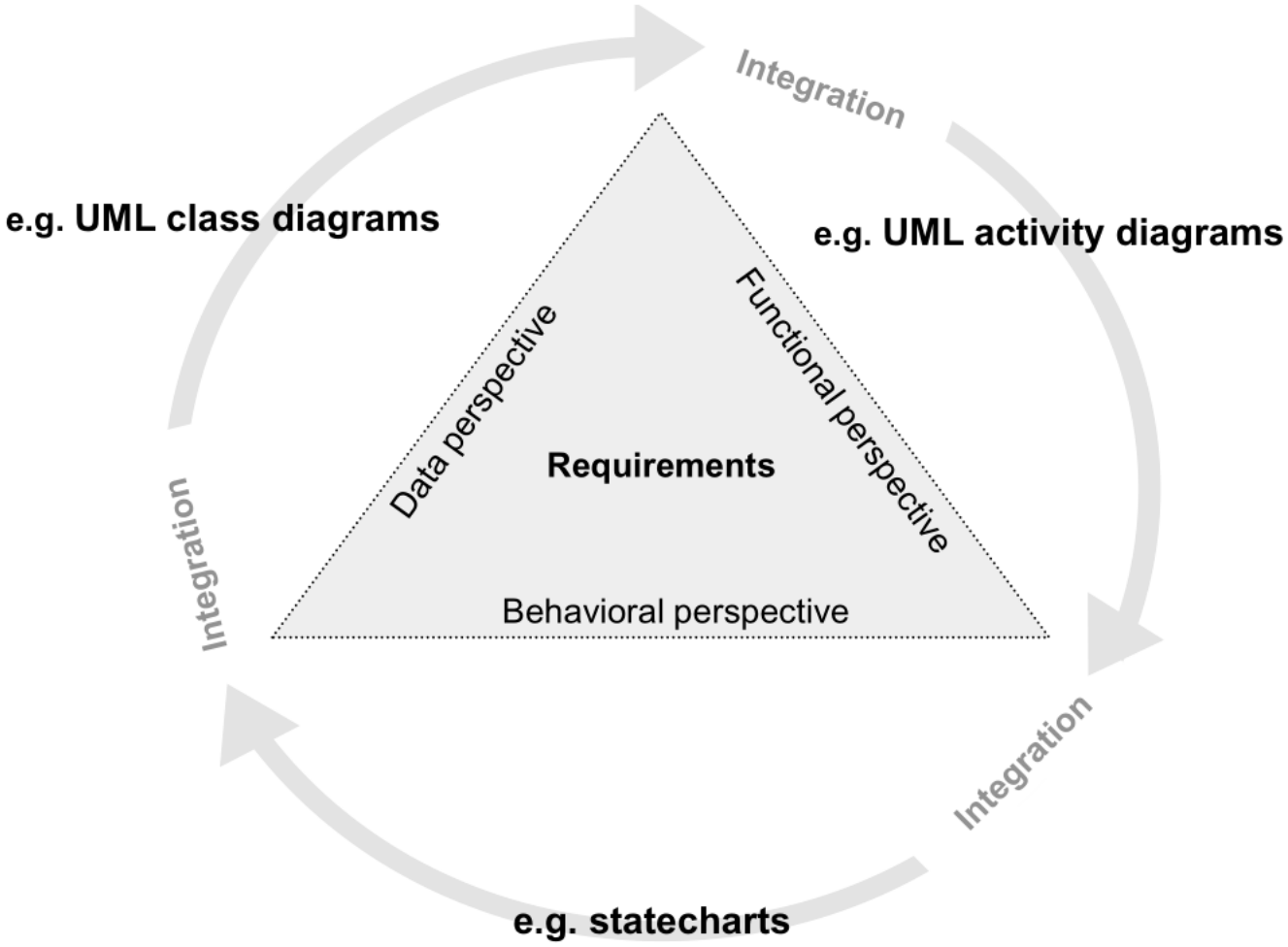
Lecture 4

Use case modeling
& Conceptual Data Modeling

Faculty of Information Technology
Hanoi University

3 Perspectives of Requirements



Part 1

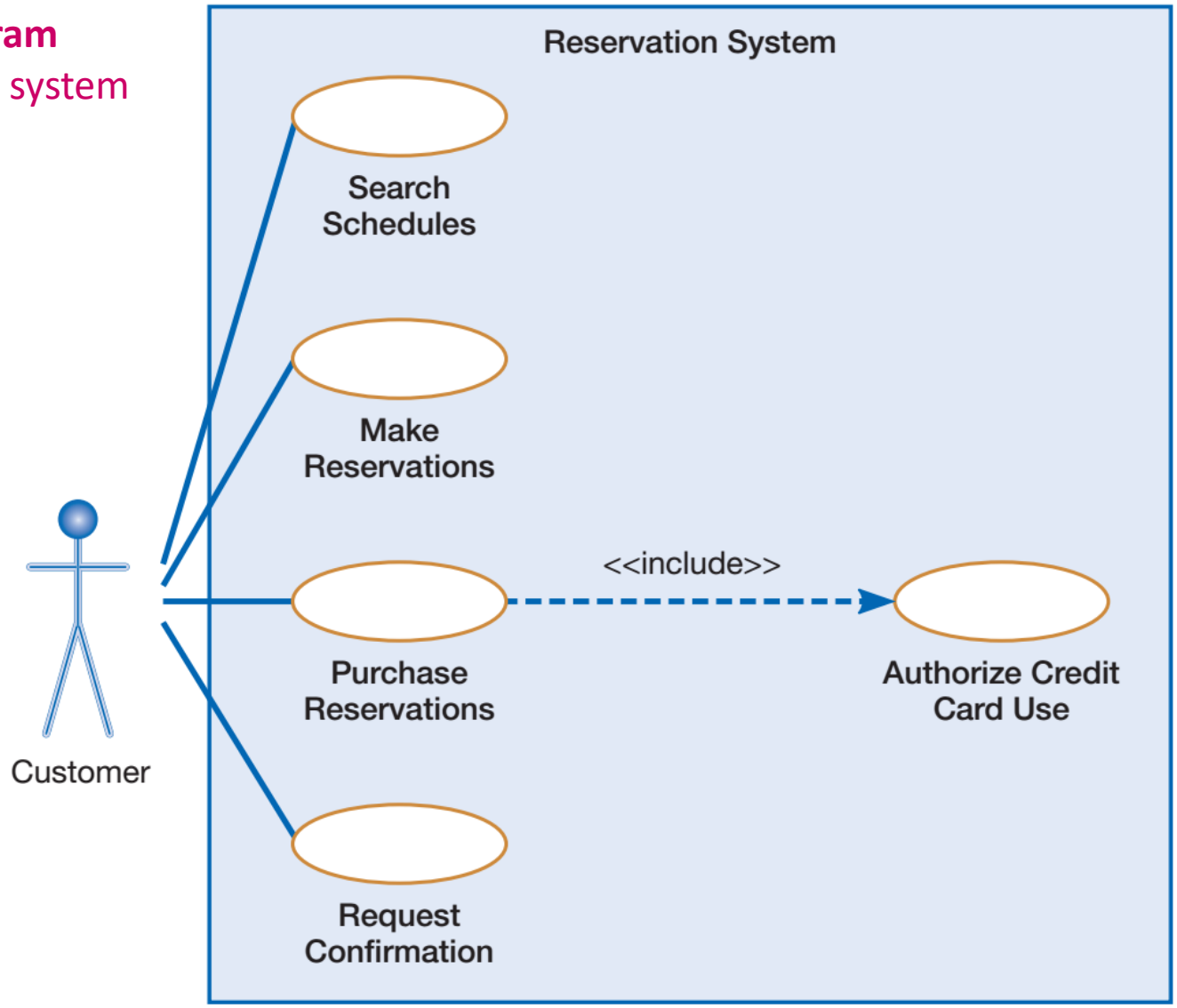
USE CASE MODELING

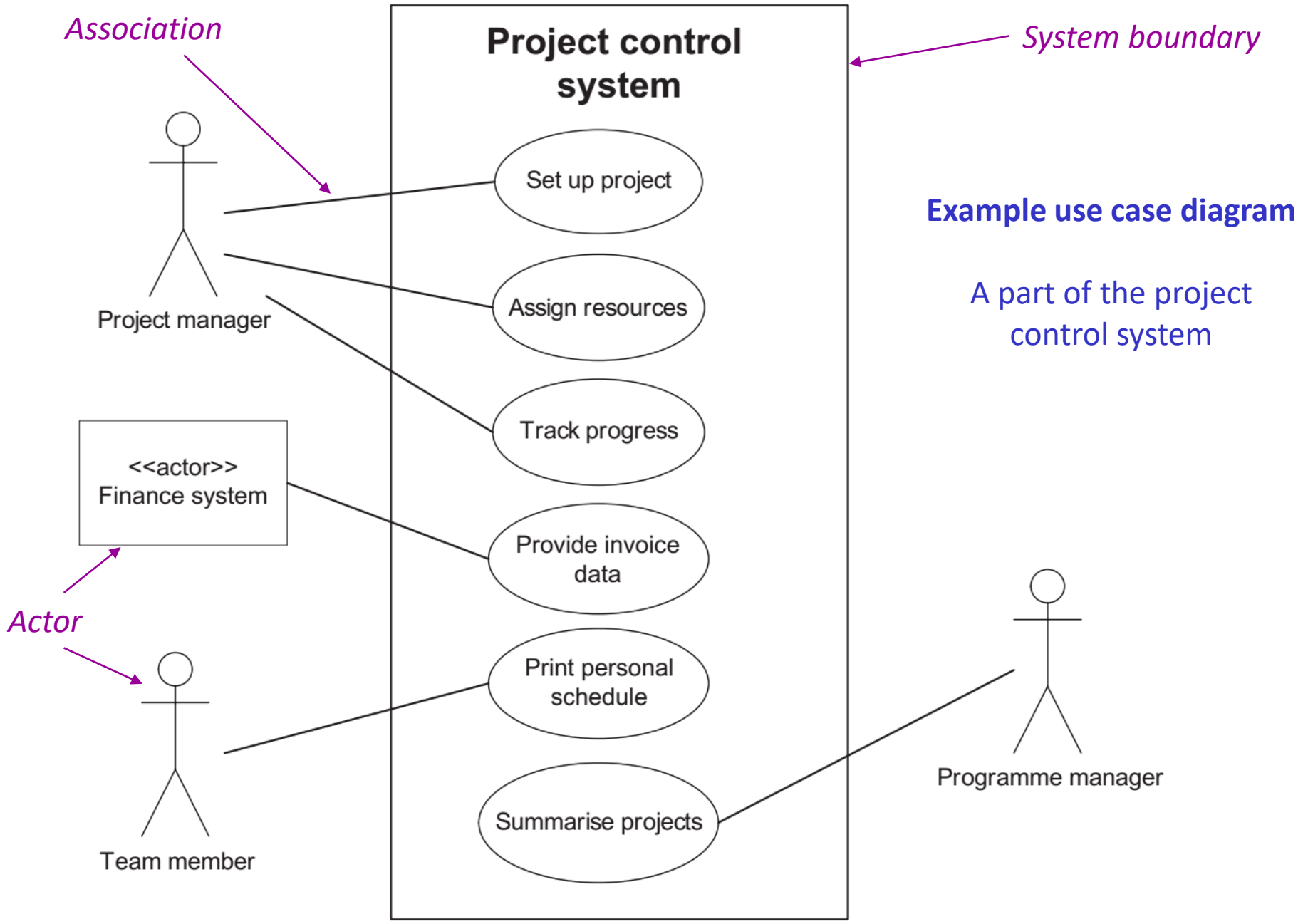
Use case diagram

- **Purpose:** Analyze/record the functional requirements of a system.
- **Use case:** a function that the system performs
  - Usually in response to a trigger from an actor
- **Actor:** An external entity that interacts with a system
  - An actor is usually a user role but can also be an external system
- Use case modeling is part of the Unified Modeling Language (UML)

Example use case diagram

Hotel room reservation system



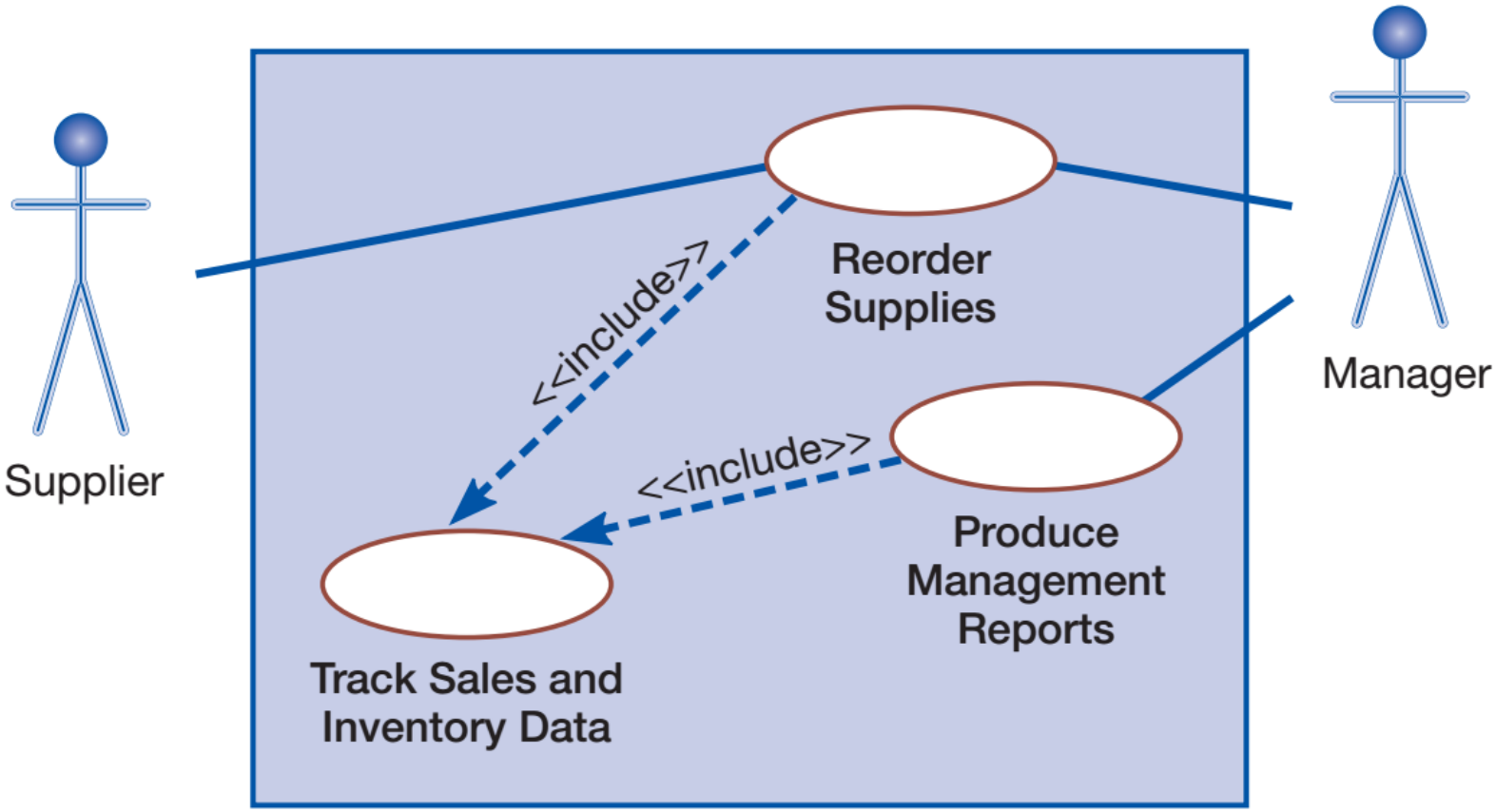


Example use case diagram

A part of the project control system

<<include>> and <<extend>>

