

# ABSTRACT CLASSES AND INTERFACES

# ABSTRACT CLASSES

- ▶ Unlike classes, these **cannot** be instantiated.
- ▶ Like classes, they introduce **types**.
  - ▶ but no objects can have as actual type the type of an abstract class.
- ▶ Why use them?
  - ▶ Because there is a set of common features and implementation for all derived classes but...
    - ▶ We want to prevent users from handling objects that are too generic (Example 1)
    - ▶ We cannot give a full implementation for the class (Example 2)



# EXAMPLE 1

The problem:

- Students are either undergraduate, PhD or MsC.
- We want to guarantee that nobody creates a Student object. The application always creates a specific kind of Student.

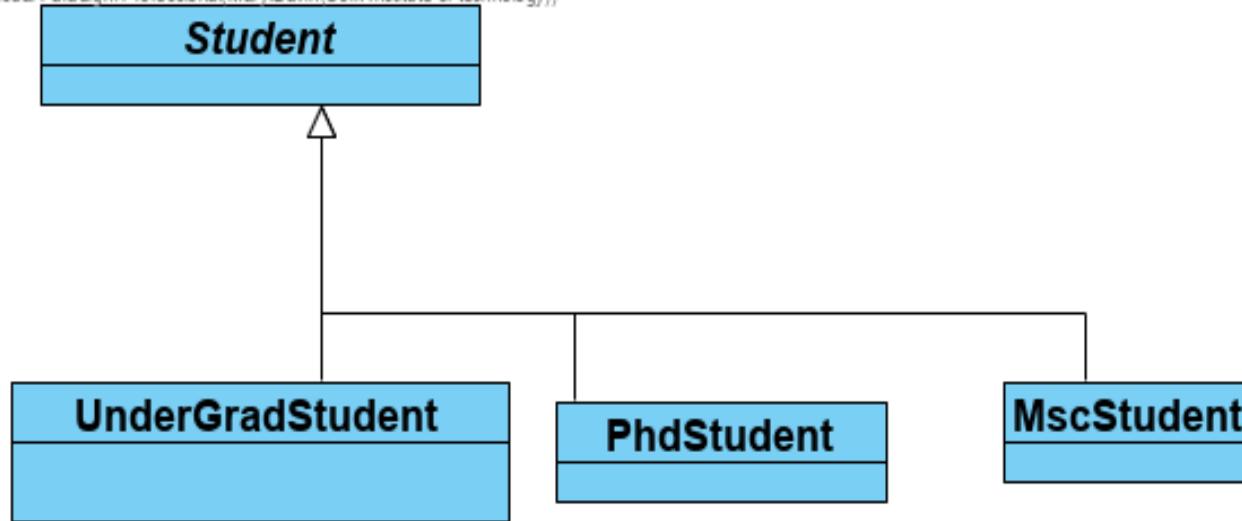
The solution: Declare Student as abstract.

Why have the Student class in the first place?

- A common implementation of common aspects of all students. (e.g. setLogin() and getLogin())
- A place holder in my hierarchy that corresponds to a significant concept in my problem domain
- To handle all students independently of their subclass using type Student and polymorphism.

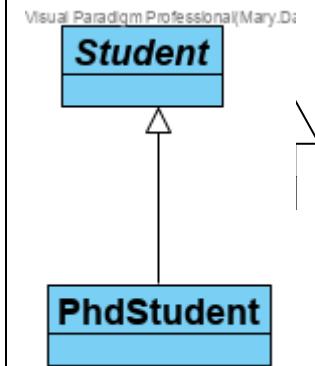
# EXAMPLE 1 SOLUTION

Visual Paradigm Professional (Mary.Davin/Cork Institute of technology))



# ABSTRACT CLASSES IN JAVA

```
public abstract class Student {  
  
    protected String login, department, name;  
  
    public Student() {  
        login = ""; department = ""; name = "";  
    }  
  
    public void setLogin(String login) {  
        this.login = new String(login);  
    }  
  
    public String getLogin() {  
        return new String(login);  
    }  
}
```



PhdStudent is said  
to be a **concrete class**

```
public class PhdStudent extends Student{  
    private String supervisor;  
  
    public void setSupervisor(String login) {  
        ...  
    }  
}
```

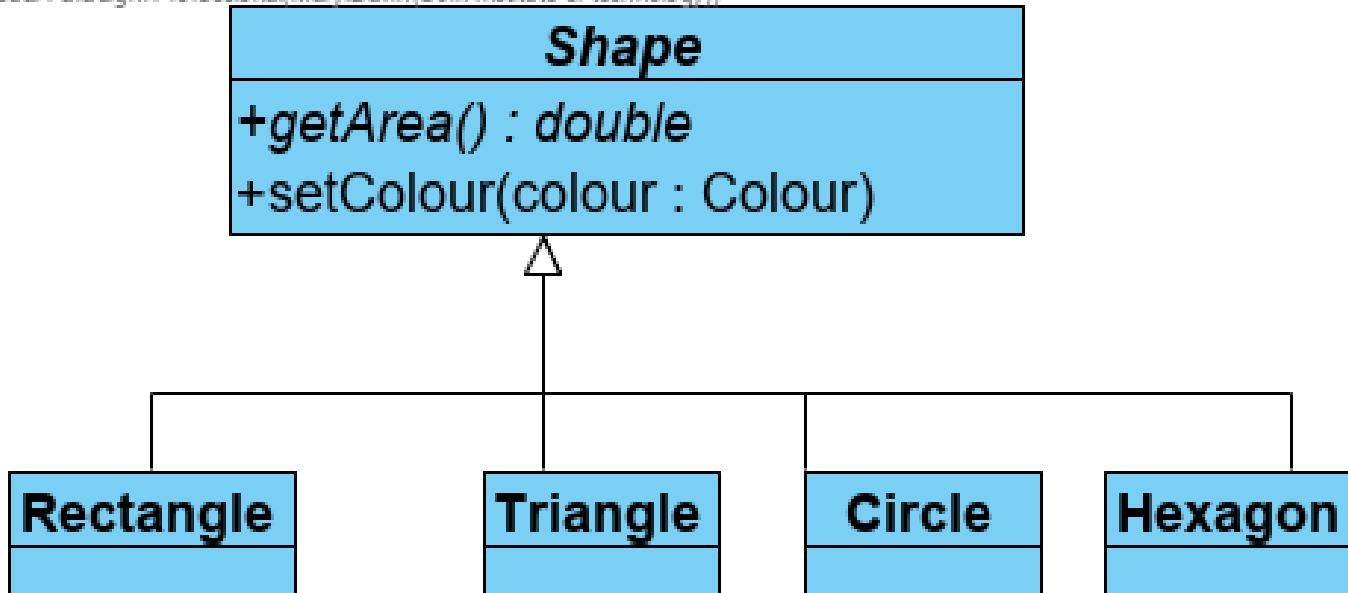
# EXAMPLE 2

- The Problem
  - How do we calculate the area of an arbitrary shape?
  - We cannot allow Shape objects, because we cannot provide a reasonable implementation of getArea();
- The Solution
  - So we declare the Shape to be an abstract class.
  - Furthermore, we declare getArea() as an abstract method because it has no implementation
- Why have the Shape class in the first place?
  - Same reasons as for Student: a common implementation, a placeholder in the hierarchy and polymorphism.
  - Plus that we want to force all shapes to provide an implementation for getArea();



# EXAMPLE 2 SOLUTION

Visual Paradigm Professional(Mary.Darin(Cork Institute of technology))

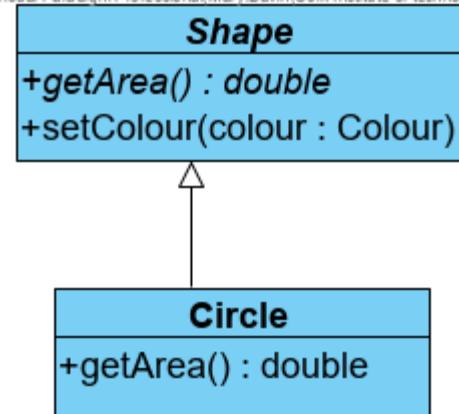


# ABSTRACT METHODS IN JAVA

```
public abstract class Shape {  
    final static int BLACK = 0;  
    private Colour colour;  
  
    public Shape() {}  
  
    public void setColour(Colour theColour) {  
        this.colour = theColour;  
    }  
  
    public abstract double getArea();  
}
```

Abstract methods  
have no body

Visual Paradigm Professional/Mary Davin/Cork Institute of technology



```
public class Circle extends Shape {  
    final static double PI = 3.1419;  
    private int radius;  
  
    public Circle(int r) {  
        radius = r;  
    }  
  
    public double getArea() {  
        return (radius^2)*PI;  
    }  
}
```

If Circle did not implement getArea() then it would have to be declared abstract too!

# ABSTRACT CLASSES

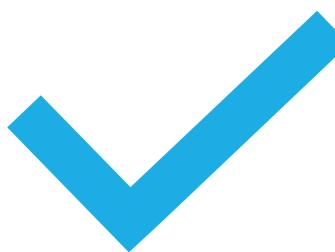
What are the differences between both examples?

In Example 1

- I **choose** to declare Student abstract because I think it is convenient to prevent the existence of plain Students

In Example 2

- I **must** declare Shape abstract because it lacks an implementation for getArea();



# USING ABSTRACT CLASSES

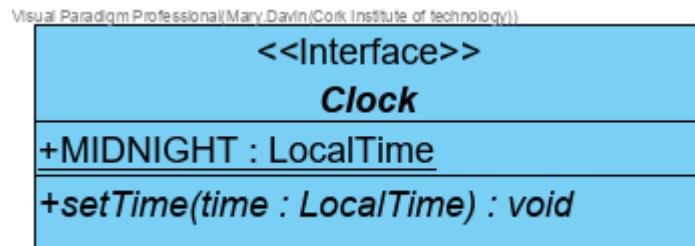
```
// Shape s = new Shape(); // ERROR  
Shape s = new Circle(4); // Ok  
double area = s.getArea(); // Ok – Remember polymorphism?  
Circle c = new Circle(3); // Ok  
c.setColour(Colour.GREEN); // Ok  
area = c.getArea(); // Ok
```

Class `Shape` cannot be instantiated (it provides a partial implementation)

Abstract methods can be called on an object of apparent type `Shape` (they are provided by `Circle`) (**Polymorphism**)

# INTERFACES

An interface is a set of methods and constants that is identified with a name.



They are similar to abstract classes

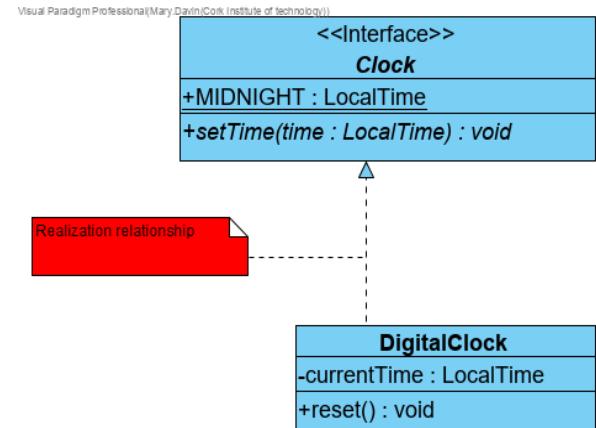
- You cannot instantiate interfaces
- An interface introduces types
- But, they are completely abstract (no implementation)

Classes and abstract classes realize or implement interfaces.

- They must have (at least) all the methods and constants of the interface with public visibility

# EXAMPLE: CLOCK INTERFACE

```
interface Clock {  
    public static final LocalTime MIDNIGHT = new LocalTime(0, 0, 0,0);  
  
    void setTime(Time t);  
}  
  
class DigitalClock implements Clock {  
  
    private LocalTime currentTime;  
  
    public DigitalClock() {reset();}  
  
    public void setTime(LocalTime t) {currentTime = new LocalTime(t);}  
  
    public void reset() {setTime(MIDNIGHT);}  
}
```



# INTERFACE HIERARCHIES

Interfaces can extend each other

```
interface Clock {  
    LocalTime MIDNIGHT = new LocalTime(0, 0, 0,0);  
    void setTime(Time t);  
}
```

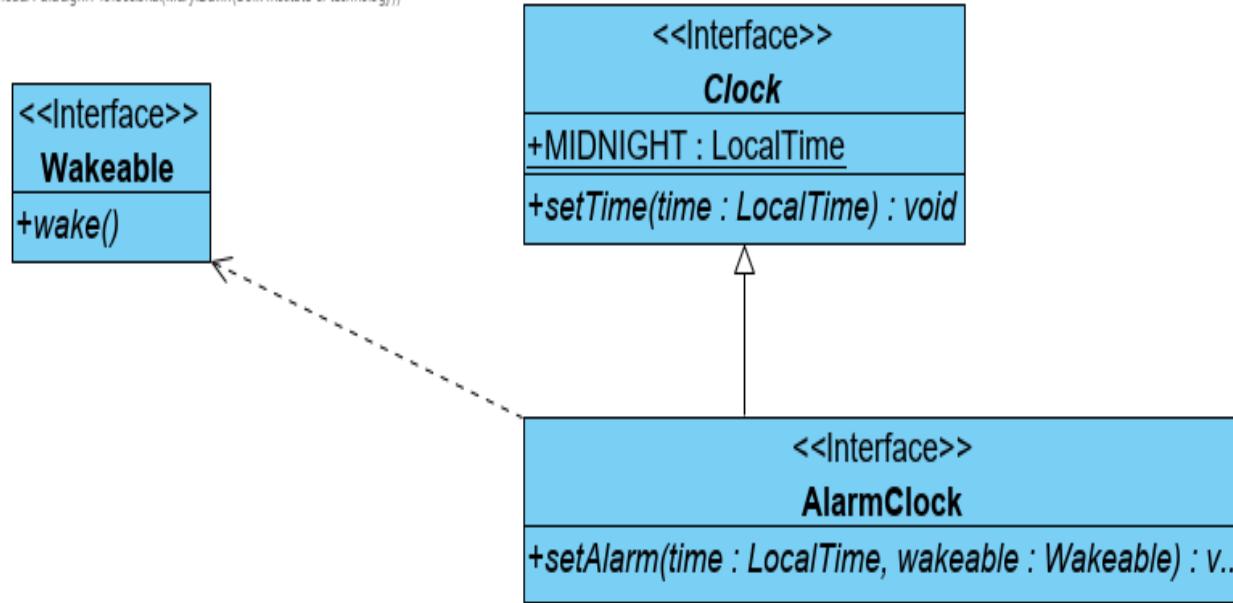
```
interface AlarmClock extends Clock {
```



```
    void setAlarm(LocalTime t, Wakeable w);  
}
```

# INTERFACE HIERARCHIES

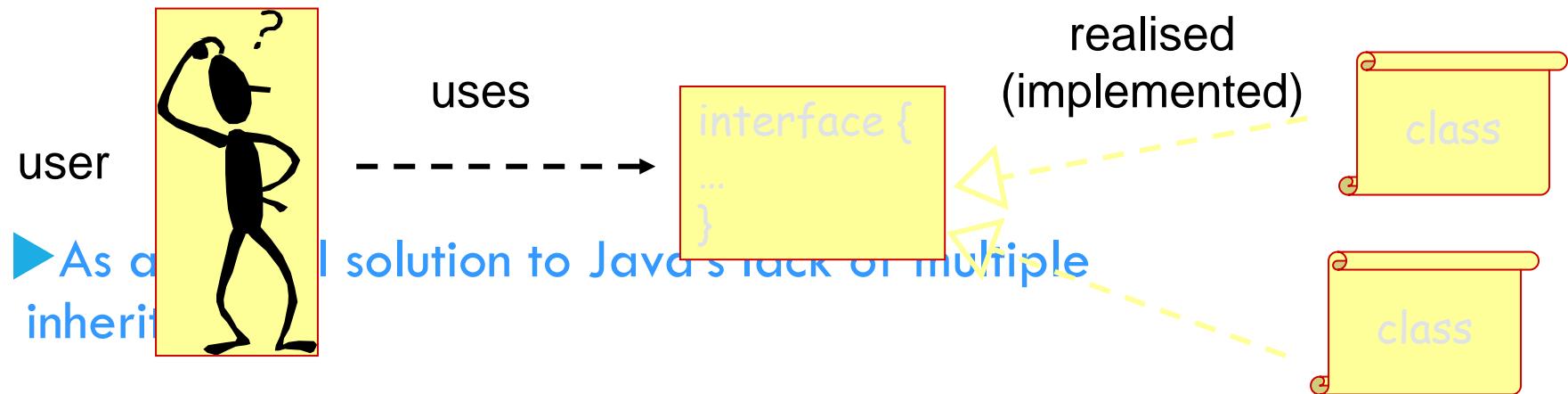
Visual Paradigm Professional(Mary.Davin(Cork Institute of technology))



An *interface can extend another interface. This is modelled using generalization / specialization relationship. An interface can also depend on another interface. In the example `setAlarm` method requires a `wakeable` argument.*

# WHY USE INTERFACES?

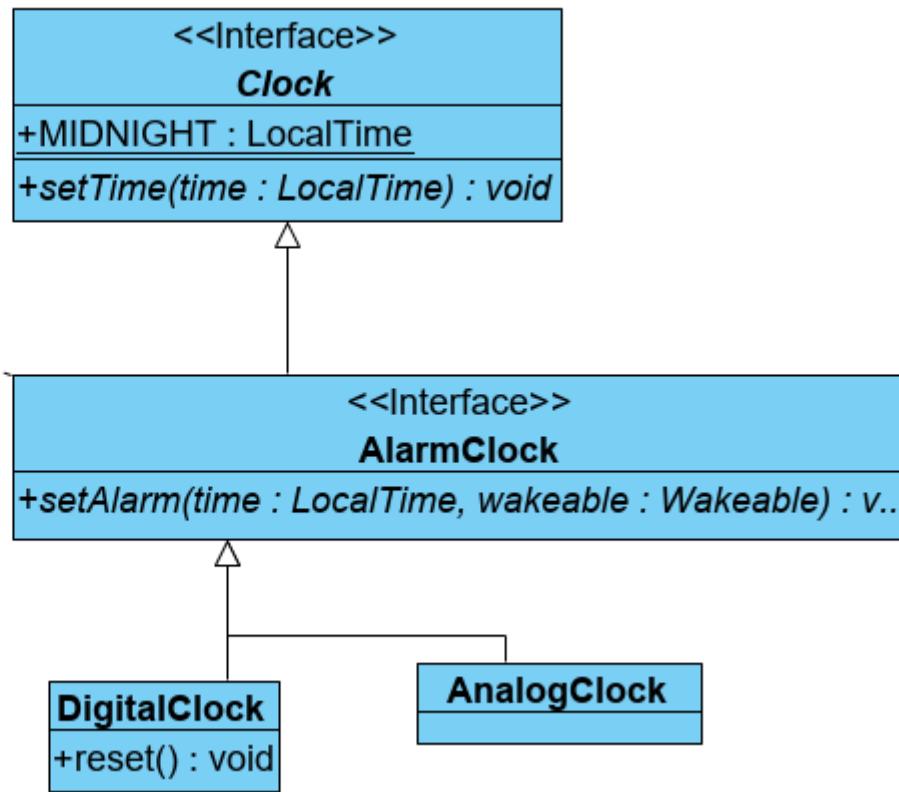
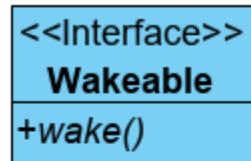
- ▶ To **separate (decouple)** the specification available to the user from implementation
- ▶ I can use any class that implements the interface through the interface type (i.e. polymorphism)



