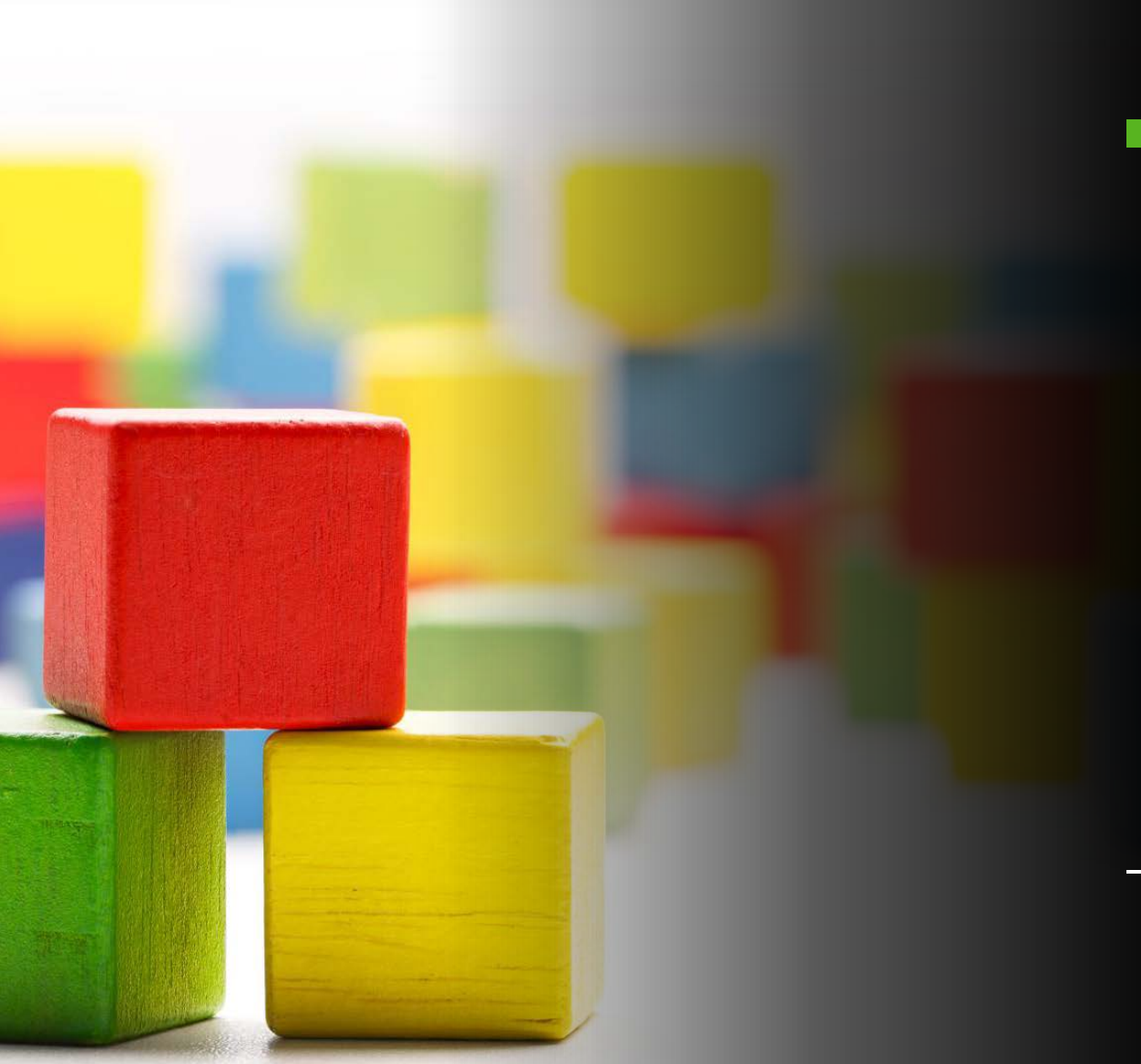


Software Engineering

Dr. Arijit Karati

Contents of the slides are prepared based on the materials from web and textbooks. It is stated that this material will be used to make the students aware of the topics and practiced for non-profit purposes.

(slide can be found in this secure domain)

Object-Oriented Software Design

Contents...

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

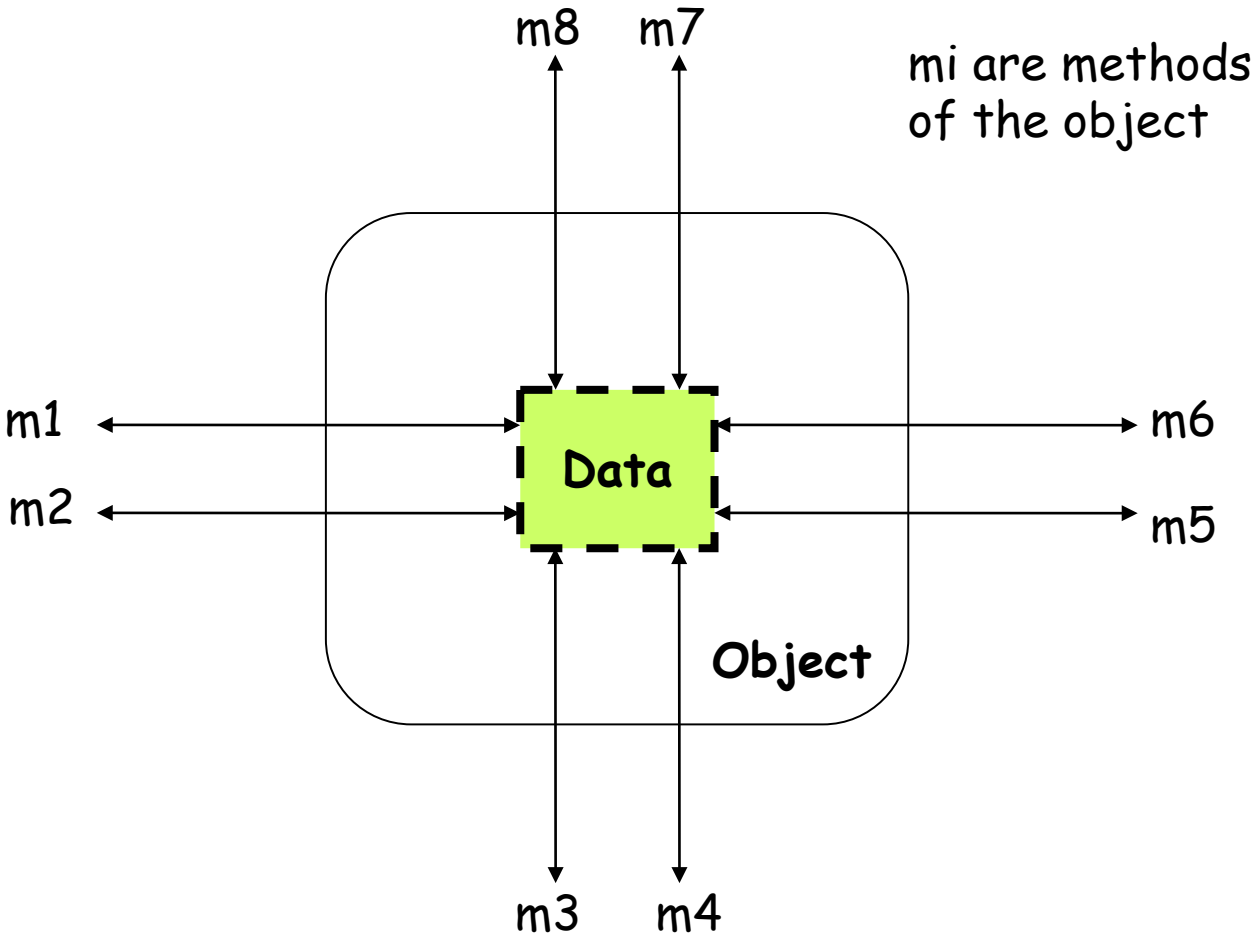
Object-Oriented Concepts

- 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).

