

# Object-Orientation Concepts, UML, and OOAD

**Prof. R. Mall**

Dept. of CSE, IIT, Kharagpur

# Organization of This Lecture

- Object-oriented concepts
- Object modelling using Unified Modelling Language (UML)
- Object-oriented software development and patterns
- CASE tools
- Summary

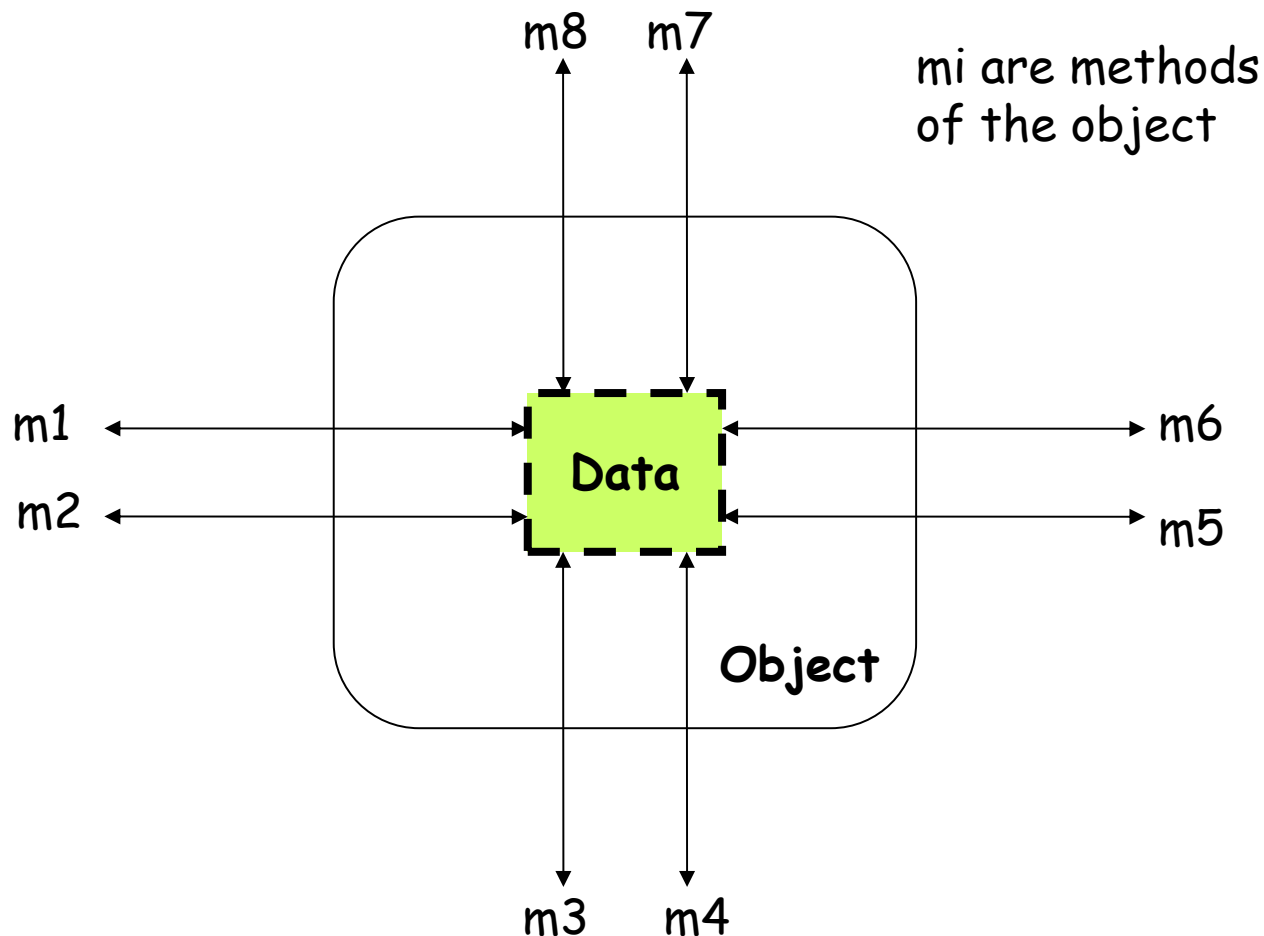
# Object-Orientation Concepts

- Object-oriented (OO) design techniques are extremely popular:
  - 📁 Inception in early 1980's and nearing maturity.
  - 📁 Widespread acceptance in industry and academics.
  - 📁 Unified Modelling Language (UML) already an ISO standard (ISO/IEC 19501).

# Objects

- A system is designed as a set of interacting objects:
  - 📖 Often, real-world entities:
    - Examples: an employee, a book etc.
  - 📖 Can be conceptual objects also:
    - Controller, manager, etc.
- Consists of data (**attributes**) and functions (**methods**) that operate on data.
  - 📖 Hides organization of internal information (**Data abstraction**).

# Model of an Object

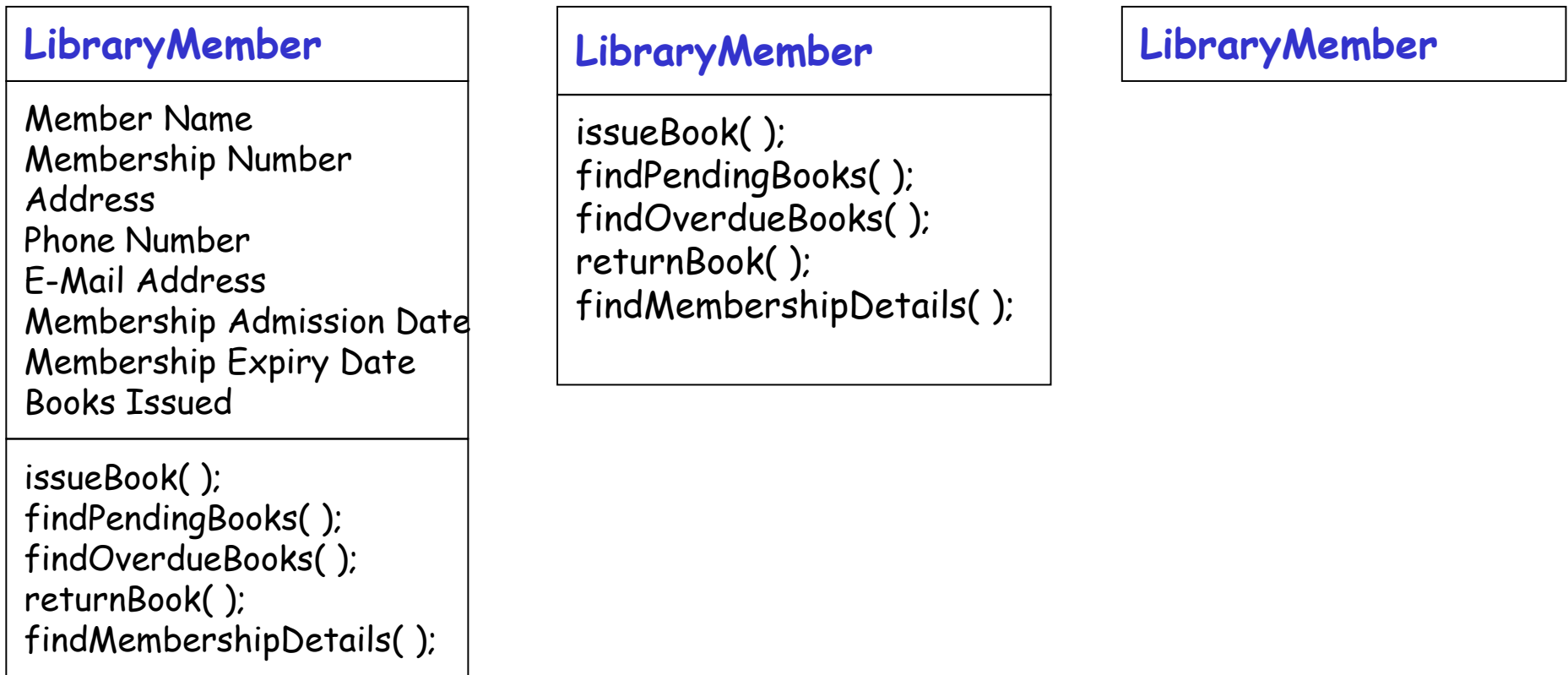


# Class

- Instances are objects
- Template for object creation
- Considered as abstract data type (ADT)
- Examples: Employees, Books, etc.
- Sometimes not intended to produce instances:

 Abstract classes

# Example Class Diagram



Different representations of the `LibraryMember` class

# Methods and Messages

- Operations supported by an object:

- 📁 Means for manipulating the data of other objects.

- 📁 Invoked by sending a message (method call).

- 📁 Examples: `calculate_salary`, `issue-book`, `member_details`, etc.



# What are the Different Types of Relationships Among Classes?

- Four types of relationships:

 Inheritance

 Association

 Aggregation/Composition

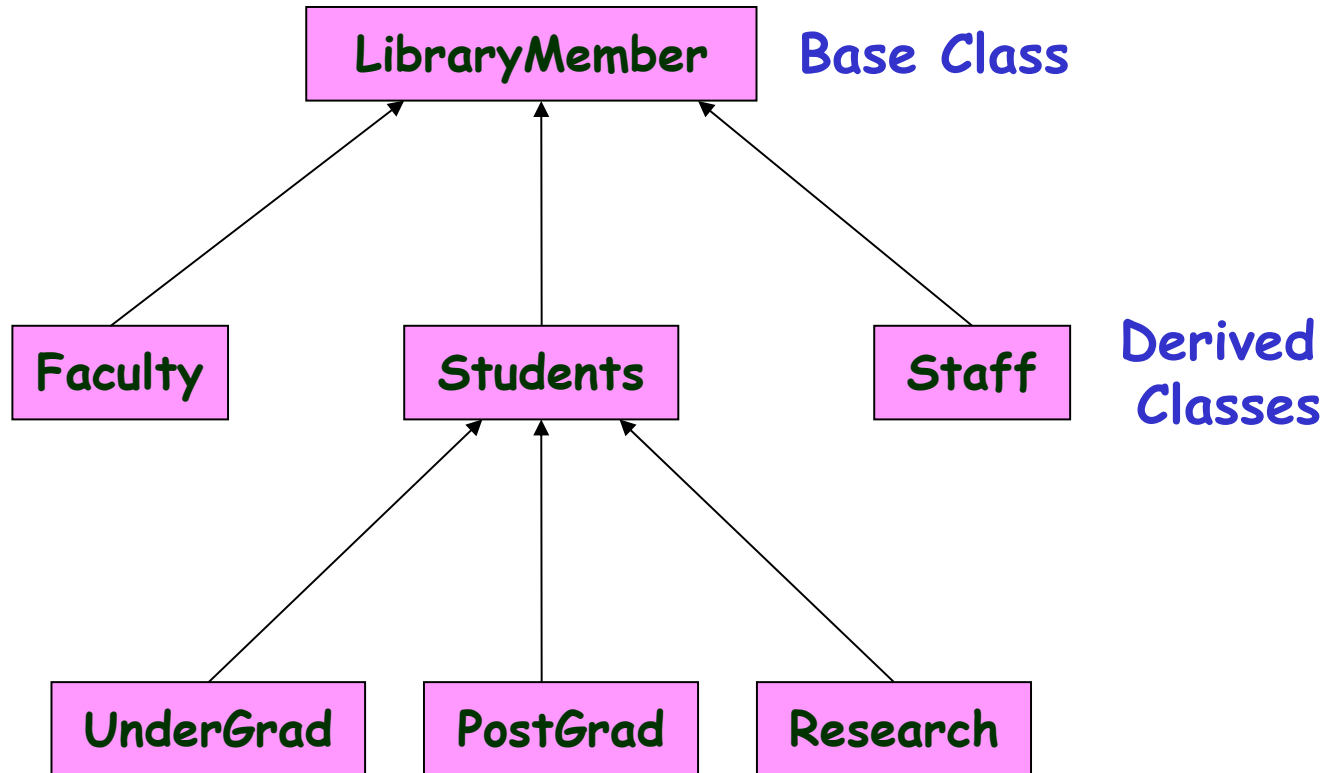
 Dependency

# Inheritance

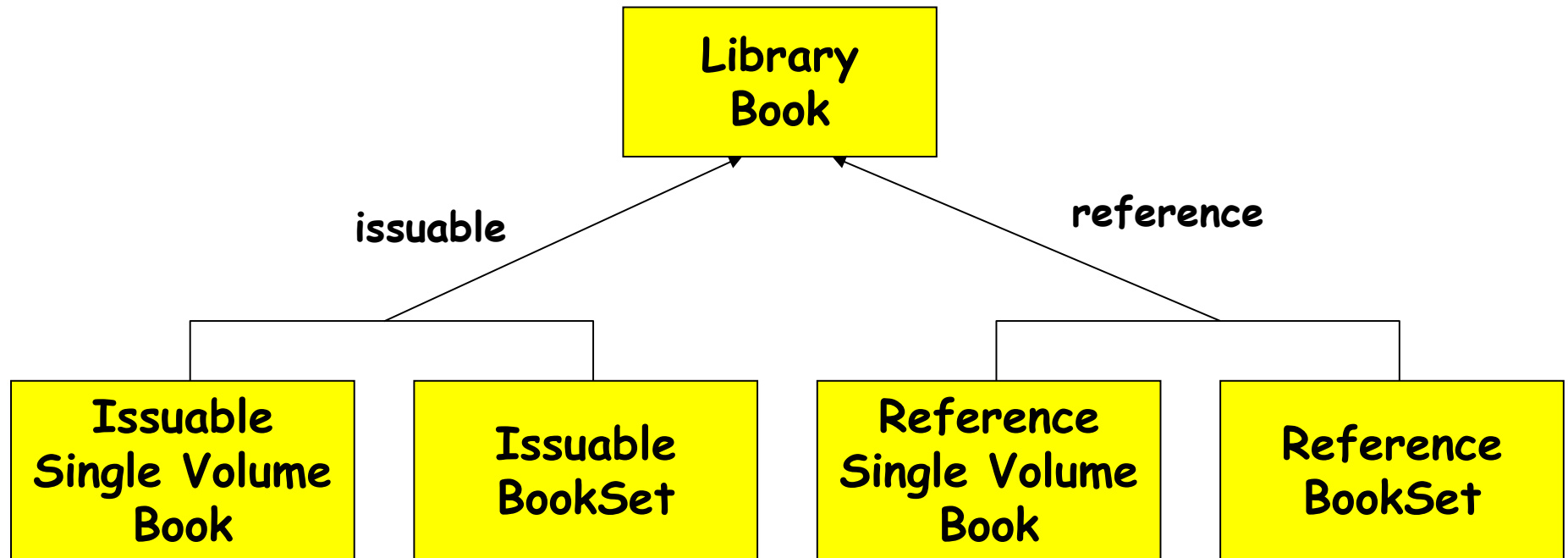
- Allows to define a new class (**derived class**) by extending or modifying existing class (**base class**).
  - ☞ Represents **generalization-specialization** relationship.
  - ☞ Allows redefinition of the existing methods (**method overriding**).

# Inheritance

- Lets a subclass inherit attributes and methods from more than one base class.



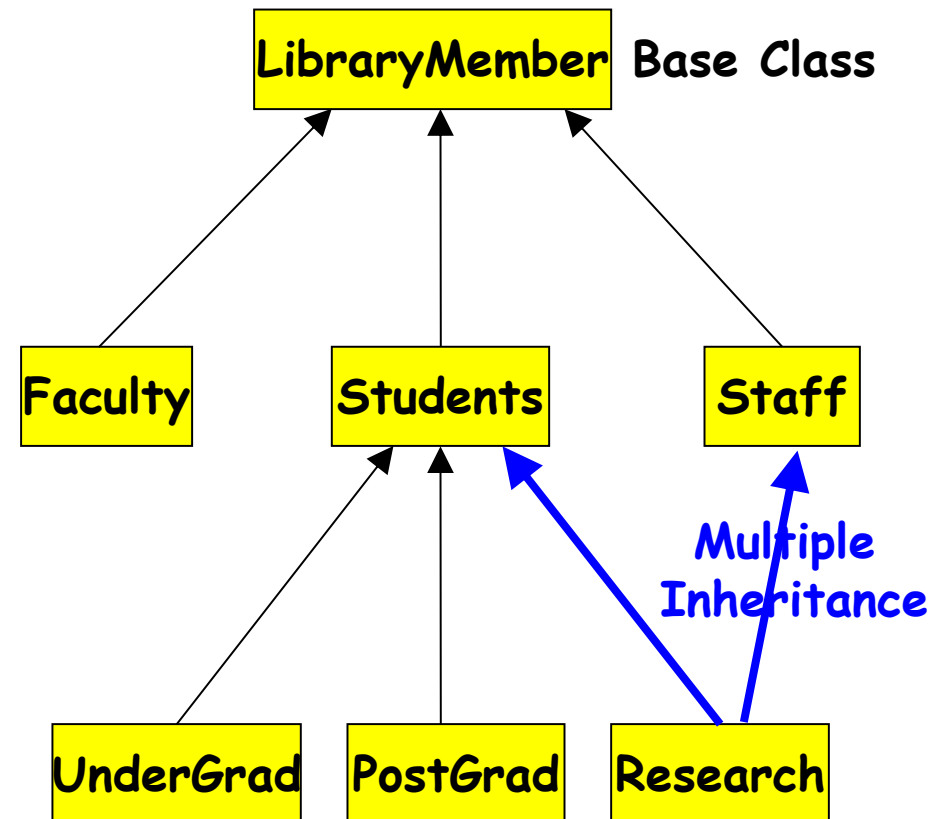
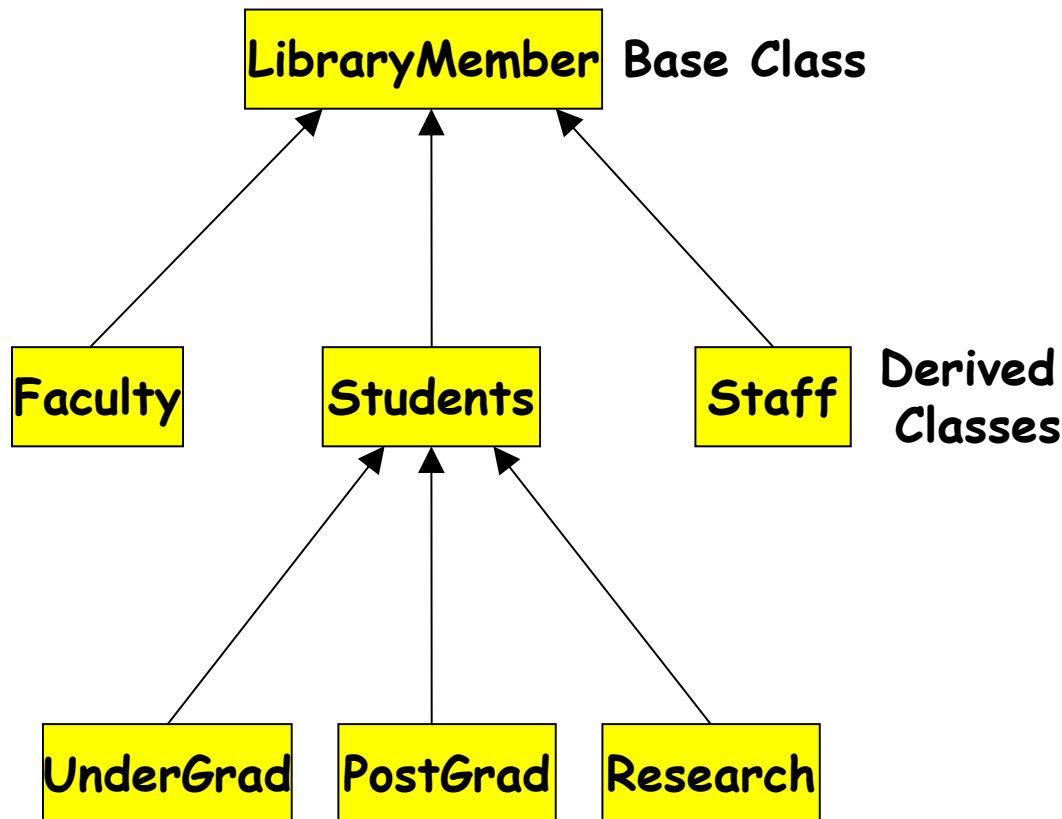
# Inheritance Example



Representation of the inheritance relationship

# Multiple Inheritance

cont...



# Association Relationship

- Enables objects to communicate with each other:
  - 📁 Thus one object must "know" the address of the corresponding object in the association.
- Usually binary:
  - 📁 But in general can be n-ary.

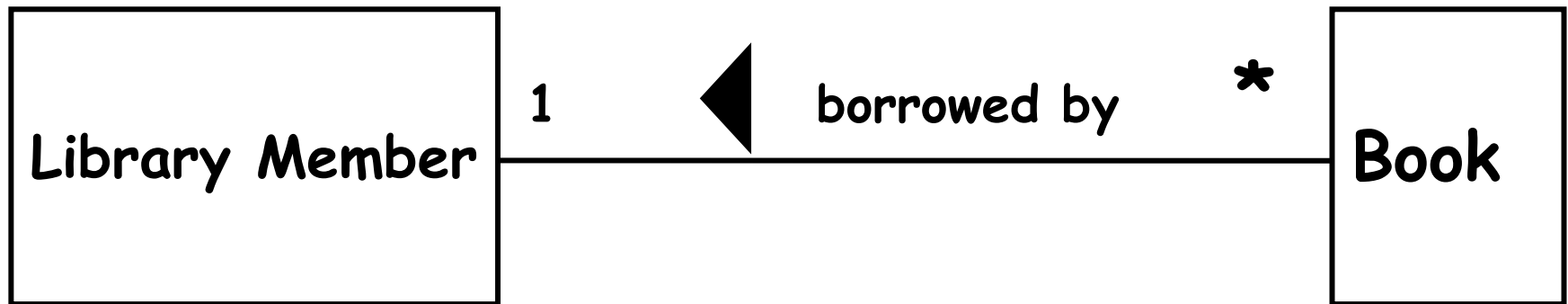
# Association Relationship

- A class can be associated with itself (**recursive** association).

 Give an example?

