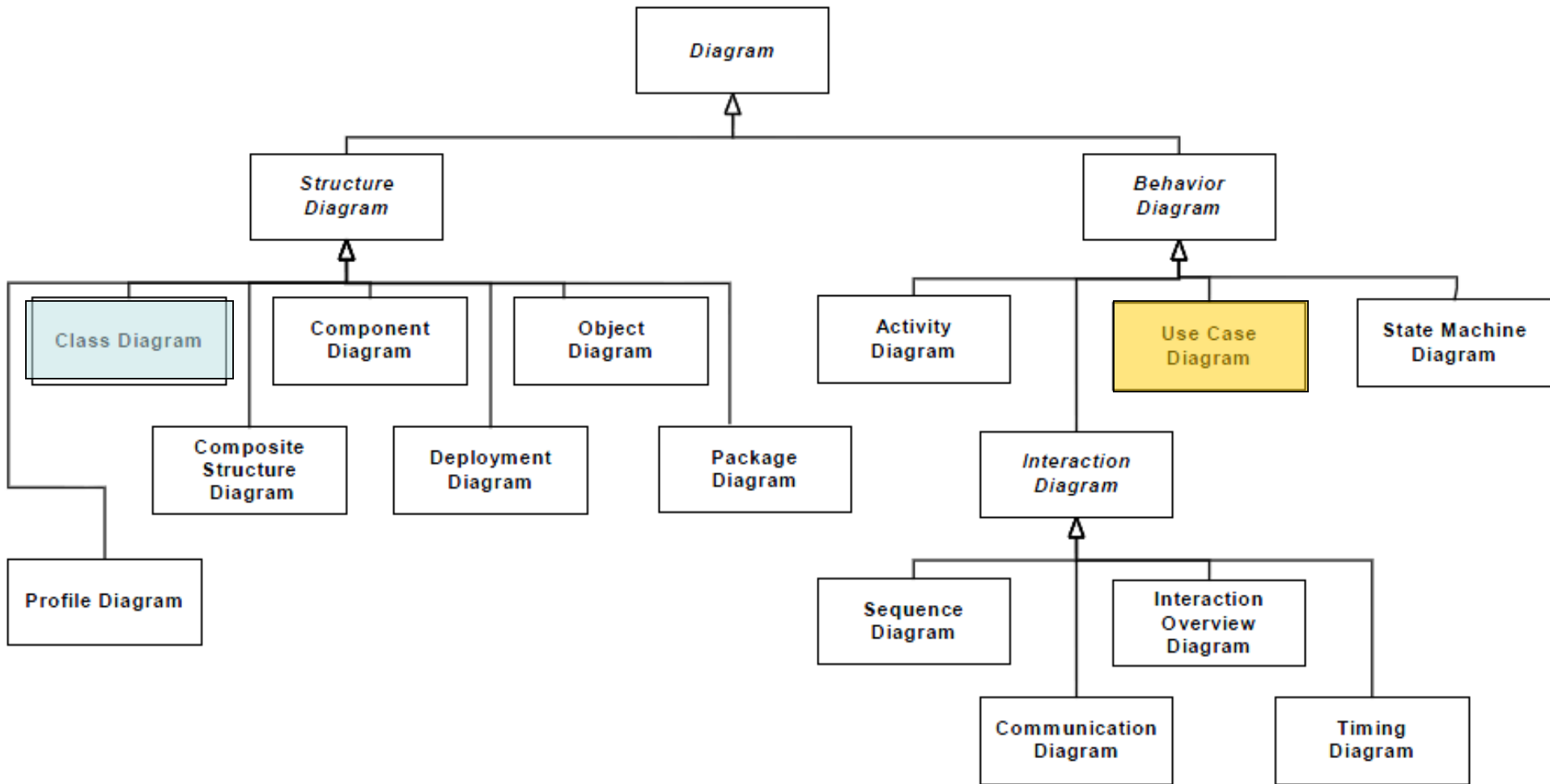


Object-Oriented Technology and UML Class Diagram

Diagrams of UML2.X



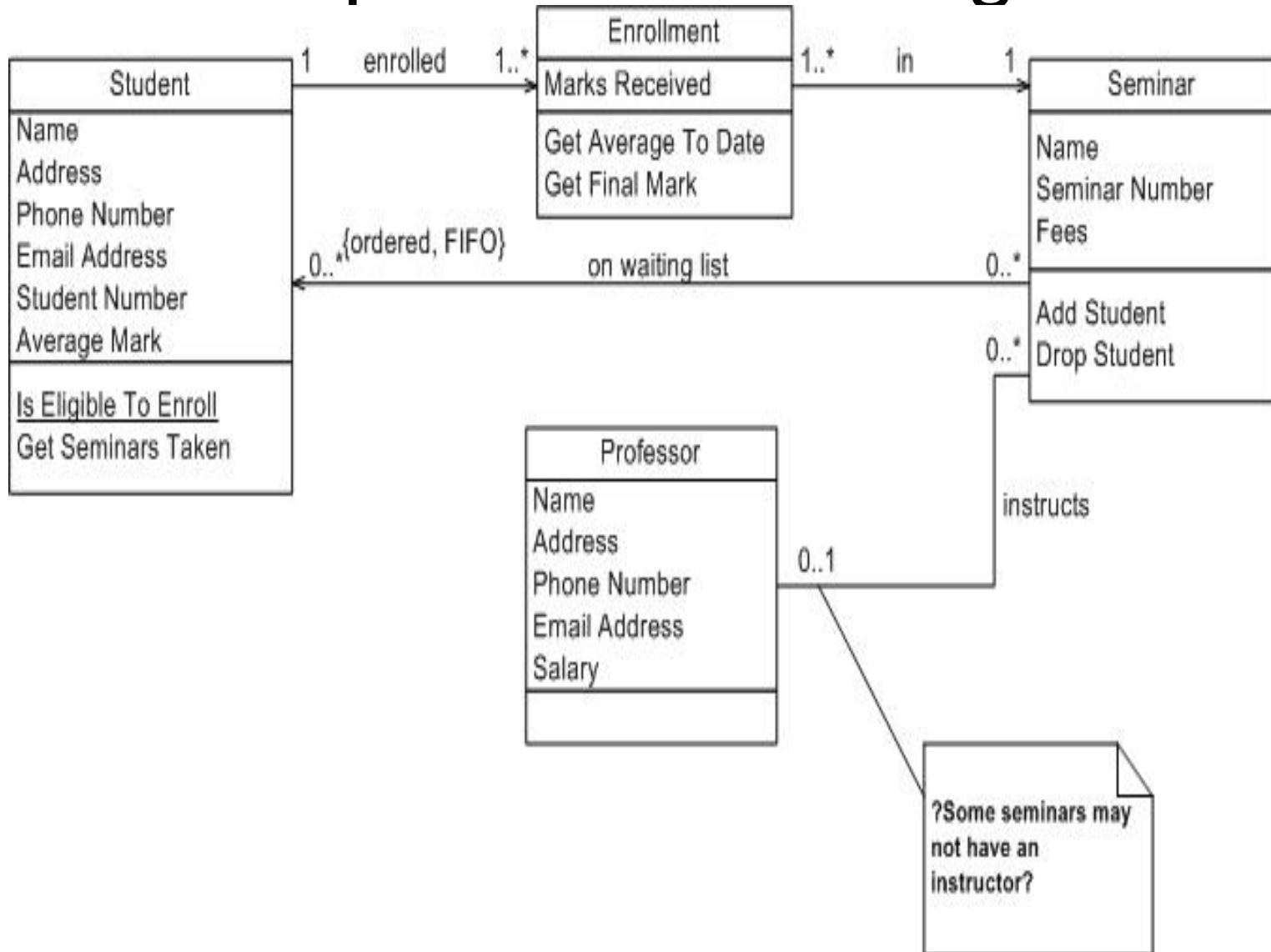
Topics

- Basic concepts
- Representation
- Reading Method
- Modeling

The Important Points

- Class
- Relationships between classes
- Class diagram
- Class diagram modeling
- Object oriented design principles and attention points

Example of Class Diagram



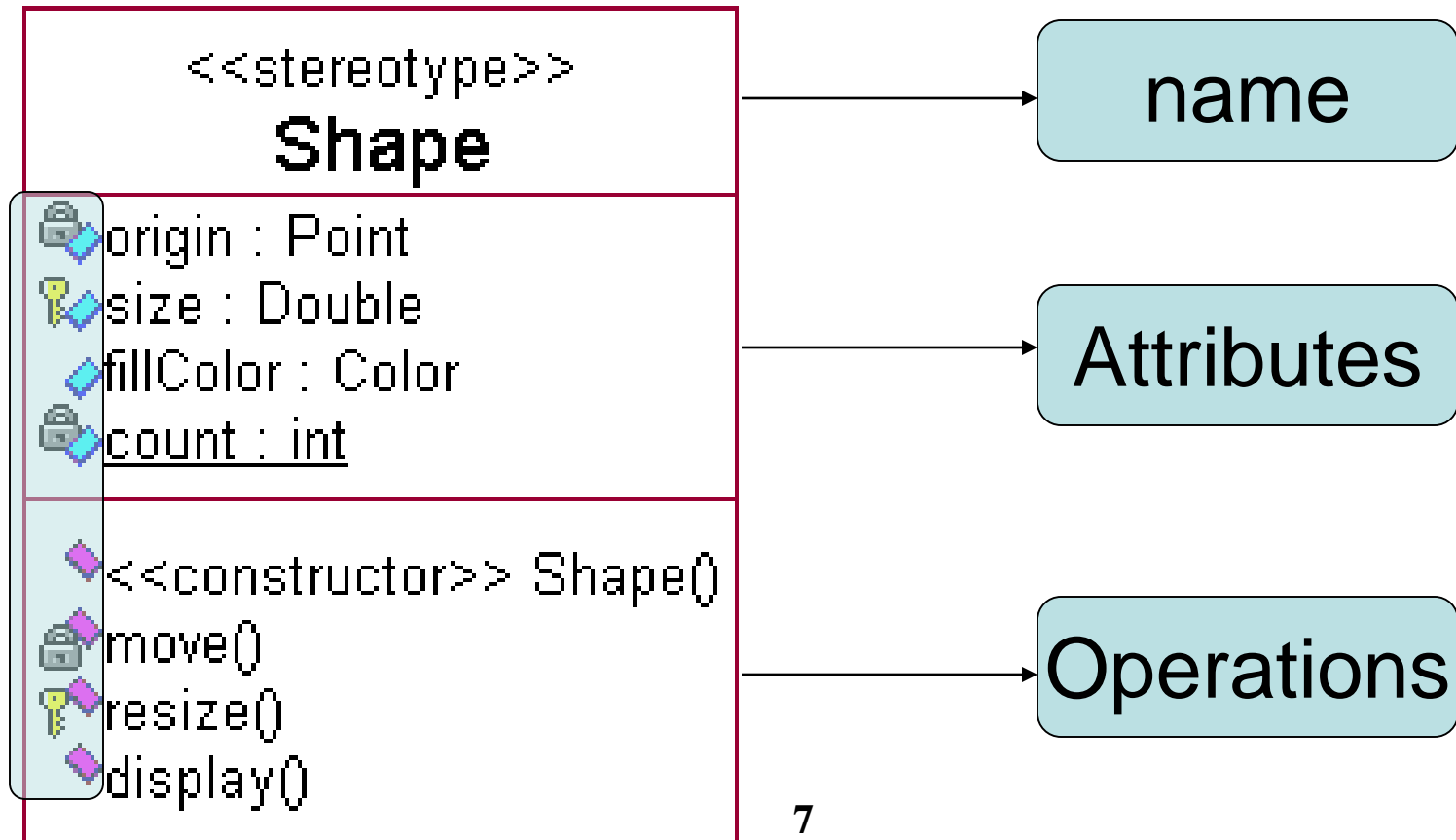
Class

● Definition

- A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics

Symbol of class

- Graphically, a class is rendered as a rectangle



Names of Class

● Names of Class

- Every class must have a name that distinguishes it from other classes
- A name is a textual string
- A class name must be unique within its enclosing package
- In practice, class names are short nouns or noun phrases drawn from the vocabulary of the system you are modeling. Typically, you capitalize the first letter of every word in a class name, as in Customer or TemperatureSensor