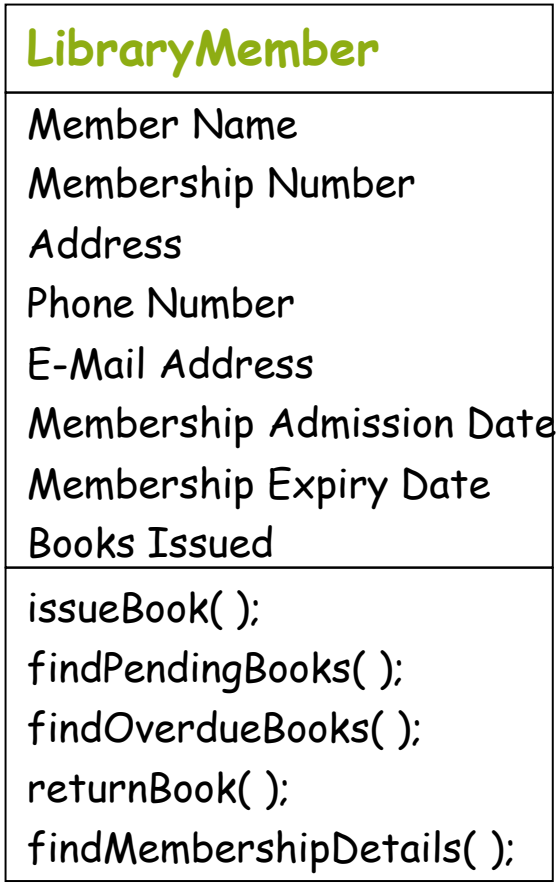
Model of an Object

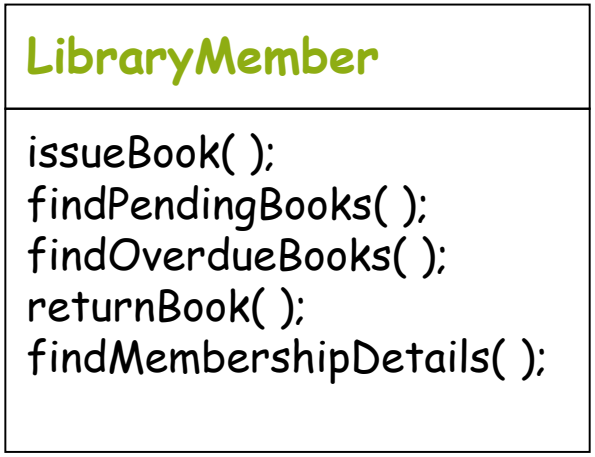


Class

- 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?

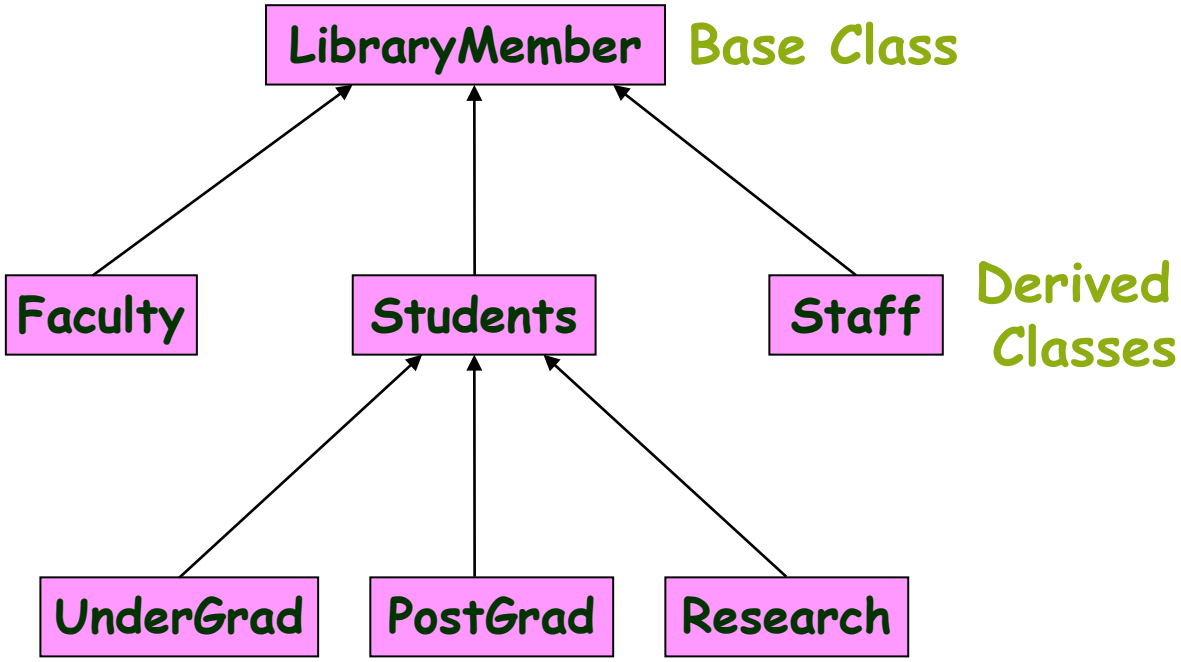
- Four types of relationships:
  - ✓ Inheritance
  - ✓ Association
  - ✓ Aggregation/Composition
  - ✓ Dependency

Inheritance

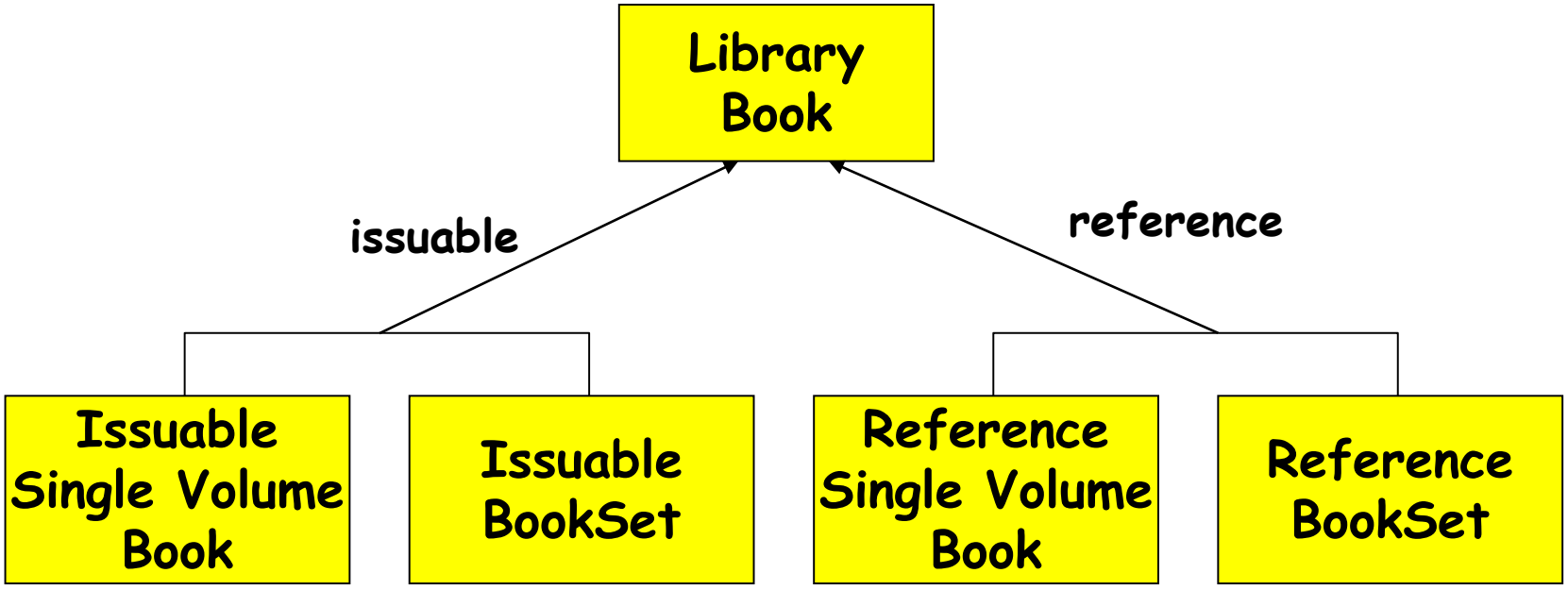
- Allows to define a new class (derived class) by extending or modifying existing class (base class).
  - ✓ Represents generalization-specialization relationship.
  - ✓ Allows redefinition of the existing methods (method overriding).

