UML Class Diagram

Deepali Londhe

What is a Class Diagram?

- A class diagram describes the types of objects in the system and the various kinds of static relationships that exist among them.
  - A graphical representation of a static view on declarative static elements.
- A central modeling technique that runs through nearly all object-oriented methods.
- The richest notation in UML.

Essential Elements of a UML Class Diagram

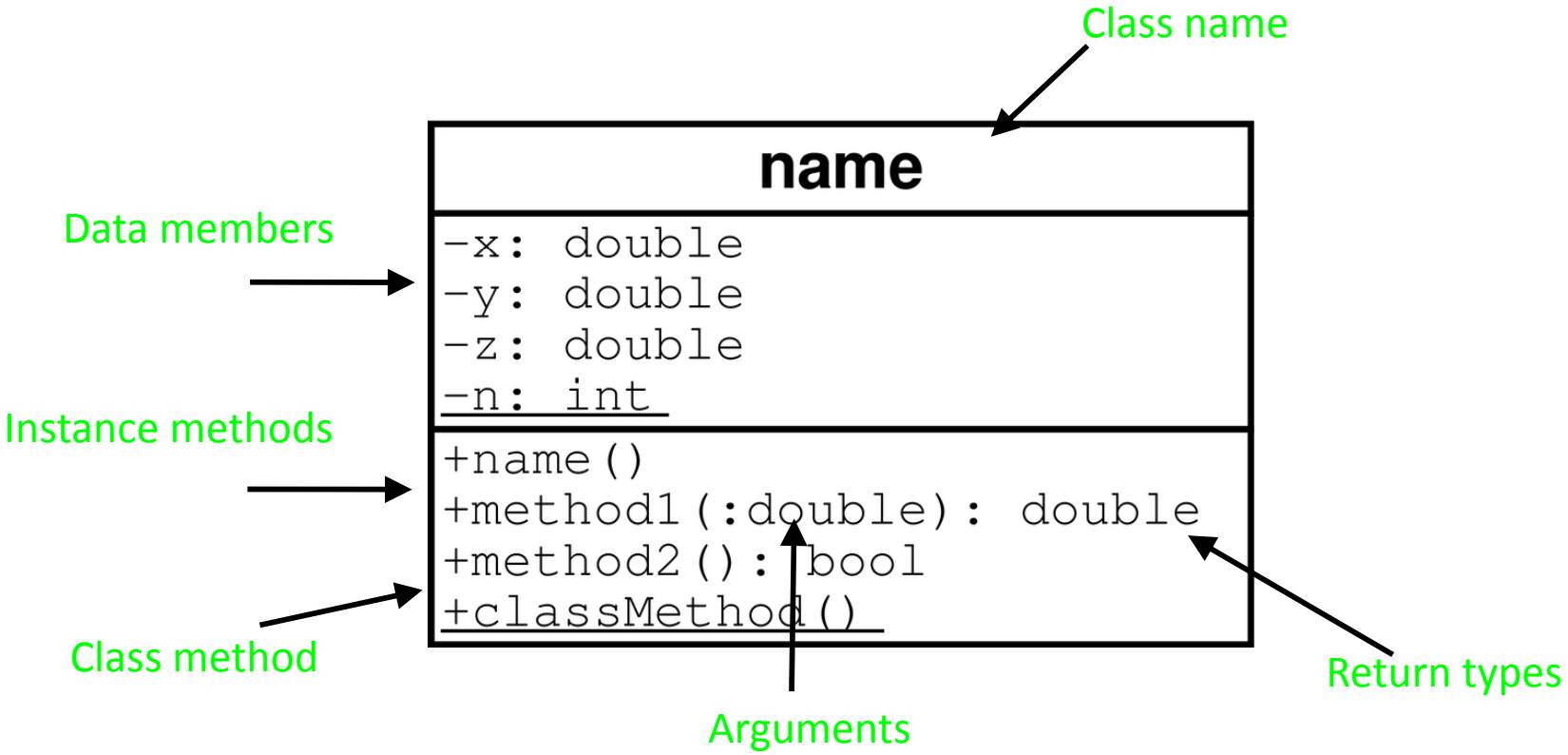
- Class
- Attributes
- Operations
- Relationships
  - Associations
  - Generalization
  - Dependency
  - Realization
- Constraint Rules and Notes

Classes in UML

- Classes describe objects
  - Interface (member function signature)
  - Behaviour (member function implementation)
  - State bookkeeping (values of data members)
  - Creation and destruction
- Objects described by classes collaborate
  - Class relations $\rightarrow$ object relations
  - Dependencies between classes

UML Class

A class is the description of a set of objects having similar attributes, operations, relationships and behavior.



Data members, arguments and methods are specified by
visibility name : type

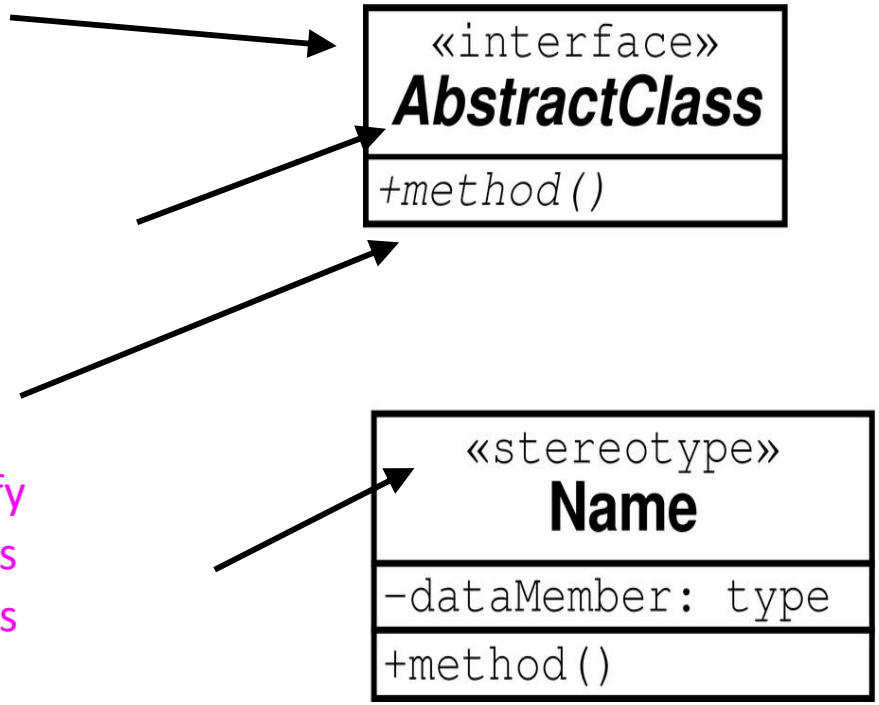
Class Name

The top compartment contains the class name

Abstract classes have italicised names

Abstract methods also have italicised names

Stereotypes are used to identify groups of classes, e.g interfaces or persistent (storeable) classes



Class Attributes

Attributes are the instance and class data members

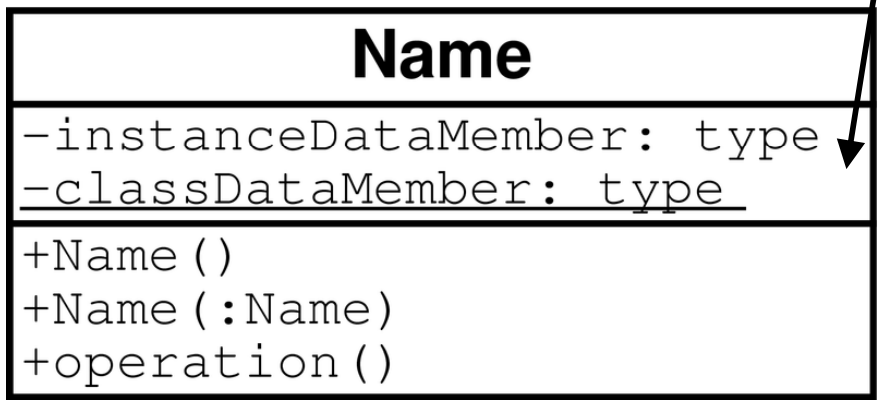
Class data members (underlined) are shared between all instances (objects) of a given class

Data types shown after ":"

Visibility shown as

+	public
-	private
#	protected

Attribute compartment



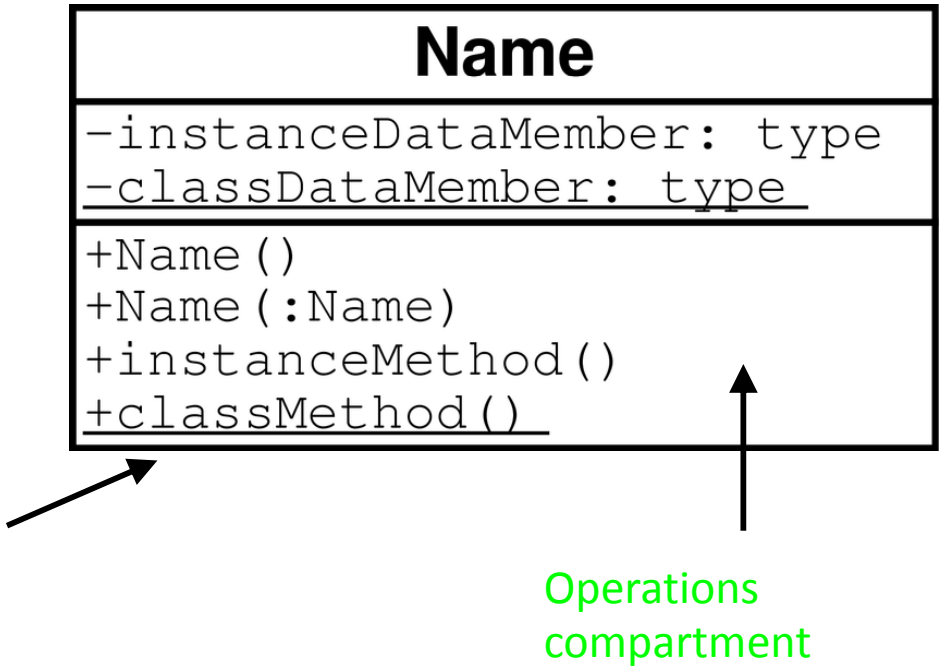
visibility name : type

Class Operations (Interface)

Operations are the class methods with their argument and return types

Public (+) operations define the class interface

Class methods (underlined) have only access to class data members, no need for a class instance (object)



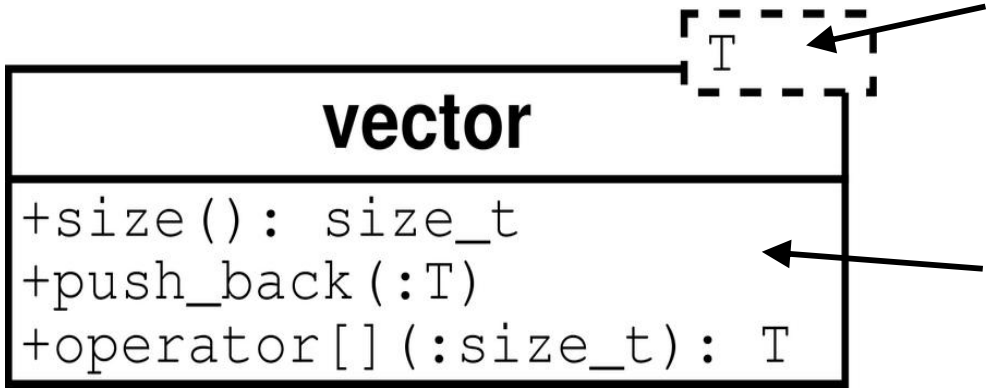
visibility name : type

Visibility

+	-	#
public	private	protected
Anyone can access	No-one can access	Subclasses can access
Interface operations	Data members	Operations where sub-classes collaborate
Not data members	Helper functions	Not data members (creates dependency off subclass on im-plementation of parent)
	"Friends" are allowd in though	

Template Classes

Generic classes depending on parametrised types



Type parameter(s)

Operations compartment as usual, but may have type parameter instead of concrete type

Relations

- Association
- Aggregation
- Composition
- Parametric and Friendship
- Inheritance

Associations

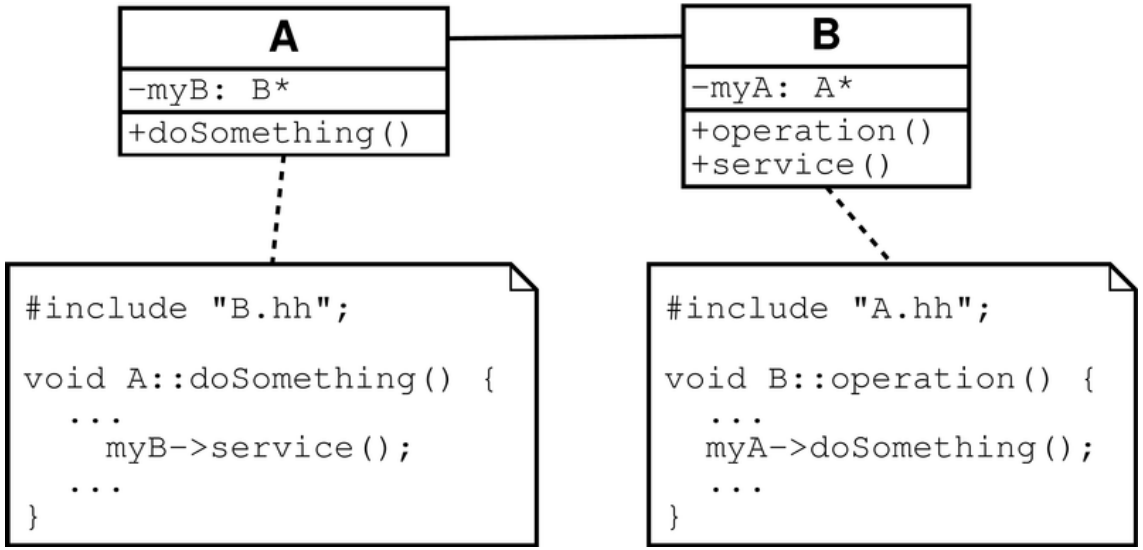
- A semantic relationship between two or more classes that specifies connections among their instances.
- A structural relationship, specifying that objects of one class are connected to objects of a second (possibly the same) class.
- Example: “An Employee works for a Company”

Associations (cont.)

- An association between two classes indicates that objects at one end of an association "recognize" objects at the other end and may send messages to them.
  - This property will help us discover less trivial associations using interaction diagrams.

Binary Association

Binary association: both classes know each other



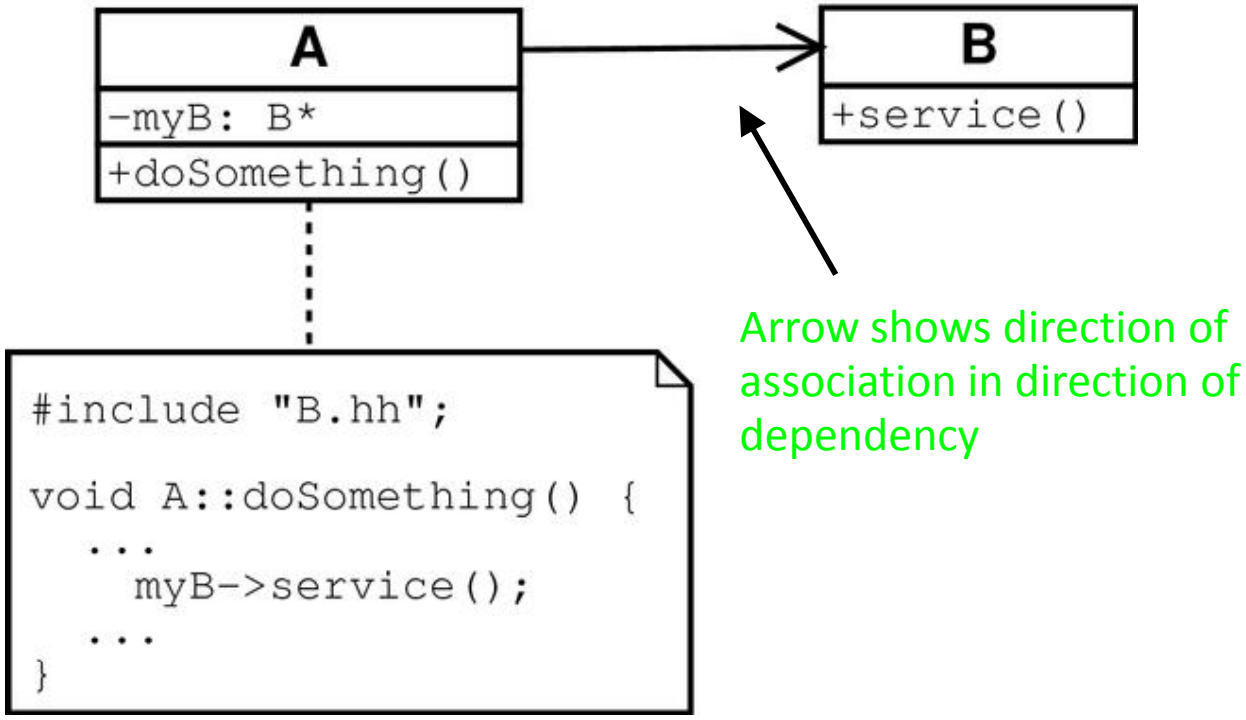
Usually "knows about" means a pointer or reference

Other methods possible: method argument, tables, database, ...