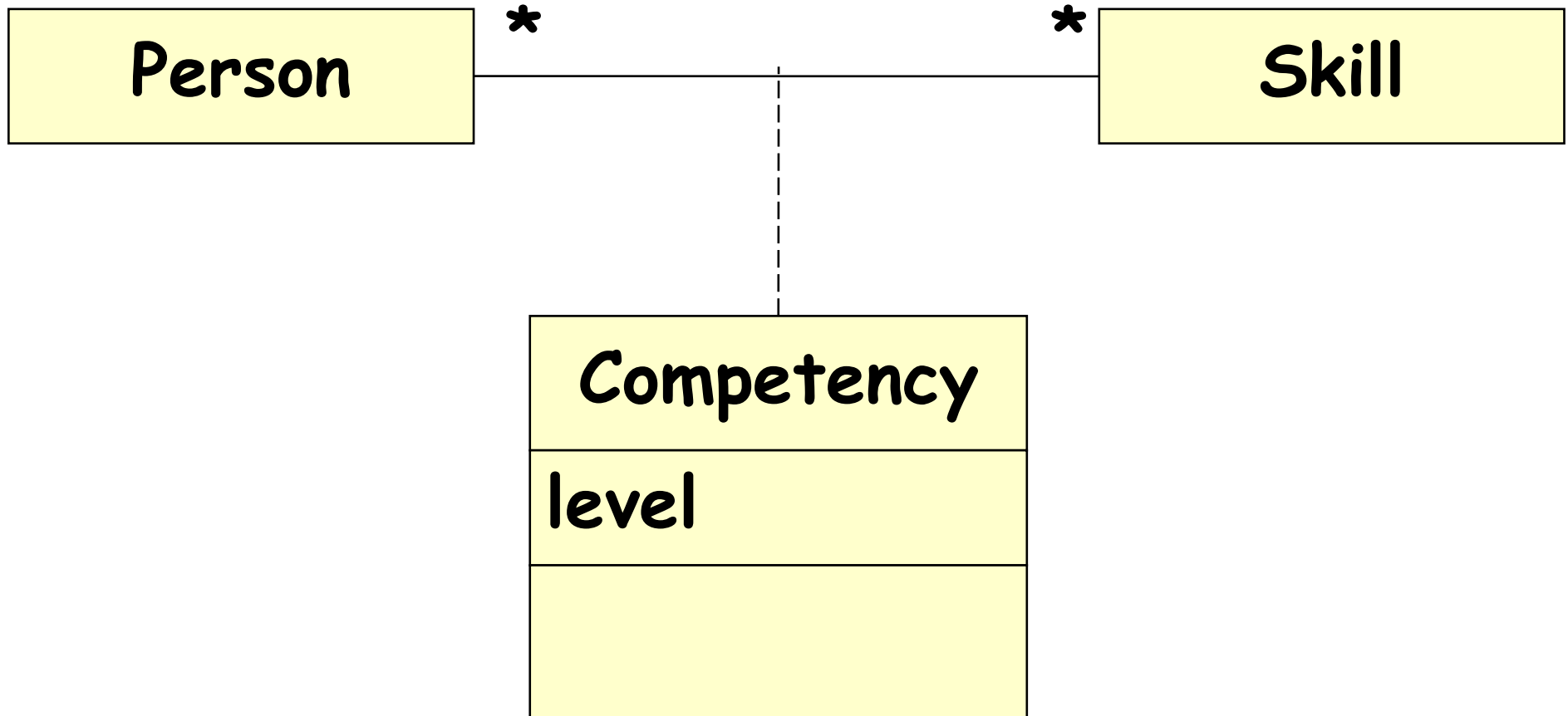
- An arrowhead used along with name, indicates direction of association.
- Multiplicity indicates # of instances taking part in the association.

# Association Relationship





# 3-ary Association



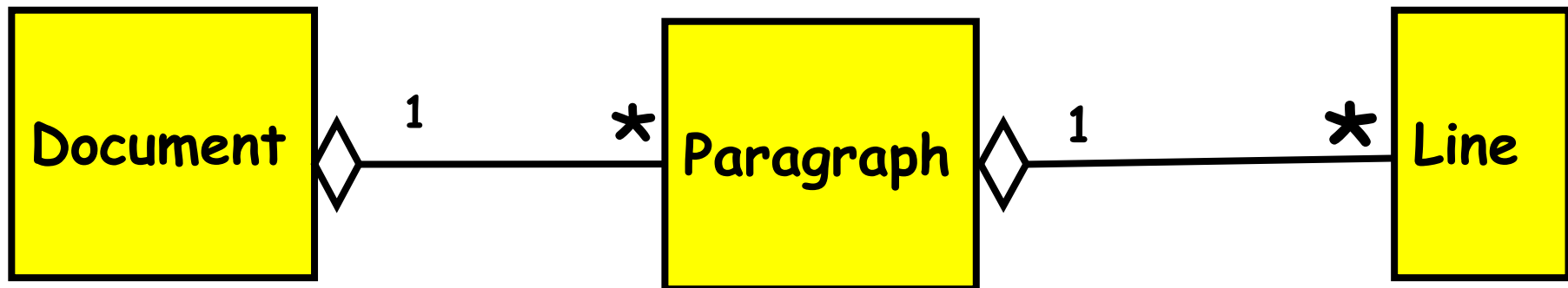
# Association and Link

- A link:
  - 📁 An instance of an association
  - 📁 Exists between two or more objects
  - 📁 Dynamically created and destroyed as the run of a system proceeds
- For example:
  - 📁 An employee joins an organization,
  - 📁 Leaves that organization and joins a new organization etc.

# Aggregation Relationship

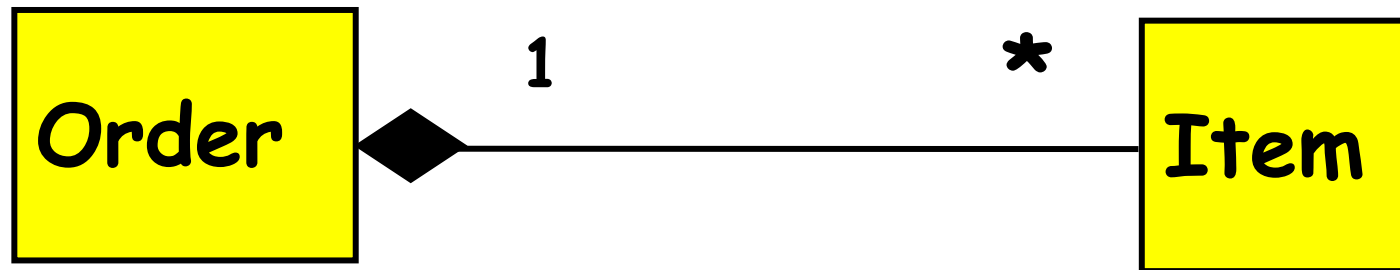
- Represents whole-part relationship
- Represented by a **diamond** symbol at the composite end
- Cannot be reflexive(i.e. recursive)
- Not symmetric
- It can be transitive

# Aggregation Relationship



# Composition Relationship

- Life of item is same as the order



# Aggregation

cont...

- A aggregate object contains other objects.
- Aggregation limited to **tree hierarchy**:
  - 📁 No circular inclusion relation.

# Aggregation vs. Inheritance

Cont...

- Inheritance:

- 📁 Different object types with similar features.

- 📁 Necessary semantics for similarity of behavior is in place.

- Aggregation:

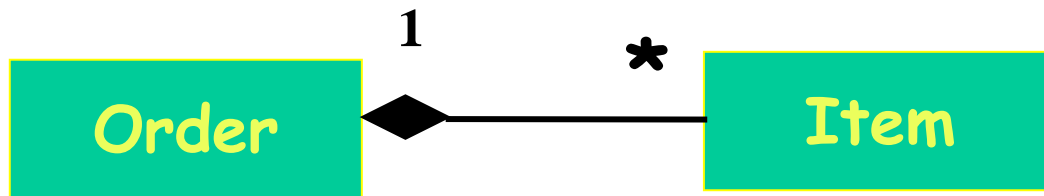
- 📁 Containment allows construction of complex objects.

# Aggregation vs. Composition

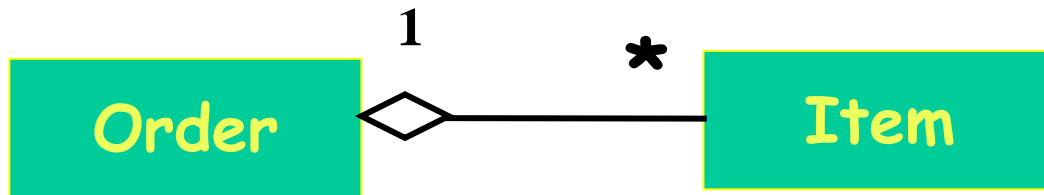
- Composition:
  - 📁 Composite and components have the same life.
- Aggregation:
  - 📁 Lifelines are different.
- Consider an **order** object:
  - 📁 **Aggregation:** If order items can be changed or deleted after placing the order.
  - 📁 **Composition:** Otherwise.



# Composition versus Aggregation

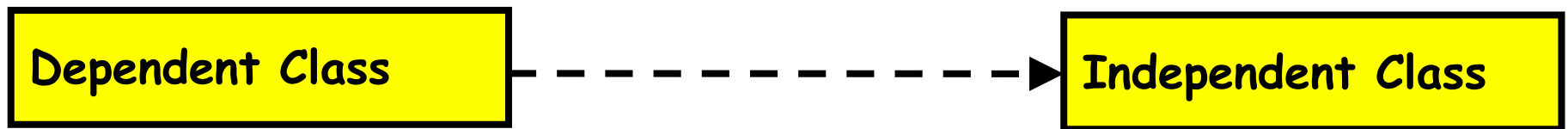


Composition



Aggregation


# Class Dependency



Representation of dependence between classes

# Abstraction

- Consider aspects relevant for certain purpose

 Suppress non-relevant aspects

- Types of abstraction:

 Data abstraction

 Behaviour abstraction

# Abstraction

cont...

- Advantages of abstraction:

- 📁 Reduces complexity of design

- 📁 Enhances understandability

- 📁 Increases productivity

- It has been observed that:

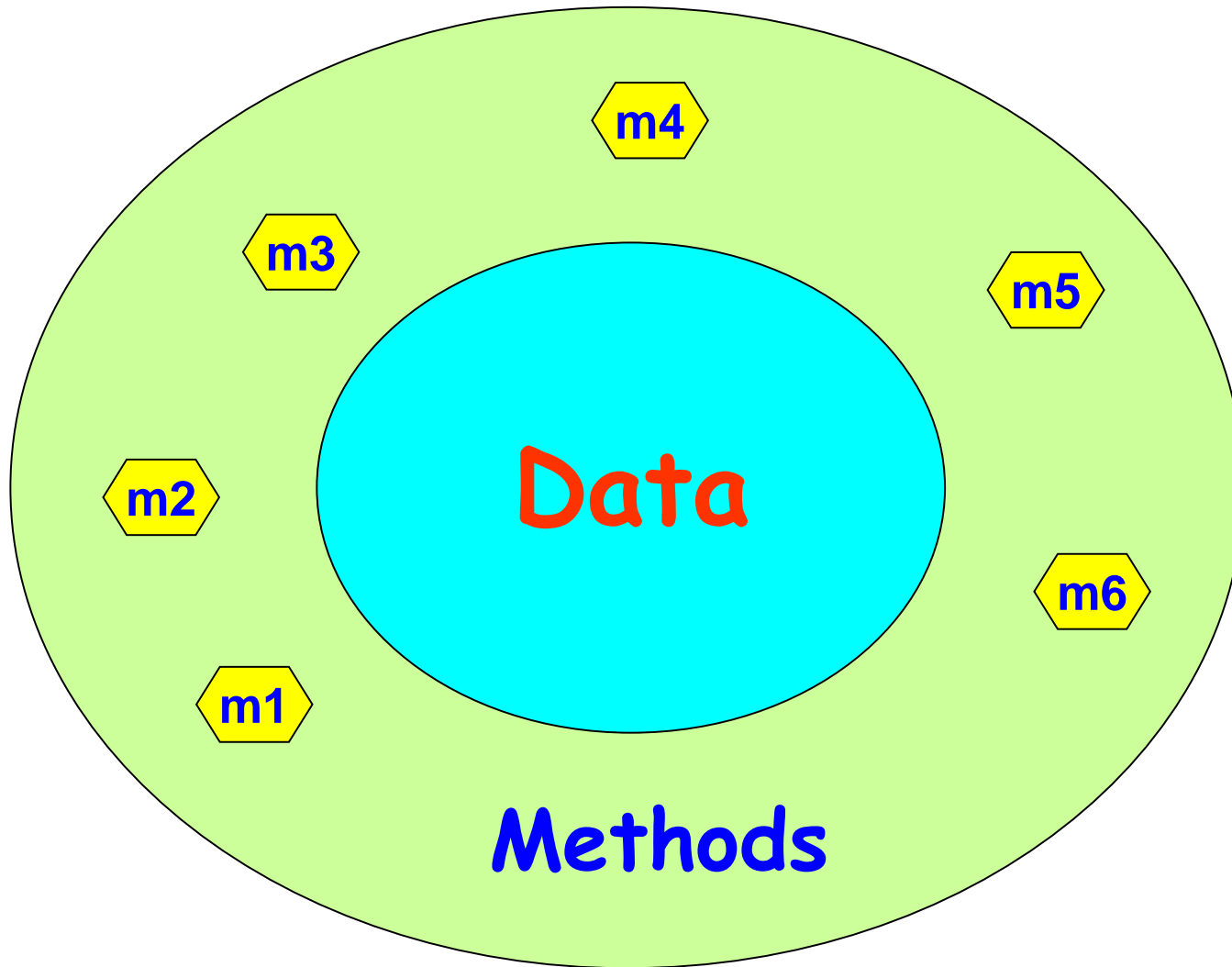
- 📁 Productivity is inversely proportional to complexity.

# Encapsulation

- Objects communicate with outside world through messages:
  - Data of objects encapsulated within its methods.
  - Data accessible only through methods.

# Encapsulation

cont...



Concept of encapsulation

# Polymorphism

- Denotes poly (**many**) morphism (**forms**).
- Under different situations:
  - 📁 Same message to the same object can result in different actions:
    - Static binding
    - Dynamic binding

# An 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);
- }



# An Example of Static Binding

cont...

- A class named **Circle** has three definitions for **create** operation
  - 📁 Without any parameter, default
  - 📁 Centre and radius as parameter
  - 📁 Centre, radius and fillType as parameter
  - 📁 Depending upon parameters, method will be invoked
  - 📁 Method **create** is said to be **overloaded**

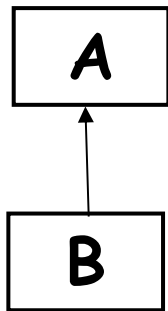
# Dynamic Binding

- A method call to an object of an ancestor class:
  - 📁 Would result in the invocation of the method of an appropriate object of the derived class.
- Following principles are involved:
  - 📁 Inheritance hierarchy
  - 📁 Method overriding
  - 📁 Assignment to compatible types

# Dynamic Binding

- Principle of substitutability (Liskov's substitutability principle):

📁 An object can be assigned to an object of its ancestor class, but not vice versa.



A a; B b;

a=b; (OK)

b=a; (not OK)

# Dynamic Binding

Cont ...

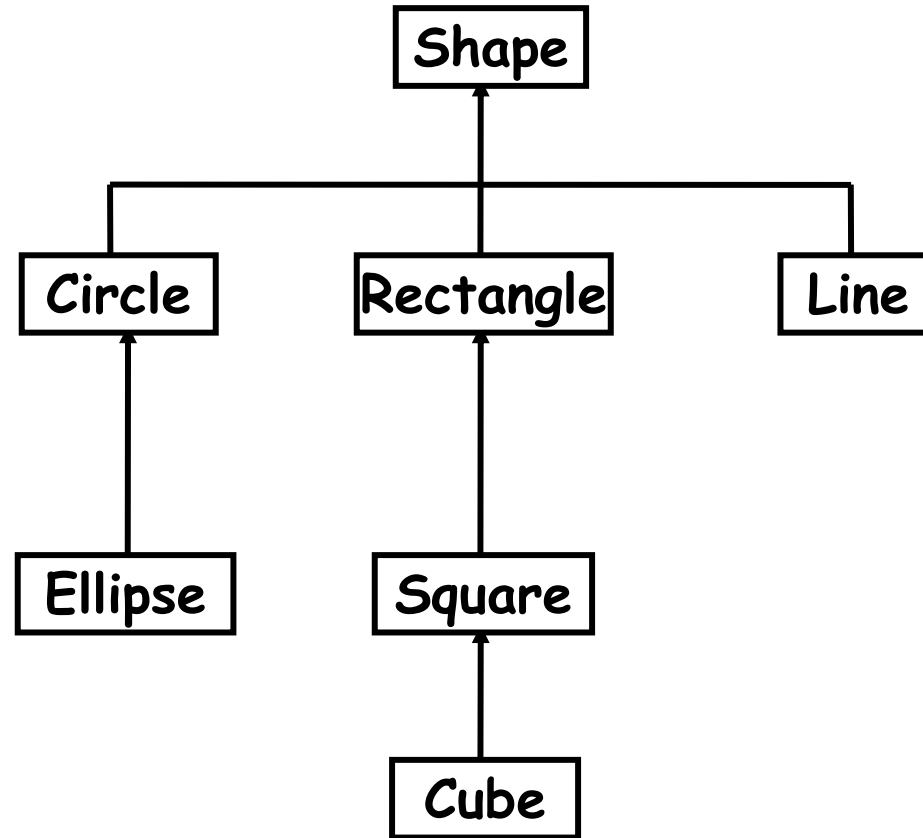
- Exact method to be bound on a method call:
  - ❏ Not possible to determine at compile time.
  - ❏ Dynamically decided at runtime.

# An Example of Dynamic Binding

- Consider a class hierarchy of different geometric objects:
  - 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 of displaying the appropriate element.

# An Example of Dynamic Binding

cont...



Class hierarchy of geometric objects

# An Example

cont...

## Traditional code

```
Shape s[1000];  
For(i=0;i<1000;i++){  
    If (s[i] == Circle)  then  
        draw_circle();  
    else if (s[i]== Rectangle)  
then  
        draw_rectangle();  
    -  
}  
}
```

## Object-oriented code

```
Shape s[1000];  
For(i=0;i<1000;i++){  
    Shape.draw();  
    -  
    -  
    -  
}
```

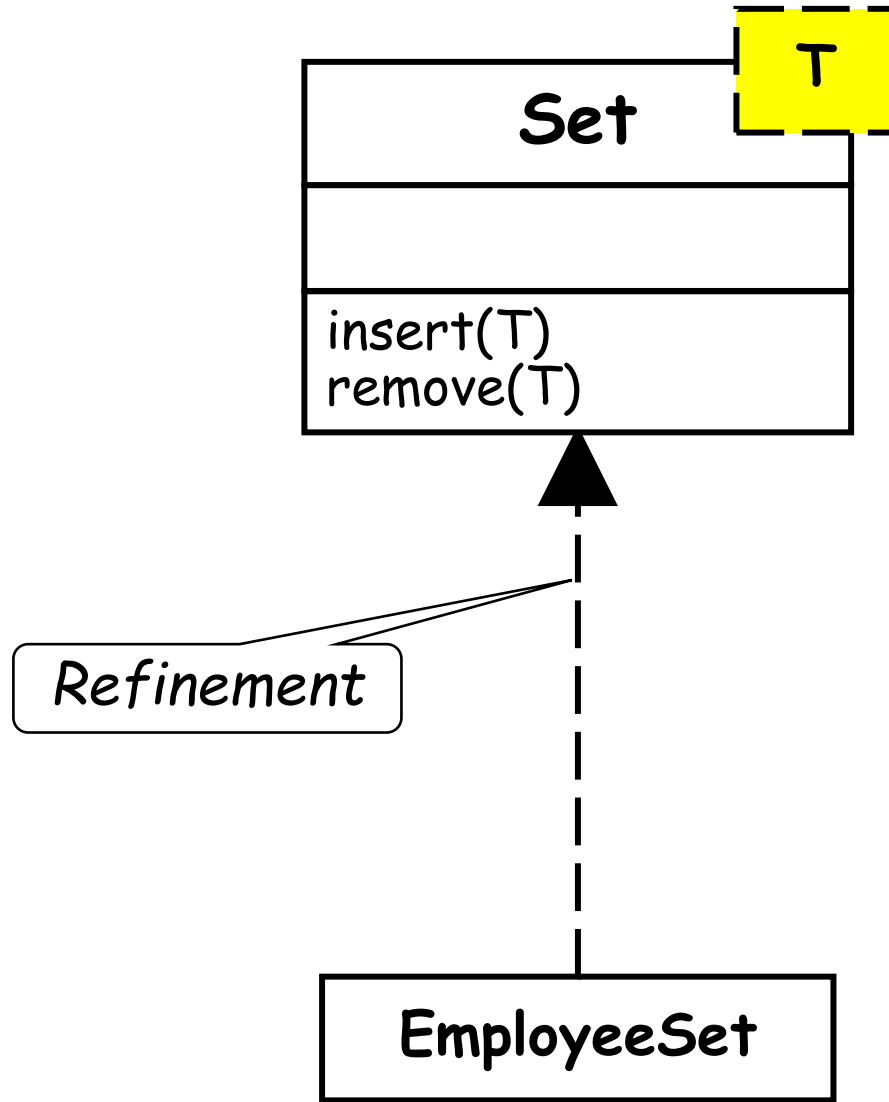
**Traditional code and OO code using dynamic binding**

# Genericity

- Ability to parameterize class definitions.
- Example: class `stack` of different types of elements:
  - 📁 Integer stack
  - 📁 Character stack
  - 📁 Floating point stack
- Define generic class `stack`:
  - 📁 Later instantiate as required



# Genericity



# Advantages of Object-Oriented Development

- Code and design reuse
- Increased productivity
- Ease of testing (?) and maintenance
- Better understandability
- Elegant design:
  - 📁 Loosely coupled, highly cohesive objects:
  - 📁 Essential for solving large problems.

# Advantages of Object-Oriented Development

cont...

- Initially incurs higher costs
  - 📁 After completion of some projects reduction in cost become possible
- Using well-established OO methodology and environment:
  - 📁 Projects 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 results.
- Independent of any specific design methodology.