Inheritance

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

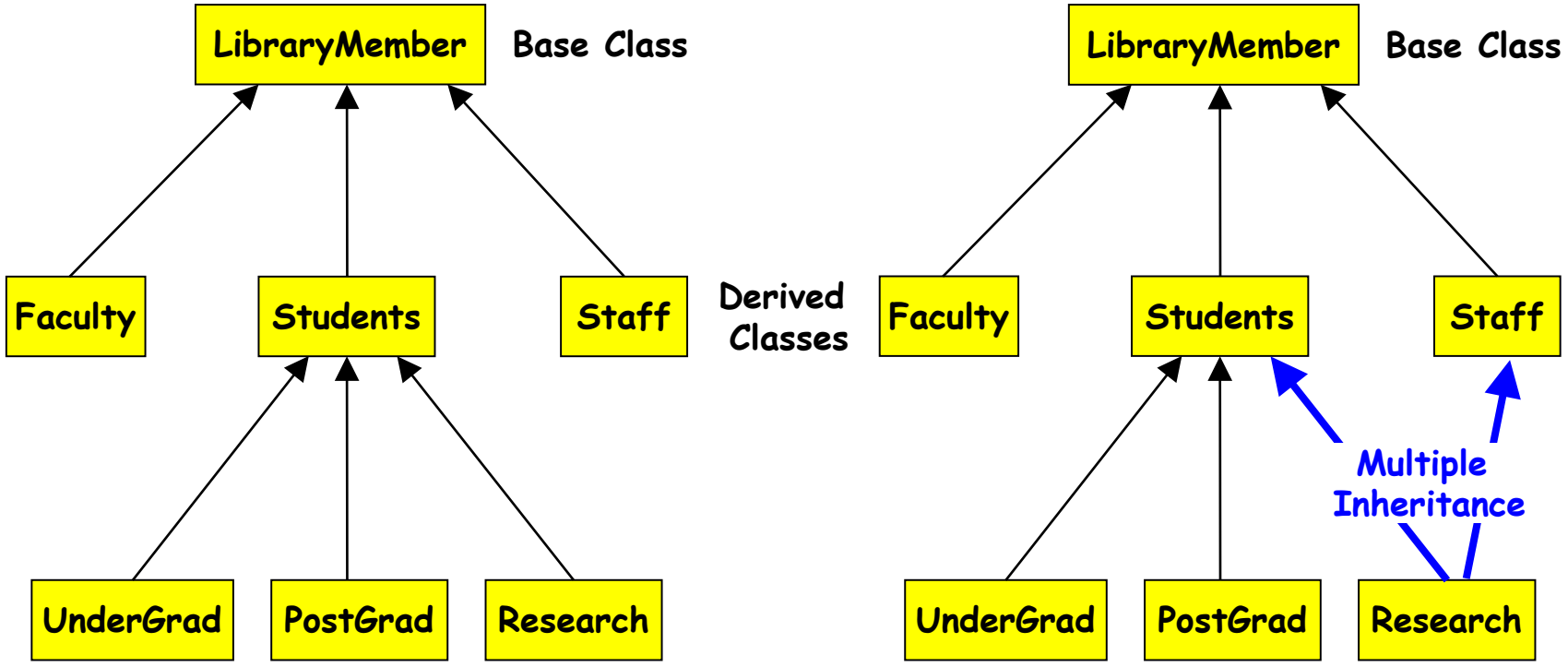


Inheritance Example



Representation of the inheritance relationship

Multiple Inheritance



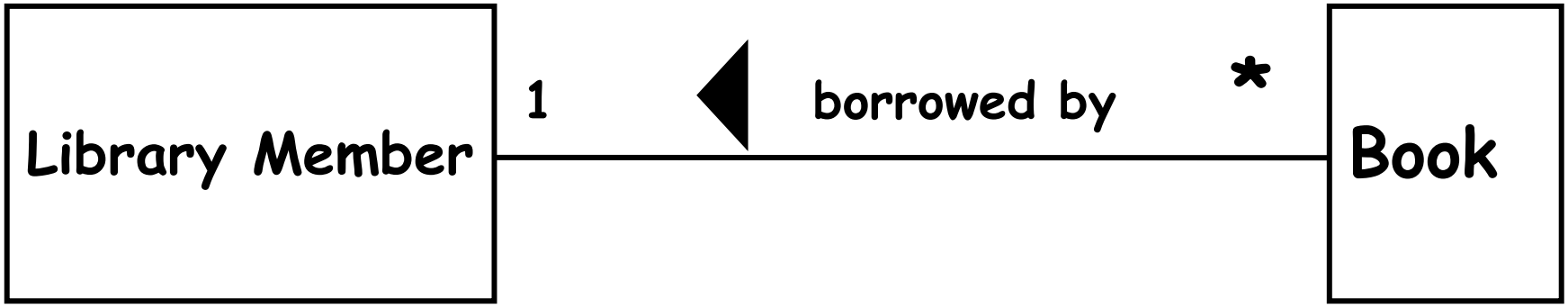
Association Relationship

- 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.

