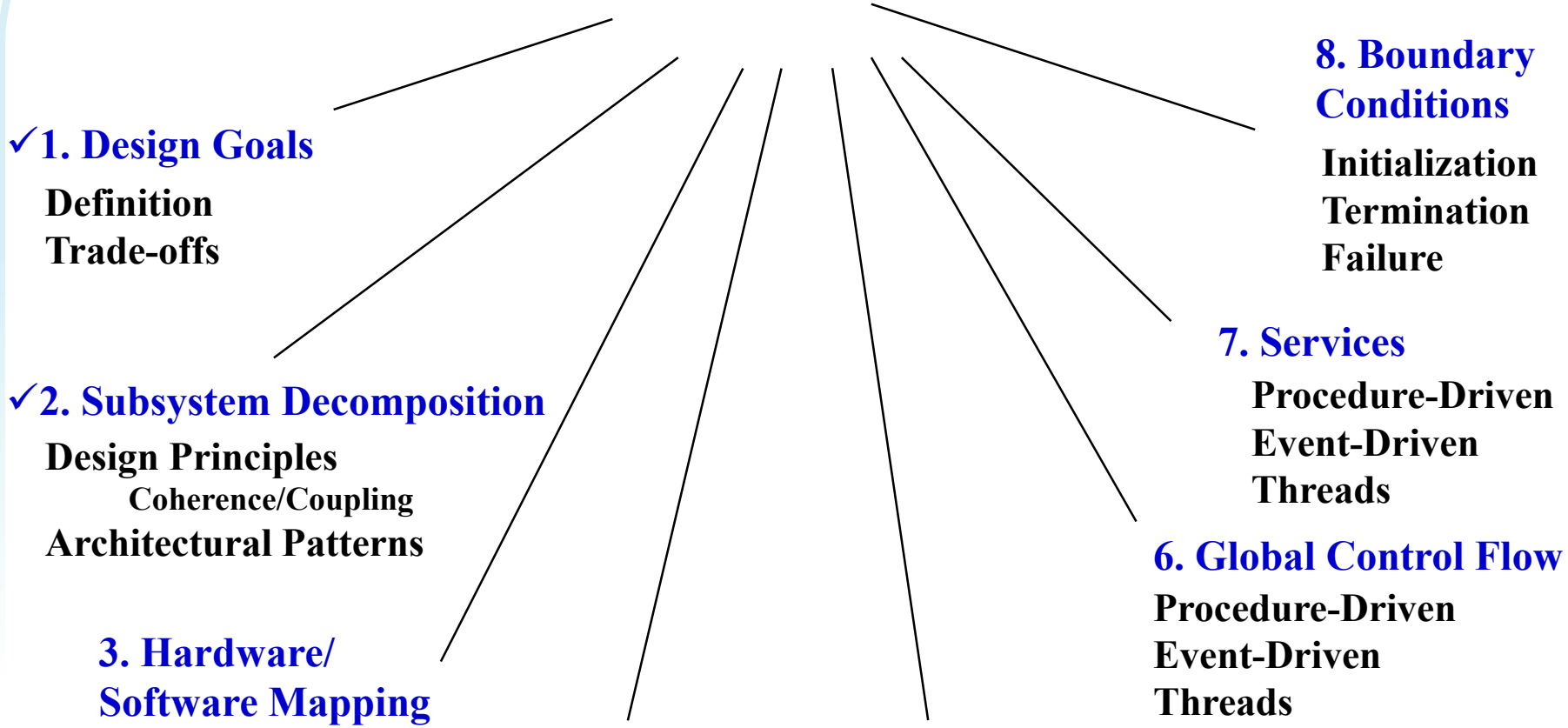
Primary Key

Course Number	Name	Credit
CptS323	Software Design	3
EE311	Electronics	3
CptS421	Senior Design	3