- **<<include>>**
  - In a system, certain actions may be repeated
  - Such a general-purpose action can be written as a separate use case and then be used by / contained in other use cases
- **<<extend>>**
  - An *extend* relationship extends a use case by adding new behaviors or actions
  - The extending use case has all the actions in the original one, and some more
  - Specialized use case extends the general use case



Include relationship

An association between two use cases where one use case uses the functionality contained in the other.

<<include>> relationship

- **<<include>>** can be used to decompose a use case into smaller use cases.
- The source use case is incomplete without the included use cases.

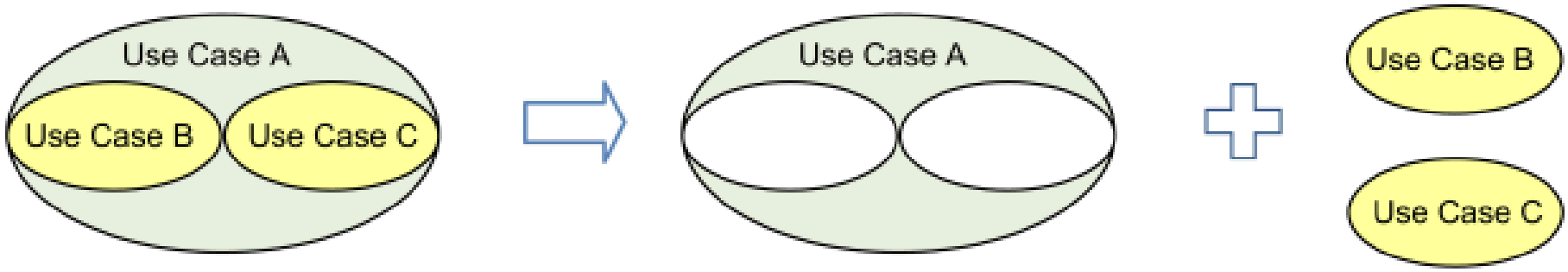


Image Credit: uml-diagrams.org

<<include>> relationship

- **<<include>>** can be used to reuse a common use case.