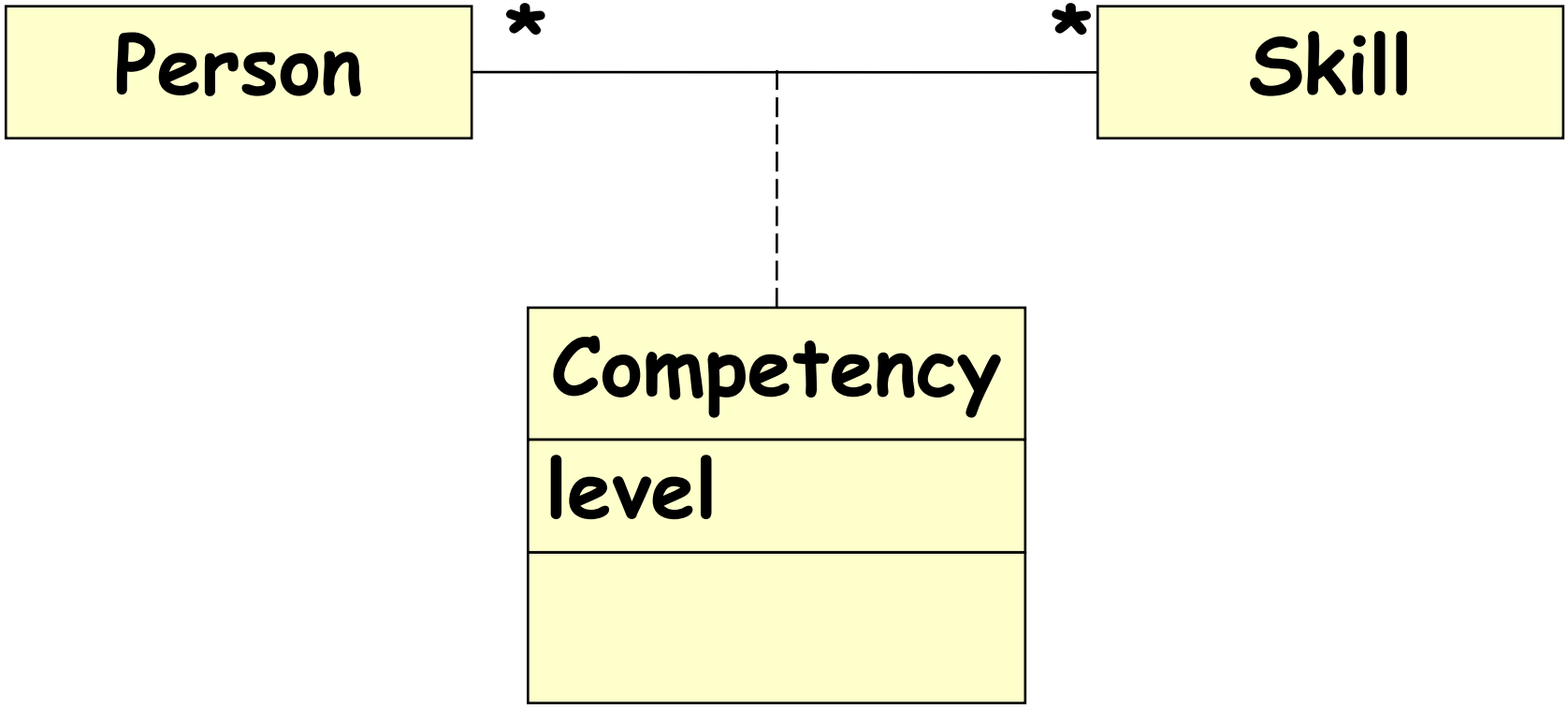
Association Relationship

- 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



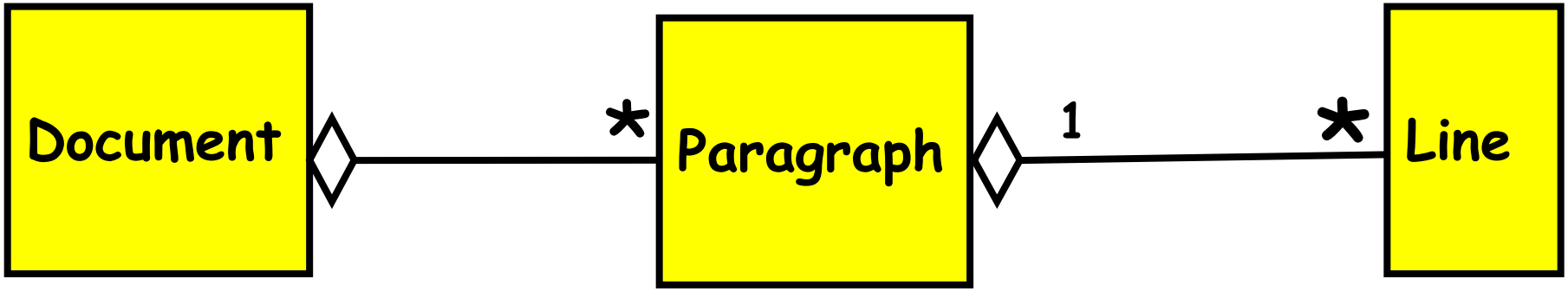
Association and Link

- 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

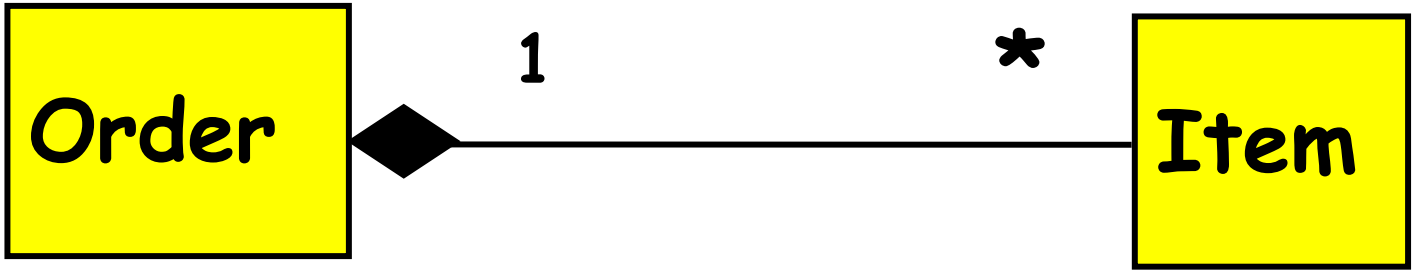
- Represents whole-part relationship
- Represented by a diamond symbol at the composite end
- Cannot be reflexive(i.e. recursive)
- Not symmetric
- It can be transitive

Aggregation Relationship



Composition Relationship

- Life of item is same as the order



Aggregation

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

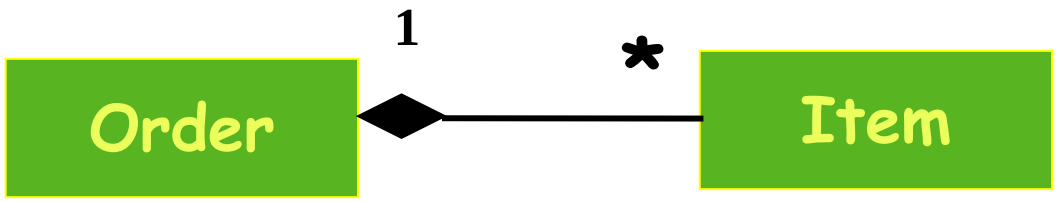
Aggregation vs. Inheritance

- Inheritance:
  - ✓ Different object types with similar features.
  - ✓ Necessary semantics for similarity of behaviour is in place.
- Aggregation:
  - ✓ Containment allows construction of complex objects.

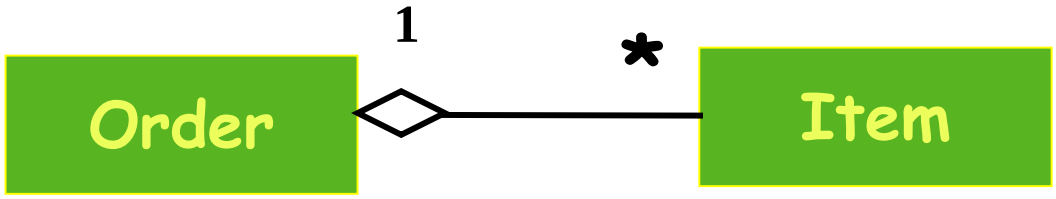
Aggregation vs. Composition

- 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.

Aggregation vs. Composition

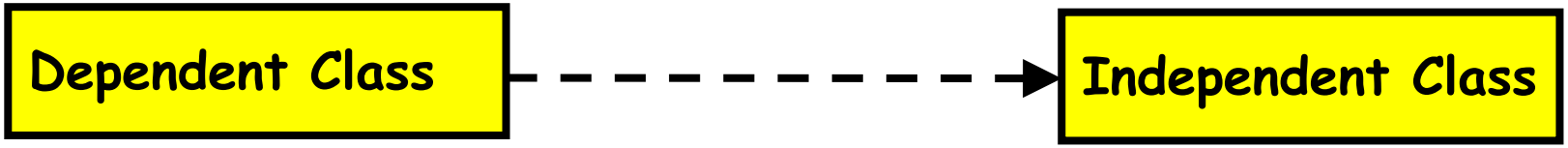


Composition



Aggregation

Class Dependency



Representation of dependence between classes

Abstraction

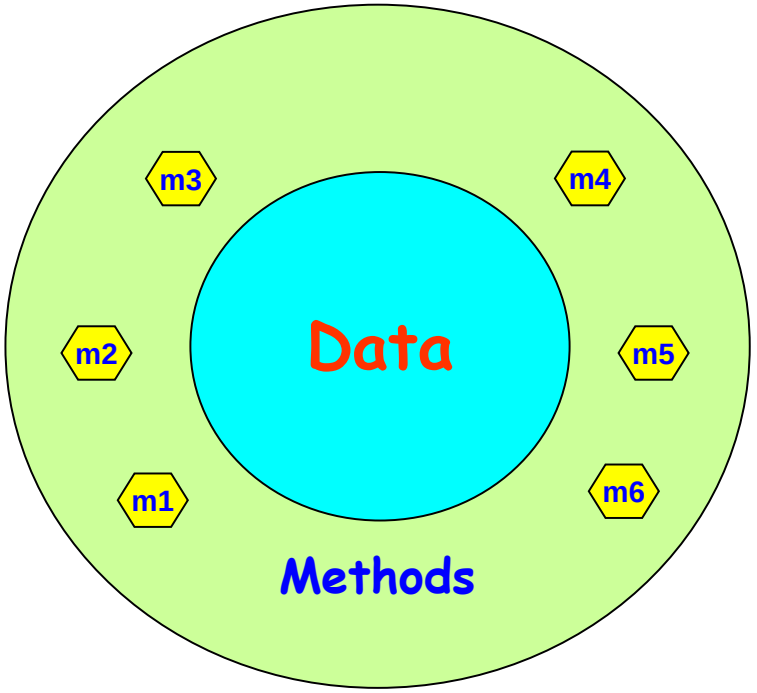
- Consider aspects relevant for certain purpose
  - ✓ Suppress non-relevant aspects
- Types of abstraction:
  - ✓ Data abstraction
  - ✓ Behaviour abstraction

Abstraction

- 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.



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

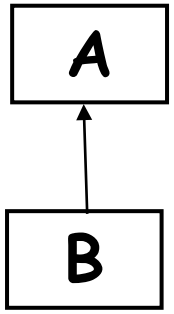
- A class named Circle has three definitions for create operation
  - ✓ Without any parameter, default
  - ✓ Centre and radius as parameter
  - ✓ Centre, radius and fillType as parameter
  - ✓ Depending upon parameters, method will be invoked
  - ✓ Method create is said to be overloaded

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)

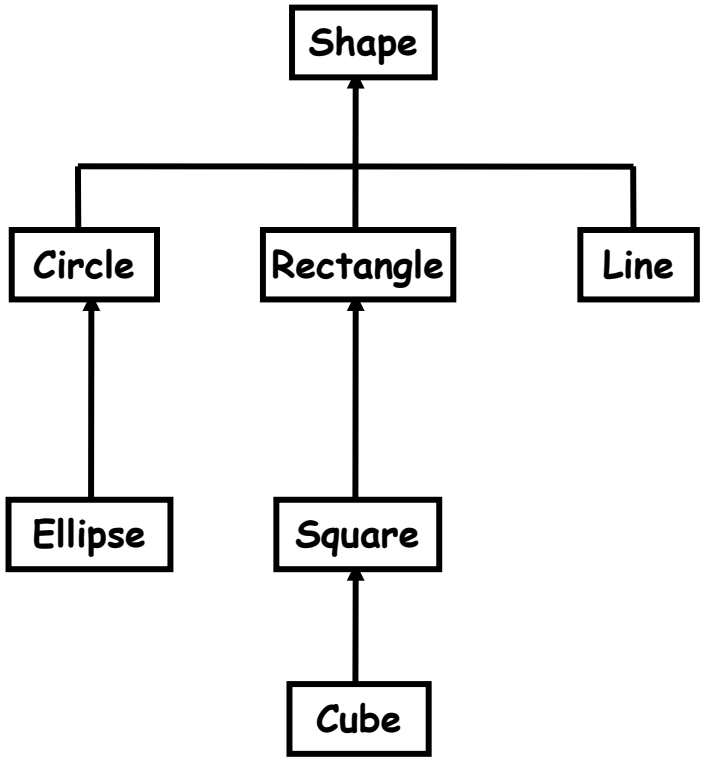
b=a; (not OK)

Dynamic Binding

- 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.



