

CSSE2003

Software Engineering Studio

Semester 2, 2009

6: Software Design Overview & UML Class Diagrams

Lecture Summary

- ◆ Software development phases
- ◆ Object-oriented design method
- ◆ UML
- ◆ Simple example (from the Larman's text book)
- ◆ UML class diagrams:
 - class notation
 - relationship notation

Phases of Software Development

Requirements Analysis *(answers "WHAT?")*

- Specifying what the application must do

Design *(answers "HOW?")*

- Specifying what the parts will be, and how they will fit together

Implementation *(A.K.A. "CODING")*

- Writing the code

Testing *(type of VERIFICATION)*

- Executing the application with test data for input

Maintenance *(REPAIR or ENHANCEMENT)*

- Repairing defects and adding capability

What is Design ? -1

- ◆ A bridge from specification to implementation
 - separate from both (specification and implementation), with its own methods and notations
 - requires both design expertise and domain knowledge
 - design increases in importance with system size and complexity

What is Design ? -2

- ◆ A product:
 - a document describing what is to be built and how it is to be built
- ◆ An activity:
 - a process of developing the product
 - begins by assuming desired end result is possible and seeks ways to achieve it

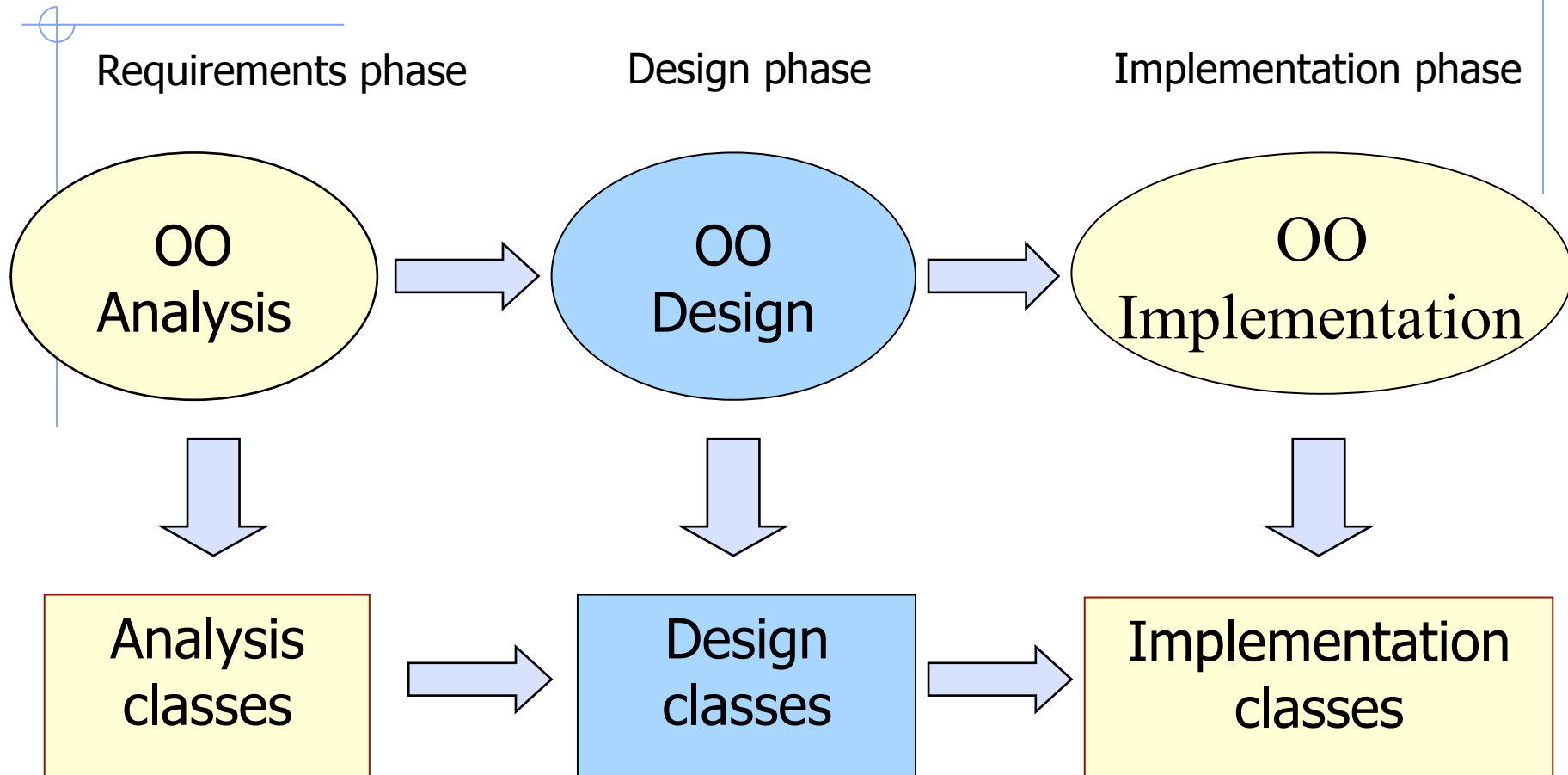
From Specification to Design

- ◆ A design product must satisfy its specification (at the least)
- ◆ It must be possible to map requirements to design elements
- ◆ A design activity may anticipate possible changes to requirements
- ◆ A design activity may suggest changes to the specification of requirements

From Design to Implementation

- ◆ A design product must be able to be built:
 - it is a plan for mapping design elements onto software elements
 - all design elements and actions must be realisable (implementable)
 - design elements should be individually testable and capable of integration
 - it should lead to an implementation that meets performance constraints

Object-oriented Development



OO Design -1

- ◆ Concerned with developing an object-oriented design model of a software system to implement requirements
- ◆ A design model contains both static (structural) and dynamic (behavioural) aspects:

Static model: Identifies design classes and their (static) class relationships: inheritance, aggregation, composition, dependency, association [Larman Ch. 16]

Dynamic model: Identifies interaction (messaging) between objects to fulfil requirements [Larman Ch. 15]

OO Design -2

- ◆ During the design process, analysis classes can evolve into design classes via refinement (i.e., by adding detail)
- ◆ New (non-domain) classes may be created to support design goals (e.g. promoting a primitive type to an object type)