- ▶ As a solution to Java's lack of multiple inheritance

# DECOUPLING: ALARM CLOCK EXAMPLE (1/3)

Visual Paradigm Professional(Mary.Davin(Cork Institute of technology))



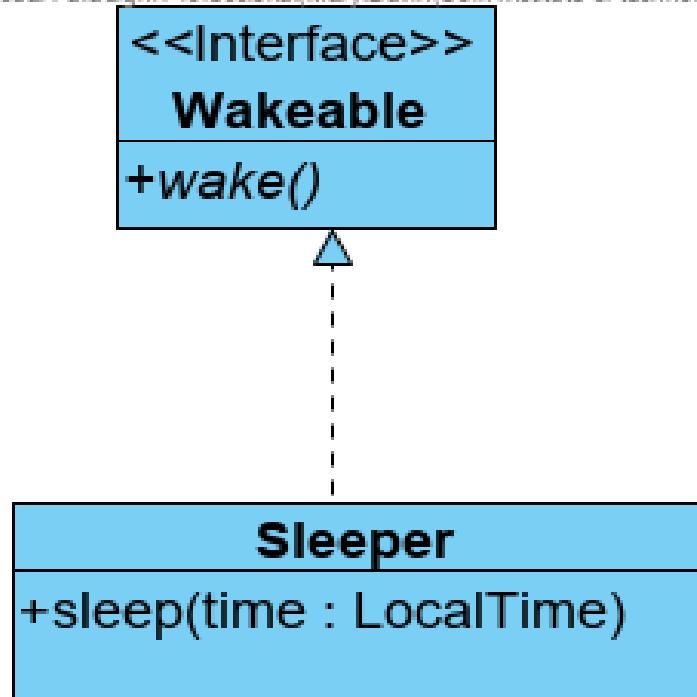
# DECOUPLING: ALARM CLOCK EXAMPLE (2/3)

```
class Sleeper implements Wakeable {  
    private boolean sleeping;  
  
    public Sleeper() {  
        sleeping = false;  
    }  
  
    public void sleep() {  
        if (!sleeping) {  
            sleeping = true;  
            System.out.println("Yawn... Time for a nap!...Zzzz...");  
        }  
    }  
  
    public void wake() {  
        if (sleeping) {  
            sleeping = false;  
            System.out.println("What? Is it time to wake up?");  
        }  
    }  
  
    public boolean sleeping() {  
        return sleeping;  
    }  
}
```

```
interface Wakeable {  
    void wake();  
}
```

# EXAMPLE 2

Visual Paradigm Professional | Mary Davin | Cork | Institute of technology |



# DECOUPLING: ALARM CLOCK EXAMPLE (3/3)

```
public static void main(String [] args) {  
  
    AlarmClock a = new DigitalClock();  
    AlarmClock b = new AnalogueClock();
```

```
Sleeper me = new Sleeper();  
Sleeper you = new Sleeper();
```

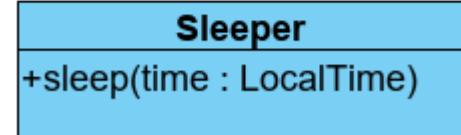
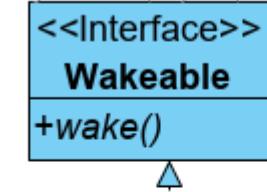
```
me.sleep();  
you.sleep();
```

```
a.setAlarm(new LocalTime(7, 0, 0,0), me); //Wake up at 7am,  
//plenty of time before  
//lecture  
b.setAlarm(new LocalTime(10, 50, 0,0), you);  
//Wake up just before  
//end of lecture
```

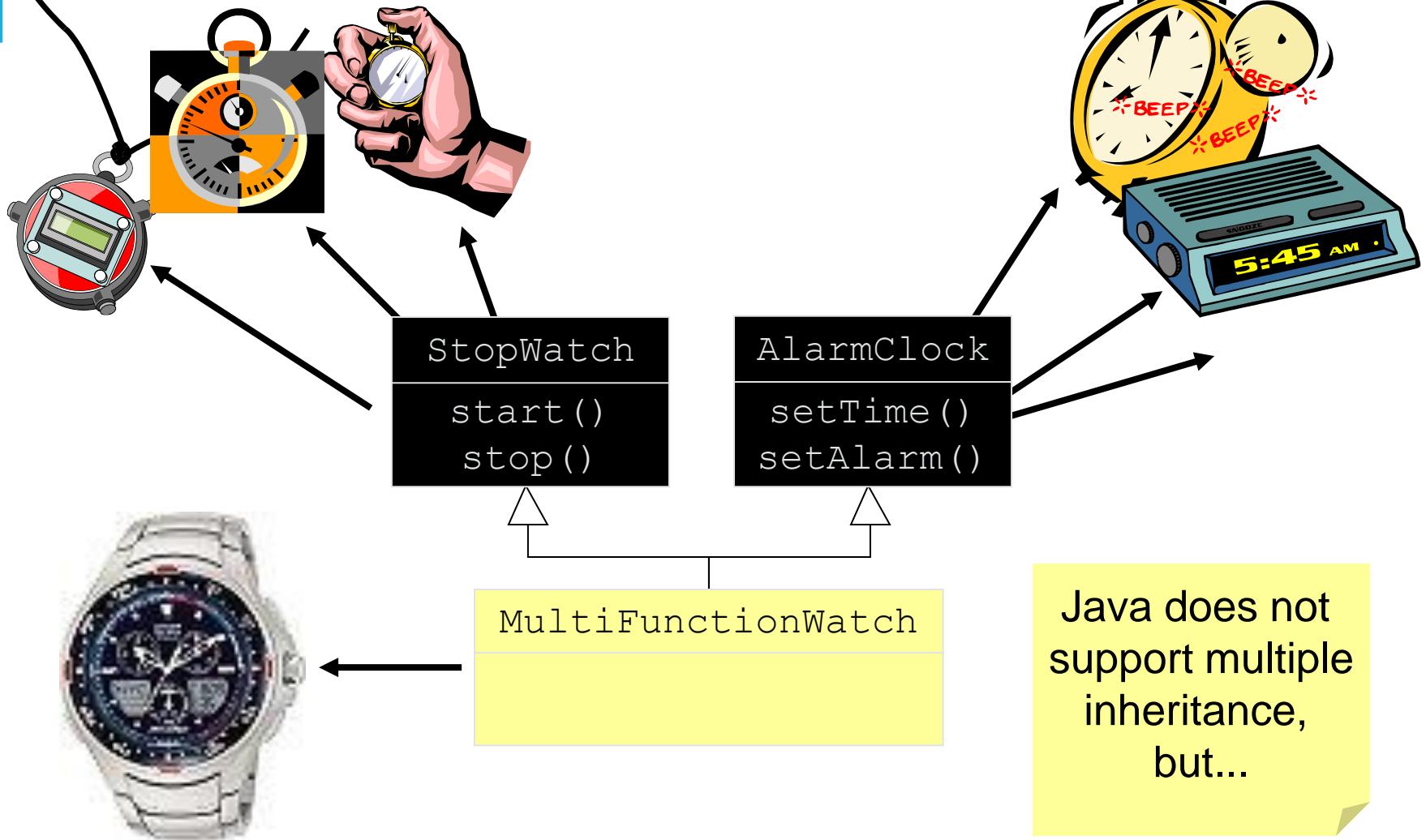
```
}
```

Polymorphism at work.  
Using a digital clock as in  
implementation for  
AlarmClock

Visual Paradigm Professional/Mary.Davin/Cork Institute of technology



# MULTIPLE INHERITANCE



# MULTIPLE INTERFACES

interface  
**StopWatch**

+start()  
+stop():

interface  
**AlarmClock**

+setTime()  
+setAlarm():

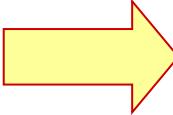
Q: Why is this  
not the same  
as multiple  
inheritance?

**MultiFunctionWatch**

A: There is no  
implementation  
to inherit

# CONFLICT RESOLUTION RULES

Classes can implement multiple interfaces



Name conflicts: multiple methods with the same name

## ■ Name conflict resolution:

- different signatures: overloading
- same signature and return type: same method
- same signature and different return type: compile error

# ABSTRACT CLASSES VS. INTERFACES

- ▶ Can have data fields
- ▶ Methods may have an implementation
- ▶ Classes and abstract classes **extend** abstract classes.
- ▶ Class cannot extend multiple abstract classes
- ▶ Substitution principle is assumed
- ▶ Can only have constants
- ▶ Methods have **no** implementation
- ▶ Classes and abstract classes **implement** interfaces
- ▶ Interfaces can **extend** multiple interfaces
- ▶ A class can implement multiple interfaces
- ▶ Substitution principle not assumed



# Abstraction Occurrence Pattern

CLASS/OBJECT LEVEL PATTERN

Mary Davin | OO Analysis & Design |

## CONTEXT

- There is a set of related objects (occurrences)
- The members share common information but also differ in important ways.

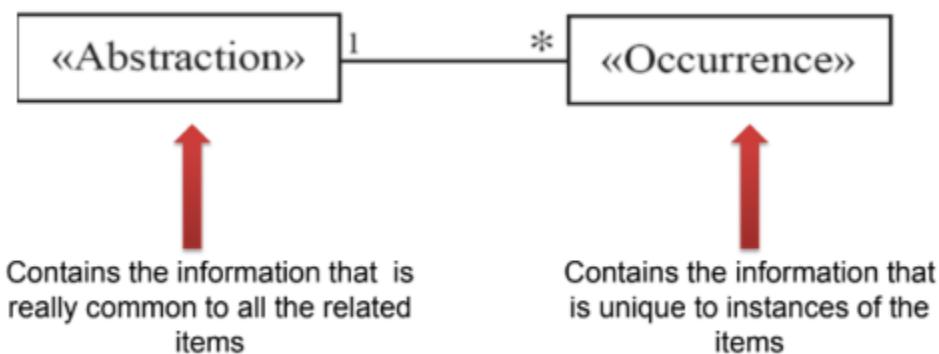
## PROBLEM

- Need to implement the objects without duplicating common information.

## SCENARIO

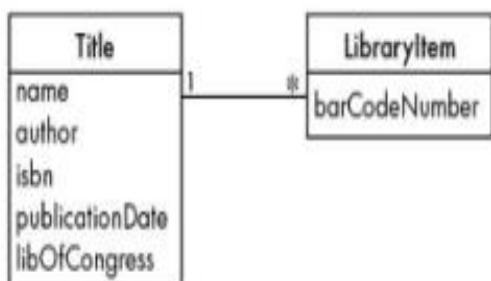
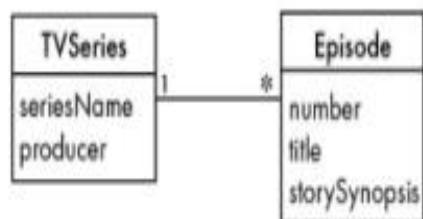
- A library has multiple copies of the same book.
- Author, ISBN, title etc. is all the same... maybe something like an ID is different so each book can be distinguished, but the items are all the same.
- Is there a point or a need of replicating all these objects? No
- This is the abstraction-occurrence problem.

## SOLUTION.



- We create an **Occurrence** class to represent the individual things.
- There is a one to many relationship between the item **Abstraction** class and Occurrence class.
- The **Abstraction class** has all the common things, e.g. Book contains title, author...
- The **Occurrence** class contains the different things, like the barcode inside the book, which is unique.
- For each Book, we can make an instance of the **Occurrence** class, which references the appropriate Book.

## EXAMPLES

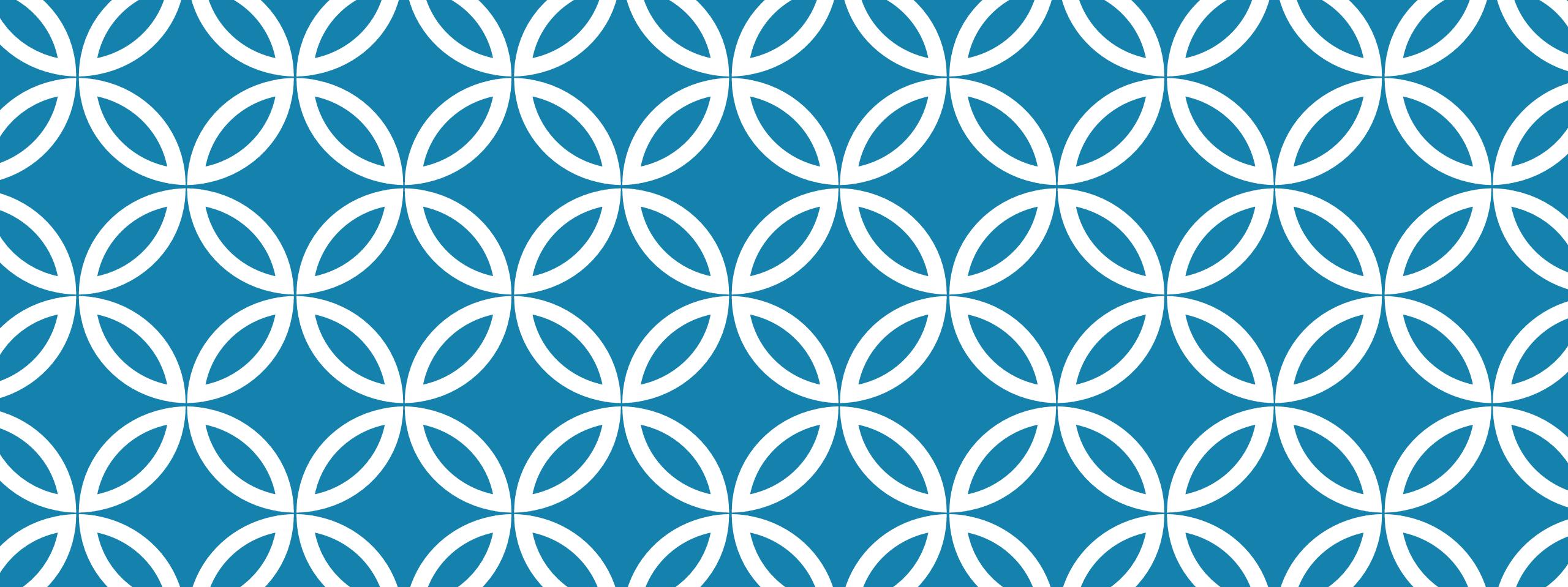


```

public class TVSeries {
    String title;
    String producer;
    Collection<Episode> episodes; //Many episodes of a TV series
}

class Episode {
    TVSeries series; //One series for all episodes
    String title;
    int season;
    int number;
    Collection<Actor> actors;
}

class Actor{
}
  
```



# ASSOCIATIONS BETWEEN CLASSES

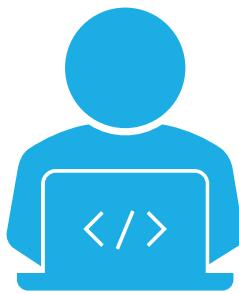
---

# DIFFERENCE BETWEEN ASSOCIATION, COMPOSITION AND AGGREGATION

In Object-oriented programming, one object is related to another other to use functionality and service provided by that object.

This relationship between two objects is known as the *association*

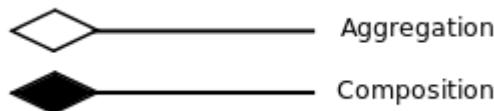
Both **Composition** and **Aggregation** are a form of association between two objects, but there is a **subtle difference between composition and aggregation**, which is also reflected by their UML notation.



# DIFFERENCE BETWEEN COMPOSITION AND AGGREGATION

Composition ----- Filled in diamond

Aggregation ----- unfilled diamond



**Aggregation :** An Aggregation notation between two classes is suitable whenever an instance of class A **holds a collection** of instances of class B (e.g. a List, Array, whatever). The objects can exist independently and which class A object knows about which class B objects can vary over time.

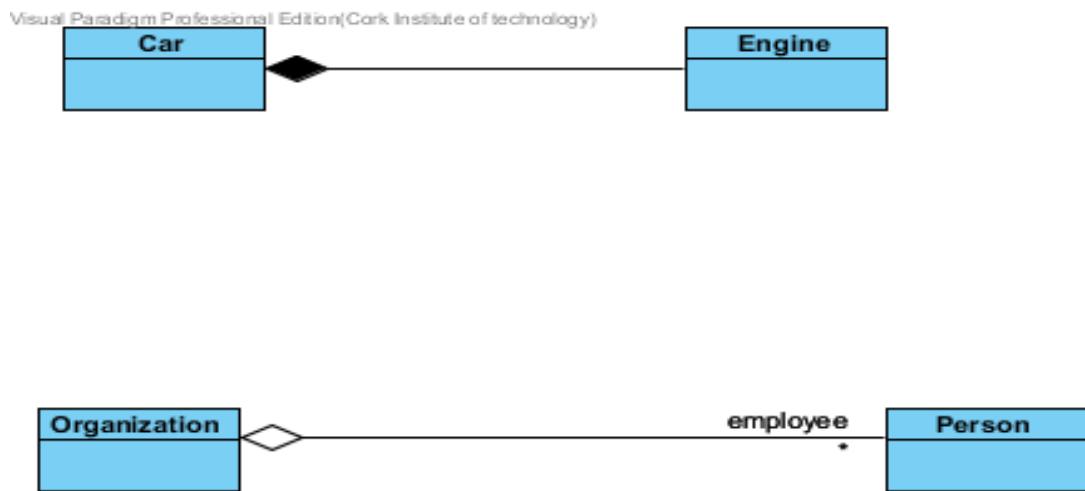
**Aggregation** is used to Model has a relationship

**Composition** represents a part -- whole relationship such that class B is an integral part of class A. This relationship is typically used if objects of class A can't logically exist without having a class B object.

# EXAMPLE

An engine is part of a car.