● Names of Class

- Simple name
- Qualified name

Sensor

Customer

Wall

Banking::CheckingAccount

Java:awt:Rectangle





Attribute

- An attribute is a named property of a class that describes a range of values that instances of the property may hold
- Graphically, attributes are listed in a compartment just below the class name

Form of Attribute

- In its full form, the syntax of an attribute in the UML is
 - [visibility] name [':' type] ['[' multiplicity] ']'] ['=' initial-value] [property-string {',' property-string}]

Visibility of Attribute

- **[visibility]** name [':' type] ['[' multiplicity] ']' ['=' initial-value] [property-string {',' property-string}]
- The visibility of a feature specifies whether it can be used by other classifiers. In the UML, you can specify any of four levels of visibility
 - +:Public
 - #:Protected
 - -:Private
 - ~:Package
- Rational Rose
 -  Public
 -  Protected
 -  Private
 -  Implementation

Multiplicity of Attribute

- [visibility] name [':' type] [' multiplicity '] ['=' initial-value] [property-string {' , ' property-string}]

- Multiplicity

- 0..1 0 or 1
- 1 1 exactly
- 0..* 0 1 or more
- * 0 1 or more
- 1..* 1 or more
- 3 3 exactly
- 0..5 between 0 and 5
- 5..15 between 5 and 15

Property-string of Attribute

- [visibility] name [':' type] ['[multiplicity] ']'
['=' initial-value] [property-string {' ,'
property-string}]

- Constraint to the attribute

 - Name:String="COSE SEU." {readOnly}

```
public class ManagementSystem
```

```
{
```

```
private final String Name = "COSE SEU.";
```

```
}
```

Example of Attributes

- origin Name only
- + origin Visibility and name
- origin : Point Name and type
- name : String[0..1]
multiplicity Name, type, and
- origin : Point = (0,0) Name, type, and initial
value
- id: Integer {readonly} Name and property

Organizing Attributes

- When drawing a class, you don't have to show every attribute at once

Operation

- An operation is the implementation of a service that can be requested from any object of the class to affect behavior
 - In other words, an operation is an abstraction of something you can do to an object that is shared by all objects of that class
- A class may have any number of operations or no operations at all
- Graphically, operations are listed in a compartment just below the class attributes
- In its full form, the syntax of an operation in the UML is
 - `[[visibility]] name ['(' parameter-list ')'] [[:' return-type']
[[property-string {' , ' property-string }]]`

Example of Operations

- Display

- Name only

- + display

- Visibility and name

- set(n : Name, s : String)

- Name and parameters

- getID() : Integer

- Name and return type

- restart() {guarded}

- Name and property

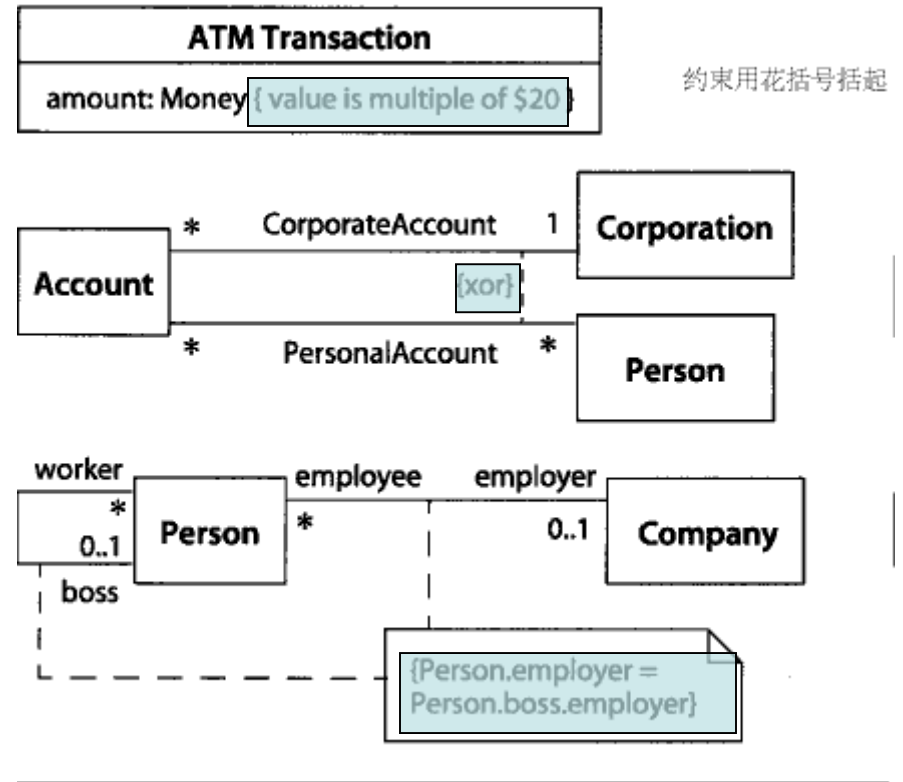
Constraint

- With constraints, you can add new semantics or extend existing rules
- A constraint specifies conditions that a run-time configuration must satisfy to conform to the model
- A constraint is rendered as **a string enclosed by brackets** and placed near the associated element

Constraint of Class

Title
name : String author : String isbn : String / number of reservations
\$find() create() destroy()

{Constraint}

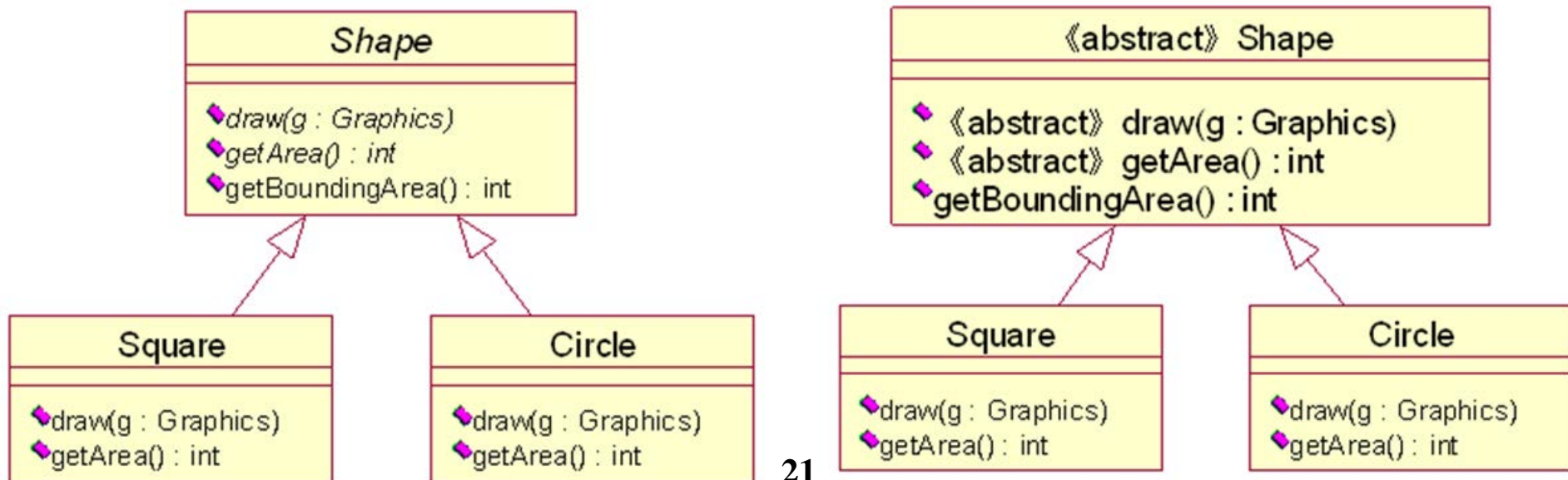


Advanced Classes

- Abstract class
- Interface
- Association class
- Template class
- Active class
- Nested class

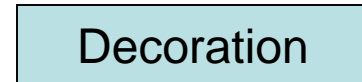
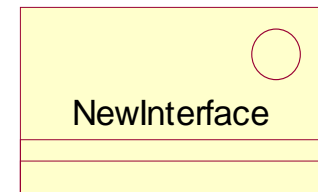
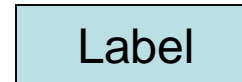
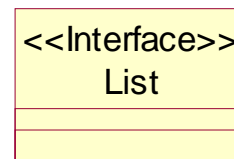
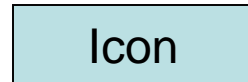
Abstract class

- A class that cannot be directly instantiated
- In the UML, you specify that a class is abstract by writing its name in italics



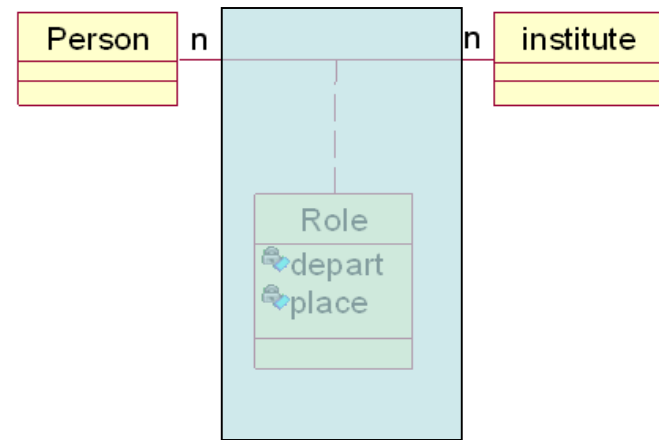
Interface

- An interface is a collection of operations that are used to specify a service of a class or a component
- Graphically, an interface may be rendered as a stereotyped class in order to expose its operations and other properties



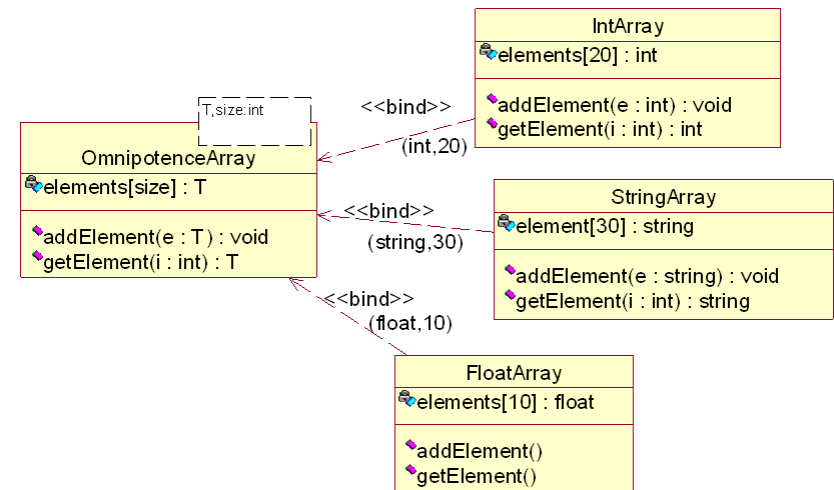
Association class

- A modeling element that has both association and class properties
 - An association class can be seen as an association that also has class properties or as a class that also has association properties
 - You render an association class as a class symbol attached by a dashed line to an association line