UML

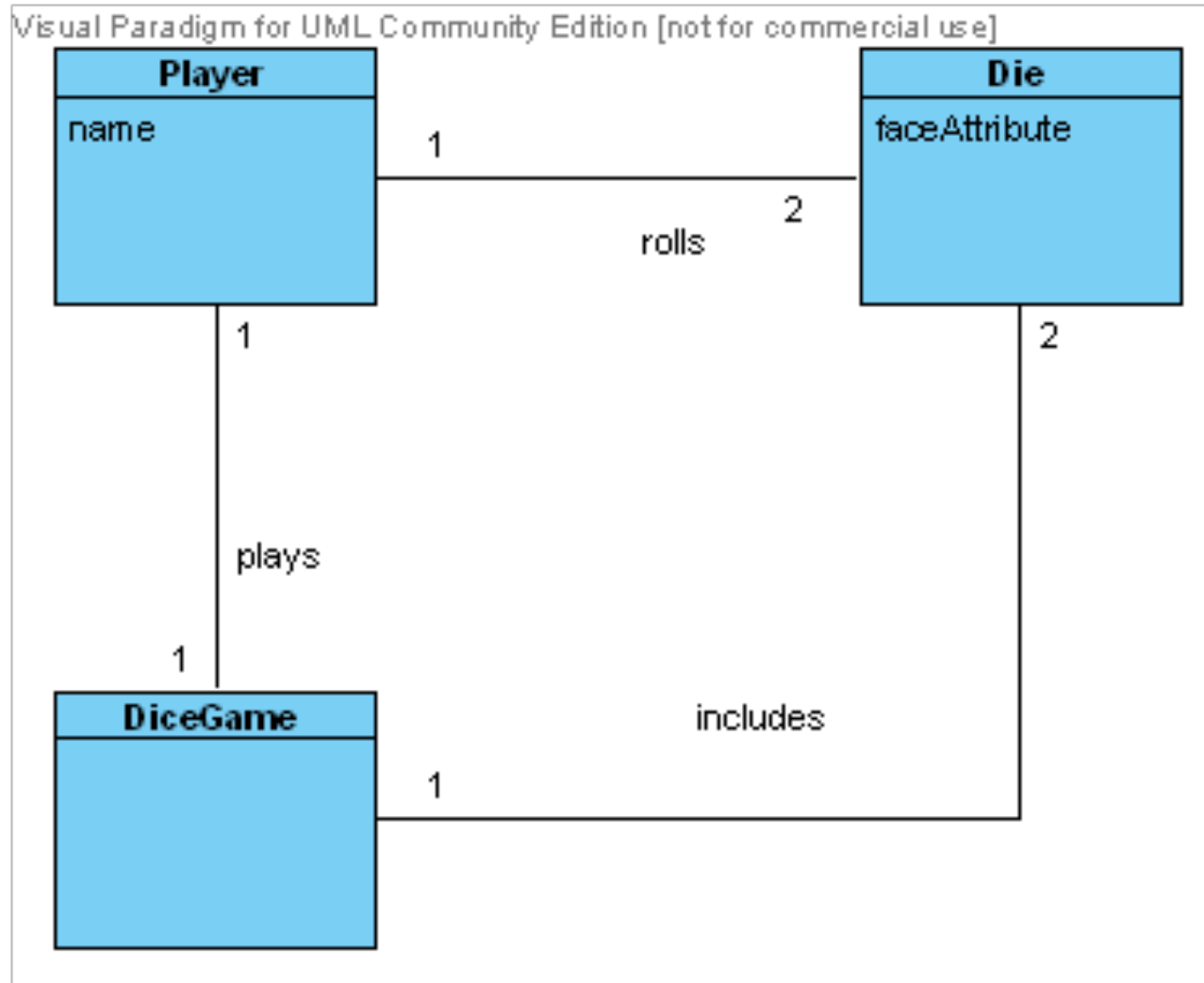
- ◆ UML – the Unified Modelling Language
- ◆ UML is a standard diagramming *notation* for representing information about software systems, particularly their design
- ◆ UML is useful for thinking and for communication
- ◆ This course will cover a small subset of UML:
 - most frequently used diagram types, and
 - most commonly used features
 - e.g. class and sequence diagrams

Example: Requirements (Larman, 1.5)

- ◆ Play a dice game:
 - a player requests to roll the dice
 - the system presents results:
 - if the dice face value totals seven, the player wins
 - otherwise, the player loses.



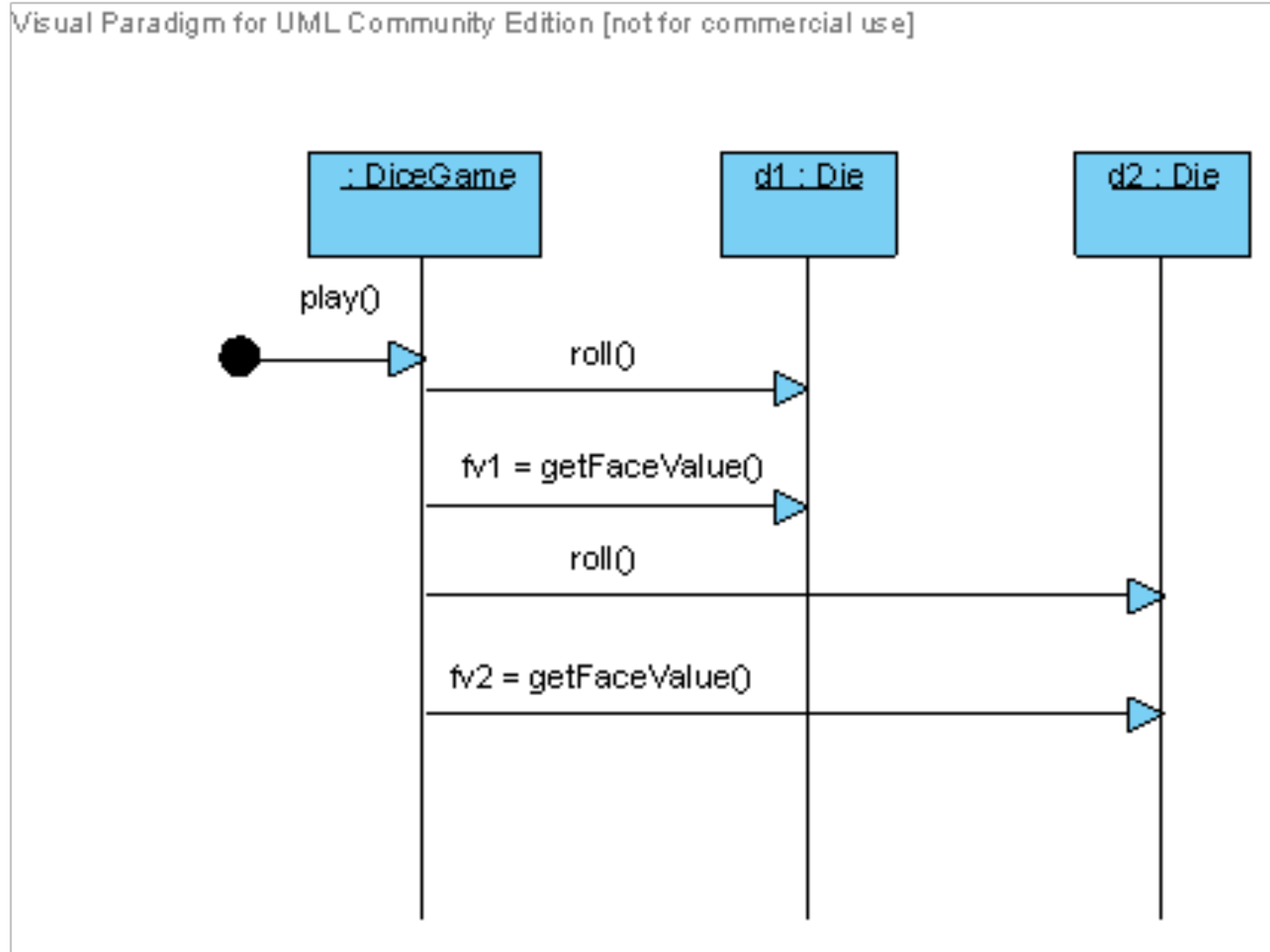
Example: Partial Domain Model



Domain Models

- ◆ Not a description of software objects
- ◆ Visualisation of the concepts or mental models of a real-world domain
- ◆ Also known as a “conceptual object model”

