Object-Oriented Software Design

Organization of this Lecture

- Brief review of last lecture
- Introduction to object-oriented concepts
- Object modelling using Unified Modelling Language (UML)
- Object-oriented software development and patterns
- Summary

Review of last lecture

- Last lecture we started
  - with an introduction to function-oriented design.
- We looked at goals of structured analysis (SA)
- Result of SA (i.e. DFD)

Review of last lecture

- We looked at DFD modelling techniques
- We looked at importance of Data Dictionary

Review of last lecture

- We looked at balancing a DFD
- During structured design (SD), transformation of DFD to structure chart
- Also we discussed few examples of structured analysis & structured design

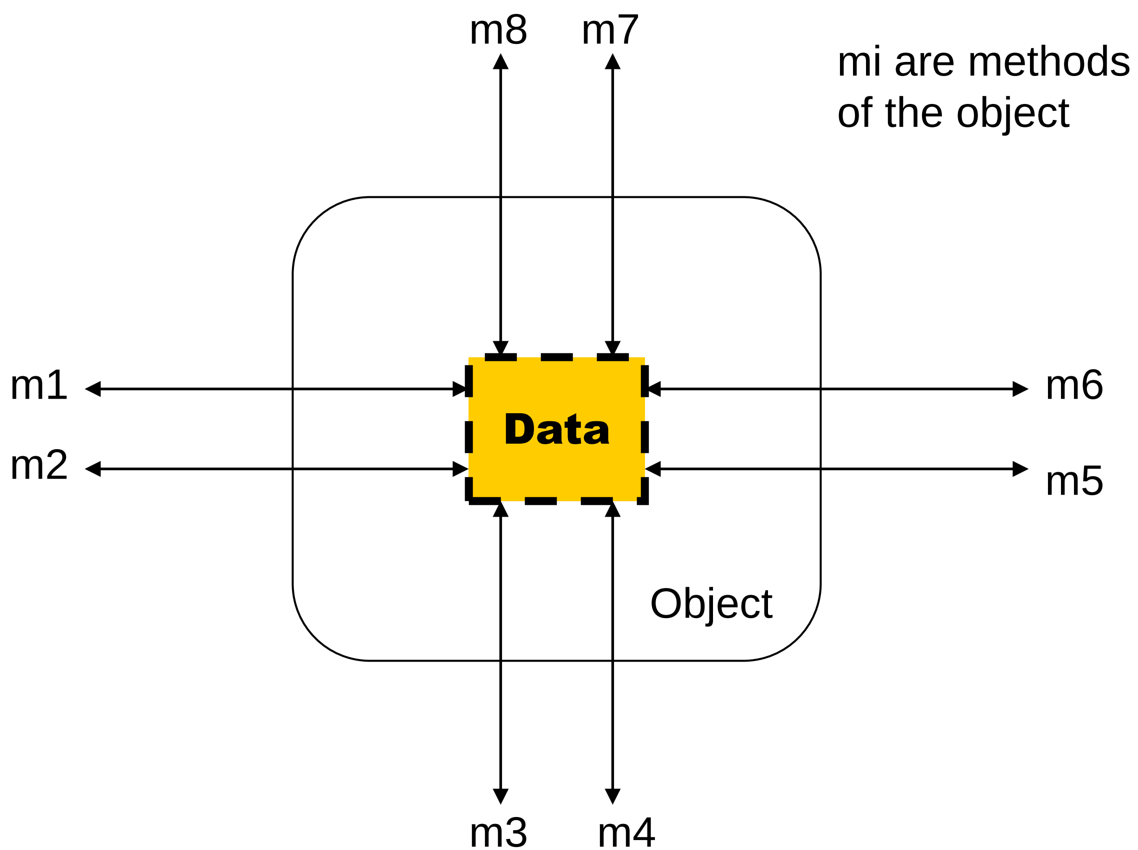
Object-oriented concepts

- Object-oriented (OO) design techniques are becoming popular:
  - Inception in early 1980 and nearing maturity.
  - Widespread acceptance in industry and academics
  - Unified Modelling Language (UML) poised to become a standard for modelling OO systems

Object-oriented concepts

- Basic Mechanisms:
  - Objects:
    - A real-world entity.
    - A system is designed as a set of interacting objects.
    - Consists of data (attributes) and functions (methods) that operate on data
    - Hides organization of internal information (Data abstraction)
    - Examples: an employee, a book etc.

Object-oriented concepts



Model of an object

Object-oriented concepts

- **Class:**
  - Similar objects
  - Template for object creation
  - Sometimes not intended to produce instances (abstract classes)
  - Considered as abstract data type (ADT)
  - Examples: set of all employees, different types of book

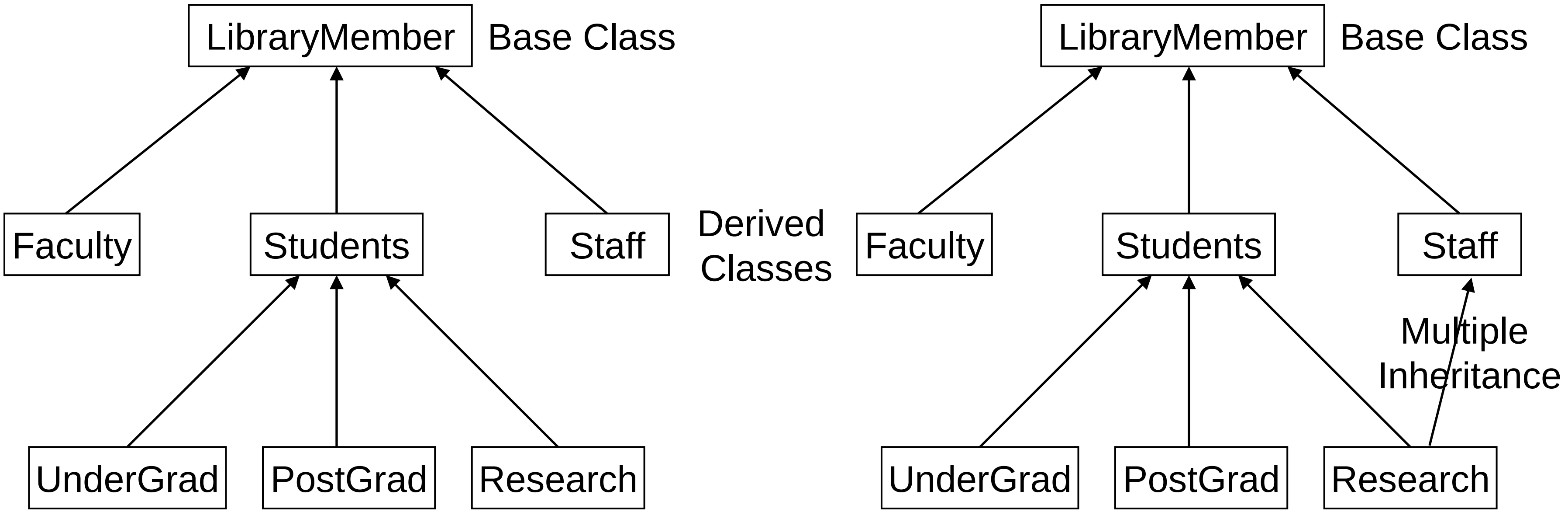
Object-oriented concepts

- Methods and message:
  - Operations supported by an object
  - Means for manipulating the data of other objects
  - Invoked by sending message
  - Examples: calculate_salary, issue-book, member_details, etc.

Object-oriented concepts

- **Inheritance:**
  - Allows to define a new class (**derived class**) by extending or modifying existing class (**base class**) or classes (**multiple inheritance**)
  - Represents **Generalization-specialization** relationship

Object-oriented concepts



Object-oriented concepts

- Key Concepts:
  - **Abstraction:**
    - Consider aspects relevant for certain purpose
    - Suppress non-relevant aspects
    - Supported at two levels i.e. **class level** where **base class is an abstraction** & **object level** where **object is a data abstraction entity**

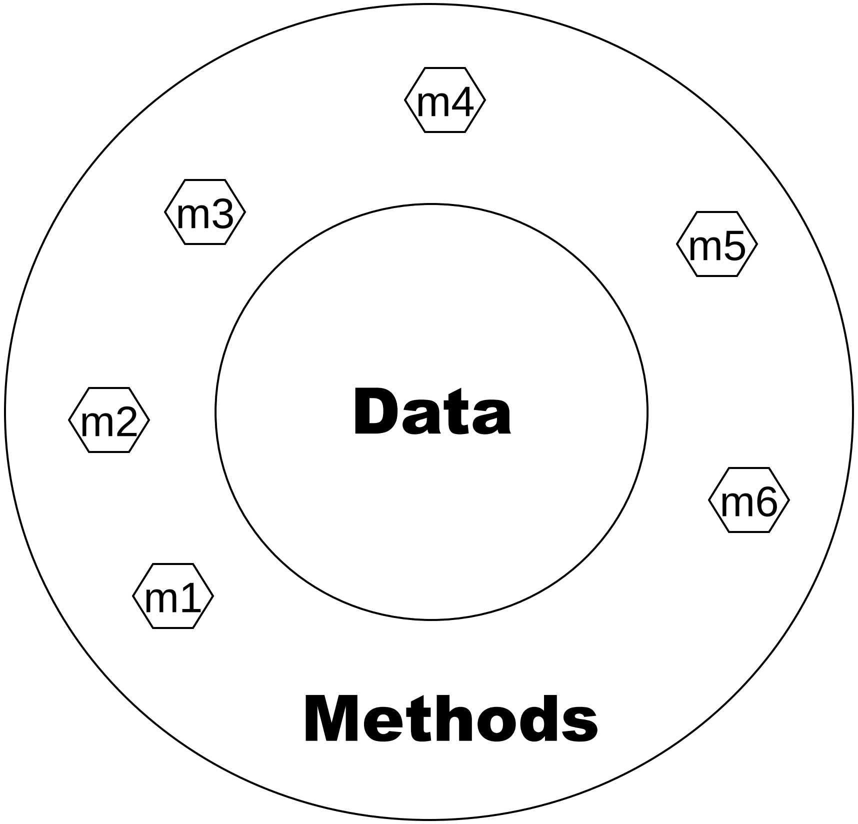
Object-oriented concepts

- **Advantages of abstraction:**
  - **Reduces complexity of software**
  - **Increases software productivity**
- **It is shown that software productivity is inversely proportional to software complexity**

Object-oriented concepts

- **Encapsulation:**
  - **Objects property to communicate outside world through messages**
  - **Objects data encapsulated within its methods**

Object-oriented concepts



Concept of encapsulation

Object-oriented concepts

- **Polymorphism:**
  - **Refers to poly (many) morphism (forms)**
  - **Same message result in different actions at different objects (static binding)**

Object-oriented concepts

- Example of static binding:
  - Class Circle{
  - private float x, y, radius;
  - private int fillType;
  -
  - public create ();
  - public create (float x, float y, float centre);
  - public create (float x, float y, float centre, int fillType);
  - }

Object-oriented concepts

- In this example:
  - A class named **Circle** has three definitions for **create** operation
  - Without any parameter, default
  - Centre and radius as parameter

Object-oriented concepts

- Centre, radius and filltype as parameter
- Depending upon parameters given, method will be invoked
- Method create is overloaded

Object-oriented concepts

- **Dynamic binding**:
  - In inheritance hierarchy, object can be assigned to another object of its ancestor class
  - A method call to ancestor object would result in the invocation of appropriate method of object of the derived class

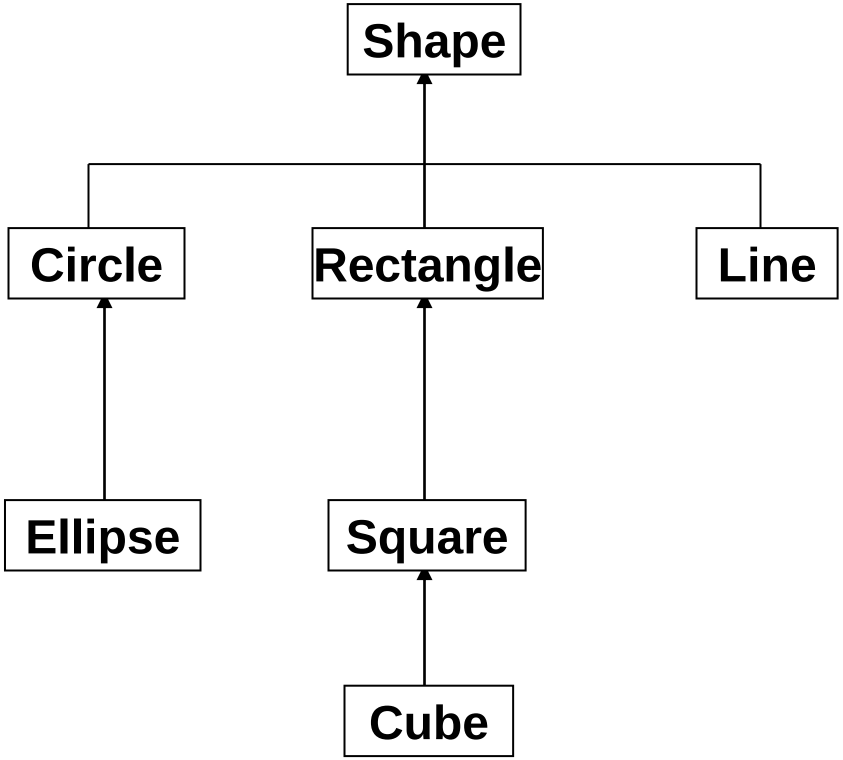
Object-oriented concepts

- **Dynamic binding:**
  - **Exact method cannot be known at compile time**
  - **Dynamically decided at runtime**