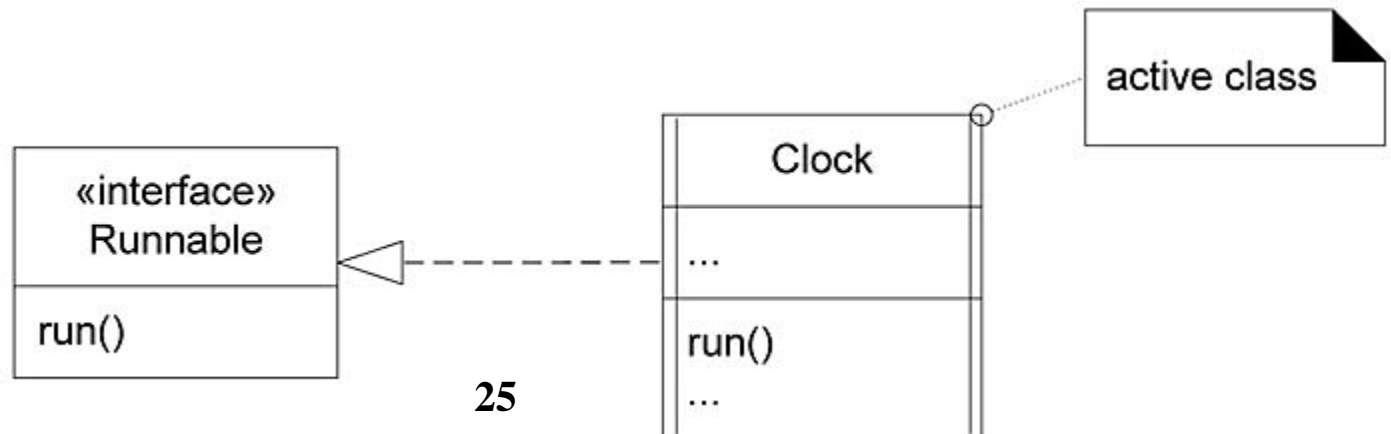
Template class

- A template is a parameterized element
 - A template may include slots for classes, objects, and values, and these slots serve as the template's parameters
 - You can't use a template directly; you have to instantiate it first
 - Instantiation involves binding these formal template parameters to actual ones
 - For a template class, the result is a concrete class that can be used just like any ordinary class



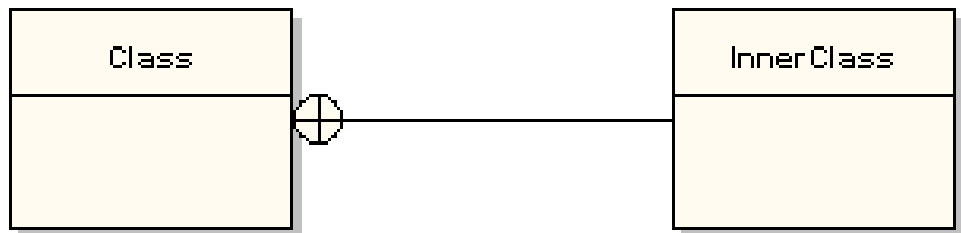
Active class

- An active object is an object that owns a process or thread and can initiate control activity
- An active class is a class whose instances are active objects
- Graphically, an active class is rendered as a rectangle with double lines for left and right sides. Processes and threads are rendered as stereotyped active classes



Nested class

- A class can be declared within the scope of another class. Such a class is called a "nested class."
- Nested classes are considered to be within the scope of the enclosing class and are available for use within that scope

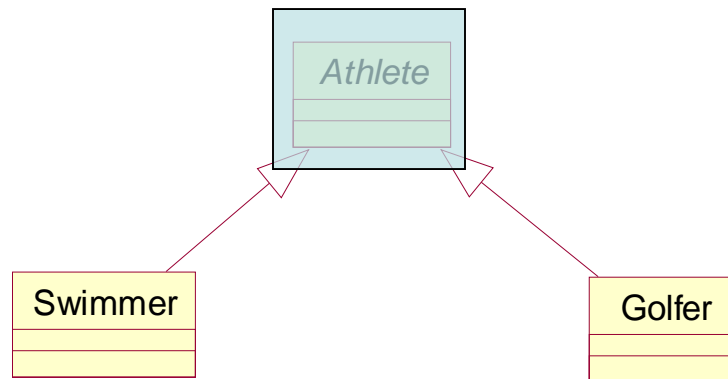


Relationships Between Classes

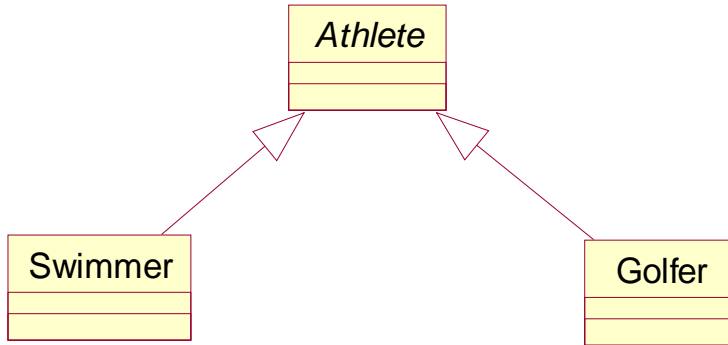
- Generalization
- Association
- Aggregation, Composition
- Realization
- Dependency

Generalization

- A generalization is a relationship between a general kind of thing (called the superclass or parent) and a more specific kind of thing (called the subclass or child)
- Generalization is sometimes called an "is-a-kind-of" relationship: one thing (like the class BayWindow) is-a-kind-of a more general thing (for example, the class Window)
- Graphically, generalization is rendered as a solid directed line with a large unfilled triangular arrowhead, pointing to the parent



Java Code Example (Generalization)



```
public class Golfer extends Athlete
{
    /**
     * @roseuid 4ABBFA2B0050
     */
    public Golfer()
    {
    }
}
```

```
public abstract class Athlete
{
    /**
     * @roseuid 4ABBFA2B0083
     */
    public Athlete()
    {
    }
}
```

```
public class Swimmer extends Athlete
{
    /**
     * @roseuid 4ABBFA2B0031
     */
    public Swimmer()
    {
    }
}
```

Association

- An association is a structural relationship that specifies that objects of one thing are connected to objects of another
 - Given an association connecting two classes, you can relate objects of one class to objects of the other class
- Graphically, an association is rendered as a solid line connecting the same or different classes



Association

- An association have at least two association ends. Each association end should be connected to a class
- Navigation
 - Unless otherwise specified, navigation across an association is **bidirectional**. However, there are some circumstances in which you'll want to limit navigation to just **one direction (unidirectional)**



Java Code Example (Association)



```
public class A
{
    public B theB;

    /**
     * @roseuid 4ABC122802AC
     */
    public A()
    {
    }
}

public class B
{
    /**
     * @roseuid 4ABC122802CC
     */
    public B()
    {
    }
}
```



```
public class A
{
    public B theB;

    /**
     * @roseuid 4ABBF567016B
     */
    public A()
    {
    }
}

public class B
{
    public A theA;

    /**
     * @roseuid 4ABBF5670190
     */
    public B()
    {
    }
}
```

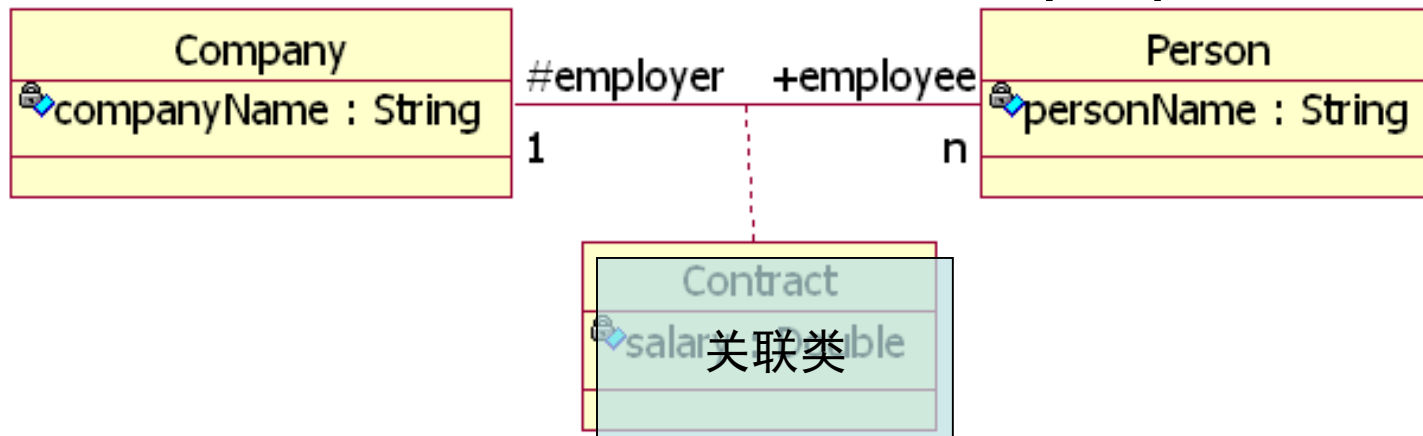

Name of Association

- An association can have a name, and you use that name to describe the nature of the relationship
 - Although an association **may** have a name, you typically don't need to include one if you explicitly provide end names for the association
 - If you have more than one association connecting the same classes, it is **necessary** to use either association names or association end names to distinguish them
 - If an association has more than one end on the same class, it is **necessary** to use association end names to distinguish the ends
 - If there is only one association between a pair of classes, some modelers omit the names, but it is **better** to provide them to make the purpose of the association clear

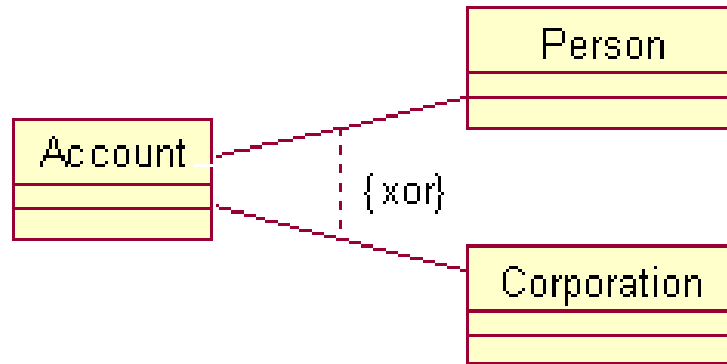


Association class

- In an association between two classes, the association itself might have properties
 - An association class can be seen as an association that also has class properties or as a class that also has association properties



Constraint of Association



Role of Association

- When a class participates in an association, it has a specific role that it plays in that relationship
 - a role is just the face the class at the far end of the association presents to the class at the near end of the association