# UML Origin

- OOD in late 1980s and early 1990s:
  - 📁 Different software development houses were using different notations.
  - 📁 Methodologies were tied to notations.
- UML developed in early 1990s to:
  - 📁 Standardize the large number of object-oriented modelling notations

# UML Lineology

- Based Principally on:

 **OMT** [Rumbaugh 1991]

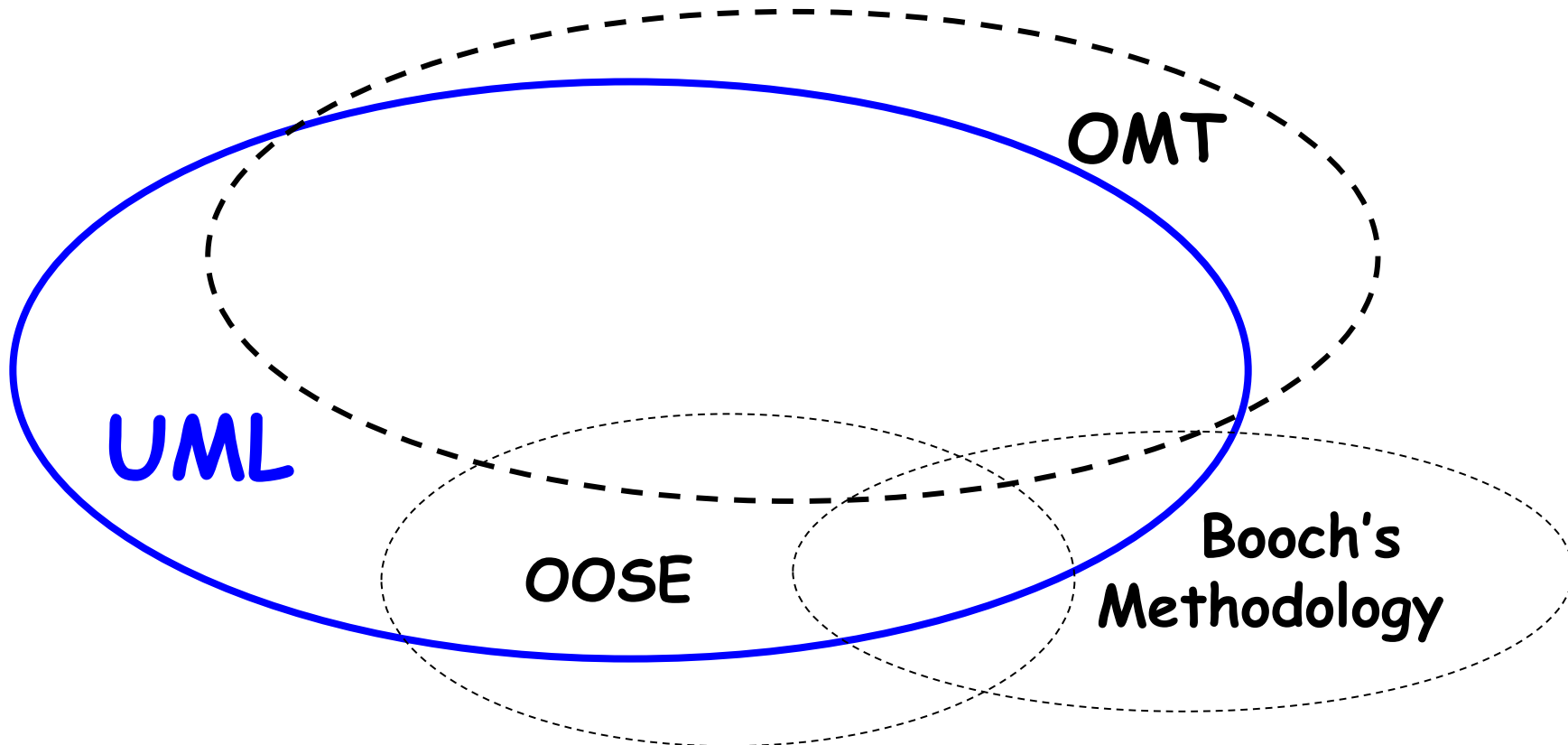
 **Booch's methodology** [Booch 1991]

 **OOSE** [Jacobson 1992]

 **Odell's methodology** [Odell 1992]

 **Shlaer and Mellor** [Shlaer 1992]

# Different Object Modeling Techniques in UML



# UML as A Standard

- Adopted by Object Management Group (OMG) in 1997
- OMG is an association of industries
- Promotes consensus notations and techniques
- Used outside software development
  - 📁 Example car manufacturing

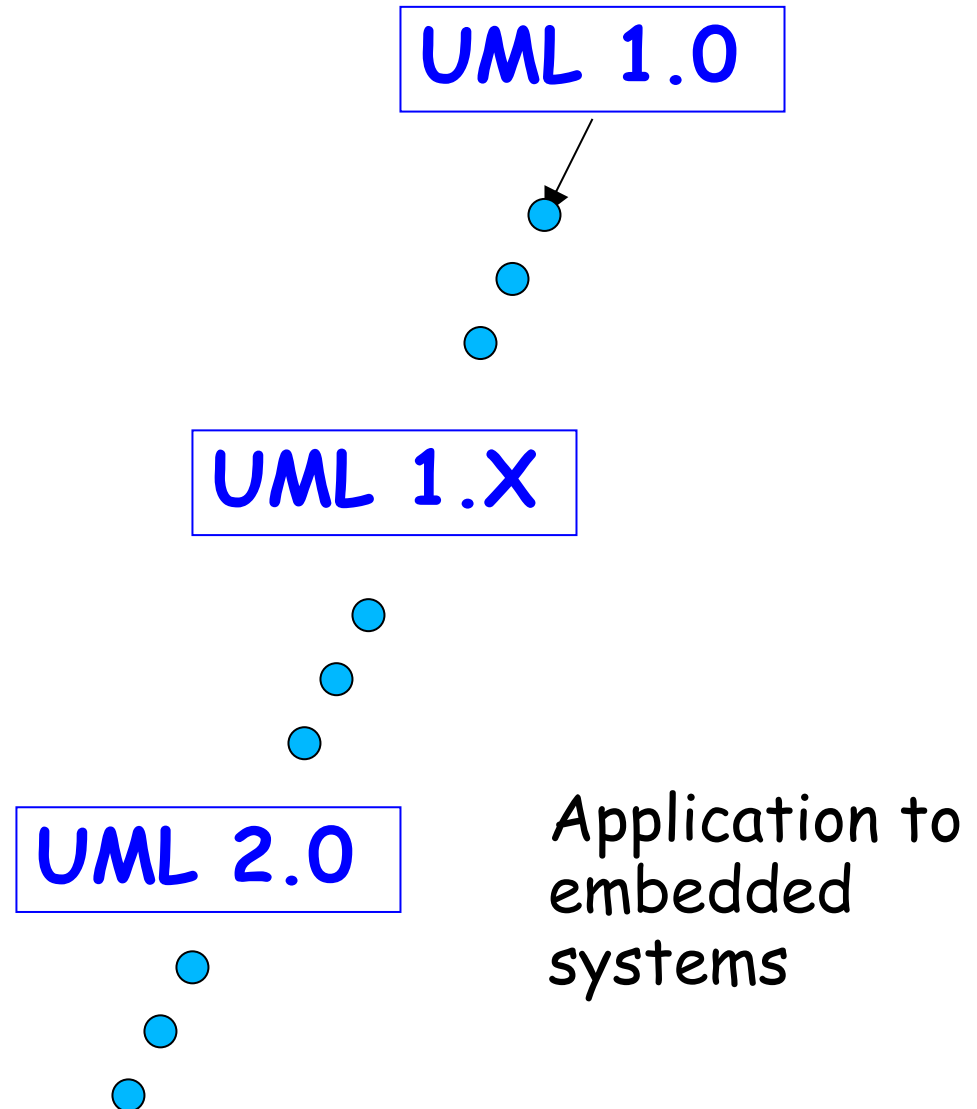


# Developments to UML

- UML continues to develop:

 Refinements

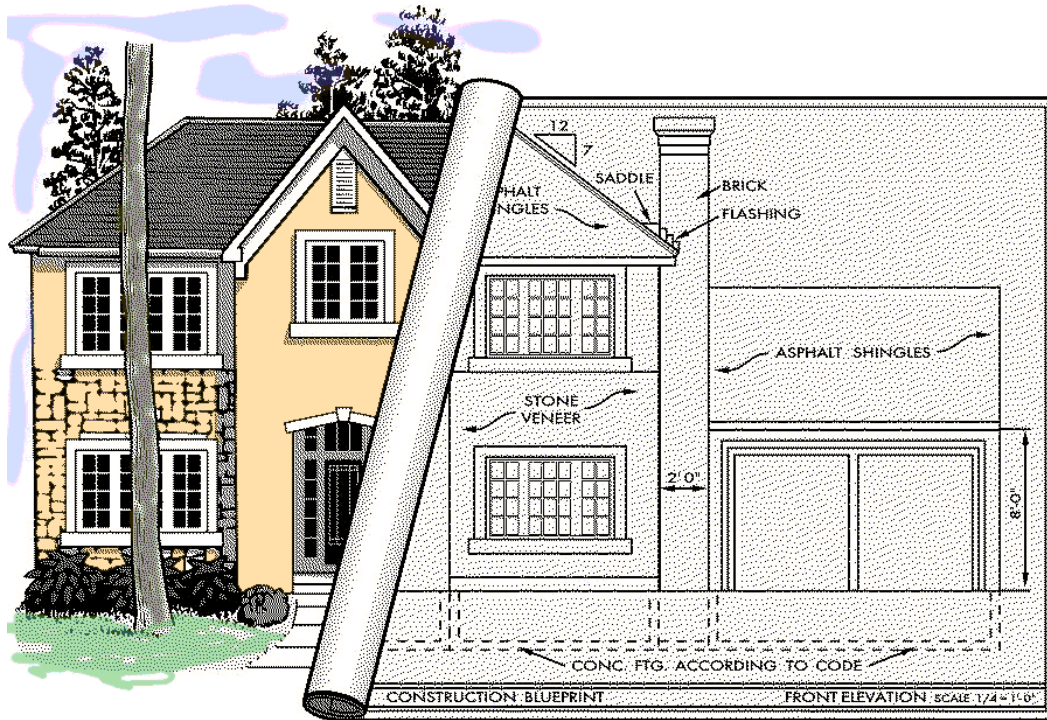
 Making it applicable to new contexts




# Why are UML Models Required?

- A model is an abstraction mechanism:
  - 📁 Capture only important aspects and ignores the rest.
  - 📁 Different models result when different aspects are ignored.
  - 📁 An effective mechanism to handle complexity.
- UML is a graphical modelling tool
- Easy to understand and construct

# Modeling a House



# UML Diagrams

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

# UML Model Views

- Views of a system:

-  User's view

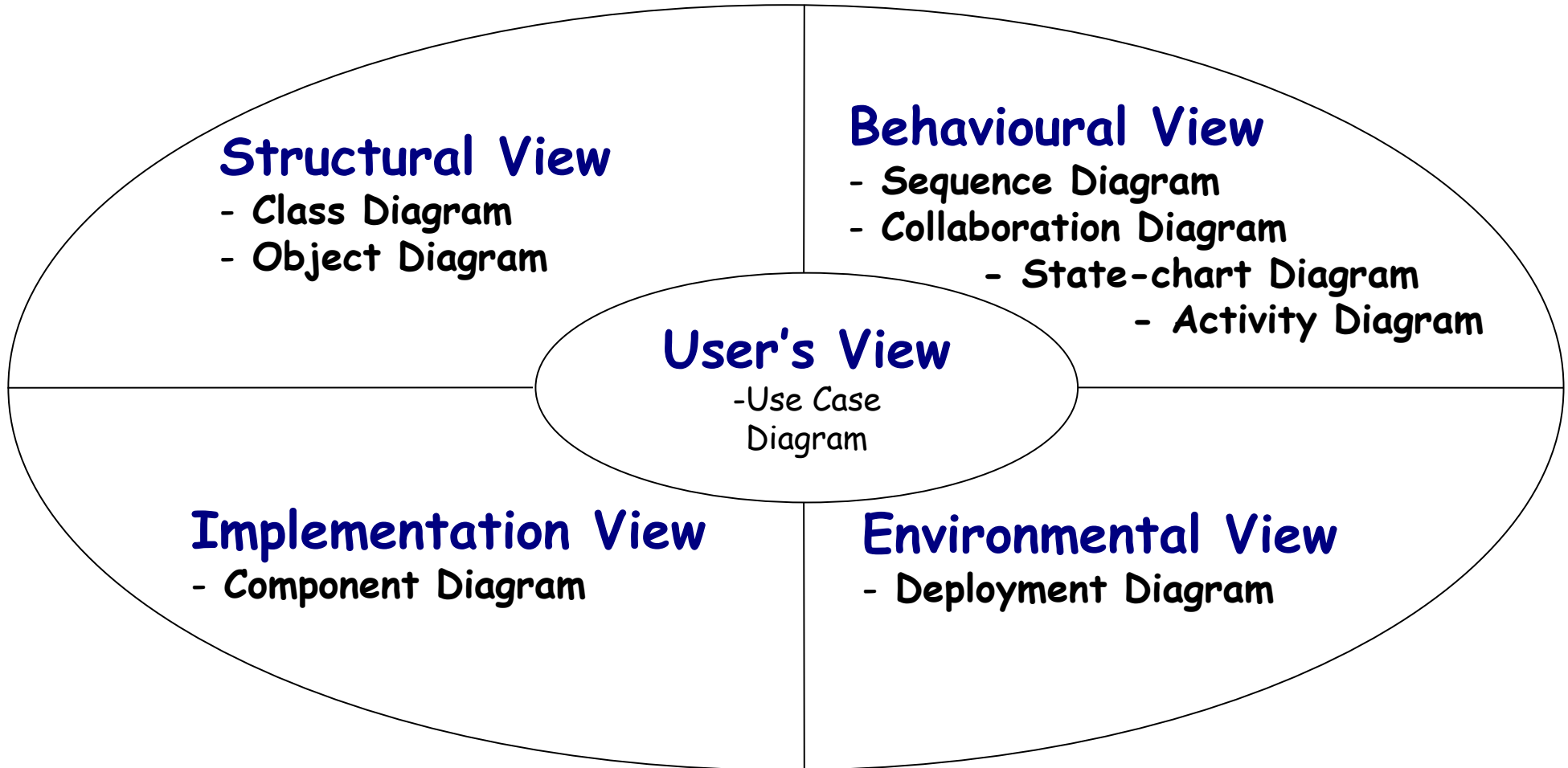
-  Structural view

-  Behavioral view

-  Implementation view

-  Environmental view

# UML Diagrams



Diagrams and views in UML

# Are All Views Required for Developing A Typical System?

- **NO**

- Use case diagram, class diagram and one of the interaction diagram for a simple system
- State chart diagram required to be developed when a class state changes
- However, when states are only one or two, state chart model becomes trivial
- Deployment diagram in case of large number of hardware components used to develop the system

# Use Case Model

- Consists of set of "use cases"
- An important analysis and design artifact
- The central model:
  - 📁 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 a system can be used by the users
- Corresponds to the high-level requirements
- Represents transaction between the user and the system
- Defines external behavior without revealing internal structure of system
- Set of related scenarios tied together by a common goal.

# Use Cases

Cont...

- 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 Use Cases

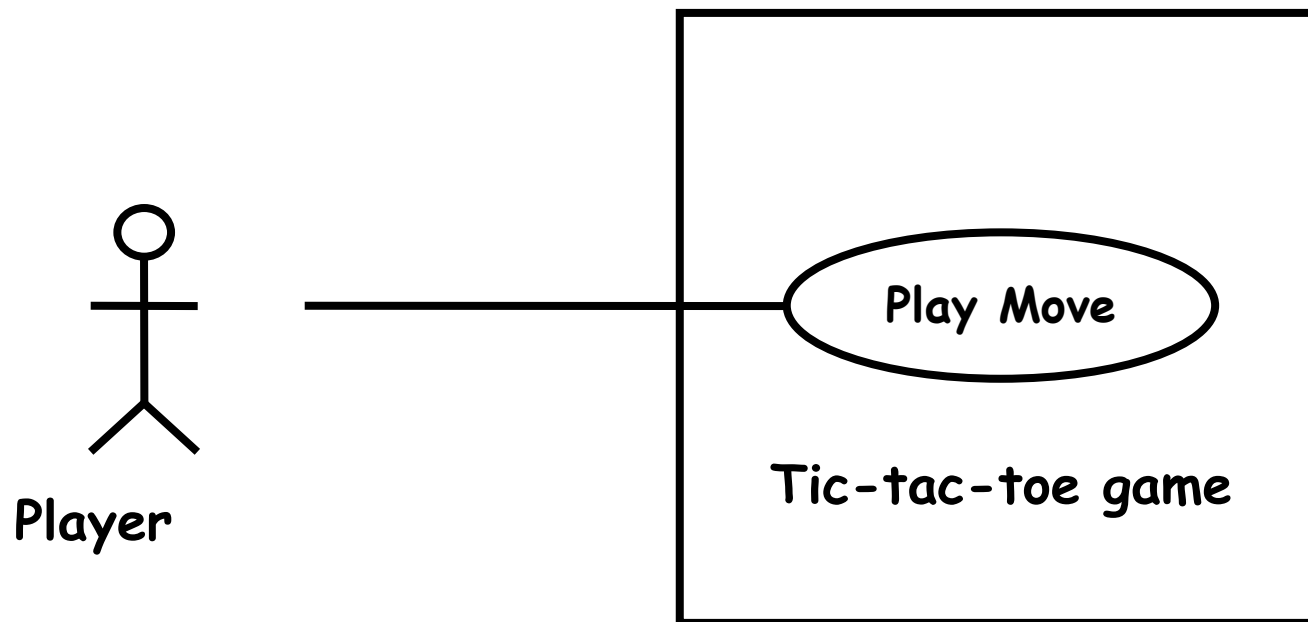
 For library information system

- issue-book
- query-book
- return-book
- create-member
- add-book, etc.

# Representation of Use Cases

- 📁 Represented by use case diagram
- 📁 A use case is represented by an ellipse
- 📁 System boundary is represented by a rectangle
- 📁 Users are represented by stick person icons (actor)
- 📁 Communication relationship between actor and use case by a line
- 📁 External system by a stereotype

# An Example Use Case Diagram



Use case model

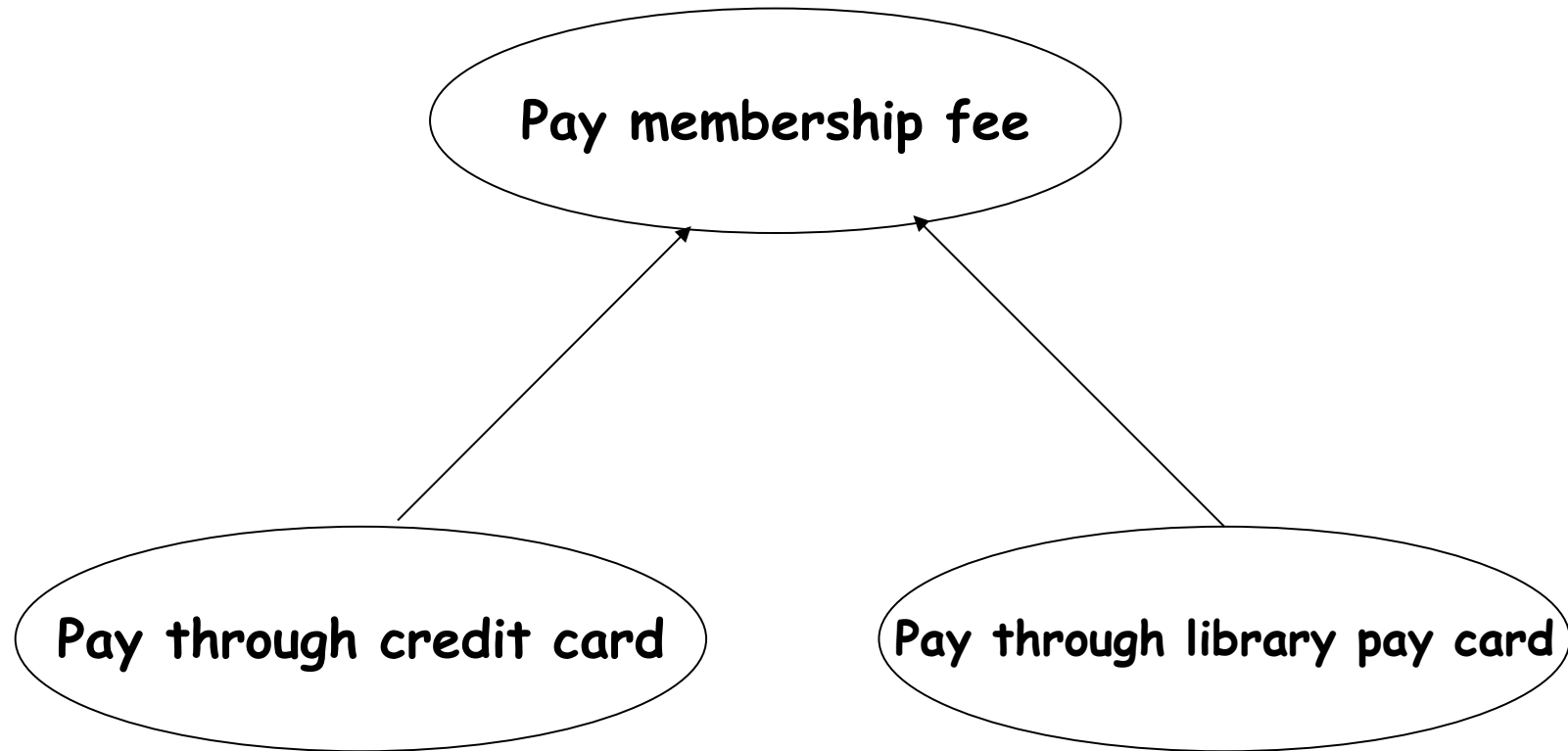
# Why Develop A Use Case Diagram?

- Serves as requirements specification
- How are actor identification useful in software development:
  - 📁 User identification helps in implementing appropriate interfaces for different categories of users
  - 📁 Another use in preparing appropriate documents (e.g. **user's manual**).