Class hierarchy of geometric objects

An Example

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

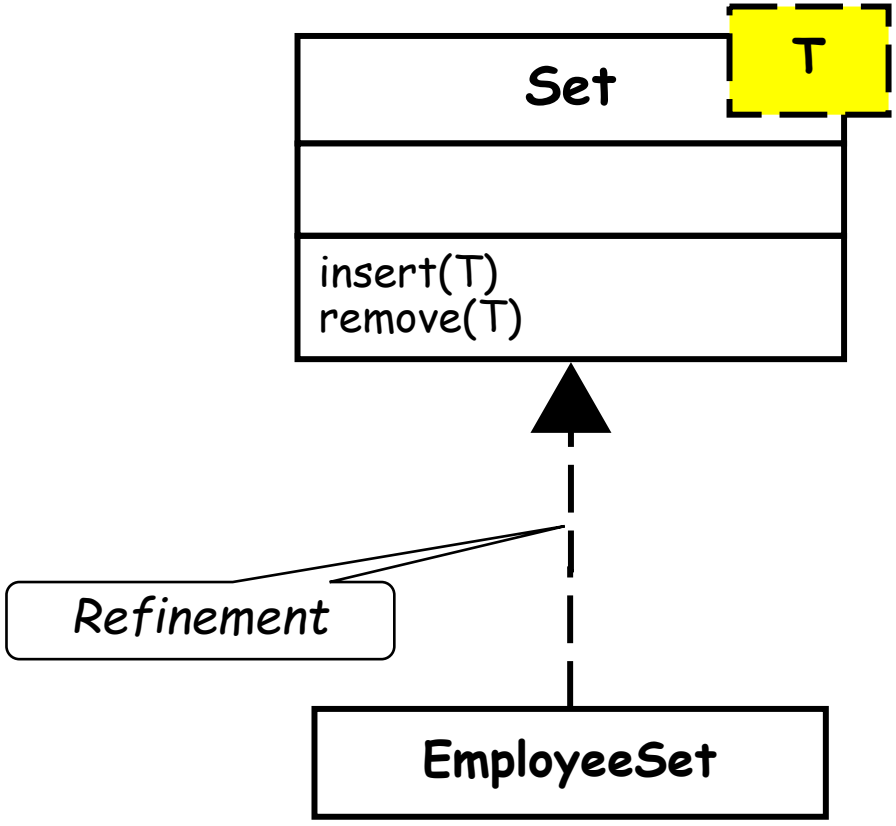
```
Shape s[1000];
For(i=0;i<1000;i++)
         Shape.draw();

            -
-
-
-
```

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.

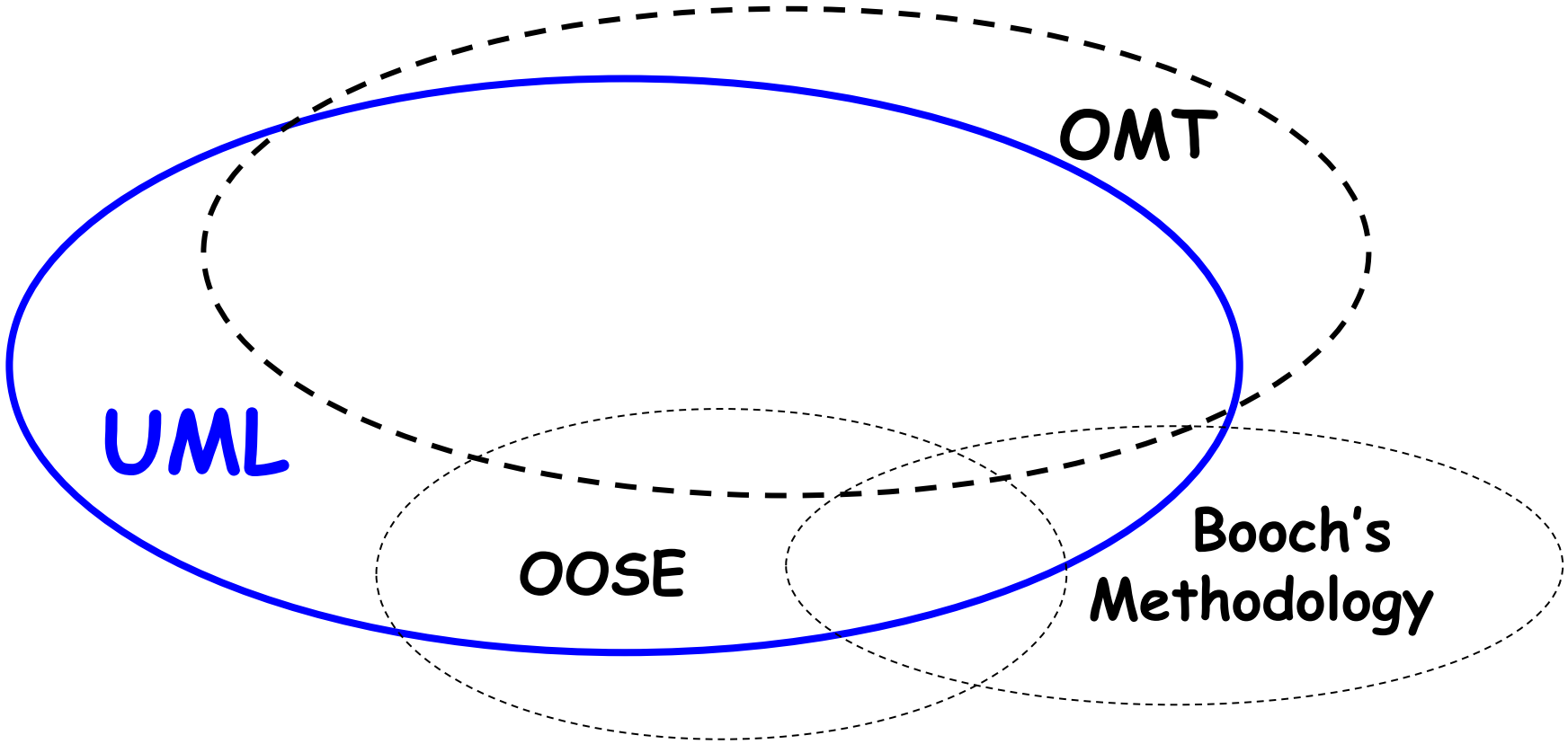
UML Origin

- 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 Modelling Techniques in UML