Object-oriented concepts

- Example of dynamic binding:
  - Consider class hierarchy of different geometric objects
  - Now display method is declared in the shape class and overridden in each derived class
  - A single call to the display method for each object would take care displaying appropriate element

Object-oriented concepts



Class hierarchy of geometric objects

Object-oriented concepts

Traditional code	Object-oriented code

If (shape == Circle) then
draw_circle();
else if (shape == Rectangle) then
draw_rectangle();
 	
-
-
-
-
	Shape.draw();
	
	-
	-
	-
	-

Traditional code and OO code using dynamic binding

Object-oriented concepts

- Advantages of dynamic binding:
  - Leads to elegant programming
  - Facilitates code reuse and maintenance
  - New objects can be added with minimal changes to existing objects

Object-oriented concepts

- Composite objects:
  - Object containing other objects
  - Composition limited to tree hierarchy i.e. no circular inclusion relation
  - Inheritance hierarchy different from containment hierarchy

Object-oriented concepts

- Composite objects:
  - Inheritance hierarchy, different object types with similar features
  - Containment allows construction of complex objects

Object-oriented concepts

- Genericity:
  - Ability to parameterize class definitions
  - Example: class stack of different types of elements such as integer, character and floating point stack
  - Generacity permits to define generic class stack and later instantiate as required

Advantages of Object-oriented design

- Code and design reuse
- Increased productivity
- Ease of testing & maintenance
- Better understandability
- Its agreed that increased productivity is chief advantage

Advantages of Object-oriented design

- Initially incur higher costs, but after completion of some projects reduction in cost become possible

- Well-established OO methodology and environment can be managed with 20-50% of traditional cost of development

Object modelling using UML

- **UML** is a modelling language
- Not a system design or development methodology
- Used to document object-oriented analysis and design
- Independent of any specific design methodology

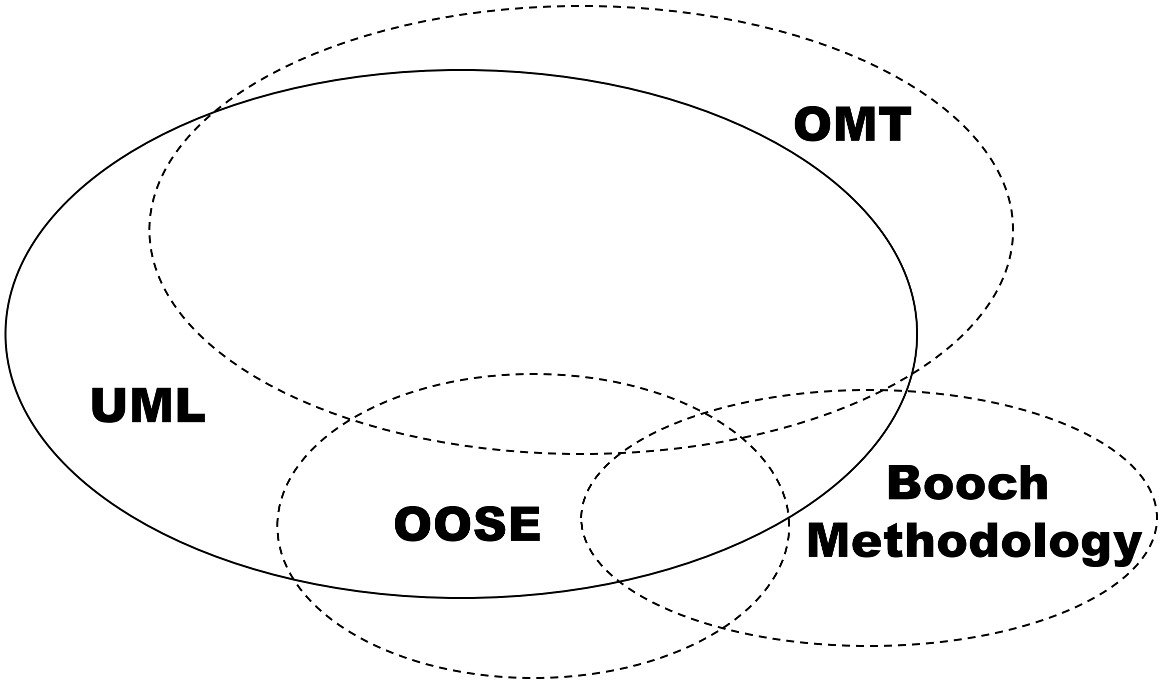
Unified Modelling Language (UML)

- Origin
  - In late 1980s and early 1990s different software development houses were using different notations
  - Developed in early 1990s to standardize the large number of object-oriented modelling notations

UML

- Principal methodologies used
  - **OMT** [Rumbaugh 1991]
  - **Booch's methodology**[Booch 1991]
  - **OOSE** [Jacobson 1992]
  - **Odell's methodology**[Odell 1992]
  - **Shlaer and Mellor** [Shlaer 1992]

UML



Different object modelling techniques in UML

UML

- As a Standard
  - Adopted by **Object Management Group (OMG)** in 1997
  - **OMG** association of industries
  - Promote consensus notations and techniques
  - Used outside software development, example **car manufacturing**

Why UML is required?

- Model is required to capture only important aspects
- UML a graphical modelling tool, easy to understand and construct
- Helps in managing complexity

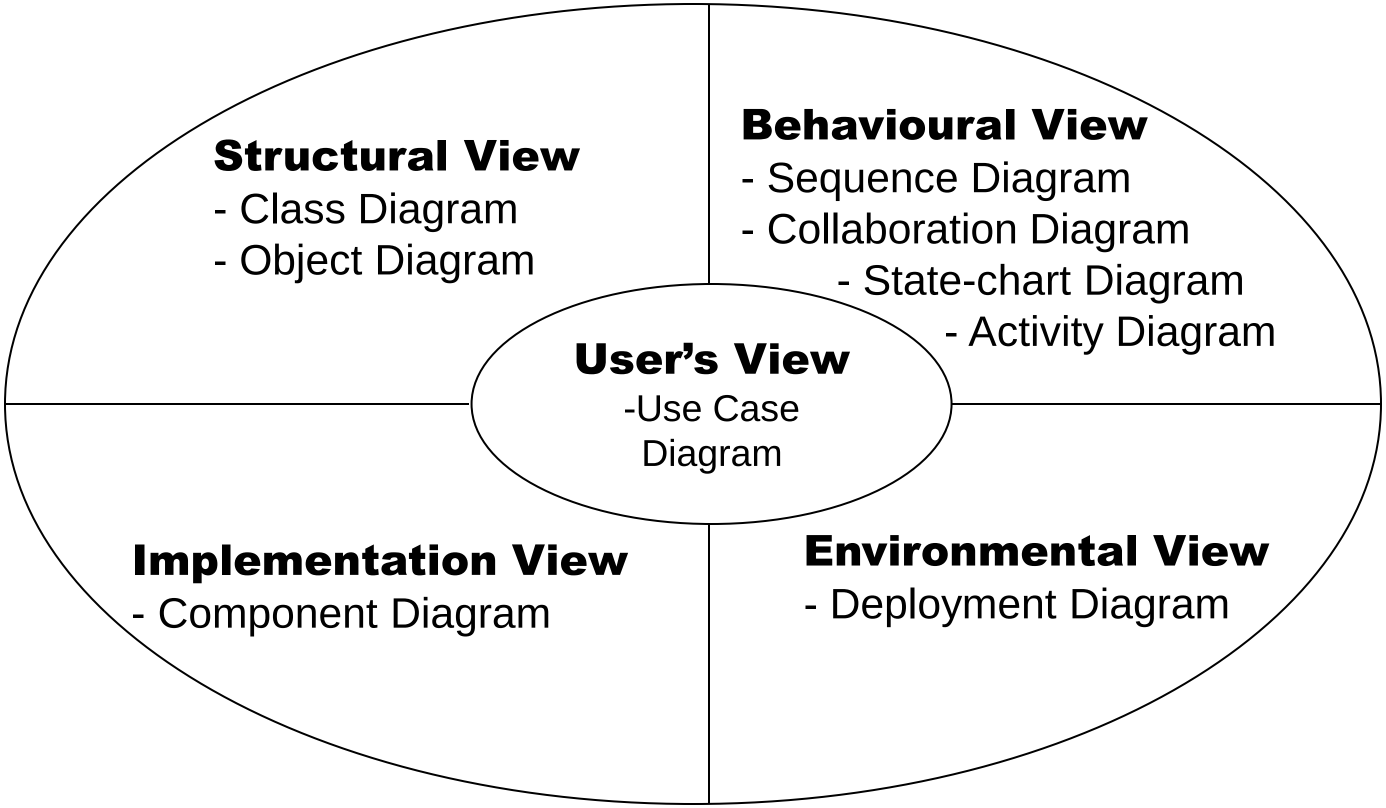
UML diagrams

- Nine diagrams to capture different views of a system
- Provide different perspectives of the software system
- Diagrams can be refined to get the actual implementation of the system

UML diagrams

- Views of a system
  - **User's** view
  - **Structural** view
  - **Behavioral** view
  - **Implementation** view
  - **Environmental** view

UML diagrams



Diagrams and views in UML

All views required?

- NO
  - Use case model, class diagram and one of the interaction diagram for a simple system
  - State chart diagram in case of many state changes
  - Deployment diagram in case of large number of hardware components

USE CASE model

- Consists of set of “**use cases**”
- An important analysis and design artifact
- Other models must confirm to this model
- Not really an object-oriented model
- Represents a functional or process model

USE CASES

- Different ways in which system can be used by the users
- Corresponds to the high-level requirements
- Represents transaction between the user and the system
- Define behavior without revealing internal structure of system
- Set of related scenarios tied together by a common goal

USE CASES

- Normally, use cases are independent of each other
- Implicit dependencies may exist
- **Example**: In Library Automation System, renew-book & reserve-book are independent use cases. But in actual implementation of renew-book, a check is made to see if any book has been reserved using reserve-book

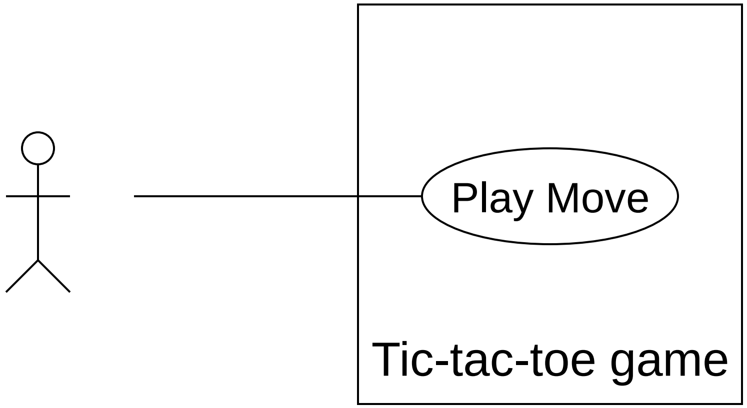
Example of USE CASES

- For library information system
  - issue-book
  - Query-book
  - Return-book
  - Create-member
  - Add-book, etc.