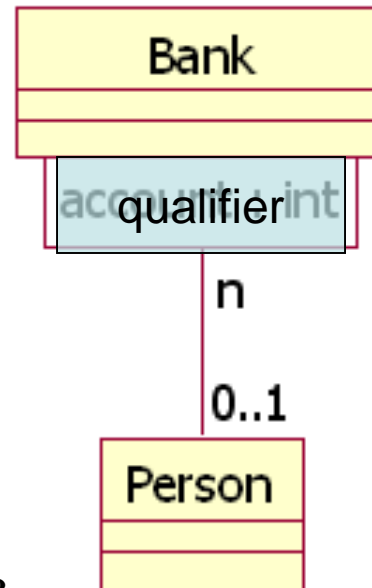


Multiplicity of Association's Role

- An association represents a structural relationship among objects. In many modeling situations, it's important for you to state how many objects may be connected across an instance of an association. This "how many" is called the multiplicity of an association's role
- Representation
 - 0..1
 - 0..n
 - 1
 - 1..*
 - 0..*
 - 7
 - 3, 6..9
 - 0

Qualification of Association

- In the UML, a qualifier is an association attribute whose values identify a subset of objects (usually a single object) related to an object across an association
- You render a qualifier as a small rectangle attached to the end of an association, placing the attributes in the rectangle



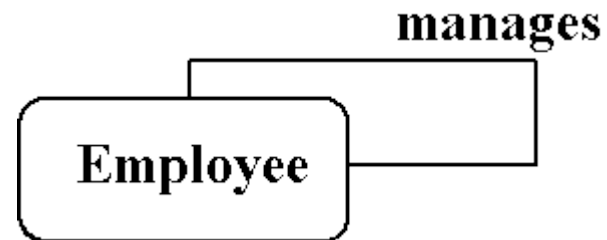
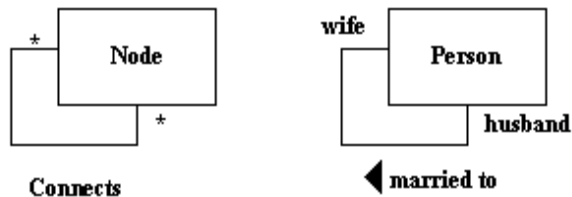
Arity of Association

- Each association has specific arity as it could relate two or more items
 - Recursive or Reflexive Association
 - Binary Association
 - N-ary Association

Recursive or Reflexive Association

- A recursive association is where a class is associated to itself
 - When a class is associated with itself, this does not mean that a class's instance is related to itself, but that instance of the class is related to another instance of the class

Recursive Association

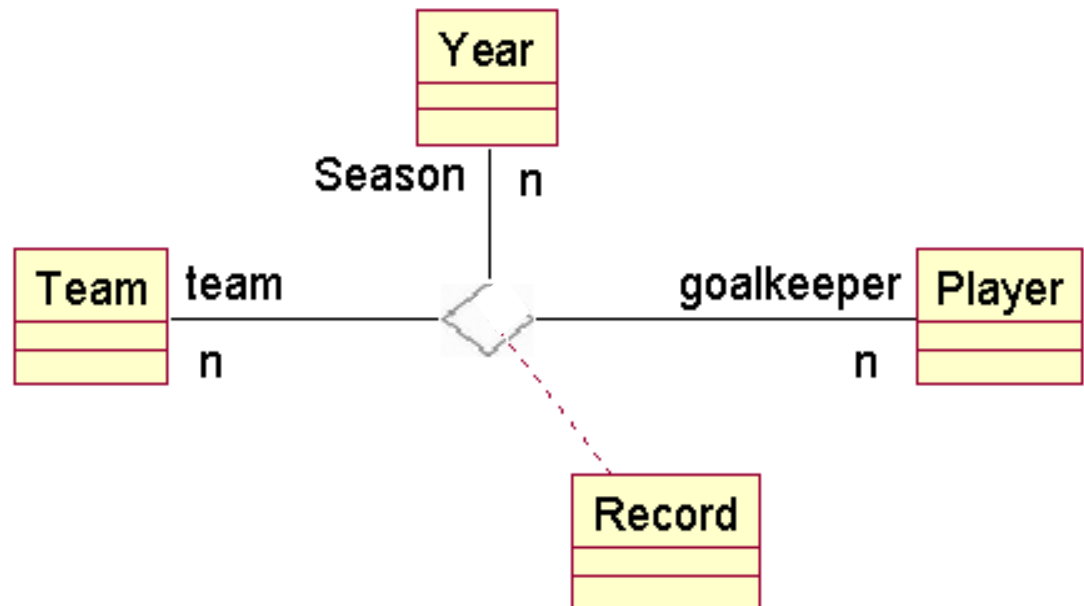


Binary Association

- An association that connects exactly two classes is called a binary association

N-ary Association

- Although it's not as common, you can have associations that connect more than two classes; these are called n-ary associations



Aggregation

- A special form of association that specifies a whole-part relationship between the aggregate (the whole) and a component (the part)
 - Aggregation is specified by adorning a plain association with an unfilled diamond at the whole end



Java Code Example (Aggregation)



```
public class Circle
{
    public Style theStyle;
    /**
     * @roseuid 4ABC6ECE0334
     */
    public Circle()
    {
    }
}
```

```
public class Rectangle
{
    public Style theStyle;
    /**
     * @roseuid 4ABC6ECE0363
     */
    public Rectangle()
    {
    }
}
```

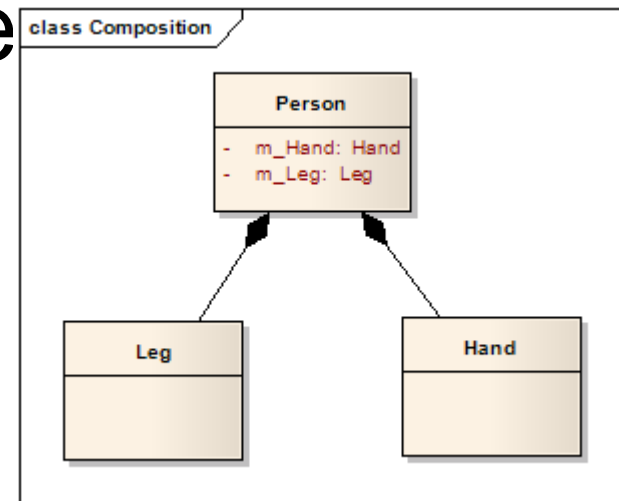
```
public class Style
{
    private int Color;
    private int isFilled;
    public Circle theCircle;
    public Rectangle theRectangle;
    /**
     * @roseuid 4ABC6ECE03A9
     */
    public Style()
    {
    }
}
```

Aggregation

- A **plain association** between two classes represents a structural relationship between peers, meaning that both classes are conceptually **at the same level**, no one more important than the other
- Sometimes you will want to model a "**whole/part**" relationship, in which one class represents a larger thing (the "whole"), which consists of smaller things (the "parts")
- This kind of relationship is called aggregation, which represents a "**has-a**" relationship, meaning that an object of the whole has objects of the part

Composition

- There is a variation of simple aggregation composition that does add some important semantics
- Composition is a form of aggregation, with strong ownership and coincident lifetime as part of the whole

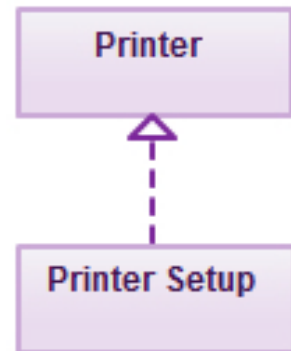


Aggregation and Composition

- Aggregation: has-a
- Composition: contains-a

Realization

- A realization is a semantic relationship between classifiers in which one classifier specifies a contract that another classifier guarantees to carry out
 - Graphically, a realization is rendered as a dashed directed line with a large open arrowhead pointing to the classifier that specifies the contract



Dependency

- A dependency is a relationship that states that one thing (for example, class Window) uses the information and services of another thing (for example, class Event), but not necessarily the reverse
- Graphically, a dependency is rendered as a dashed directed line, directed to the thing being depended on
- Choose dependencies when you want to show one thing using another



Relationships Between Classes

- Generalization and Realization

- The concept is more clear

- Association, Aggregation, Composition and Dependency

- These relationships are not fully distinguished on the code level
- Strength of the relationship: Composition, Aggregation, Association, Dependency