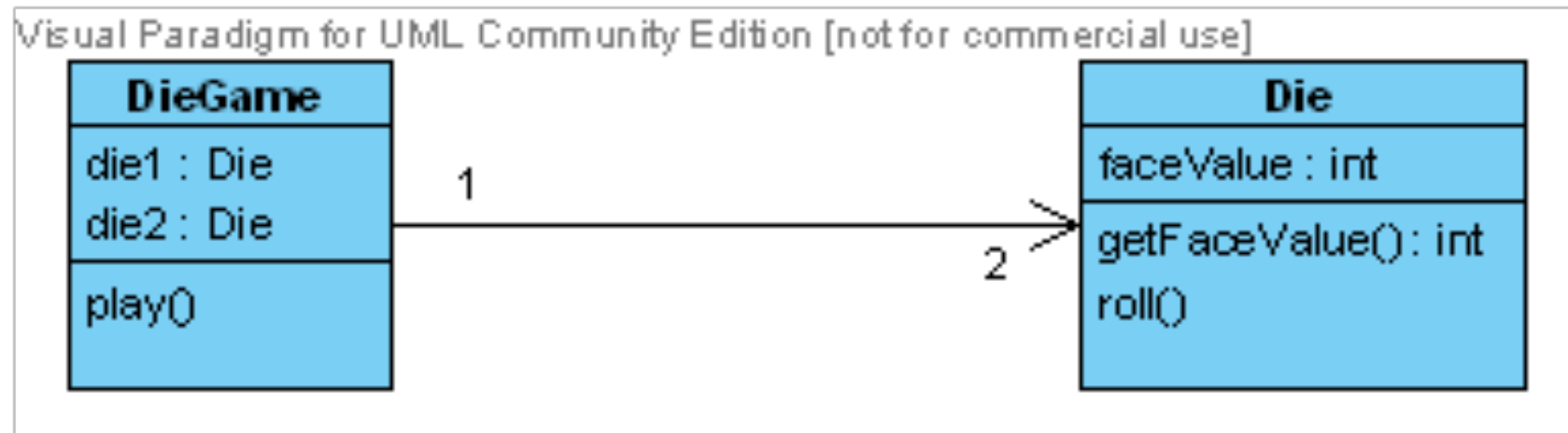
Example: Sequence Diagram



Sequence Diagrams

- ◆ Sequence diagrams show the flow of messages between software objects and hence how objects collaborate
- ◆ From the information in sequence diagrams about methods, we can develop a ***design class diagram*** showing software classes

Design Class Diagram



Ways to Apply UML

- ◆ UML as sketching language:
 - informal and incomplete
 - hand-drawn, often on white-boards
- ◆ UML as blueprint:
 - detailed design diagrams for reverse engineering from existing code, or for code generation
- ◆ UML as programming language:
 - complete executable specification of a system
 - developers do not see or modify generated code
 - currently research rather than practice

Perspectives for Applying UML

- ◆ Conceptual:
 - diagrams show objects from real world domain of interest
- ◆ Specification:
 - diagrams show software abstractions or components but with no commitment to a particular implementation
- ◆ Implementation:
 - diagrams describe software implementation for a particular technology (e.g., Java)

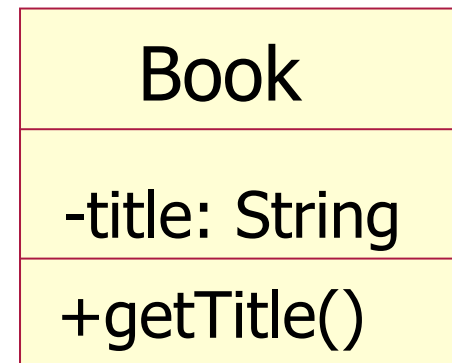
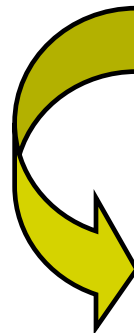
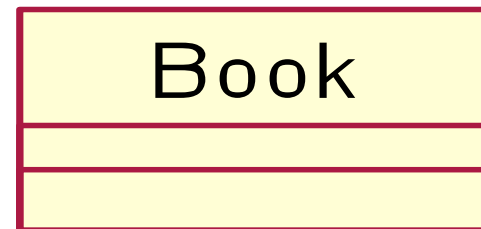
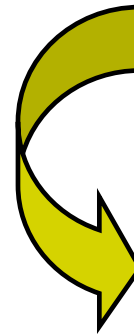
Classes Evolve

- ◆ At analysis, a domain concept *book* is made an analysis class Book
- ◆ At design, a design class Book is given attributes (e.g. title) and methods (e.g. getTitle)
- ◆ At implementation, a Java class

```
public class Book
{
    private String title;
    public String getTitle() {
        ...
    }
}
```