Representation of USE CASES

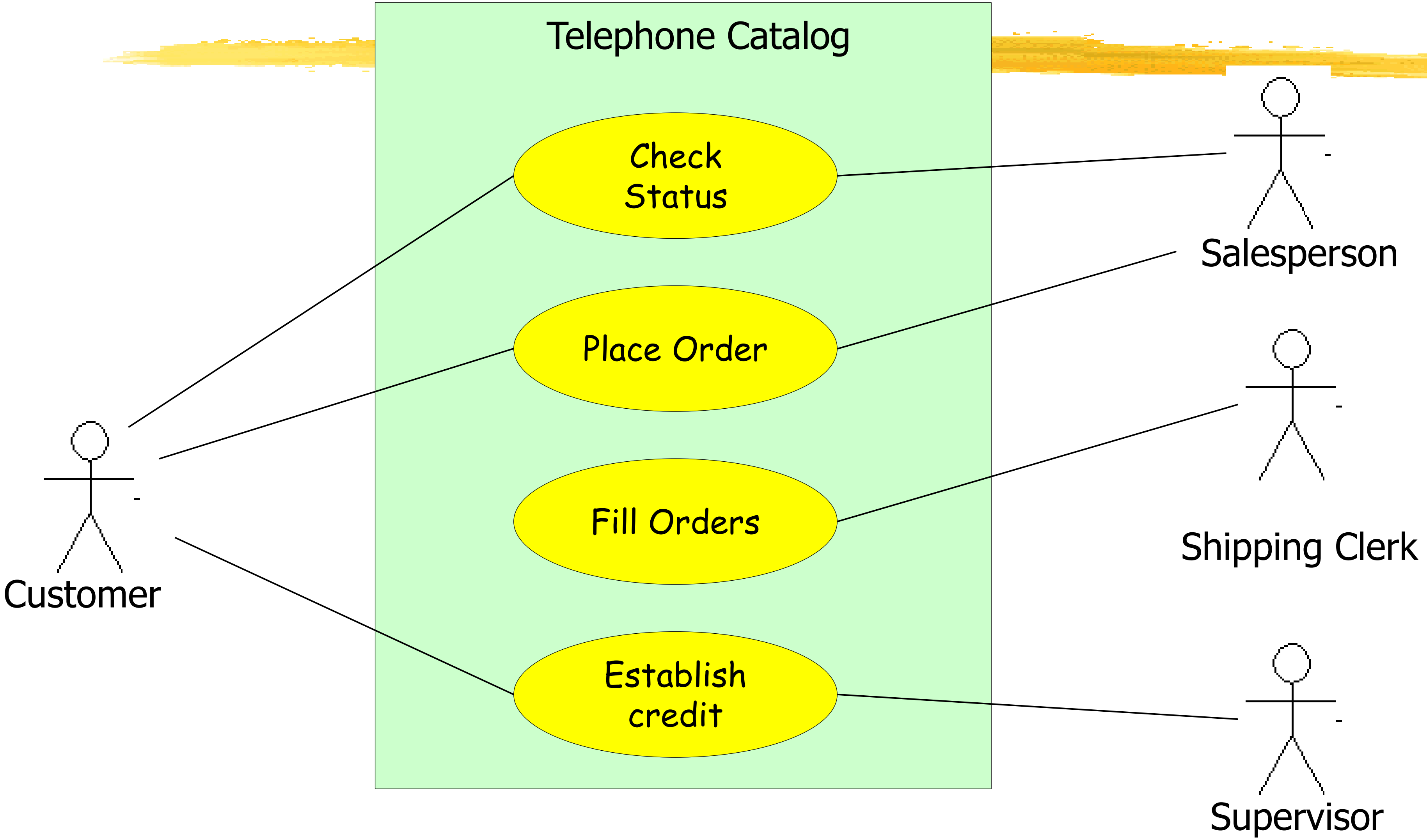
- Represented by use case diagram
- **Use case** is represented by **ellipse**
- **System boundary** is represented by **rectangle**
- **Users** are represented by **stick person** icon (**actor**)
- **Communication relationship** between actor and use case by **line**
- **External system** by **stereotype**

Example of USE CASES



Use case model

Yet Another Use Case Example



Why develop USE CASE diagram?

- Serves as requirements specification
- Users identification helps in implementing security mechanism through **login system**
- Another use in preparing the documents (e.g. **user's manual**)

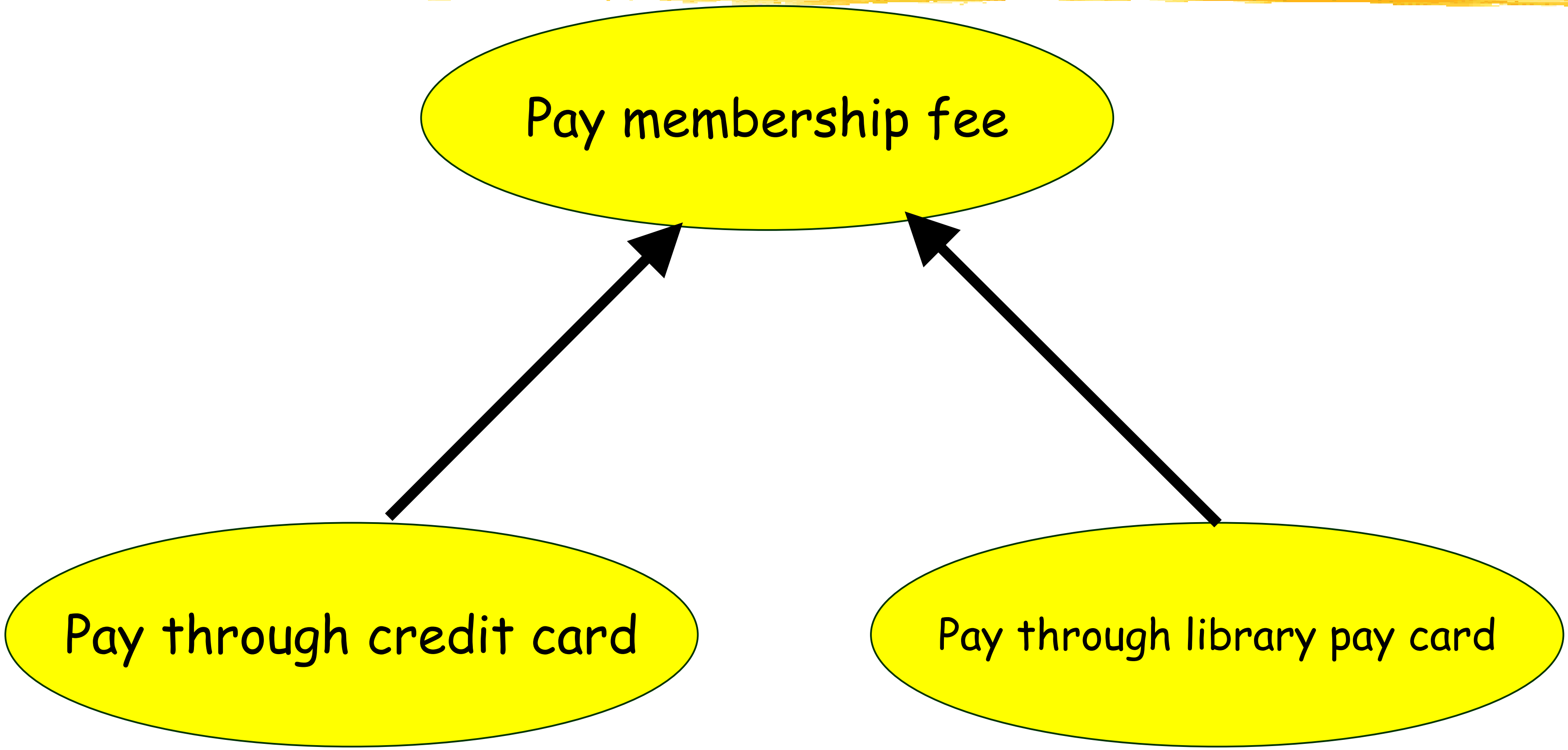
Use Case Description: Change Flight

- Actors: traveler, airline reservation system
- Preconditions:
  - Traveler has logged on to the system and selected 'change flight itinerary' option
- Scenario 1: Mainline Sequence
  1. System retrieves traveler's account and flight itinerary from client account database
  2. System asks traveler to select itinerary segment she wants to change; traveler selects itinerary segment.
  3. System asks traveler for new departure and destination information; traveler provides information.
  4. If flights are available then…
  5. …
  6. System displays transaction summary.
- Scenario 2: Alternative Sequence
  4. If no flights are available then …

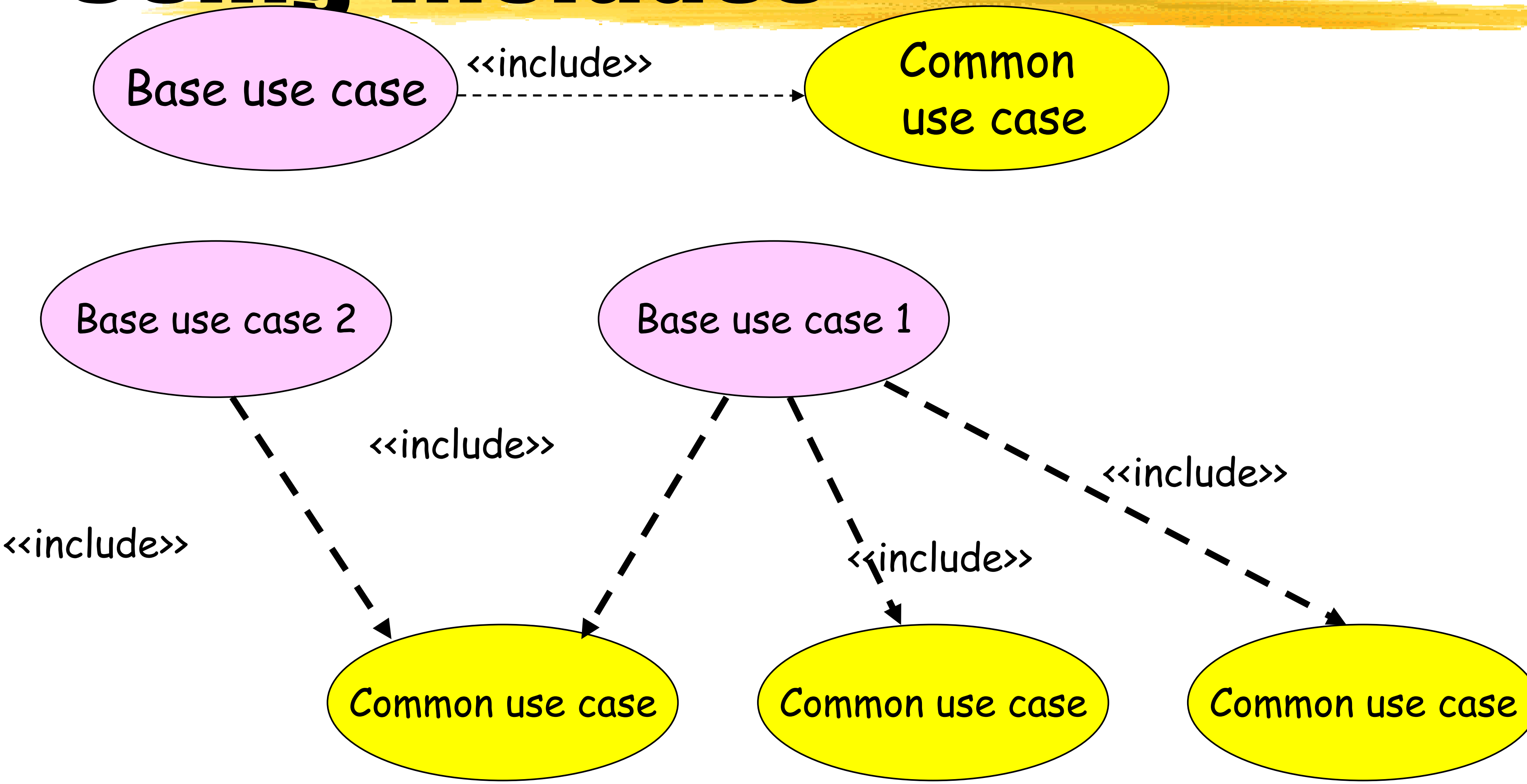
Factoring Use Cases

- Two main reasons for factoring:
  - Complex use cases need to be factored into simpler use cases
  - To represent common behavior across different use cases
- Three ways of factoring:
  - Generalization
  - Includes
  - Extends