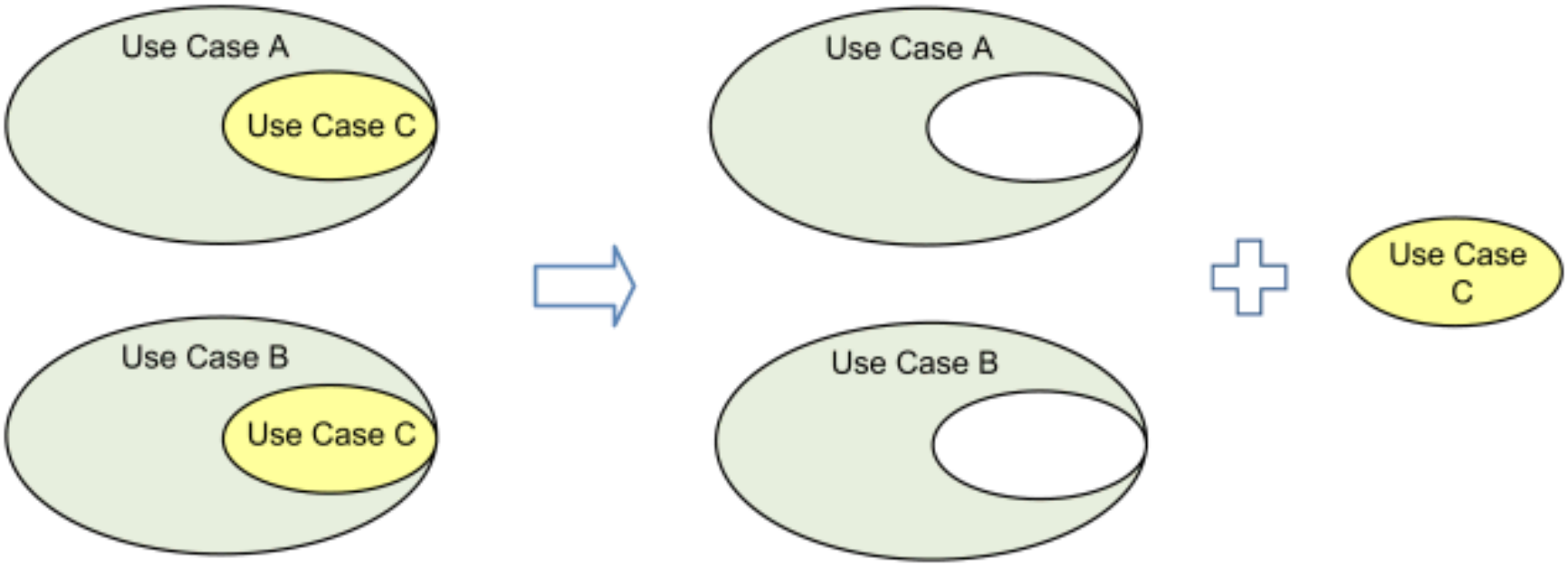
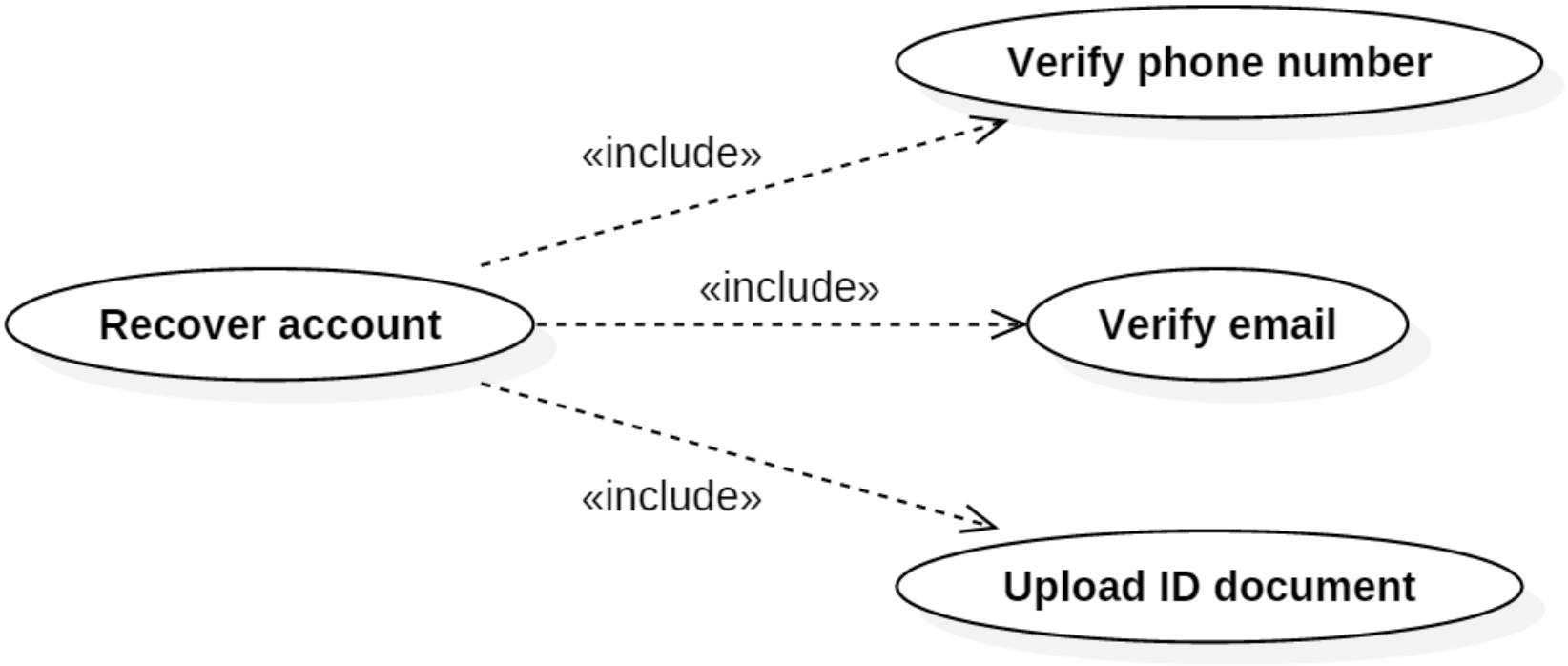
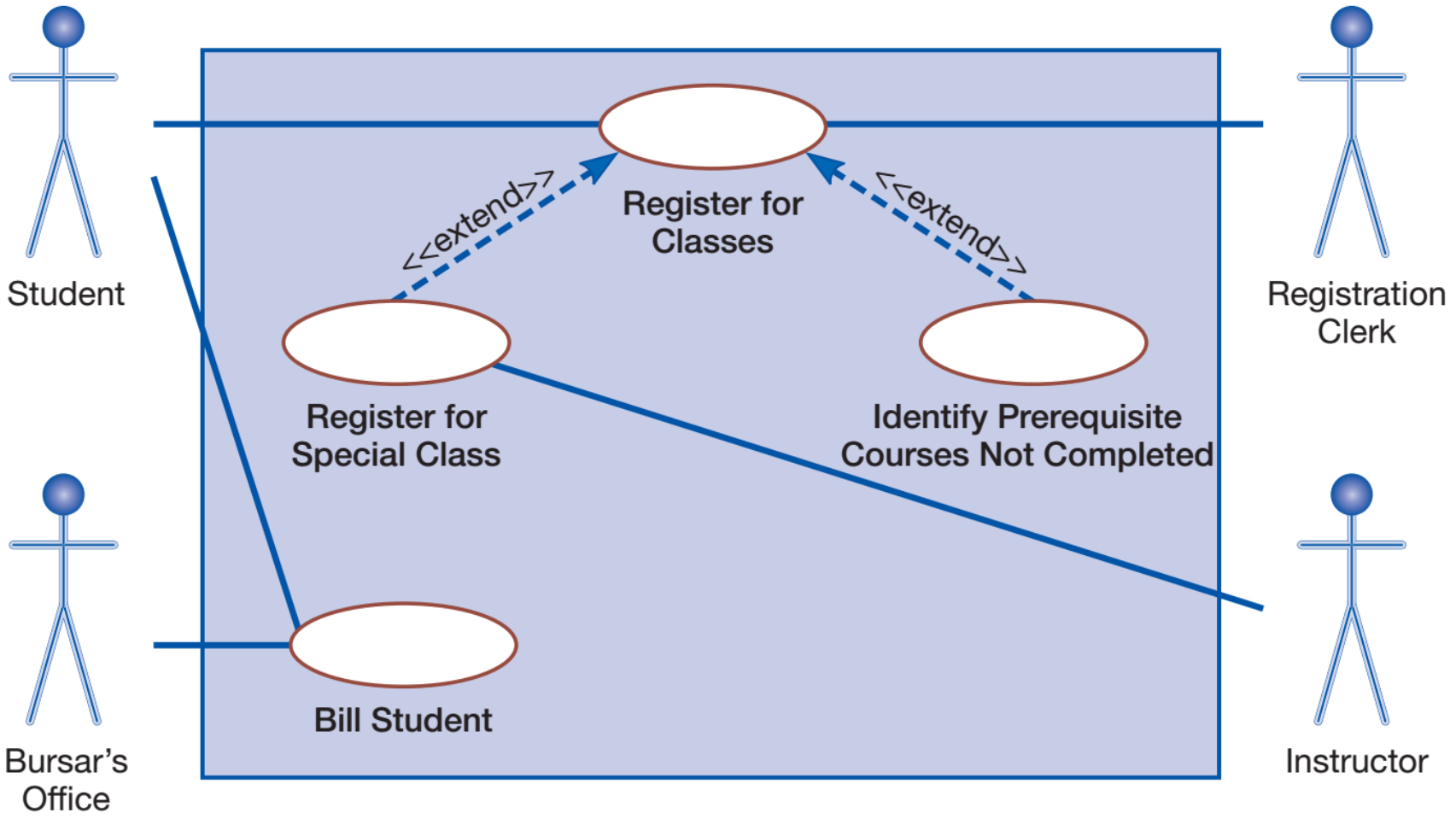