An organization has employees



# COMPOSITION

We refer to association between two objects as **Composition**, when one class **owns** other class and the other class cannot meaningfully exist, when its owner is destroyed.

**Composition:** Since Engine is-part-of Car, the relationship between them is Composition. Here is how they are implemented between Java classes.

# COMPOSITION MAPPED TO JAVA EXAMPLE

```
public class Car {  
    //final will make sure engine is initialized  
    private final Engine engine;
```

```
public Car(){  
    engine = new Engine();  
}  
}
```

```
class Engine {  
    private String type;  
}
```

# AGGREGATION

**Aggregation:** Since Organization has Person as employees, the relationship between them is Aggregation. Here is how they look like in terms of Java classes

```
public class Organization {  
    private List <Person>employees;  
}
```

```
public class Person {  
    private String name;  
}
```

# AGGREGATION [ 'HAS-A' RELATIONSHIP ]

```
class StereoSystem {  
    private boolean state ;  
    StereoSystem() {}  
    StereoSystem(boolean state) {  
        this.state = state ;  
        System.out.println("Stereo System State: " + (state == true ? "On!" : "Off!")) ;  
    }  
}  
class Car {  
    private StereoSystem s ;  
    Car() {}  
    Car(String name, StereoSystem s) {  
        this.s = s ;  
    }  
    public static void main(String[] args) {  
        StereoSystem ss = new StereoSystem(true) ; // true(System is ON.) or false (System is OFF)  
        Car c = new Car("BMW", ss) ;  
    }  
}
```

# CODE REVERSE ENGINEERED



# COMPOSITION [ 'CONTAINS' RELATIONSHIP ]

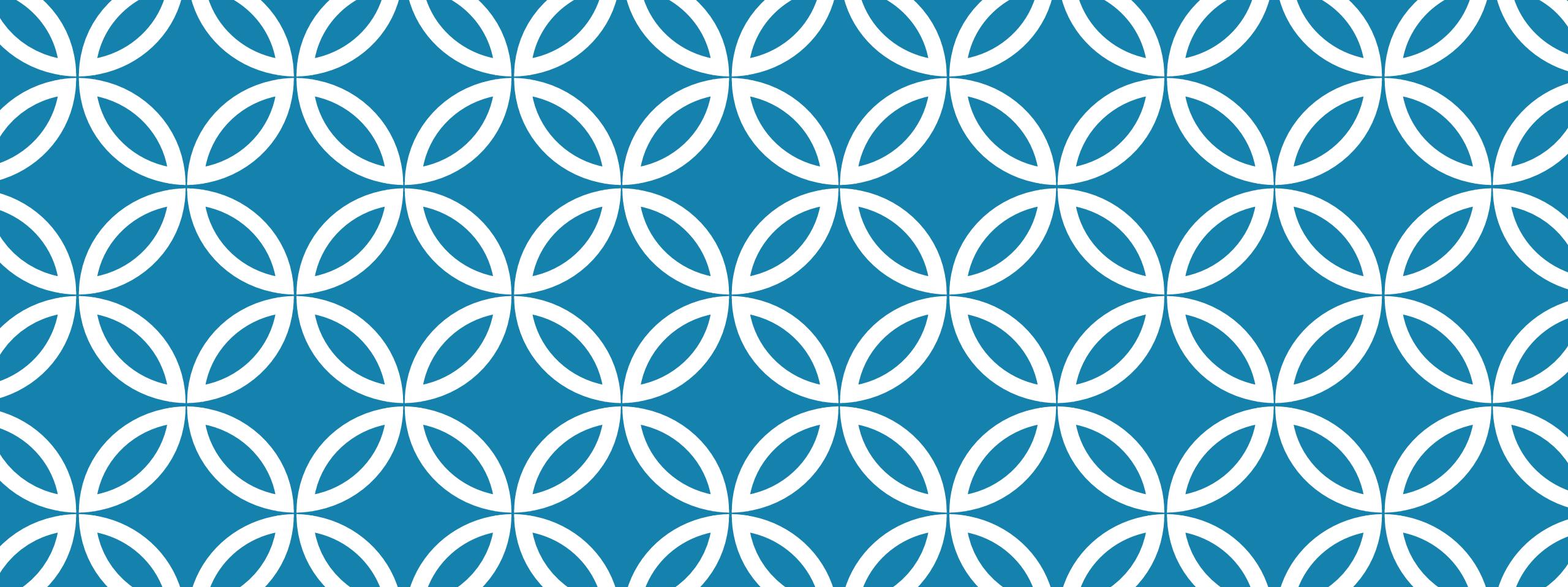
```
import java.util.Date ;
class Piston {
    private Date pistonDate ;
    Piston() {
        pistonDate = new Date() ;
        System.out.println("Manufactured Date :: " +
pistonDate) ;
    }
}
class Engine {
    private Piston piston ;
    Engine() {
        piston = new Piston() ;
    }
    public static void main(String[] args) {
        Engine engine = new Engine() ;
    }
}
```

# CODE REVERSE ENGINEERED



# JAVA COMPOSITION BENEFITS

- Benefit of using composition in java is that we can control the visibility of other object to client classes and reuse only what we need.
- A Client object will be able to communicate with an engine object but not a piston object.
- If we change anything in the Piston class then only the Car class may need to change to accommodate but a client class will not be affected.



# ASSOCIATION CLASSES

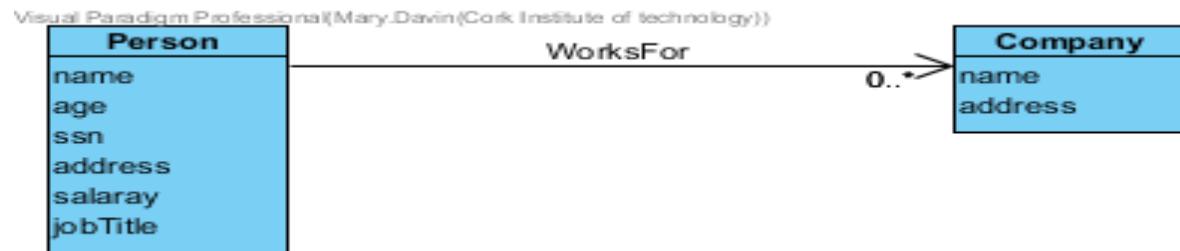
---

# WHAT ARE ASSOCIATION CLASSES

**Association classes** allow you to add attributes, operations, and other features to associations.

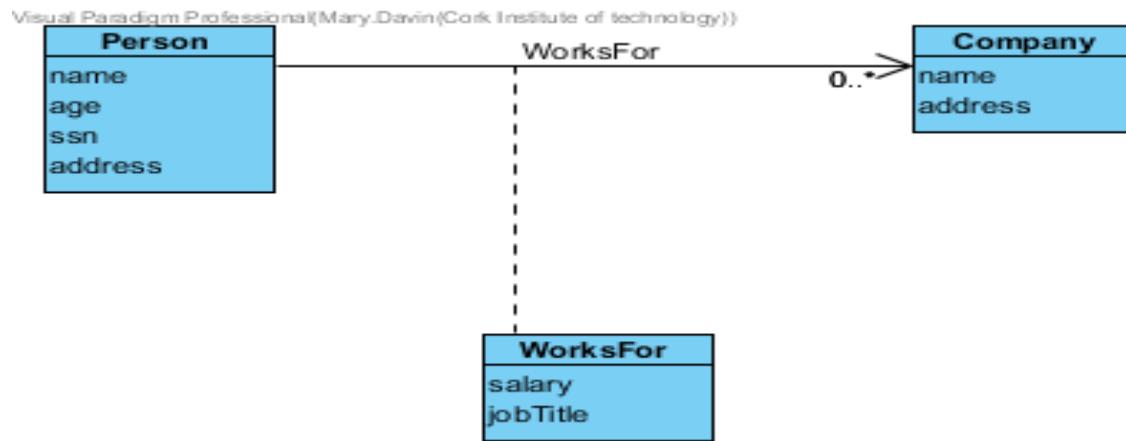
# LINK ATTRIBUTES

How to represent salary and job title?



Why not this?

Salary and job title are **not** properties of a person[ link attributes only solution]

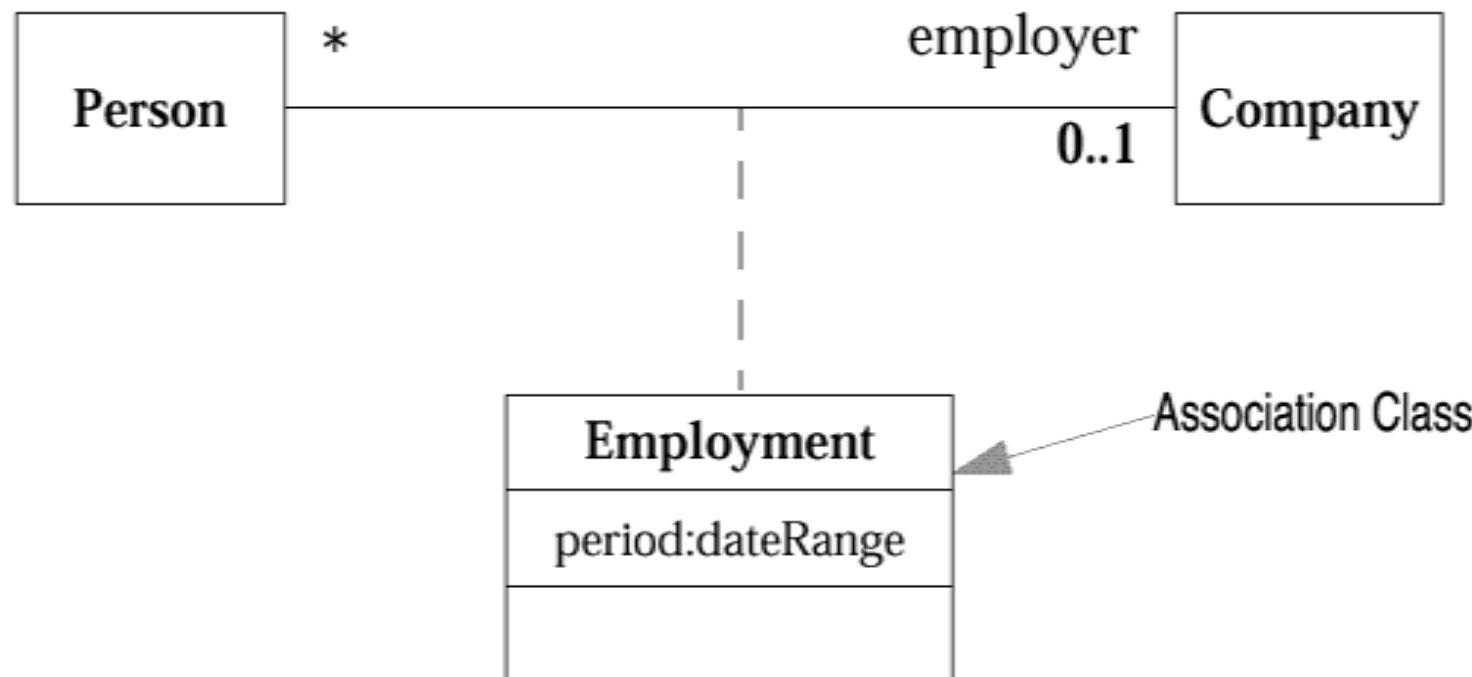


# PURPOSE OF ASSOCIATION CLASS IN EXAMPLE

We can see from the diagram in next slide that a Person may work for a single Company. We need to keep information about the period of time that each employee works for each Company.

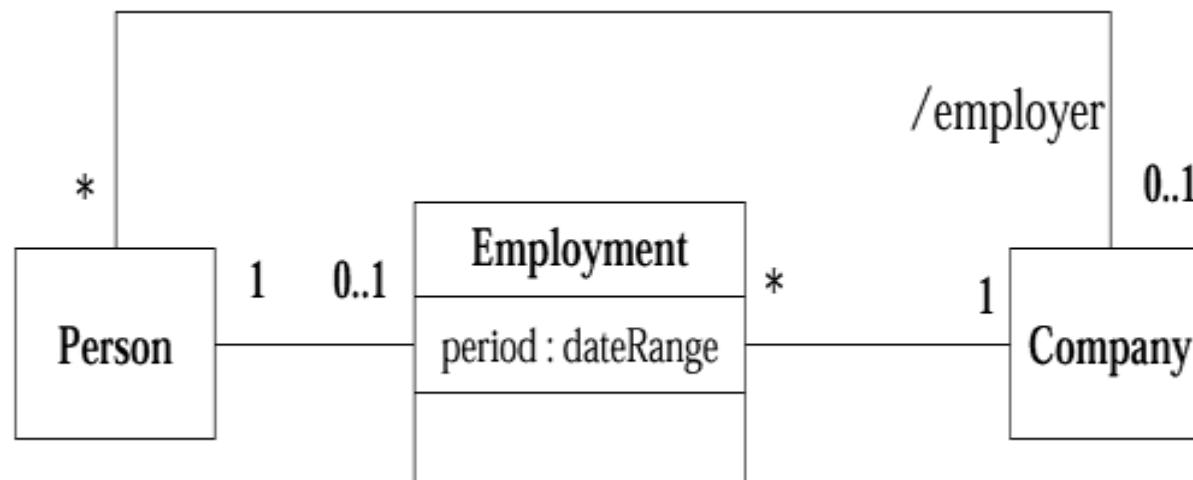
We can do this by adding a `dateRange` attribute to the association. We could add this attribute to the Person class, but it is really a fact about a Person's relationship to a Company, which will change should the person's employer change.

# EXAMPLE OF AN ASSOCIATION CLASS



# PROMOTING AN ASSOCIATION CLASS TO A FULL CLASS

The following image is another way to represent this information: make Employment a full class in its own right. (Note how the multiplicities have been moved accordingly.) In this case, each of the classes in the original association has a single-valued association end with regard to the Employment class. The "employer" end now is derived, although you don't have to show this.



# BENEFITS OF THE ASSOCIATION CLASS

What benefit do you gain with the association class to offset the extra notation you have to remember?

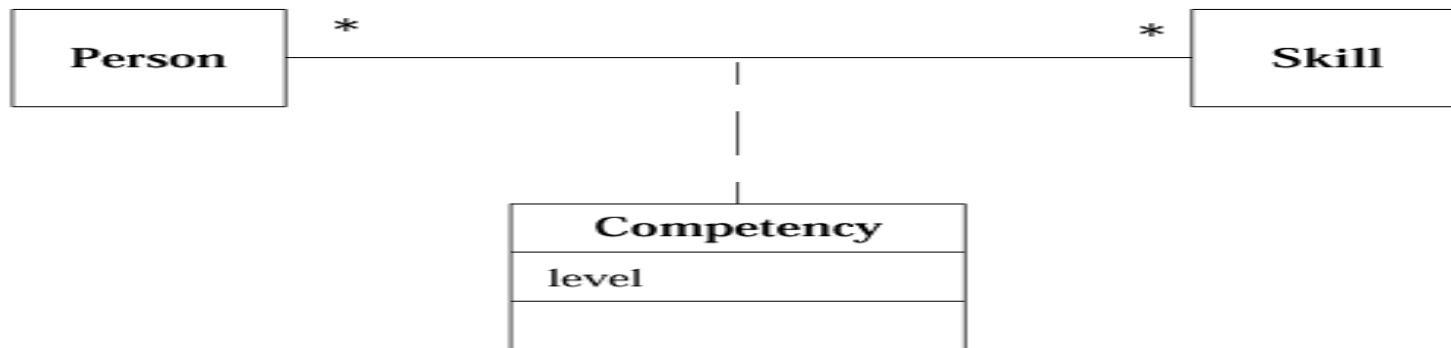
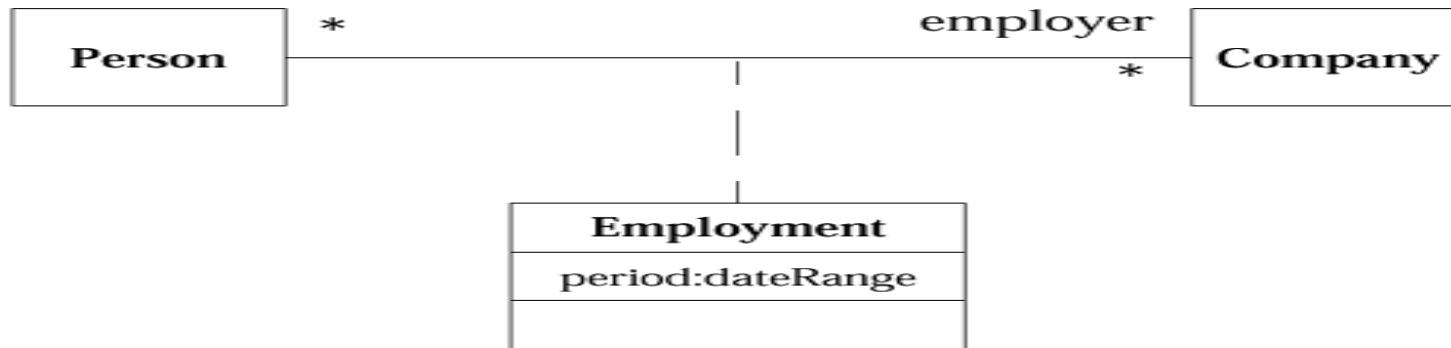
The association class adds an extra constraint, in that there can be only one instance of the association class between any two participating objects.

Take a look at the two diagrams in next slide .

These diagrams have much the same form.

However, we could imagine a Person working for the same Company at different periods of time that is, he or she leaves and later returns. This means that a Person could have more than one Employment association with the same Company over time. With regard to the Person and Skill classes, it would be hard to see why a Person would have more than one Competency in the same Skill; indeed, you would probably consider that an error.

# ASSOCIATION CLASS SUBTLETIES



# ASSOCIATION CLASS SUBTLETIES EXPLAINED

In the UML, only the latter case is legal. You can have only one Competency for each combination of Person and Skill.

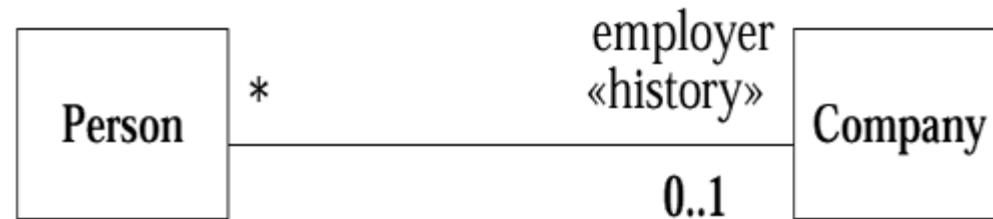
The top diagram in Figure would not allow a **Person** to have more than one **Employment** with the same **Company**. If you need to allow this, you need to make **Employment** a full class.