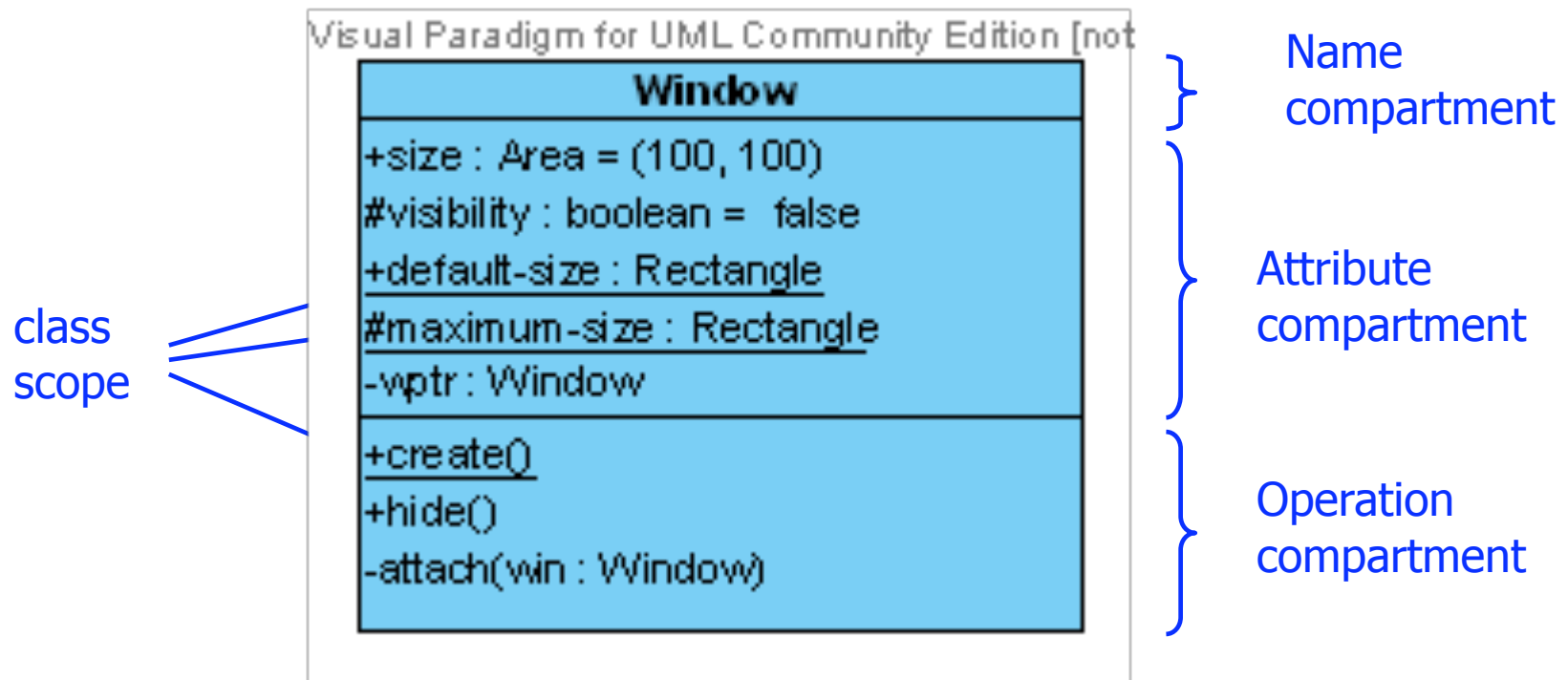
UML Class Notation - 1

- ◆ Concepts exist in the problem domain
- ◆ At analysis, an analysis class
- ◆ At design, a design class



UML Class Notation -2

- A class has attributes and operations



- An abstract class cannot create its own instances (italics indicates abstract class names).

Name Compartment

- ◆ A class name begins with an uppercase letter, after which either case can be used, according to the convention that each 'word' starts with uppercase
- ◆ E.g.,
 - DepositAccount
 - CreditAccount
 - SalesLineItem
 - FlightReservation
- ◆ Singleton classes with multiplicity



Attribute Compartment - 1

- ◆ The syntax of an attribute is
visibility name : type multiplicity = defaultValue
{properties and constraints}
- ◆ Visibility:
 - + (public)
 - - (private)
 - # (protected)
 - ~ (package)

Attribute Compartment - 2

- ◆ Multiplicity (omitted when [1..1])
 - rainbowColors : Colour [7]
a set of 7 Colour objects
 - name : String [2..*]
a set of 2 or more Strings
 - emailAddress : String [0..1]
 - ◆ a null reference or a String
- ◆ Constraints:
 - rainbowColors : Colour [7] {readOnly}
 - name : String [2..*] {ordered}
- ◆ OwnerScope
 - instance scope or
 - class scope (underlined) - static attributes

Operation Compartment

- ◆ The syntax of an operation is:
visibility name (parameterList) : returnType
{properties and constraints}
- ◆ OwnerScope : instance scope or class scope

Scope

- ◆ Instance scope:
 - Every object of a class has its own distinct copy of an instance scope attribute
 - An instance scope operation can only be applied to an instance of the class in which the operation is defined
- ◆ Class scope:
 - Every object of a class shares the same class scope attribute (this is as close as OO gets to global variables)
 - A class scope operation belongs to a class, not to an instance of that class (analogous to a static method in Java)

Object Constructors

- ◆ An object constructor is a special operation that creates a new instance of a class
- ◆ A constructor need not be introduced into an analysis model; it is more important in a design model
- ◆ It is usually identified by a generic name such as 'create'