Image Credit: uml-diagrams.org

<<include>> example

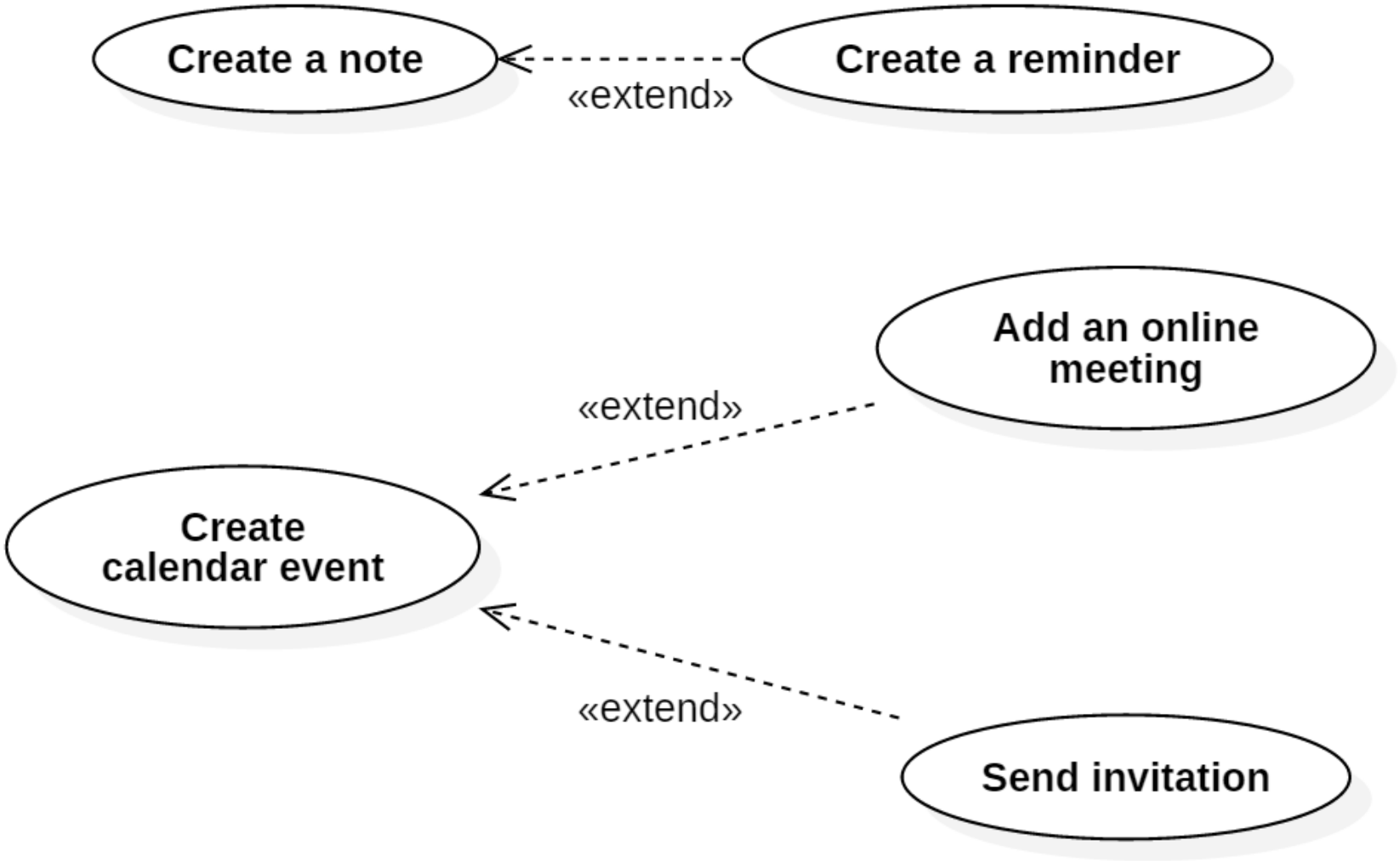




Extend relationship

An association between two use cases where one adds new behaviors or actions to the other.

<<extend>> example



Written use cases

- Document containing detailed specifications for a use case
- Contents can be written as simple text or in a specified format
- Step-by-step description of what must occur in a successful use case
- Each ellipse in the use case diagram should have a corresponding written use case.

A template for written use cases

ID & Name:	UC-3: Write an article
Primary Actor:	Editor
Description:	Briefly describe the use case scenario…
Trigger:	The event that starts the use case… e.g. Editor clicks “Compose” button
Pre-conditions:	The User is logged in. The User has the Editor role.
Post-conditions:	An article is saved to the Pending list (to be reviewed by Chief Editor)
Normal Flow:	1.1. Compose text 1.2. Click submit 1.3. System displays “Successful”
Alternative Flows:	2.1. Compose text 2.2. Save as draft 2.3. Open draft 2.4. Click submit
Exceptions:	**Exception #1**: Article title is duplicated 1.2. Click submit 1.3. System displays *“Error: Duplicated title!”* **Exception #2**: Draft storage is full 2.2. Save as draft 2.3. System displays *“Error: Cannot save draft, storage is full!”*
Priority:	High

Written Use Case Guidelines

- ***Normal Flow*** – description of sequence of interactions between actor and system during the use case execution
- ***Alternative flows*** – scenarios which are different from normal flow but still deliver the same business outcome
- ***Exceptions*** – potential conditions that prevent a use case from succeeding