# Factoring Use Cases

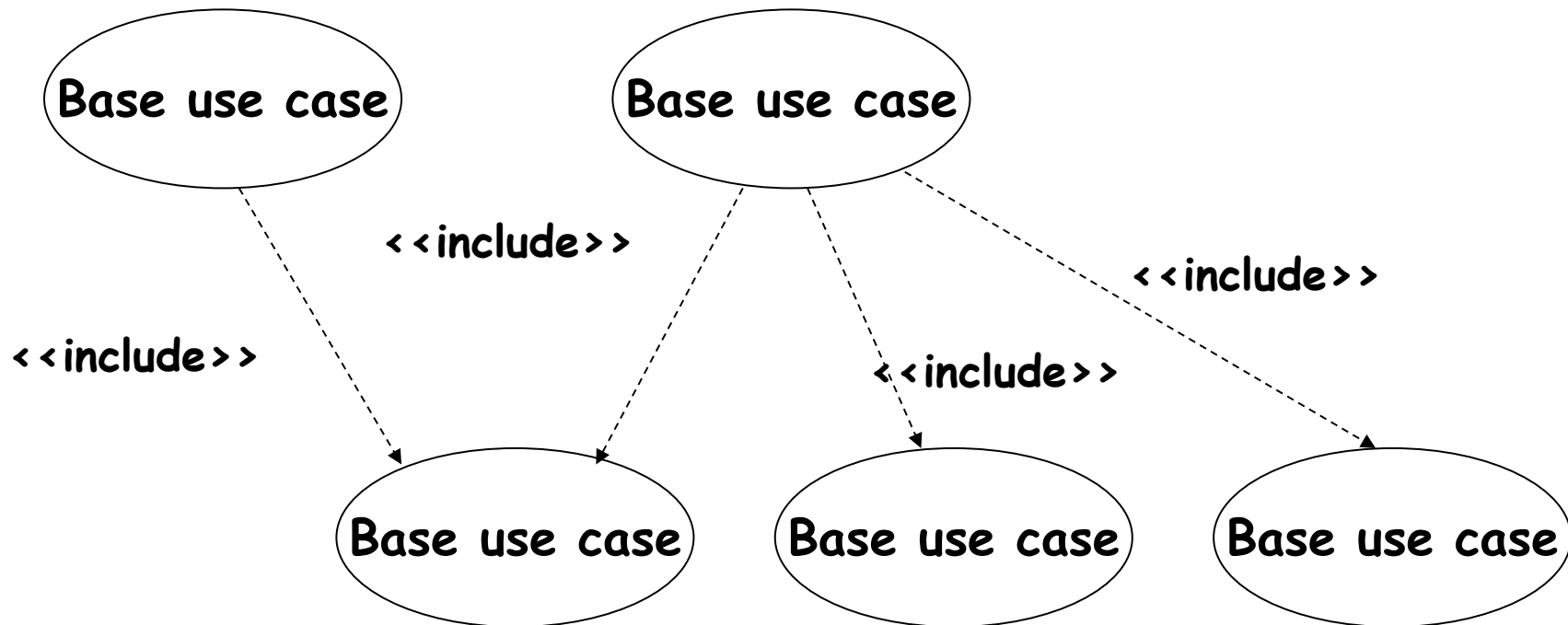
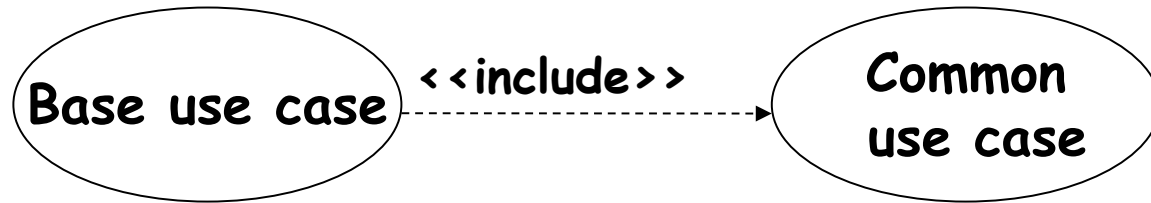
- Two main reasons for factoring:
  - 📁 Complex use cases need to be factored into simpler use cases
  - 📁 To represent common behavior across different use cases
- Three ways of factoring:
  - 📁 Generalization
  - 📁 Includes
  - 📁 Extends

# Factoring Use Cases Using Generalization





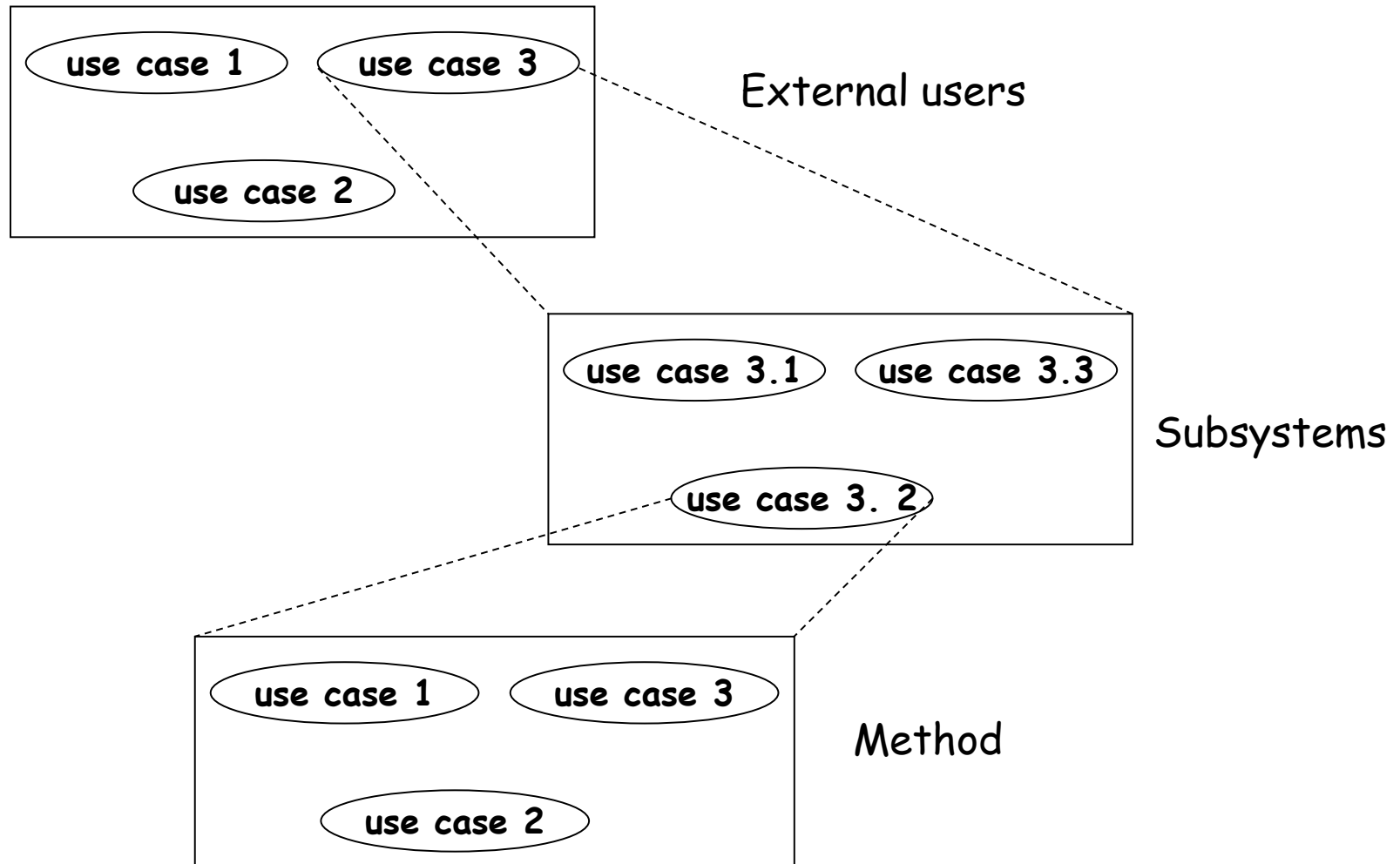
# Factoring Use Cases Using Includes



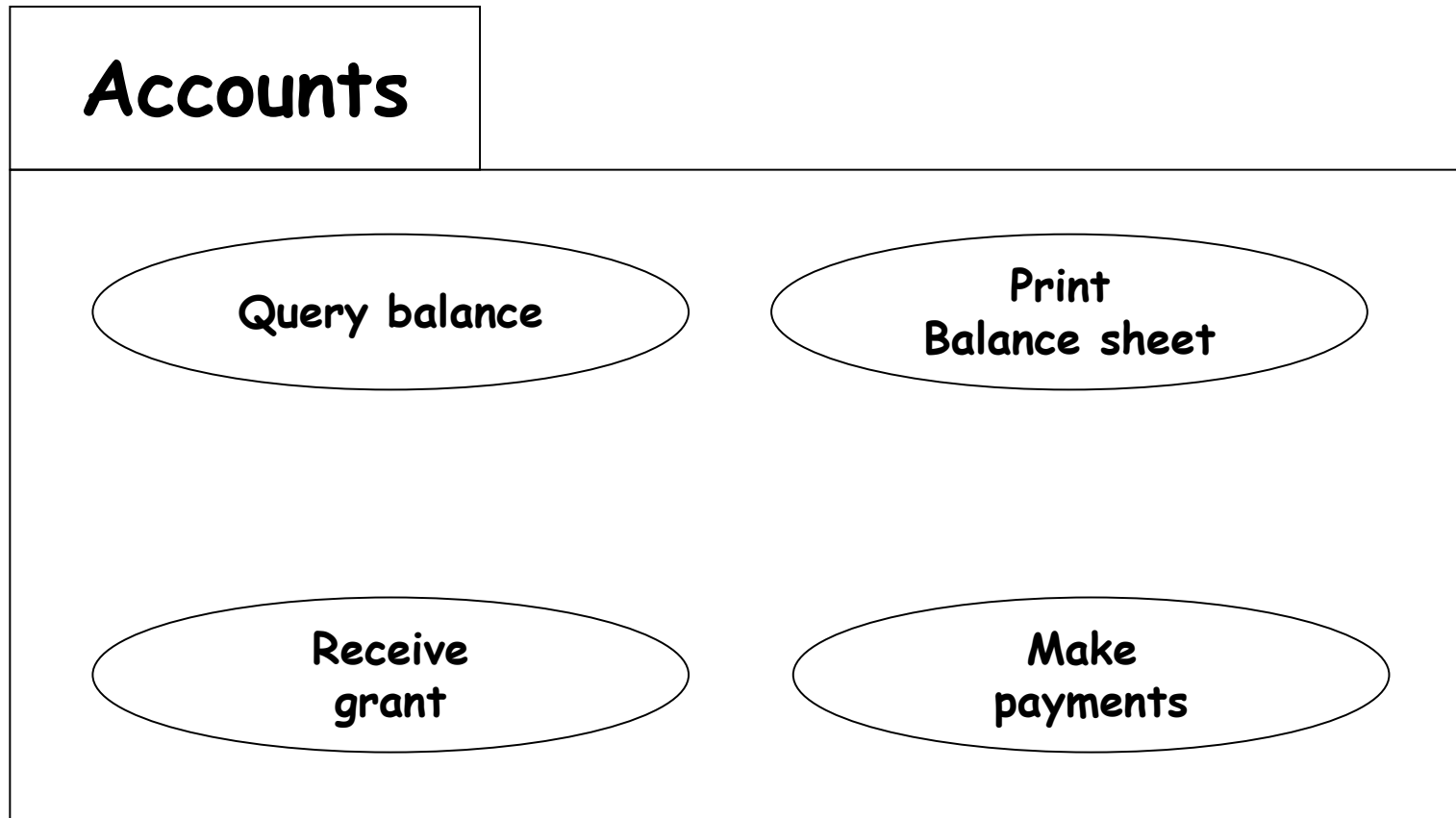
# Factoring Use Cases Using Extends



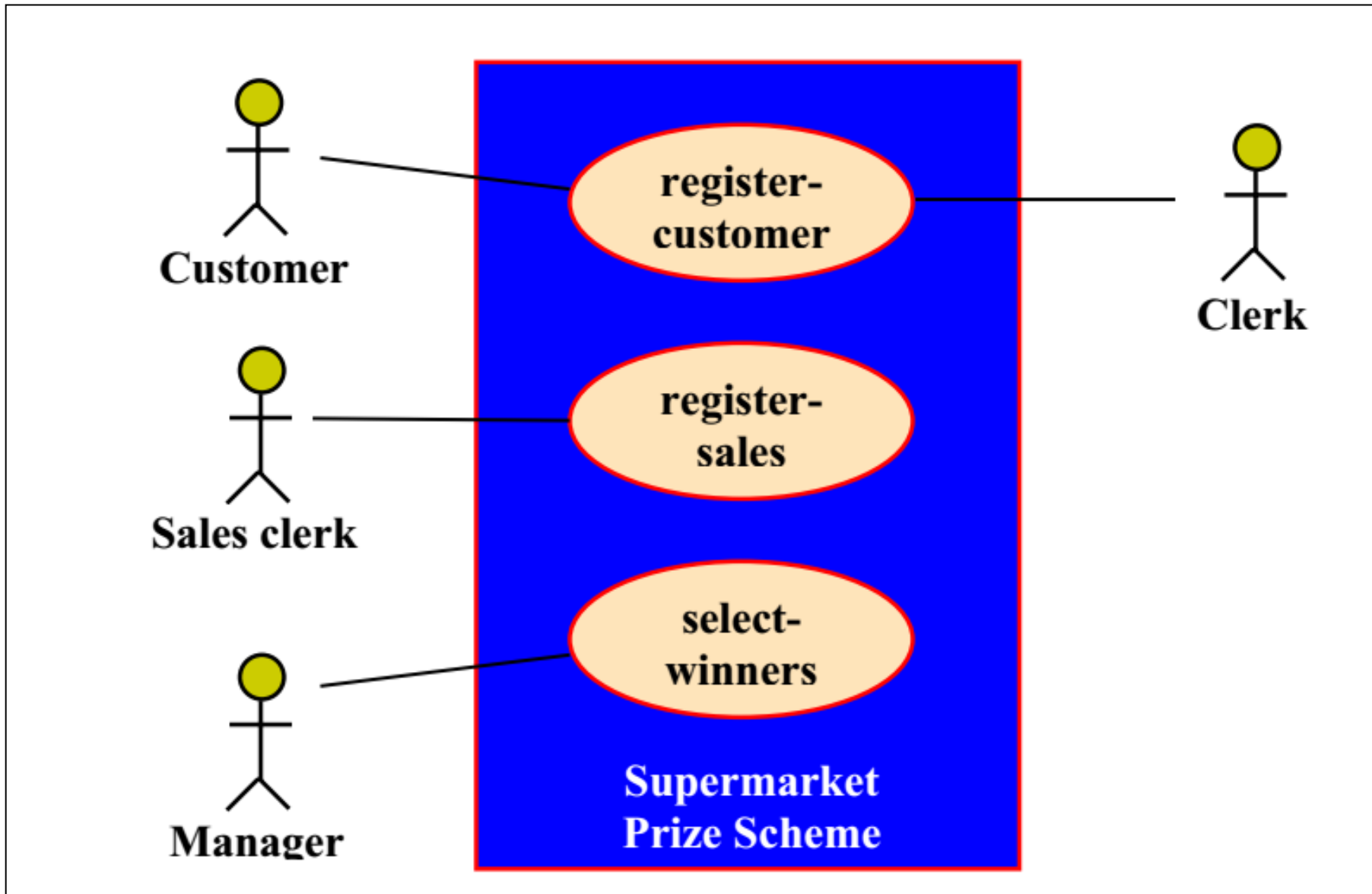
# Hierarchical Organization of Use Cases



# Use Case Packaging



# Use Case Diagram for Supermarket Prize scheme



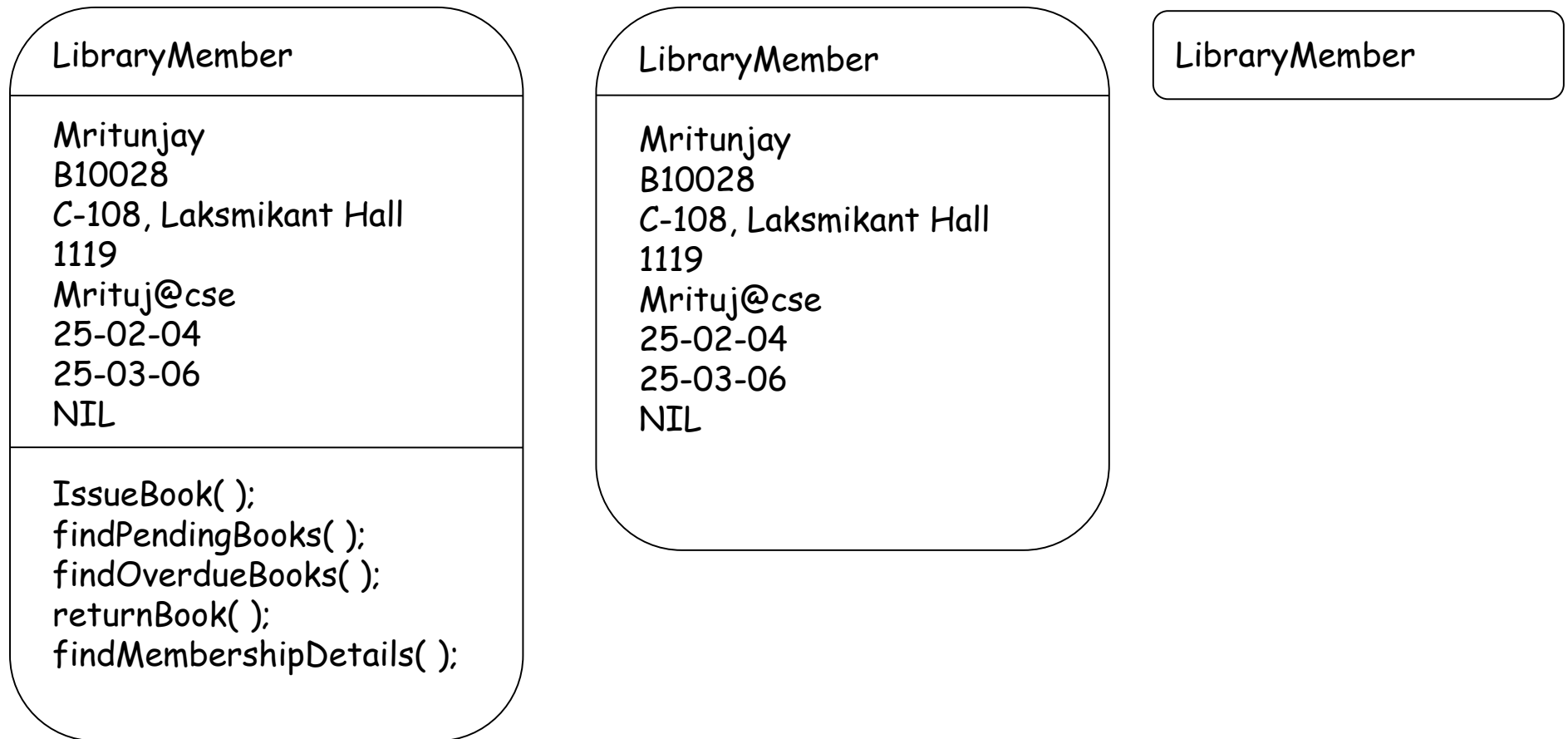
# Class Diagram

- 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, and operations.
- Attribute and operation compartments are optional depending on the purpose of a diagram.

# 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

# Interaction Diagram

- Two kinds: *Sequence* and *Collaboration* diagrams.
- Two diagrams are equivalent
  - 📖 Portray different perspectives
- These diagrams play a very important role in the design process.

# Sequence Diagram

- Shows interaction among objects as a 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 a **rectangle** on lifeline

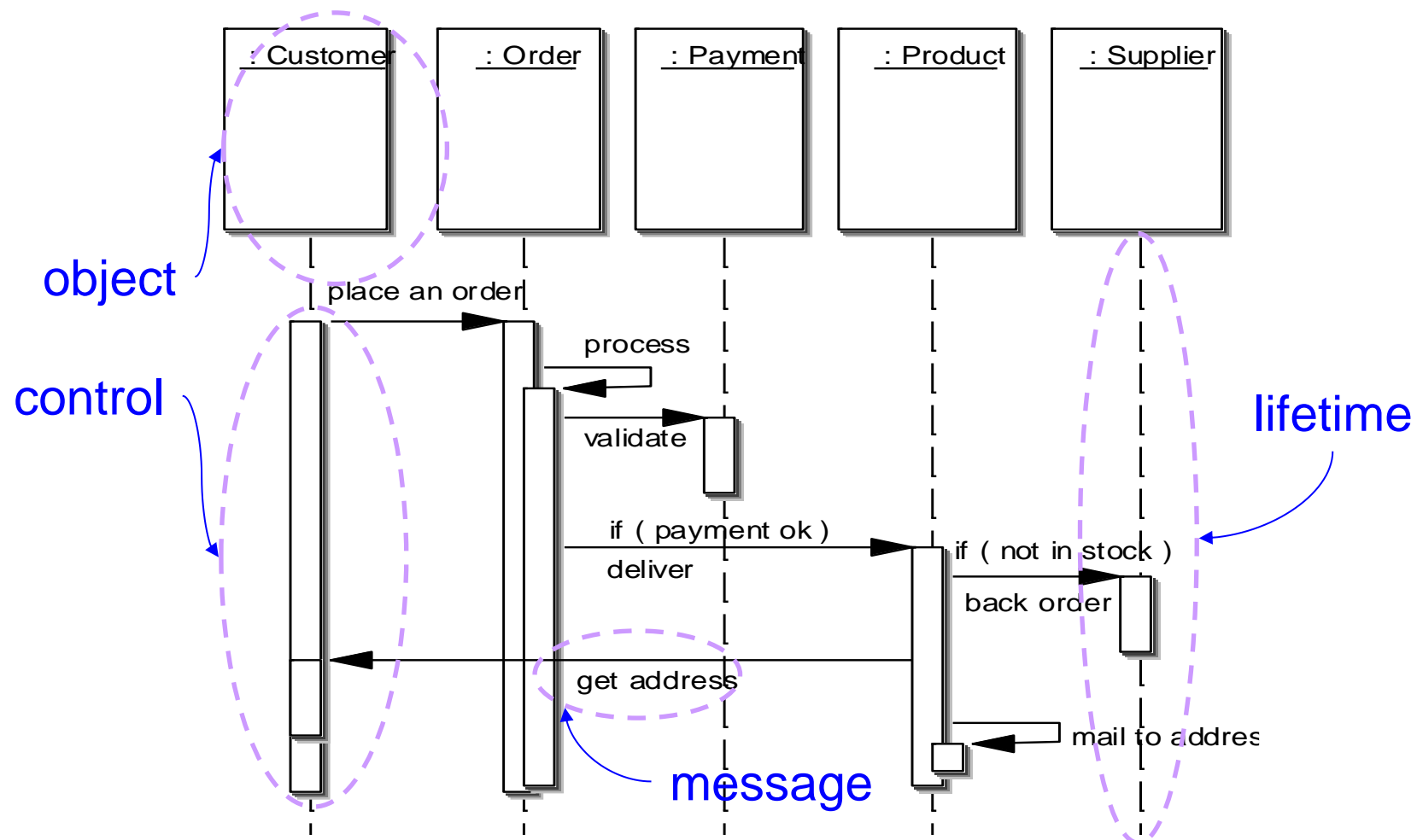
# Sequence Diagram Cont...