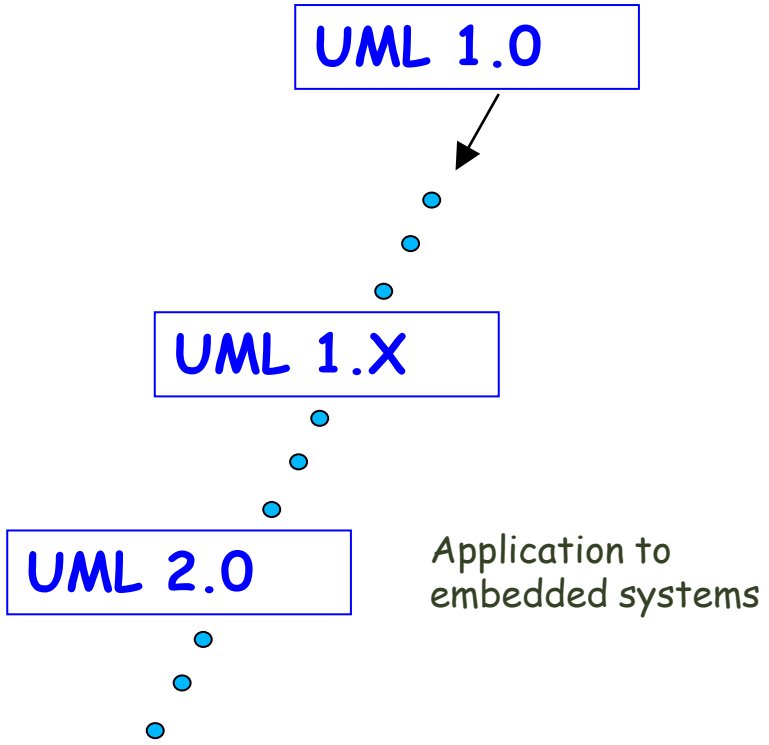


UML as A Standard

- 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

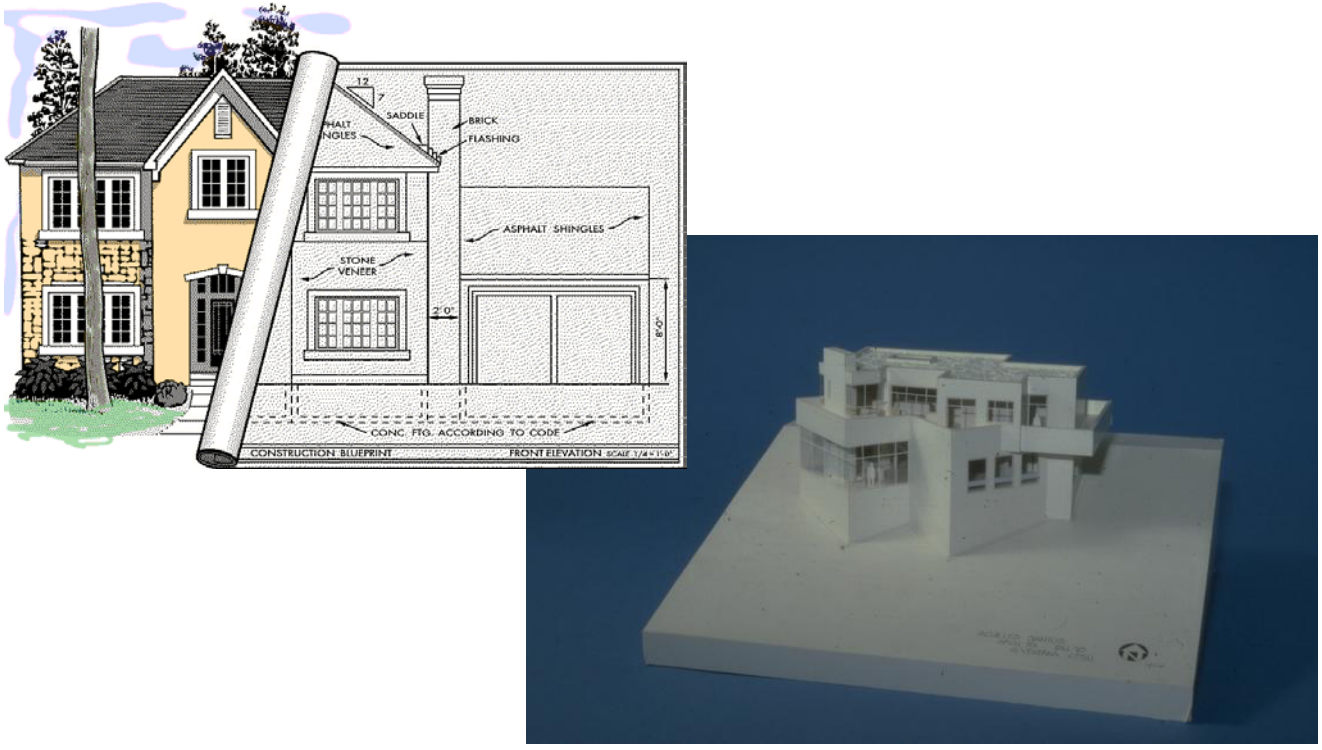
- UML continues to develop:
  - ✓ Refinements
  - ✓ Making it applicable to new contexts



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

Modelling a House



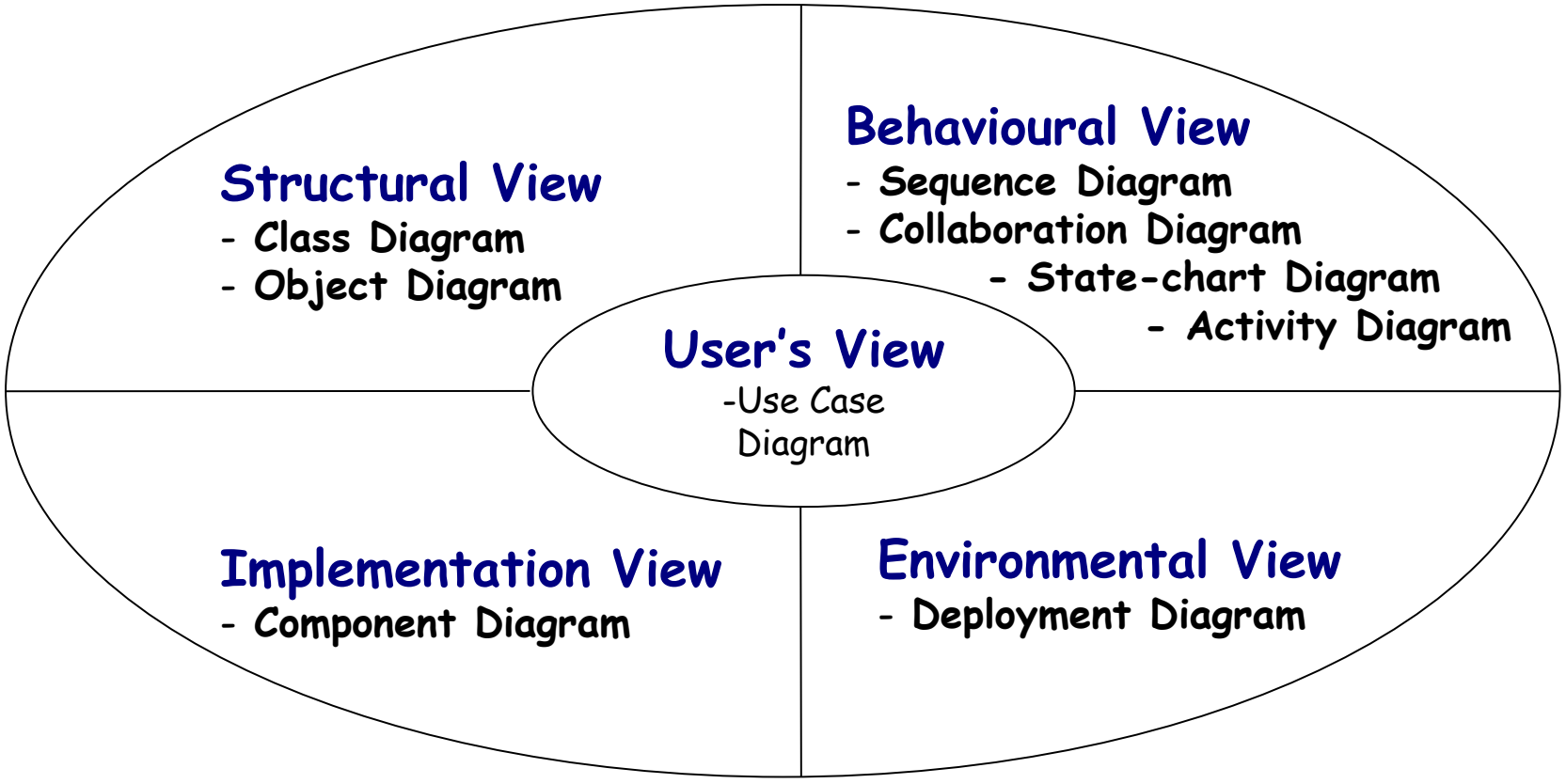
UML Diagrams

- 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
  - ✓ Behavioural 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 behaviour 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.