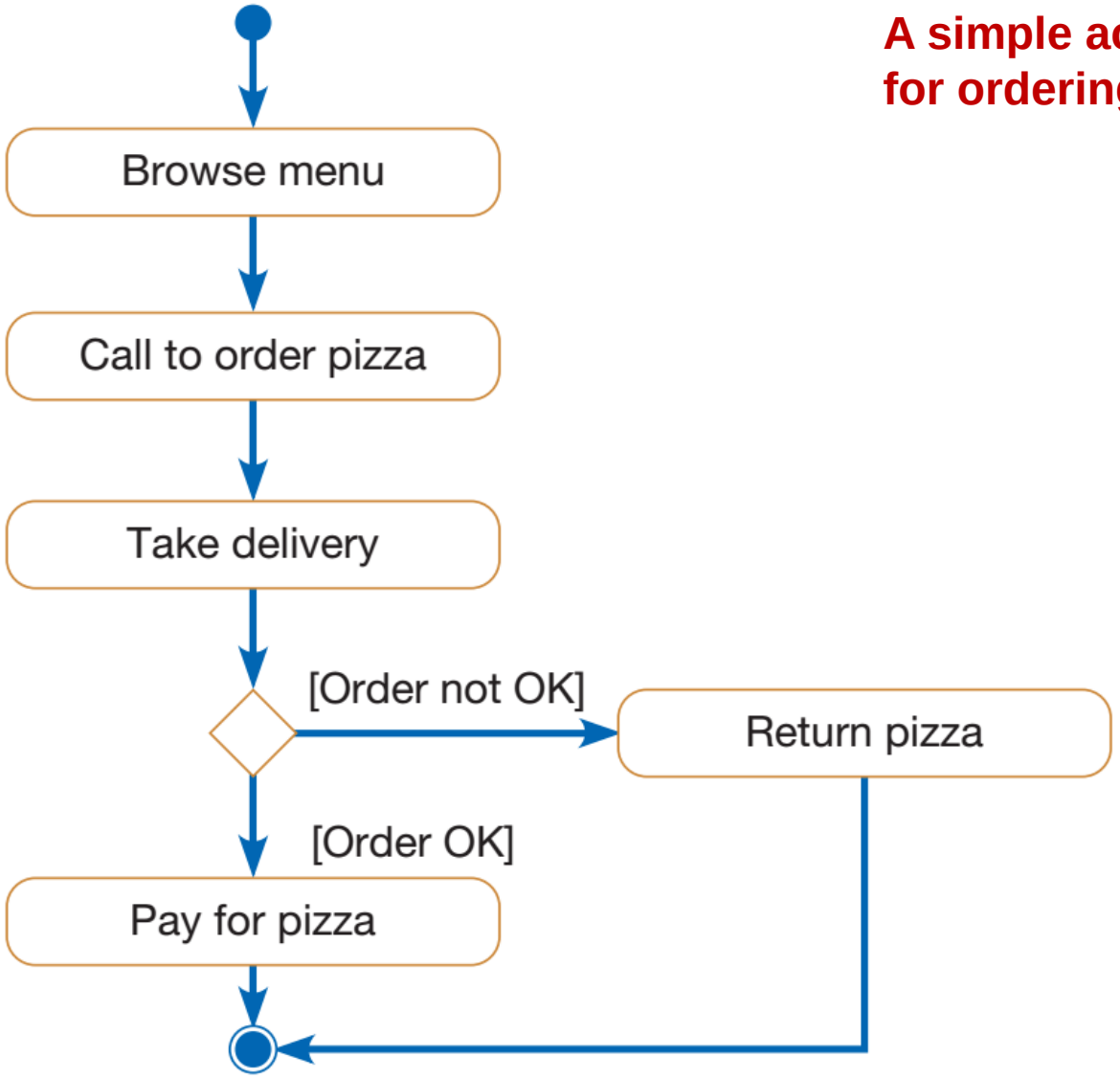
Part 2

ACTIVITY DIAGRAM

Process Modeling: Activity Diagrams

- **Activity Diagrams**
  - Show the conditional logic for the sequence of system activities needed to accomplish a business process.
  - Clearly show parallel and alternative behaviors.
  - Can be used to show the logic of a use case.

A simple activity diagram for ordering pizza



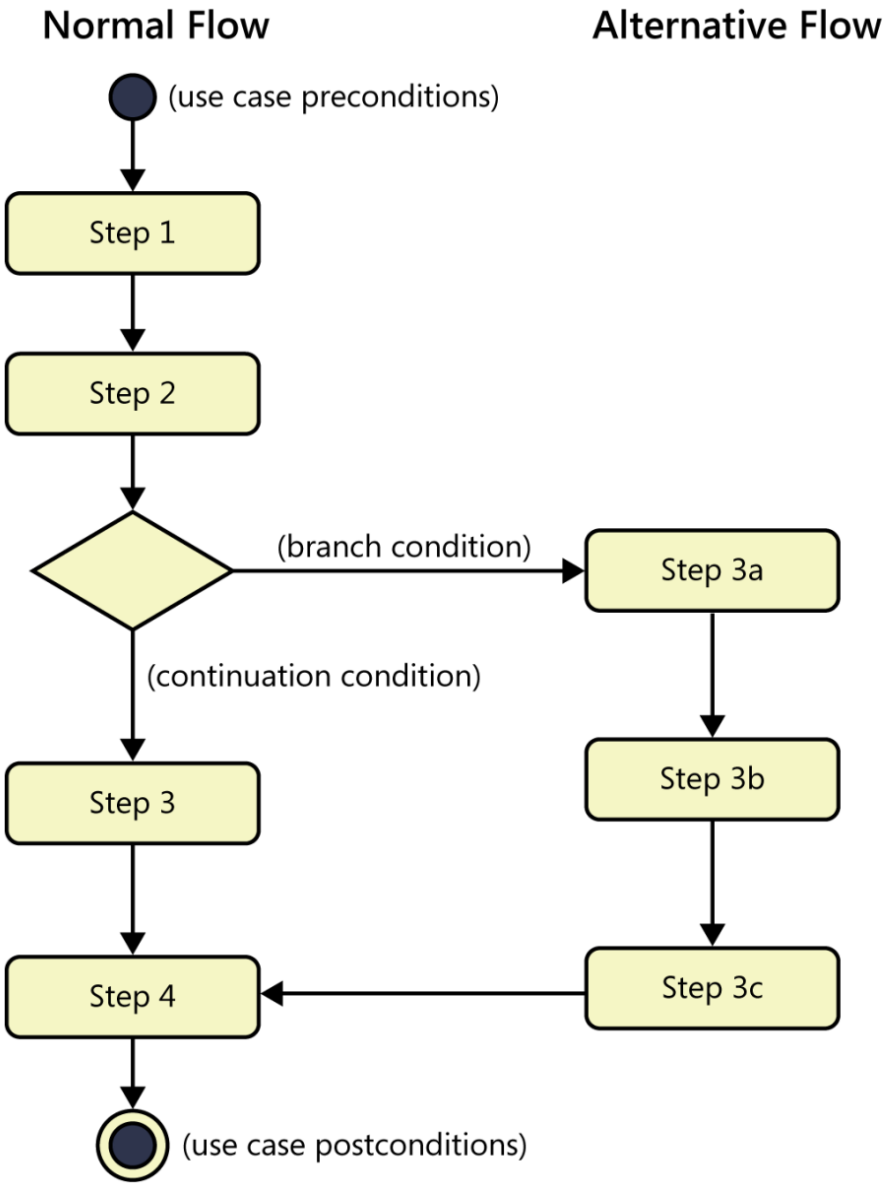
Use Activity Diagrams to:

- Depict the flow of control from activity to activity.
- Help in use case analysis to understand what actions need to take place.
- Help in identifying extensions in a use case.
- Model work flow and business processes.
- Model the sequential and concurrent steps in a computation process.

From Use Case to Activity Diagram

- A flowchart or a UML activity diagram is a useful way to visualize a (complex) use case

- These diagrams show the decision points and conditions that cause a branch from the *normal flow* into an *alternative flow*


Normal Flow
Alternative Flow
(use case preconditions)
Step 1
Step 2
(branch condition)
Step 3a
(continuation condition)
Step 3
Step 3b
Step 4
Step 3c
(use case postconditions)

Part 2

CONCEPTUAL DATA MODELING

Conceptual Data Modeling

- **Conceptual data modeling**: a detailed model that captures the overall structure of data in an organization
  - Entities, attributes, and relationships extracted from analyzing captured requirements
- No assumptions about underlying technology
  - Independent of any DBMS or implementation

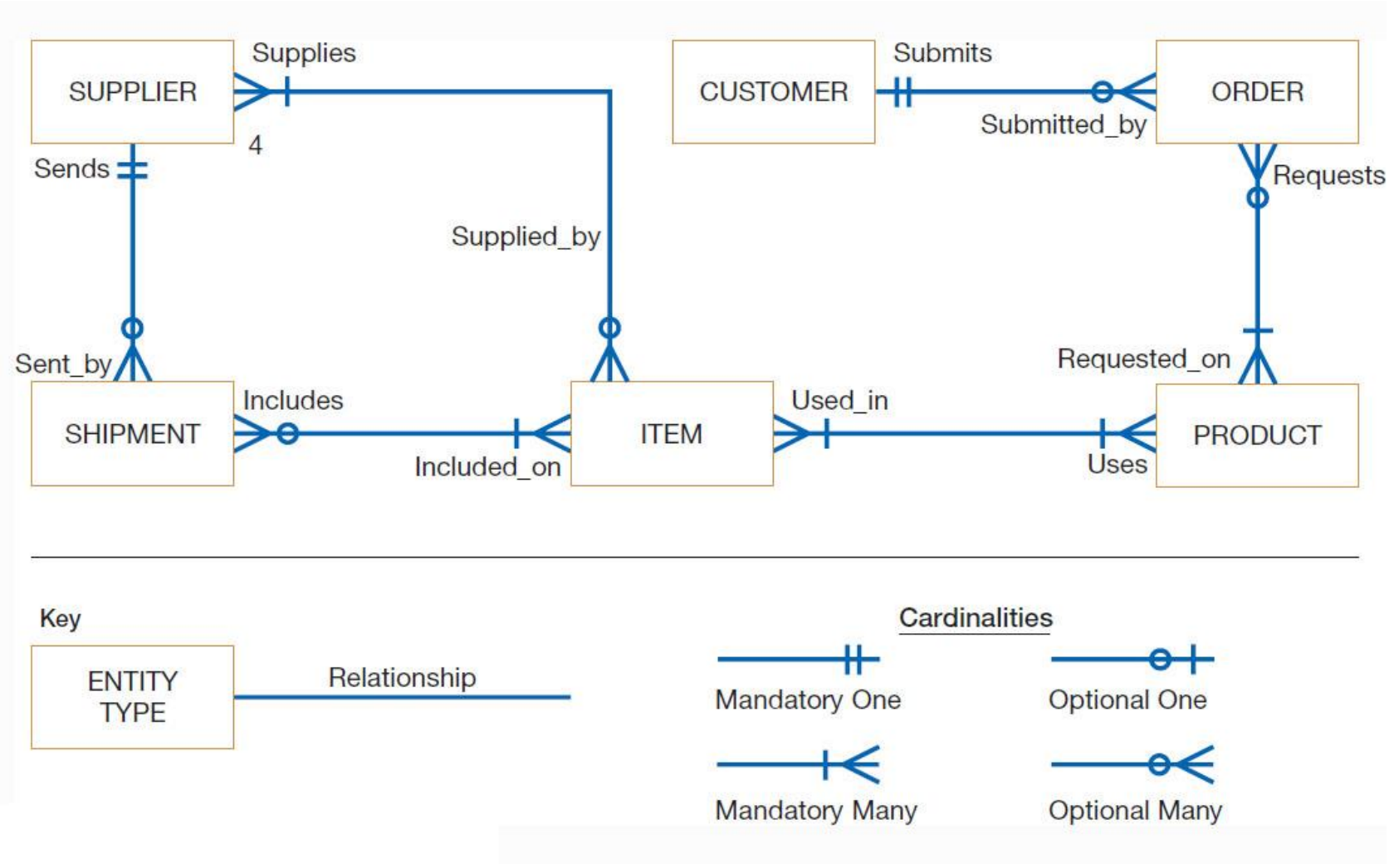
Conceptual Data Modeling

- Conceptual data modeling is typically done in parallel with other requirements analysis and specification activities.
  - E.g. Process modeling, use case modeling...
- In the design stage: a conceptual data model is translated into a physical design.
- Conceptual data modeling is also useful for project planning.