- ◆ It has class scope

BankAccount
<u>create(aNumber: Int)</u>

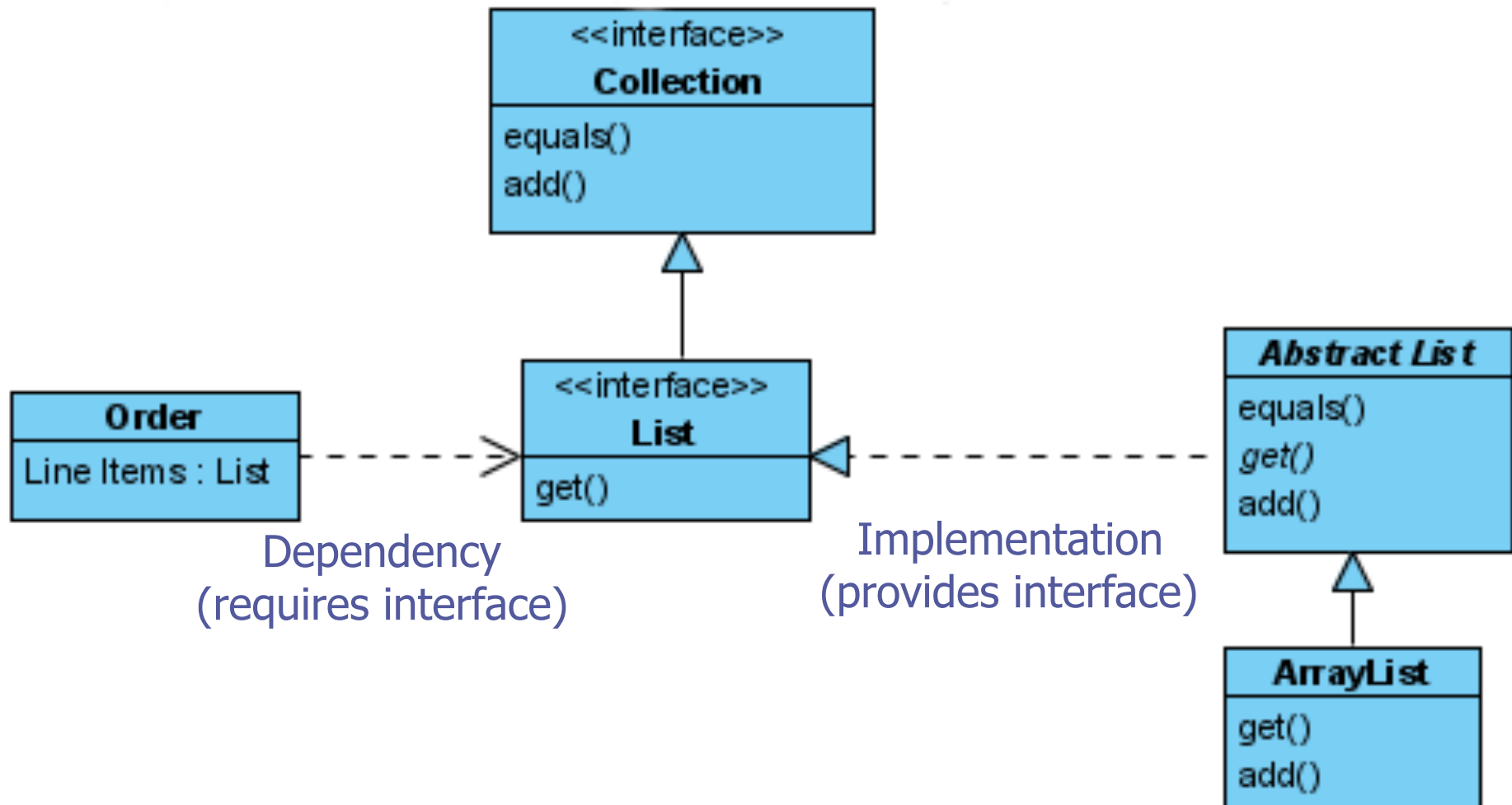
- ◆ *Compare*

BankAccount
<u>BankAccount(aNumber: Int)</u>

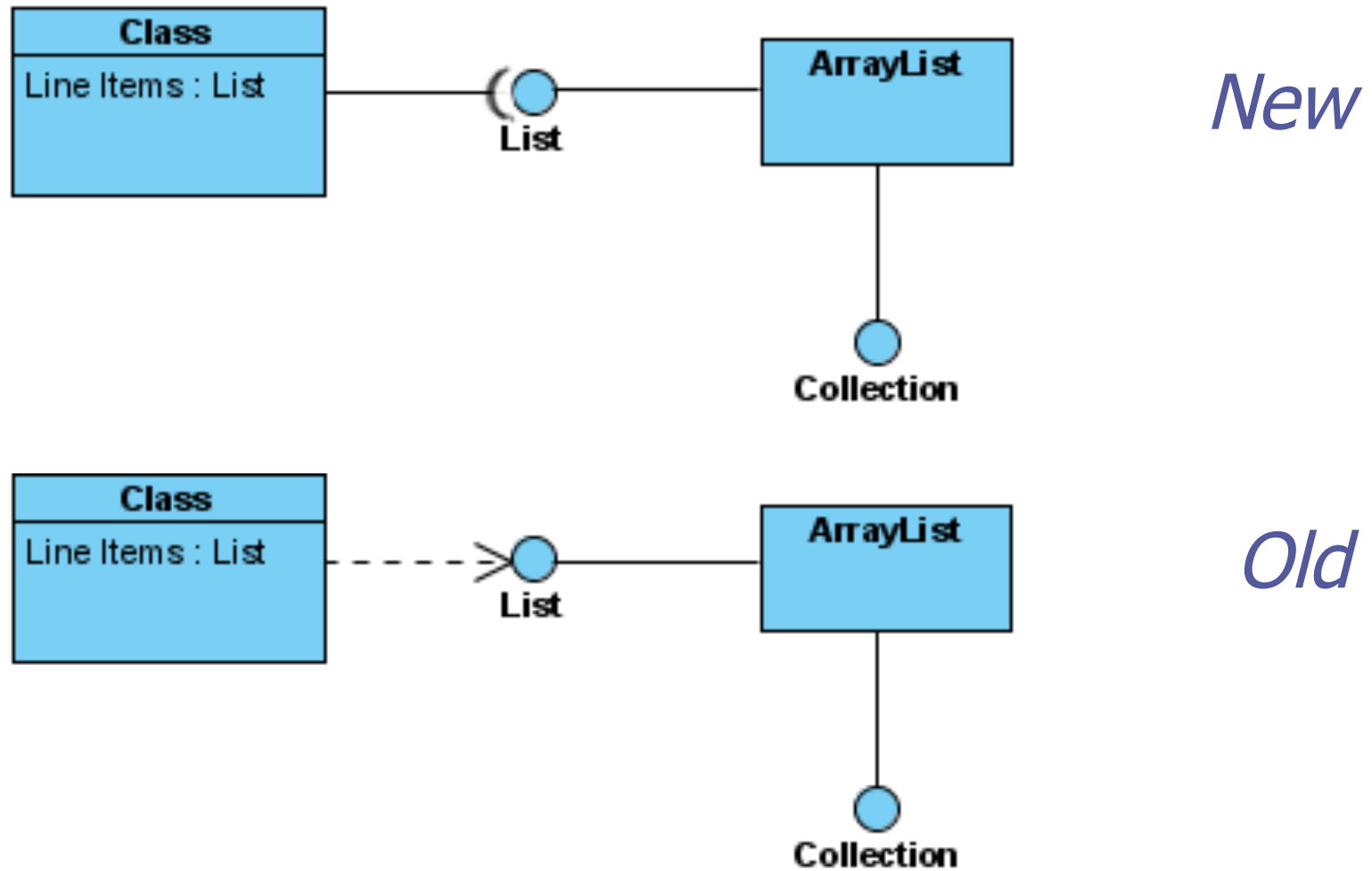
Interfaces in UML

- ◆ Special type of class with only public operations and no method bodies or attributes
- ◆ Designated in several ways:
 - class icon with keyword «interface» before name
 - ball/lollipop icon
- ◆ Two types of relationships with interfaces
 - A class **provides** an interface if it implements the interface (is substitutable for the interface)
 - A class **requires** an interface if it needs an instance of that interface (by having a dependency on the interface)

Interface Example -1 (Fowler)

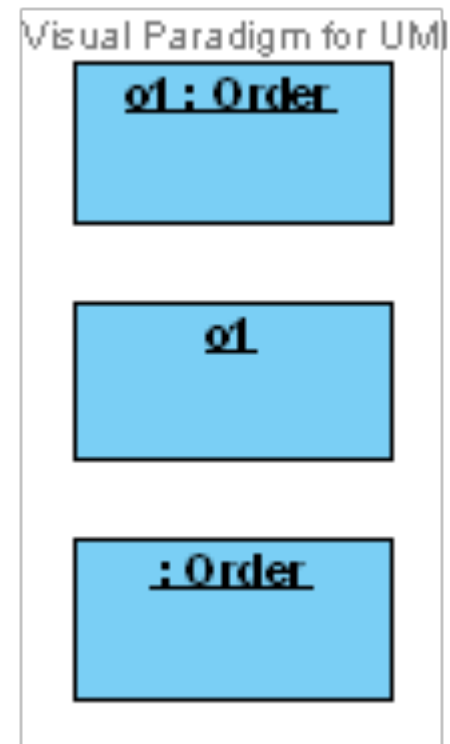
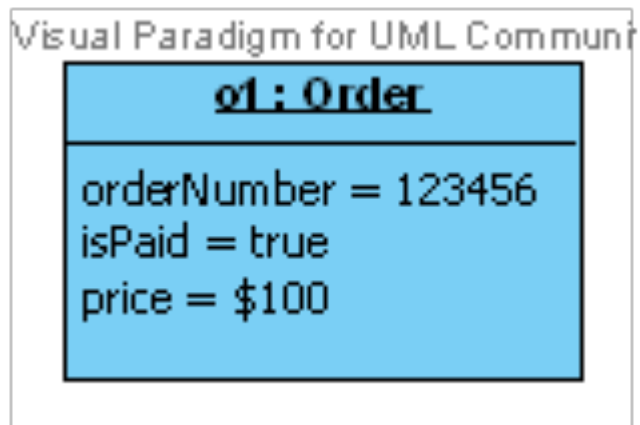