Primary Key

ID	Name	major
001	John	CS
002	Mary	EE
003	Andy	CE
004	Bill	CS
005	Tina	EE

System Design



✓1. Design Goals

Definition
Trade-offs

✓2. Subsystem Decomposition

Design Principles
Coherence/Coupling
Architectural Patterns

3. Hardware/ Software Mapping

Special Purpose
Buy vs Build
Allocation of Resources
Connectivity

4. Persistent Data Management

Persistent Objects
File system vs Database

5. Access Control

Global Access Table vs Access Control List vs Capabilities
Security

6. Global Control Flow

Procedure-Driven
Event-Driven
Threads

7. Services

Procedure-Driven
Event-Driven
Threads

8. Boundary Conditions

Initialization
Termination
Failure

5. Providing Access Control

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

Defining Access Control

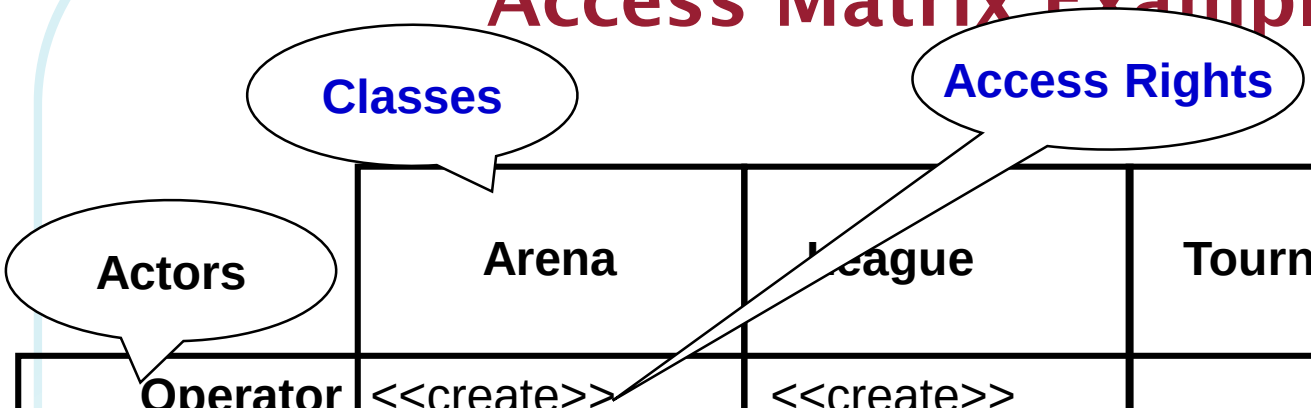
- In multi-user systems different actors usually have different access rights to different functionality and data
- How do we model these accesses?
  - During analysis we model them by associating different use cases with different actors
  - During system design we model them determining which objects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



	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

- The access matrix can be represented using one of the three approaches:

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

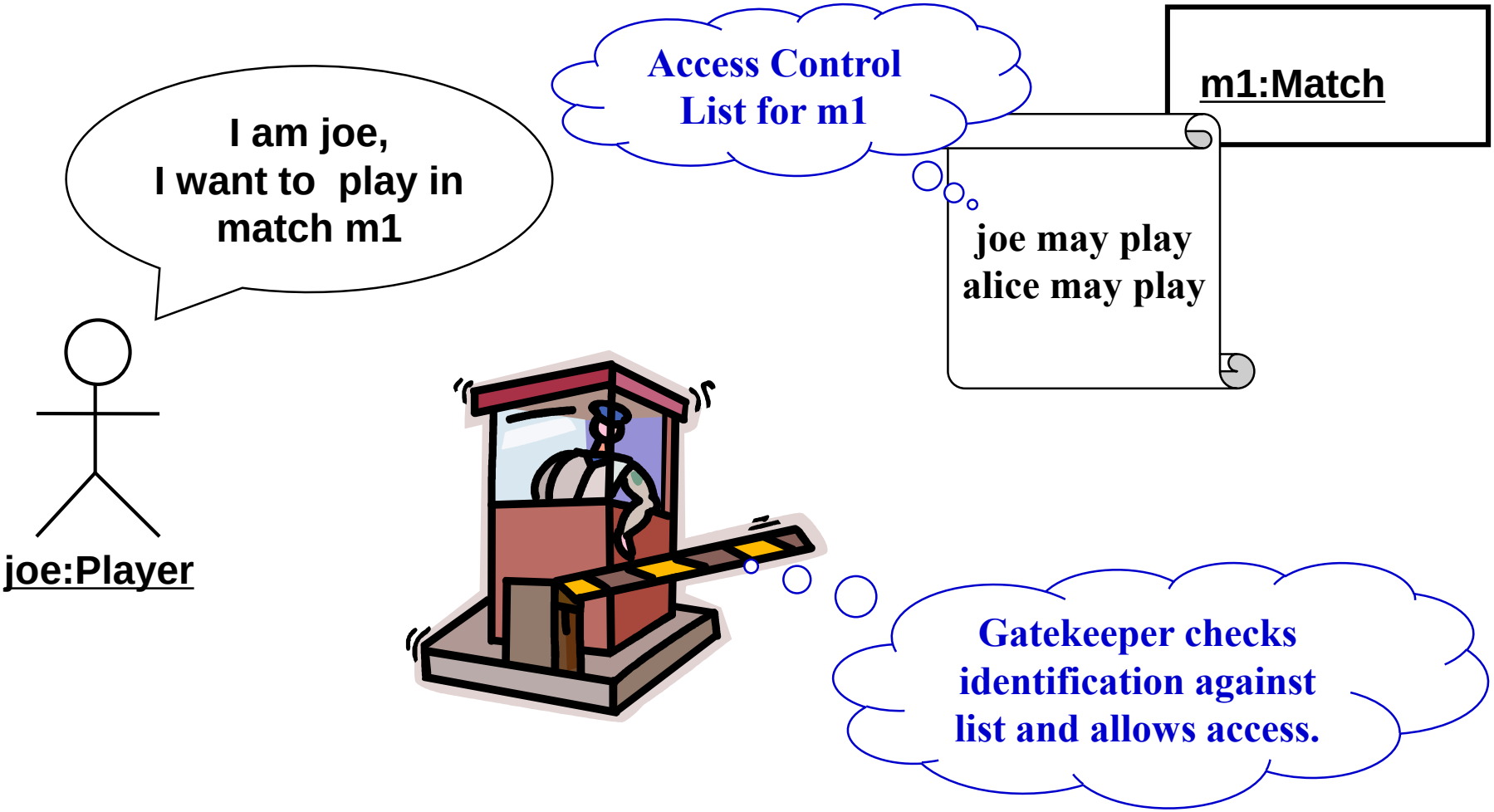
LeagueOwner, **Arena**, view()
LeagueOwner, **League, edit()**
LeagueOwner, **Tournament**, <<create>>
LeagueOwner, **Tournament**, view()
LeagueOwner, **Tournament**, schedule()
LeagueOwner, **Tournament**, archive()
LeagueOwner, **Match**, <<create>>
LeagueOwner, **Match**, end()