Outcome and deliverables

- Entity-relationship (E-R) diagram or UML class diagram
  - Entities (or classes) – categories of data
  - Relationships (or associations) – connections between entities
- Data elements included in the data flow diagram (DFD) must appear in the data model and vice versa.
- Each data store in a process model must relate to business objects represented in the data model.

Conceptual data model example



Entity-Relationship (E-R) Modeling

- **Entity-Relationship data model** (**E-R model**): a detailed, logical representation of the entities, associations and data elements for an organization or business area
- **Entity-relationship diagram** (**E-R diagram**): a graphical representation of an E-R model

Entity-Relationship (E-R) Modeling

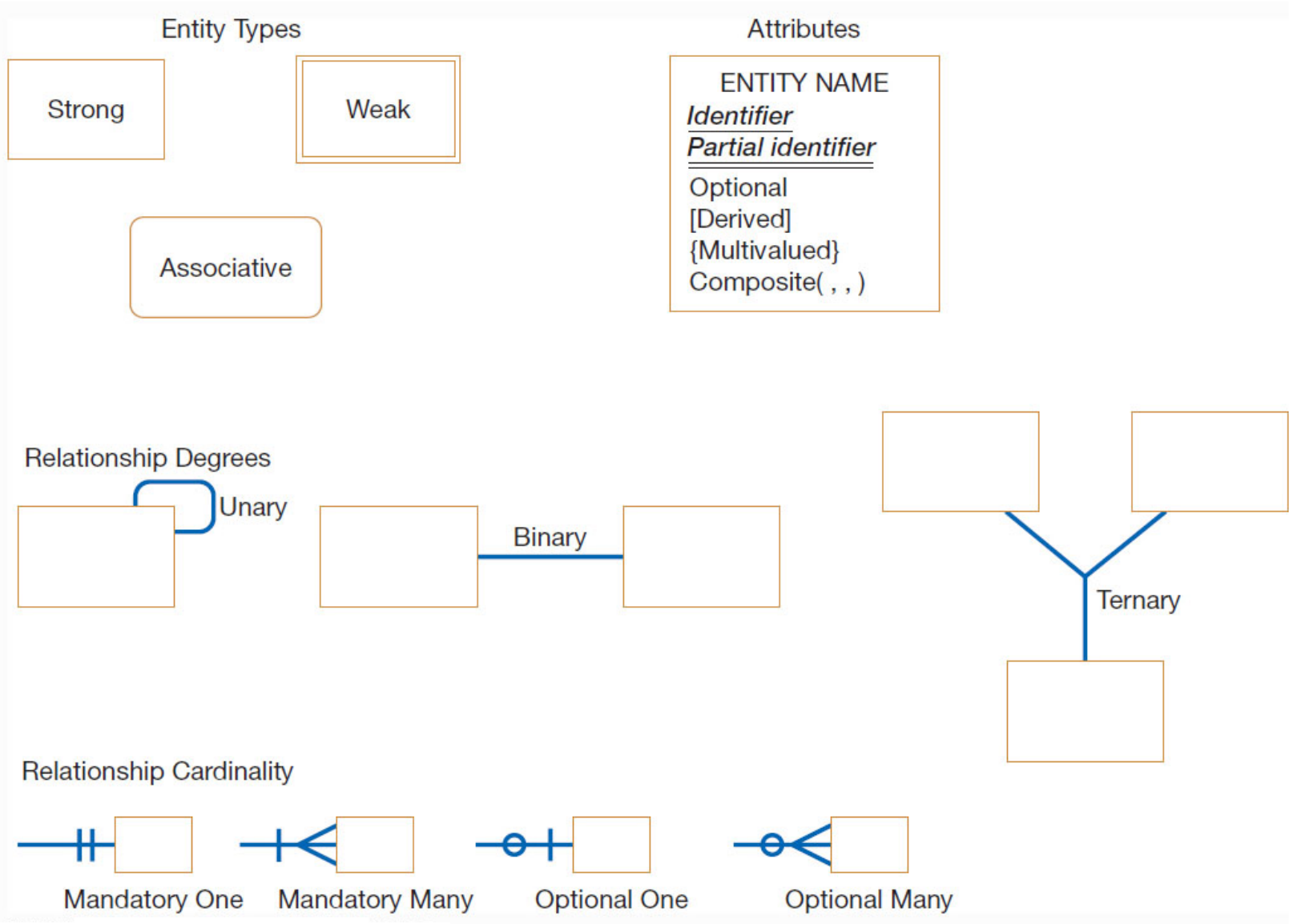
- The E-R model is expressed in terms of:
  - Data entities in the business environment
  - Relationships or associations among those entities
  - Attributes or properties of both the entities and their relationships

Entity-Relationship (E-R) Modeling

- **Entity**: a person, place, object, event or concept in the user environment about which data is to be maintained
  - An entity should be about something that is being represented by the system.
- **Entity type**: collection of entities that share common properties or characteristics
- **Entity instance**: single occurrence of an entity type

(*) The terms *Entity* and *Entity Type* are often used interchangeably

Basic E-R notations



Attributes

- **Attribute**: a named property or characteristic of an entity that is of interest to the organization
  - e.g. Vehicle_ID
- An attribute name is a noun and should be unique.
- To make an attribute name unique and for clarity, each attribute name should follow a standard format.
- Similar attributes of different entity types should use similar but distinguishing names.

Candidate Keys and Identifiers

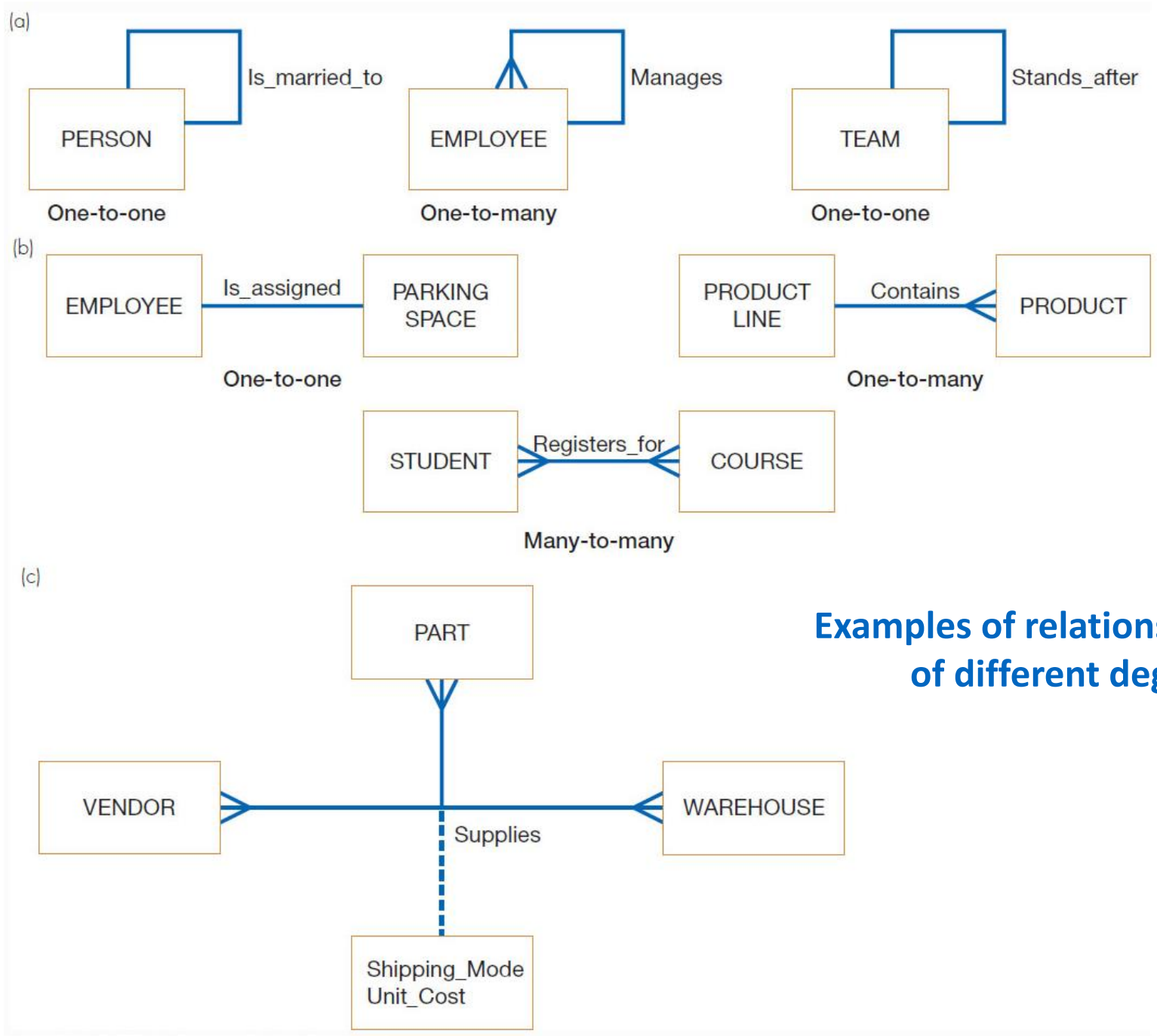
- **Candidate key**: an attribute (or combination of attributes) that uniquely identifies each instance of an entity type
- **Identifier**: a candidate key that has been selected as the unique, identifying characteristic for an entity type