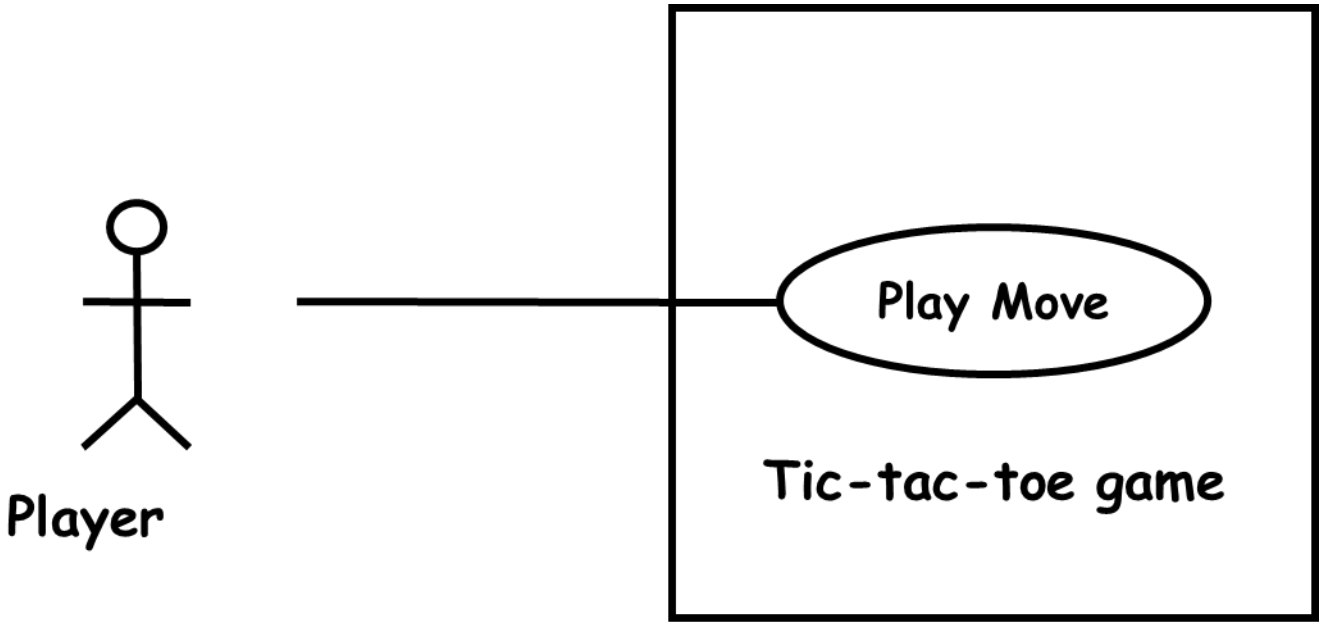
Example Use Cases

- 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

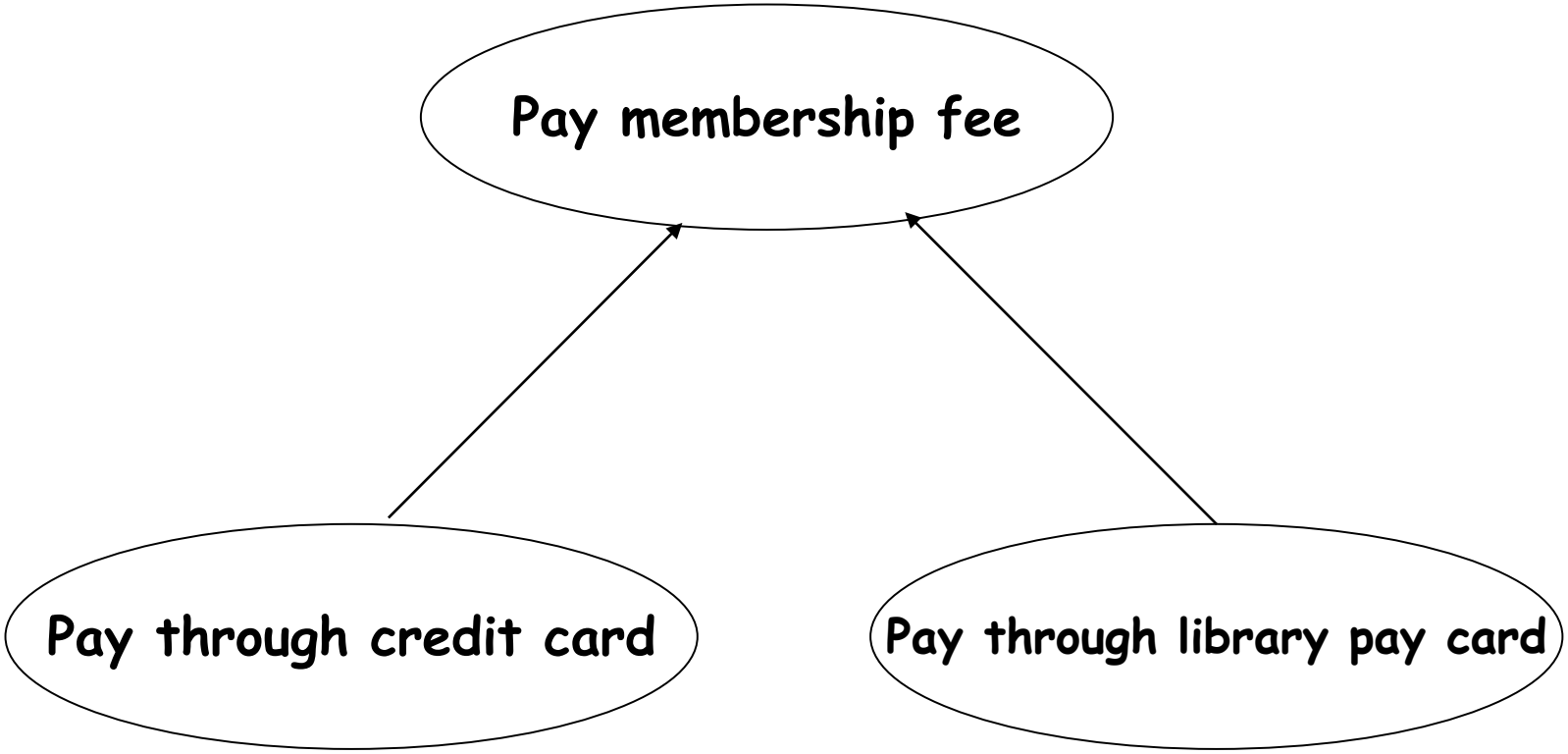
Why Develop A Use Case Diagram?

- 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).

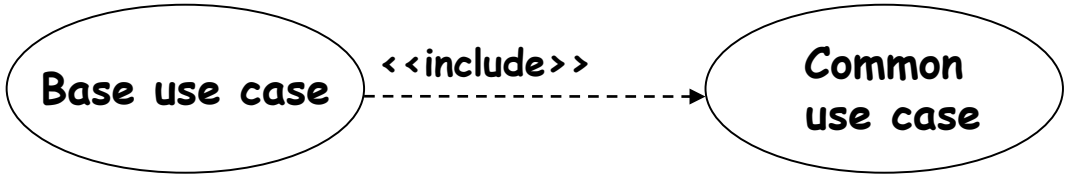
Factoring Use Cases

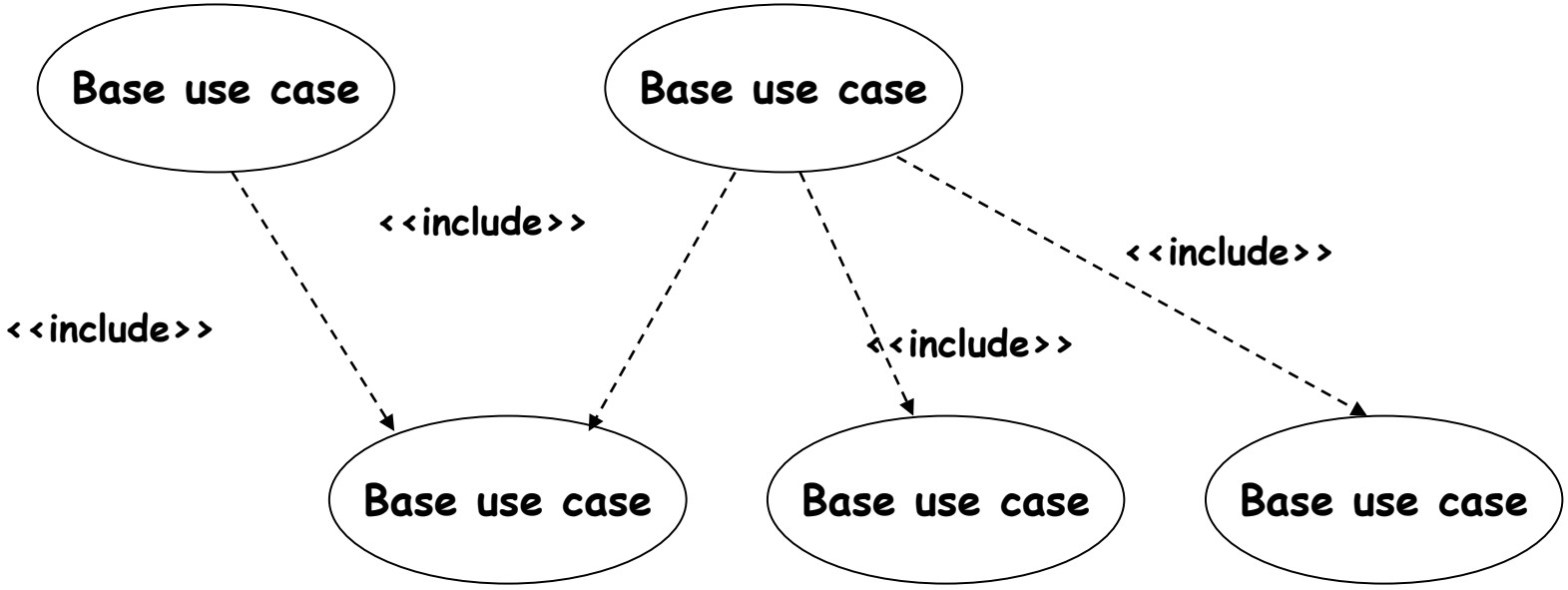
- Two main reasons for factoring:
  - ✓ Complex use cases need to be factored into simpler use cases
  - ✓ To represent common behaviour across different use cases
- Three ways of factoring:
  - ✓ Generalization
  - ✓ Includes
  - ✓ Extends