Representation of Relations between Classes

- Generalization



- Realization



- Dependency



- Association



- Aggregation



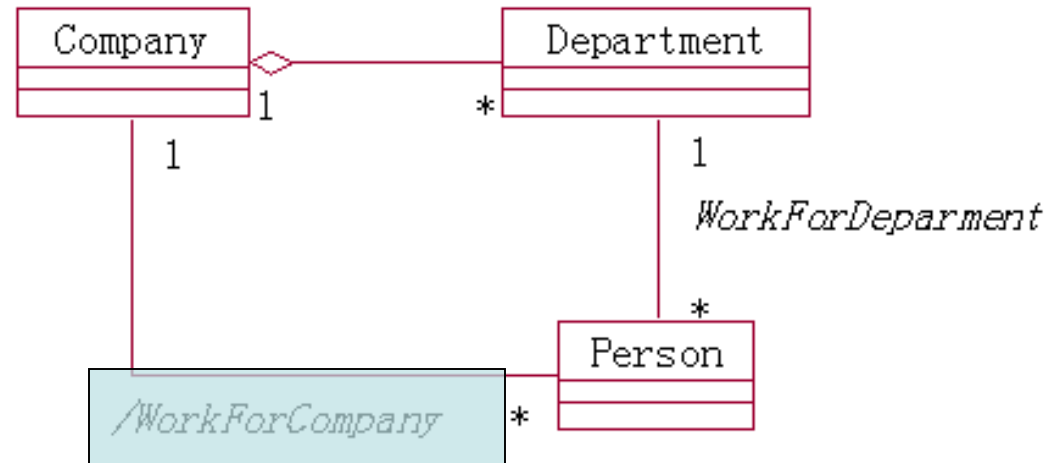
- Composition



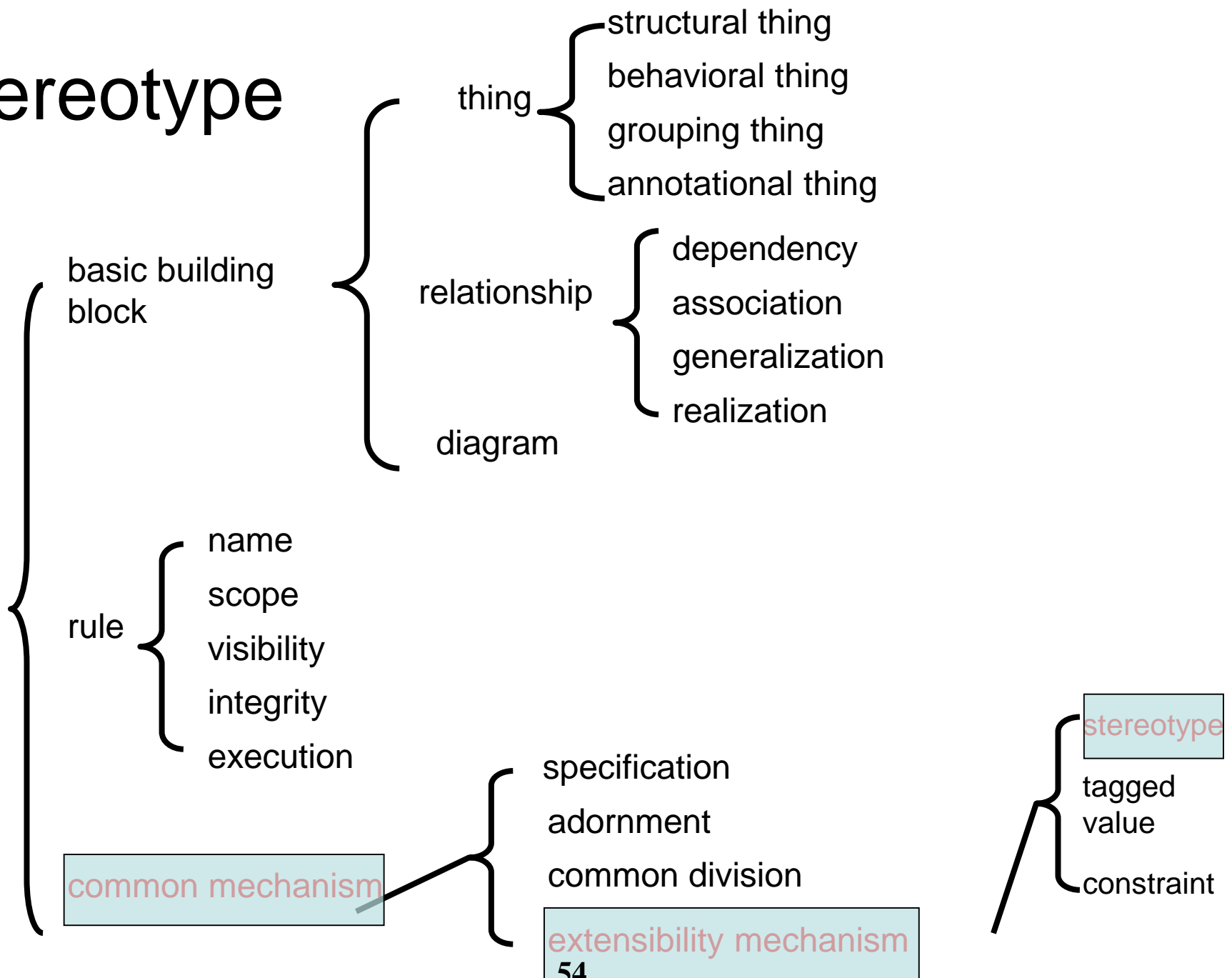
Derived Associations and Attributes

- Derived associations and derived attributes can be calculated from other associations and attributes, respectively, on a class diagram
 - For example, an age attribute of a Person can be derived if you know that Person's date of birth
- The names of derived attributes are automatically prefixed with a slash (/)
- As with a derived attribute, the name of a derived association in a class diagram is preceded by a slash (/)

Example of Derived Associations and Attributes



Stereotype



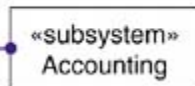
Stereotype

- When you stereotype an element such as a node or a class, you are in effect extending the UML by creating a new building block just like an existing one but with its **own special modeling properties** (each stereotype may provide its own set of tagged values), **semantics** (each stereotype may provide its own constraints), and **notation** (each stereotype may provide its own icon)

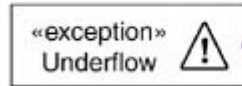
Stereotype

- In its simplest form, a stereotype is rendered as a name enclosed by **guillemets** (for example, «name») and placed above the name of another element. As a visual cue, you may define **an icon** for the stereotype and render that icon to the right of the name (if you are using the basic notation for the element) or **use that icon as the basic symbol** for the stereotyped item

named stereotype



named stereotype with icon



stereotyped element as icon

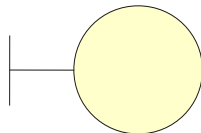


Three Stereotypes of class

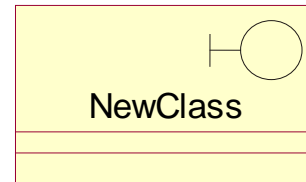
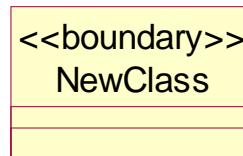
- Boundary class
- Entity class
- Control class

Boundary class

- A boundary class is a class used to model interaction between the system's surroundings and its inner workings
 - User interface classes
 - classes which intermediate communication with human users of the system
 - System interface classes
 - classes which intermediate communication with other systems
 - Device interface classes
 - classes which provide the interface to devices (such as sensors), which detect external events



NewClass



Boundary class

- Find User-Interface Classes

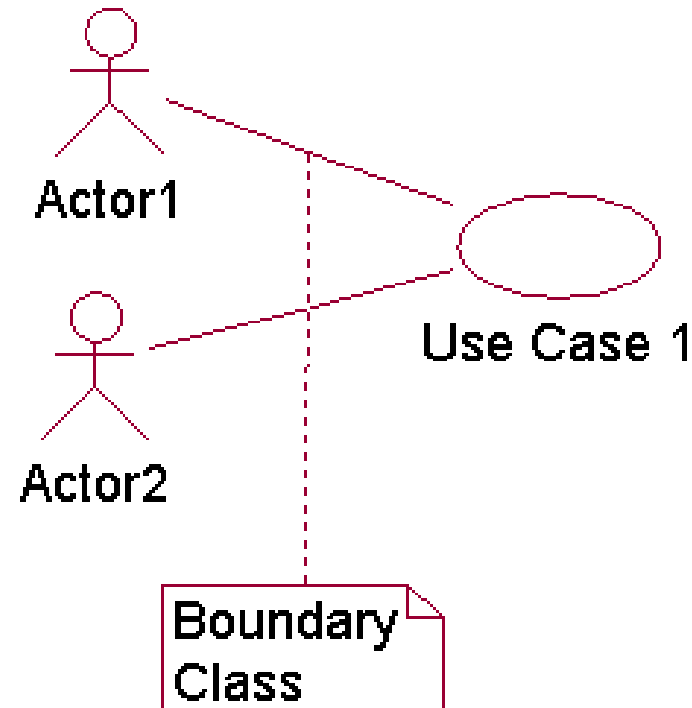
- Boundary classes representing the user interface may exist from user-interface modeling activities
- There is at least one boundary object for each use-case actor-pair

- Find System-Interface Classes

- A boundary class which communicates with an external system is responsible for managing the dialogue with the external system

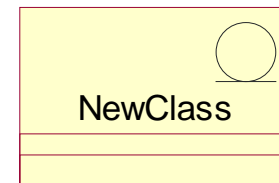
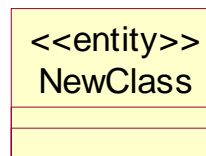
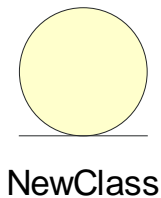
- Find Device Interface Classes

- The system may contain elements that act as if they were external (change value spontaneously without any object in the system affecting them), such as sensor equipment



Entity Class

- An entity class is a class used to model information and associated behavior that must be stored
- Typical examples of entity classes in a banking system are Account and Customer

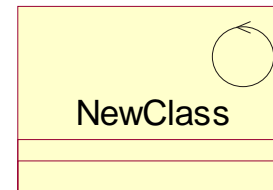
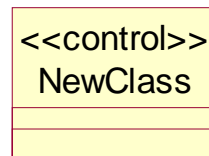
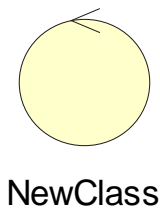


Entity Class

- Entity classes represent stores of information in the system; they are typically used to represent the key concepts the system manages
- A frequent source of inspiration for entity classes are the **Glossary** (developed during requirements) and a **business-domain model** (developed during business modeling, if business modeling has been performed)

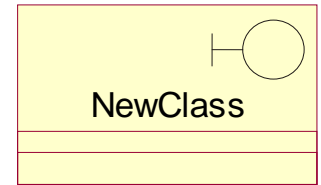
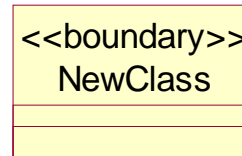
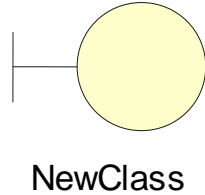
Control Class

- A control class is a class used to model control behavior specific to one or a few use cases

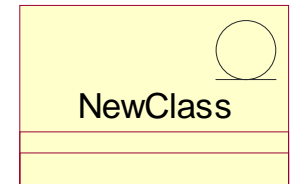
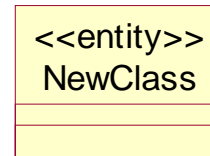
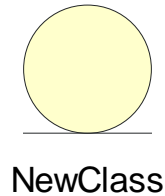


Boundary, Control, and Entity Classes

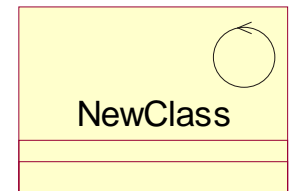
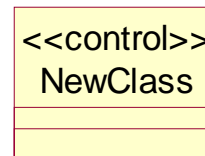
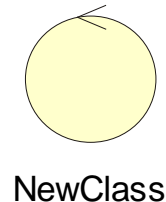
● Boundary



● Entity



● Control



Class Diagram

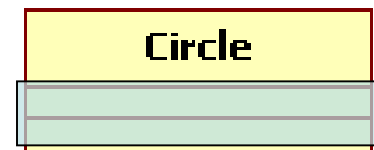
- A class diagram is a diagram that shows a set of classes, interfaces, and collaborations and their relationships
- Graphically, a class diagram is a collection of vertices and arcs
- Elements
 - Classes, Interfaces and relations
 - May contain notes and constraints
 -
- Important
 - For visualizing, specifying, and documenting structural models
 - For constructing executable systems through forward and reverse engineering
 -

Three Perspectives of Class Diagram

- Unified Modeling Language (UML)
Models represent systems at different levels of detail
- Some models describe a system from a higher, more abstract level, while other models provide greater detail
 - Conceptual
 - Specification
 - Implementation

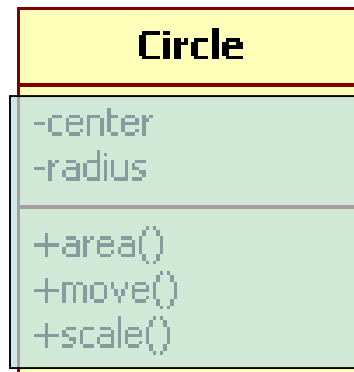
Conceptual

- In this case you are drawing a diagram that represents the concepts in the domain under study
 - These concepts will naturally relate to the classes that implement them, but it is often not a direct mapping
- Indeed the model is drawn with little or no regard for the software that might implement it, and is generally language independent



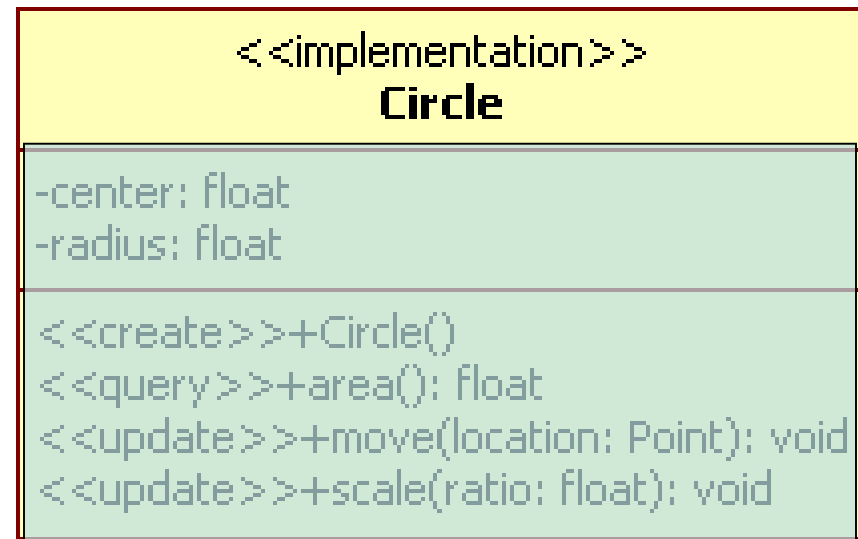
Specification

- At the specification level, you want to show the interfaces of each class
- At this level you'd want to make explicit the class responsibilities, as embodied in the public operations for each class.