Relationships

- **Relationship**: an association between the instances of one or more entity types that is of interest to the organization
- **Degree**: the number of entity types that participate in a relationship

Relationships

- **Unary relationship**: a relationship between the instances of one entity type
  - Also called a *recursive relationship*
- **Binary relationship**: a relationship between instances of two entity types
  - Most common type of relationship encountered in data modeling
- **Ternary relationship**: a simultaneous relationship among instances of three entity types



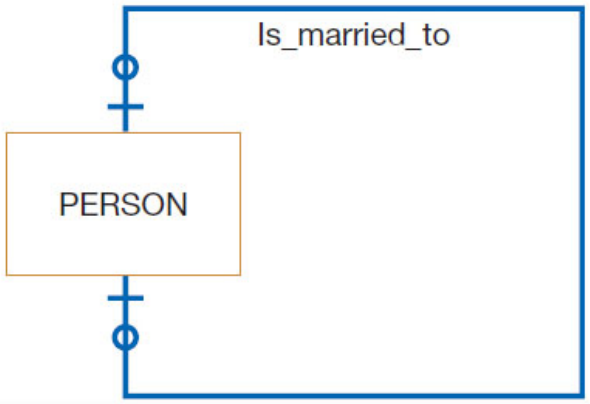
Examples of relationships of different degrees

Cardinalities

- **Cardinality**: the number of instances of entity B that can (or must) be associated with each instance of entity A
- Minimum Cardinality
  - The minimum number of instances of entity B that may be associated with each instance of entity A
- Maximum Cardinality
  - The maximum number of instances of entity B that may be associated with each instance of entity A

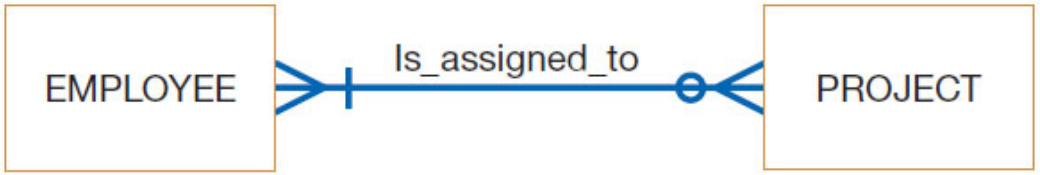
Cardinalities

- Mandatory vs. Optional Cardinalities
  - Specifies whether an instance must exist or can be absent in the relationship



Optional cardinalities

Mandatory cardinalities

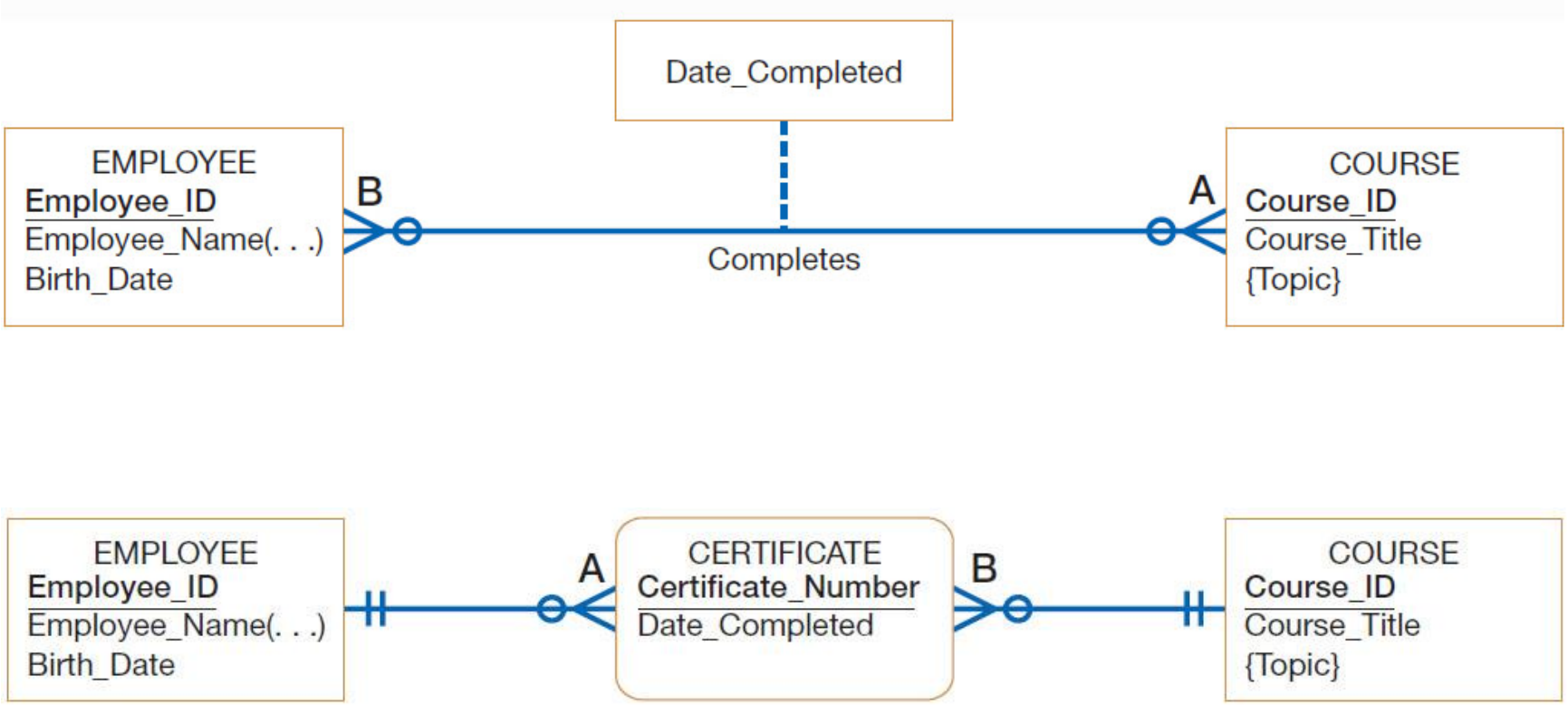


One mandatory, one optional

Associative Entities

- **Associative Entity**: an entity type that associates the instances of one or more entity types and contains attributes that are peculiar to the relationship between those entity instances
  - Sometimes called a gerund
- The data modeler chooses to model the relationship as an entity type.