Many people do not think about it at all and may assume the constraint in some places and not in others. So when using the UML, remember that the constraint is always there.

You often find this kind of construct with historical information

# HISTORY STEREOTYPE FOR ASSOCIATIONS

The model indicates that a Person may work for only a single Company at one time.  
Over time, however, a Person may work for several Companies.



# POSSIBLE INTERFACE FOR PREVIOUS MODEL

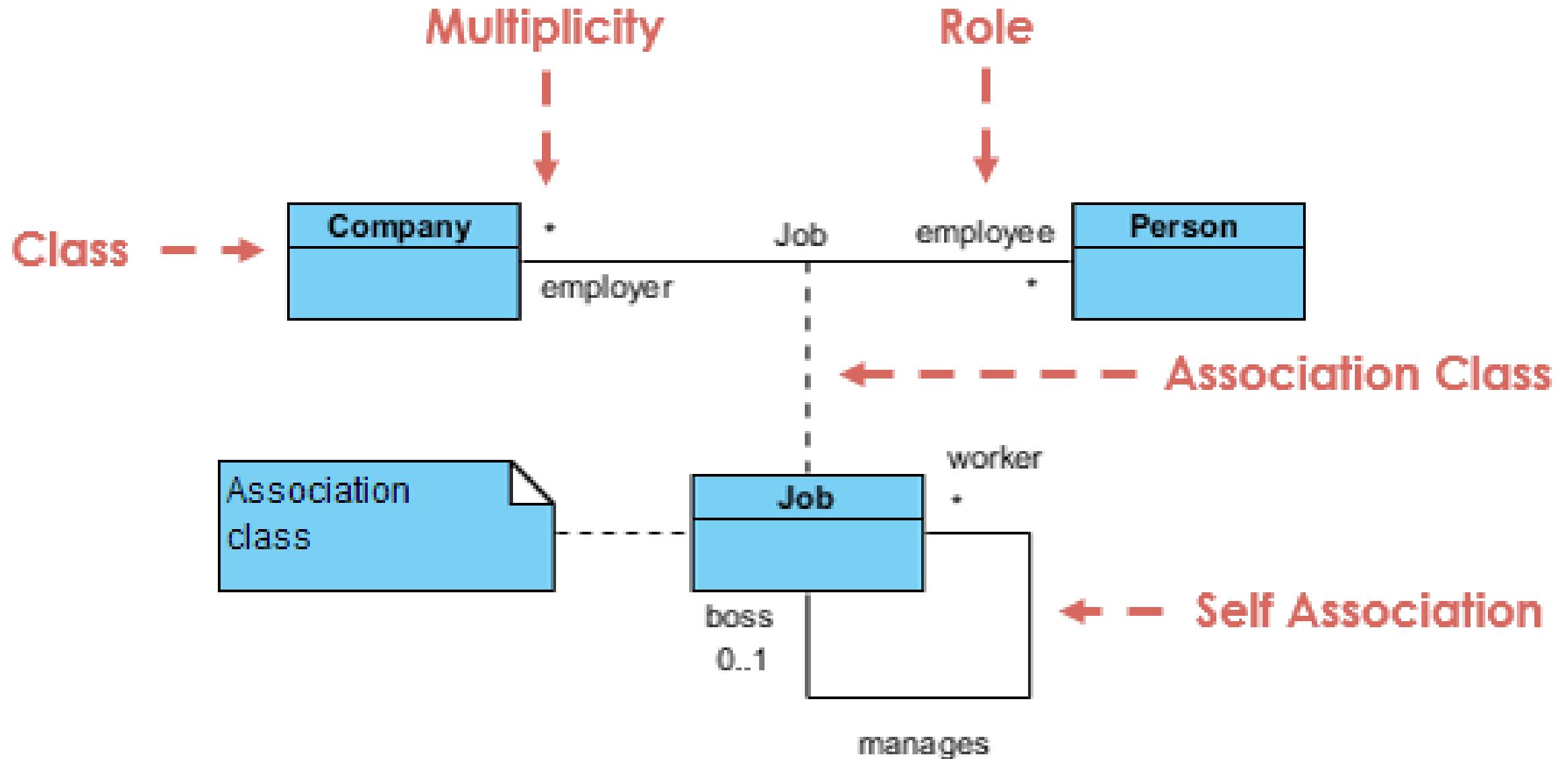
```
class Person {  
    Company employer;  
    //get current employer  
    Company getEmployer();  
  
    //employer at a given date  
    Company getEmployer(Date);  
  
    void changeEmployer(Company newEmployer,  
        Date changeDate);  
  
    void leaveEmployer (Date changeDate);
```

# ASSOCIATION CLASS AND SELF ASSOCIATION

Some people have several jobs, but with different employers.

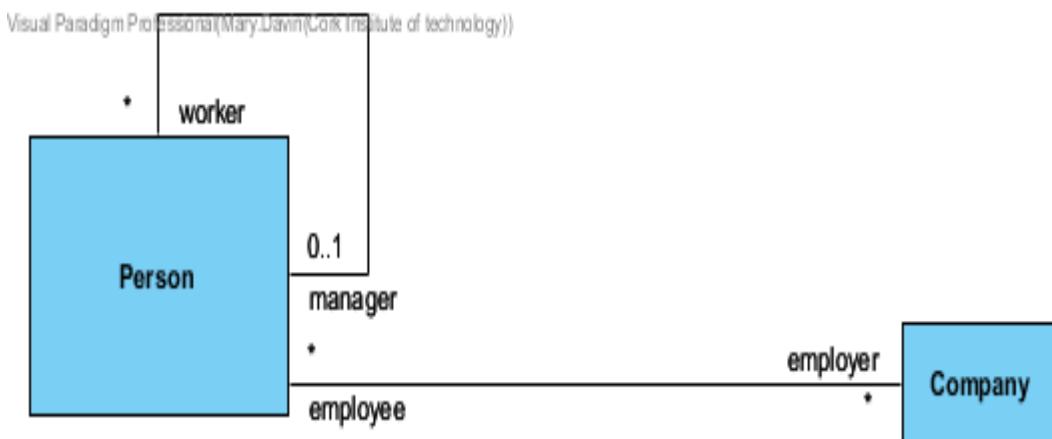
For example, at night John is a security guard at Pub 911, where he earns € 9/hr and his boss is the club's bar tender. In the day he is a programmer for MicroHard, where he earns € 10/hr and his boss is a MicroHard project manager. Where can we store John's salary and boss information?

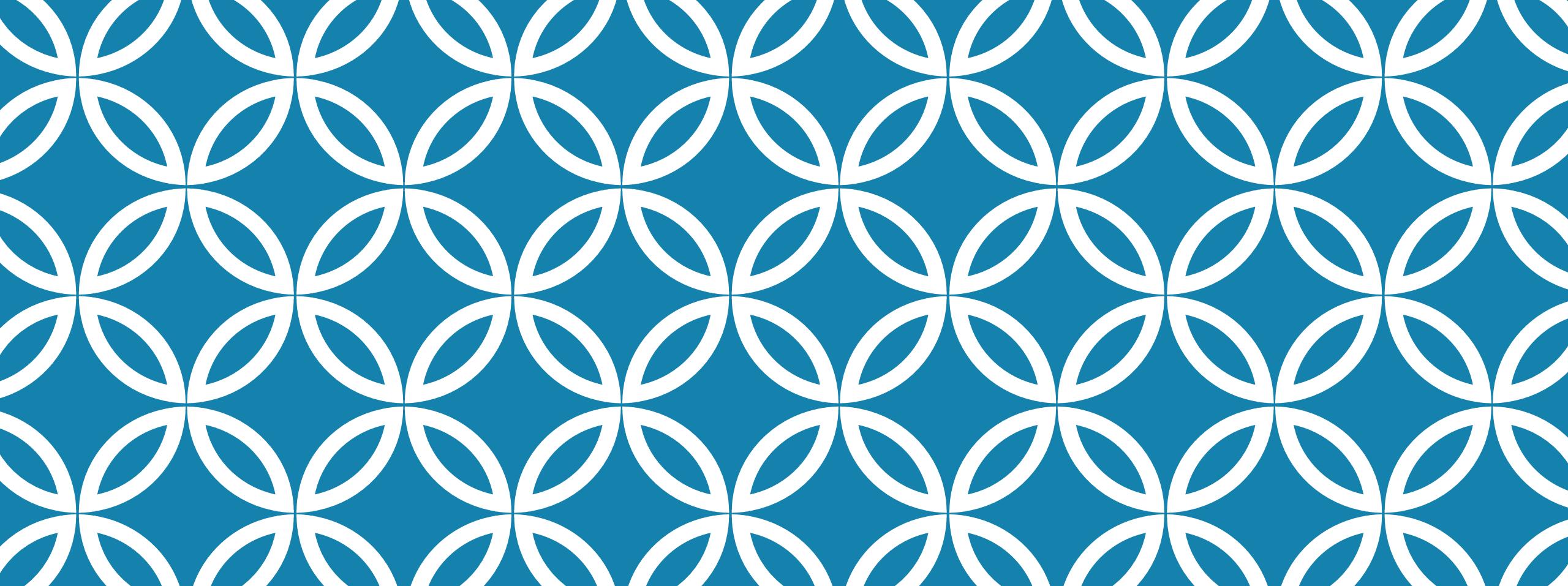
To solve this problem we can use the Job association class. Now John's security guard job and programmer job are association objects: instances of a Job class. Each instance encapsulates a salary, an employer link, and an employee link. We can represent the Boss relationship as a Job self-association. Here is the UML notation:



# SELF ASSOCIATIONS

**Self Association** also called **reflexive association**, that is, If the same class appears twice in an association, the two instances do not have to be the same object, and usually they are not.





# CLASS SCOPE OPERATIONS AND ATTRIBUTES

---

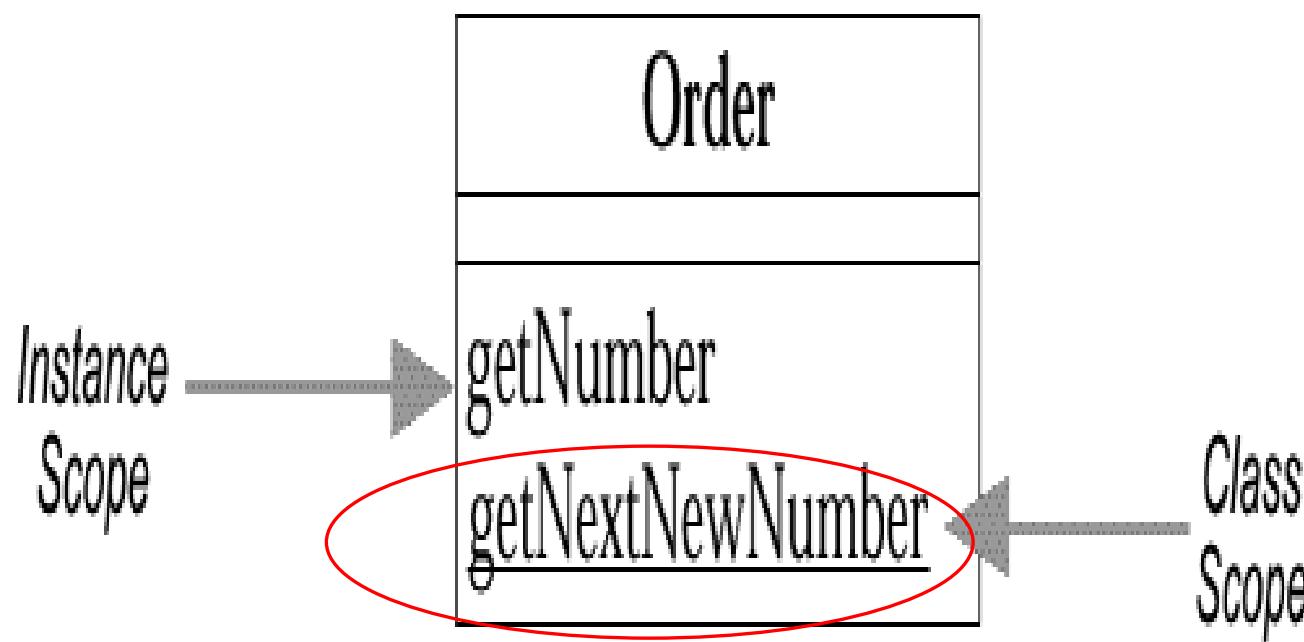
# SCOPE

UML refers to an operation or an attribute that applies to a class, rather than an instance, as having **class scope**.

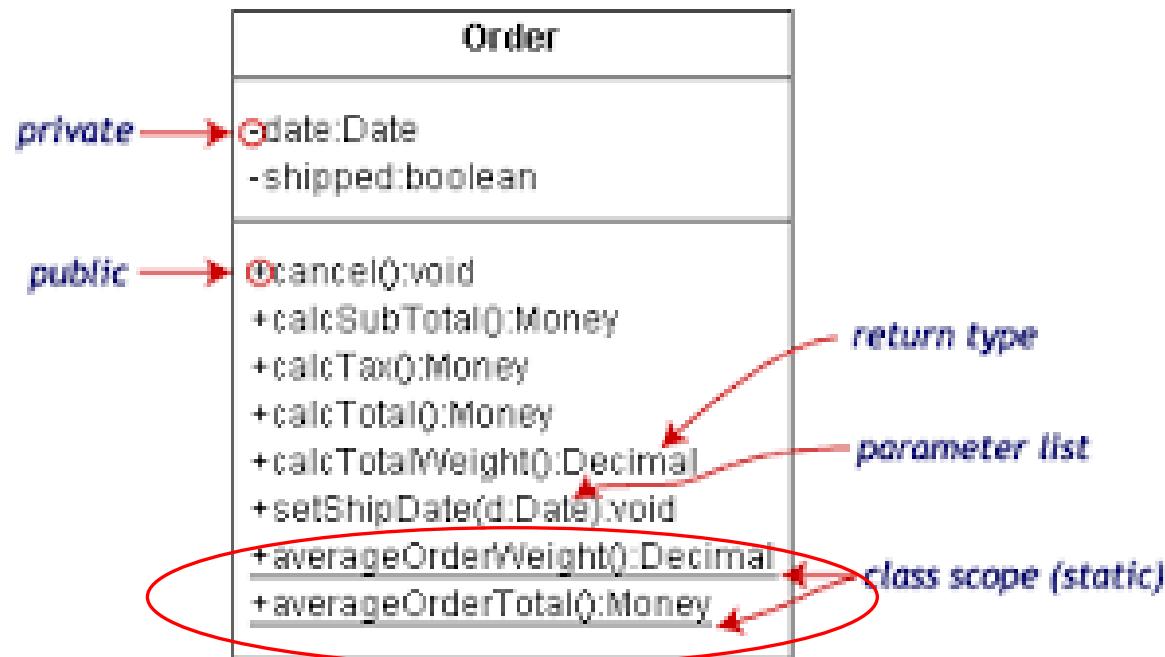
This is equivalent to **static members** in C++ or Java.

**Class scope** features are underlined on a class diagram

# HOW TO MODEL CLASS SCOPE EXAMPLE



# CLASS SCOPED OPERATION EXAMPLE



# EXAMPLE OF STATIC VARIABLE IN JAVA

```
public class Car {  
    private String name;  
    private String engine;  
    public static int numberOfCars;  
    public Car(String name, String engine)  
    { this.name = name;  
        this.engine = engine;  
        numberOfCars++; } // getters and setters }
```

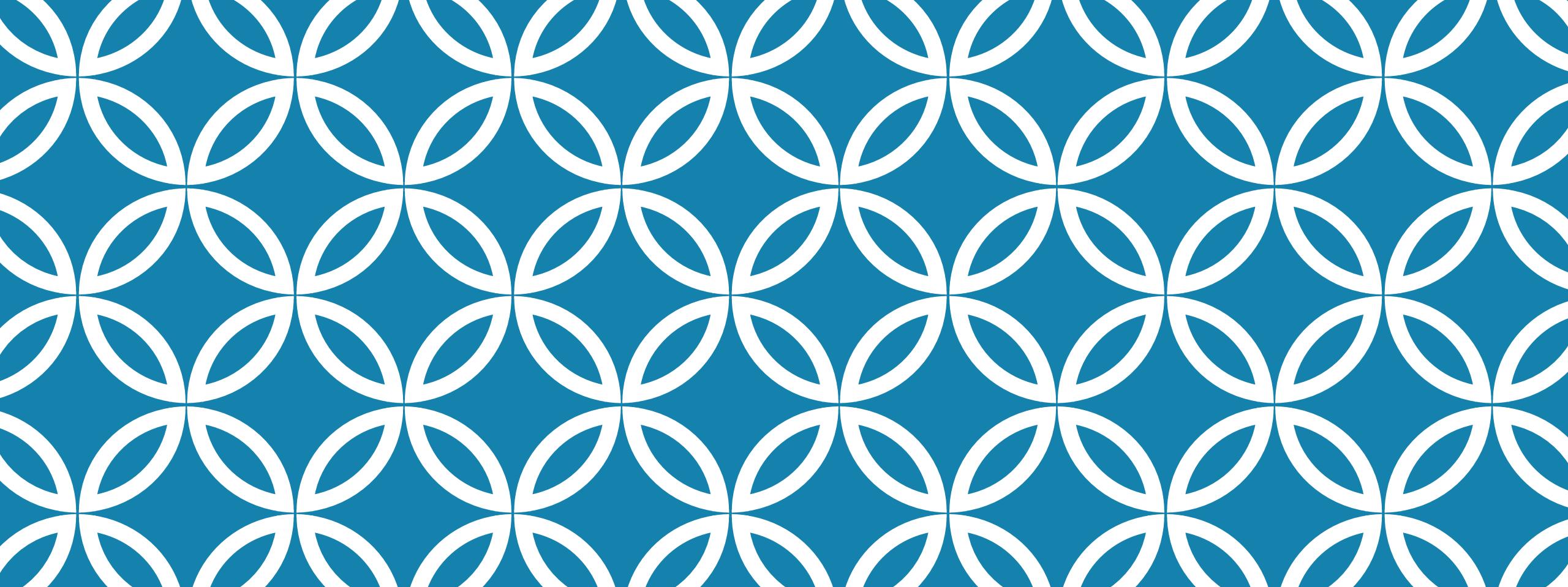
# EXAMPLE OF A STATIC METHOD IN JAVA

```
public class A {  
  
    static void doSomething() {  
        // this is a static method  
    }  
  
    void doOtherThing() {  
        // this is a non-static method  
    }  
}  
  
A a1 = new A();  
a1.doOtherThing();  
  
A.doSomething()  
}
```

# USAGE OF STATIC VARIABLES AND METHODS

Provide constants and utility methods, such as `Math.PI`, `Math.sqrt()` and `Integer.parseInt()` which are public static and can be used directly through the class without construction instances of the class.