.

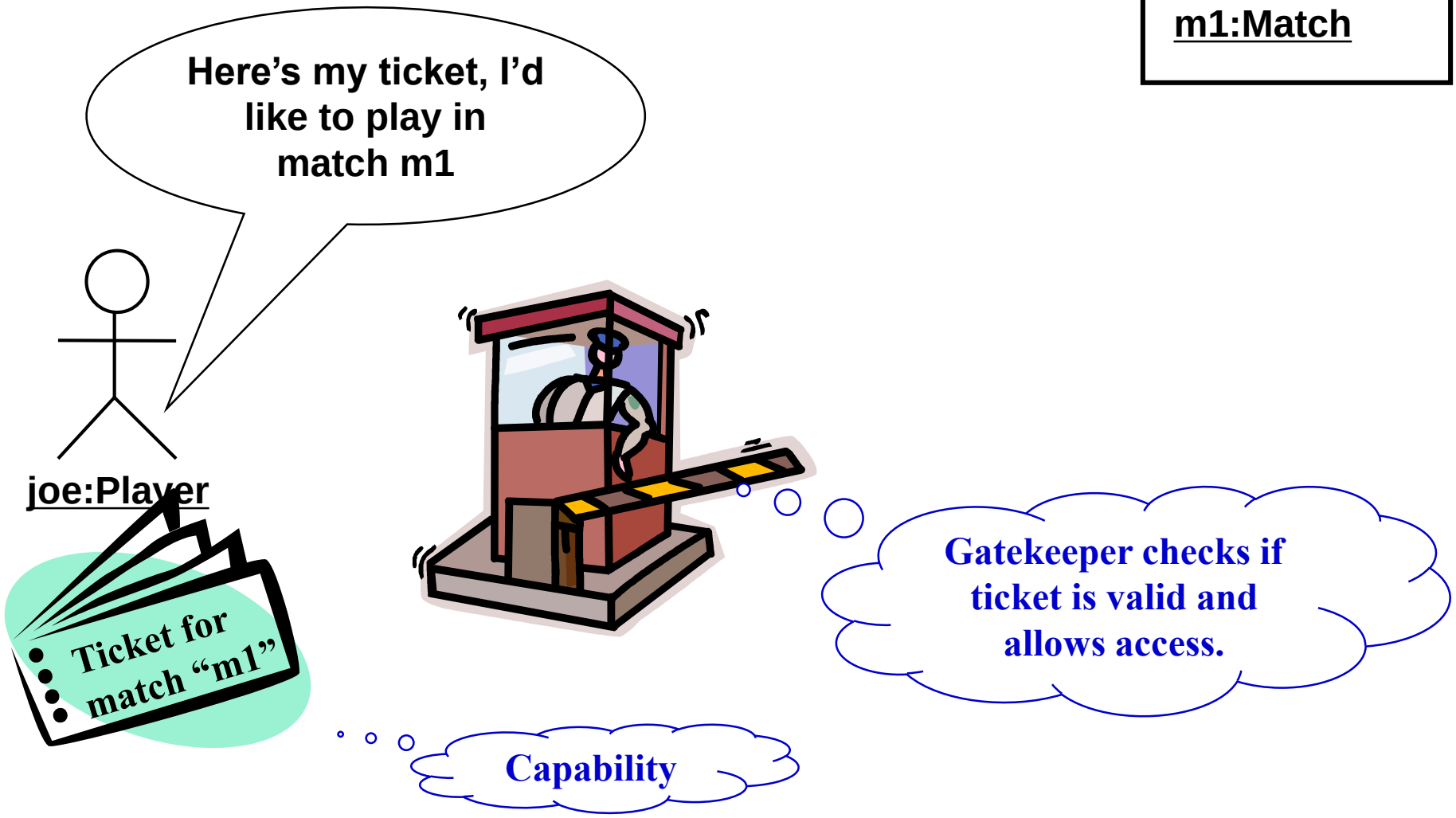
Access Matrix Implementations

- 2. 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.
- 3. 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 Control List Realization



Capability Realization



Access Control 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?