Associative Entities

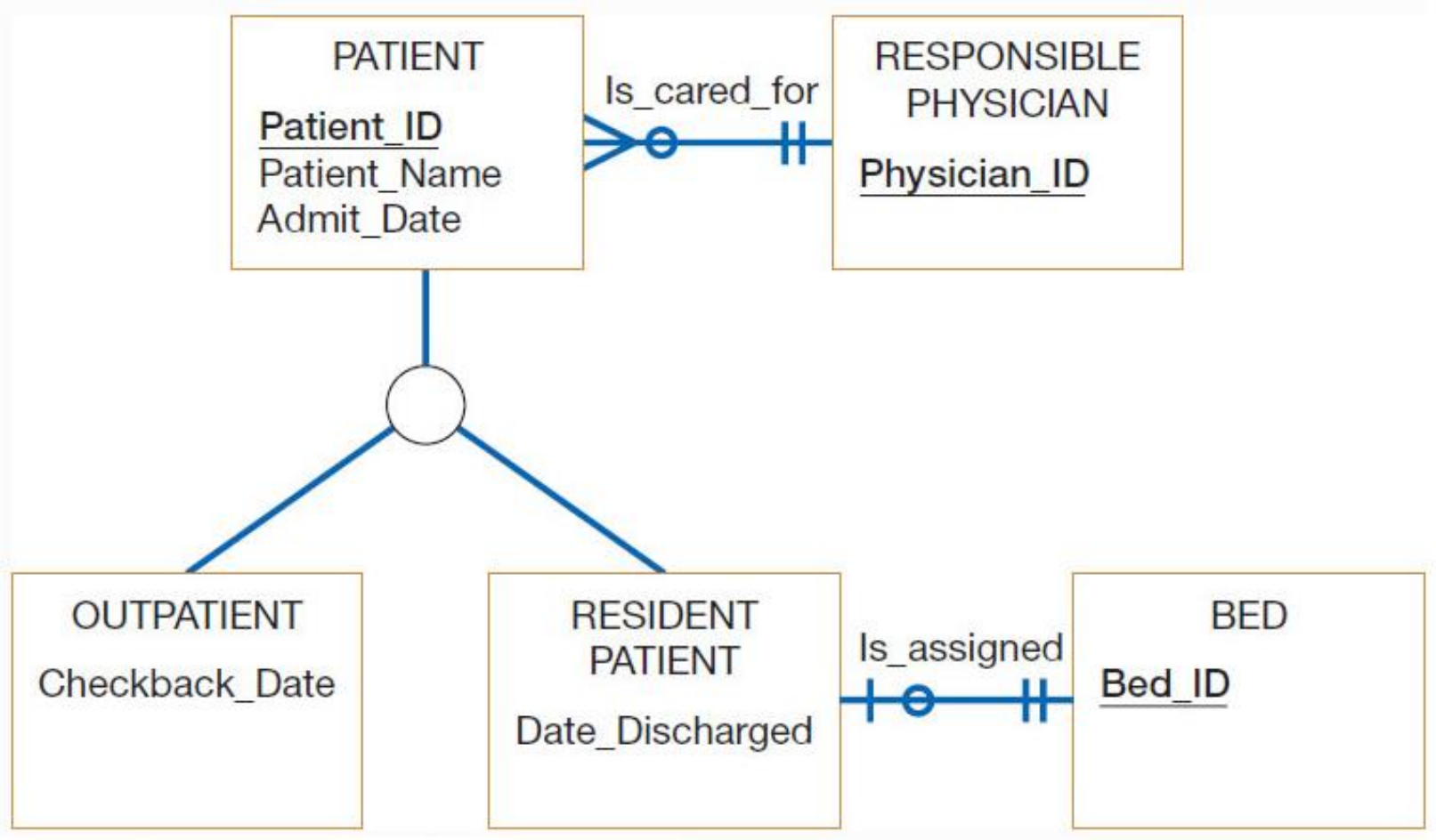


EXTENDED E-R DIAGRAM

Supertypes and subtypes

- **Subtype**: a subgrouping of the entities in an entity type
  - Is meaningful to the organization
  - Shares common attributes or relationships distinct from other subgroupings
- **Supertype**: a generic entity type that has a relationship with one or more subtypes

Supertype/subtype relationships in a hospital



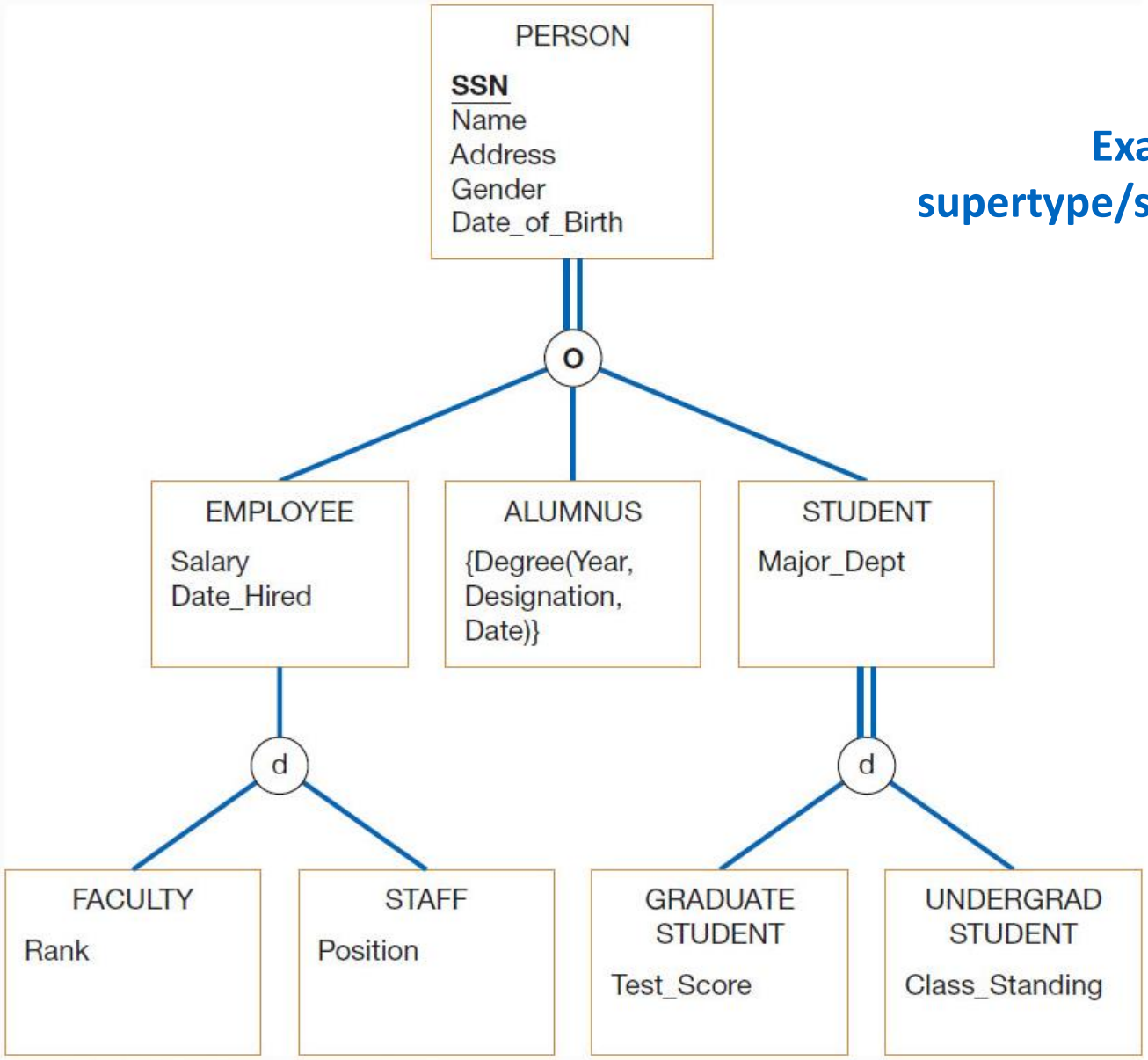
Supertypes and subtypes

- **Total specialization** specifies that each entity instance of the supertype must be a member of some subtype in the relationship.
  - Represented by double lines.
- **Partial specialization** specifies that an entity instance of the supertype does not have to belong to any subtype, and may or may not be an instance of one of the subtypes.
  - Represented by single lines.

Supertypes and subtypes

- **Disjoint rule** specifies that if an entity instance of the supertype is a member of one subtype, it cannot simultaneously be a member of any other subtype.
  - Represented with “d”.
- **Overlap rule** specifies that an entity instance can simultaneously be a member of two (or more) subtypes.
  - Represented with “o”.

Example of supertype/subtype hierarchy



Part 3

HIBERNATE ENTITIES

Annotations & Relationships

Hibernate Entity

- Annotated with `@Entity`
- Must have a *primary key* field with `@Id` annotation
- Can add a `@Table` annotation to specify the mapped table's name
  - Automatically use entity name as table name
- Can add a `@Column` annotation to specify the mapped column
  - Automatically use attribute name as column name

Primary Key Auto-Generation

- For the Entity's primary key, a `@GeneratedValue` annotation is often attached
  - `GenerationType.IDENTITY`: primary key value generated by the database system
  - `GenerationType.AUTO`: lets database system select appropriate generation method
  - `GenerationType.TABLE`: uses a separate database table to generate primary key values
  - `GenerationType.SEQUENCE`: uses a database sequence to generate primary key values

One-To-Many Association

- We can attach `@OneToMany` and `@ManyToOne` annotation to an Entity attribute

```
@Entity
public class Album {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @ManyToOne
    private Singer author;
}
```

One-To-Many Association

- Choosing `@ManyToOne` or `@OneToMany` depends on the nature of the association
  - In this case, one singer owns many albums

```
@Entity
public class Singer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @OneToMany
    List<Album> albums;
}
```

The Owning Side

- It's a good practice to mark the *many-to-one* side as the *owning side*.
  - To ensure data consistency
  - Using the `mappedBy` property *on the inverse side*

```
@Entity
public class Singer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @OneToMany(mappedBy = "author")
    List<Album> albums;
}
```

Many-To-Many Association

- Just attach `@ManyToMany` annotations to entity attributes

```
@Entity
public class Student {
    @Id
    private Integer id;
    @ManyToMany
    List<Course> likedCourses;
}
```

```
@Entity
class Course {
    @Id
    private Integer id;
    @ManyToMany
    private List<Student> likes;
}
```

Configure Many-To-Many mapping

- Specify the join table's name and column names
  - If you want to enforce your naming convention.

```
@ManyToMany
@JoinTable(
        name = "course_like",
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "course_id"))
List<Course> likedCourses;
```