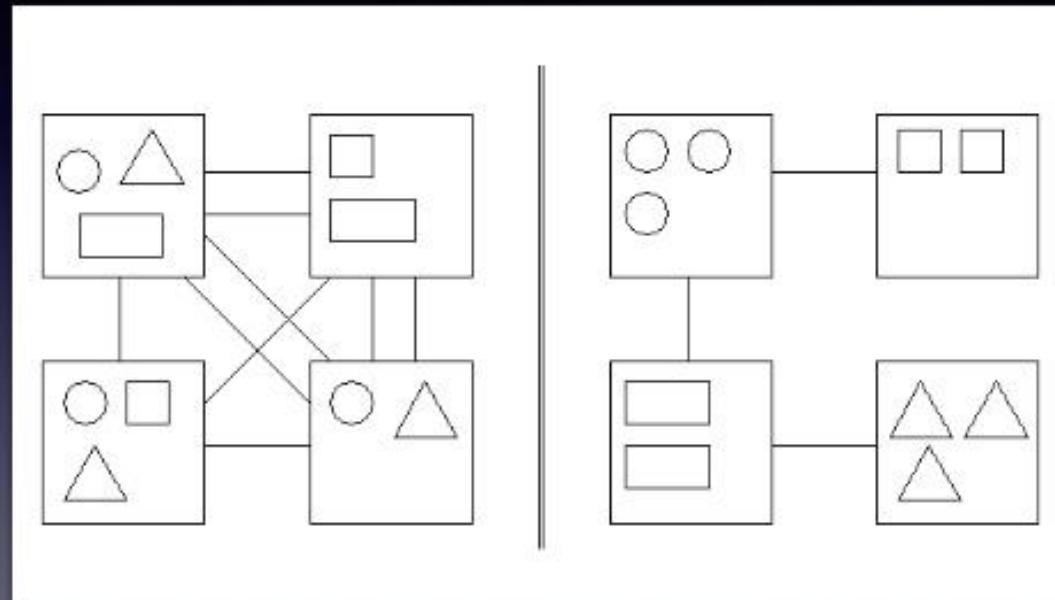
Provide a “global” variable, which is applicable to all the instances of that particular class, for purposes such as counting the number of instances.



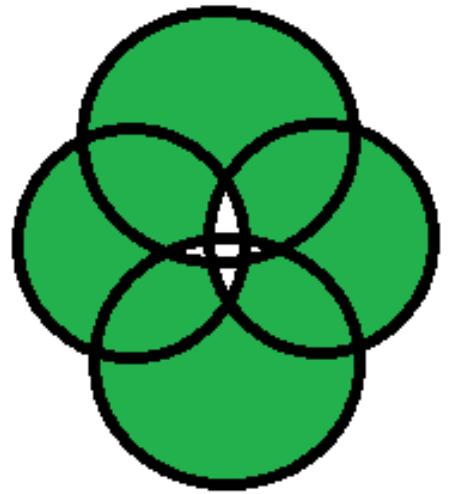
# OO DESIGN PRINCIPLES: COHESION AND COUPLING



# Cohesion & Coupling

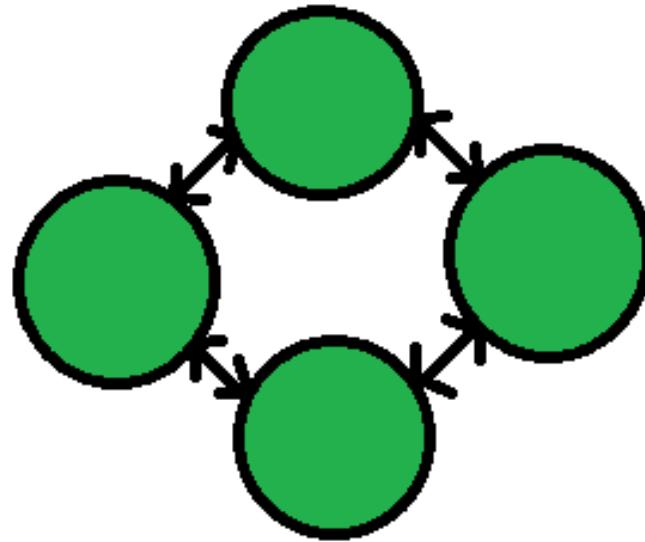


Goal: high cohesion, low coupling



**Tight coupling:**

1. More Interdependency
2. More coordination
3. More information flow



**Loose coupling:**

1. Less Interdependency
2. Less coordination
3. Less information flow

# PURPOSE OF PRINCIPLES

**Cohesion** and **Coupling** deal with the quality of an OO design.

Generally, good OO design should be loosely coupled and highly cohesive.

Lots of the design principles and design patterns which have been created are based on the idea of “Loose coupling and high cohesion”.

# THE AIM OF THE DESIGN SHOULD BE TO MAKE THE APPLICATION:

easier to develop

easier to maintain

easier to add new features

less Fragile.

# COUPLING:

In object oriented design, **Coupling** refers to the degree of direct knowledge that one element has of another. **There are two types of coupling:**

**Tight coupling.**

**Loose coupling.**

# TIGHT COUPLING

In general, **Tight coupling** means the two classes often change together. In other words, if A knows more than it should about the way in which B was implemented, then A and B are tightly coupled.

# EXAMPLE OF TIGHT COUPLING

if in the Topic class the **understand()** method changes to **gotit()** method then you have to change the **startReading()** method as it will call **gotit()** method instead of calling **understand()** method.

```
// Java program to illustrate  
// tight coupling concept  
  
class Subject {  
  
    Topic t = new Topic();  
  
    public void startReading()  
    {  
        t.understand();  
    }  
}  
  
class Topic {  
  
    public void understand()  
    {  
        System.out.println("Tight coupling concept");  
    }  
}
```

# EXAMPLE OF TIGHT COUPLING

In example, there is a strong **inter-dependency** between both the classes. If there is any changes in Box class then they will reflects in the result of Class Volume.

```
// Java program to illustrate  
// tight coupling concept  
  
class Volume  
{  
  
    public static void main(String args[])  
{  
  
        Box b = new Box(5,5,5);  
  
        System.out.println(b.volume);  
    }  
  
    class Box  
{  
  
        public int volume;  
  
        Box(int length, int width, int height)  
        {  
  
            this.volume = length * width * height;  
        }  
    }  
}
```

# LOOSE COUPLING

In simple words, **loose coupling** means they are mostly independent. If the only knowledge that class A has about class B, is what class B has exposed through its interface, then class A and class B are said to be loosely coupled.

# EXAMPLE OF LOOSE COUPLING

```
public interface Topic
{
    void understand();
}

class Topic1 implements Topic {
    public void understand()
    {
        System.out.println("Got it");
    }
}

class Topic2 implements Topic {
    public void understand()
    {
        System.out.println("understand");
    }
}

public class Subject {
    public static void main(String[] args)
    {
        Topic t = new Topic1();
        t.understand();
    }
}
```

# COHESION

**Cohesion** is used to indicate the degree to which a class has a single, well-focused purpose.

**Coupling** is all about how classes interact with each other, on the other hand cohesion focuses on how single class is designed.

Higher the cohesiveness of the class, better is the OO design.

```
class A  
checkEmail()  
validateEmail()  
sendEmail()  
printLetter()  
printAddress()
```

Fig: Low cohesion

```
class A  
checkEmail()
```

```
class B  
validateEmail()
```

```
class C  
sendEmail()
```

```
class D  
printLetter()
```

Fig: High cohesion

# LOW COHESION EXAMPLE

```
/*
 Low cohesion example
*/
class AllInStaff {
    void getStaffSalary();
    void getStaffDetails();
    void getStaffSalesReport();
}
```

# HIGH COHESION EXAMPLE

```
/*
    High cohesion example
*/
class Accounts {
    void getStaffSalary();
    ...
}

class Personnel {
    void getStaffDetails();
    ...
}

class SalesReporting {
    void getStaffSalesReport();
    ...
}
```

# HIGH COHESION EXAMPLE

```
/**  
 * The Ship Class  
 */  
public class Ship {  
  
    /**  
     * Function – performs the behavior (task) of turning the Ship  
     */  
public void rotate() {  
    // Code that turns the ship  
}  
  
    /**  
     * Function – performs the behavior (task) of moving the Ship  
     */  
public void move() {  
    // Code that moves the ship  
}  
  
    /**  
     * Function – performs the behavior (task) of firing the Ship's gun  
     */  
public void fire() {  
    // Code that makes the ship fire a bullet  
}
```

# EXAMPLE OF HIGH COHESION

```
/**  
 * The Ghost Class  
 */  
public class Ghost {  
  
    /**  
     * Function – moves the Ghost  
     */  
public void move() {  
    // Code that moves the ghost in the current direction  
}  
  
    /**  
     * Function - change Ghost direction  
     */  
public void changeDirection() {  
    // Code that changes the Ghost's direction  
}
```

The behaviour of changing state (what happens when Pac-Man eats a power pellet) requires three different tasks to be performed: turn deep blue, reverse direction, and move more slowly. To maintain cohesion, we don't want one function to do all three of these tasks, so we divide them up into three subtasks that the function will call upon to complete its one main task.

```
/**  
 * Function – change Ghost speed  
 */  
public void changeSpeed() {  
    // Code that changes the Ghost's speed  
}  
  
/**  
 * Function – change Ghost color  
 */  
public void changeColor() {  
    // Code that changes the Ghost's color  
}  
  
/**  
 * Function – change Ghost state  
 */  
public void changeState() {  
    // Code that changes the Ghost's state  
    // This function also will call the three functions of  
    // changeDirection, changeSpeed, and changeColor  
}
```

# BENEFITS OF HIGHER COHESION:

Highly **cohesive classes** are much easier to maintain and less frequently changed.

Such classes are more usable than others as they are designed with a well-focused purpose.

# DELEGATION PATTERN



**MTU**

Ollscoil Teicneolaíochta na Mumhan  
Munster Technological University

# WHAT IS THE PURPOSE OF THE DELEGATION PATTERN?

In software engineering, the **delegation** pattern is an object-oriented design pattern that allows object composition to achieve the same code reuse as inheritance.

In **delegation**, an object handles a request by delegating to a second object. The delegate is a helper object to original object.

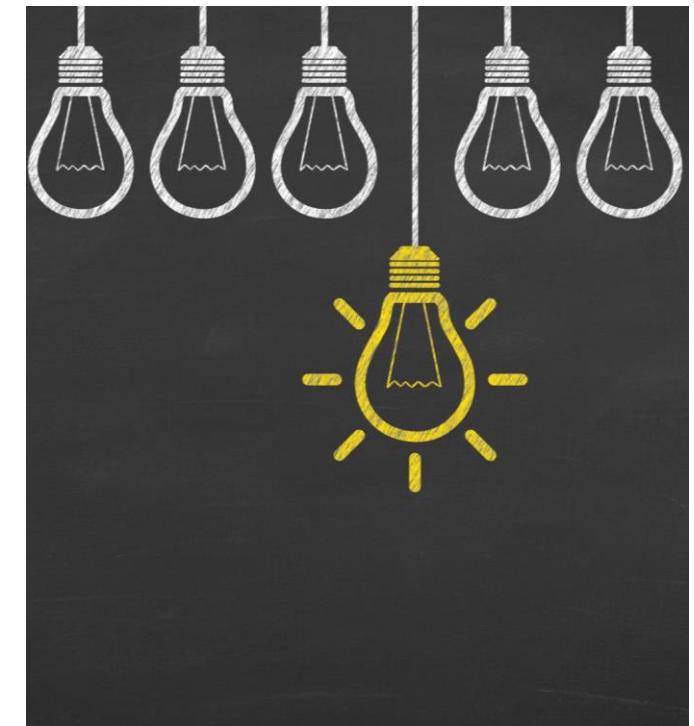
# FAVOUR COMPOSITION OVER INHERITANCE

**Composition** is favourable over inheritance.

**Composition** has several advantages over inheritance. One of the intuitions for this is as follows: Consider a sub-class which inherits from a base class. So, any change in the base class will make the sub-class fragile as the sub-class depends on the base class. By using inheritance, we are making a binding on the sub-class to depend on the base class, which makes our code fragile. However, by using composition, we can remove this limitation.

**Composition** is done by establishing a '**has-a relationship**' between classes instead of '**is-a**' relationship as in inheritance.

Inheritance is one of the great features of object-oriented programming languages such as Java, but it isn't the answer to every programming problem. Also, quite frankly, many Java programmers use it too much. In many cases, simply including an instance of one class in another class is easier than using inheritance. This technique is sometimes called the **Delegation pattern**.



# EXAMPLE OF APPLYING DELEGATION PATTERN

Suppose that you need to create a class named EmployeeCollection that represents a group of employees.

One way to create this class would be to extend one of the collection classes supplied by the Java API, such as the ArrayList class.

Then your EmployeeCollection class would be a specialized version of the ArrayList class and would have all the methods that are available to the ArrayList class.

A simpler alternative, however, would be to declare a class field of type ArrayList within your EmployeeCollection class. Then you could provide methods that use this ArrayList object to add or retrieve employees from the collection.

Why is this technique called the **delegation**? Rather than write code that implements the functions of the collection, you delegate that task to an ArrayList object, because ArrayList objects already know how to perform these functions.

# EXAMPLE IN JAVA OF DELEGATION

interface TravelBooking

```
{  
    public void bookTicket();  
}
```

```
class TrainBooking implements TravelBooking
{ public void bookTicket()
{ System.out.println("Train ticket booked"); }
}
```

```
class AirBooking implements TravelBooking
{
    public void bookTicket()
    {
        System.out.println("Flight ticket booked");
    }
}
```

```
class TicketBookingByAgent implements
TravelBooking
{
    TravelBooking t;
    public TicketBookingByAgent(TravelBooking t)
    {
        this.t = t;
    }
    // Delegation --- Here ticket booking responsibility
    // is delegated to other class using polymorphism
    public void bookTicket()
    {
        t.bookTicket();
    }
}
```

# DELEGATION

*TicketBookingByAgent* provides an implementation of *TravelBooking*.

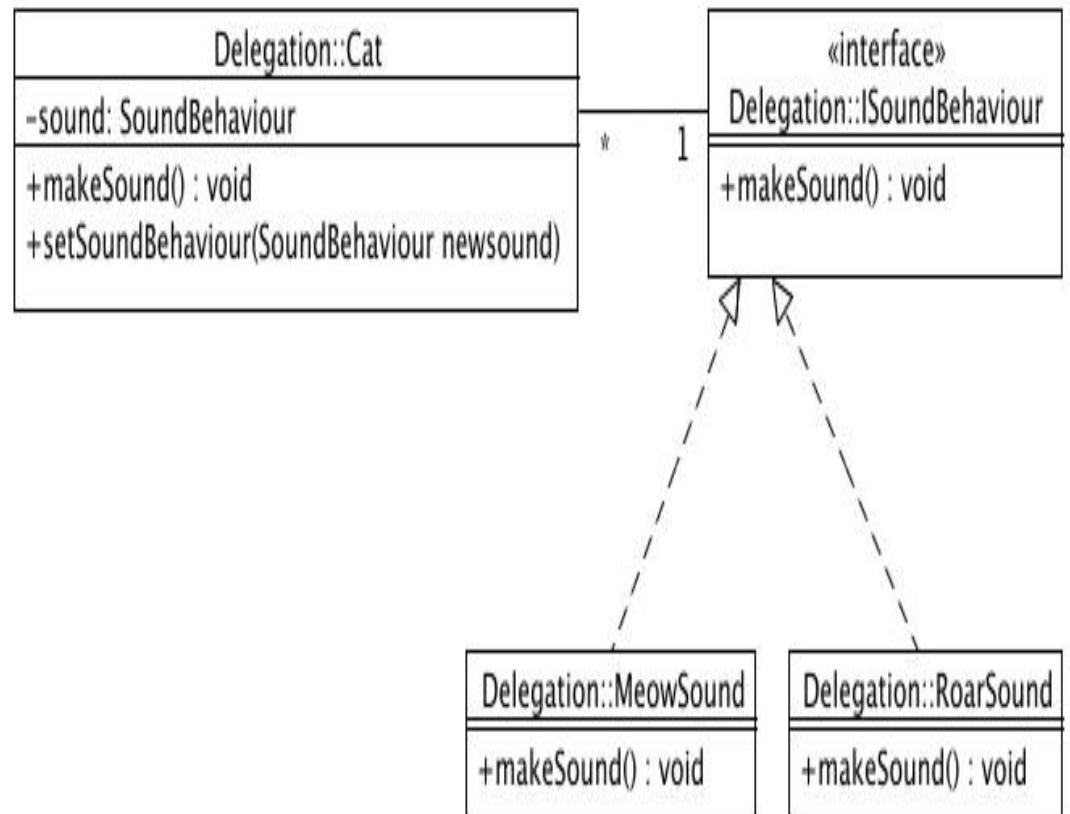
But it delegates actual ticket booking to other class at runtime using *Polymorphism*.