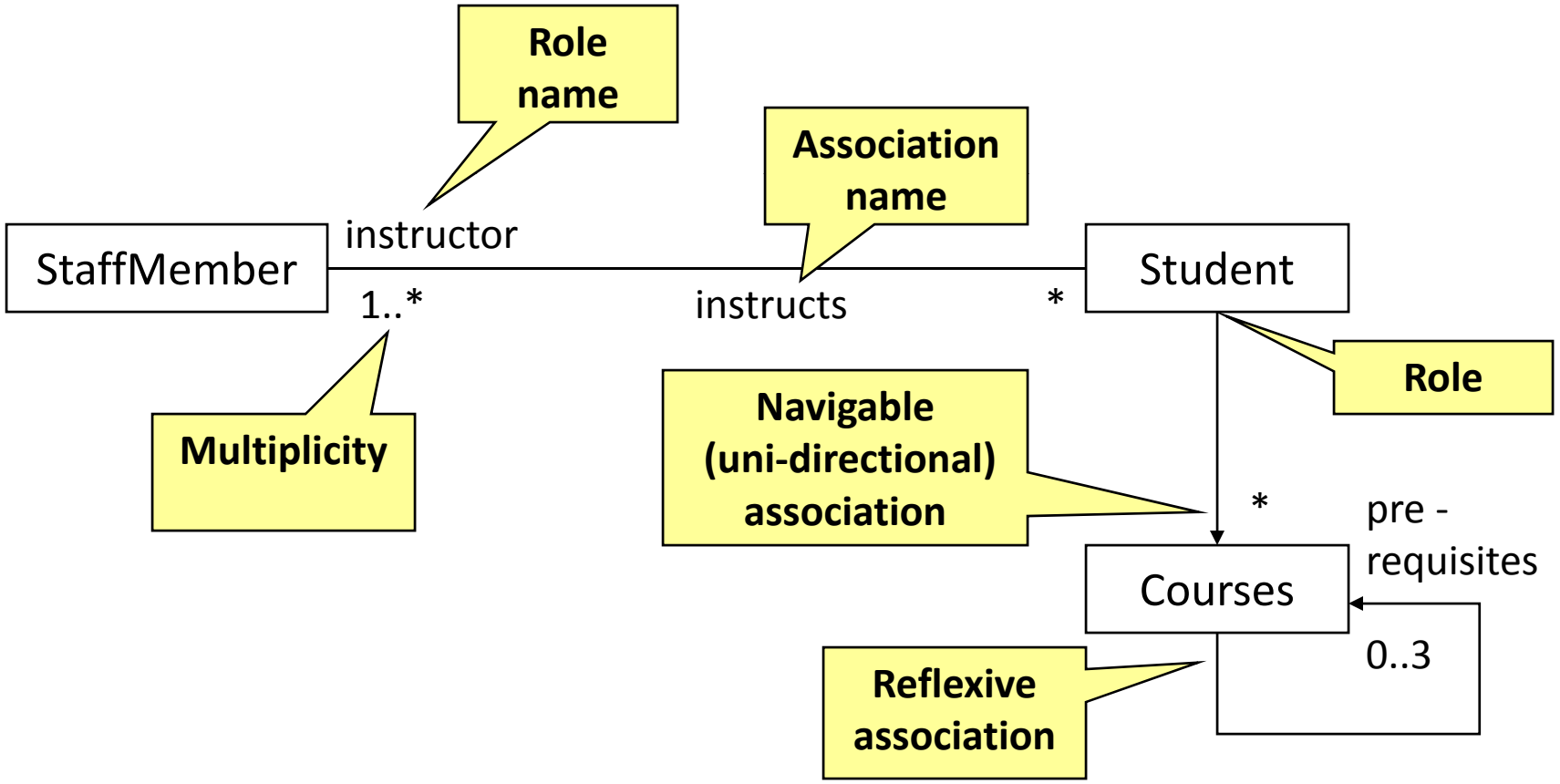
Implies dependency cycle

Unary Association

A knows about B, but B knows nothing about A



Associations (cont.)



Associations (cont.)

- To clarify its meaning, an association may be named.
  - The name is represented as a label placed midway along the association line.
  - Usually a verb or a verb phrase.
- A **role** is an end of an association where it connects to a class.
  - May be named to indicate the role played by the class attached to the end of the association path.
    - Usually a noun or noun phrase
    - Mandatory for reflexive associations

Associations (cont.)

- Multiplicity
  - The number of instances of the class, next to which the multiplicity expression appears, that are referenced by a **single** instance of the class that is at the other end of the association path.
  - Indicates whether or not an association is mandatory.
  - Provides a lower and upper bound on the number of instances.

Associations (cont.)

– Multiplicity Indicators

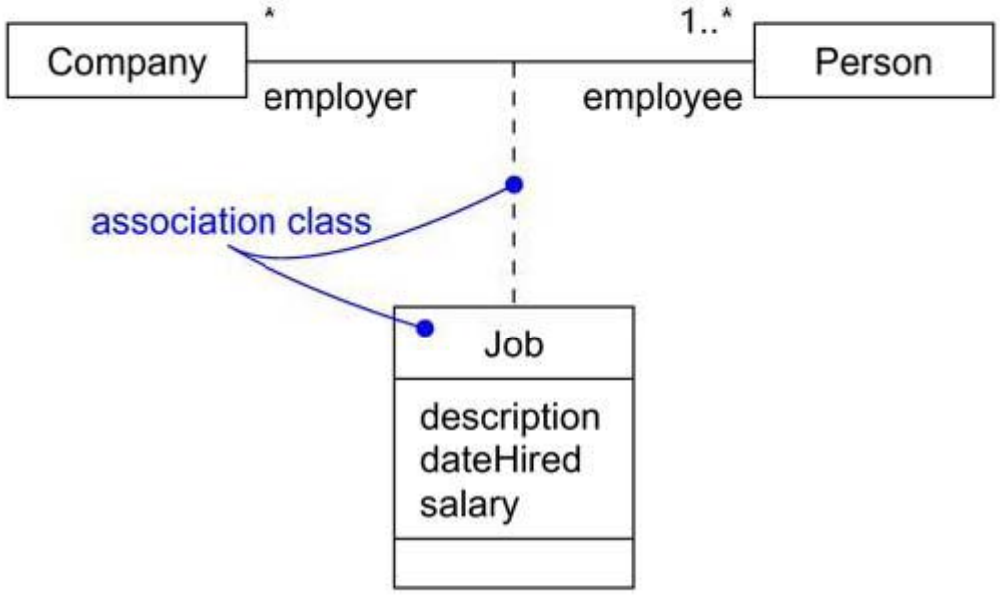
Exactly one	1
Zero or more (unlimited)	* (0..*)
One or more	1..*
Zero or one (optional association)	0..1
Specified range	2..4
Multiple, disjoint ranges	2, 4..6, 8

Association Classes

- In an association between two classes, the association itself might have properties.
- For example,in an employer/employee relationship between a Company and a Person, there is a Job that represents the properties of that relationship that apply to exactly one pairing of the Person and Company. It wouldn't be appropriate to model this situation with a Company to Job association together with a Job to Person association. That wouldn't tie a specific instance of the Job to the specific pairing of Company and Person.

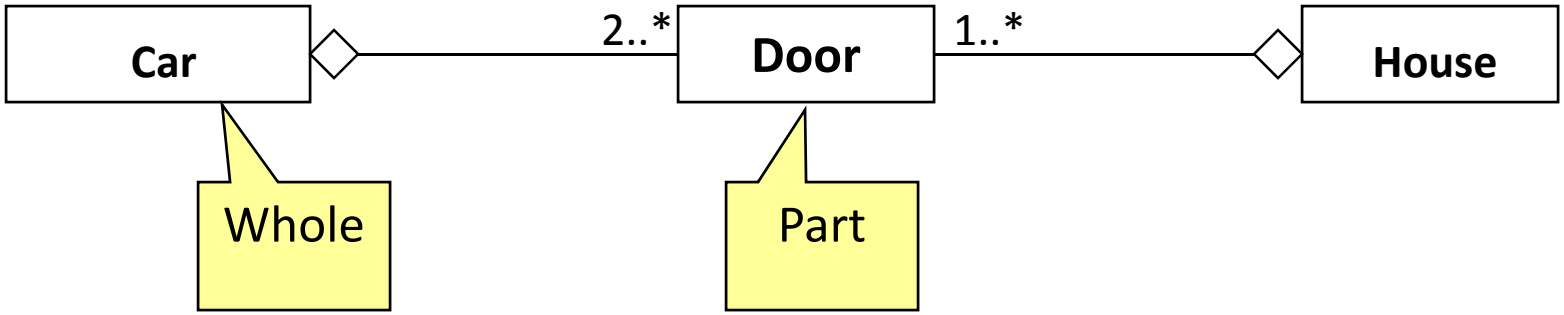
Association Classes Contd.

- In UML this as an association class, which is a modeling element that has both association and class properties.



Aggregation

- A special form of association that models a whole-part relationship between an aggregate (the whole) and its parts.
  - Models a “is a part-part of” relationship.
- One class represents a larger thing, which consists of smaller things.
- Has-a relationship, meaning “an object of whole **has** object of the parts

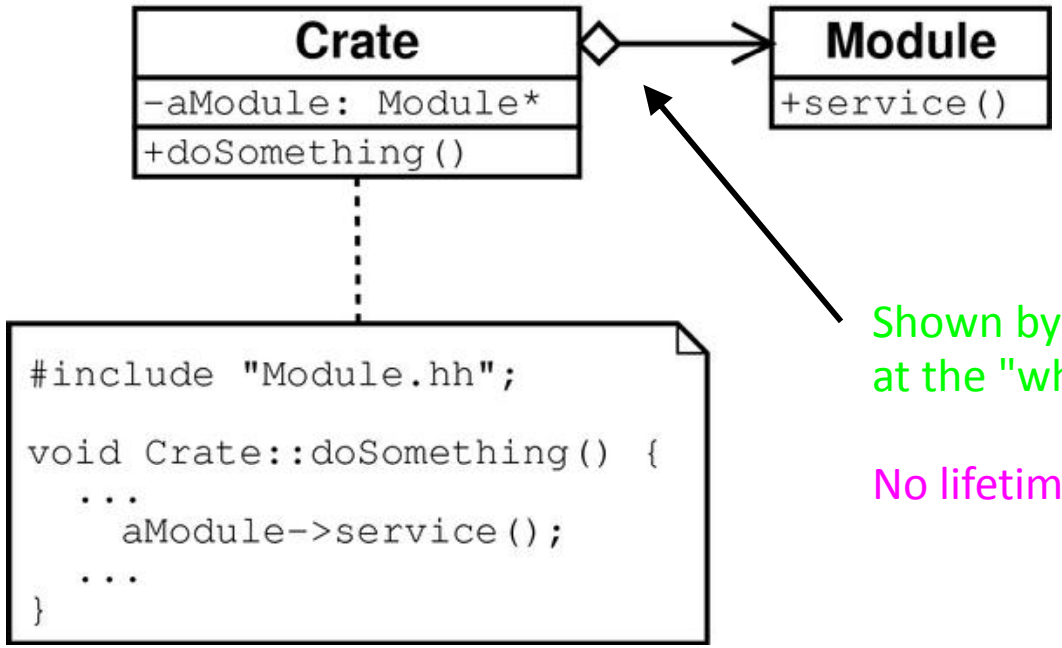


Aggregation (cont.)

- Aggregation tests:
  - Is the phrase “part of” used to describe the relationship?
    - A door is “part of” a car
  - Are some operations on the whole automatically applied to its parts?
    - Move the car, move the door.
  - Are some attribute values propagated from the whole to all or some of its parts?
    - The car is blue, therefore the door is blue.
  - Is there an intrinsic asymmetry to the relationship where one class is subordinate to the other?
    - A door **is** part of a car. A car **is not** part of a door.

Aggregation

Aggregation = Association with "whole-part" relationship

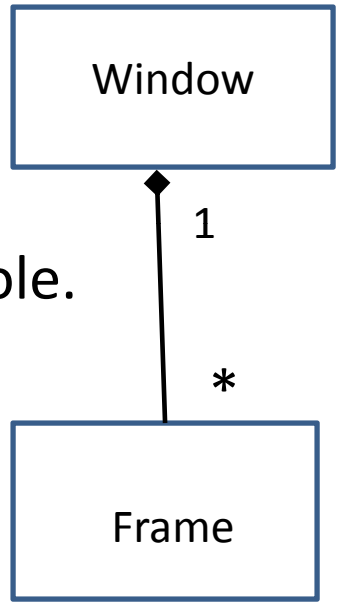


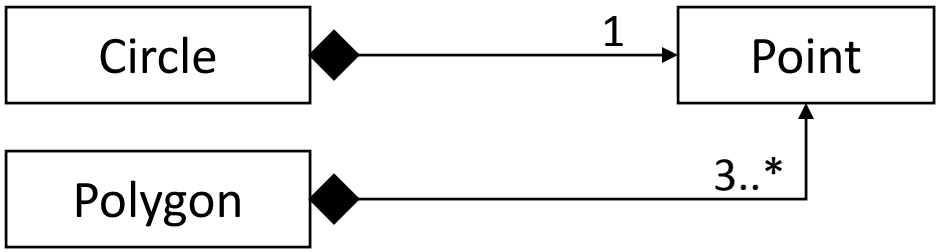
Shown by hollow diamond at the "whole" side

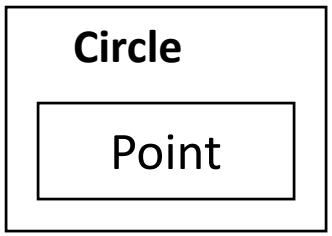
No lifetime control implied

Composition

- A strong form of aggregation
  - The whole is the sole owner of its part.
    - The part object may belong to only one whole
  - Multiplicity on the whole side must be zero or one.
  - The life time of the part is dependent upon the whole.
    - Once created parts live and die with whole.
    - The composite(Whole) must manage the creation and destruction of its parts.

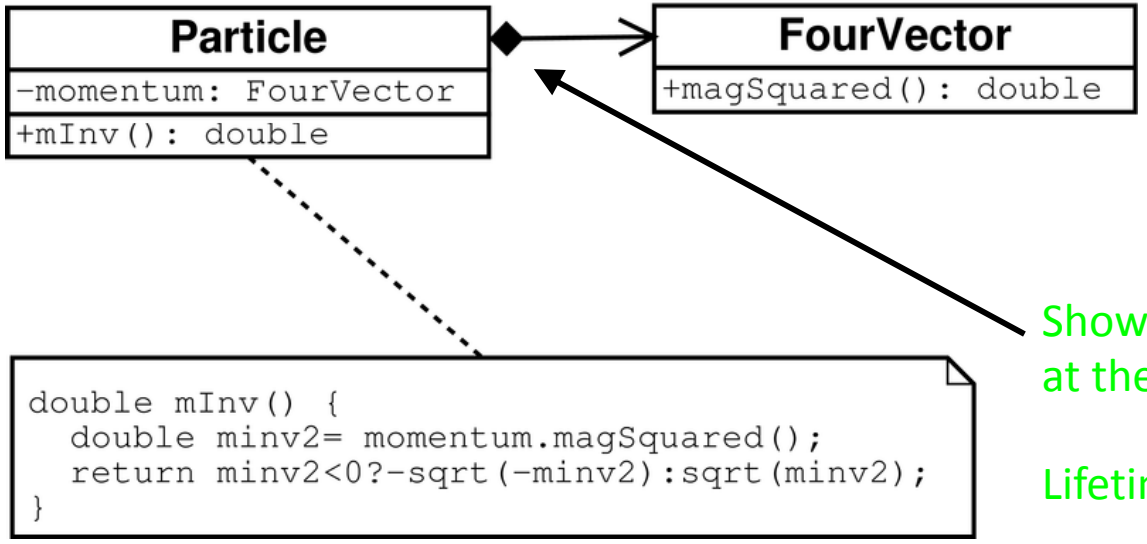






Composition

Composition = Aggregation with lifetime control



Shown by filled diamond at the "owner" side

Lifetime control implied

Lifetime control can be transferred

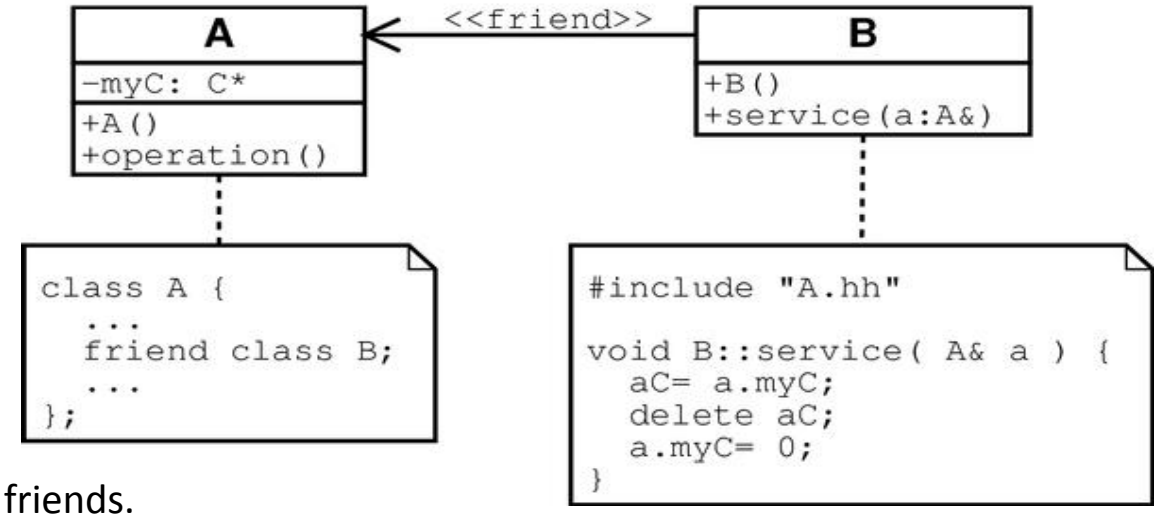
Lifetime control: construction and destruction controlled by "owner"