- Messages are shown as arrows
- Each message labelled with corresponding message name
- Each message can be labelled with some control information
- Two types of control information

 condition ([ ])

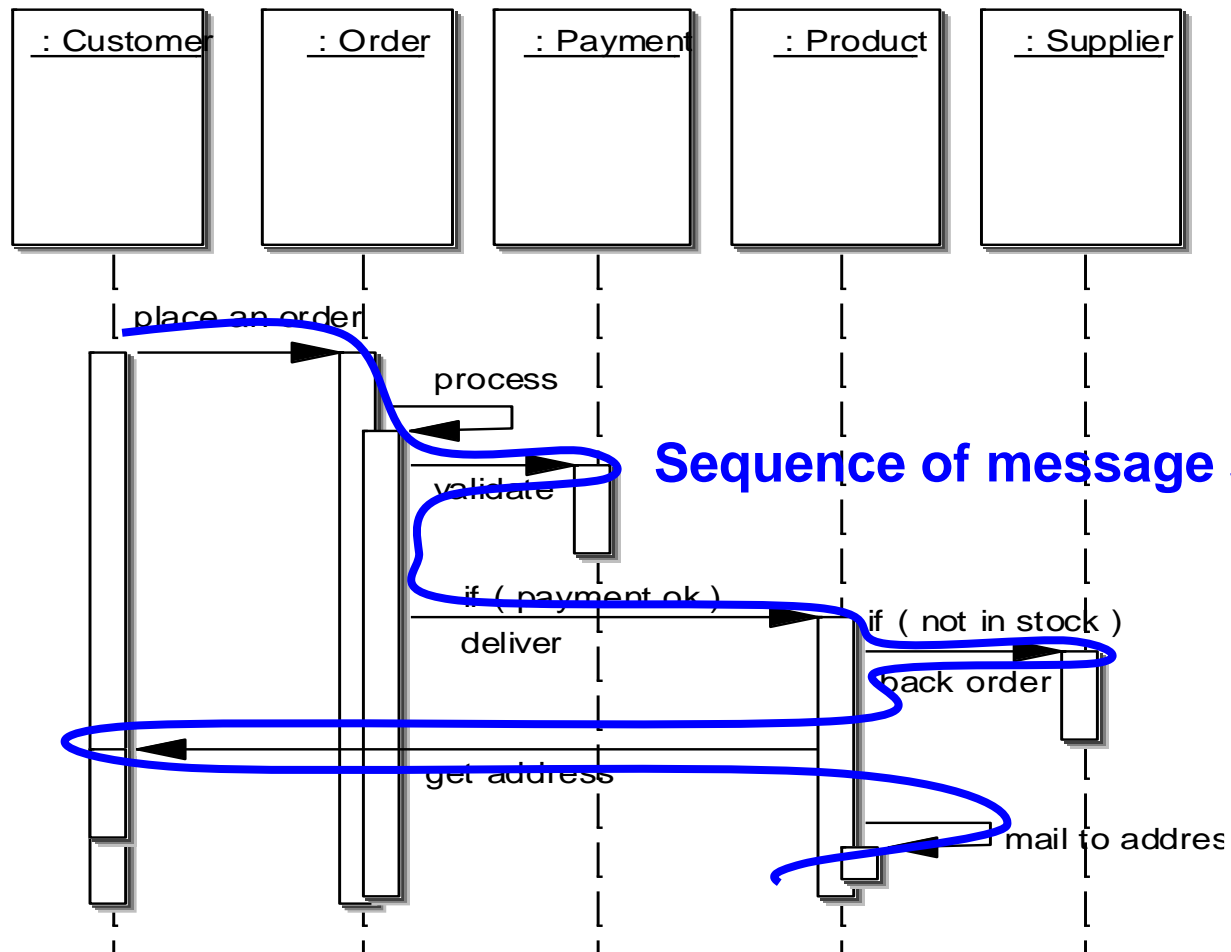
 iteration (\*)

# Elements of a Sequence Diagram

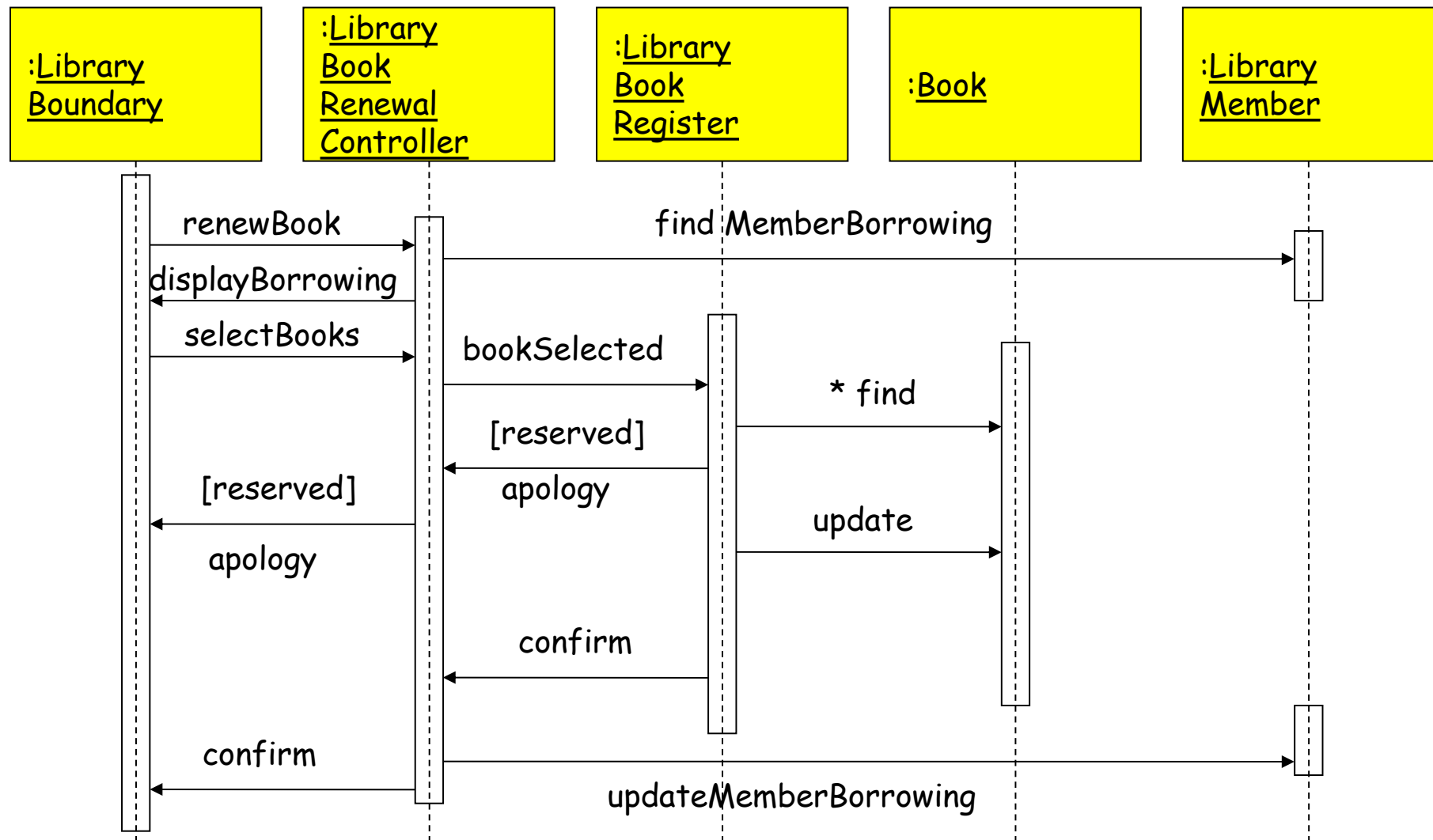


# Example

Cont...



# An Example of A Sequence Diagram



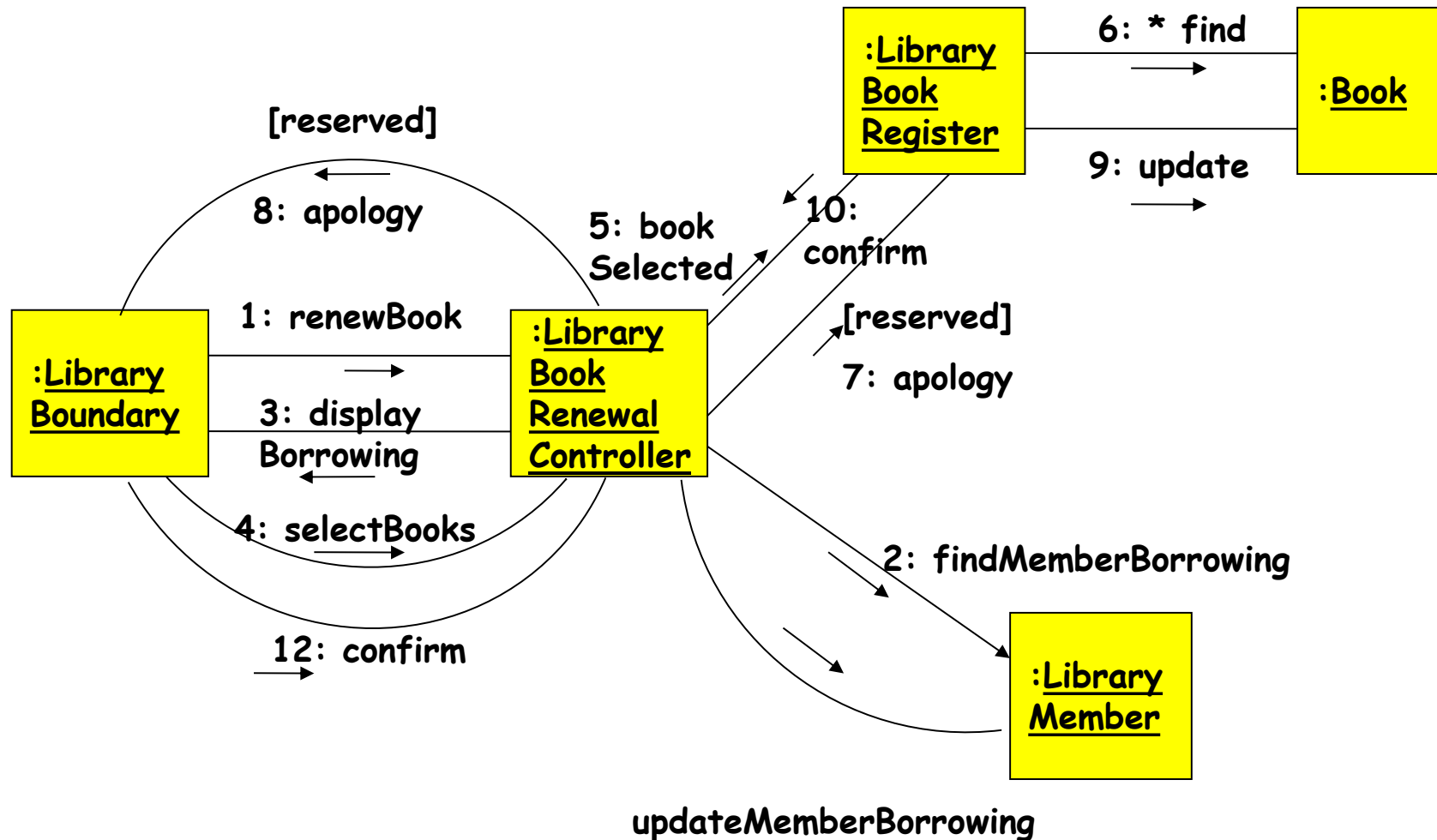
Sequence Diagram for the renew book use case

# Collaboration Diagram

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



# An Example of A Collaboration Diagram



Collaboration Diagram for the renew book use case

# Activity Diagram

- Not present in earlier modelling techniques:
  - 📖 Possibly based on event diagram of [Odell](#) [1992]
- Represents processing activity, may not correspond to methods
- Activity is a state with an internal action and one/many outgoing transitions
- Somewhat related to flowcharts

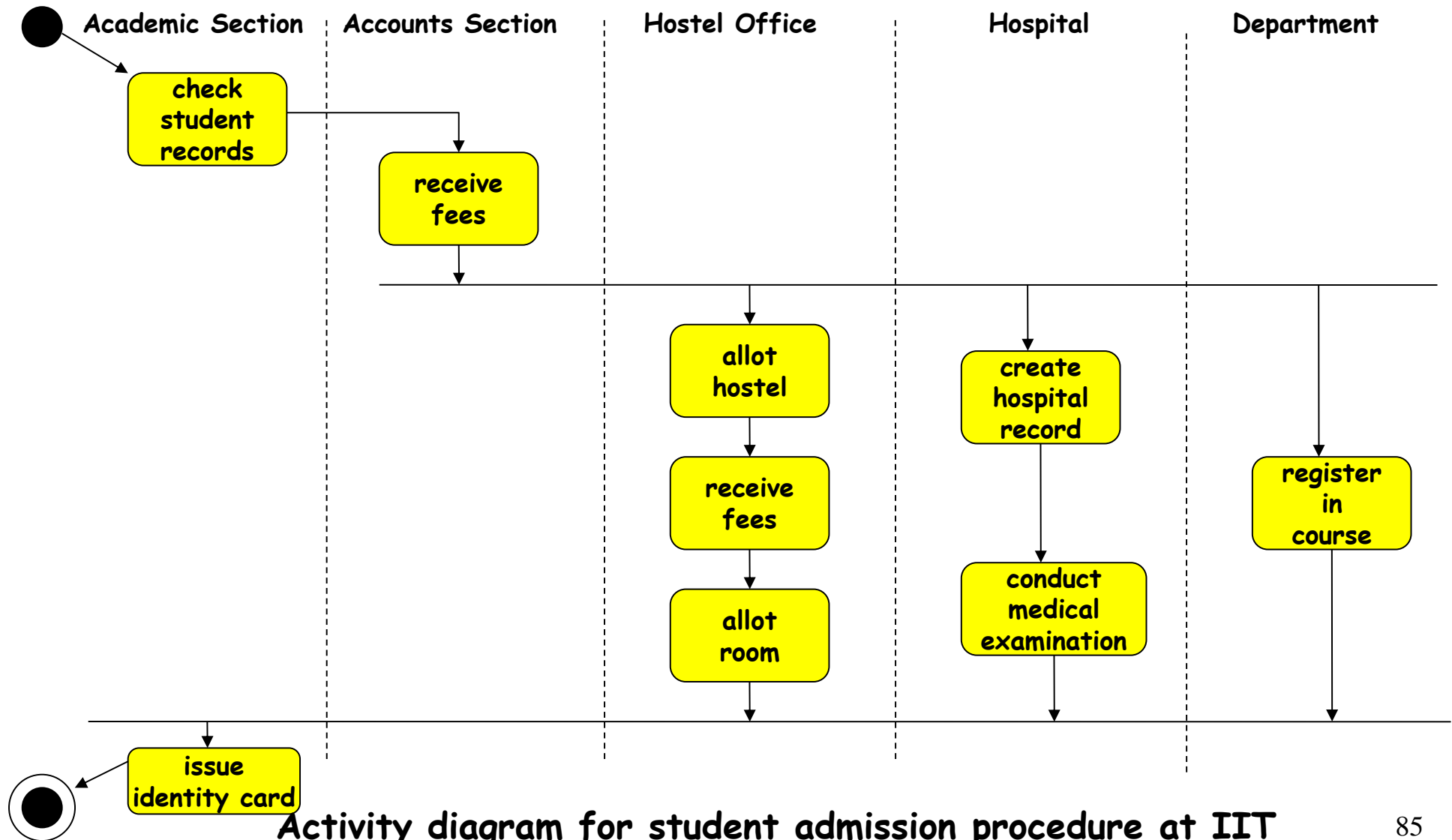
# Activity Diagram vs Flow Chart

- Can represent parallel activity and synchronization aspects
- **Swim lanes** can be used to group activities based on who is performing them
- Example: academic department vs. hostel

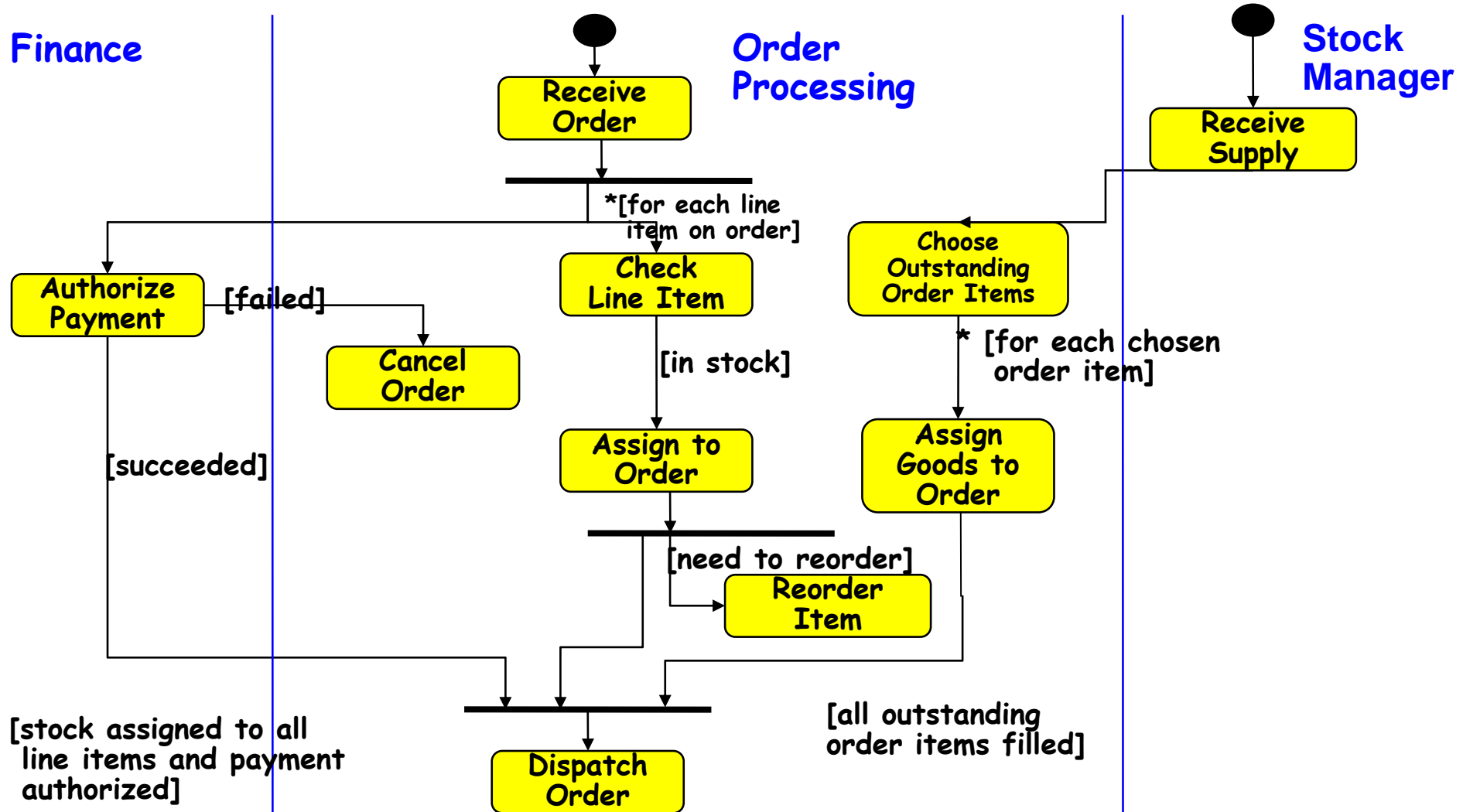
# Activity Diagram

- Normally employed in business process modelling.
- Carried out during requirements analysis and specification stage.
- Can be used to develop interaction diagrams.

# An Example of An Activity Diagram



# Activity Diagram: Example 2




# State Chart Diagram

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

# State Chart Diagram

Cont...