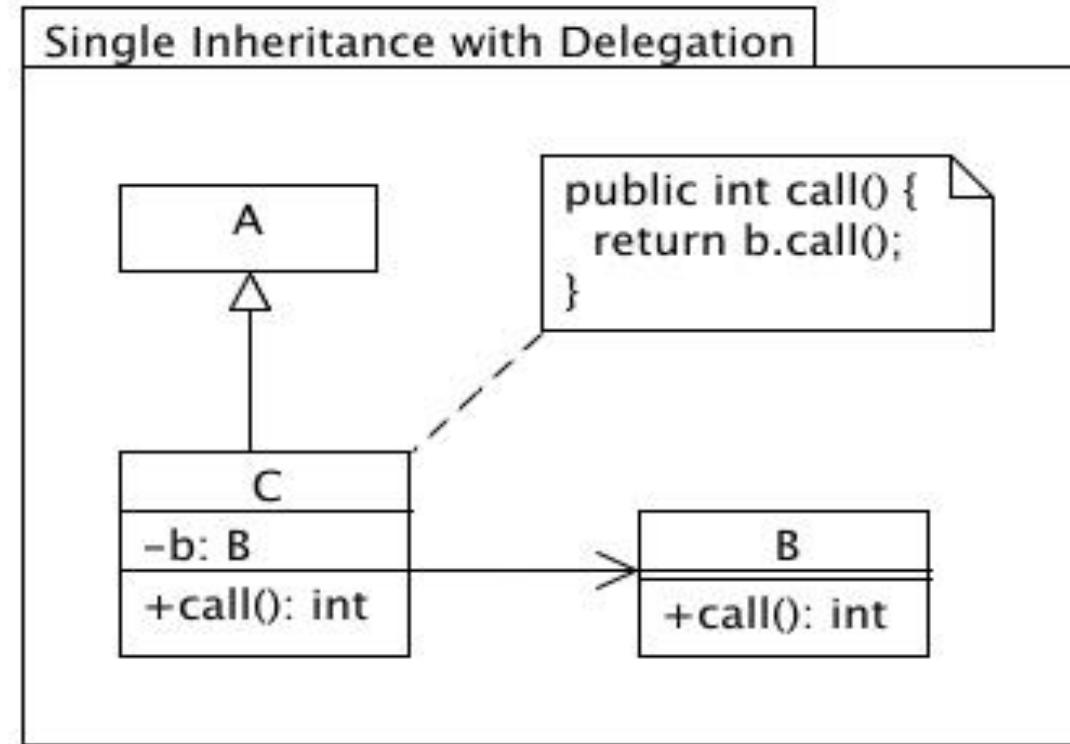
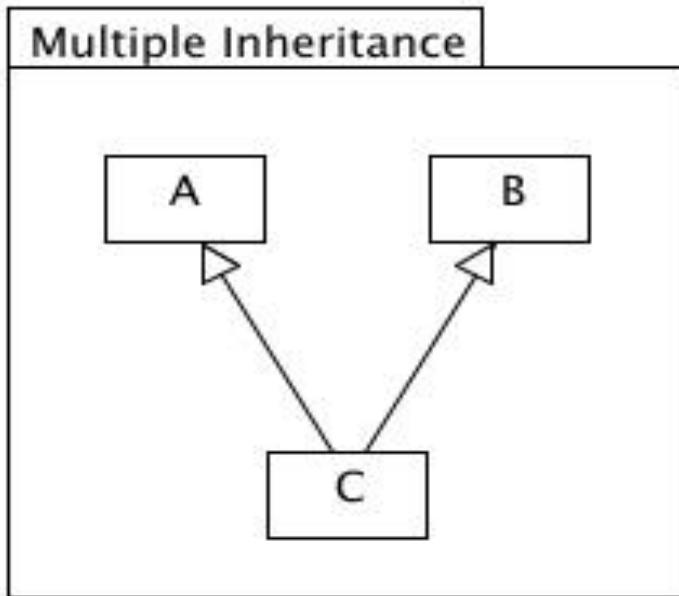
# TEST CLASS

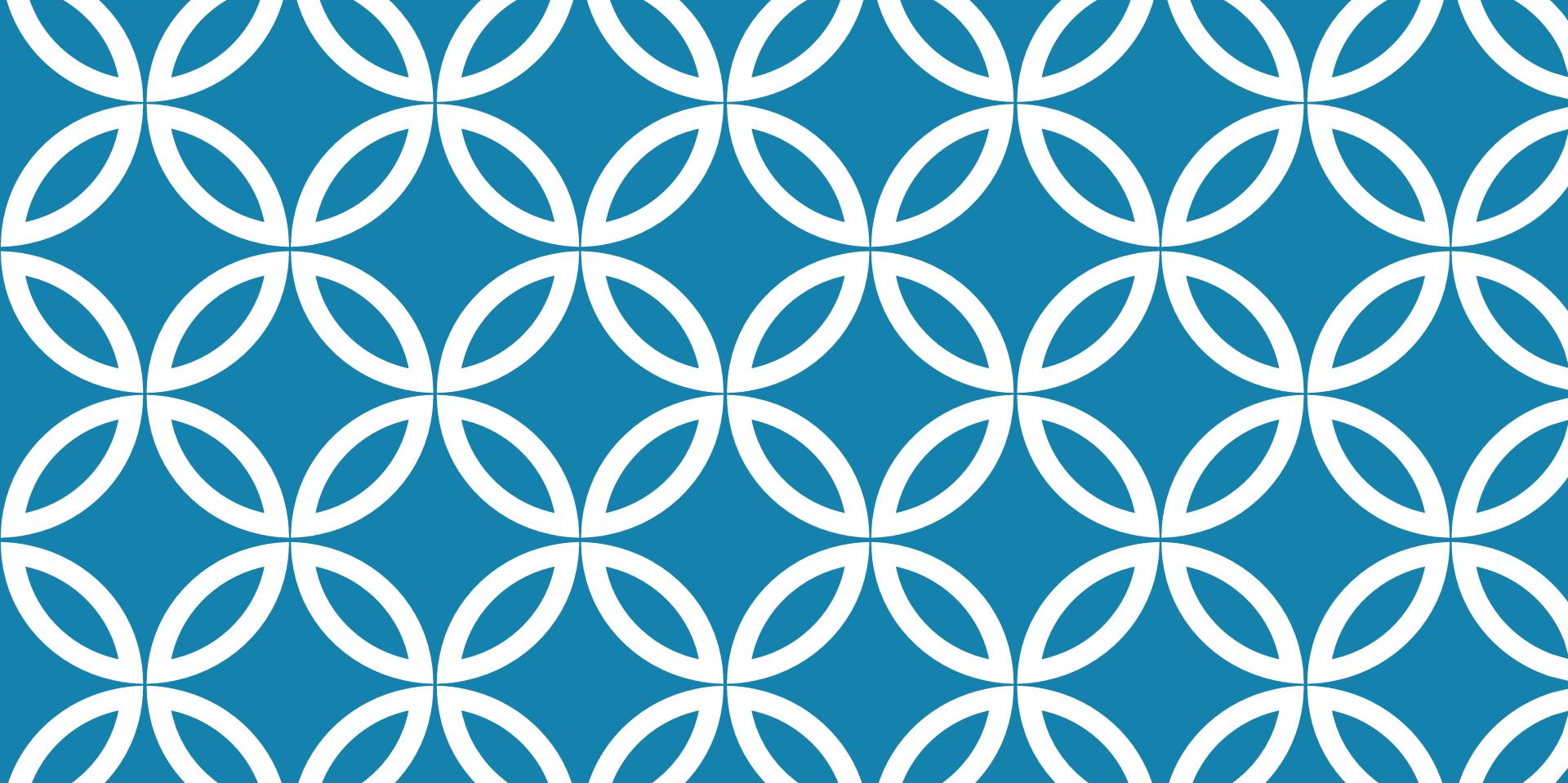
```
public class DelegationDemonstration
{
    public static void main(String[] args)
    { // Here TicketBookingByAgent class is internally
      // delegating train ticket booking responsibility to other class
      TicketBookingByAgent agent = new TicketBookingByAgent(new
      TrainBooking());
      agent.bookTicket();
      agent = new TicketBookingByAgent(new AirBooking());
      agent.bookTicket();
    }
}
```

# EXAMPLE OF DELEGATION



# EXAMPLE OF DELEGATION





# LAW OF DEMTER

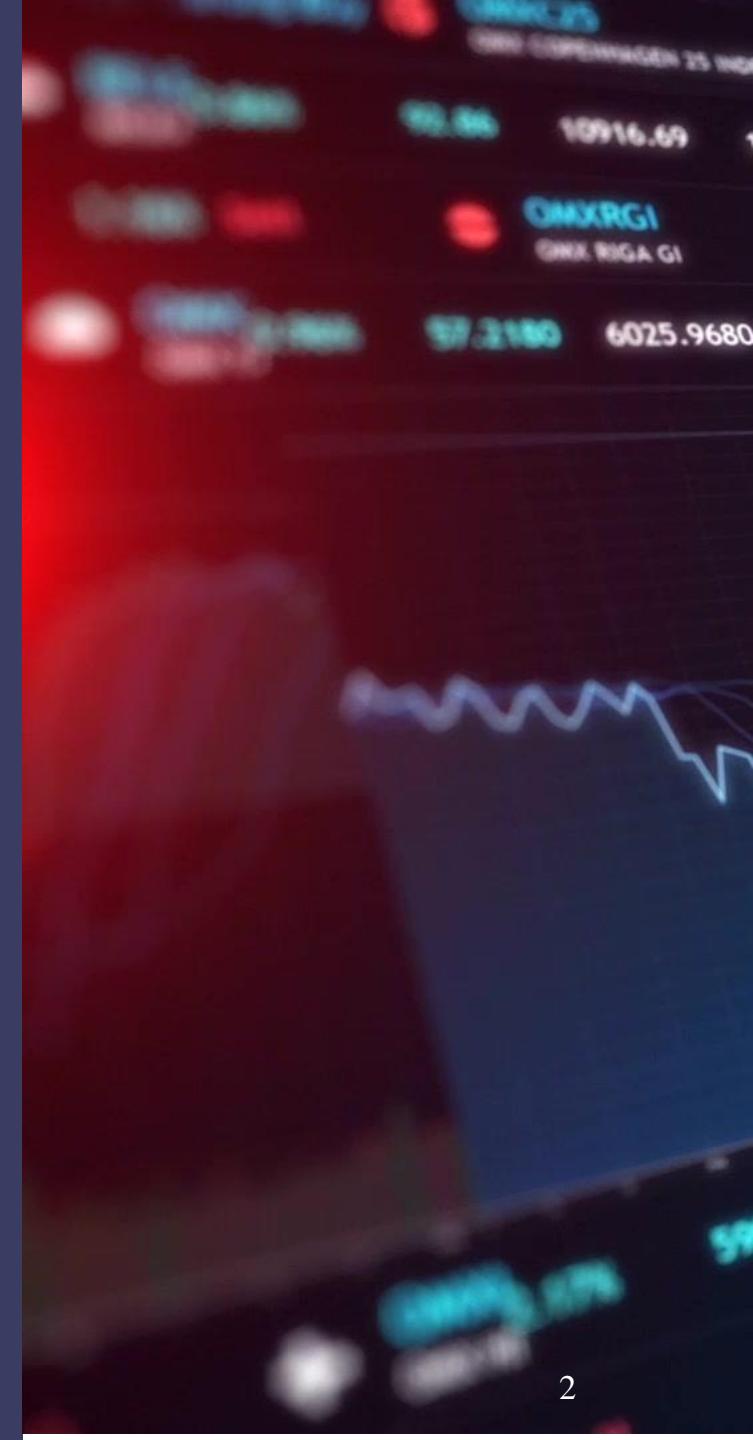
---

# SOFTWARE RIGIDITY

**Rigidity** is the tendency for software to be difficult to change, even in simple ways.

**Symptom:** Every change causes a cascade of subsequent changes in dependent modules.

**Effect:** When software behaves this way, managers fear to allow developers to fix non-critical problems. This reluctance derives from the fact that they don't know, with any reliability, when the developers will be finished.

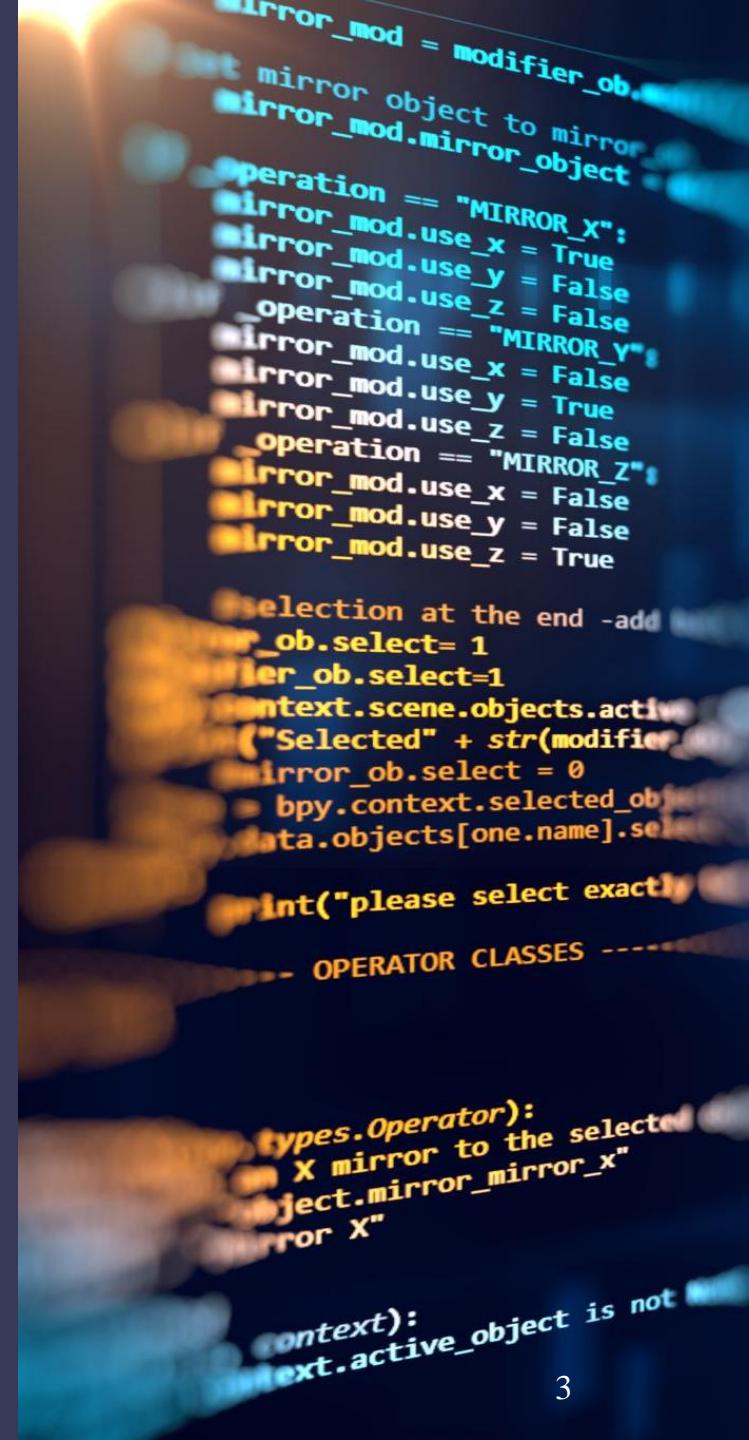


# SOFTWARE FRAGILITY

**Fragility** is the tendency of the software to break in many places every time it is changed. Often the breakage occurs in areas that have no conceptual relationship with the area that was changed.

**Symptom:** Every fix makes it worse, introducing more problems than are solved.

**Effect:** Every time managers/ team leaders authorize a fix , they fear that the software will break in some unexpected way.

A close-up photograph of a person's hand pointing their index finger towards a computer monitor. The monitor displays a dark-themed Python script, likely for a 3D modeling program like Blender. The code includes functions for mirroring objects and handling operator classes. The background is slightly blurred, emphasizing the hand and the screen.

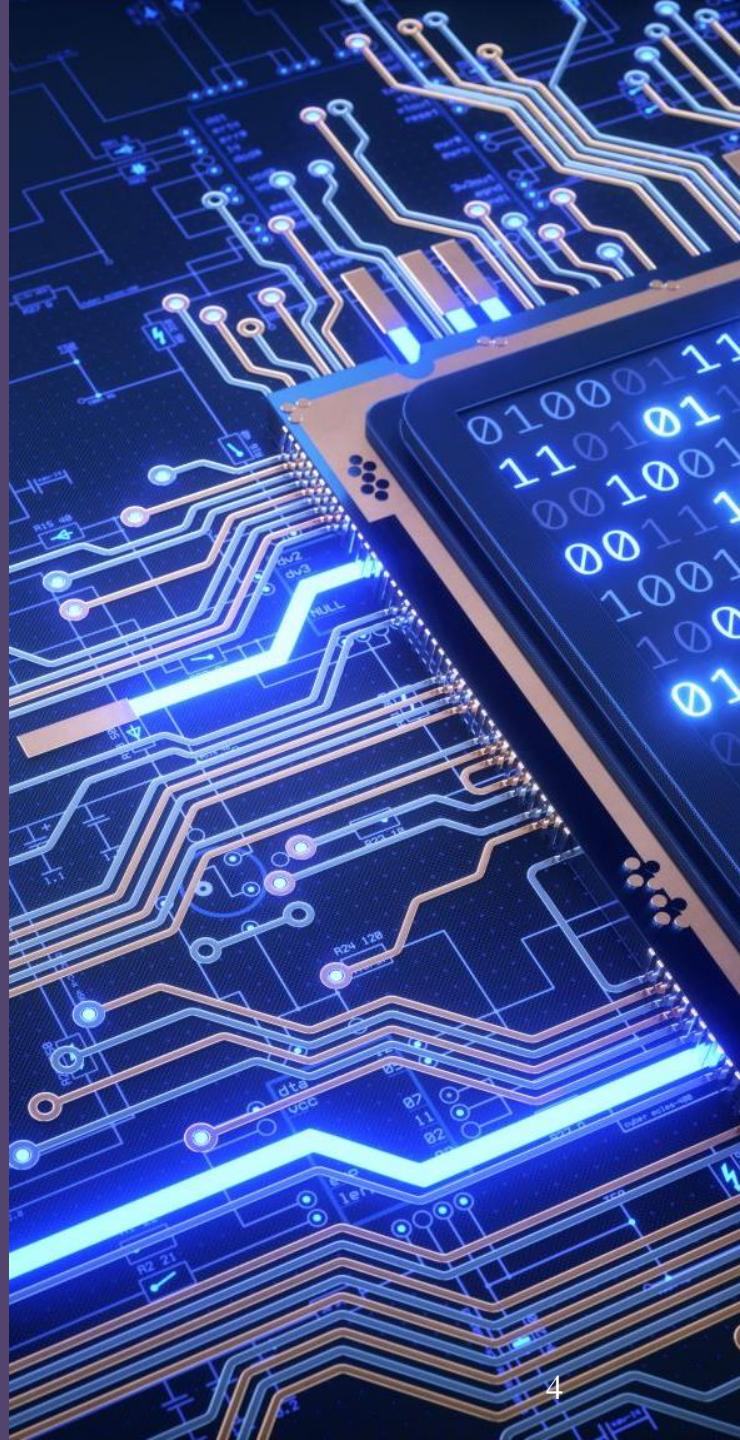
```
mirror_mod = modifier_obj
# mirror object to mirror
mirror_mod.mirror_object = ...
operation = "MIRROR_X":
    mirror_mod.use_x = True
    mirror_mod.use_y = False
    mirror_mod.use_z = False
operation == "MIRROR_Y":
    mirror_mod.use_x = False
    mirror_mod.use_y = True
    mirror_mod.use_z = False
operation == "MIRROR_Z":
    mirror_mod.use_x = False
    mirror_mod.use_y = False
    mirror_mod.use_z = True
selection at the end -add
    ob.select= 1
    ob.select=1
    context.scene.objects.active = ("Selected" + str(modifier))
    mirror_ob.select = 0
    bpy.context.selected_objects = data.objects[one.name].sel
    print("please select exactly one object")
-- OPERATOR CLASSES --
types.Operator):
    X mirror to the selected
    object.mirror_mirror_x"
    mirror X"
context):
    context.active_object is not
```

# SOFTWARE IMMOBILITY

**Immobility** is the inability to reuse software from other projects or from parts of the same project.

**Symptom:** A developer discovers that he needs a module that is similar to one that another developer wrote. But the module in question has too much baggage that it depends upon. After much work, the developer discovers that the work and risk required to separate the desirable parts of the software from the undesirable parts are too great to tolerate.