→ call constructors and destructors (or have somebody else do it)

Friendship

Friends are granted access to private data members and member functions

Friendship is given to other classes, never taken

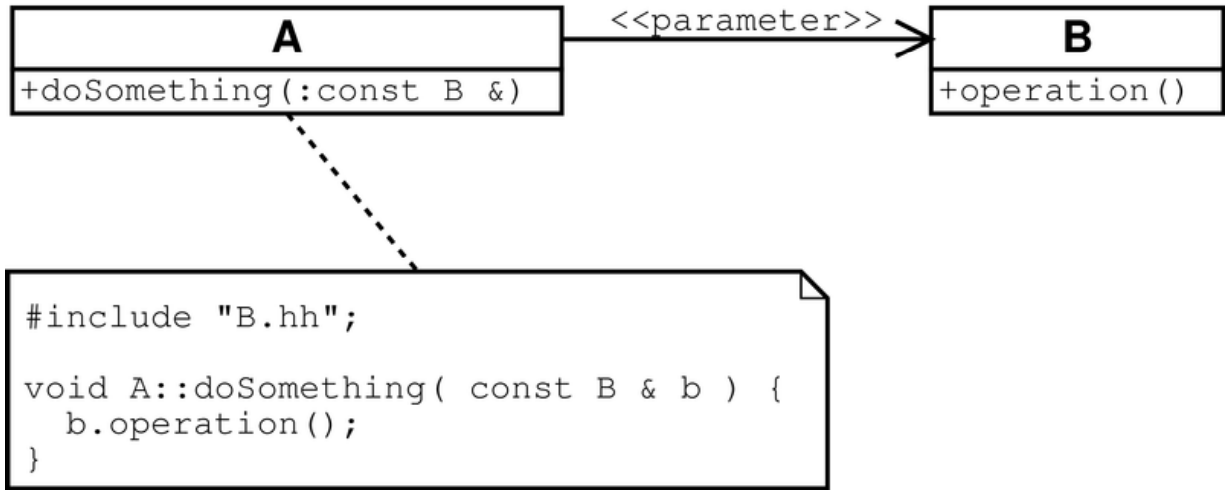


Bob Martin:
More like lovers than friends.
You can have many friends,
you should not have many lovers

Friendship breaks data hiding, use carefully

2.3 Parametric Association

Association mediated by a parameter (function call argument)

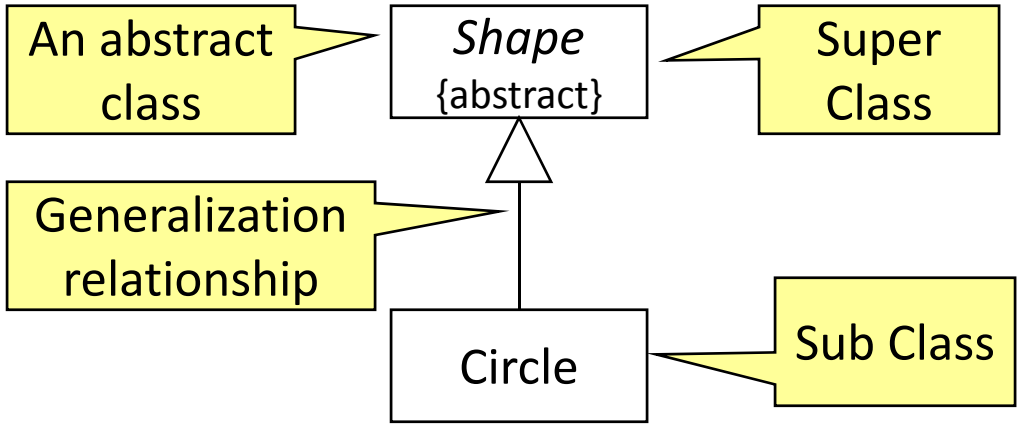


A depends upon B, because it uses B
No data member of type B in A

Generalization

- Indicates that objects of the specialized class (subclass) are substitutable for objects of the generalized class (super-class).
  - "is kind of" relationship.

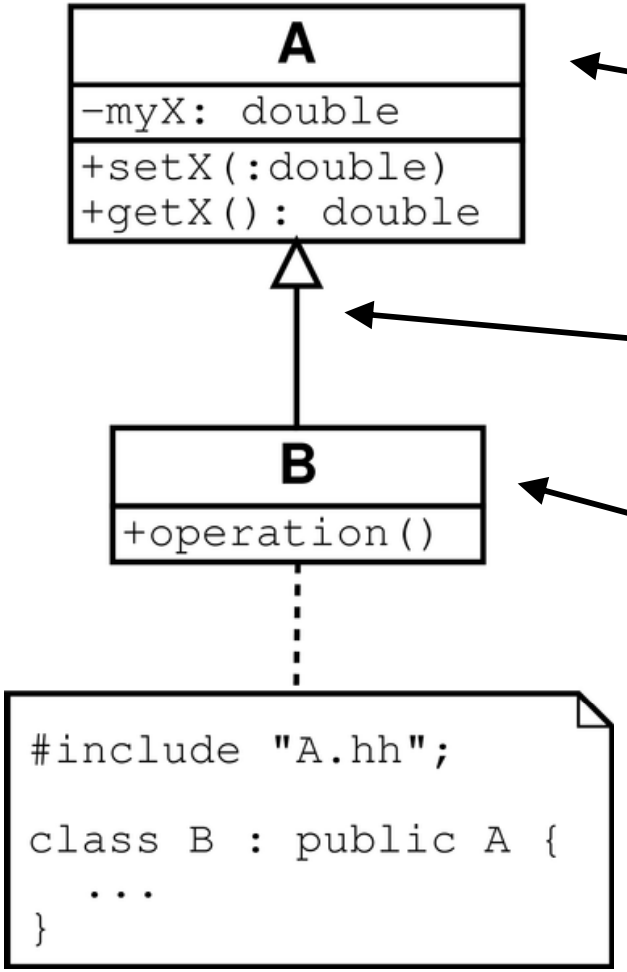
{abstract} is a tagged value that indicates that the class is abstract. The name of an abstract class should be italicized



Generalization

- A sub-class inherits from its super-class
  - Attributes
  - Operations
  - Relationships
- A sub-class may
  - Add attributes and operations
  - Add relationships
  - Refine (override) inherited operations
- A generalization relationship **may not** be used to model interface implementation.

Inheritance



Base class or super class

Arrow shows direction of dependency

Derived class or subclass

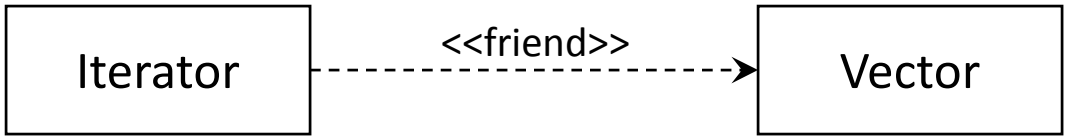
→ B inherits A's interface, behaviour and data members

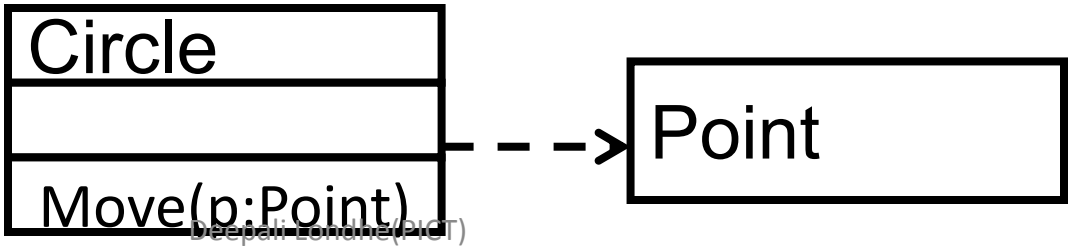
→ B can extend A, i.e. add new data members or member functions

→ B depends on A, A knows nothing about B

Dependency

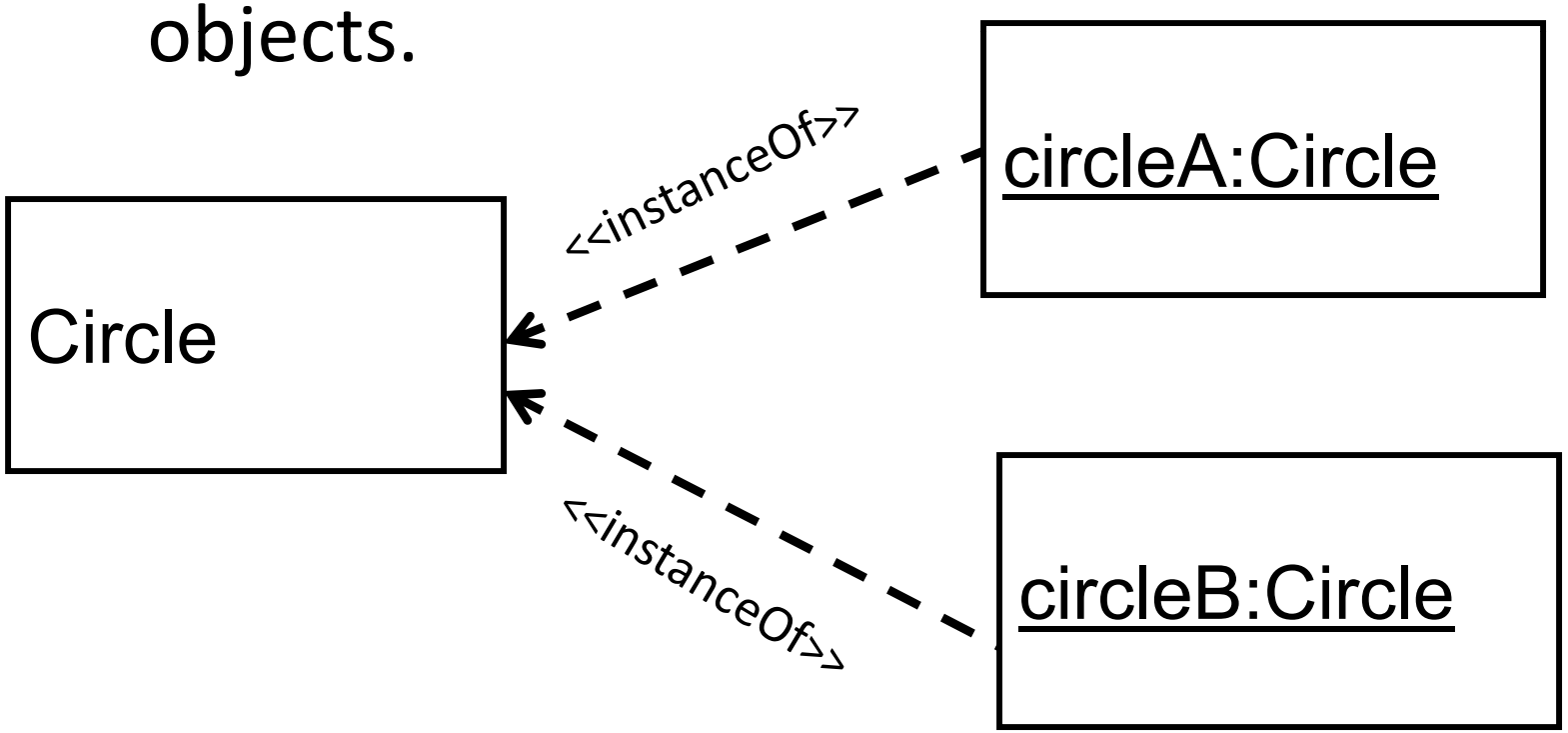
- A dependency indicates a semantic relation between two or more classes in which a change in one may force changes in the other although there is no explicit association between them.
- A stereotype may be used to denote the type of the dependency.





Dependency cont

- Dependency relationship can be used to show relationships between classes and objects.

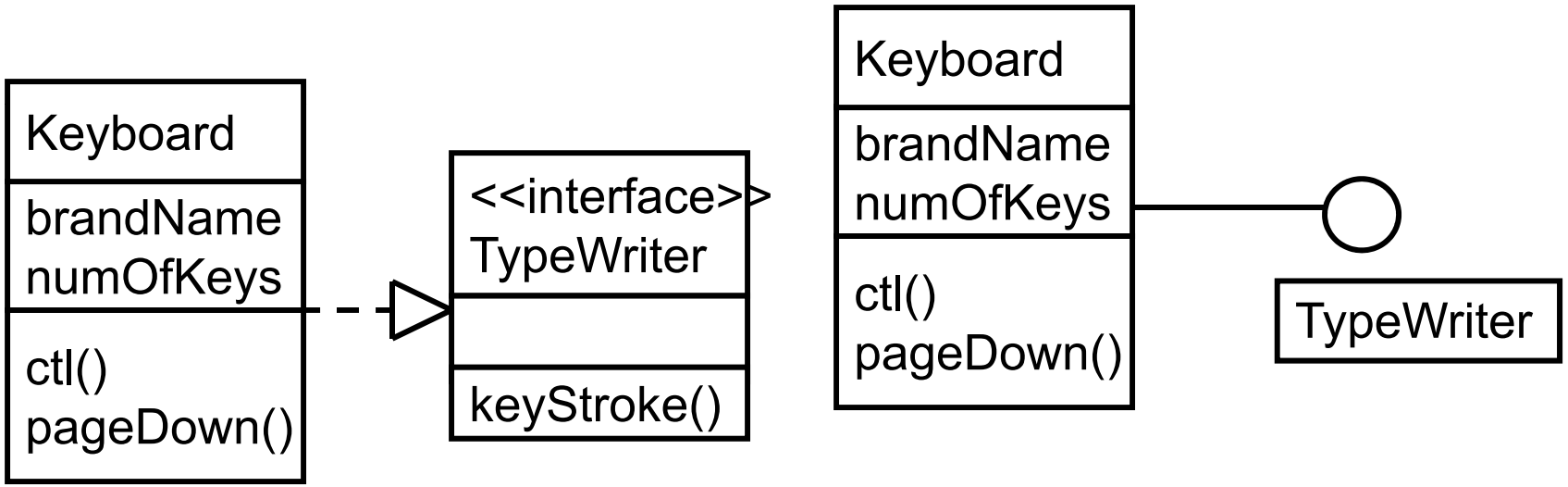


Realization

- A realization relationship indicates that one class implements a behavior specified by another class (an interface or protocol).
- An interface can be realized by many classes.
- A class may realize many interfaces.

Realization- Interface

- Interface is a set of operation the class carries out
- An interface does not have attributes. Only operations.

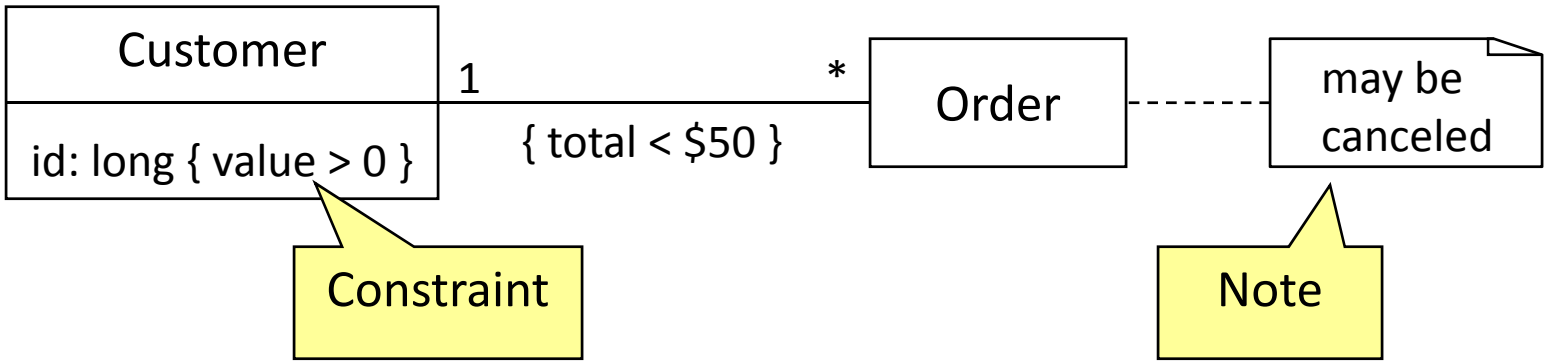


•The computer's keyboard is a reusable interface. Its keystroke operation has been reused from the typewriter. The placement of keys is the same as on a typewriter, but the main point is that the keystroke operation has been transferred from one system to another. Also on computer's keyboard you'll find a number of operations that you won't find on a typewriter (Ctrl, Alt, PageUp, PageDown...)