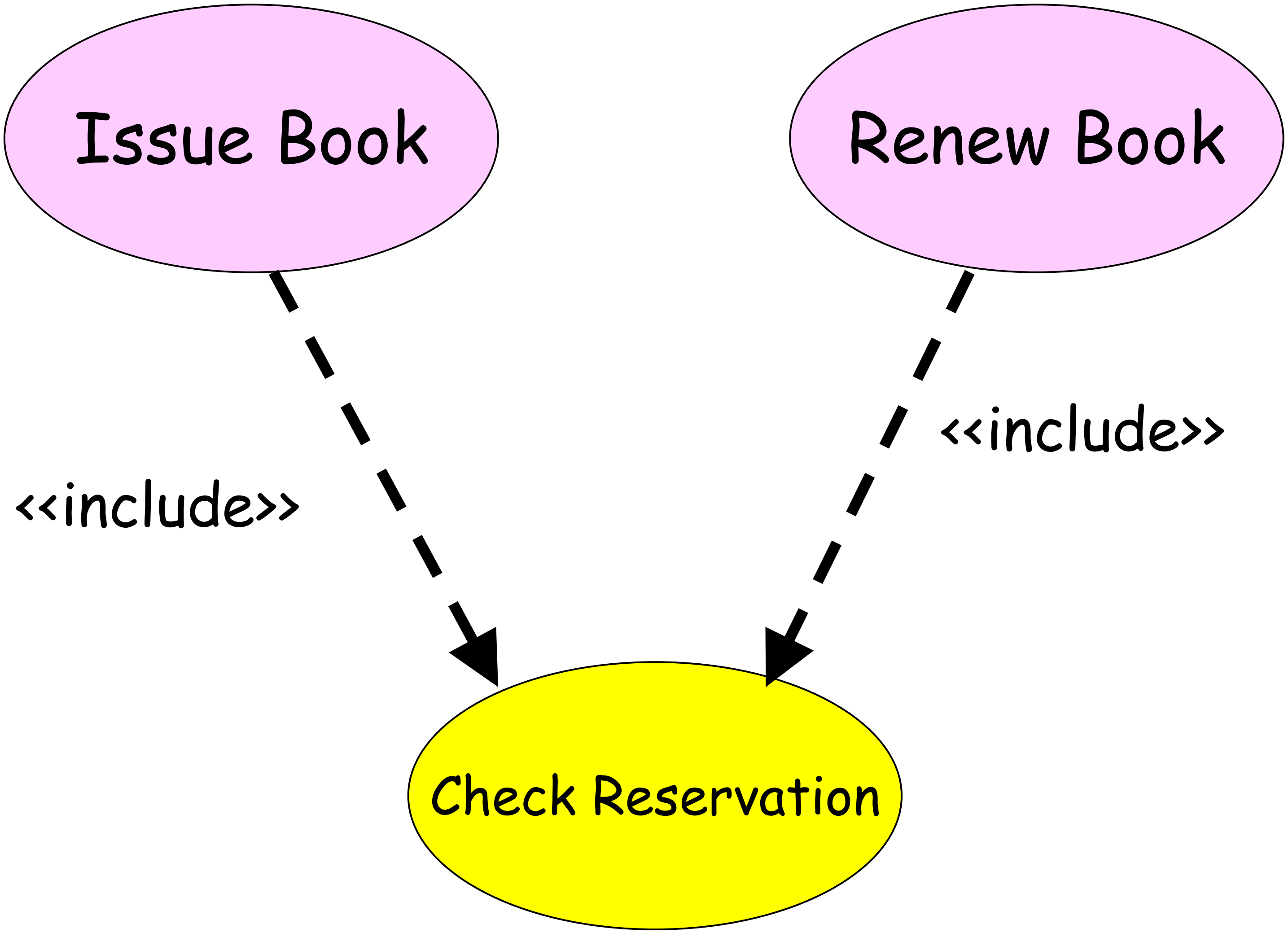
Factoring Use Cases Using Generalization



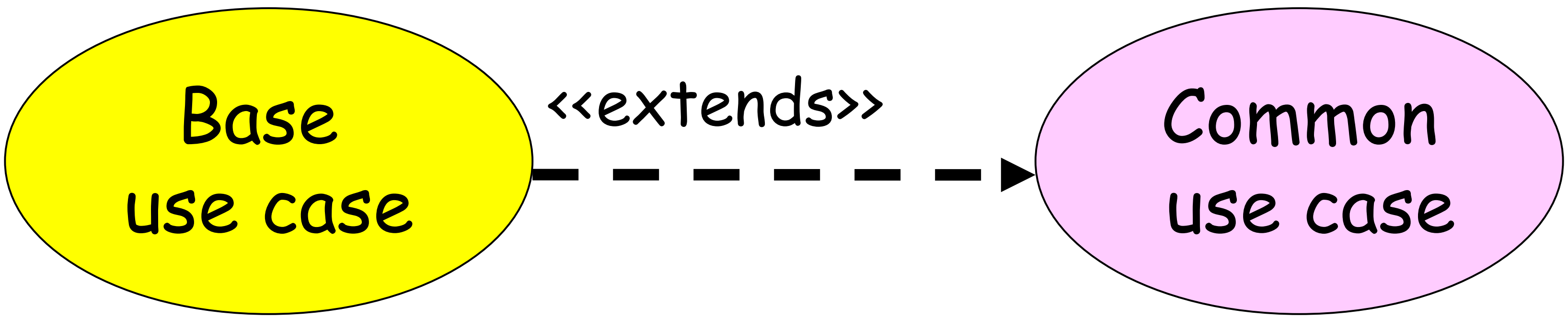
Factoring Use Cases Using Includes



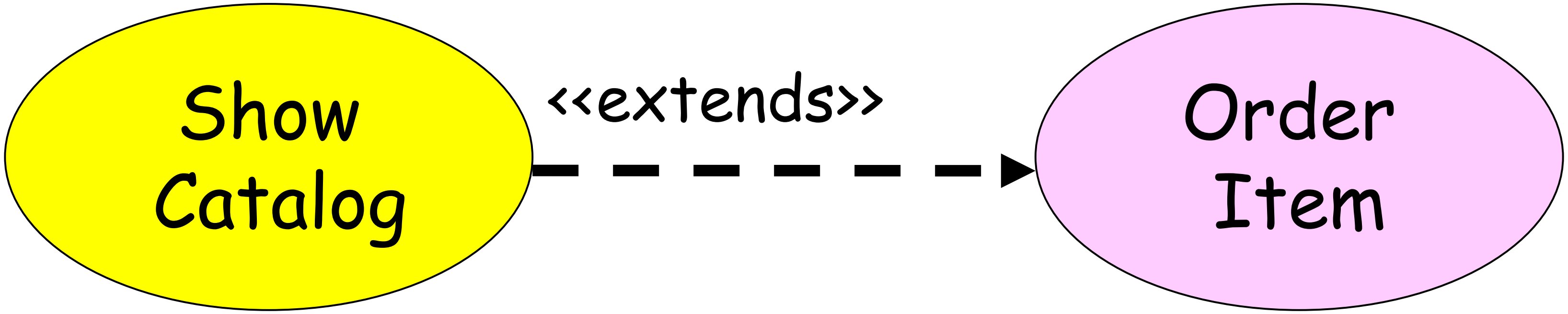
Example of Factoring Use Cases Using Includes