**Effect:** the software is simply rewritten instead of reused.



# SOFTWARE VISCOSITY

**Viscosity** is the tendency of the software/development environment to encourage software changes that are hacks rather than software changes that preserve original design intent.

**Symptom:** It is easy to do the wrong thing, but hard to do the right thing.

**Effect:** The software maintainability degenerates due to hacks, workarounds, shortcuts, temporary fixes etc.

# WHY BAD DESIGN RESULTS?

**Obvious reasons:** lack of design skills/  
design practices, changing technologies,  
time/ resource constraints, domain  
complexity etc.

**Not so obvious:**

- Software rotting is a slow process .. Even originally clean and elegant design may degenerate over the months/ years ..
- Unplanned and improper module dependencies creep in; Dependencies go unmanaged.
- Requirements often change in the way the original design or designer did not anticipate ..



# LAW OF DEMETER

Also known as the  
Principle of least  
knowledge

The law of Demeter, is a design principle which provides guidelines for designing a system with minimal dependencies. It is typically summarized as “**Only talk to your immediate friends.**”

# THE LAW OF DEMETER

Whenever you talk to a good, experienced programmer, they will tell you that "**loosely coupled**" classes are very important to good software design.

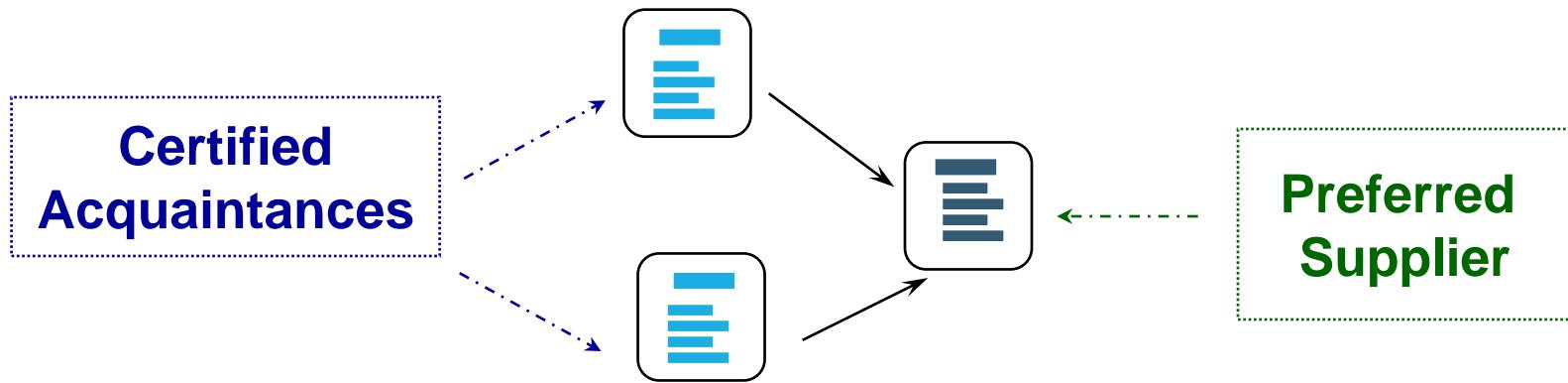
The **Law of Demeter** for functions (or methods, in Java) attempts to minimize coupling between classes in any program. In short, the intent of this "law" is to prevent you from reaching into an object to gain access to a third object's methods.

The Law of Demeter is often described this way:  
**"Only talk to your immediate friends."**

or, put another way:

**"Don't talk to strangers."**

# LAW OF DEMETER

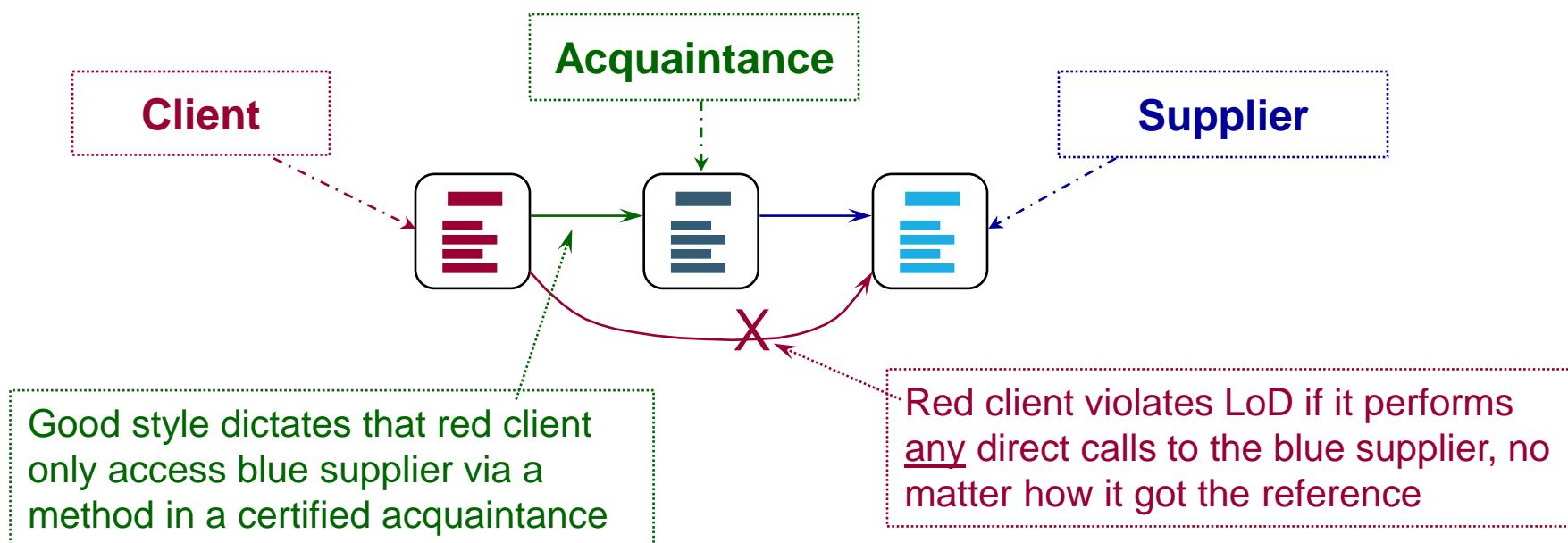


- Minimize the number of acquaintances which directly communicate with a supplier
  - easier to contain level of difficulty per method
  - reduces amount of rework needed if supplier's interface is modified
  - So only allow “certified” classes to call supplier

# DEMETER'S “GOOD STYLE”

Access supplier methods only through methods of “preferred acquaintances”

- bypass preferred supplier only if later optimization demands direct access



# A LAW OF DEMETER JAVA EXAMPLE

```
/*
 * A Law of Demeter example in Java.
 * The Pragmatic Programmer.
 */
public class LawOfDemeterInJava
{
    private Topping cheeseTopping;

    /**
     * Good examples of following the Law of Demeter.
     */
    public void goodExamples(Pizza pizza)
    {
        Foo foo = new Foo();

        // (1) it's okay to call our own methods
        doSomething();

        // (2) it's okay to call methods on objects passed in to our method
        int price = pizza.getPrice();

        // (3) it's okay to call methods on any objects we create
        cheeseTopping = new CheeseTopping();
        float weight = cheeseTopping.getWeightUsed();

        // (4) any directly held component objects
        foo.doBar();
    }

    private void doSomething()
    {
        // do something here ...
    }
}
```

# LAW OF DEMETER

The **Law of Demeter** for **methods** requires that a method  $M$  of an object  $O$  may only invoke the methods of the following kinds of objects:

- $O$  itself
- $M$ 's parameters
- Any objects created/instantiated within  $M$
- $O$ 's direct component objects
- A global variable, accessible by  $O$ , in the scope of  $M$

# ACCEPTABLE LOD VIOLATIONS

- If optimization requires violation
  - Speed or memory restrictions
- If module accessed is a fully stabilized “Black Box”
  - No changes to interface can reasonably be expected due to extensive testing, usage, etc.
- Otherwise, do not violate this law!!
  - Long-term costs will be very prohibitive

# JAVA CODE EXAMPLE OF DEMETER LAW APPLIED

Refer to Example in separate document

“ The Law of Demeter Applied”

# Value Objects

Value objects let you transform values into objects themselves

Most of the concepts we're modelling in our software have **no global identity**.

For example, if we were talking about a **Person** class, we would probably see fields like **firstName**, **lastName**, or **address**. A person has a clear, global identity. If there was an object representing me, the object would refer to a real person sitting at my desk, finishing correcting waiting to go home – I'm an entity. On the other hand, if we were talking only about a **String** like “Mary Davin”, we don't have this global scope – it's just a **value**.

Values like the ones mentioned above can have certain logic associated with them e.g. validation, transformations, or calculus. As we're using an OO language, it makes all the sense in the world to use its powers and combine the value and the logic together in an object.

```
// entity:  
class Person {  
    PersonId id; // global identity  
    FirstName firstName;  
    LastName lastName;  
    Address address;  
}  
// value objects:  
class PersonId {  
    Long value;  
}  
class FirstName {  
    String value;  
}  
class LastName {  
    String value;  
}  
class Address {  
    String street;  
    String streetNo;  
    String city;  
    String postalCode;  
}
```

```

// Address
@Override
public boolean equals(Object o) {
    // basic checks and casting cut out for brevity
    return Objects.equals(street, address.street) &&
        Objects.equals(streetNo, address.streetNo) &&
        Objects.equals(city, address.city) &&
        Objects.equals(postalCode, address.postalCode);
}
@Override
public int hashCode() {
    return Objects.hash(street, streetNo, city, postalCode);
}

```

As the value objects have no identity, we compare them together by simply comparing all the values they contain:

Usually, we also make/treat the value objects as immutable, i.e. instead of changing the value objects, we create new instances that wrap the new values:

```

// wrong:
this.address.setStreet(event.street);
// good:
this.address = new Address(event.street, ...);

```

There are practical and conceptual reasons behind this. From the practical perspective, immutable types are handy, as they can be easily shared between different objects and returned by the entities without the risk of compromising consistency. From the conceptual perspective, it makes sense to create a new instance of a value object when the value changes, as we're literally assigning a *new value*.

## Benefits

The code gets more expressive

```

// without:
Map<Long, String>
// with:
Map<PersonId, PhoneNumber>

```

1. They make our code safer, as data types prevent us doing stupid things.

```

Person(String firstName, String lastName, String email) { ... }
new Person("john@doe.com", "John", "Doe"); // compiles
Person(FirstName firstName, LastName lastName, Email email) { ... }
new Person(
    new Email("john@doe.com"),
    new FirstName("John"),
    new LastName("Doe")); // doesn't compile!

```

2. They give us the flexibility in terms of internal representation. For example, I could easily change...:

```

class PersonId {
    Long id;
    // c-ctor, getter
}

```

...to...:

```

class PersonId {
    String value;
    public PersonId(Long value) {
        this.value = value.toString();
    }
    // rest
}

```

.without changing most of the `PersonId` clients.

- 3.

As mentioned, they also encapsulate related logic e.g. validation

```

4. class Email {
5.     String value;
6.     Email(String value) {
7.         if (!value.contains("@")) { // don't use this code in a real
        product
8.             throw new InvalidEmailException(value);
9.         }
10.        this.value = value;
11.    }
12. }

```

Or calculus:

```
class Money {  
    BigDecimal amount;  
    Money(BigDecimal amount) {  
        this.amount = amount;  
    }  
    Money add(Money other) {  
        //  
        return new Money(amount + other.amount);  
    }  
    // etc.  
}
```

## Drawbacks

The obvious drawbacks of value objects are that the numbers of classes in the project might grow significantly and, as they're usually immutable, the number of created objects, too. Create value objects only when you see that there is a piece of logic to encapsulate.

## Implementing Value Objects

Basically, this boils down to implementing an immutable class.

to create such an immutable class, we have to:

- Make it `final`
- Make all the fields `final` (this also implies no field mutation in methods and creating new objects when someone wants a value object with different data)
- Copy all mutable state during construction and retrieval

```

public final class DateRange {
    private final Date fromInclusive;
    private final Date toExclusive;
    public DateRange(Date fromInclusive, Date toExclusive) {
        this.fromInclusive = (Date) fromInclusive.clone();
        this.toExclusive = (Date) toExclusive.clone();
    }
    public DateRange extend(long millis) {
        Date extendedTo = new Date(toExclusive.getTime() + millis);
        return new DateRange(fromInclusive, extendedTo);
    }
    public Date getFromInclusive() {
        return (Date) fromInclusive.clone();
    }
    public Date getToExclusive() {
        return (Date) toExclusive.clone();
    }
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        DateRange dateRange = (DateRange) o;
        return fromInclusive.equals(dateRange.fromInclusive)
            && toExclusive.equals(dateRange.toExclusive);
    }
    @Override
    public int hashCode() {
        int result = fromInclusive.hashCode();
        result = 31 * result + toExclusive.hashCode();
        return result;
    }
}

```

## Importance of Composition:

- In composition we can control the visibility of other object to client classes and reuse only what we need.
- Composition allows creation of back-end class when it is needed.

## Comparing Composition and Inheritance:

- It is easier to change the class which is implementing composition than inheritance.
- In inheritance, you cannot add method to a child class with the same method name but a different return type as that method inherited from a parent class. On the other hand, composition allows us to change the interface of a front-end class without affecting back-end classes.
- Composition is done at run time i.e. dynamic binding while Inheritance is done at compile time i.e. static binding.
- If you want to reuse code and there is no **is-a** relationship, then use composition. You don't need to use inheritance for code reuse.
- If you want polymorphism, but there is no **is-a** relationship, then use composition with interfaces. You don't need to use inheritance to get polymorphism.

# Interface Separation Principle

# What does it state?

- ▶ The **Interface Segregation Principle** states that clients should not be forced to implement interfaces they don't use.
- ▶ Instead of one fat interface many small interfaces are preferred based on groups of methods, each one serving one submodule.

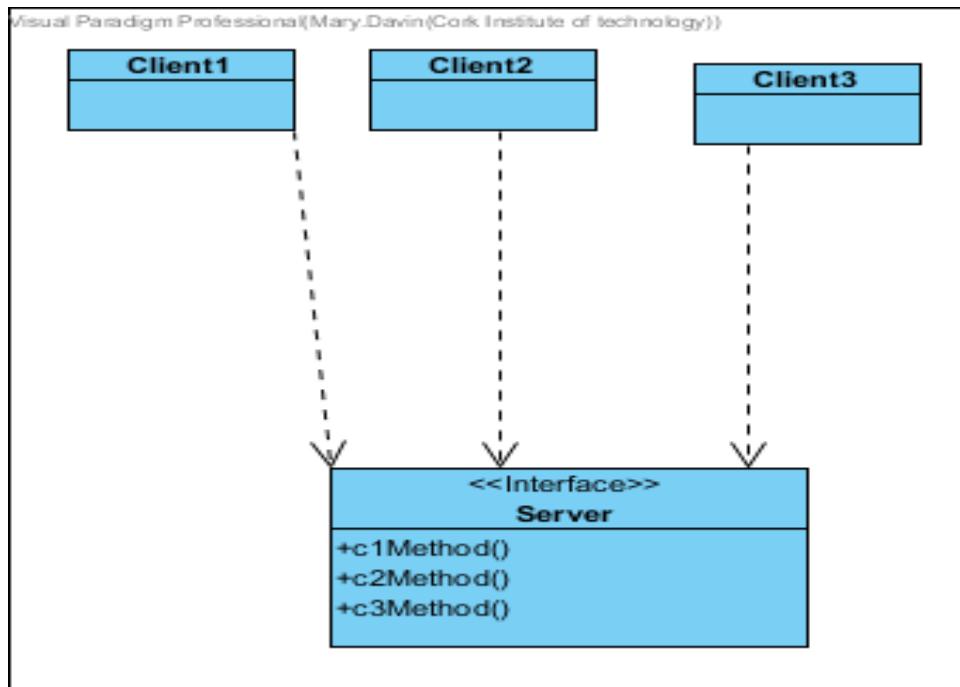
# Interface Segregation Principle

Helps deal with “fat” or inappropriate interfaces

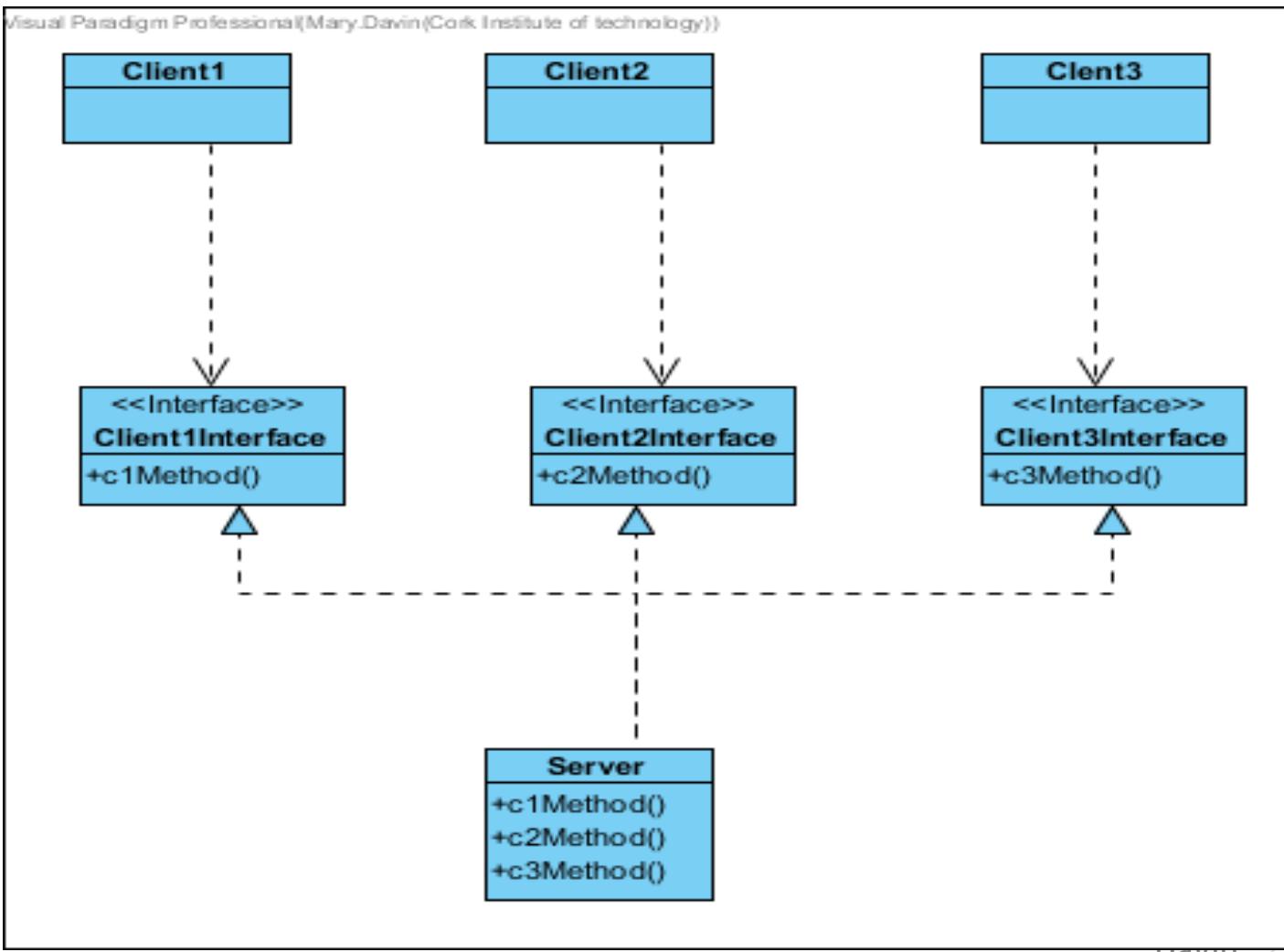
- ▶ Sometimes class methods have various groupings.
- ▶ These classes are used for different purposes.
- ▶ Not all users rely upon all methods.
- ▶ This lack of cohesion can cause serious dependency problems
- ▶ These problems can be refactored away.