•The relationship between a class and an interface is called realization.

Constraint Rules and Notes

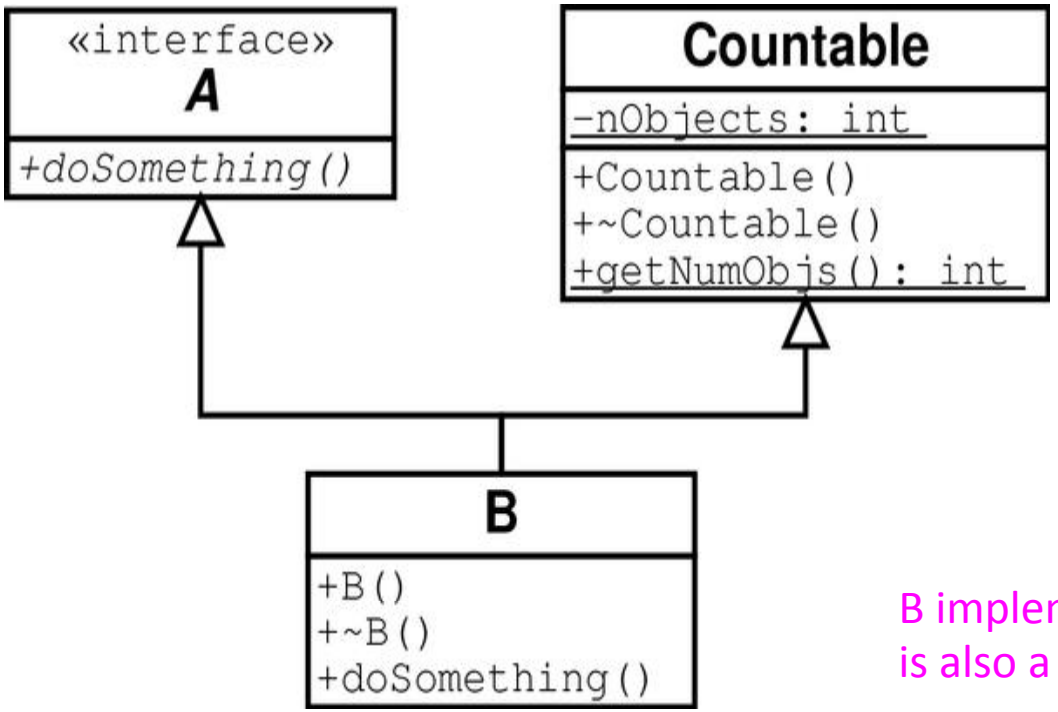
- **Constraints** and **notes** annotate among other things associations, attributes, operations and classes.
- Constraints are semantic restrictions noted as Boolean expressions.
  - UML offers many pre-defined constraints.



2.3 Associations Summary

- Can express different kinds of associations between classes/objects with UML
  - Association, aggregation, composition, inheritance
  - Friendship, parametric association
- Can go from simple sketches to more detailed design by adding *adornments*
  - *Name, roles, multiplicities*
  - *lifetime control*

2.3 Multiple Inheritance



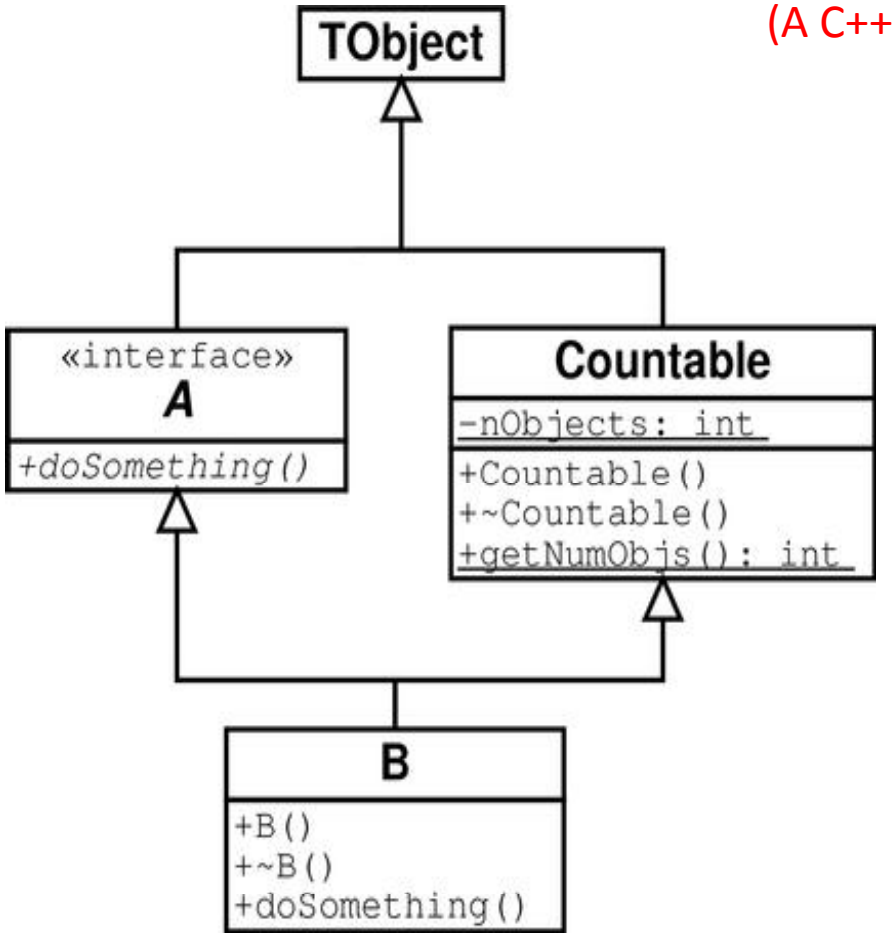
The derived class inherits interface, behaviour and data members of all its base classes

Extension and overriding works as before

B implements the interface A and is also a "countable" class

Countable also called a "Mixin class"

2.3 Deadly Diamond of Death



(A C++ feature)

Now the @*#! hits the %&$?

Data members of TObject are inherited twice in B, which ones are valid?

Fortunately, there is a solution to this problem:

→ virtual inheritance in C++: only one copy of a multiply inherited structure will be created

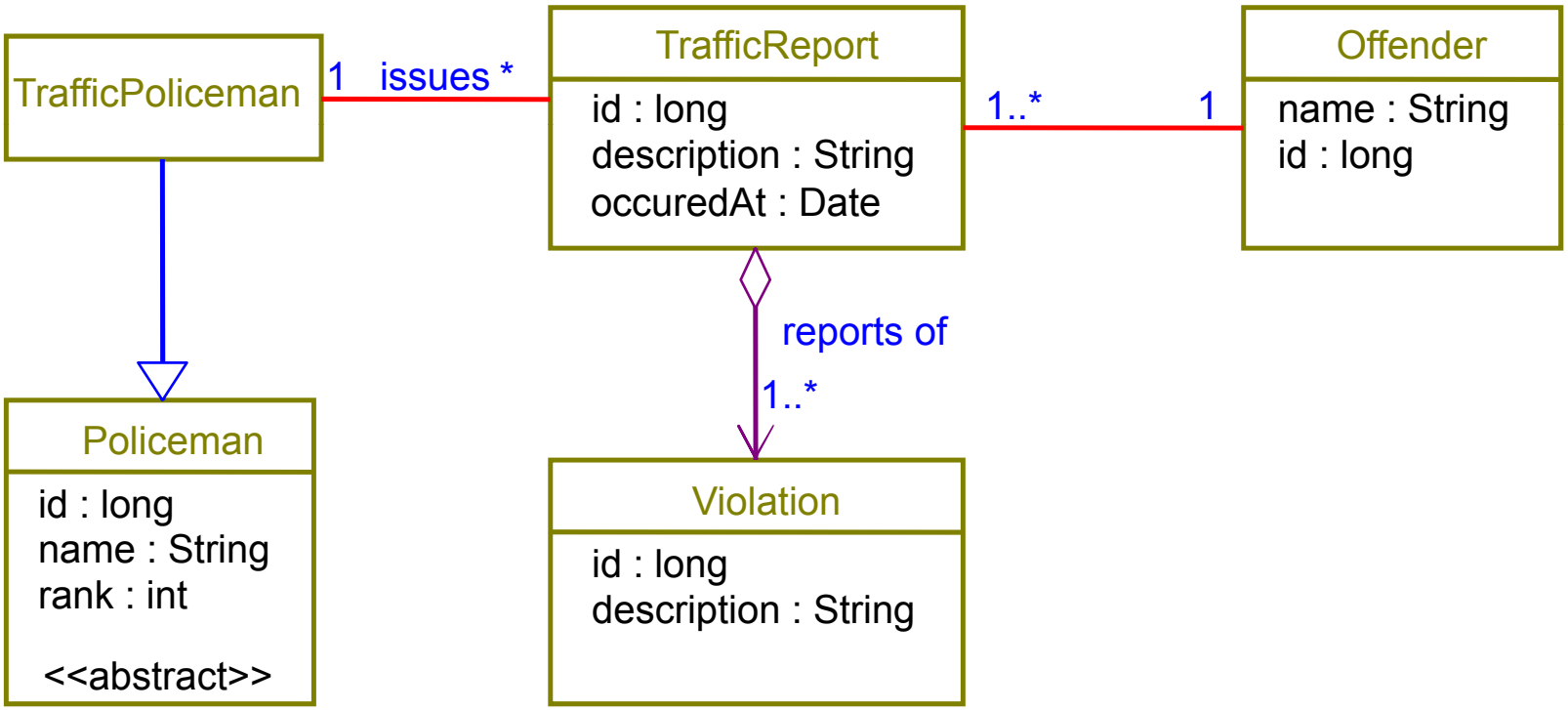
2.4 Static and Dynamic Design

- Static design describes code structure and object relations
  - Class relations
  - Objects at a given time
- Dynamic design shows communication between objects
  - Similarity to class relations
  - can follow sequences of events

Class Diagram

- Show static relations between classes
  - we have seen them already
  - interfaces, data members
  - associations
- Subdivide into diagrams for specific purpose
  - showing all classes usually too much
  - ok to show only relevant class members
  - set of all diagrams should describe system

TVRS Example

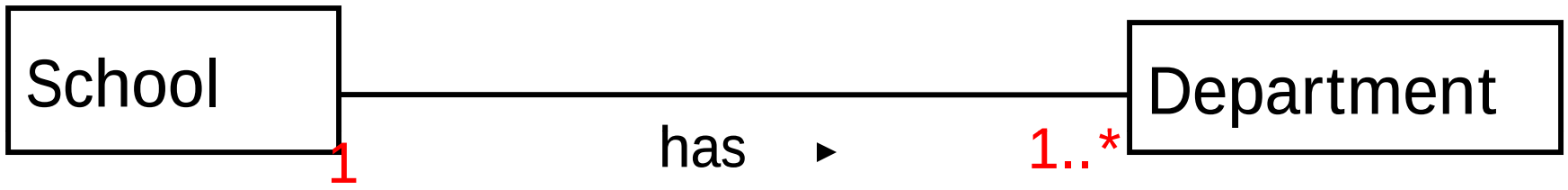


Class Diagram - Example

- Draw a class diagram for a information modeling system for a school.
  - School has one or more Departments.
  - Department offers one or more Subjects.
  - A particular subject will be offered by only one department.
  - Department has instructors and instructors can work for one or more departments.
  - Student can enrol in upto 5 subjects in a School.
  - Instructors can teach upto 3 subjects.
  - The same subject can be taught by different instructors.
  - Students can be enrolled in more than one school.

Class Diagram - Example

- School has one or more Departments.



- Department offers one or more Subjects.
- A particular subject will be offered by only one department.

Class Diagram - Example

- Department has Instructors and instructors can work for one or more departments.

- Student can enrol in upto 5 Subjects.

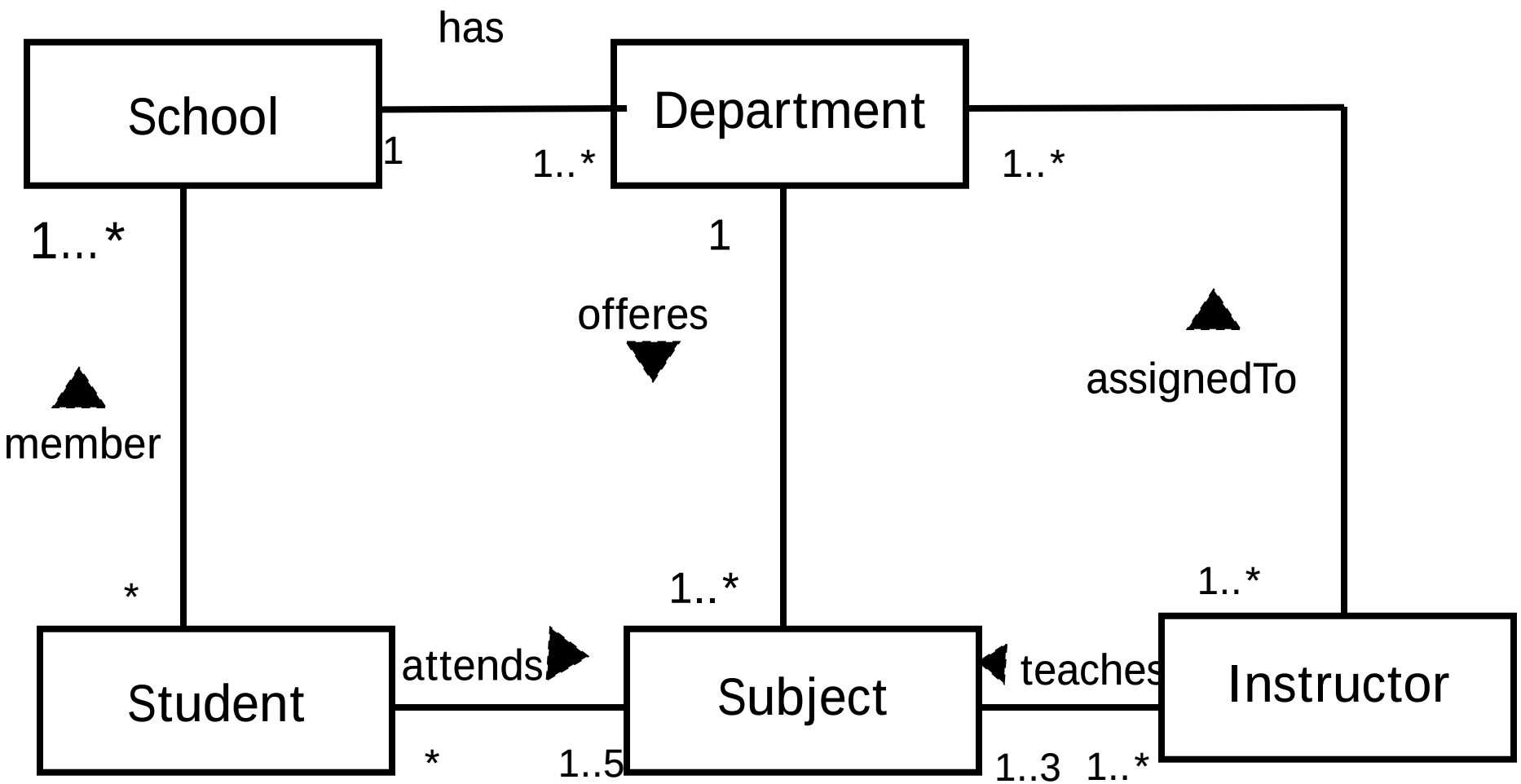
Class Diagram - Example

- Instructors can teach up to 3 subjects.
- The same subject can be taught by different instructors.