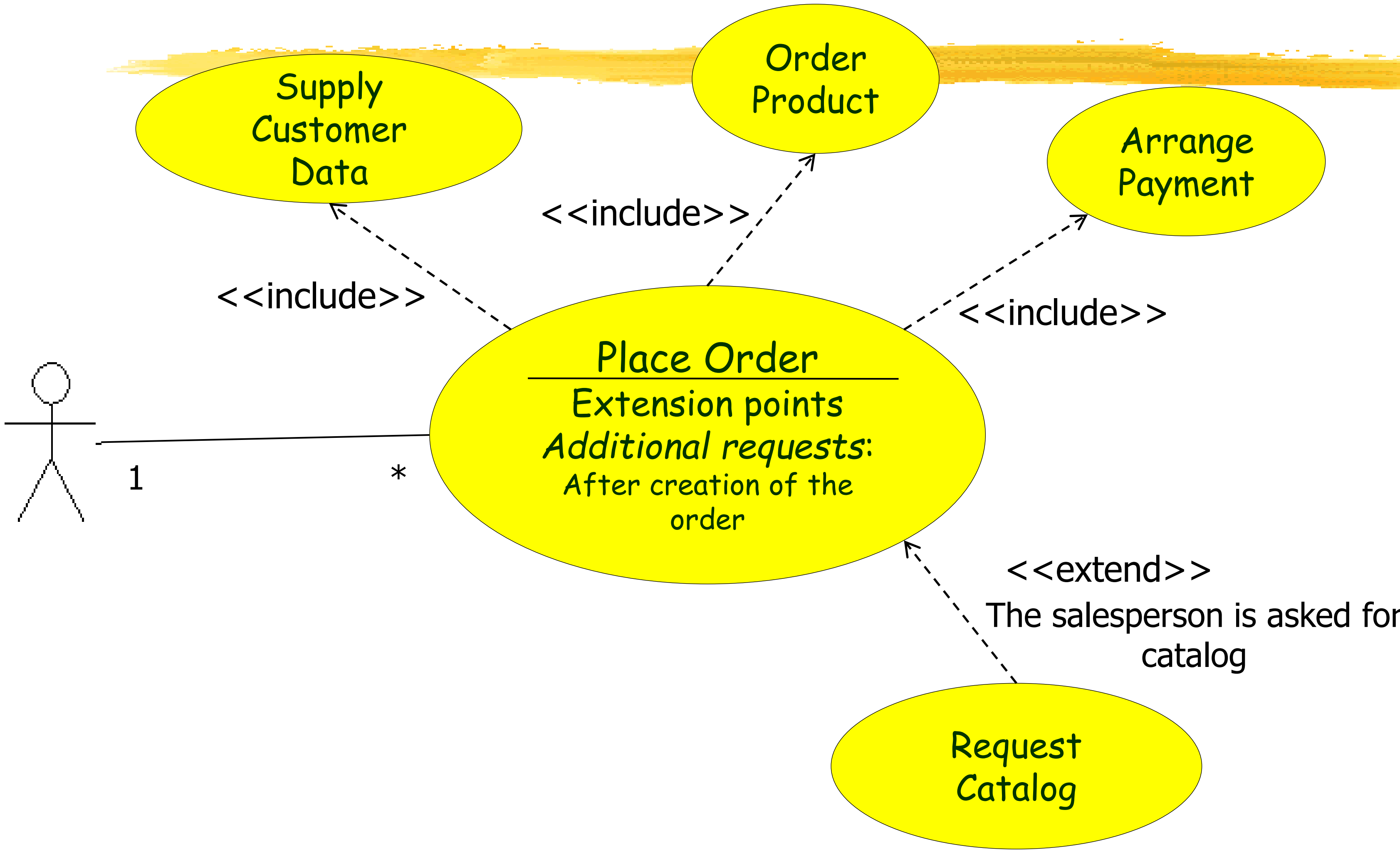
Factoring Use Cases Using Extends



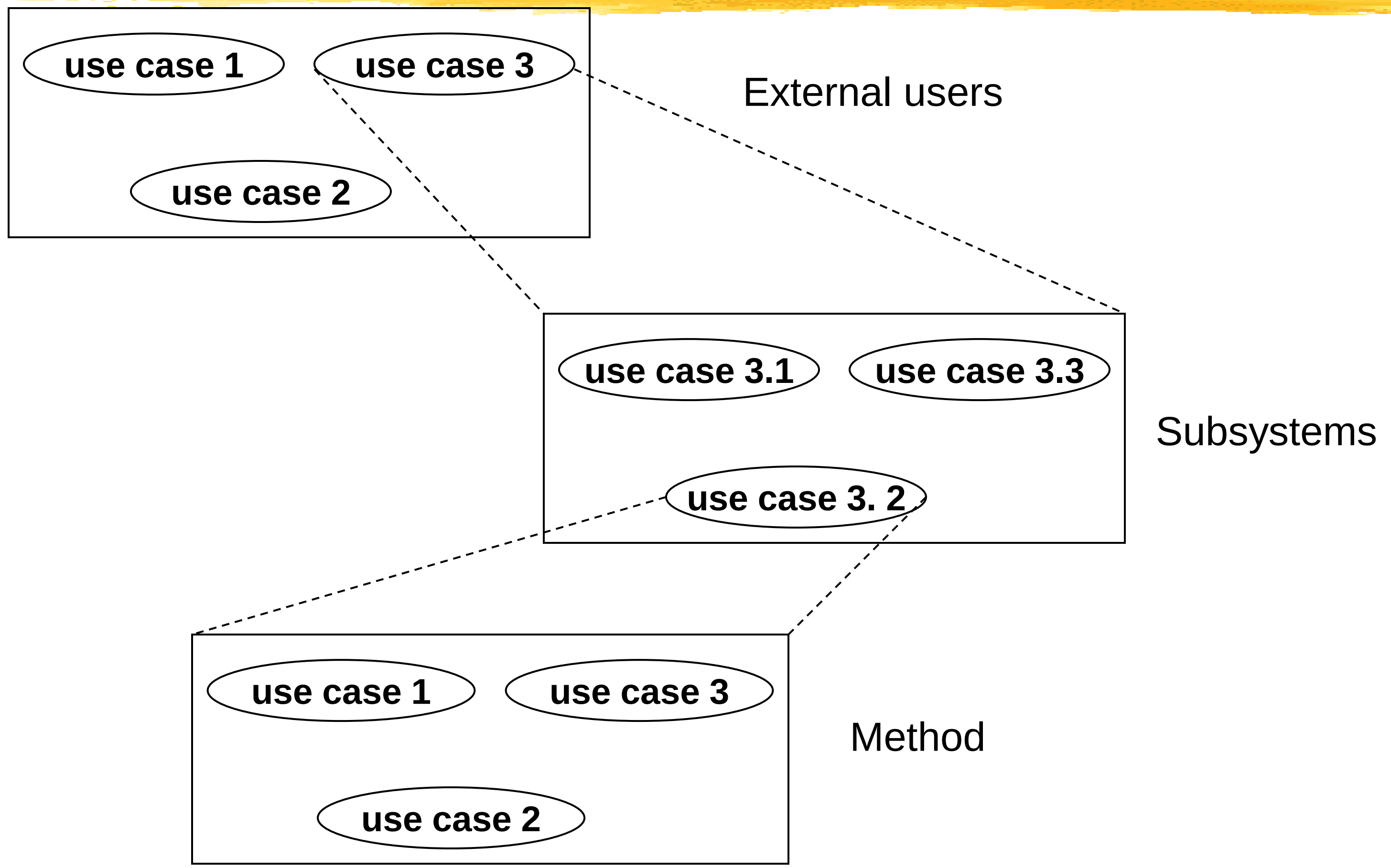
Factoring Use Cases Using Extends



Use Case Relationships

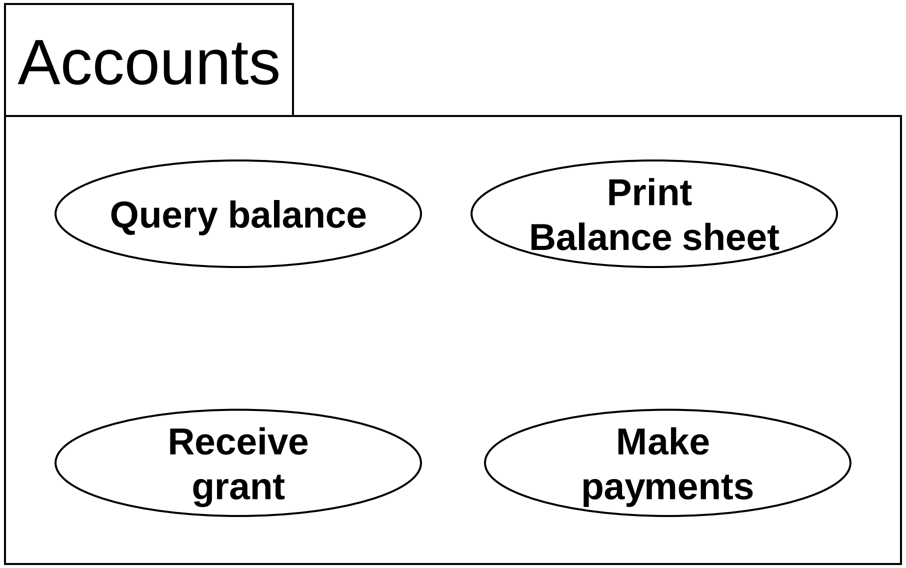


Hierarchical organization of USE CASES



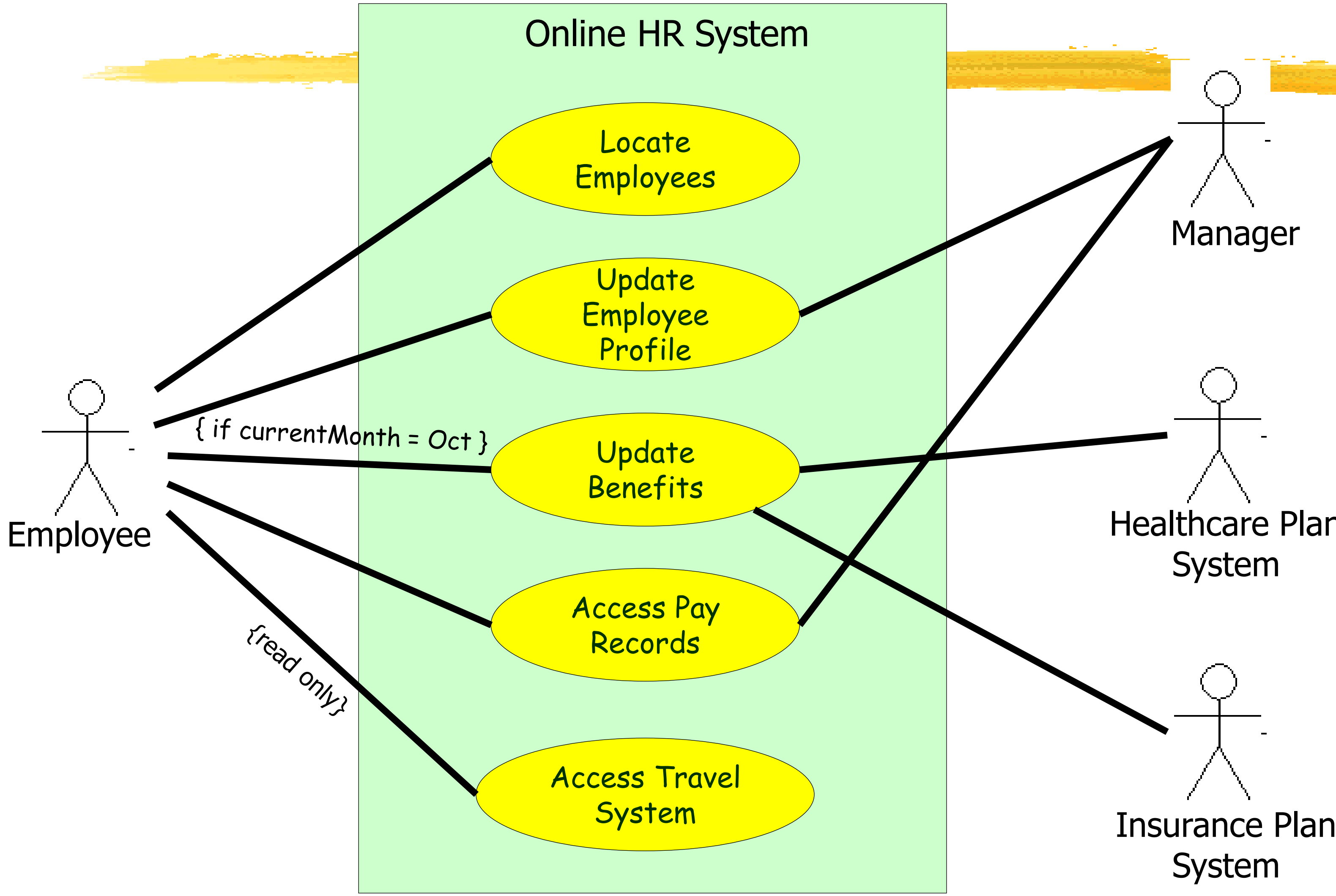
Hierarchical organization of use cases

USE CASE packaging

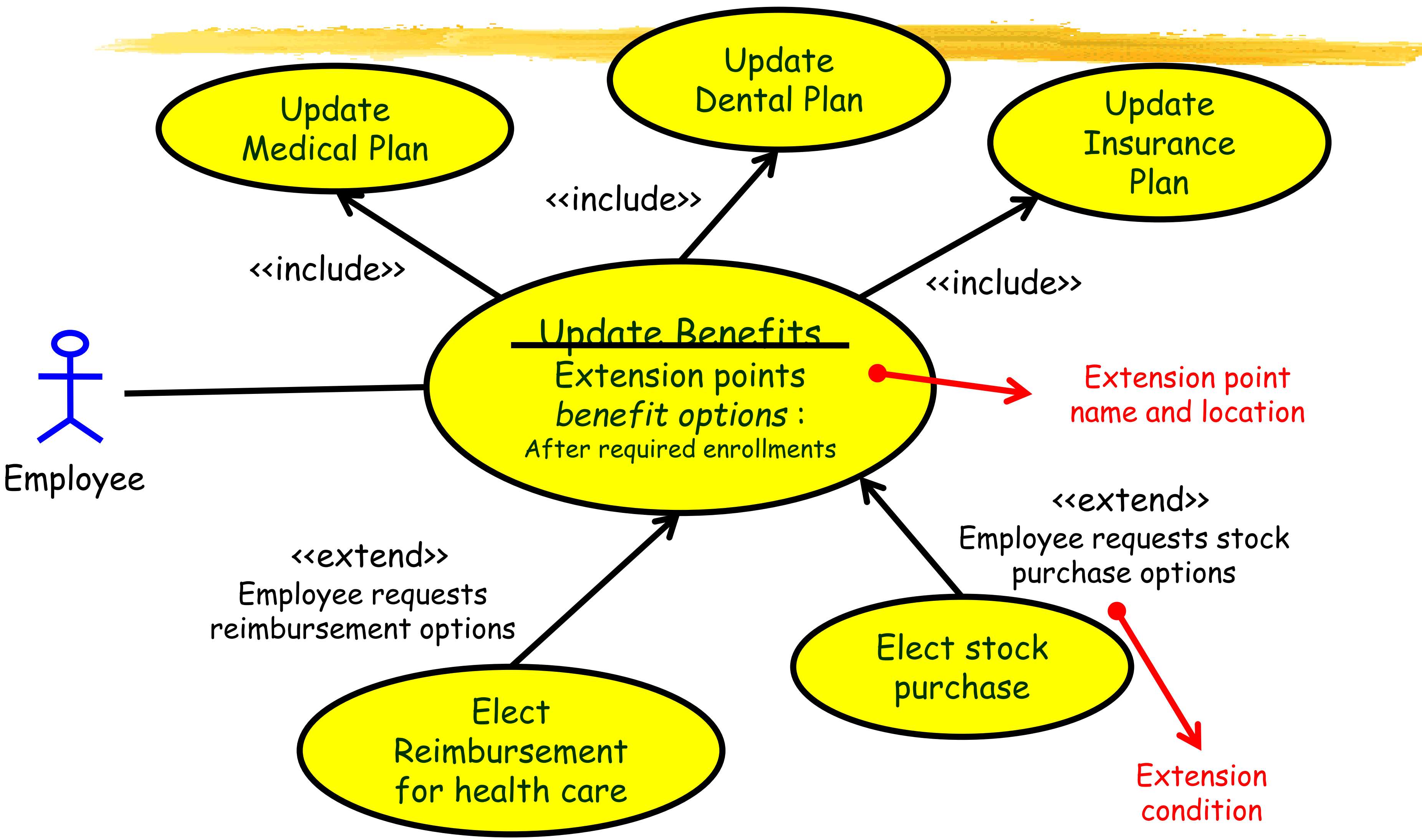


Use case packaging

Example: Online HR System



Online HR System: Use Case Relationships



Online HR System: Update Benefits Use Case

- Actors: employee, employee account db, healthcare plan system, insurance plan system
- Preconditions:
  - Employee has logged on to the system and selected 'update benefits' option
- Scenario 1: Mainline Sequence
  - 1.System retrieves employee account from employee account db
  - 2.System asks employee to select medical plan type; include Update Medical Plan.
  - 3. System asks employee to select dental plan type; include Update Dental Plan.
  - …
- Scenario 2: Alternative Sequence
  - 2. If health plan is not available in the employee's area the employee is informed and asked to select another plan...

CLASS diagram

- Describes static structure of a system
- Main constituents are classes and their relationships:
  - Generalization
  - Aggregation
  - Association
  - Various kinds of dependencies

CLASS diagram

- Entities with common features, i.e. attributes and operations
- Classes are represented as solid outline rectangle with compartments
- Compartments for **name**, **attributes** & **operations**
- Attribute and operation compartment are optional for reuse purpose

Example of CLASS diagram

LibraryMember
Member Name Membership Number Address Phone Number E-Mail Address Membership Admission Date Membership Expiry Date Books Issued

issueBook( );
findPendingBooks( );
findOverdueBooks( );
returnBook( );
findMembershipDetails( );

LibraryMember
Member Name Membership Number Address Phone Number E-Mail Address Membership Admission Date Membership Expiry Date Books Issued

LibraryMember

Different representations of the LibraryMember class

Class Attributes

- May specify:
  - Type (class)
  - Initial value
  - Multiplicity
- E.g., SensorStatus[10]:Int=0
- Constraints: e.g., ISBN number {sorted} for class Book

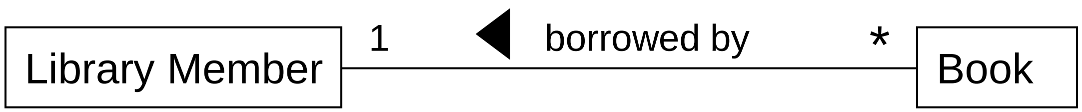
Class Operations

- Abstract operations written in italics
- May specify:
  - Parameters
  - Their kinds: in/out/inout
  - Return type
- E.g., issueBook(in BookName): Boolean
- Methods and operations are distinguishable only when there is polymorphism

ASSOCIATION relationship

- Enable objects to communicate with each other
- Usually binary but more classes can be involved
- Class can have relationship with itself (**recursive** association)
- Arrowhead used along with name, indicates direction of association
- Multiplicity indicates # of instances