Interface Example -2 (Fowler)



UML Object Notation

- ◆ The syntax is objectname: ClassName
- ◆ Used in sequence (and collaboration) diagrams



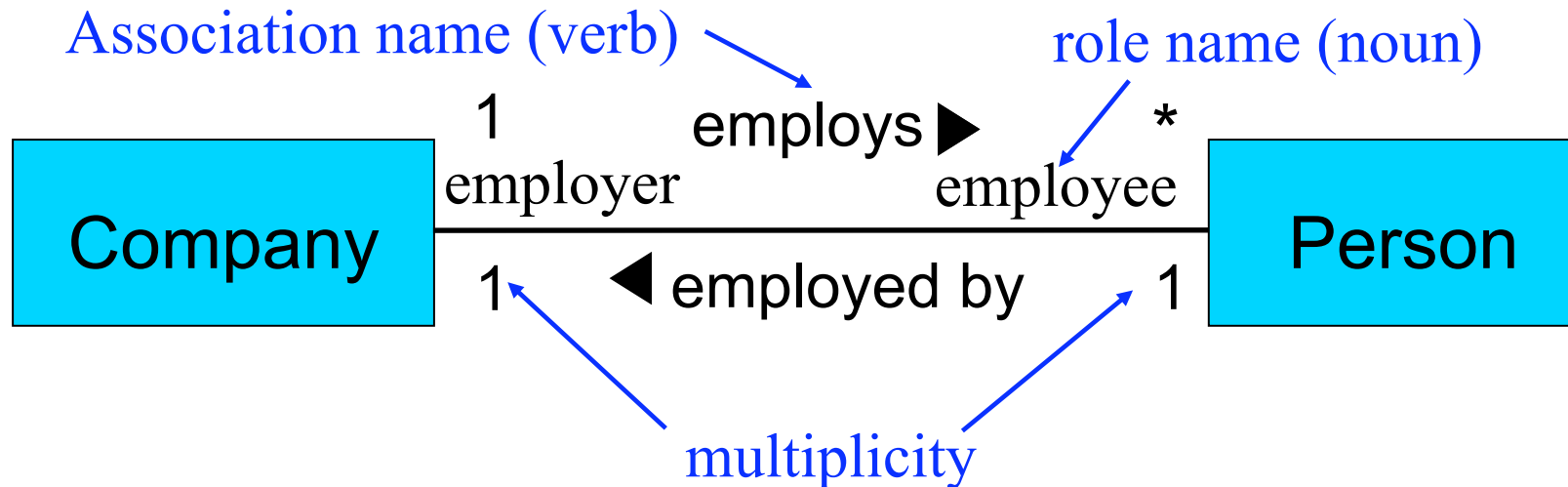
Types of Class Relationship

- ◆ Association
- ◆ Aggregation (and composition)
- ◆ Inheritance (a.k.a. generalisation)
- ◆ Dependency

What is an Association?

- ◆ Describes a relationship of structural dependency between classes
- ◆ An association may have:
 - a name
 - role names
 - a multiplicity, e.g. 0..1 (0 or 1), 1 (just 1), 1..* (1 or more), * (any number, same as 0..*) , 7..10 (between 7 and 10), ...
 - Navigability (the association may be uni/bi-directional)
- ◆ [Larman Ch 16]