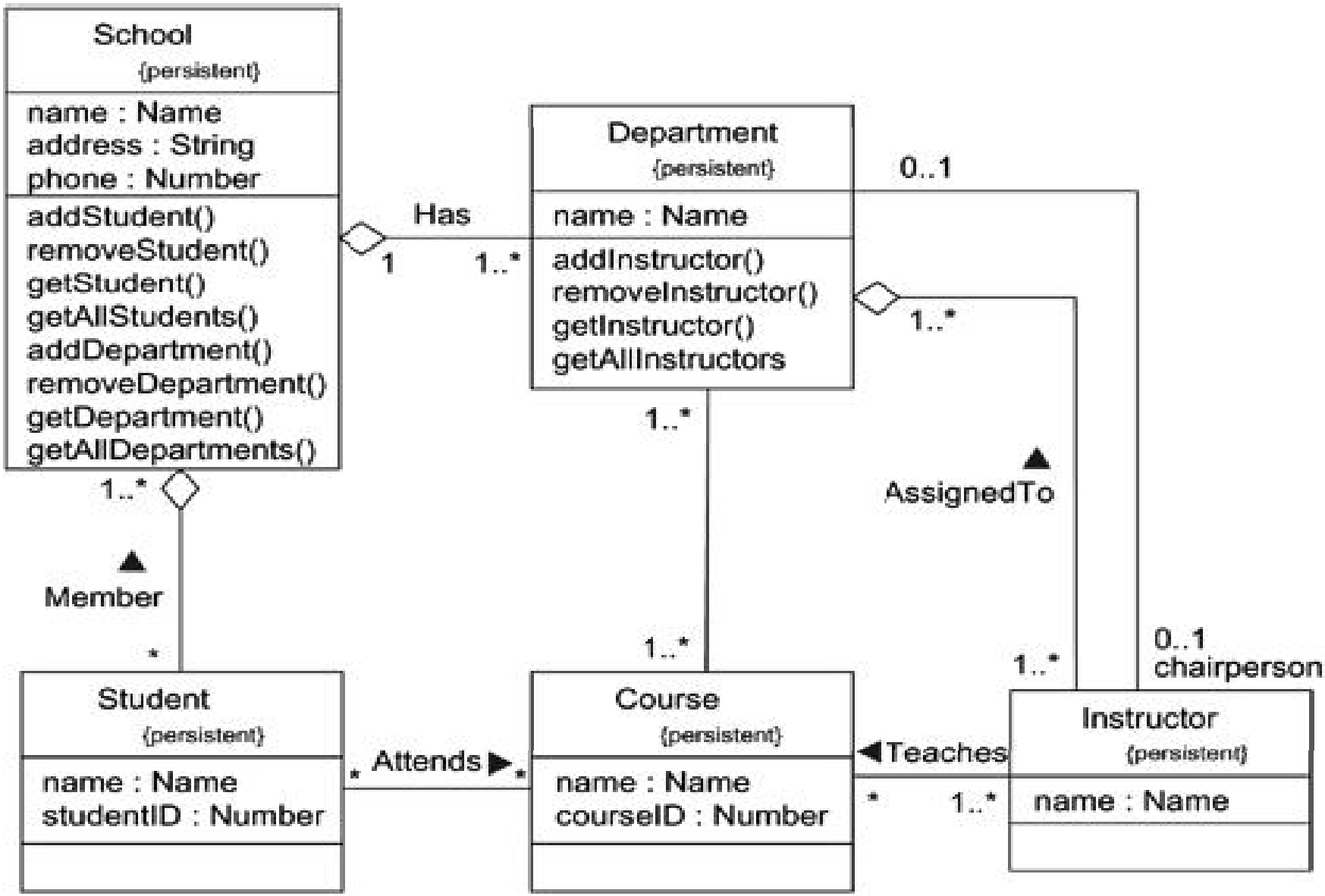
Class Diagram - Example

- Students can be enrolled in more than one school.

Class Diagram Example

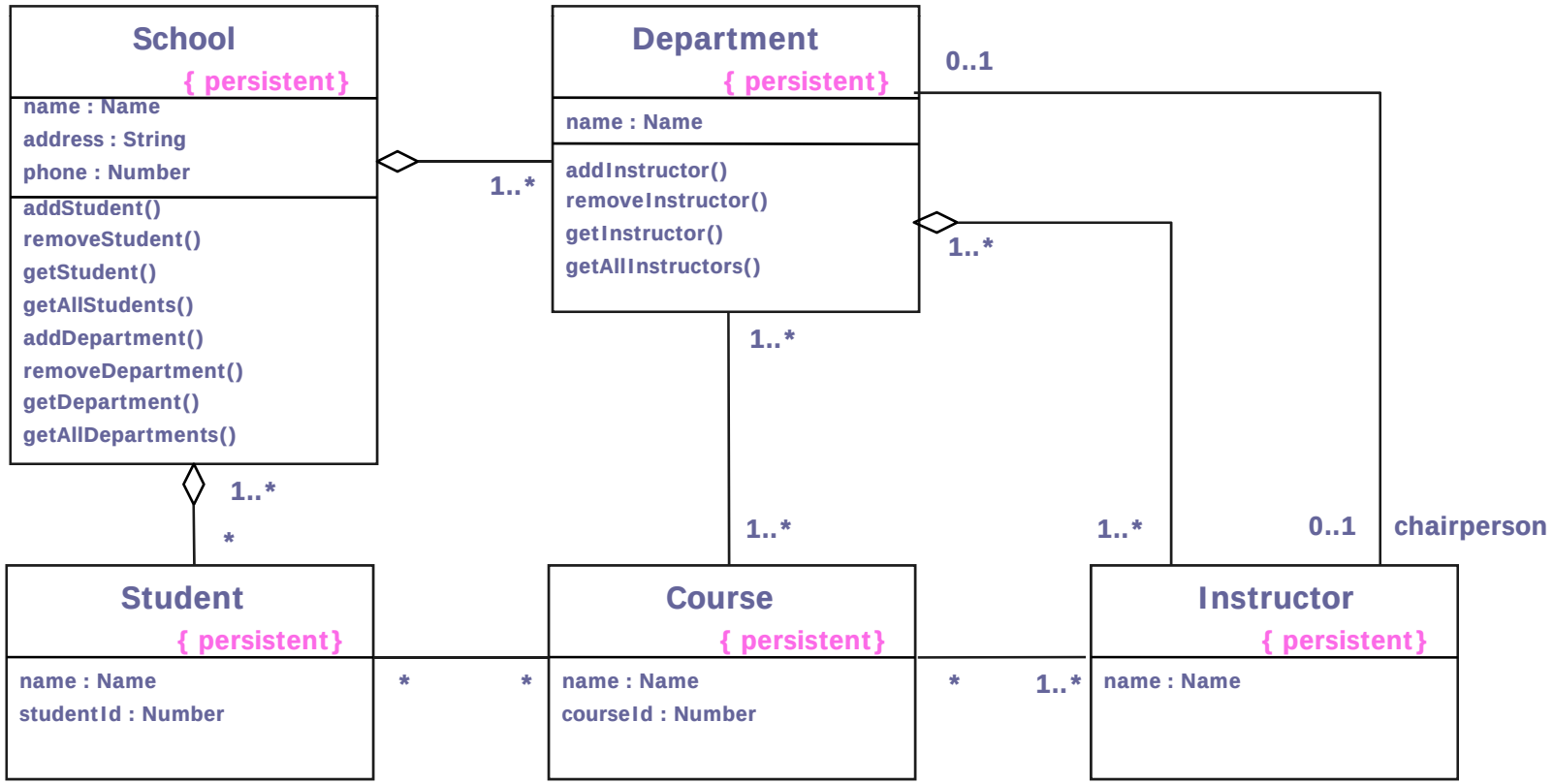


Detailed Class Diagram



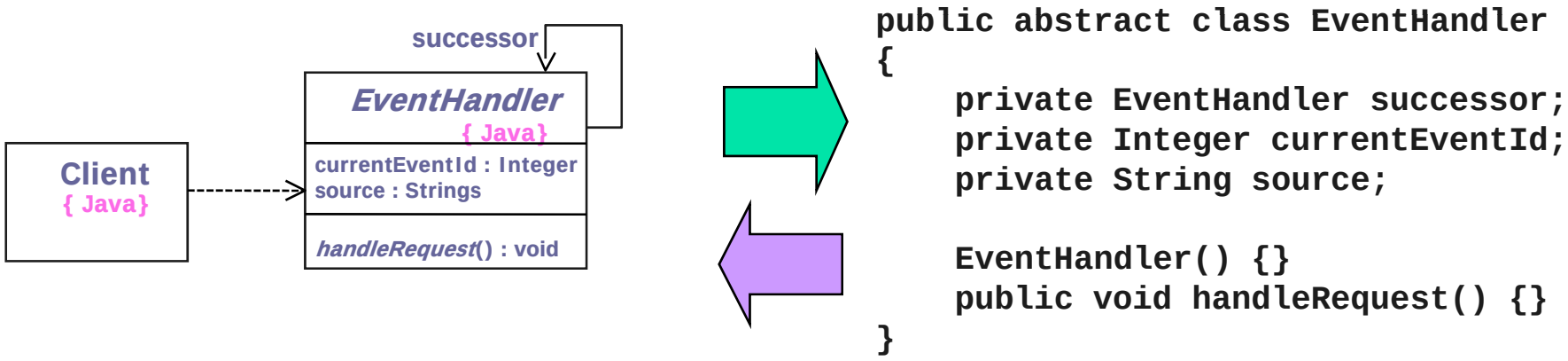
Modeling a Logical Database

- Class diagrams to provide more semantics
- From a general class diagram, first identify classes whose state must be persistent (e.g. after you turn off the computer, the data survives, although the program doesn't).
- Create a class diagram using standard tagged value, (e.g. {persistent}).
- Include attributes and associations.
- Use tools, if available, to transform logical design (e.g., tables and attributes) into physical design (e.g., layout of data on disk and indexing mechanisms for fast access to the data).



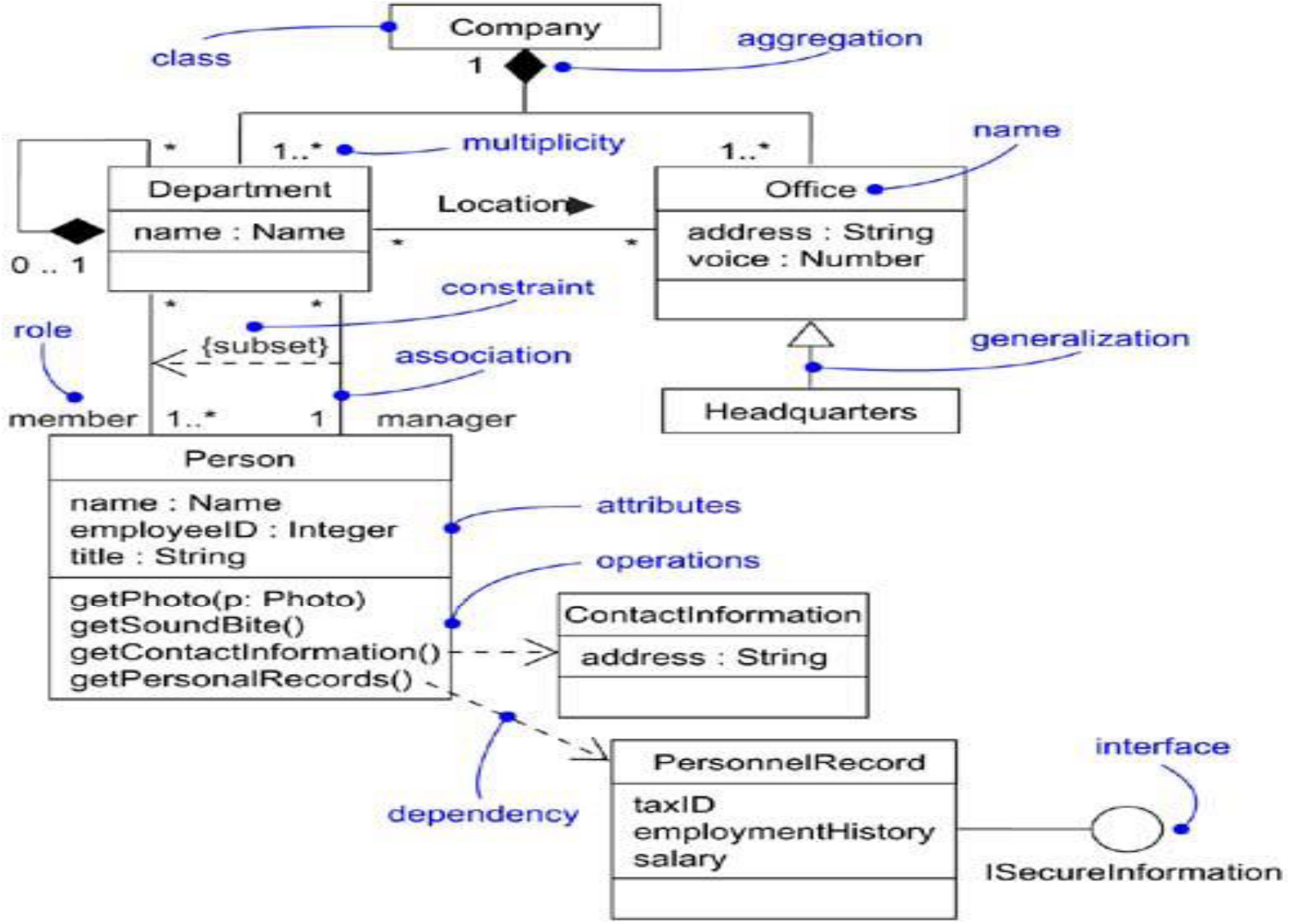
Forward / Reverse Engineering

- translate a collaboration into a logical database schema/operations
- transform a model into code through a mapping to an implementation language.
- Steps
  - Selectively use UML to match language semantics (e.g. mapping multiple inheritance in a collaboration diagram into a programming language with only single inheritance mechanism).
  - Use **tagged values** to identify language..



- translate a logical database schema/operations into a collaboration
- transform code into a model through mapping from a specific implementation language.

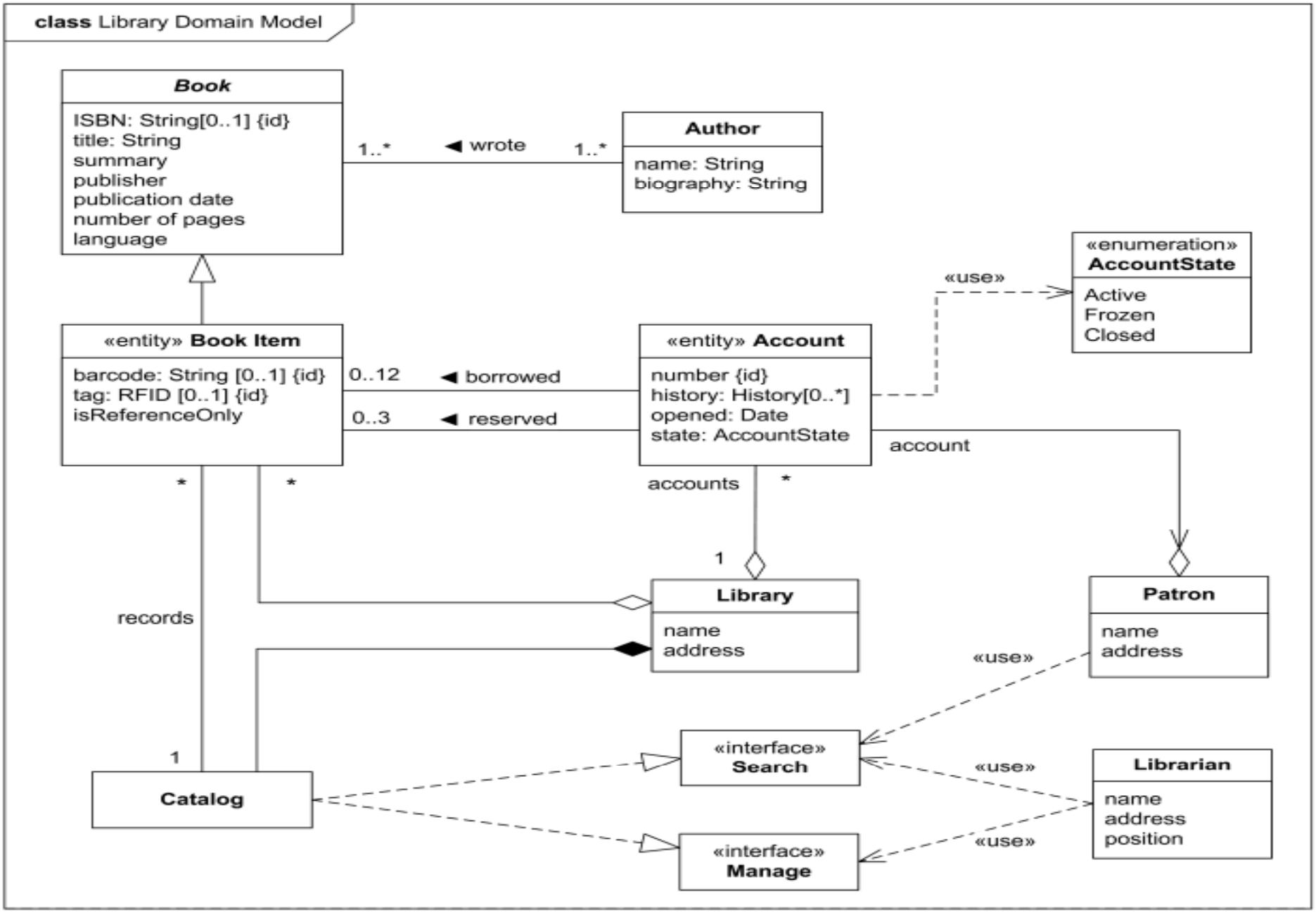
Company scenario



Library Management System (LMS)

- Each physical library item - book, tape cassette, CD, DVD, etc. could have its own item number. To support it, the items may be barcoded. The purpose of barcoding is to provide a unique and scannable identifier that links the barcoded physical item to the electronic record in the catalog. Barcode must be physically attached to the item, and barcode number is entered into the corresponding field in the electronic item record.
- Barcodes on library items could be replaced by RFID tags. The RFID tag can contain item's identifier, title, material type, etc. It is read by an RFID reader, without the need to open a book cover or CD/DVD case to scan it with barcode reader.
- Library has some rules on what could be borrowed and what is for reference only. Rules are also defined on how many books could be borrowed by patrons and how many could be reserved.