ASSOCIATION relationship



Association between two classes

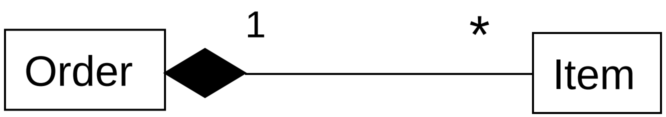
AGGREGATION relationship

- Represent a whole-part relationship
- Represented by **diamond** symbol at the composite end
- Cannot be reflexive(i.e. recursive)
- Not symmetric
- It can be transitive

AGGREGATION relationship

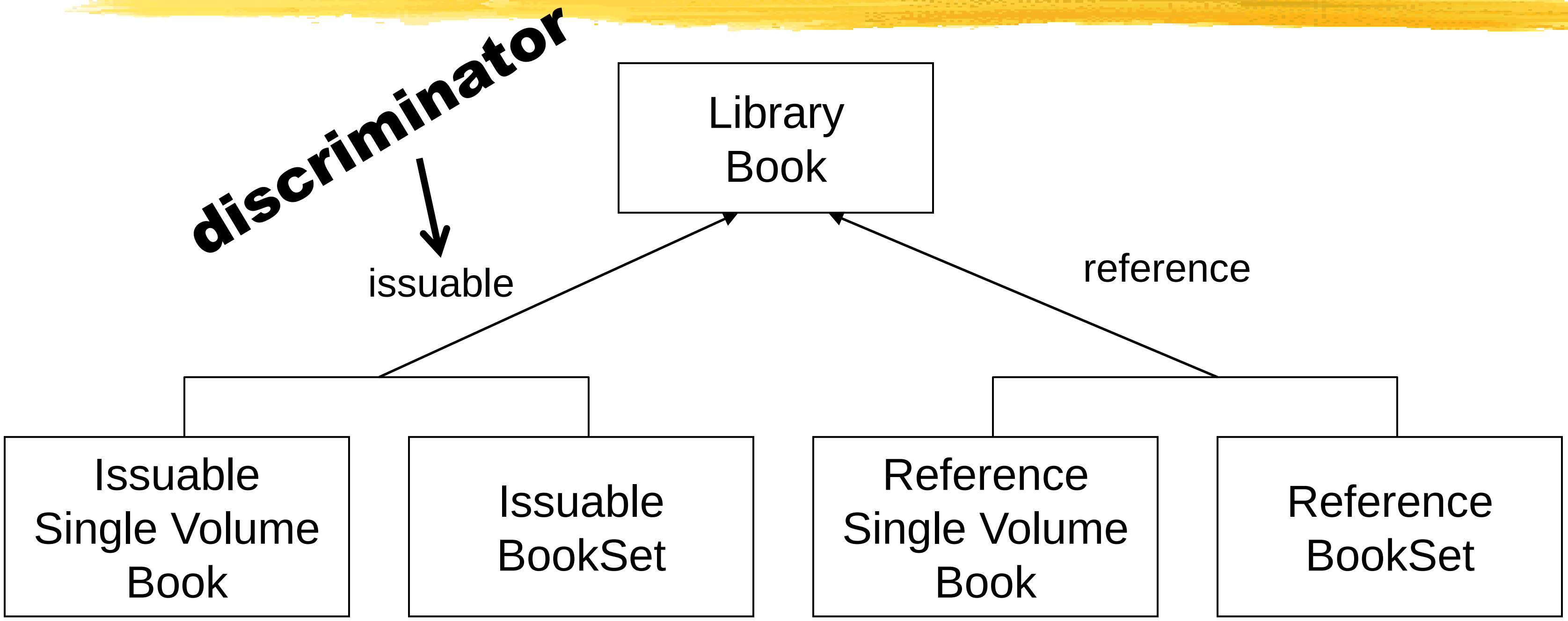
Representation of aggregation

COMPOSITION relationship



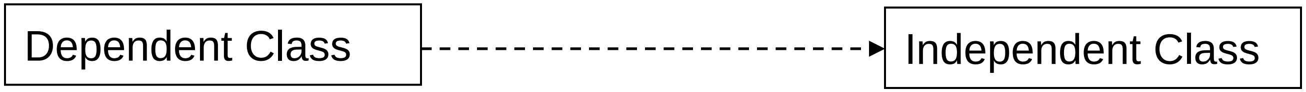
Representation of composition

INHERITANCE relationship



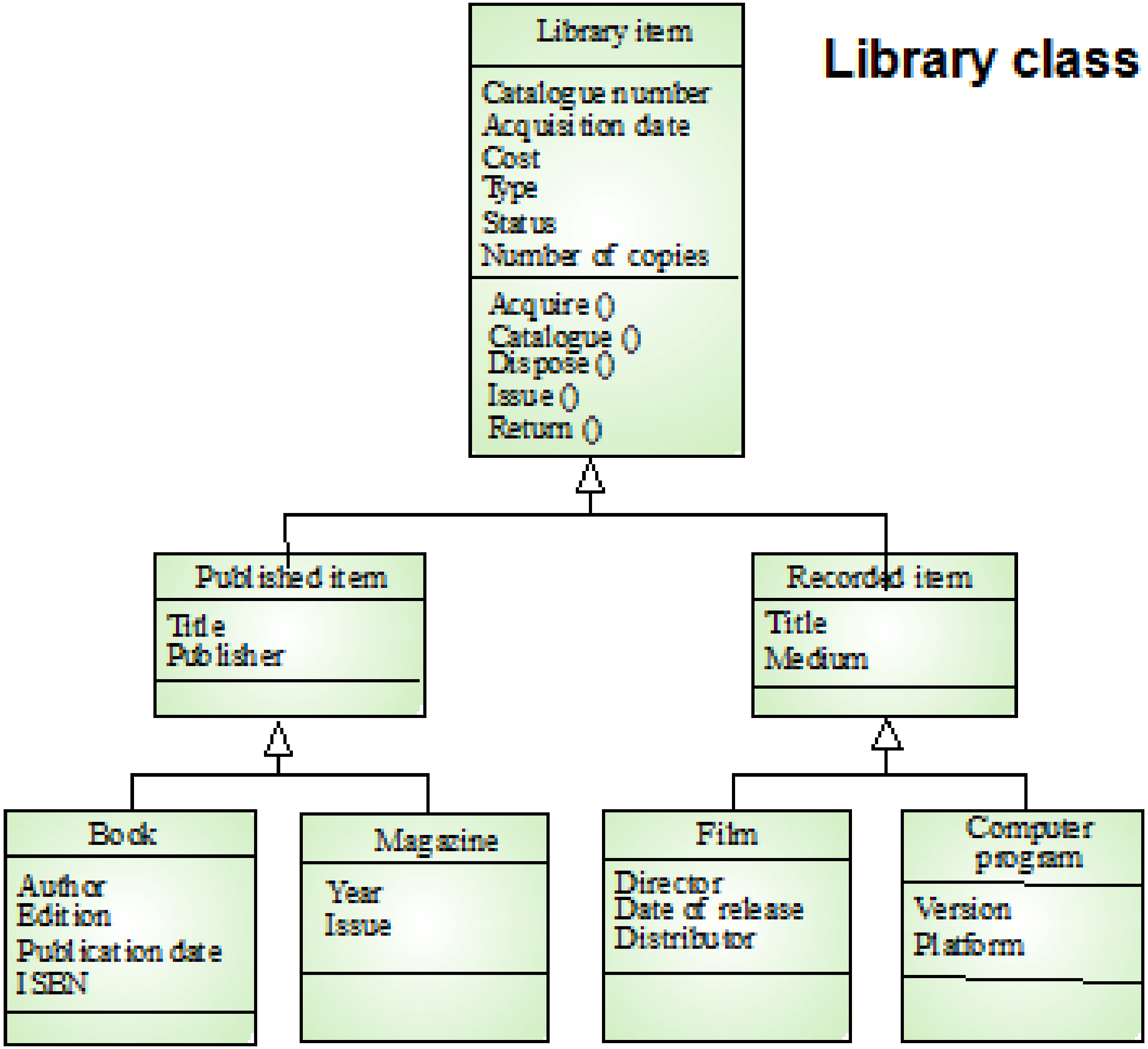
Representation of the inheritance relationship

CLASS dependency

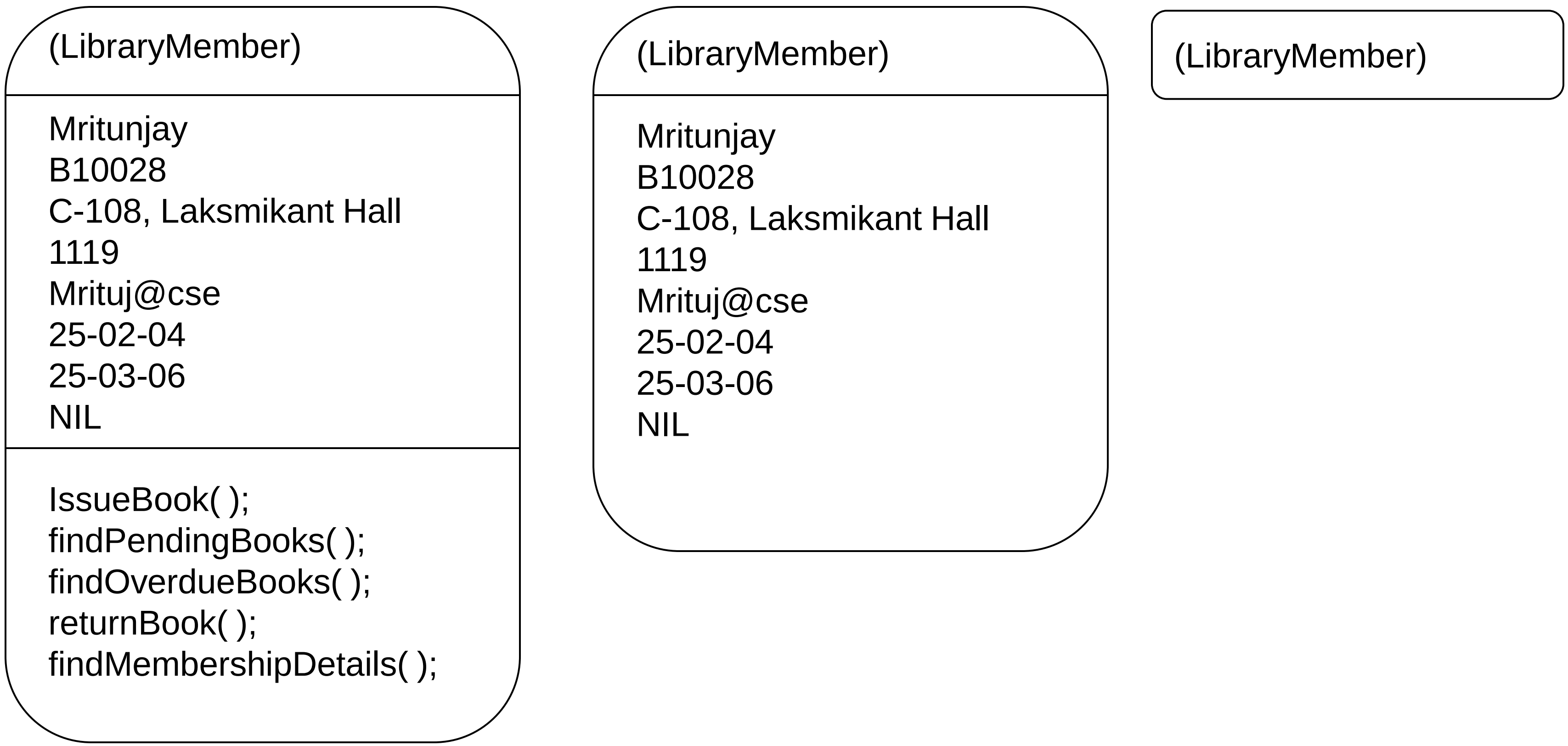


Representation of dependence between class

Library class hierarchy



OBJECT diagram



Different representations of the LibraryMember object

INTERACTION diagram

- Models how groups of objects collaborate to realize some behaviour
- Typically each interaction diagram realizes behaviour of a single use case
- Two kinds: **Sequence** & **Collaboration**
- Two diagrams are equivalent but portrays different perspective