- State chart avoids the problem of state explosion of FSM.
- Hierarchical model of a system:
  -  Represents composite **nested** states

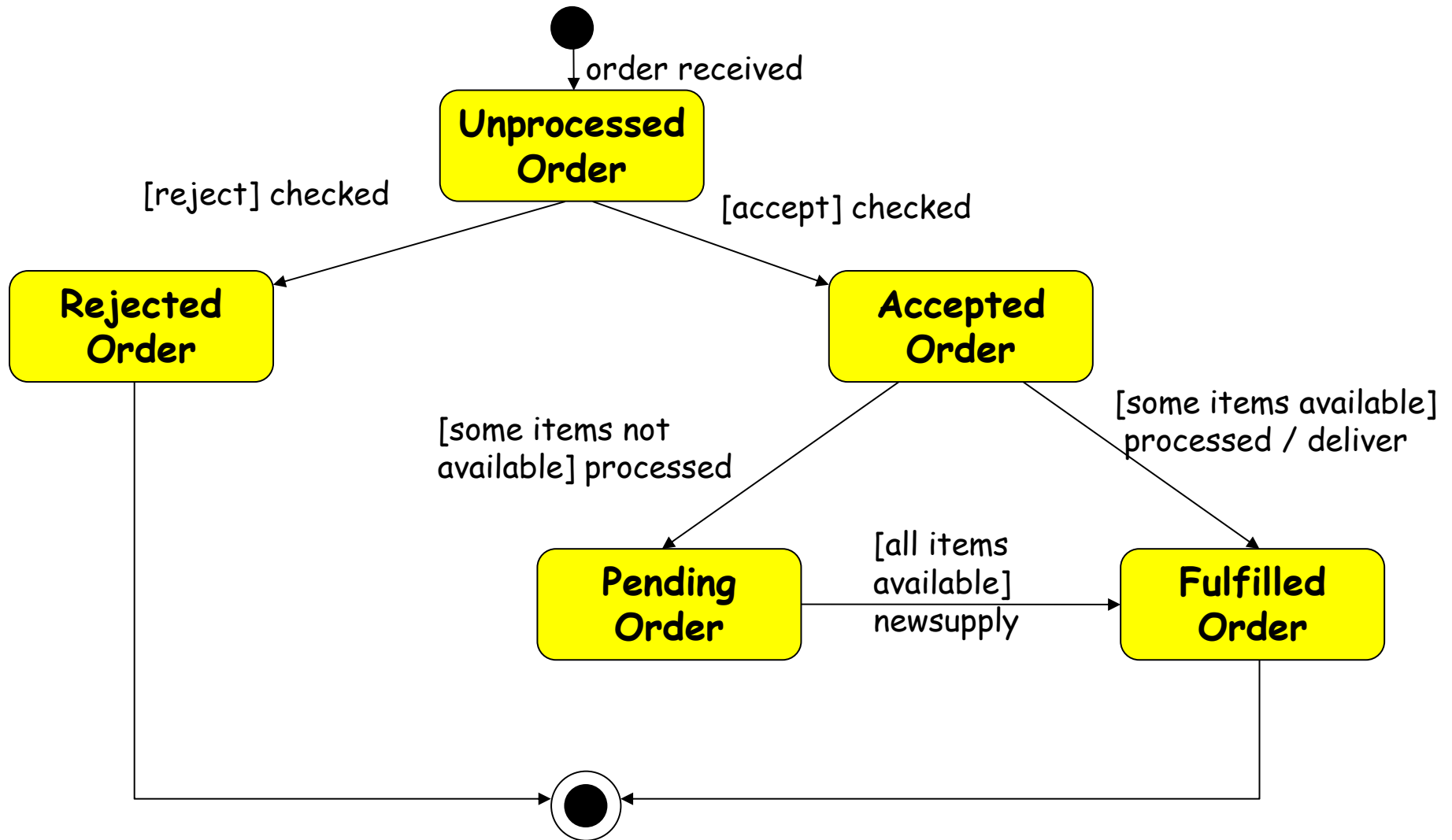


# State Chart Diagram

Cont...

- 📁 Elements of state chart diagram
- 📁 Initial State: A filled circle
- 📁 Final State: A filled circle inside a larger circle
- 📁 State: Rectangle with rounded corners
- 📁 Transitions: Arrow between states, also boolean logic condition (*guard*)

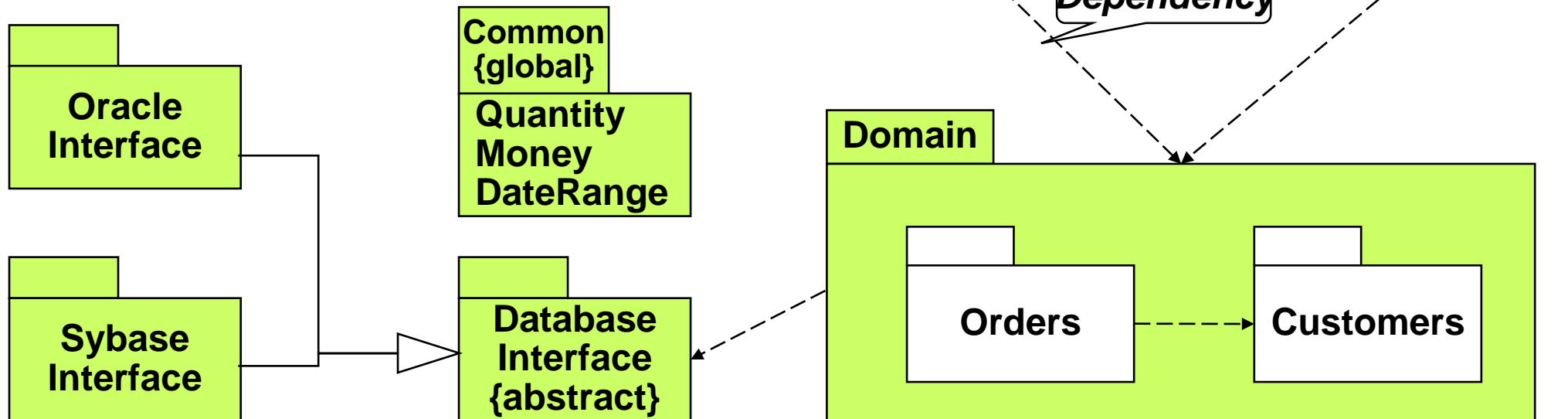
# An Example of A State Chart Diagram



Example: State chart diagram for an order object

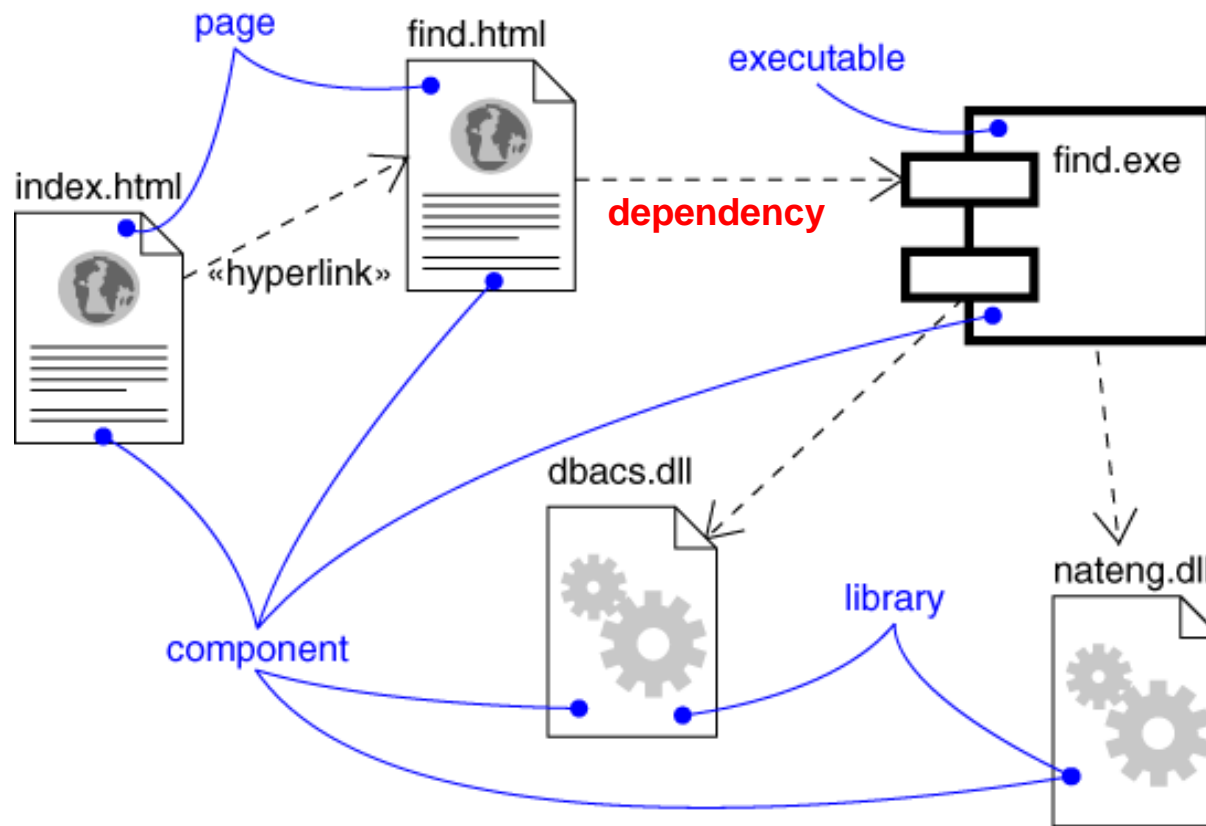
# Package Diagrams

- A package is a grouping of several classes:
  - Java packages are a good example
- Package diagrams show module dependencies.
- Useful for large projects with multiple binary files



# Component Diagram

- Captures the physical structure of the implementation (code components)



## Components:

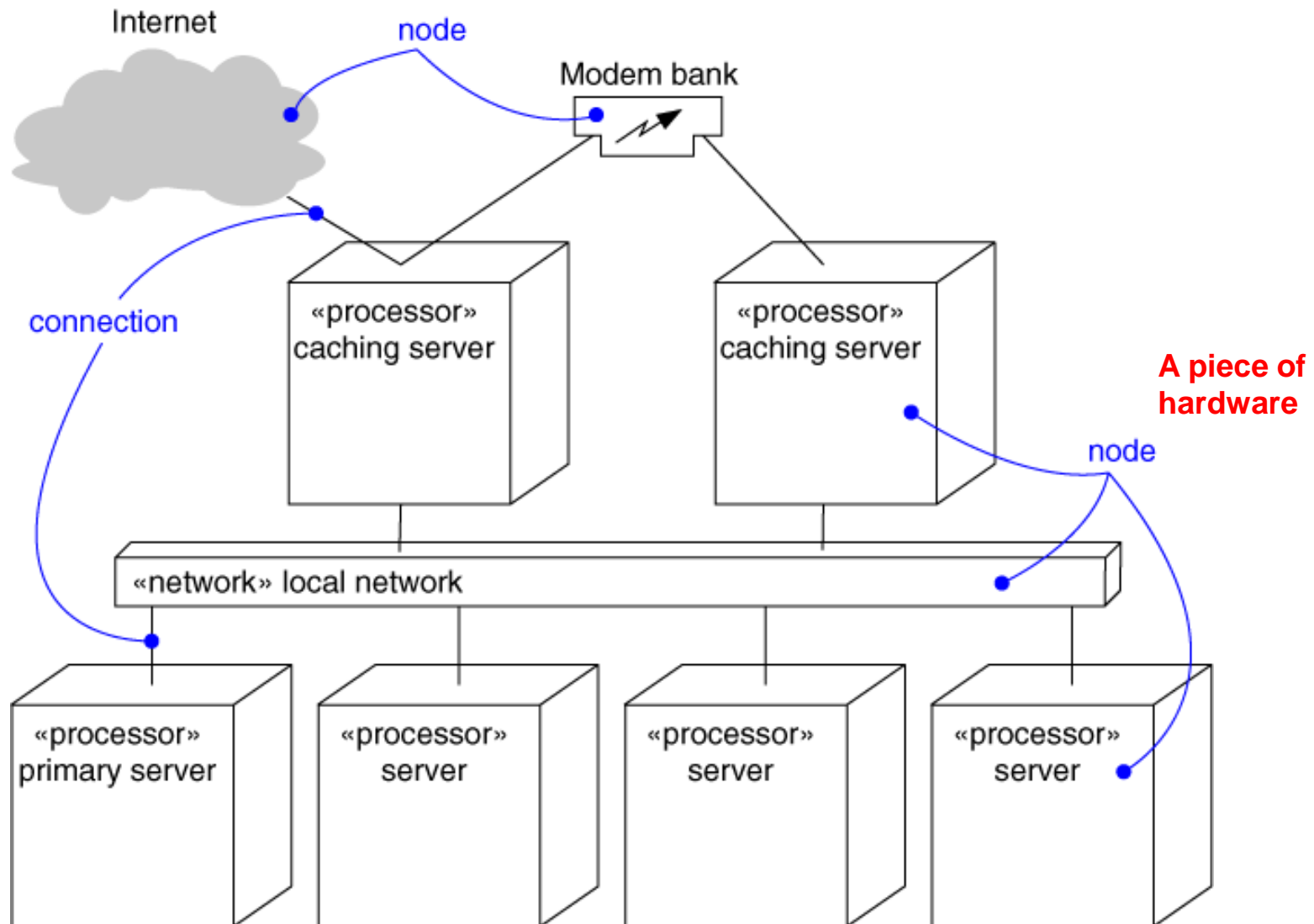
- Executables
- Library
- Table
- File
- Document

# Component Diagram

- Captures the physical structure of the implementation
- Built as part of architectural specification
- Purpose
  - 📁 Organize source code
  - 📁 Construct an executable release
  - 📁 Specify a physical database
- Developed by architects and programmers

# Deployment Diagram

- Captures the topology of a system's hardware



# A Design Process

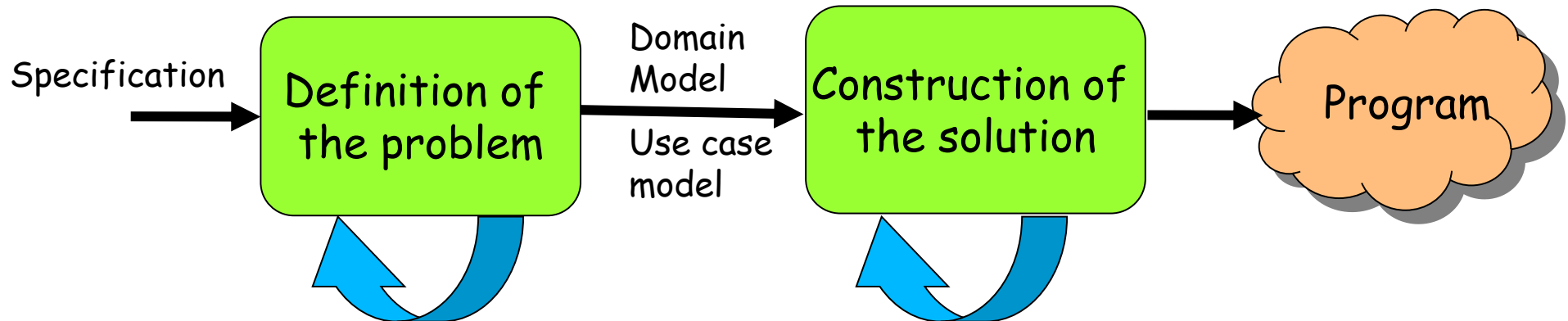
- 📁 Developed from various methodologies.
  - However, UML has been designed to be usable with any design methodology.
- 📁 From requirements specification, initial model is developed (OOA)
  - Analysis model is iteratively refined into a design model
- 📁 Design model is implemented using OO concepts

# OOAD

Iterative and Incremental

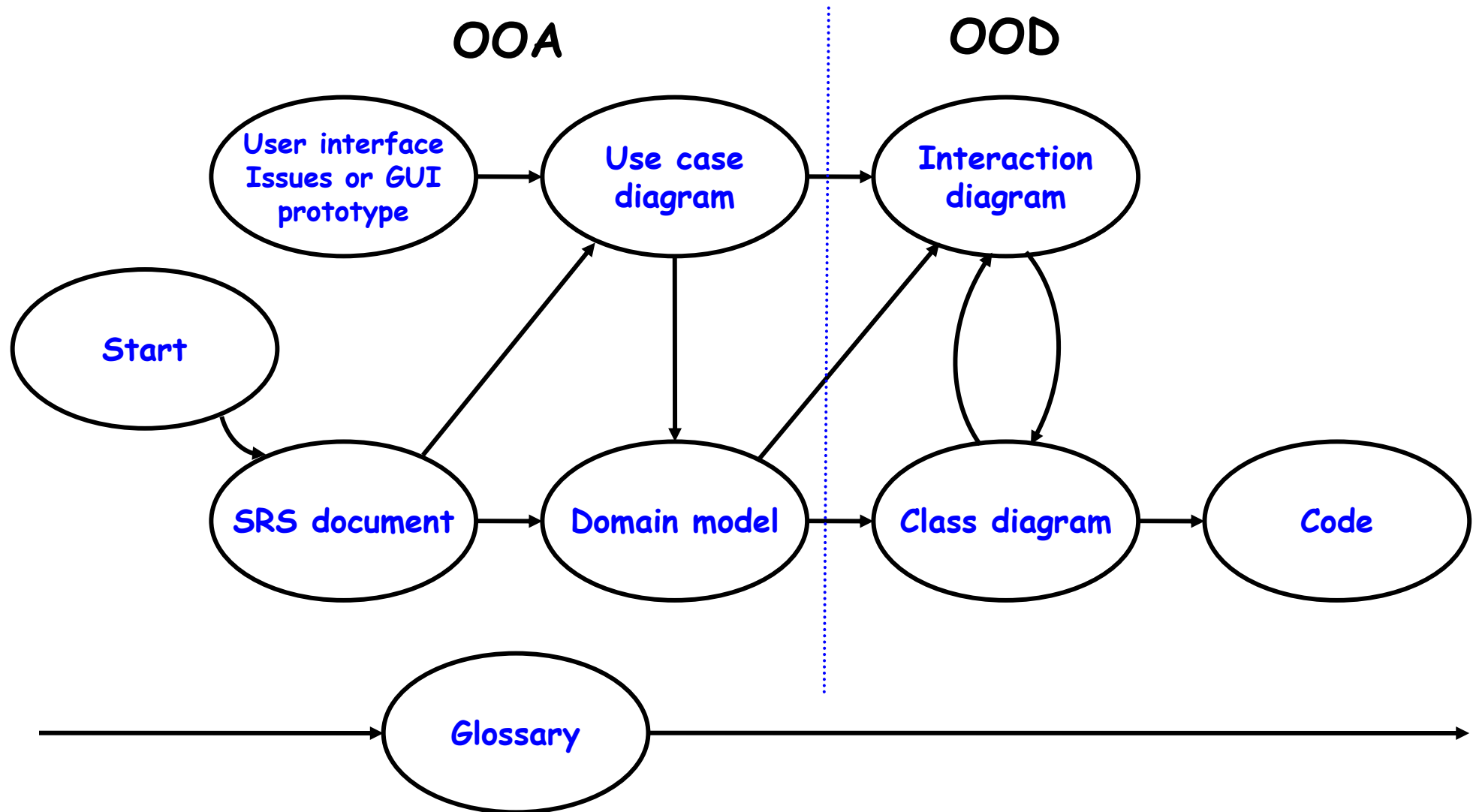
OOA

OOD/OOP





# Unified Development Process Cont...



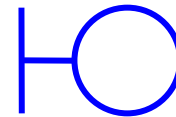
# Domain Modelling

- 📁 Represents 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

# Class Stereotypes

Three different stereotypes on classes are used: <<boundary>>, <<control>>, <<entity>>.

*Boundary*



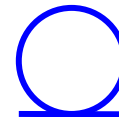
Cashier Interface

*Control*



Withdrawal

*Entity*




Account

# Boundary Objects

 Interact with actors:

- User interface objects

 Include screens, menus, forms, dialogs etc.

 Do not perform processing but validates, formats etc.

# Entity Objects

📁 Hold information:

- Such as data tables & files, e.g. Book, BookRegister

📁 Normally are dumb servers

📁 Responsible for storing data, fetching data etc.

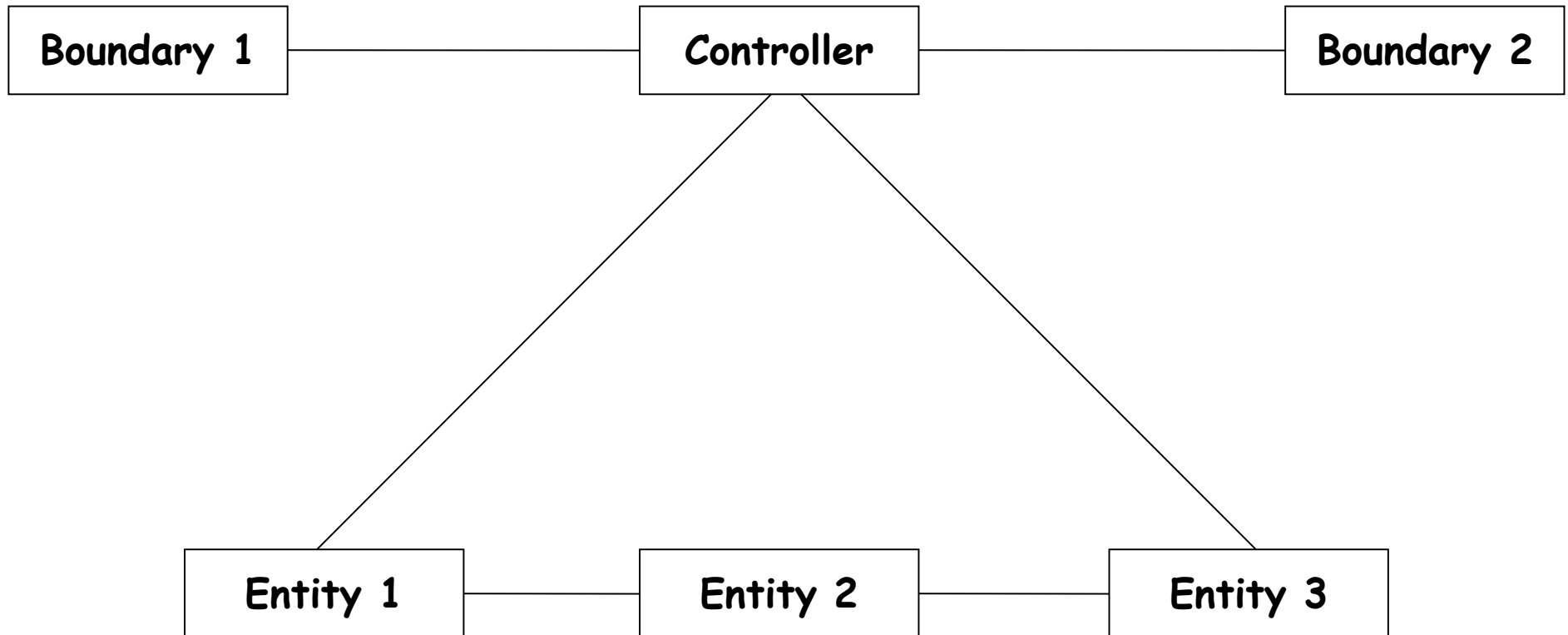
📁 Elementary operations on data such as searching, sorting, etc.