LMS

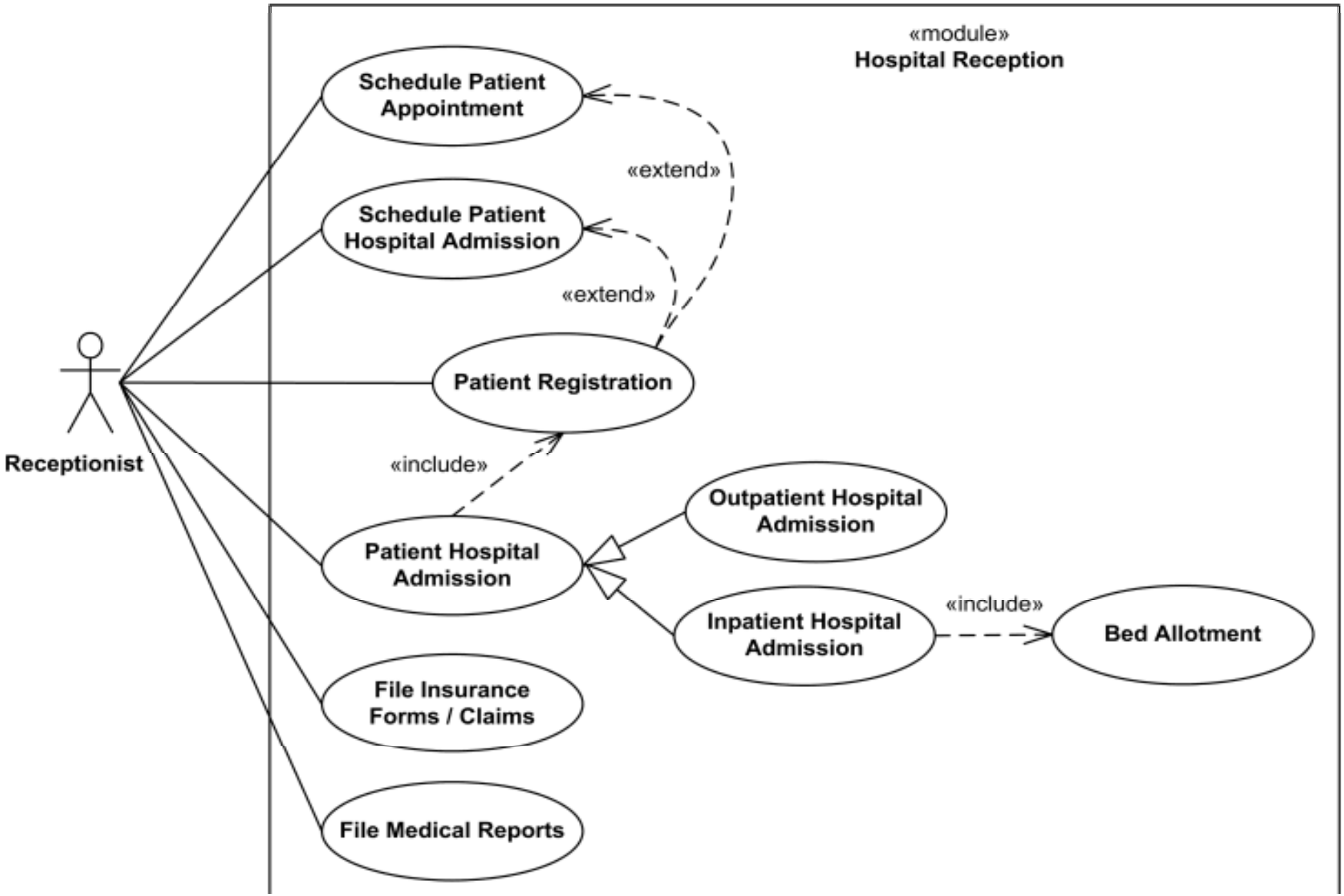

class Library Domain Model
Book
ISBN: String[0..1] {id}
title: String
summary
publisher
publication date
number of pages
language
1..*
wrote
1..*
Author
name: String
biography: String
«enumeration»
AccountState
Active
Frozen
Closed
«use»
«entity» Book Item
barcode: String [0..1] {id}
tag: RFID [0..1] {id}
isReferenceOnly
0..12
borrowed
0..3
reserved
«entity» Account
number {id}
history: History[0..*]
opened: Date
state: AccountState
account
*
*
accounts
*
1
records
Library
name
address
Patron
name
address
«use»
1
«interface»
Search
«use»
Librarian
name
address
position
Catalog
«interface»
Manage
«use»

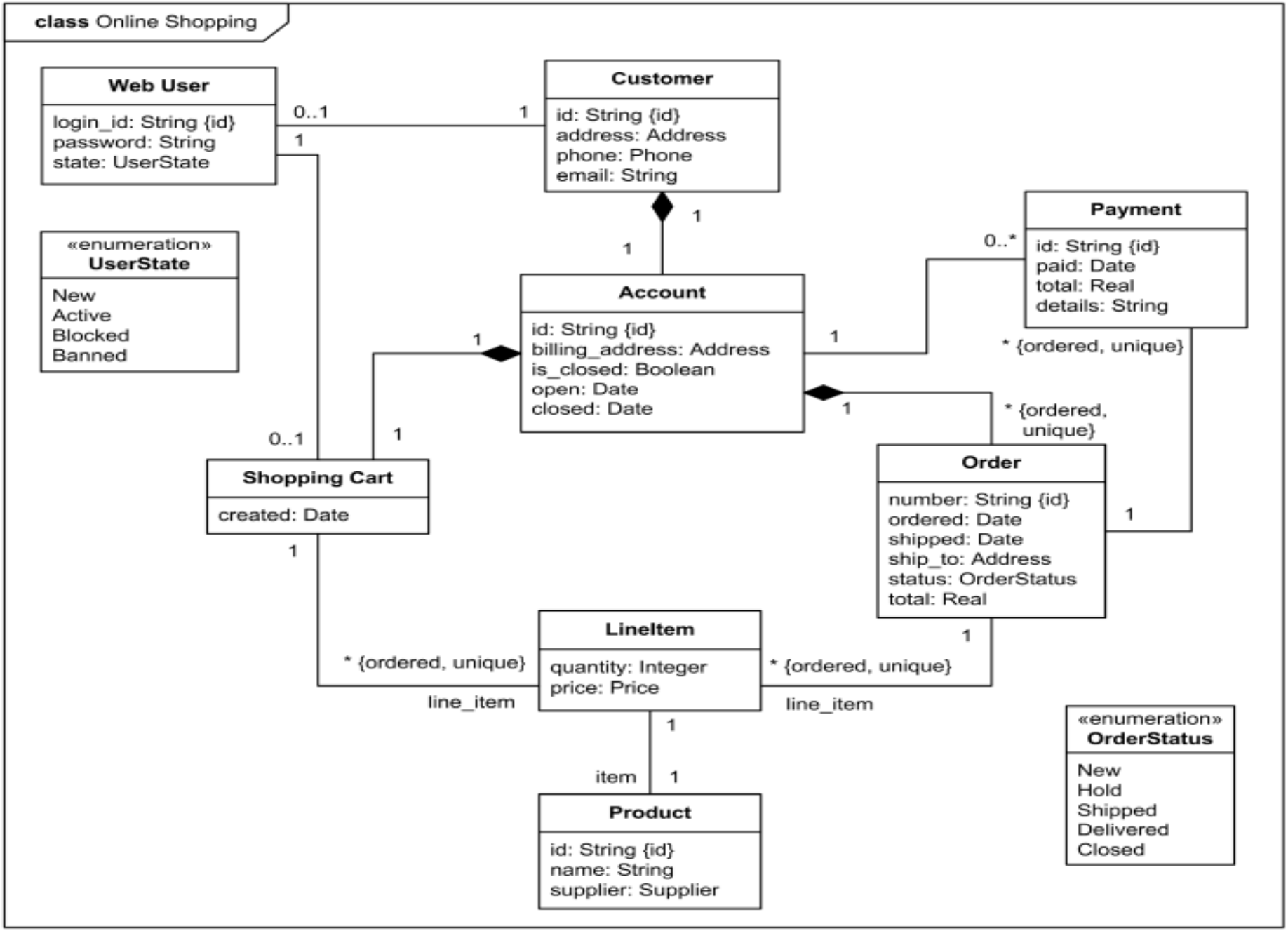
Example

Hospital Management System is a large system including several subsystems or modules providing variety of functions.

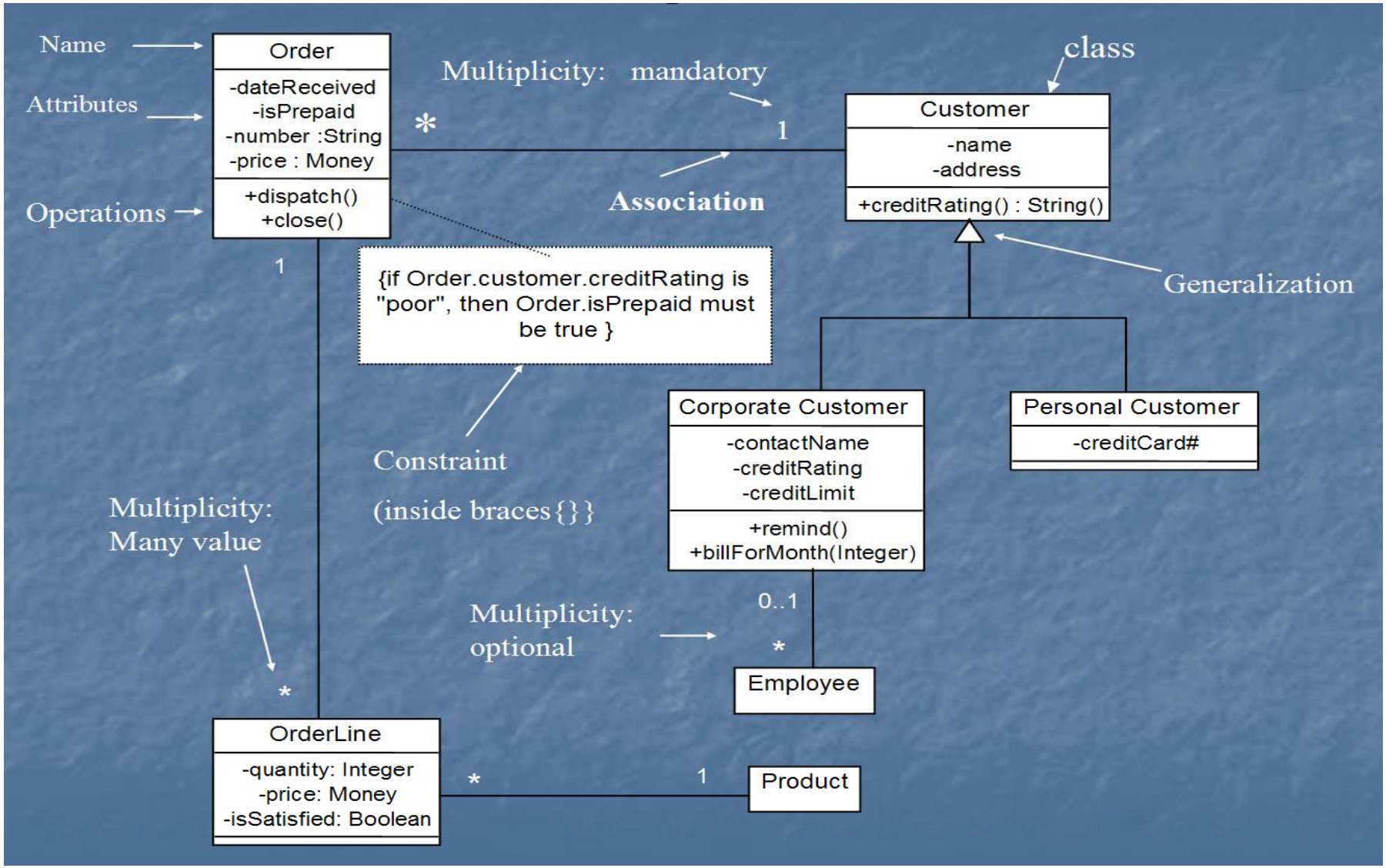
Hospital Reception subsystem or module supports some of the many job duties of hospital receptionist. Receptionist schedules patient's appointments and admission to the hospital, collects information from patient upon patient's arrival and/or by phone. For the patient that will stay in the hospital (inpatient) s/he should have a bed allotted in a ward. Receptionists might also receive patient's payments, record them in a database and provide receipts, file insurance claims and medical reports.

Use Case diagrams




class Online Shopping
Web User
login_id: String {id}
password: String
state: UserState
0..1
1
Customer
id: String {id}
address: Address
phone: Phone
email: String
1
1
«enumeration»
UserState
New
Active
Blocked
Banned
Payment
id: String {id}
paid: Date
total: Real
details: String
0..*
* {ordered, unique}
Account
id: String {id}
billing_address: Address
is_closed: Boolean
open: Date
closed: Date
1
1
1
1
* {ordered, unique}
0..1
1
Shopping Cart
created: Date
Order
number: String {id}
ordered: Date
shipped: Date
ship_to: Address
status: OrderStatus
total: Real
1
1
1
* {ordered, unique}
line_item
LineItem
quantity: Integer
price: Price
* {ordered, unique}
line_item
1
item
1
Product
id: String {id}
name: String
supplier: Supplier
«enumeration»
OrderStatus
New
Hold
Shipped
Delivered
Closed

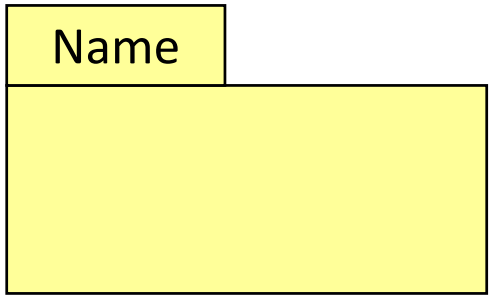
Class diagram



[from *UML Distilled Third Edition*]

UML Packages

- A package is a general purpose grouping mechanism.
  - Can be used to group any UML element (e.g. use case, actors, classes, components and other packages.
- Commonly used for specifying the logical distribution of classes.
- A package does not necessarily translate into a physical sub-system.

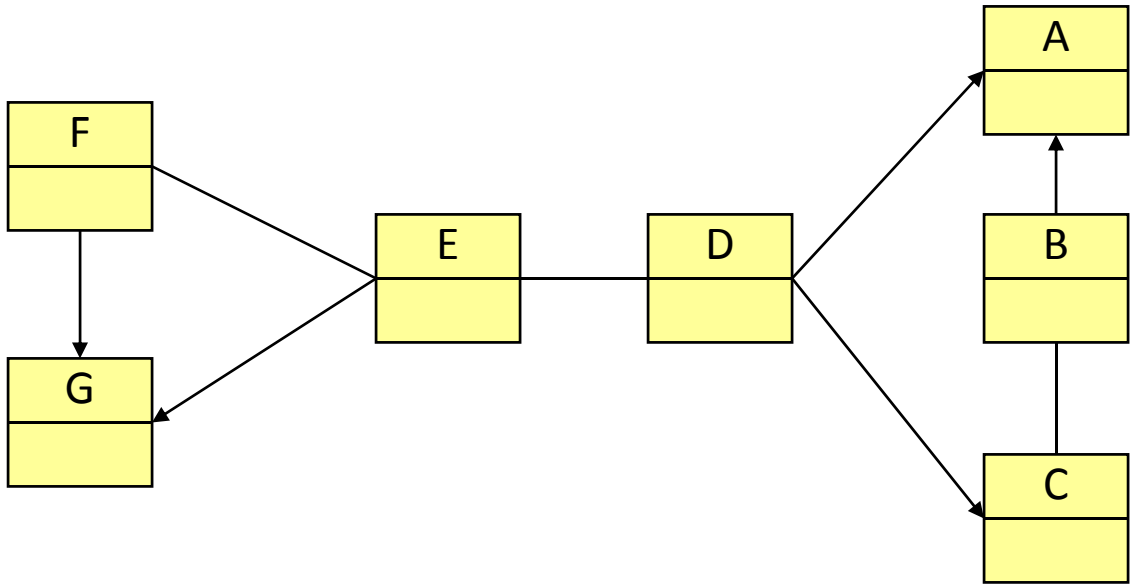


Logical Distribution of Classes

- Emphasize the logical structure of the system (High level view)
  - Higher level of abstraction over classes.
  - Aids in administration and coordination of the development process.
  - Contributes to the scalability of the system.
- Logical distribution of classes is inferred from the logical architecture of the system.