Implementation

- At the implementation level, you want to show more precisely how a class was (or needs to be) implemented in code
 - Now you'd include the private and the protected attributes and operations as well.



Comparison of Three Perspectives

Circle

Circle
-center -radius
+area() +move() +scale()

<<implementation>> Circle
-center: float -radius: float
<<create>>+Circle() <<query>>+area(): float <<update>>+move(location: Point): void <<update>>+scale(ratio: float): void

Three Perspectives of Class Diagram

- The lines between the perspectives are not sharp, and most modelers do not take care to get their perspective sorted out when they are drawing
- Understanding the perspective is crucial to both drawing and reading class diagrams
 - When you are drawing a diagram, draw it from a single clear perspective
 - when you read a diagram make sure you know which perspective the drawer drew it in

How To Find Classes

- Determine candidate classes from the nouns of use case description
- Use CRC (Class responsibility collaborator) method to discover classes
- Use the division of Boundary, Control, and Entity Classes
- Determine classes according to design patterns
- Find classes under the guidance of the software development process

A Few Suggestions for Class Diagram Modeling

- Do not try to use all of the symbols

- Use only when really needed

- Do not fall into the details too early

- Analysis phase

Conceptual

- Design phase

Specification

- Implementation phase

Implementation

- After completion of class diagram modeling, you should consider

- Whether the model reflect the real situation of the domain

- The granularity of model elements is suitable?

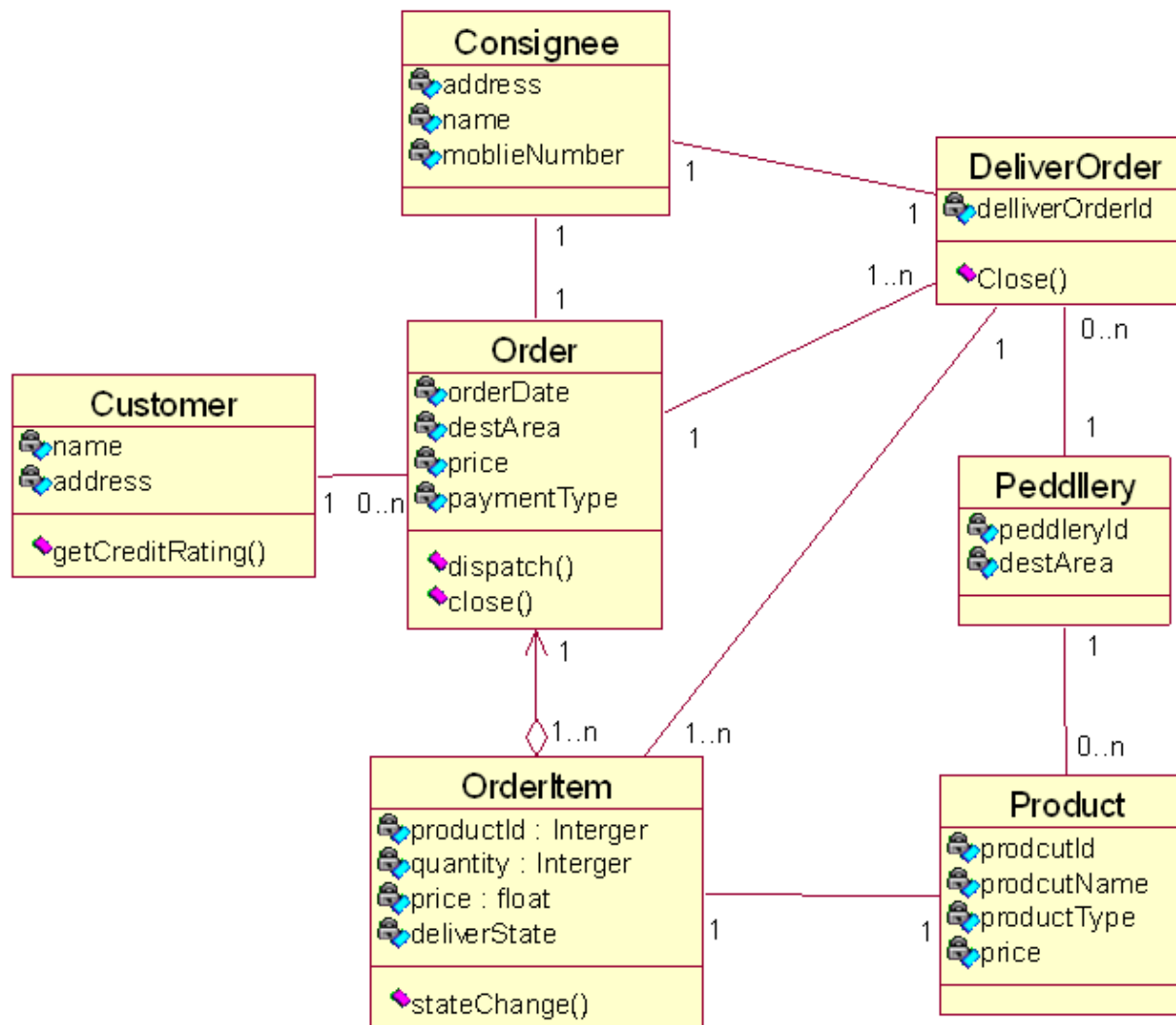
Steps of Class Diagram Modeling

- Domain analysis, determine system requirements
- Determine classes and their responsibility, determine the attributes and operations
- Initially determine the relationships between classes
- Refine the relationships between classes
- Draw the class diagram, and give the necessary textual description

How To Read A Class Diagram

- Grasp three key elements: **classes**, **relationships** and **multiplicity**
- Firstly, observe which **classes** exist in the class diagram
- Then focus on the **relationships** between classes
- Combined with the **multiplicity** to understand the structural characteristics of class diagram and the meaning of attributes and operations

Read A Class Diagram



Principles of Object Oriented Class Design

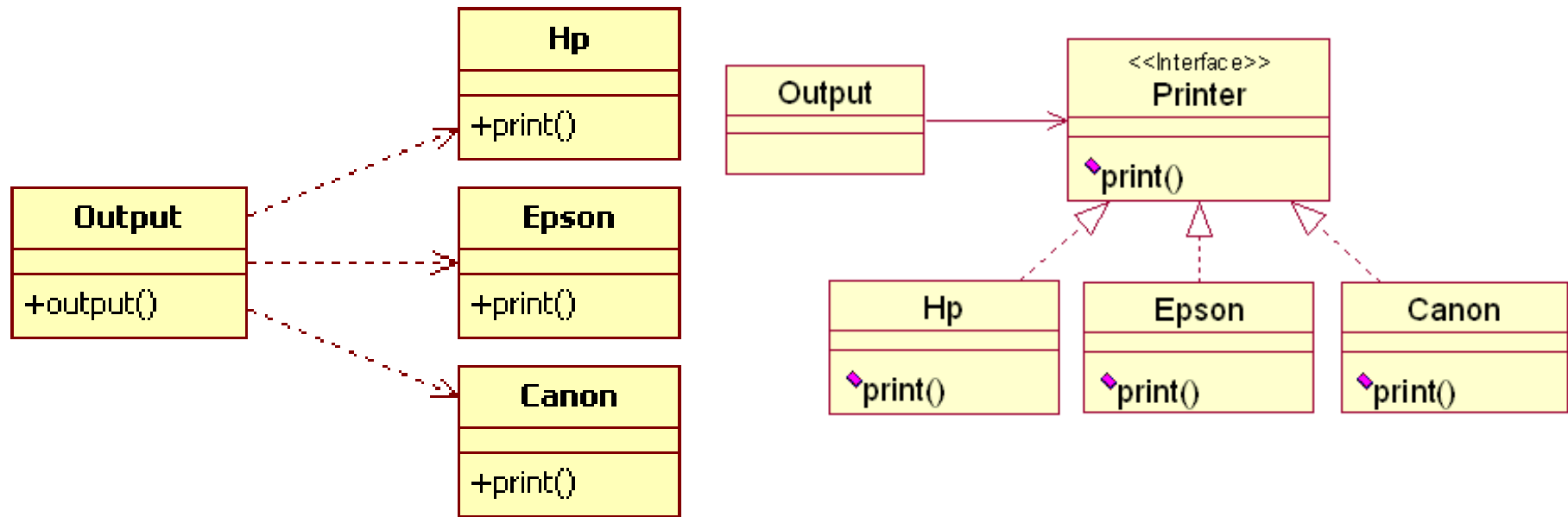
- Open Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Dependency Inversion Principle (DIP)
- Interface Segregation Principle (ISP)

Open Closed Principle (OCP)

- Software entities (classes, modules, functions, etc.) should be **open** for extension, but **closed** for modification
 - Bertrand Meyer
- Of all the principles of object oriented design, this is the most important
- We should write our modules so that they can be **extended**, without requiring them to be modified
 - In other words, we want to be able to change what the modules do, without changing the source code of the modules