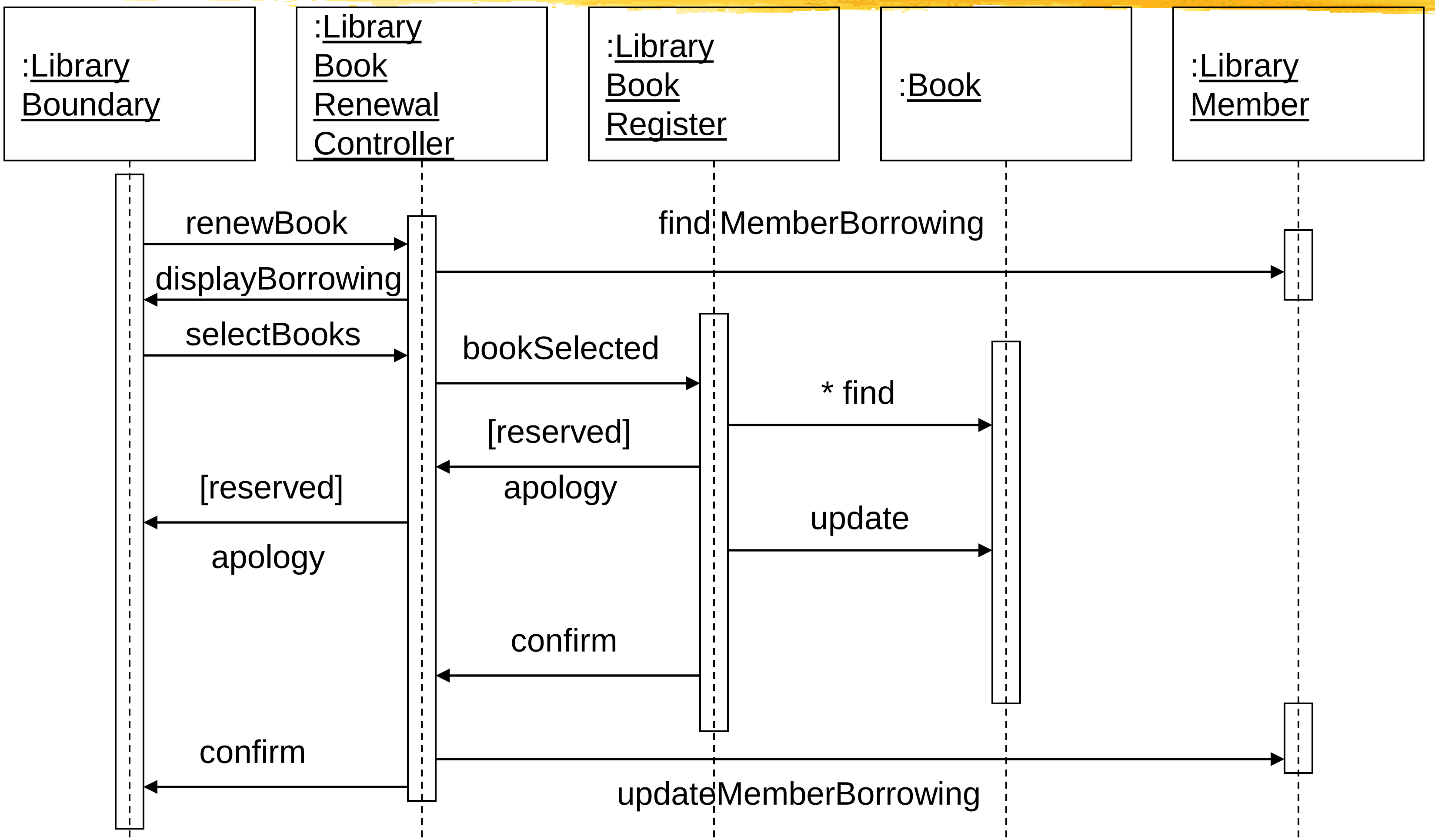
SEQUENCE diagram

- Shows interaction among objects as two-dimensional chart
- **Objects** are shown as **boxes** at top
- If object created during execution then shown at appropriate place
- **Objects existence** are shown as **dashed lines** (lifeline)
- **Objects activeness**, shown as **rectangle** on lifeline: **activation symbol**

SEQUENCE diagram

- **Messages** are shown as **arrows**
- Message labelled with message name
- Message can be labelled with **control information**
- Two types of control information: **condition** ([]) & an **iteration** (*)

Example of SEQUENCE diagram

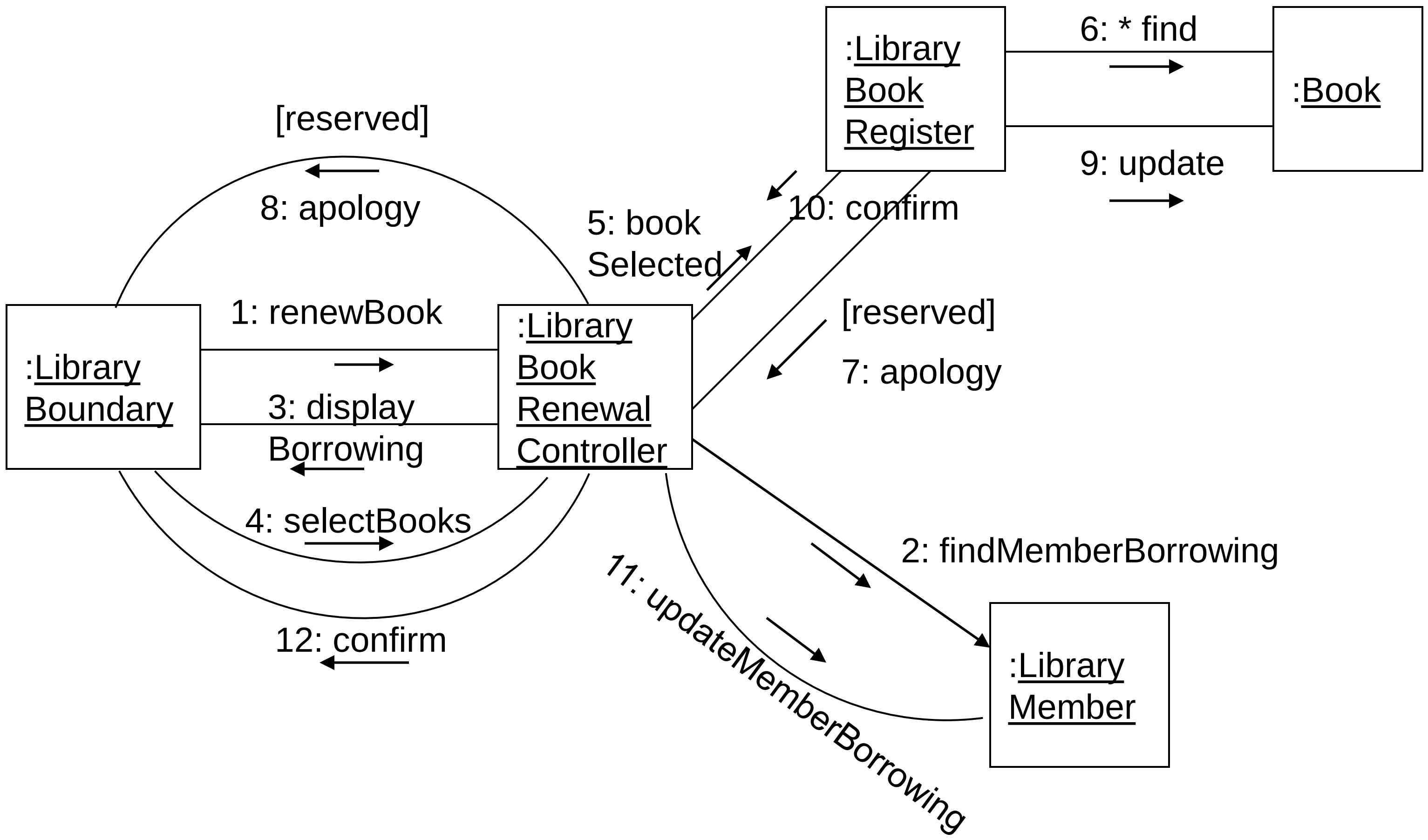


Sequence Diagram for the renew book use case

COLLABORATION diagram

- Shows both **structural** and **behavioural** aspects
- Objects are **collaborator**, shown as boxes
- Links between objects shown as a **solid line**
- Message is shown as a **labelled arrow** placed near the link
- Messages are prefixed with **sequence numbers** to show relative sequencing

Example of COLLABORATION diagram



Collaboration Diagram for the renew book use case

ACTIVITY diagram

- New concept, possibly based on event diagram of Odell [1992]
- Represent processing activity, may not correspond to methods
- Activity is a state with an internal action and one/many outgoing transition

ACTIVITY diagram

- Can represent parallel activity and synchronization aspects

- **Swim lanes** enable to group activities based on who is performing them

- Example: academic department vs. hostel

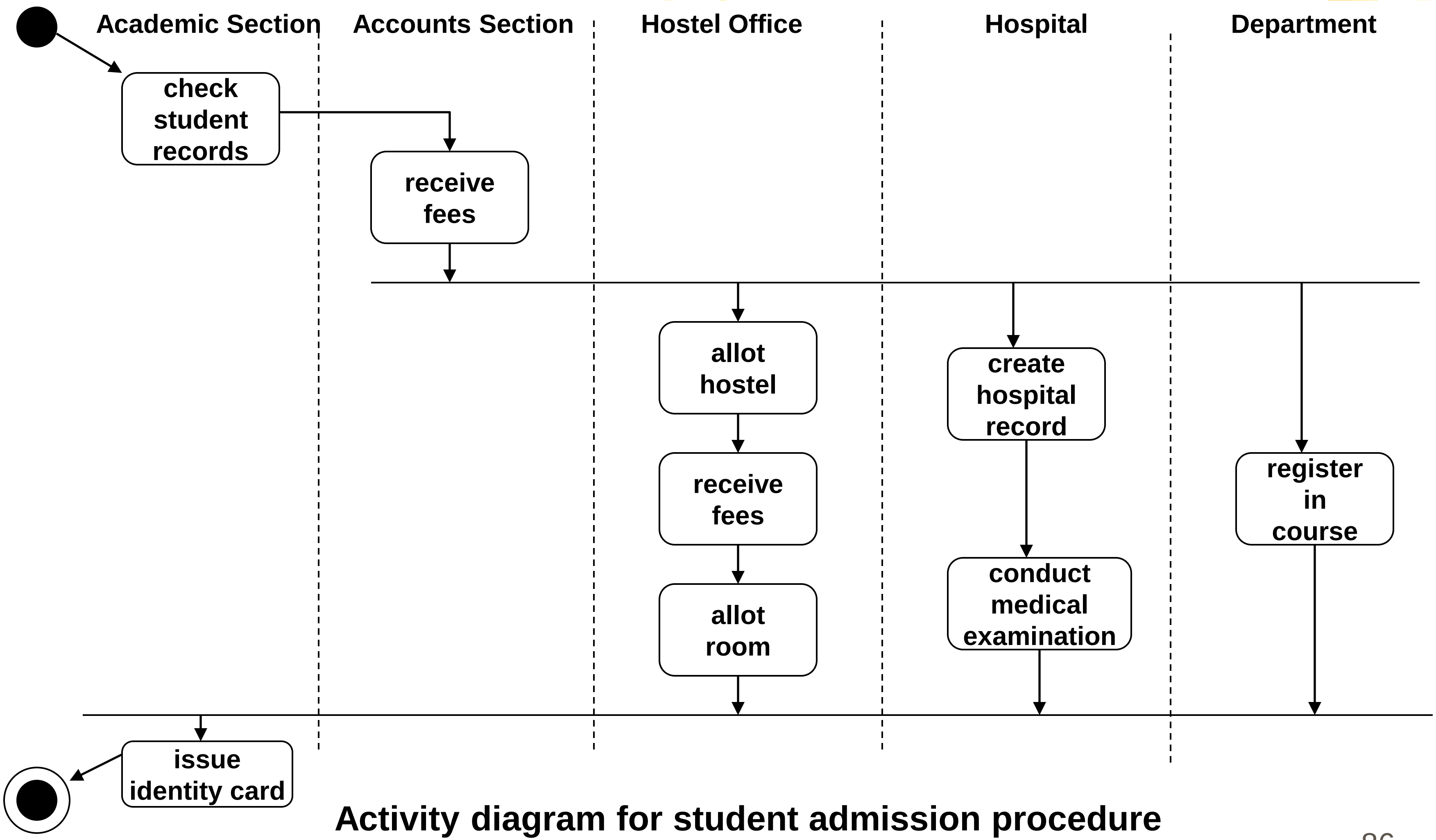
ACTIVITY diagram

- Normally employed in business process modelling

- Carried out during requirement analysis and specification

- Can be used to develop interaction diagrams

Example of ACTIVITY diagram



Activity diagram for student admission procedure

STATE CHART diagram

- Proposed by **David Harel** [1990]
- Model how the state of an object changes in its lifetime
- Based on finite state machine (FSM) formalism