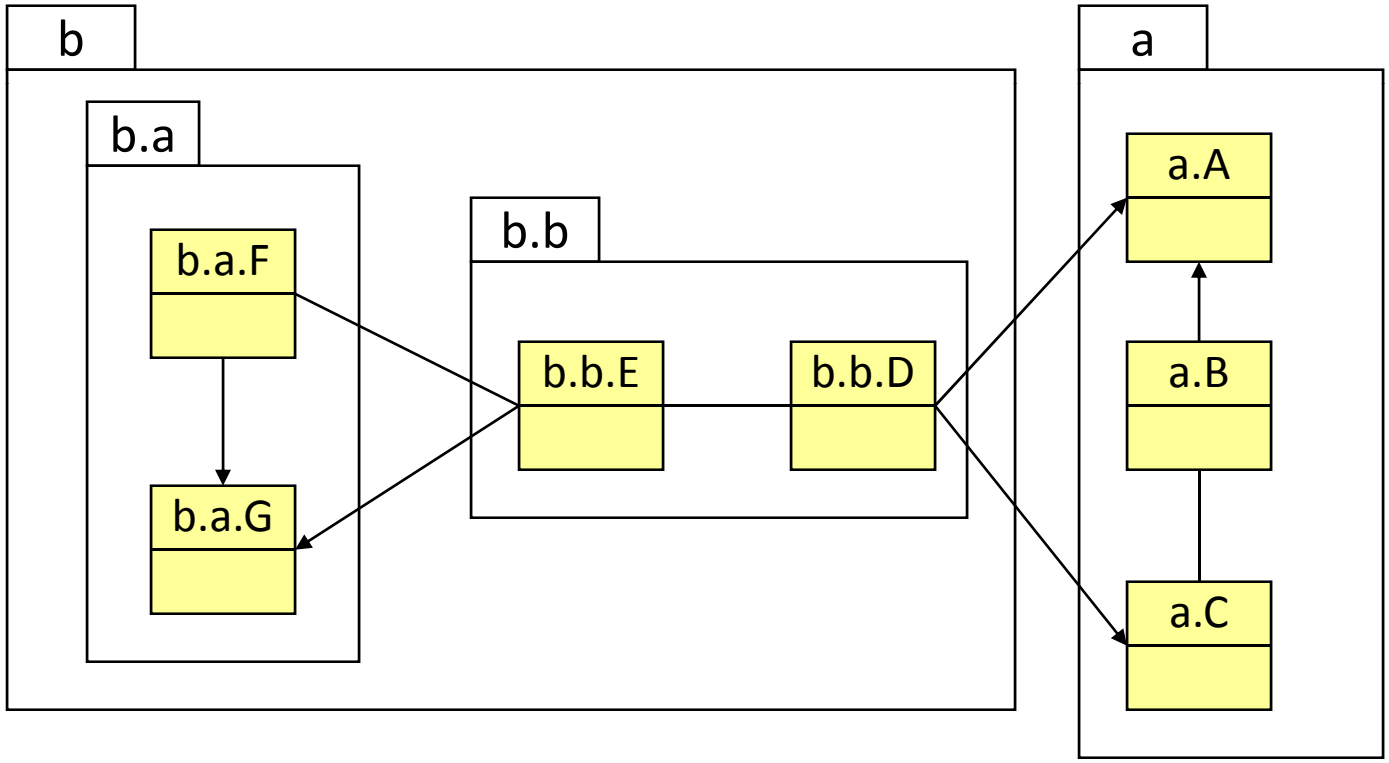
Packages and Class Diagrams (cont.)

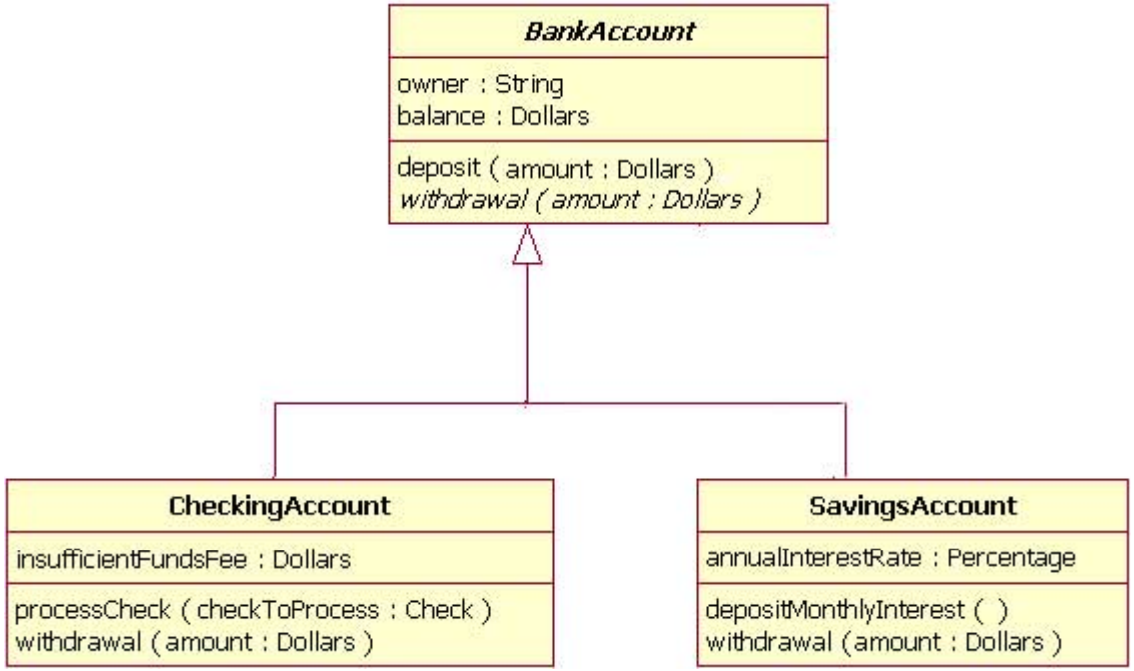
- Add package information to class diagrams



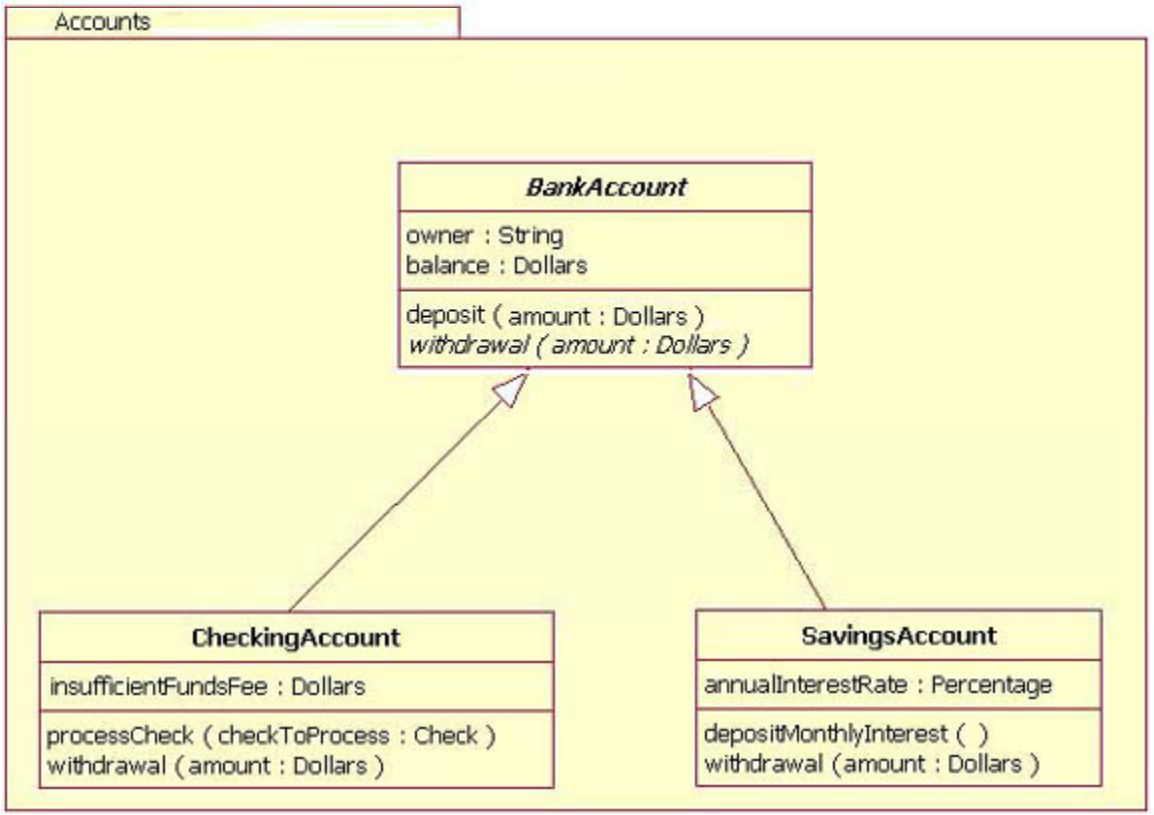
Packages and Class Diagrams (cont.)

- Add package information to class diagrams

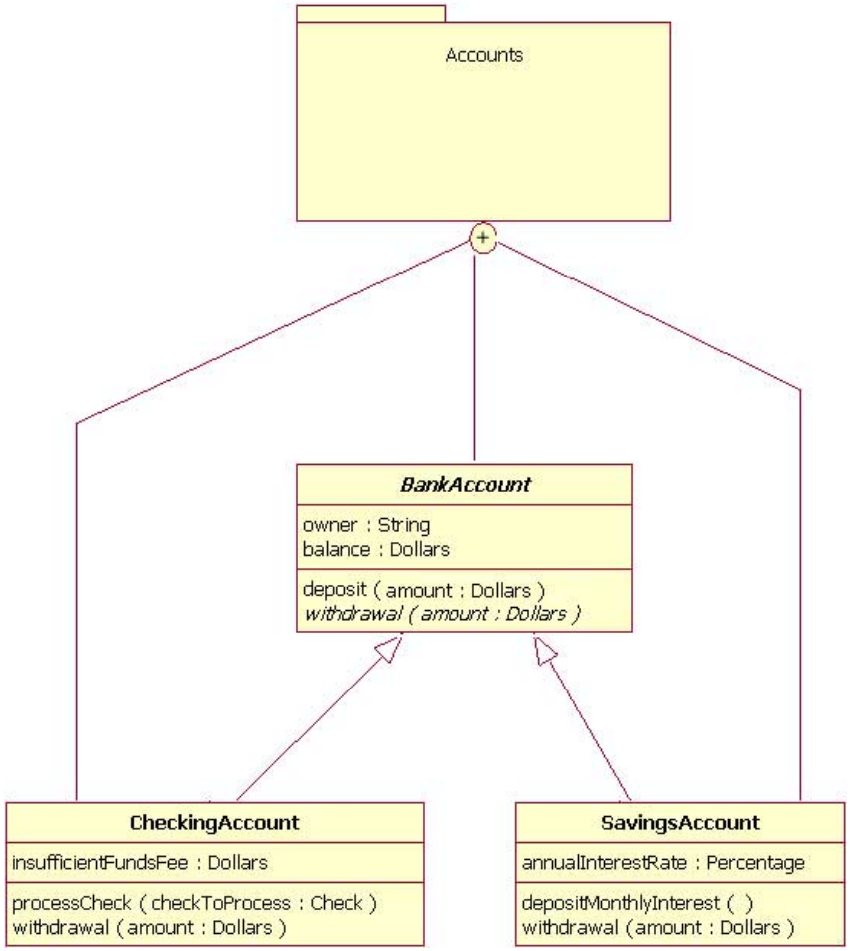



BankAccount
owner : String
balance : Dollars
deposit (amount : Dollars)
withdrawal (amount : Dollars)
CheckingAccount
insufficientFundsFee : Dollars
processCheck (checkToProcess : Check)
withdrawal (amount : Dollars)
SavingsAccount
annualInterestRate : Percentage
depositMonthlyInterest ( )
withdrawal (amount : Dollars)

members inside the package's rectangle boundaries



package element showing its membership via connected lines



Analysis Classes

- A technique for finding analysis classes which uses three different perspectives of the system:
  - The boundary between the system and its actors
  - The information the system uses
  - The control logic of the system

Boundary Classes

- Models the interaction between the system's surroundings and its inner workings
  - User interface classes
    - Concentrate on what information is presented to the user
    - Don't concentrate on user interface details
    - Example:
      - ReportDetailsForm
      - ConfirmationDialog
  - System / Device interface classes
    - Concentrate on what protocols must be defined. Don't concentrate on how the protocols are implemented

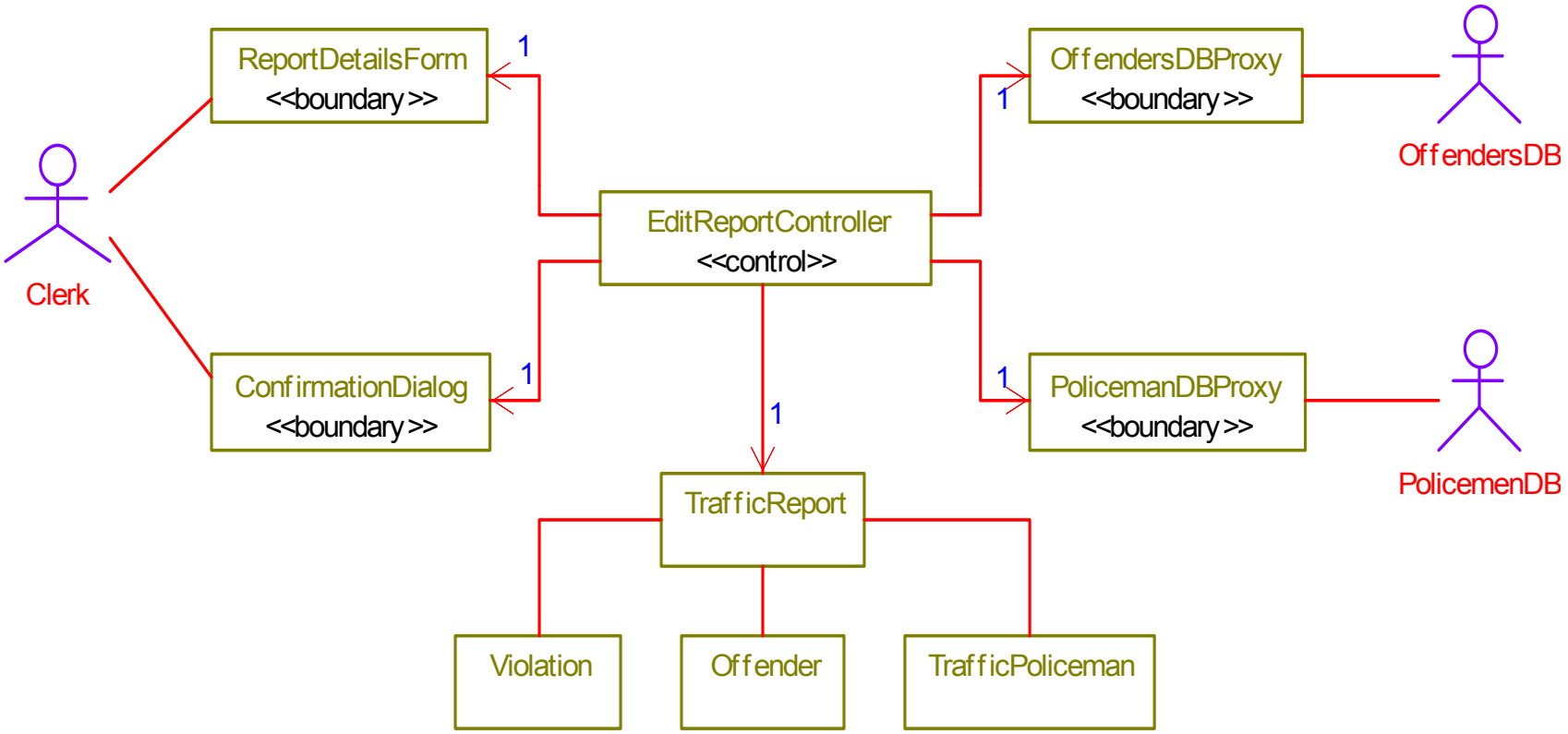
Entity Classes

- Models the key concepts of the system
- Usually models information that is persistent
- Contains the logic that solves the system problem
- Can be used in multiple behaviors
- Example: Violation, Report, Offender.

Control Classes

- Controls and coordinates the behavior of the system
- Delegates the work to other classes
  - A control class should tell other classes to do something and should never do anything except for directing
- Control classes decouple boundary and entity classes
- Example:
  - EditReportController
  - AddViolationController

TVRS Example



Identify conceptual classes from noun phrases

- Vision and Scope, Glossary and Use Cases are good for this type of linguistic analysis

However:

- Words may be ambiguous or synonymous
- Noun phrases may also be attributes or parameters rather than classes:
  - If it stores state information or it has multiple behaviors, then it's a class
  - If it's just a number or a string, then it's probably an attribute

From NPs to classes or attributes

Consider the following problem description, analyzed for Subjects, Verbs, Objects:

The ATM verifies whether the customer's card number and PIN are correct.
S C V O R O A O A
If it is, then the customer can check the account balance, deposit cash, and withdraw cash.
S R V O V A O V A O A
Checking the balance simply displays the account balance.
S M O A V O A
Depositing asks the customer to enter the amount, then updates the account balance.
S M V O R V O A V O A
Withdraw cash asks the customer for the amount to withdraw; if the account has enough cash,
S M O A O R O V A S V C O A
the account balance is updated. The ATM prints the customer's account balance on a receipt.
O A V S V C O O A

Analyze each **subject** and **object** as follows:

- Does it represent a person performing an action? Then it's an actor, '**R**'.
- Is it also a verb (such as 'deposit')? Then it may be a method, '**M**'.
- Is it a simple value, such as 'color' (string) or 'money' (number)? Then it is probably an attribute, '**A**'.
- Which NPs are unmarked? Make it '**C**' for class.