📁 **Entity Objects** are identified by examining **nouns** in problem description

# Controller Objects

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

# Use Case Realization



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

# Class-Responsibility-Collaborator(CRC) Cards

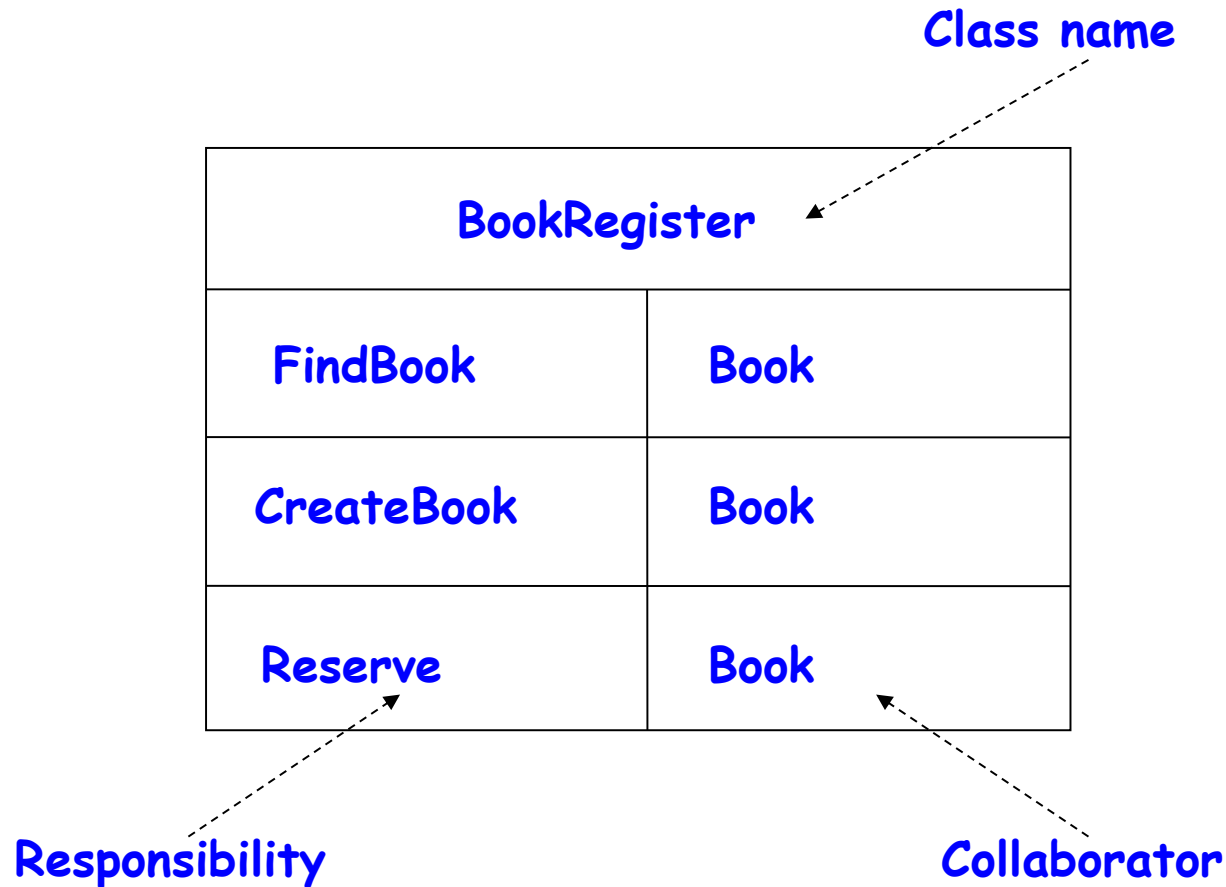
- 📁 Pioneered by Ward Cunningham and Kent Beck
- 📁 Index cards prepared one each per class
- 📁 Class responsibility is written on these cards
- 📁 Collaborating object is also written



# CRC Cards Cont...

- 📁 Required for developing interaction diagram of complex use cases
- 📁 Team members participate to determine:
  - The responsibility of classes involved in the use case realization

# An Example of A CRC Card



CRC card for the BookRegister class

# Patterns versus Idioms

- A pattern:
  - 📖 Describes a recurring problem
  - 📖 Describes the *core* of a solution
  - 📖 Is capable of generating many distinct designs
- An Idiom is more restricted
  - 📖 Still describes a recurring problem
  - 📖 Provides a more specific solution, with fewer variations
  - 📖 Applies only to a narrow context
    - e.g., the C++ language

# Patterns

- The essential idea:

 If you can master a few important patterns, you can easily spot them in application development and use the pattern solutions.

# Idioms

- In English:

- 📖 A group of words that has meaning different from a simple juxtaposition of the meanings of the individual words.

- 📖 "Raining cats and dogs"

- A C idiom:

- 📖 

```
for(i=0;i<1000;i++){  
      
}
```


# Antipattern


- If a pattern represents a best practice:
  - 📖 Antipattern represents lessons learned from a bad design.
- Antipatterns help to recognise deceptive solutions:
  - 📖 That appear attractive at first, but turn out to be a liability later.

# Design Patterns

- 📁 Standard solutions to commonly recurring problems
- 📁 Provides good solution based on common sense
- 📁 Pattern has four important parts
  - The problem
  - The context
  - The solution
  - The context in which it works or does not work

# Example Pattern: Expert

 **Problem:** Which class should be responsible for doing certain things

 **Solution:** Assign responsibility to the class that has the information necessary to fulfil the required responsibility



# Example Pattern: Expert Cont...





Class Diagram



Collaboration Diagram


# Example Pattern: Creator


 **Problem:** Which class should be responsible for creating a new instance of some class?

 **Solution:** Assign a class *C1* the responsibility to create class *C2* if


- *C1* is an aggregation of objects of type *C2*
- *C1* contains object of type *C2*


# Example Pattern: Controller


 **Problem:** Who should be responsible for handling the actor requests?

 **Solution:** Separate controller object for each use case.

# Example Pattern: Facade

 **Problem:** How should the services be requested from a service package?

 **Context (problem):** A package (cohesive set of classes), example: RDBMS interface package

 **Solution:** A class (DBfacade) can be created which provides a common interface to the services of the package

# Example Pattern: MVC

- Model-View-Controller
- How should the user interface (Boundary) objects interact with the other objects?
- Solution 1: Pull from Above
  - 📁 Boundary object invokes other objects.
  - 📁 Does not work when data needs to be asynchronously displayed, simulation experiment, stock market alert, network monitor, etc.

# Example Pattern: MVC

- Solution 2: Publish-Subscribe

- 📁 The boundary objects register themselves with an event manager object.

- 📁 Other objects, notify the event manager object as and when an event of interest occurs.

- 📁 The event manager notifies those boundary objects that have registered with it by using a call back.

# Example 1: Tic-Tac-Toe Computer Game

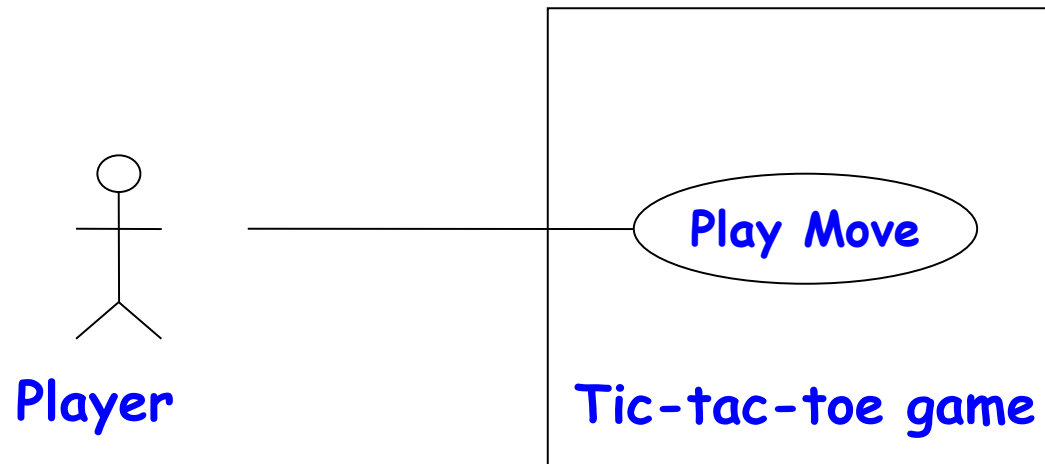
- A human player and the computer make alternate moves on a 3 3 square.
- A move consists of marking a previously unmarked square.
- The user inputs a number between 1 and 9 to mark a square
- Whoever is first to place three consecutive marks along a straight line (i.e., along a row, column, or diagonal) on the square wins.

# Example 1: Tic-Tac-Toe Computer Game

- As soon as either of the human player or the computer wins,
  - 📖 A message announcing the winner should be displayed.
- If neither player manages to get three consecutive marks along a straight line,
  - 📖 And all the squares on the board are filled up,
  - 📖 Then the game is drawn.
- The computer always tries to win a game.



# Example 1: Use Case Model



# Example 1: Initial and Refined Domain Model

Board

Initial domain model

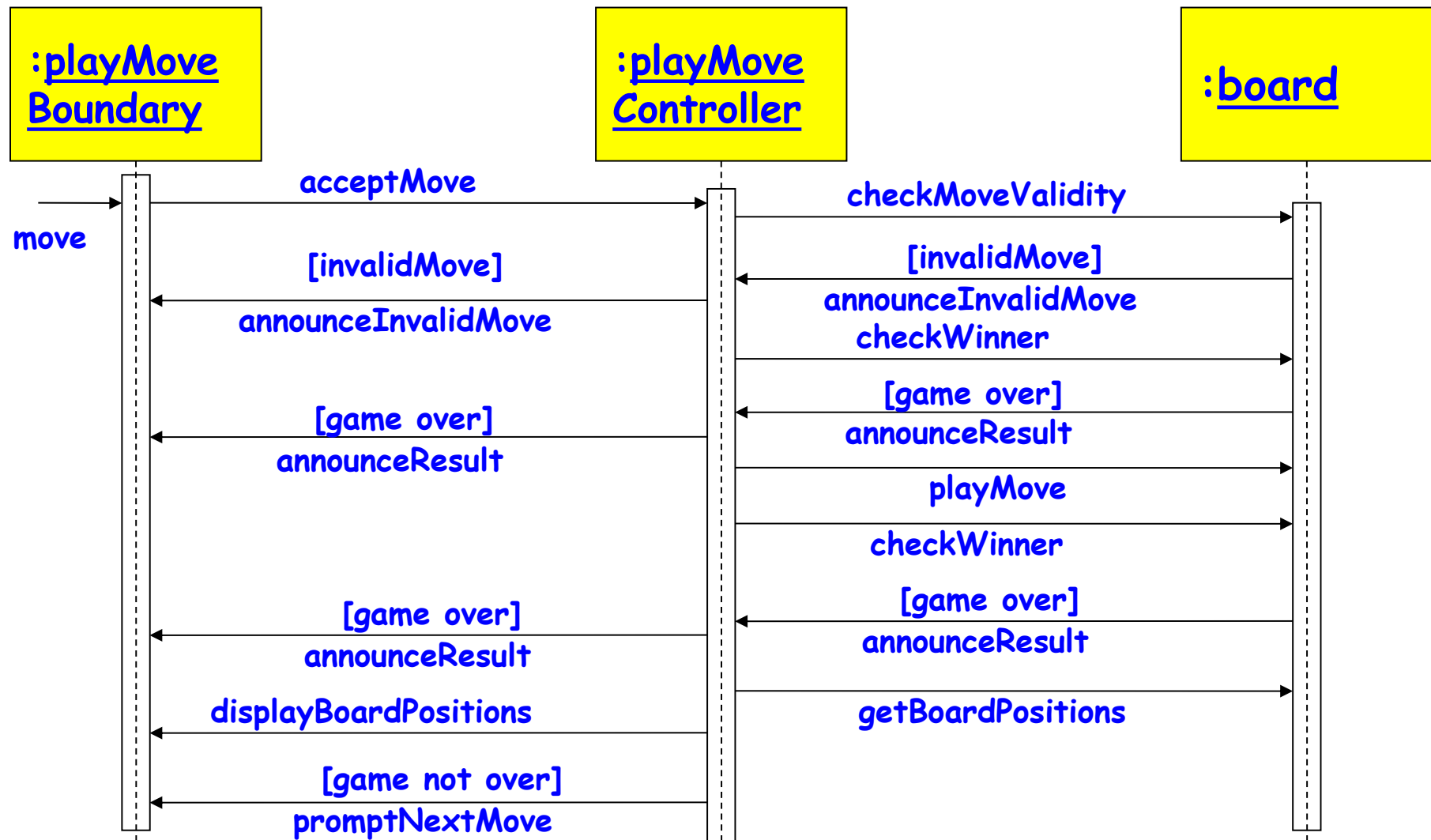
PlayMoveBoundary

PlayMoveController

Board

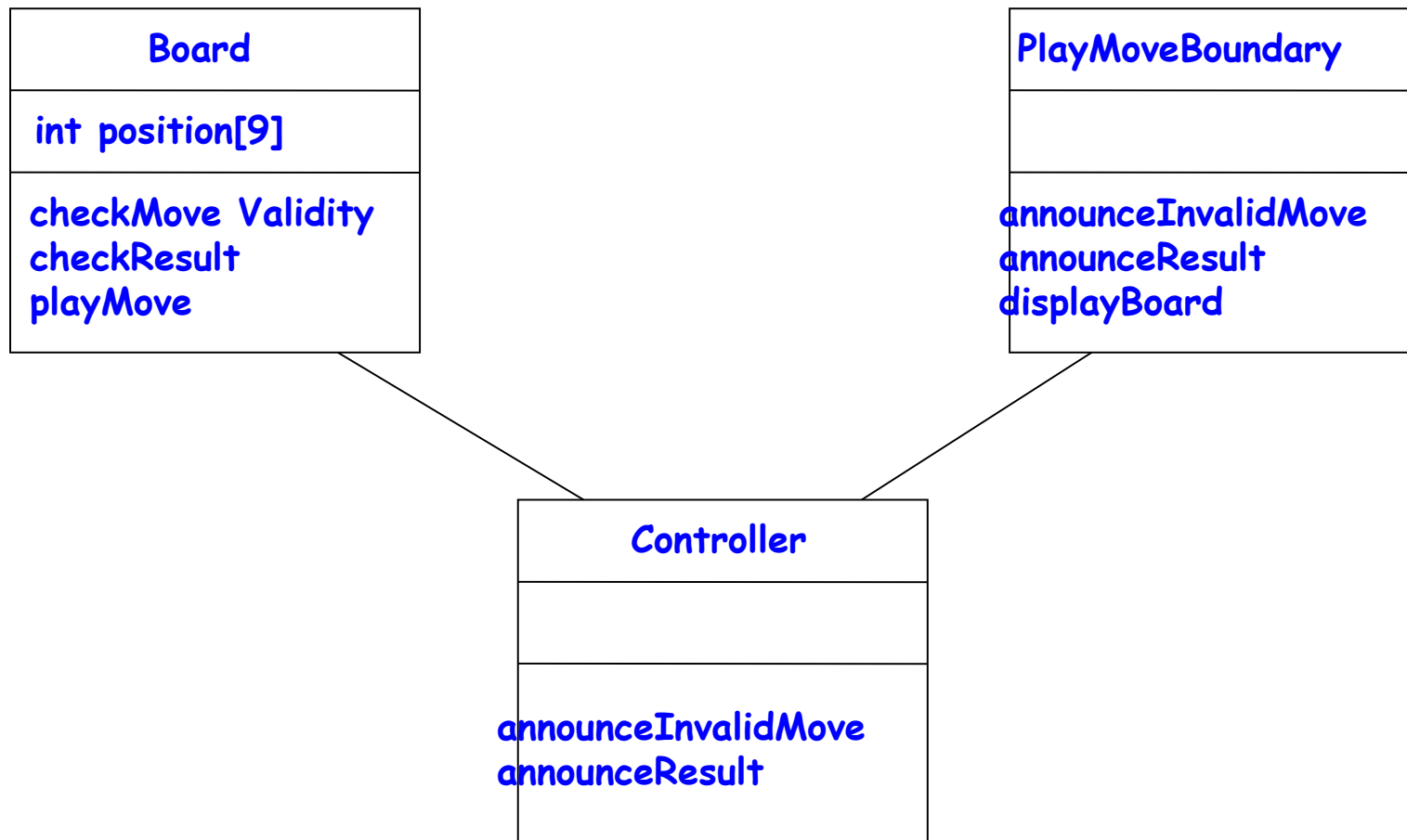
Refined domain model

# Example 1: Sequence Diagram



Sequence Diagram for the play move use case

# Example 1: Class Diagram



# Example 2: Supermarket Prize Scheme

- Supermarket needs to develop software to encourage regular customers.
- Customer needs to supply his:
  - 📁 Residence address, telephone number, and the driving licence number.
- Each customer who registers is:
  - 📁 Assigned a unique customer number (CN) by the computer.

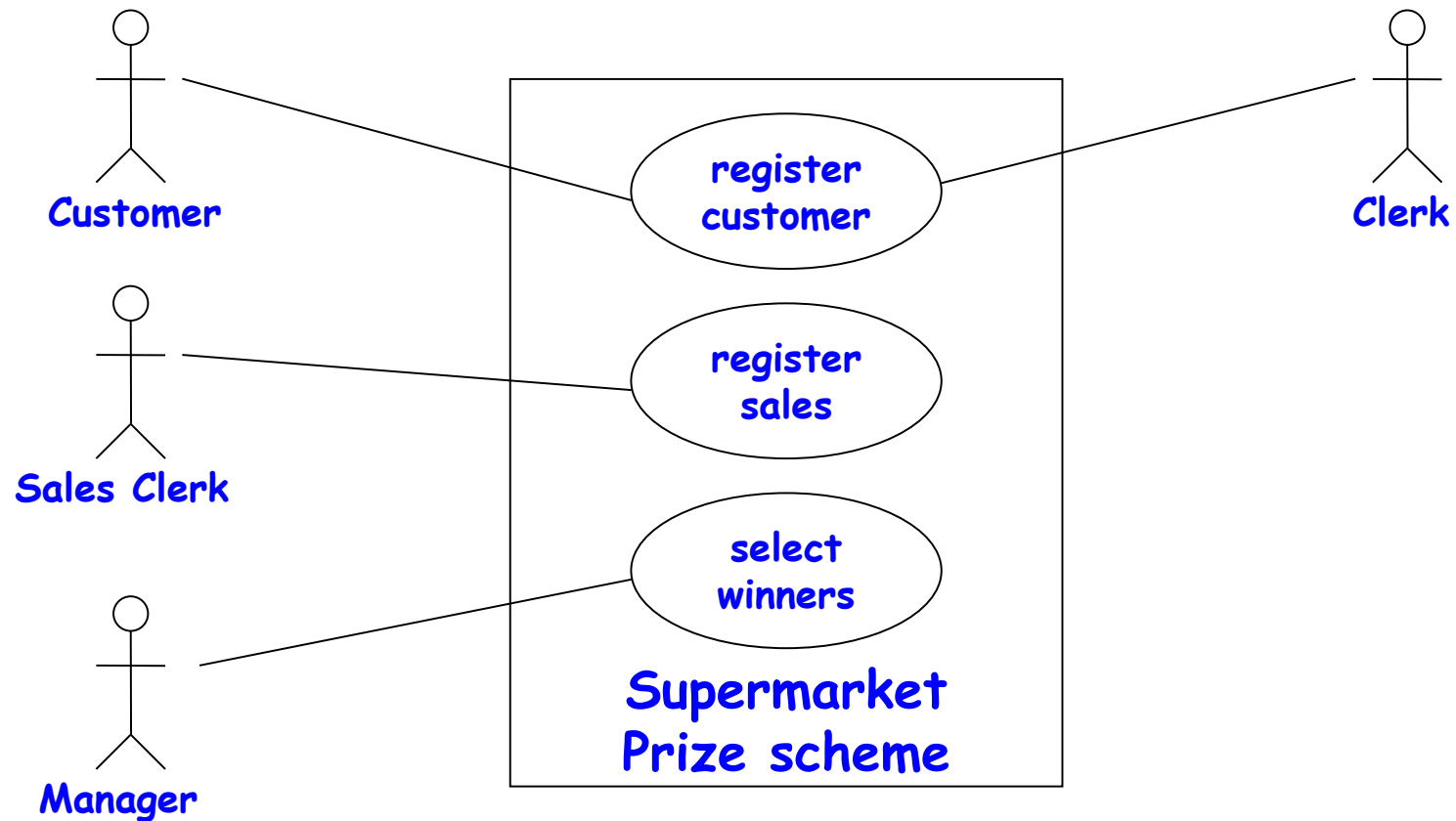
## Example 2: Supermarket Prize Scheme

- A customer can present his CN to the staff when he makes any purchase.
- The value of his purchase is credited against his CN.
- At the end of each year:
  - 📁 The supermarket awards surprise gifts to ten customers who make highest purchase.

## Example 2: Supermarket Prize Scheme

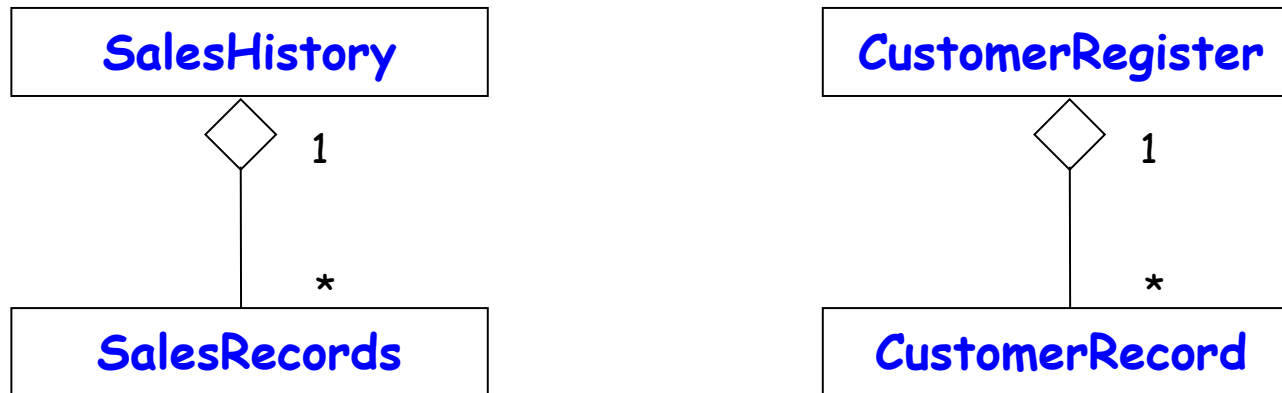
- Also, it awards a 22 carat gold coin to every customer:
  - 📁 Whose purchases exceed Rs. 10,000.
- The entries against the CN are reset:
  - 📁 On the last day of every year after the prize winner's lists are generated.