Factoring Use Cases Using Generalization

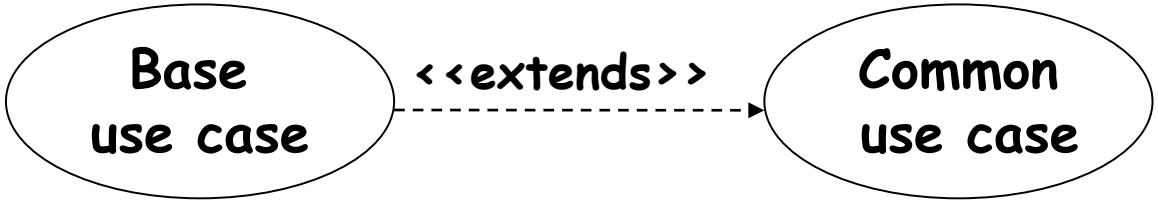


Factoring Use Cases Using Includes

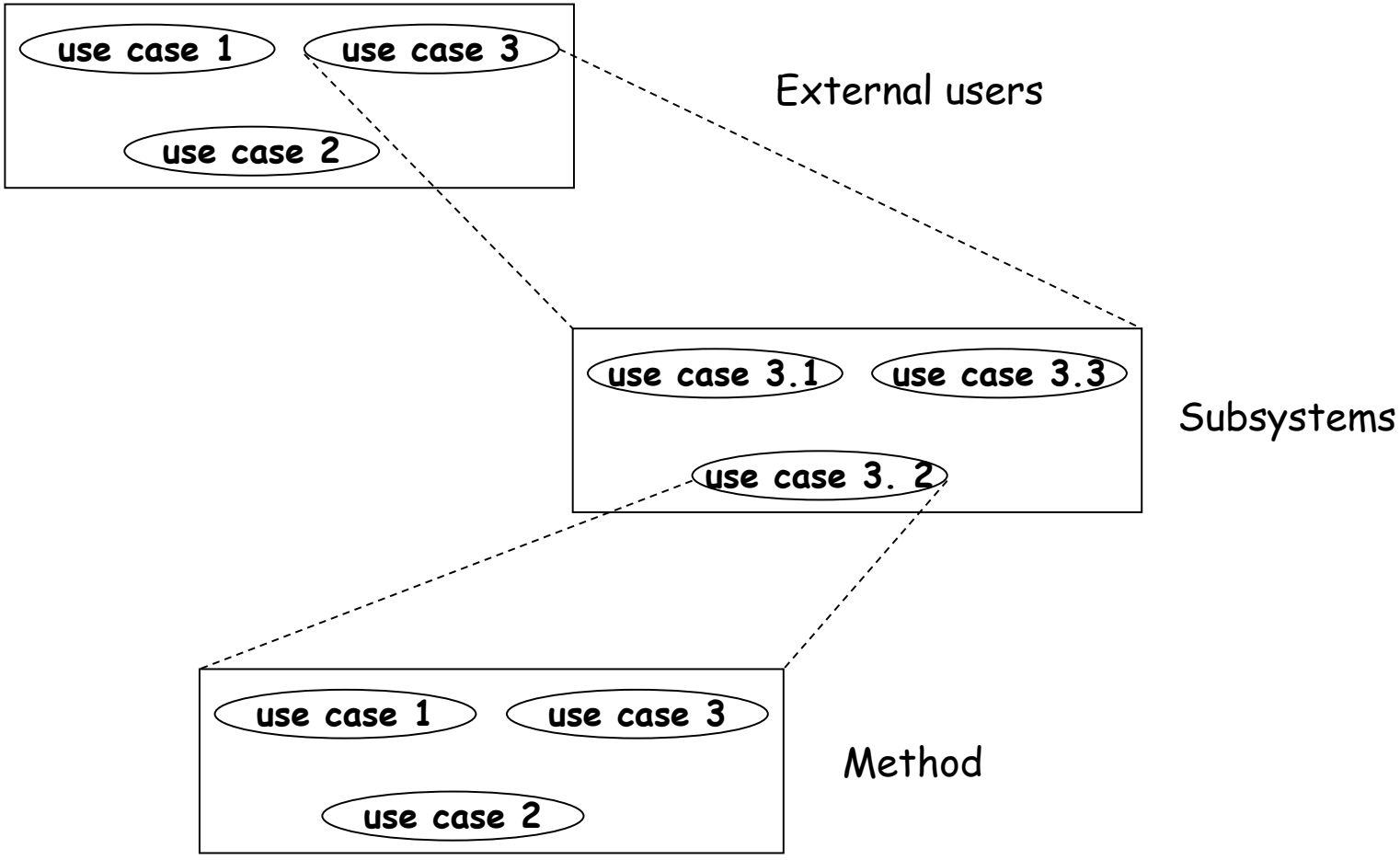




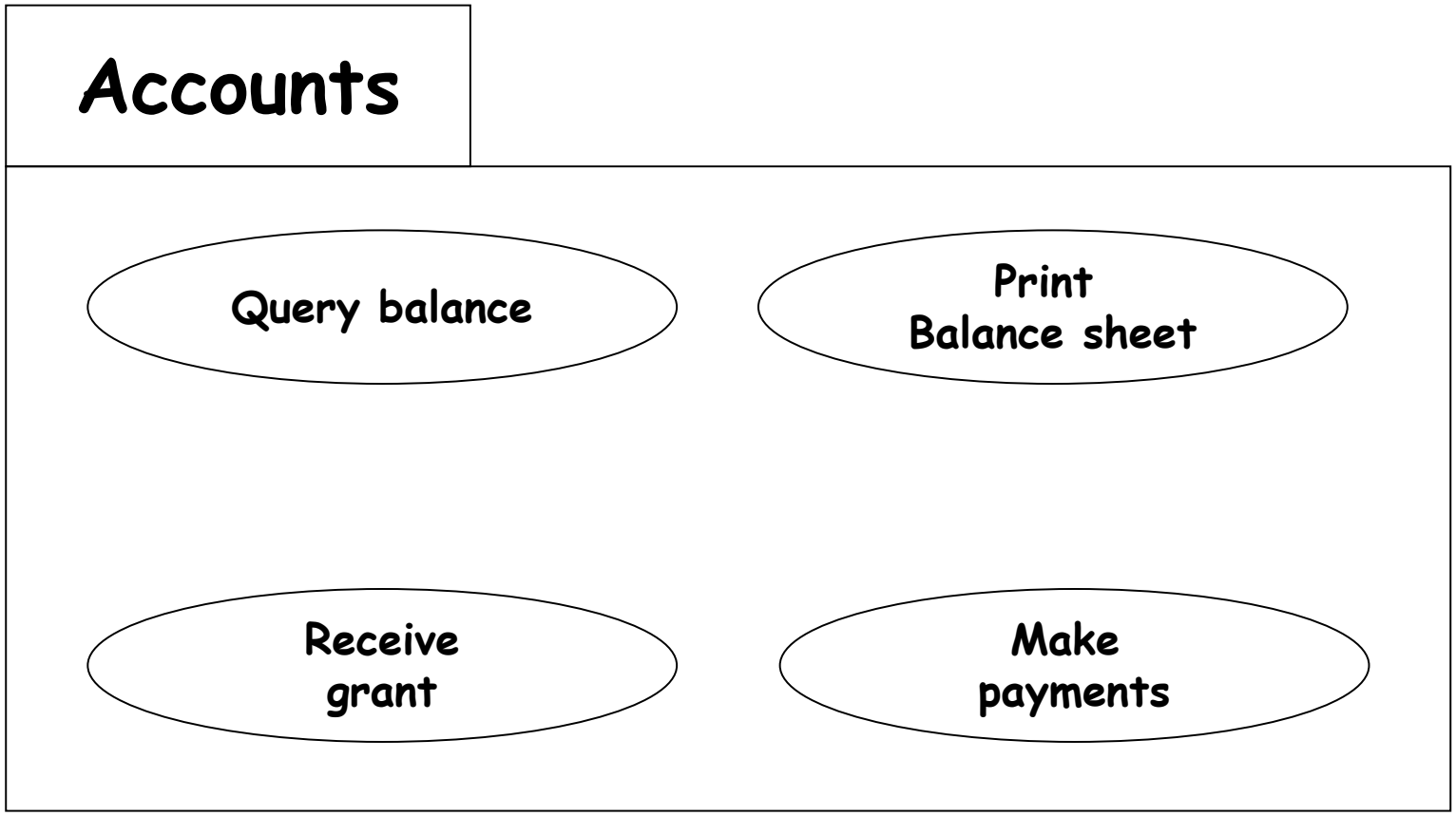
Factoring Use Cases Using Extends



Hierarchical Organization of Use Cases



Use Case Packaging



Summary (CONT.)

- We discussed structured analysis of two small examples:
  - ✓ RMS calculating software
  - ✓ Tic-tac-toe computer game software
- Several CASE tools are available:
  - ✓ Support structured analysis and design.
  - ✓ Maintain the data dictionary,
  - ✓ Check whether DFDs are balanced or not.