Example of Open Closed Principle

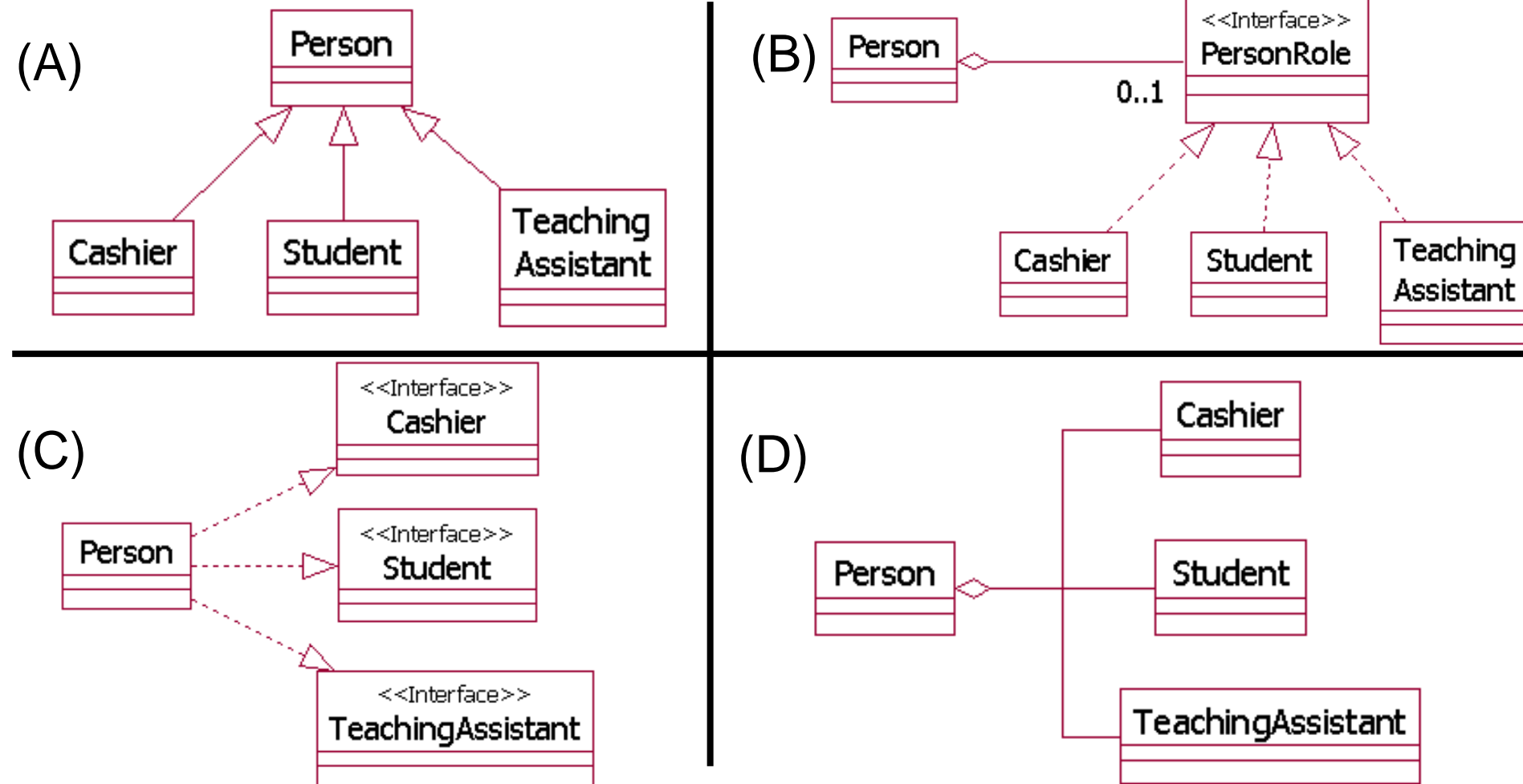
● Design of print



Example of Open Closed Principle

- A graduate is a **teaching assistant** of COSE, and he is also working as a **cashier** in the campus restaurant
- In other words, the graduate has three roles: **student, teaching assistant and cashier**, but at the same time there is only one role
- According to the above description, which design is the most reasonable?

Example of Open Closed Principle



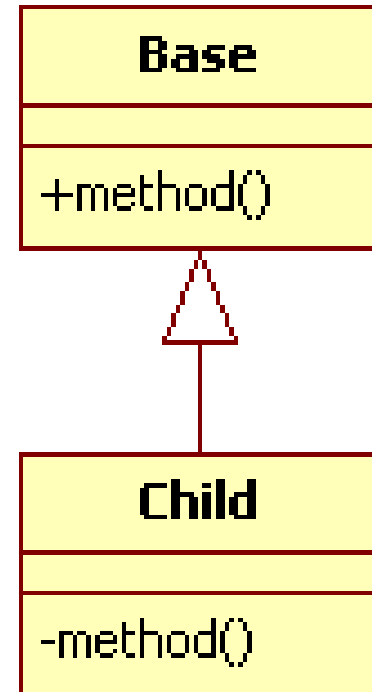
Liskov Substitution Principle (LSP)

- Subclasses should be substitutable for their base classes

- Barbar Liskov

- Derived classes should be substitutable for their base classes

- That is, a user of a base class should continue to function properly if a derivative of that base class is passed to it

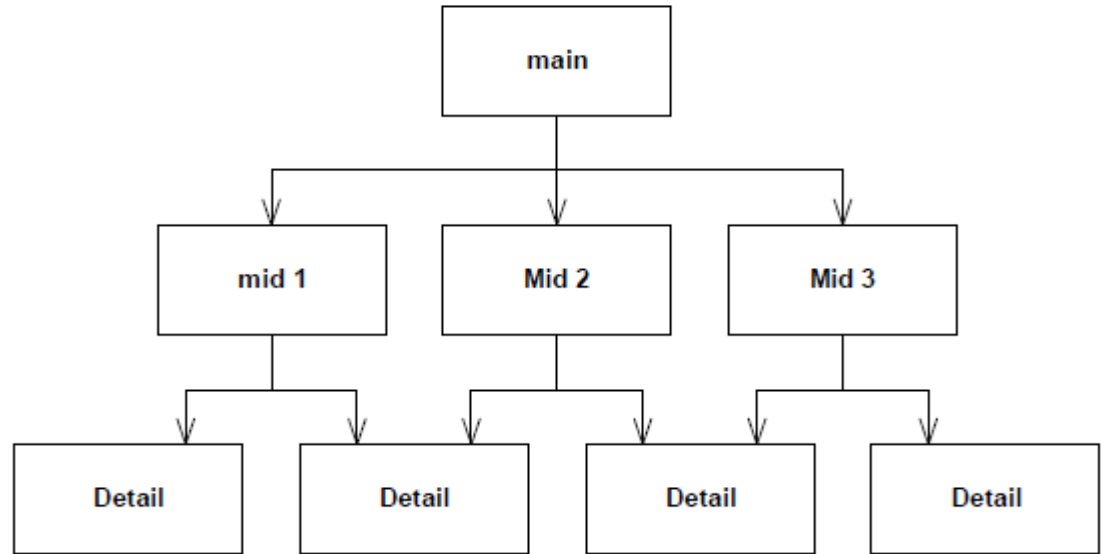


Dependency Inversion Principle (DIP)

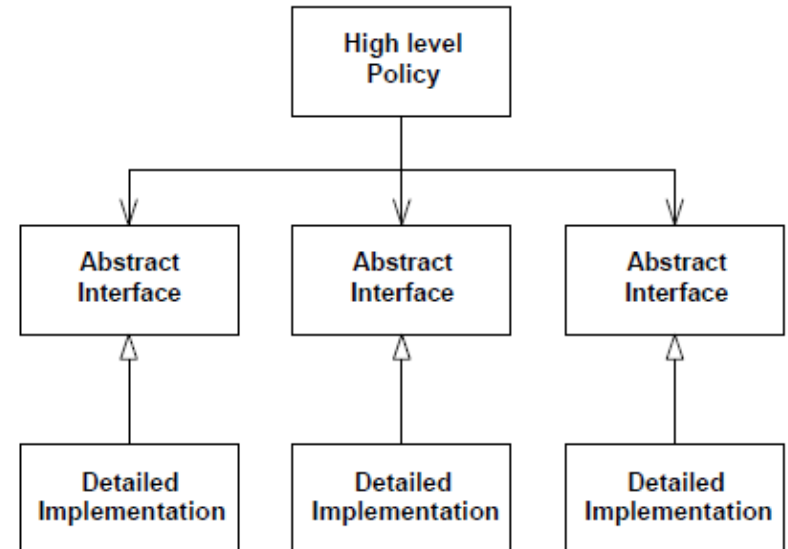
- Dependency Inversion is the strategy of depending upon interfaces or abstract functions and classes, rather than upon concrete functions and classes