# Example 2: Use Case Model



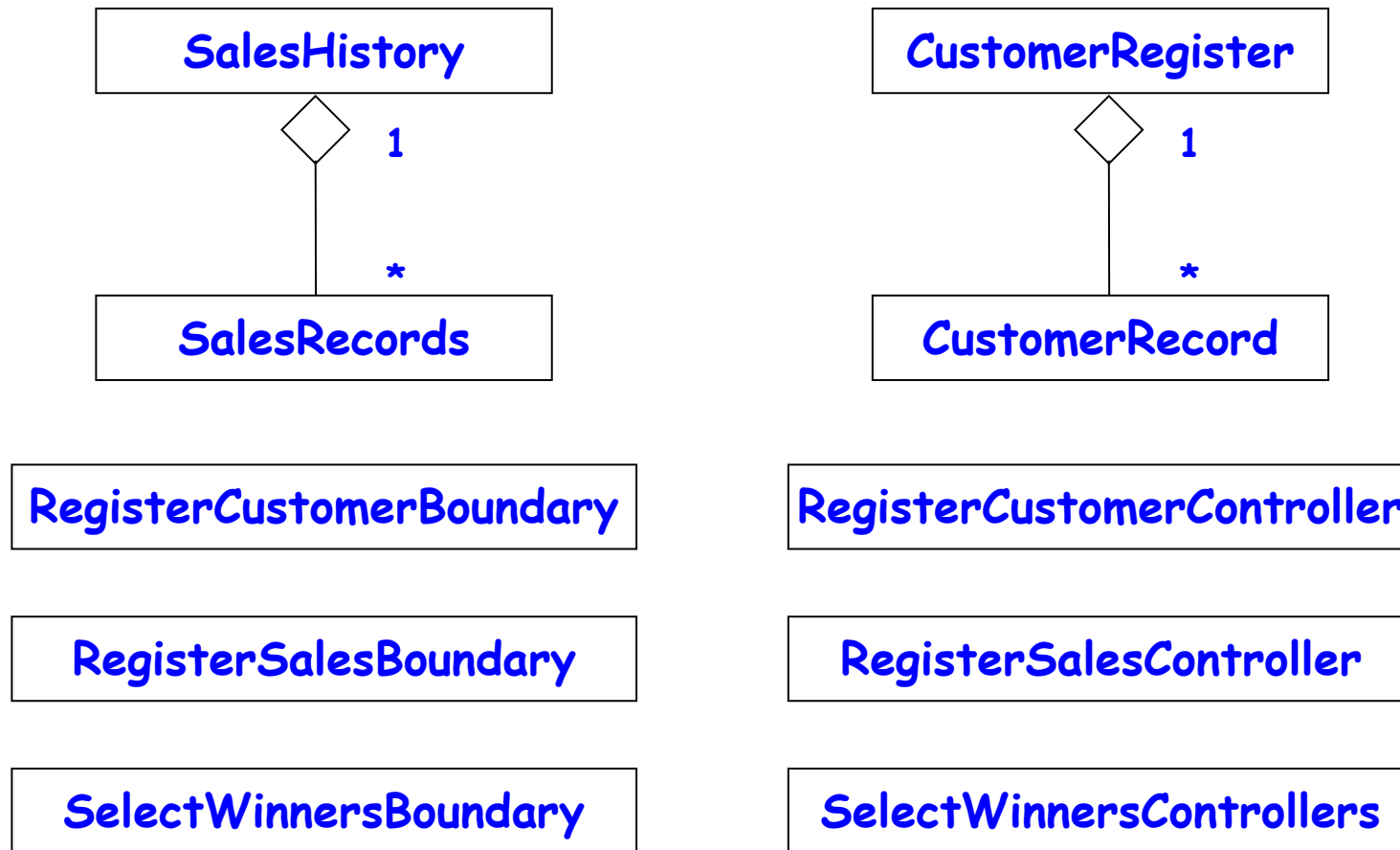


# Example 2: Initial Domain Model



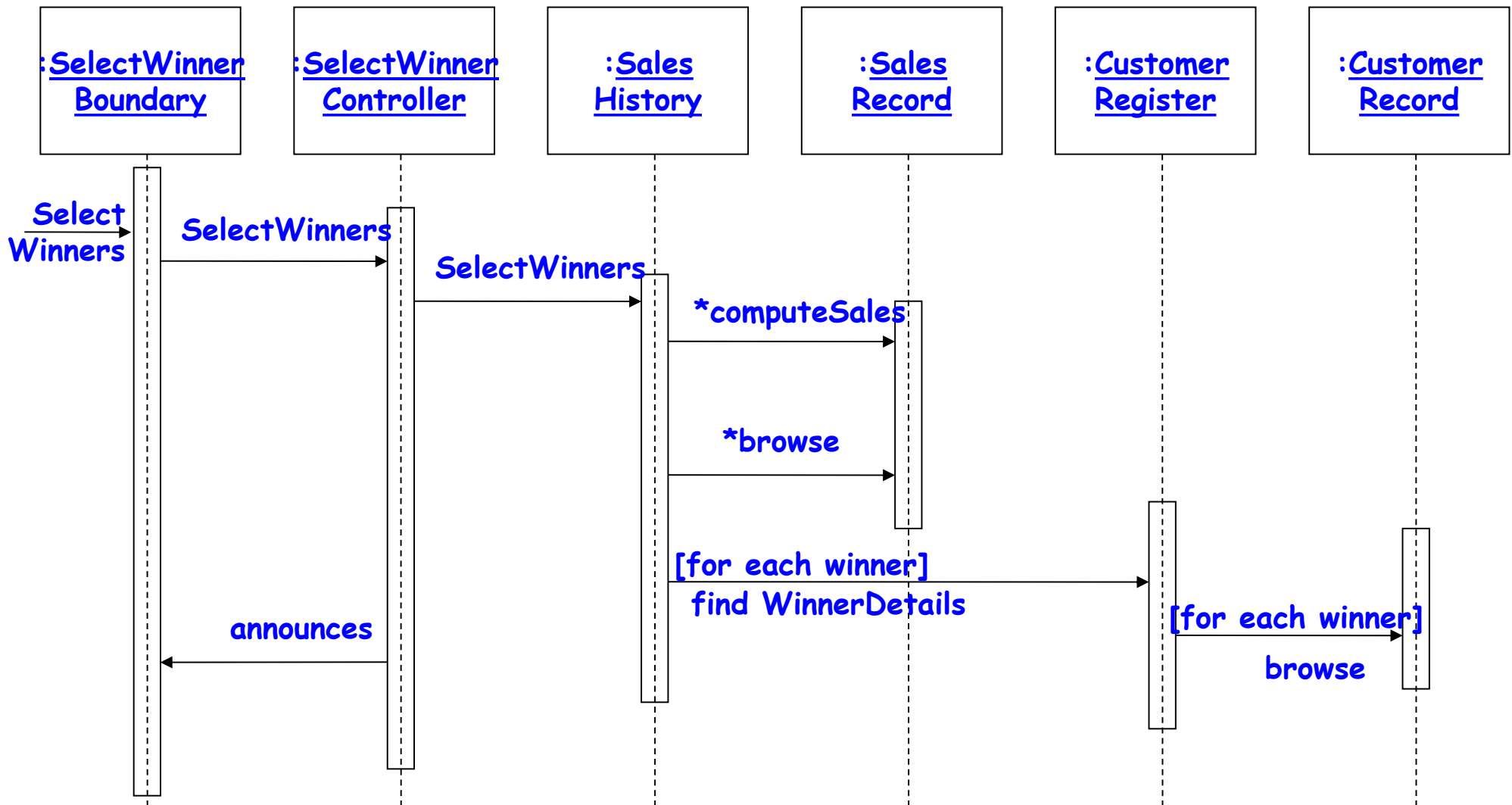
Initial domain model

# Example 2: Refined Domain Model



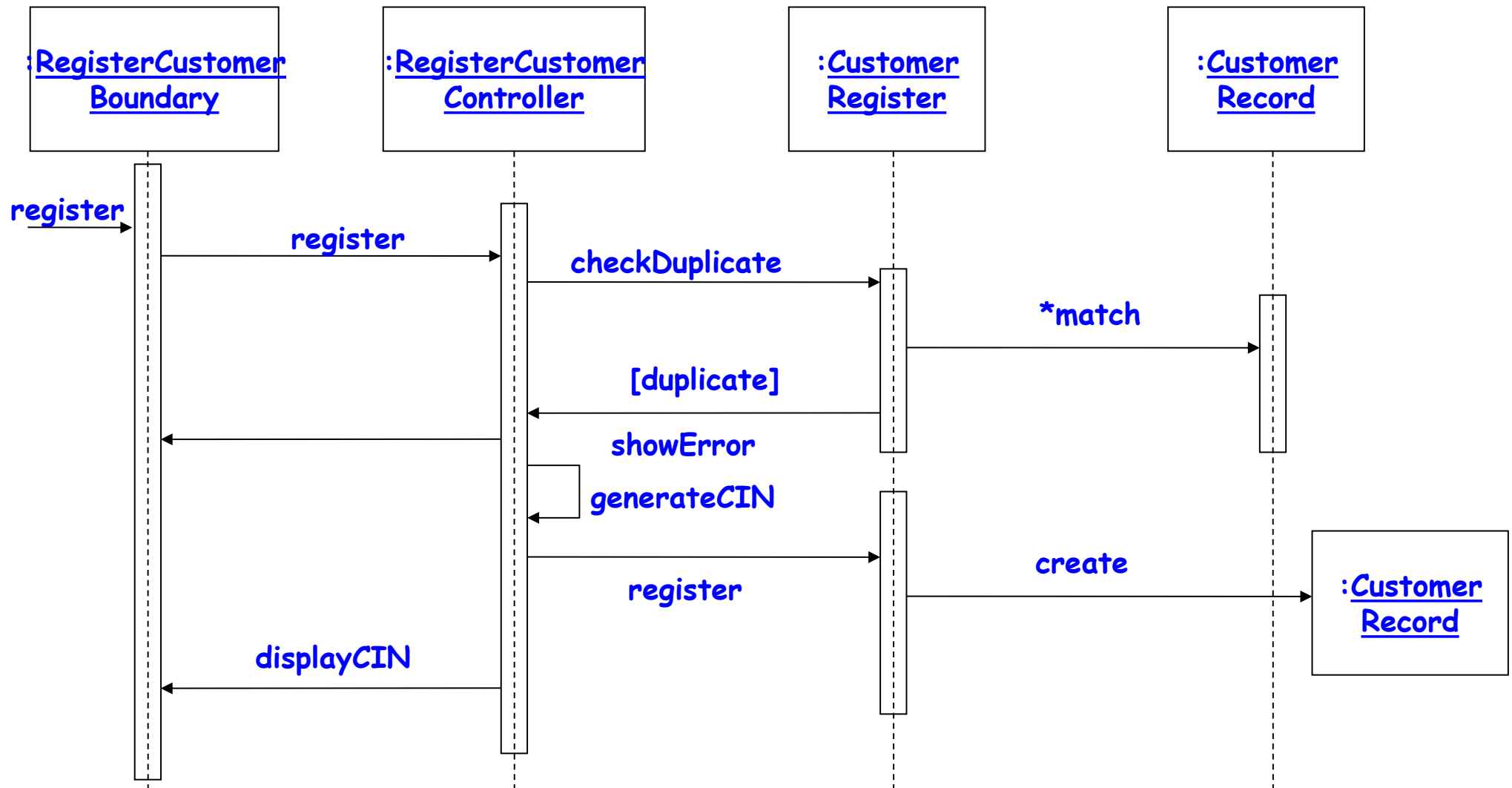
Refined domain model

# Example 2: Sequence Diagram for the Select Winners Use Case



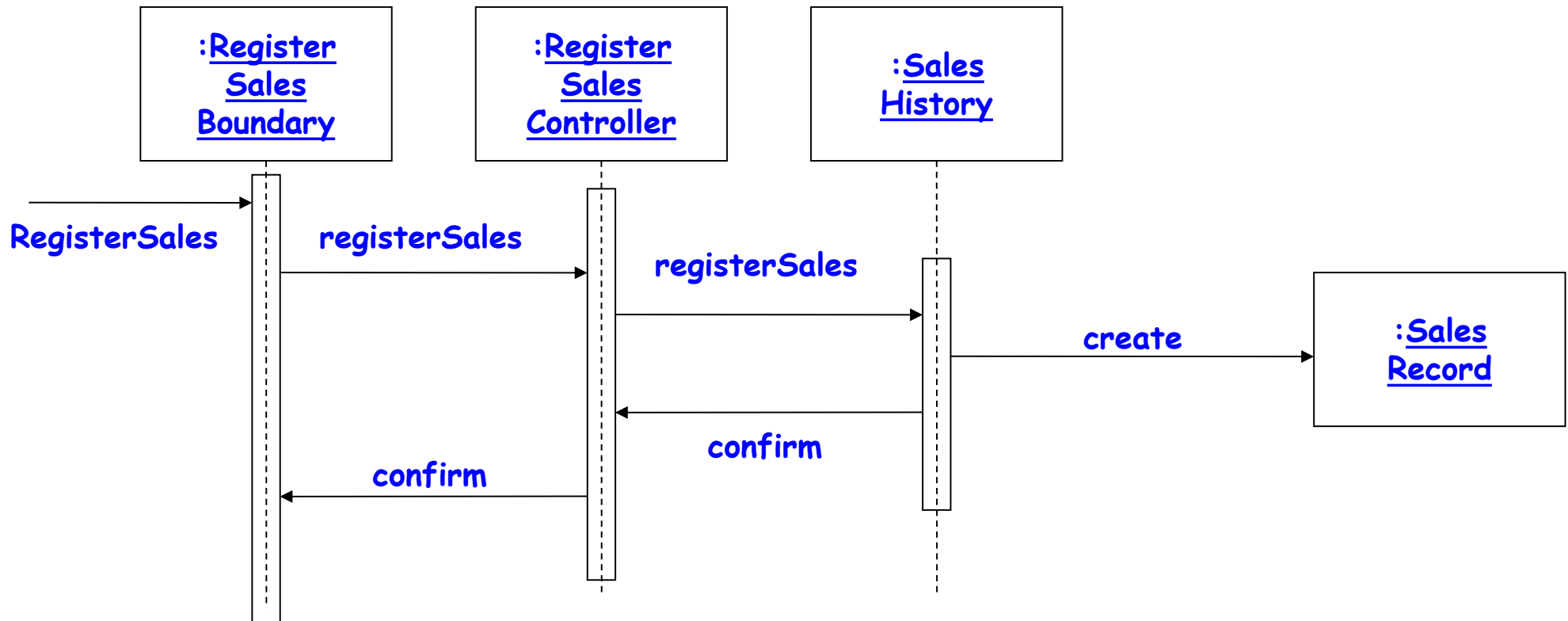
Sequence Diagram for the select winners use case

# Example 2: Sequence Diagram for the Register Customer Use Case



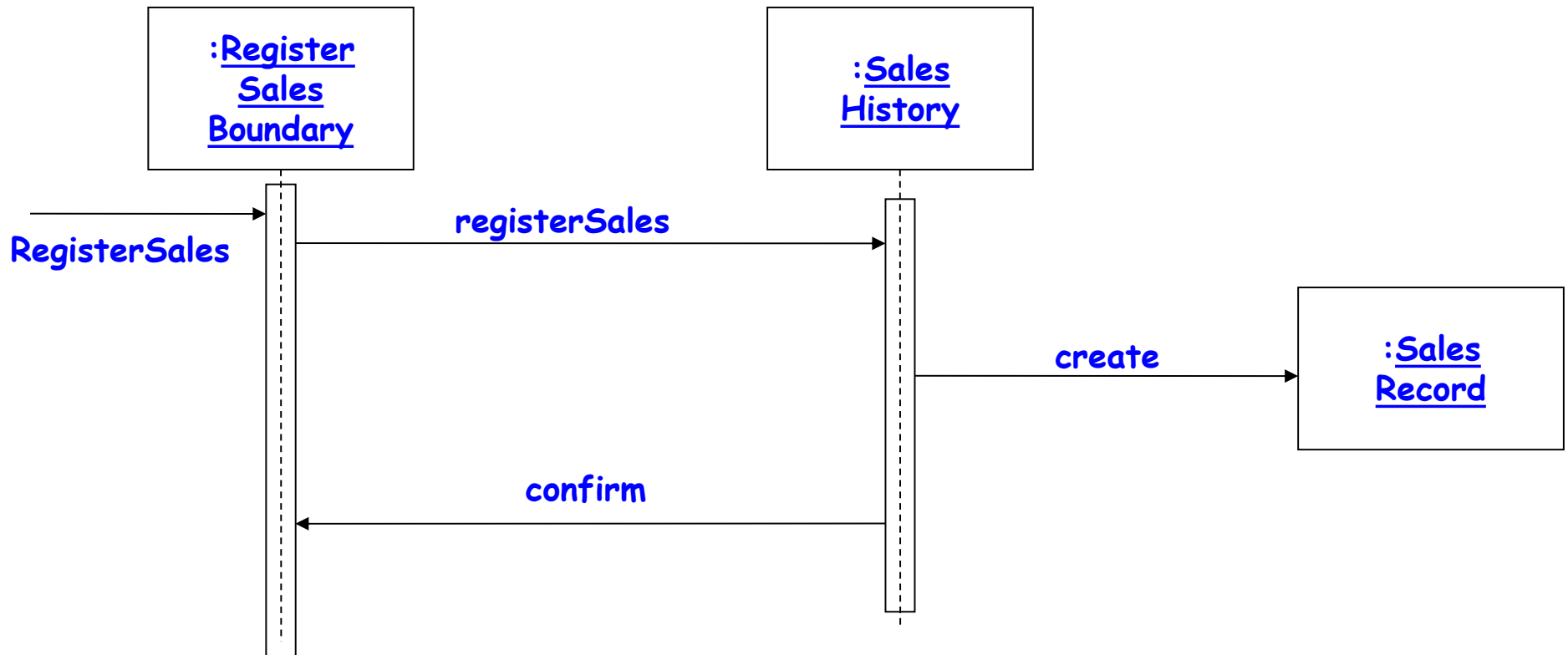
Sequence Diagram for the register customer use case

# Example 2: Sequence Diagram for the Register Sales Use Case



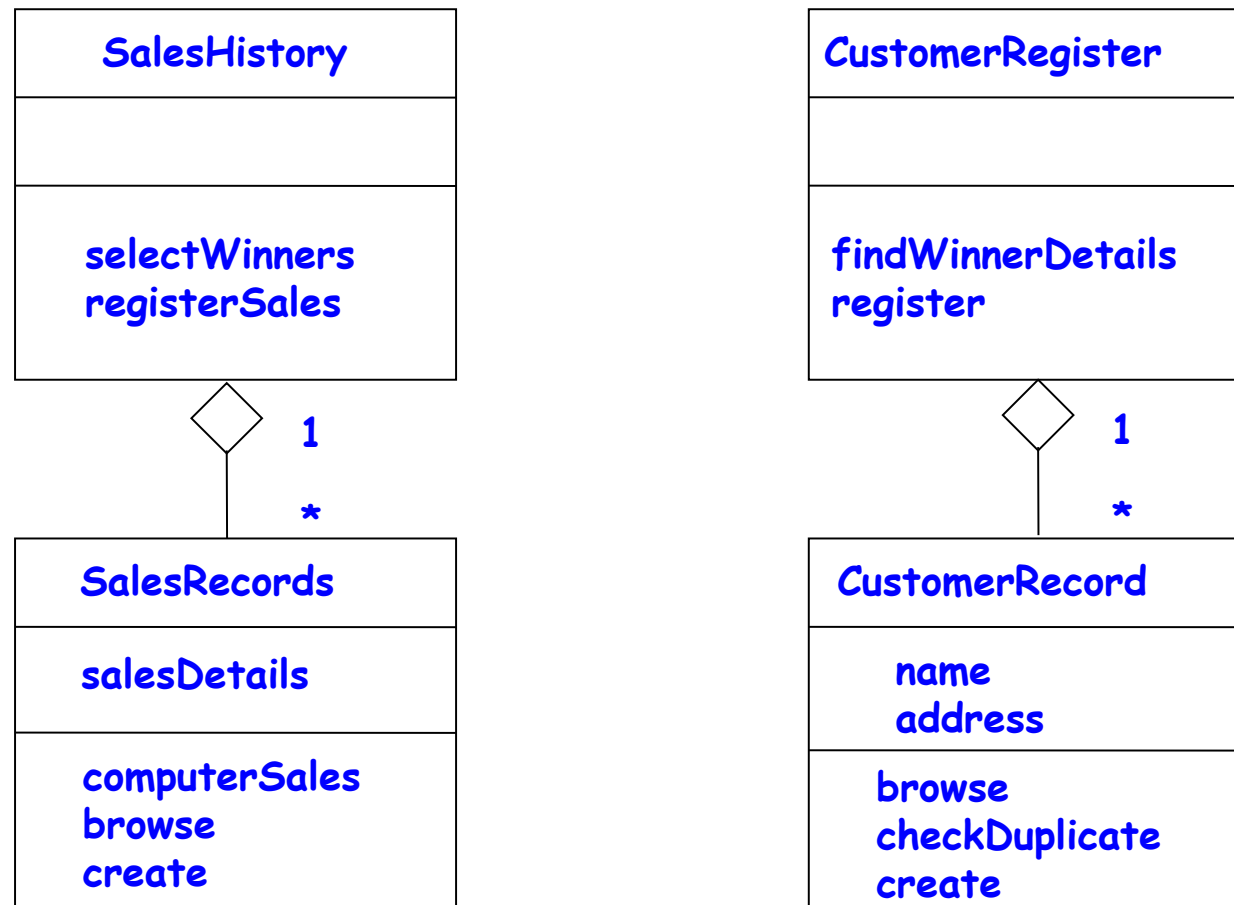
Sequence Diagram for the register sales use case

# Example 2: Sequence Diagram for the Register Sales Use Case



Refined Sequence Diagram for the register sales use case

# Example 2: Class Diagram



# 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

cont...

- Other UML diagrams:
  - 📄 Interaction diagrams (sequence and collaboration),
  - 📄 Activity diagrams,
  - 📄 State chart diagrams.
- We discussed OO software development process:
  - 📄 Use of patterns lead to increased productivity and good solutions.