An Example of a UML Association

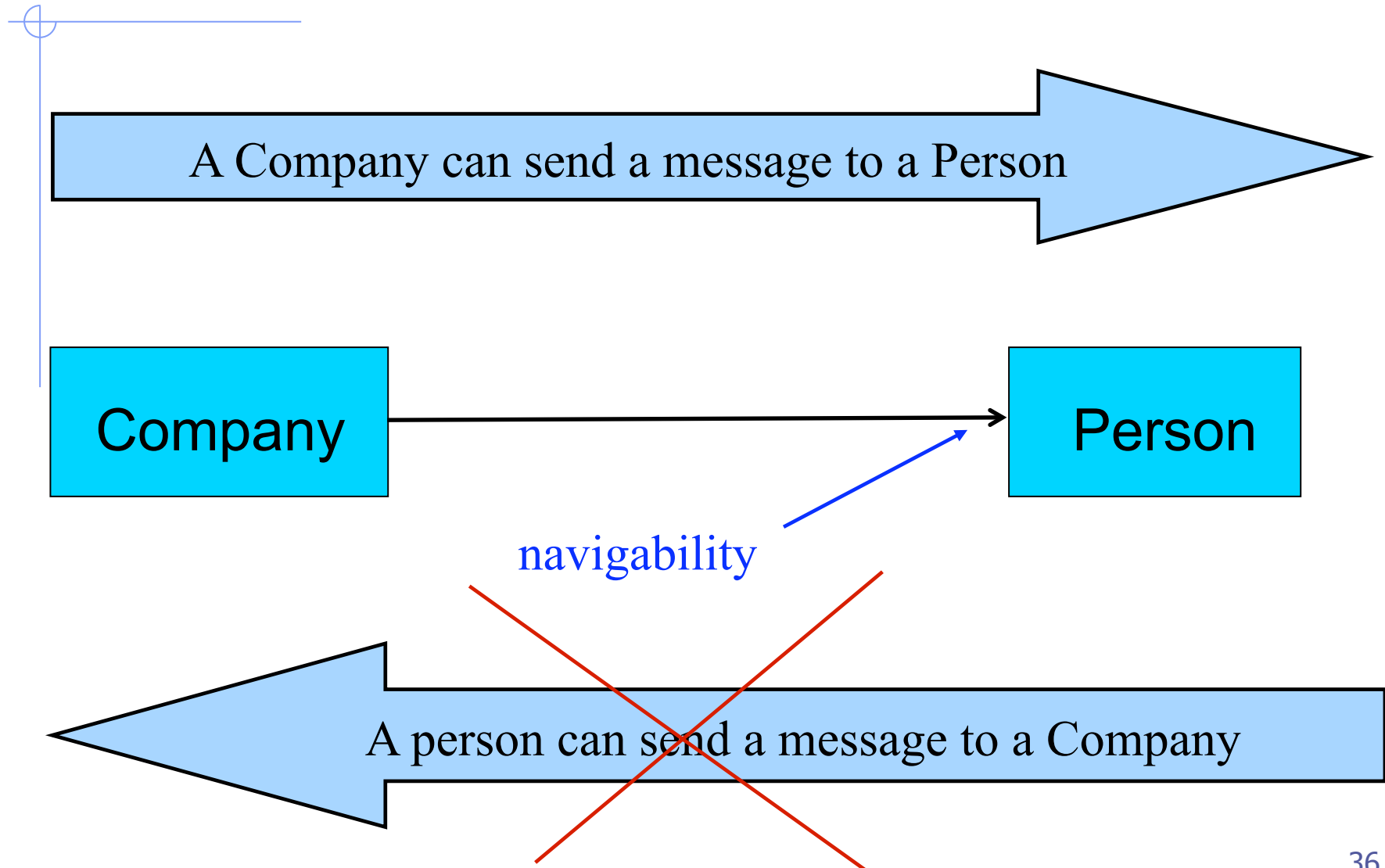


- ◆ The association expresses two relationships

A Company employs many Persons

A Person is employed by one Company

Navigability

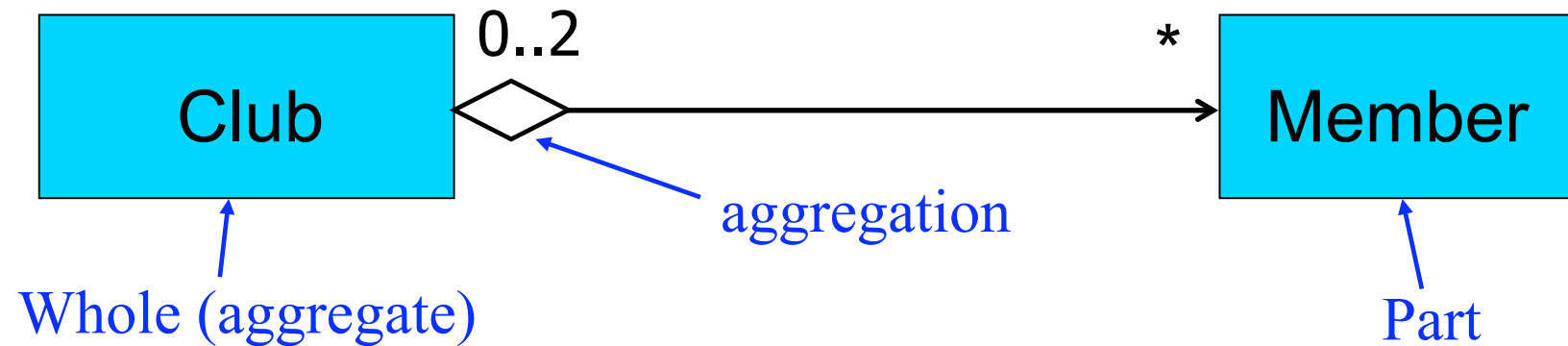


What is an Aggregation?

- ◆ A whole-part relationship where an object (the 'whole') aggregates several other objects (the 'parts') [Larman 16.13]
- ◆ Two types of aggregation :
 - An aggregation: a loose relationship between objects, e.g. a computer and its peripheral devices
 - A composition: a stronger relationship where the related objects are more tightly bound together, e.g.
 - ◆ a computer and its CPU
 - ◆ a mouse and its buttons

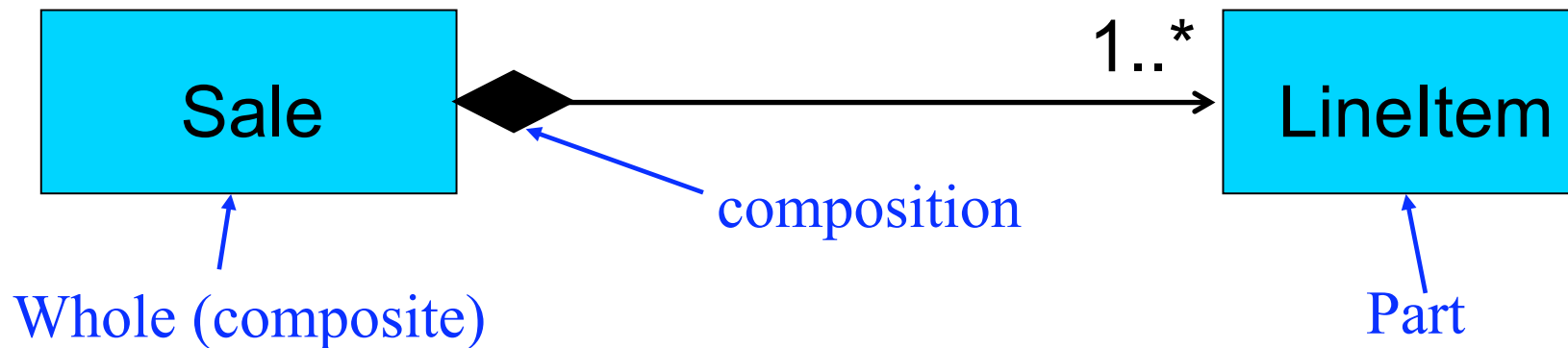
An Example of a UML Aggregation

- ◆ A Club may have 0 or more Members.
- ◆ A Member may belong to 0, 1 or 2 Clubs.