Comparison of Procedural Design and Object Oriented Design

Dependency Structure of
Procedural Architecture

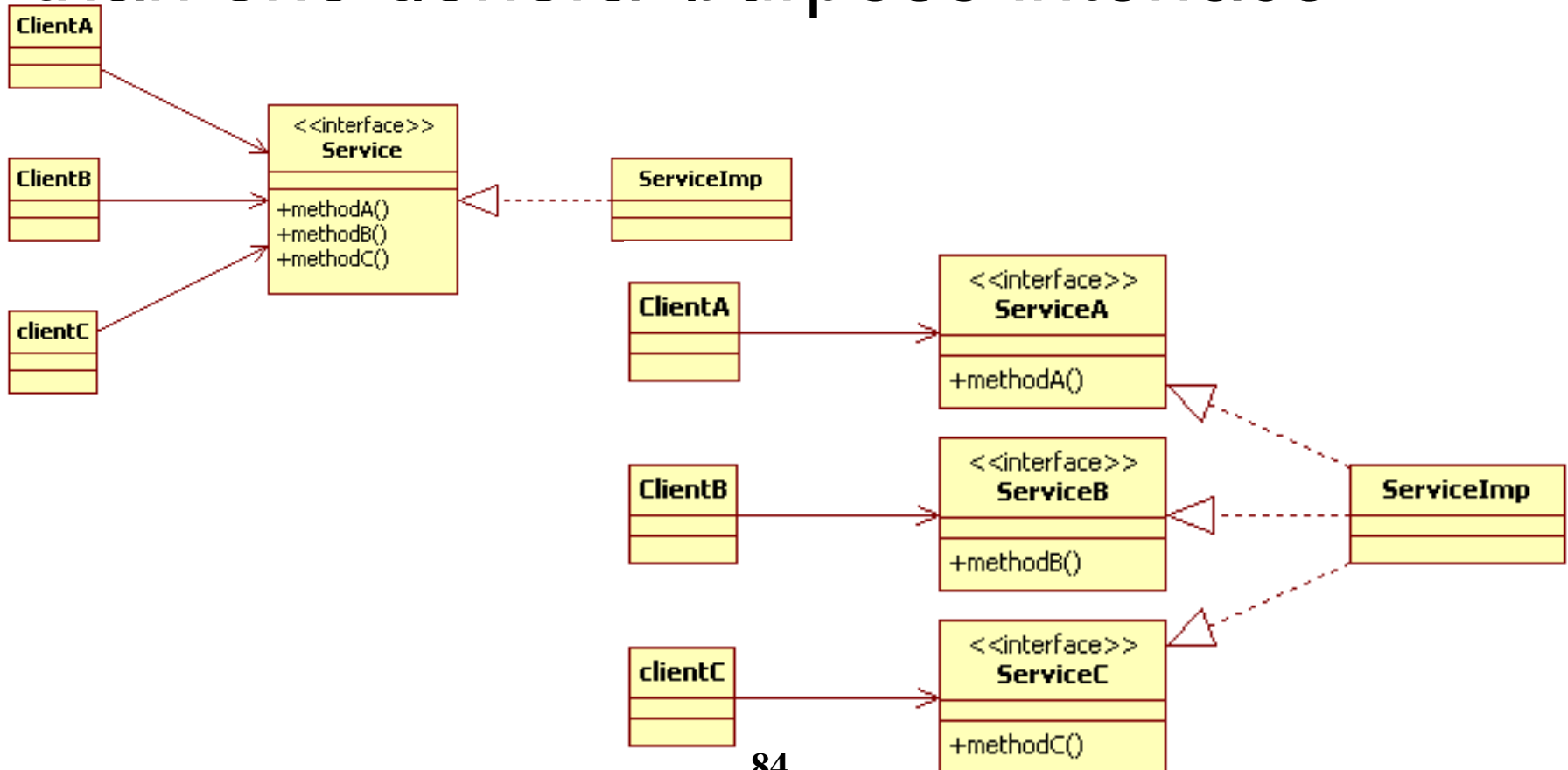


Dependency Structure
of an Object Oriented
Architecture

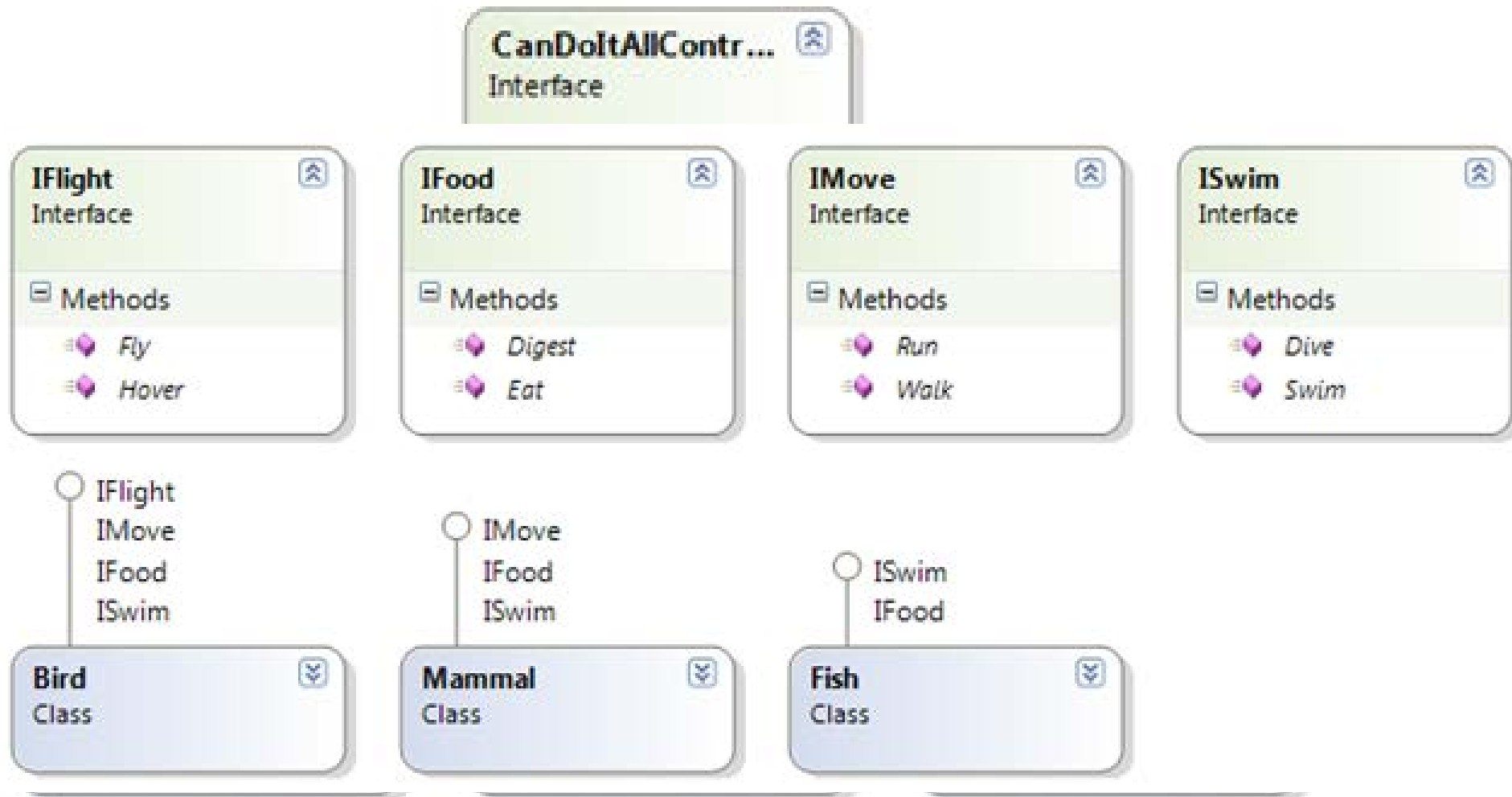


Interface Segregation Principle (ISP)

- Many client specific interfaces are better than one general purpose interface



Example of Interface Segregation Principle (ISP)



Hints and Tips of Object Oriented Design

- Names of similar operations in different classes should be the same
- Comply with existing conventions
- Design simple classes
- Define simple operations
- The levels of generalization should be appropriate

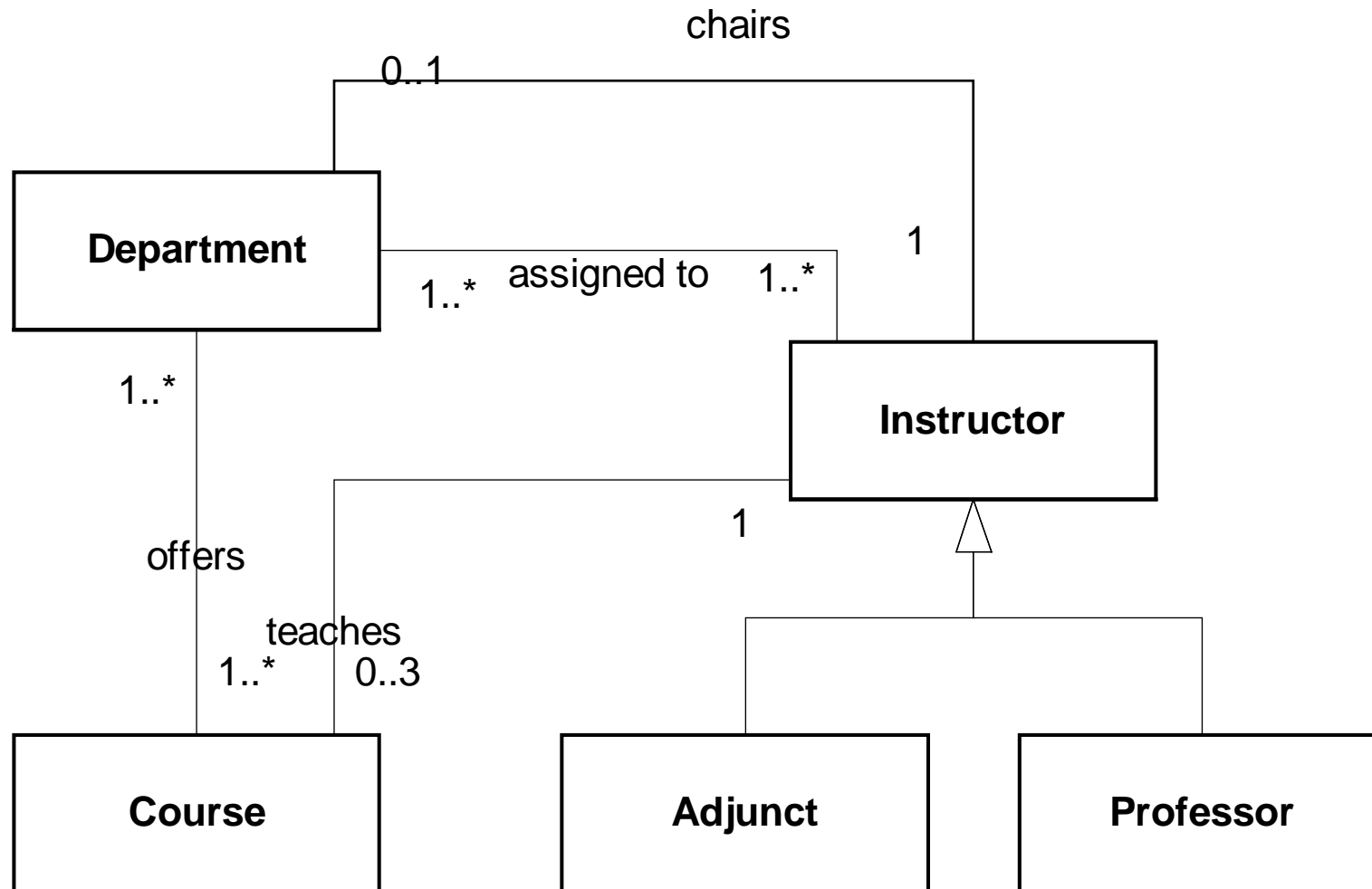
Characteristics of a well designed system

- Friendly
- Understandable
- Reliable
- Extensible
- Portable
- Scalable
- Reusable
-
- **Simplicity**: easy to implement easy to use, simple to understand, easy to maintain
 - You can model 80 percent of most problems by using about 20 percent of the UML. Basic structural things, such as classes, attributes, operations, use cases, and packages, together with basic structural relationships, such as dependency, generalization, and association, are sufficient to create static models for many kinds of problem domains.

Example: University Courses

- Some instructors are professors, while others have job title adjunct
- Departments offer many courses, but a course may be offered by >1 department
- Courses are taught by instructors, who may teach up to three courses
- Instructors are assigned to one (or more) departments
- One instructor also serves a department chair

Class Diagram for University Courses



Style of UML Class Diagram

- Be Consistent with Attribute Names and Types
 - It would not be consistent for an attribute named customer-Number to be a string, although it would make sense for it to be an integer
- Do Not Name Associations That Have Association Classes
 - The name of the association class should adequately describe it. Therefore, the association does not need an additional adornment indicating its name.

Style of UML Class Diagram

- List Operations/Attributes in Order of Decreasing Visibility
 - The greater the visibility of an operation or attribute, the greater the chance that someone else will be interested in it
- Always Indicate the Multiplicity
 - For each class involved in a relationship, there will always be a multiplicity

Style of UML Class Diagram

● Avoid a Multiplicity of “*”

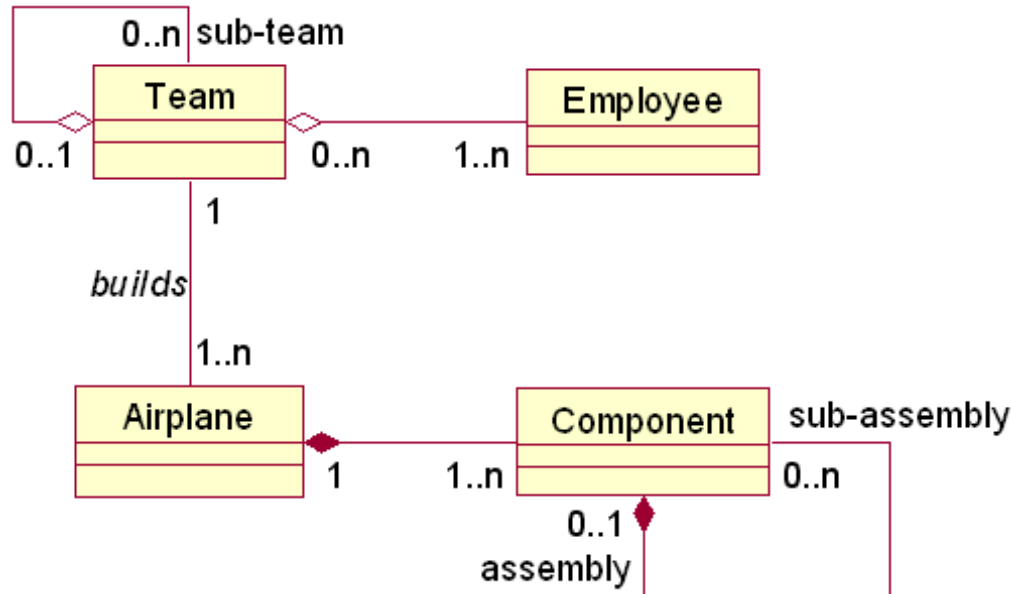
- You should avoid the use of “*” to indicate multiplicity on a UML class diagram because your reader can never be sure if you really mean “0..*” or “1..*”
- Although the UML specification clearly states that “*” implies “0..*”

● Do Not Model Every Dependency

- Model a dependency between classes only if doing so adds to the communication value of your diagram

Style of UML Class Diagram

- Place the Whole to the Left of the Part



- Don't Worry About the Diamonds

- When you are deciding whether to use aggregation or composition over association, Craig Larman (2002) says it best: "If in doubt, leave it out."