Verbs can also be classes, for example:

- Deposit is a class if it retains state information

Steps to create a Domain Model

- Identify candidate conceptual classes
- Draw them in a UML domain model
- Add associations necessary to record the relationships that must be retained
- Add attributes necessary for information to be preserved
- Use existing names for things, the vocabulary of the domain

Discovering the Domain Model with CRC cards

(See multimedia from *The Universal Machine* on CRC cards: *umwords*)

- Developed by Beck and Cunningham at Tektronix
  - See http://c2.com/doc/oopsla89/paper.html
  - This is the same Kent Beck that later wrote the book pioneering Extreme Programming (XP)
- CRC cards are now part of **XP**

Class Name	
Responsibilities	**Collaborators**
. .	. . .

Figure 15.1: Class-Responsibility-Collaborators (CRC) card

Low-tech

- Ordinary index cards
  - Each card represents a **class** of objects.
  - 3x5 is preferable to 4x6 at least early on – Why?
- Each card has three components
  - Name, Responsibilities, Collaborators

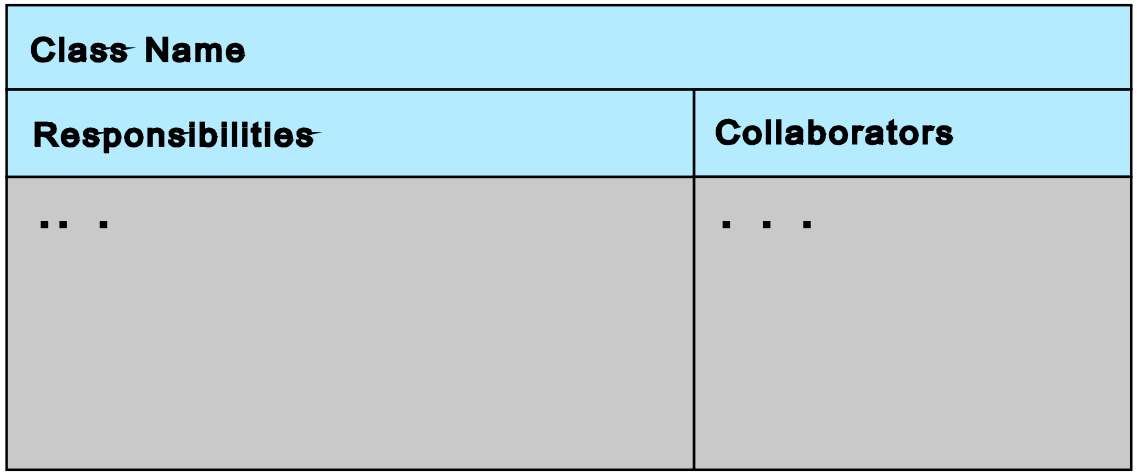


Figure 15.1: Class-Responsibility-Collaborators (CRC) card

Responsibilities

- Key idea: objects have responsibilities
  - As if they were simple agents (or actors in scenarios)
- Anthropomorphism of class responsibilities gets away from thinking about classes as just data holders
  - “Object think” focuses on their active behaviors
- Each object is responsible for specific actions
  - Client can expect predictable behaviors
- Responsibility also implies independence:
  - To trust an object to behave as expected is to rely upon its autonomy and modularity
  - Harder to trust objects easily caught up in dependencies caused by global variables and side effects.

Class names

- **Class Name** creates the vocabulary of our analysis
  - Use nouns as class names, think of them as simple agents
  - Verbs can also be made into nouns, if they are maintain state
  - E.g., “reads card” suggests **CardReader**, managing bank cards
- Use pronounceable names:
  - If you cannot read aloud, it is not a good name
- Use capitalization to initialize Class names and demarcate multi-word names
  - E.g., CardReader rather than CARDREADER or card_reader
  - Why do most OO developers prefer this convention?
- Avoid obscure, ambiguous abbreviations
  - E.g., is TermProcess something that terminates or something that runs on a terminal?
- Try *not* to use digits within a name, such as CardReader2
  - Better for instances than classes of objects

Responsibilities section

- Describes a class's **behaviors**
  - Describe *what* is to be done, not *how*!
- Use short verb phrases
  - E.g.: "reads card" or "look up words"
- How do constraints of index cards guide class analysis?
  - A good measure of appropriate complexity
  - If you cannot fit enough tasks on a card, maybe you need to divide tasks between classes, on different cards?

Collaborators

- Lists important **suppliers** and possibly clients of a class
- Why are classes that supply services more important here?
  - Suppliers are necessary for the description of responsibilities
- As you write down responsibilities for a class, add any suppliers needed for them
  - For example “read dictionary” obviously implies that a “dictionary” as a collaborator
- Developing CRC cards is first a process of discovering classes and their responsibilities
  - People naturally perceive the world as categories of objects
  - In object-oriented analysis, one discovers new categories relevant to a problem domain

CRC card simulations

- Designing for responsibility involves **simulation**
  - Objects model a world interacting behaviors
  - After developing a set of CRC cards, run **simulations**
  - *AKA* **structured walkthrough scenarios** --
  - Play “what if” to simulate scenarios that illustrate use of a system
  - Let each person be responsible for simulating one or more classes
- “Execute” a scenario of classes performing responsibilities:
  - Start a simulation with the construction of an object of a class, then perform one of its responsibilities (a behavior)
  - A responsibility may pass control to a collaborator -- another class
  - Simulation becomes visible as a sharing of responsibilities
  - You may discover missing or incompletely described responsibilities
  - See football example in multimedia

Object Diagram

- Object Diagram shows the relationship between objects.

- Unlike classes objects have a state.

Object Diagrams

Structural Diagrams

- Class;

  Object
- Component
- Deployment
- **Composite Structure**
- Package

Instances & Object Diagrams

- "instance" and "object" are largely synonymous; used interchangeably.

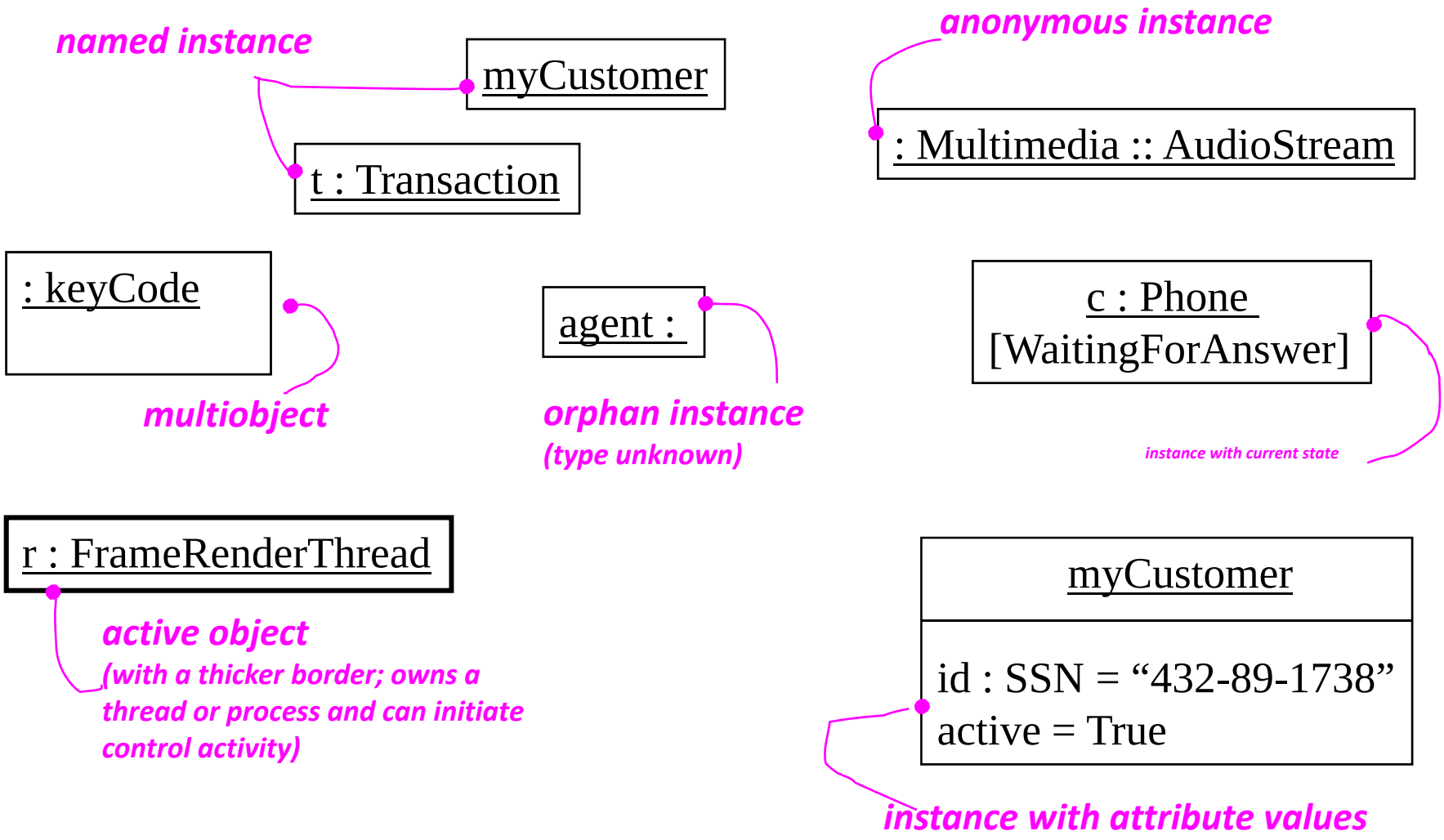
- difference:
  - instances of a **class** are called objects or instances; but
  - instances of other abstractions (components, nodes, use cases, and associations) are not called objects but only instances.

What is an instance of an association called?

Object Diagrams

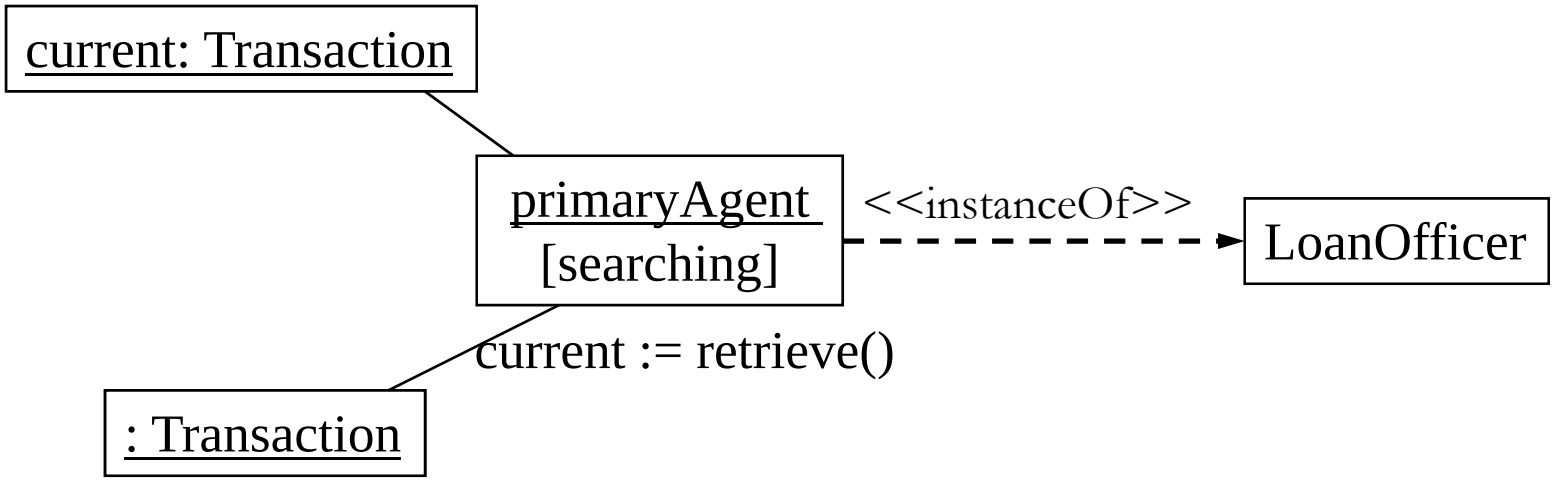
- very useful in debugging process.
  - walk through a scenario (e.g., according to use case flows).
  - Identify the set of objects that collaborate in that scenario (e.g., from use case flows).
  - Expose these object's states, attribute values and links among these objects.

Instances & Objects - Visual Representation



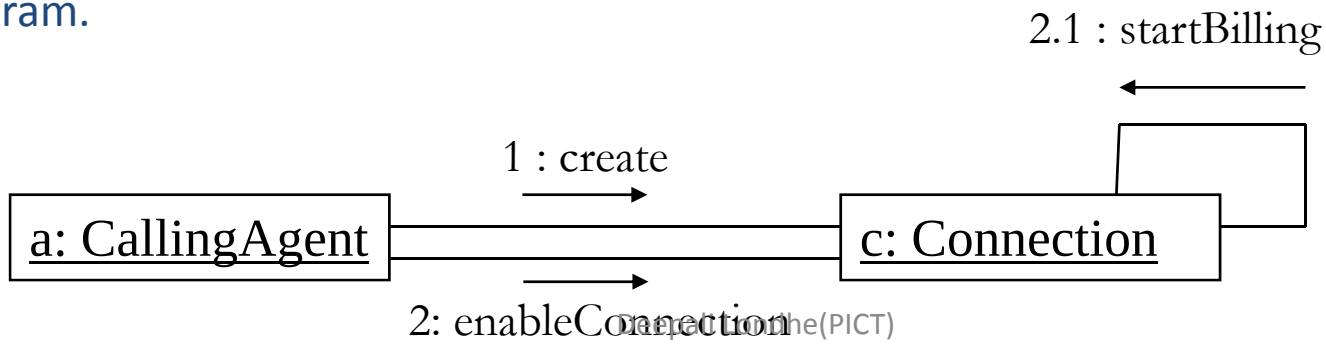
Instances & Objects - Modeling Concrete Instances

- Expose the stereotypes, tagged values, and attributes.
- Show these instances and their relationships in an object diagram.

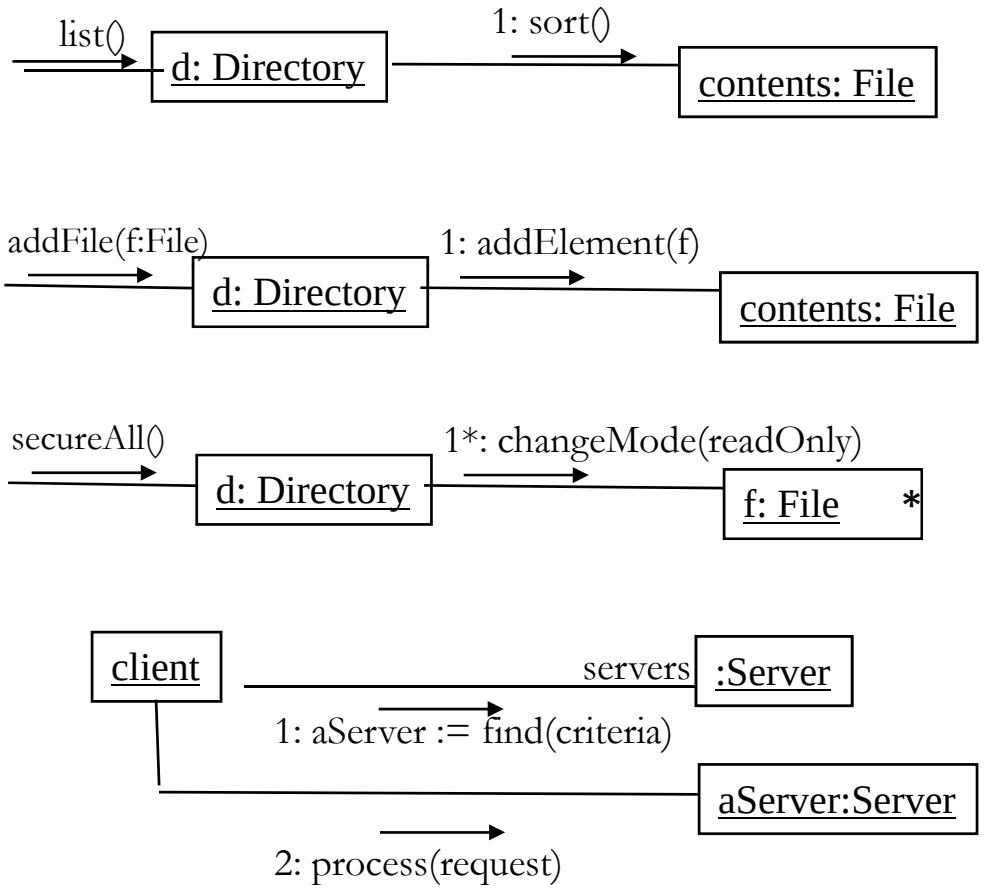


Instances & Objects - Modeling Prototypical Instances

- Show these instances and their relationships in an interaction diagram or an activity diagram.



Instances & Objects – More Examples



call ::= label [guard] ["*"] [return-val-list ":="] msg-name "(" arg-list ")"