An example of a UML Composition

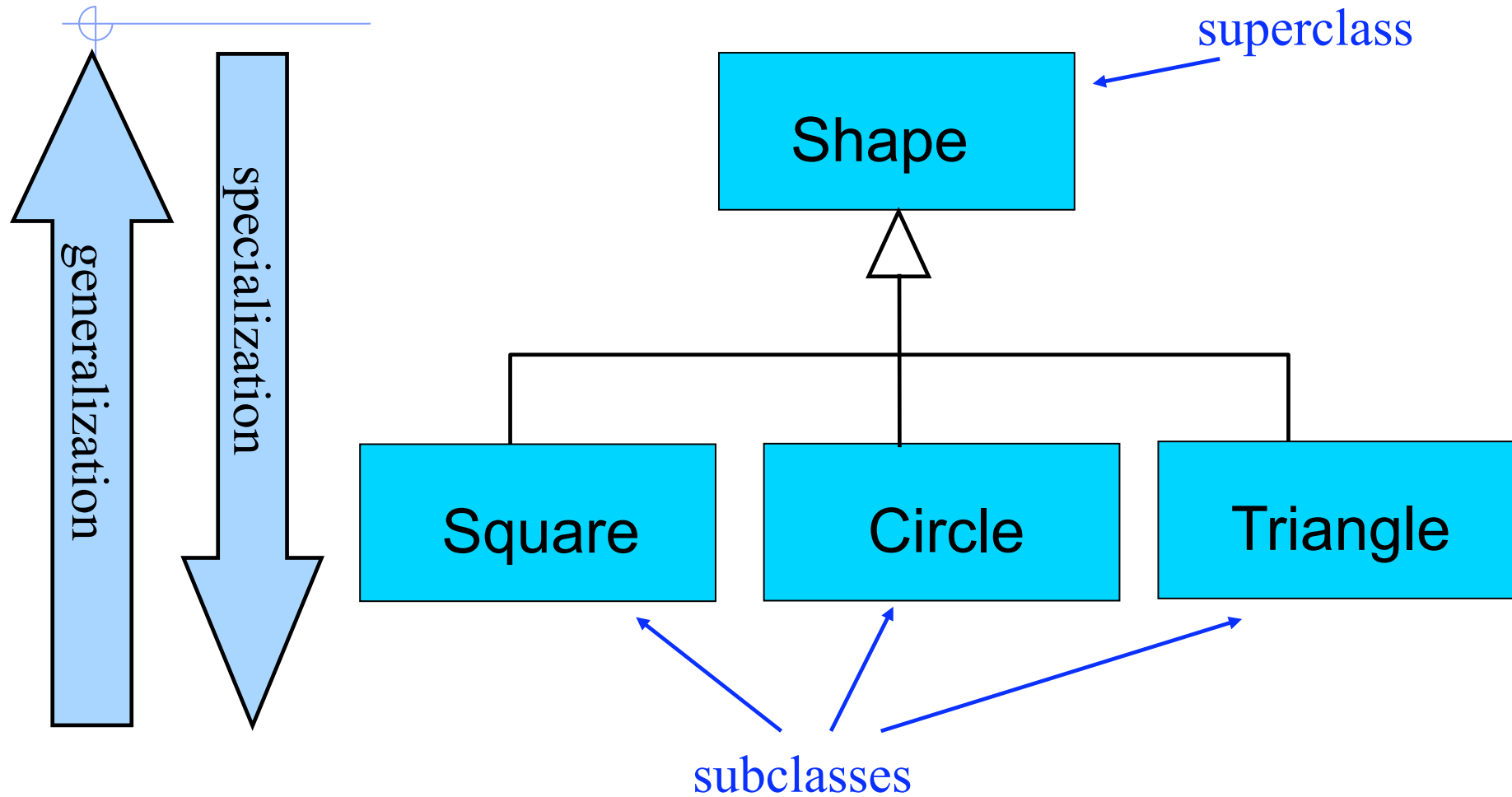
- ◆ A LineItem belongs to exactly one Sale.
- ◆ A Sale has sole responsibility for the creation, use and disposal of its LineItems.
- ◆ When a Sale is destroyed, it must either destroy its LineItems, or pass responsibility for them to another Sale.



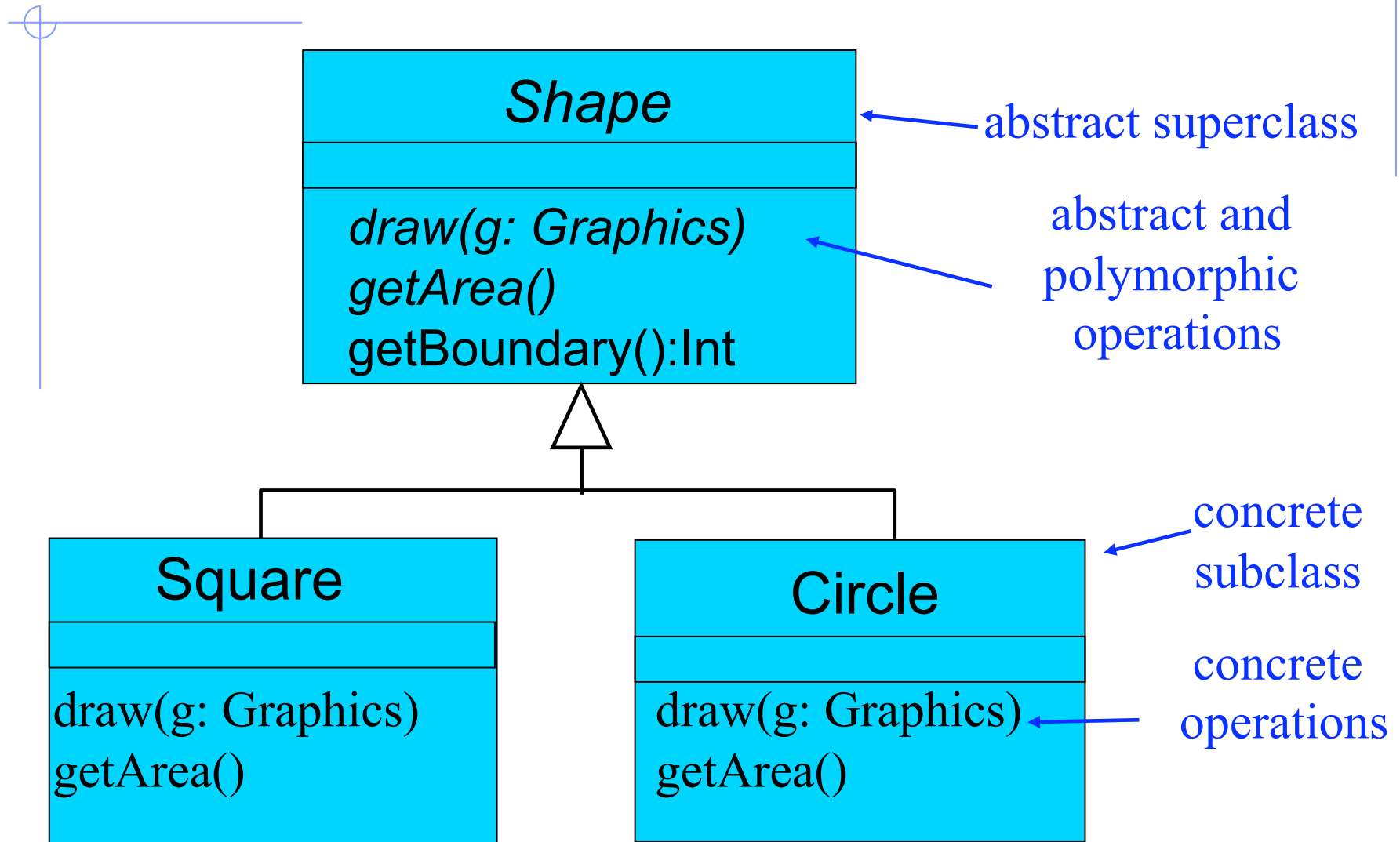
What is Inheritance ?

- ◆ A directed inheritance relationship models an “is a kind of” relationship between classes
- ◆ This is also a subtyping relationship and supports an important principle
 - Substitutability: a subclass instance can always be used where a superclass instance is expected
- ◆ A superclass method can be overridden (modified) in a subclass to capture semantic variations in the subclass
- ◆ [Larman 16.10]

An Example of Inheritance - 1



An Example of Inheritance - 2



What is a Dependency ?

- ◆ A weak relationship indicating that one class depends on another in a non-structural way, i.e., in a manner other than by association or inheritance.
- ◆ Used when a method definition (e.g., parameter variables) in class A uses class B
- ◆ [Larman 16.11]



Follow-up Reading

- ◆ Larman: Chapters 1 & 14
- ◆ Larman: Chapter 16 (UML class diagrams)
- ◆ Fowler: UML Distilled (3rd Edition)
 - Chapters 3 & 5