STATE CHART diagram

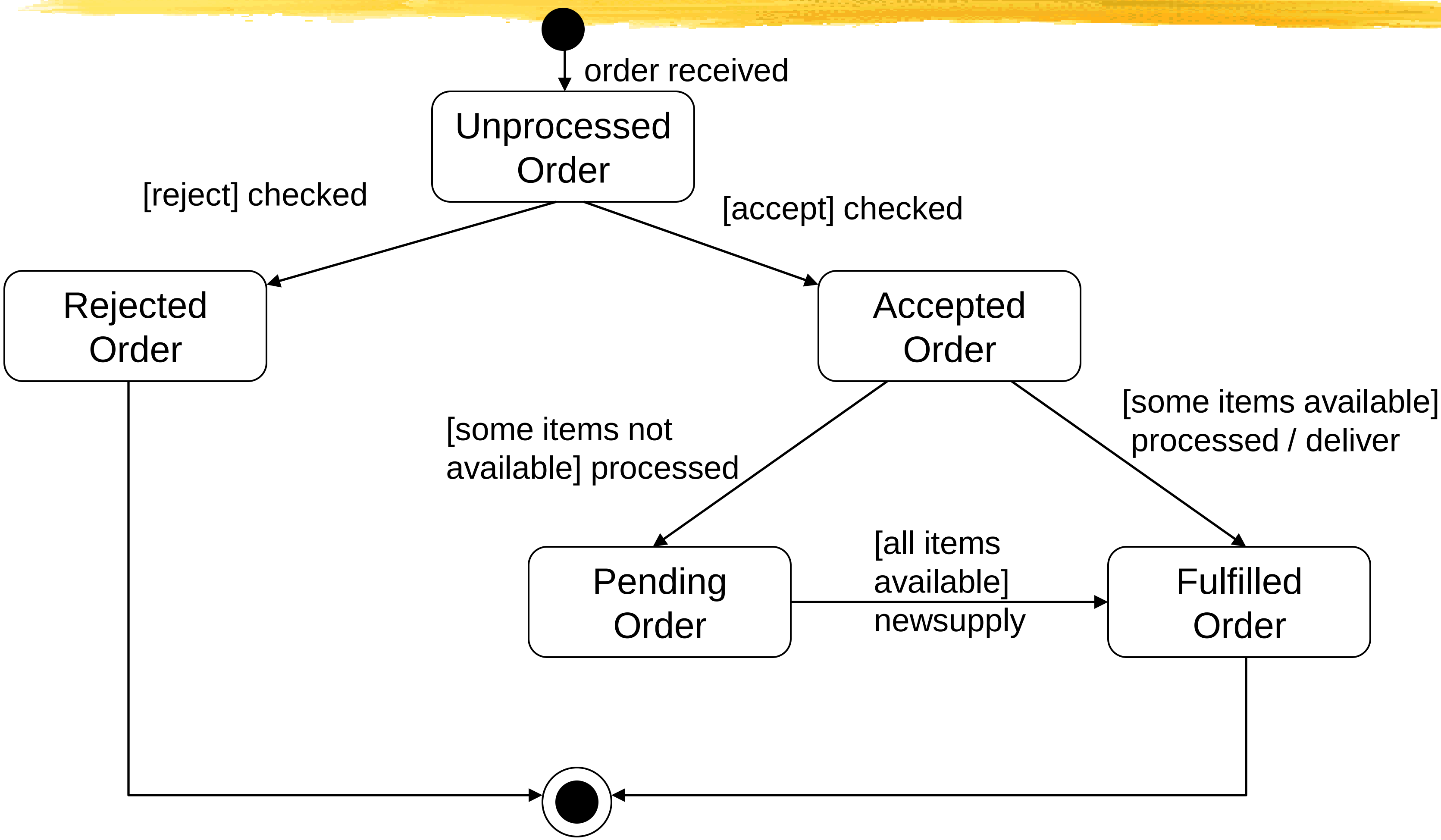
- State chart avoids problem of state explosion as in FSM

- Hierarchical model of a system, represents composite state (**nested**)

STATE CHART diagram

- Elements of state chart diagram
- **Initial State:** Filled circle
- **Final State:** Filled circle inside larger circle
- **State:** Rectangle with rounded corners
- **Transitions:** Arrow between states, also boolean logic condition (**guard**). Label of transition shown in 3 parts: [guard]event/action

Example of STATE CHART diagram

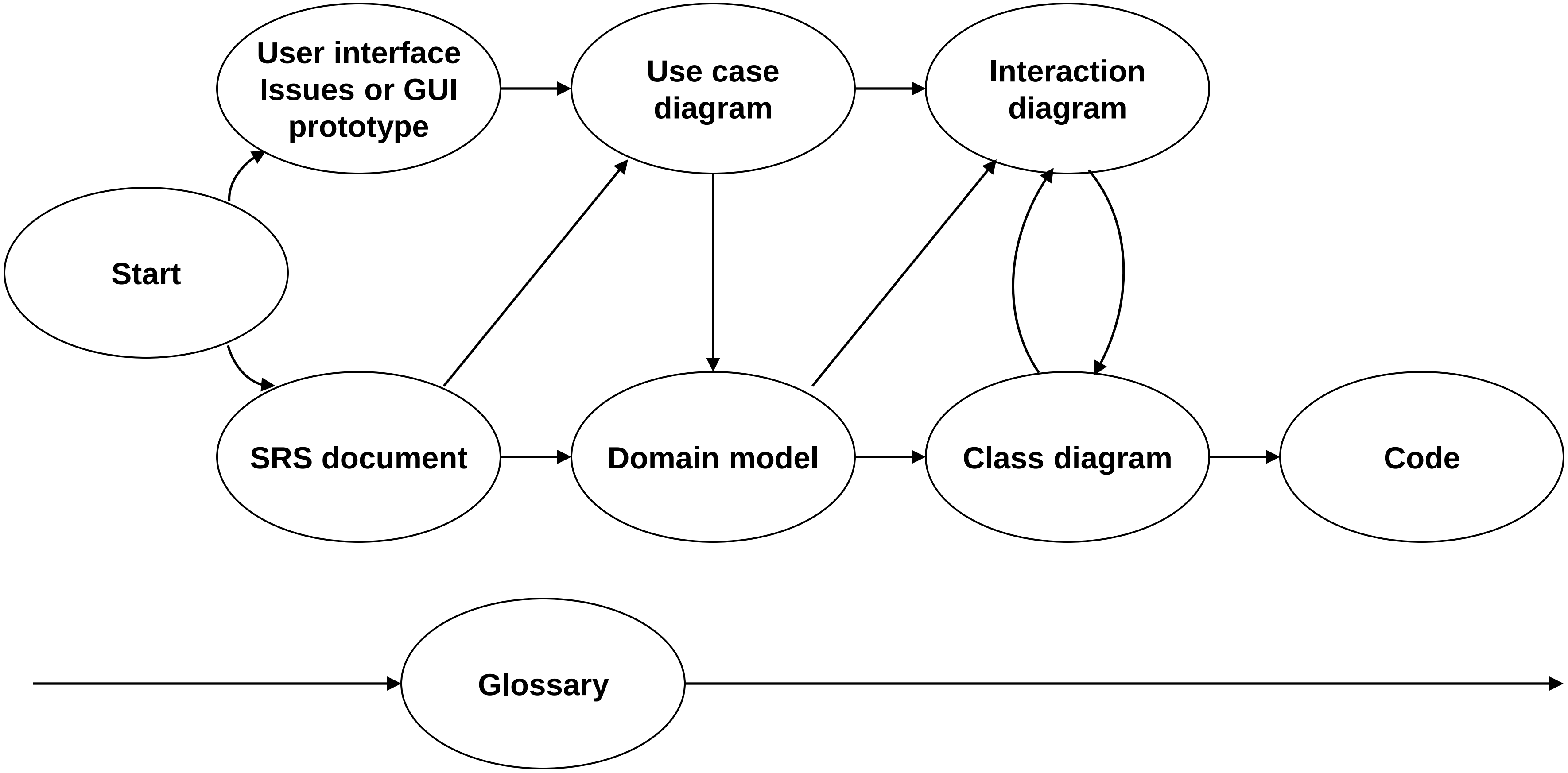


Example: State chart diagram for an order object

OBJECT-ORIENTED software design & patterns

- **Objects** are identified by examining **nouns** in problem description
- Many OOD techniques are proposed by Grady Booch [1991]
- From requirements specification, initial model is developed (OOA)
- Analysis model is refined into a design model
- Design model is implemented using OO concepts

OBJECT-ORIENTED software design process



DOMAIN modelling

- Representation of concepts or objects appearing in the problem domain
- Also captures relationships among objects
- Three types of objects are identified
- **Boundary objects**
- **Entity objects**
- **Controller objects**

BOUNDARY objects

- Interacts with actors
- Includes screens, menus, forms, dialogs etc.
- Do not perform processing but validates formats etc.
- "Interface class" term used for these in Java, COM/DCOM & UML

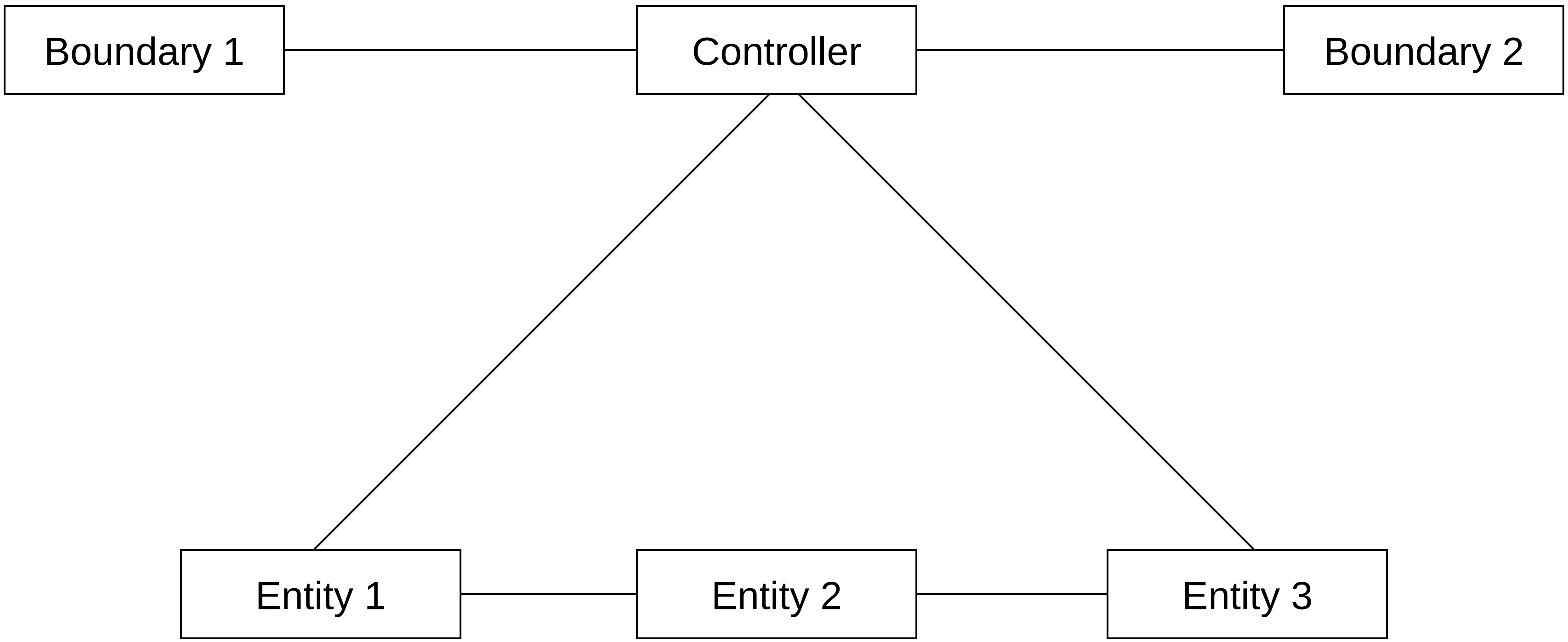
ENTITY objects

- Hold information such as data, tables & files, e.g. *Book*, *BookRegister*
- Many of these are dumb servers
- Responsible for storing data, fetching data etc.

CONTROLLER objects

- Coordinate the activities of a set of entity objects
- Interface with the boundary objects
- Realizes use case
- Embody most of the logic involved with the use case realization
- There can be more than one controller

USE CASE realization



Realization of use case through the collaboration of Boundary, controller and entity objects

SUMMARY

- We discussed object-oriented concepts
  - **Basic mechanisms:** Such as objects, class, methods, inheritance etc.
  - **Key concepts**: Such as abstraction, encapsulation, polymorphism, composite objects etc.

SUMMARY

- We discussed an important OO language UML
  - Its **origin**, as a **standard**, as a **model**
  - Use case **representation**, its **factorisation** such as generalization, includes and extends
  - Different diagrams for UML representation
  - In **class diagram** we discussed some relationships **association**, **aggregation**, **composition** and **inheritance**

SUMMARY

  - Some more diagrams such as **interaction diagrams** (sequence and collaboration), **activity diagrams**, **state chart diagram**
- We discussed OO software development process and patterns
  - In this we discussed some **patterns** example and **domain modelling**