# Interface Pollution by “collection”

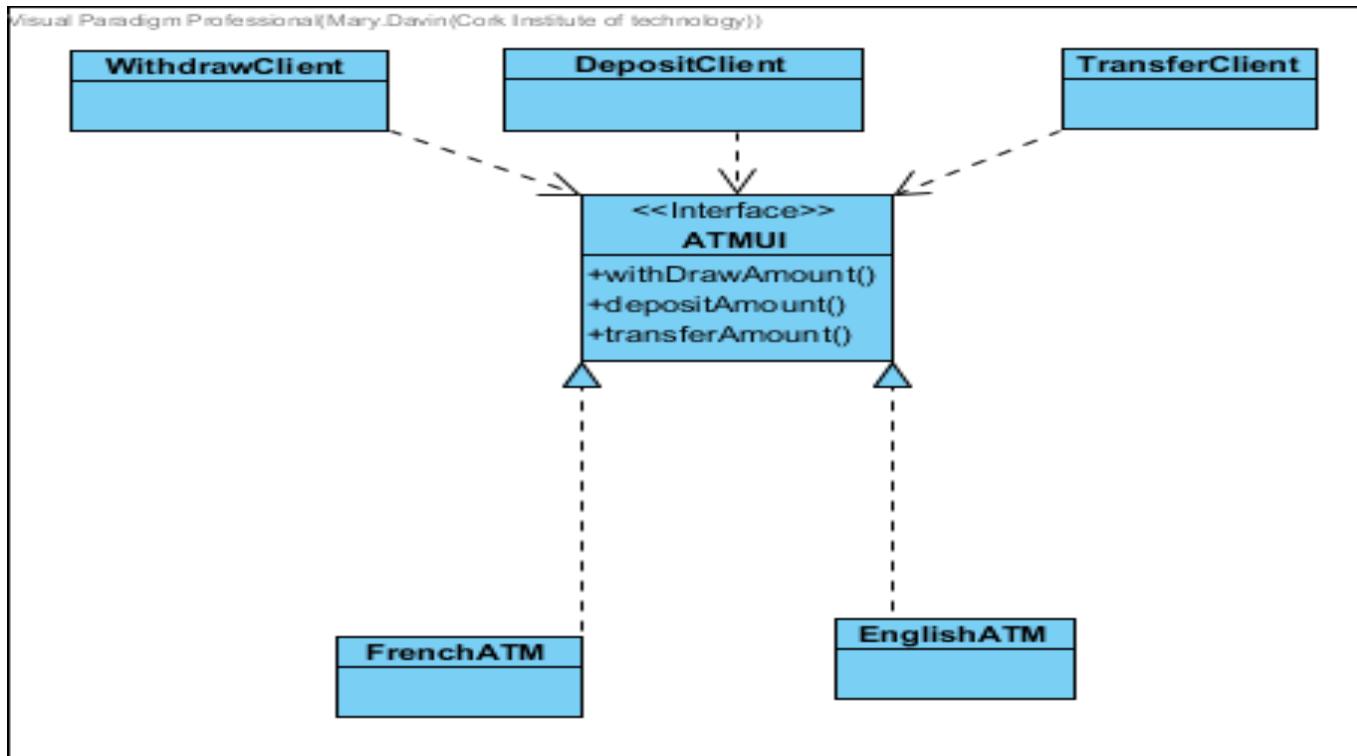
Distinct clients of our class have distinct interface needs.



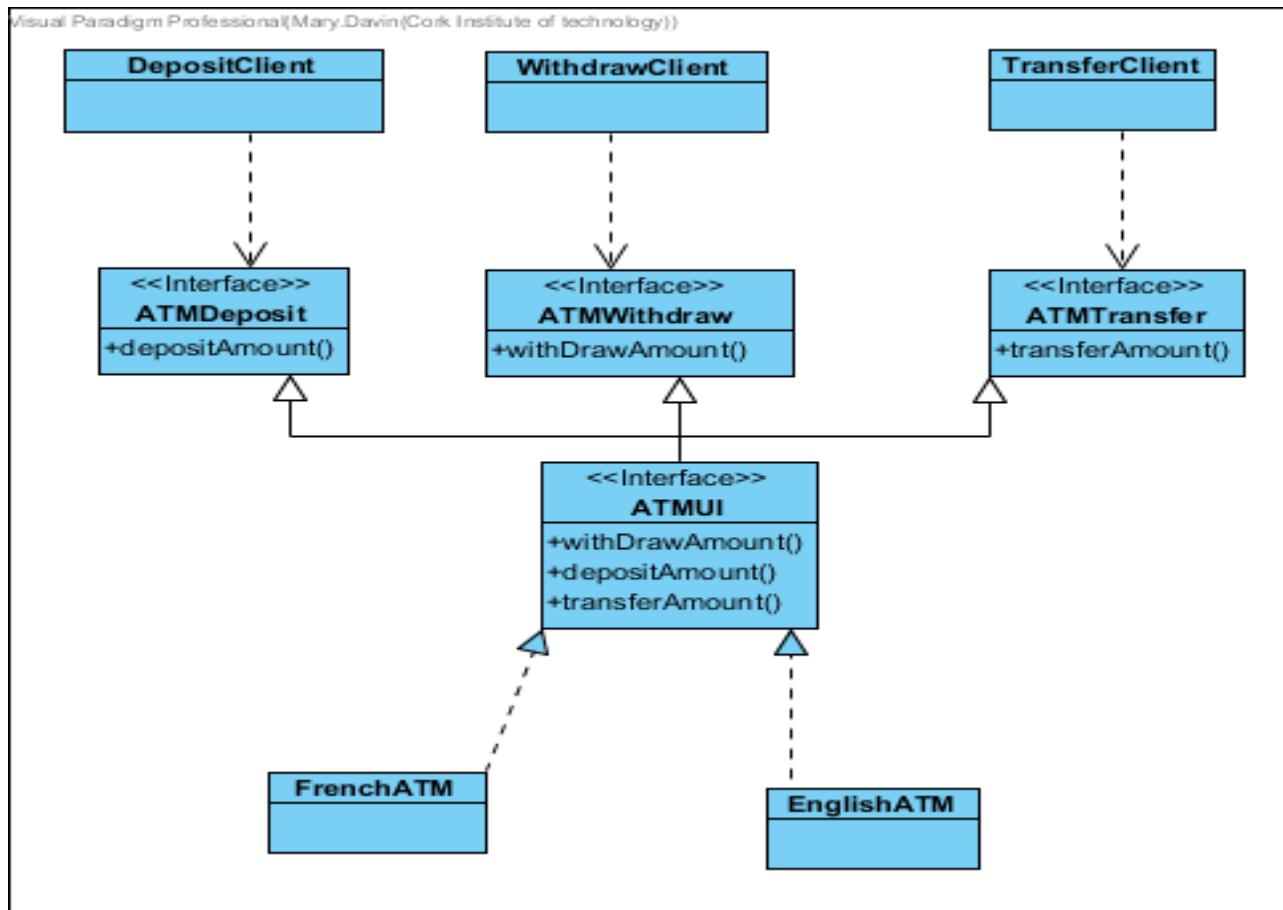
# A Segregated Example



# ATM UI Example



# A Segregated ATM UI Example



# The `java.util.List` Interface

The `List` interface specifies the methods that can be used with any implementing class, such as `ArrayList`. Programmers should use `List` as the type name of all list variables, parameters, and method return values.

```
// Specs for List and ArrayList in java.util (AP  
only)

interface java.util.List<E>  
    int size()  
    boolean add(E obj)  
    void add(int i, E obj)  
    E get(int i)  
    E set(int i)  
    E remove(int i)

class java.util.ArrayList<E> implements  
java.util.List<E>

// Example use of List and ArrayList

List<Integer> numbers = new ArrayList<Integer>();  
List<String> names = new ArrayList<String>();

for (int i = 1; i <= 10; i++){  
    numbers.add(i);  
    names.add("Name " + i);  
}

for (int i : numbers)  
    System.out.println(i);

for (String name : names)  
    System.out.println(name);
```

# Dependency Inversion Principle

---

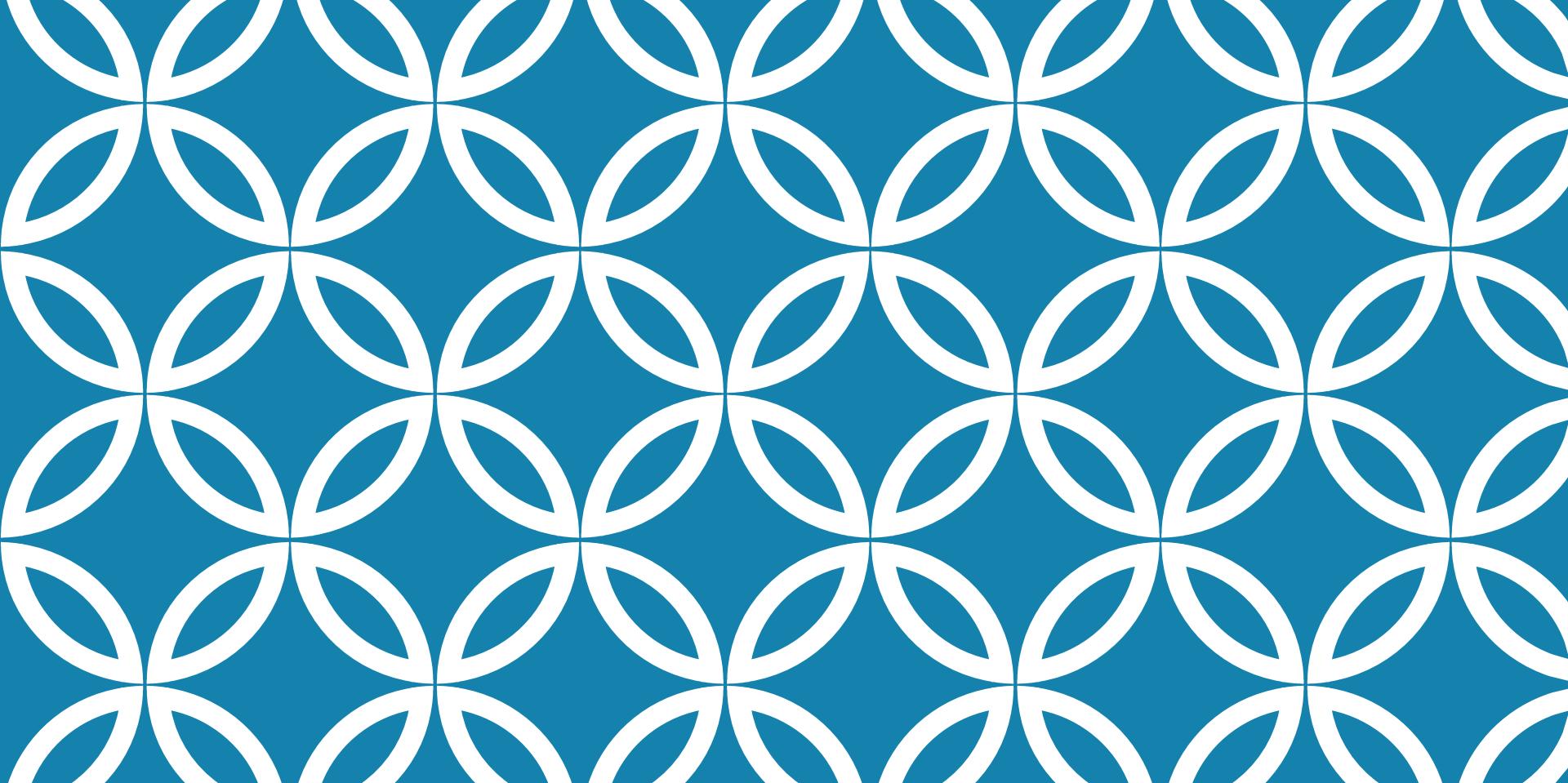
- Depend on abstractions, not concretions
- Program to interfaces not implementations
- Program to most abstract class possible
- Why? Concrete classes may change a lot. Abstract classes/Interfaces generally change very little.

How can we ensure interfaces change very seldom

# Interface Segregation Principle

---

- Don't make large multipurpose interfaces – instead use several small focused ones.
- Don't make clients depend on interfaces they don't use.
- Class should depend on each other through the smallest possible interface.
- Why? When I change something I want to minimize changes for everyone else.



# THE OBJECT MODEL

# THE OBJECT MODEL

A general view of program structure shared by UML and object-oriented programming languages like Java and C++

Computation takes place in *objects* that:

- store data and implement behaviour
- are linked together in a network
- communicate by sending messages
- are described by classes

# UML AND CODE

Because of the shared object model:

- UML diagrams can easily be implemented
- object-oriented programs can easily be documented in UML

Reverse engineering:

- creating a UML model for existing code
- often useful in dealing with undocumented legacy systems

# STOCK CONTROL EXAMPLE

Keep track of *parts* in a warehouse and how they are used

- parts have a part number, name and cost
- the system must track individual parts
- parts can be used to form *assemblies*
- assemblies have a hierarchical structure

Could be used for a variety of applications

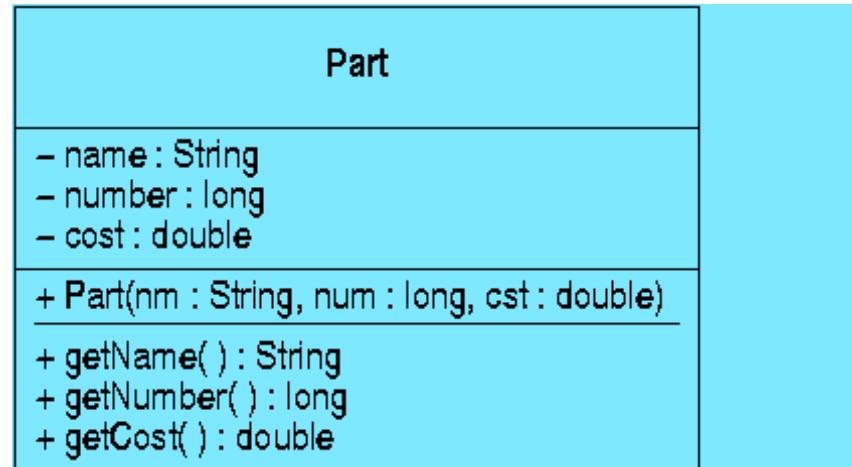
- eg find the cost of the materials in an assembly

# A SIMPLE PART CLASS

```
public class Part {  
    private String name ;  
    private long number ;  
    private double cost ;  
  
    public Part(String nm, long num, double  
               cost) {  
        name = nm ;  
        // etc  
    }  
    public String getName( ) { return name; }  
    // etc  
}
```

# THE PART CLASS IN UML

A class is represented by a single UML icon



# UML CLASS NOTATION

Class icons have three compartments

- the first shows the class *name*
- the second optionally shows *attributes*
  - derived from its fields (data members)
- the third optionally shows *operations*
  - derived from its methods (member functions)
  - constructors (and static methods) underlined

Class *features* = attributes + operations

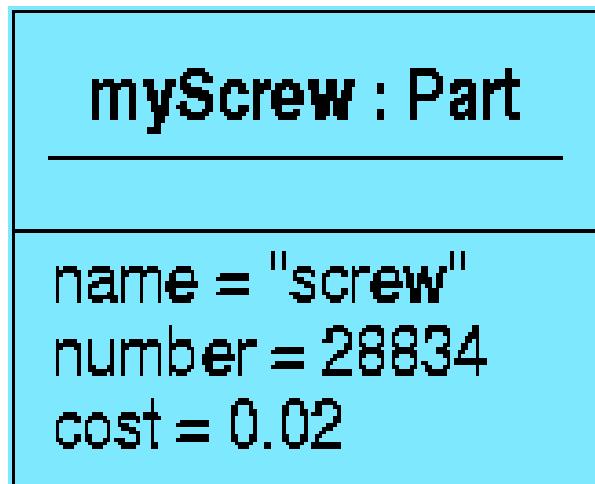
- access levels: public (+) and private (-)

# OBJECTS

Objects are created at run-time as *instances* of classes:

```
Part myScrew = new Part("screw", 28834,  
0.02) ;
```

Individual objects can be shown in UML:



# UML OBJECT NOTATION

## Object icons have two compartments

- the first shows the object and class names
  - variable names can be used as object names
  - the object name is optional
  - the class name is preceded by a colon
  - the names are always underlined
- the second shows the actual data stored in the object, as attribute/value pairs
  - this compartment is usually omitted

# OBJECT PROPERTIES

Objects have:

- state**: the data values held in the object
- behaviour**: the functionality it contains
- identity**: a distinguishing property of an object
  - not an attribute value
  - could be the object's address in memory
- names**: useful in modelling to refer to object

Objects **normally encapsulate** their data

# AVOID DATA REPLICATION

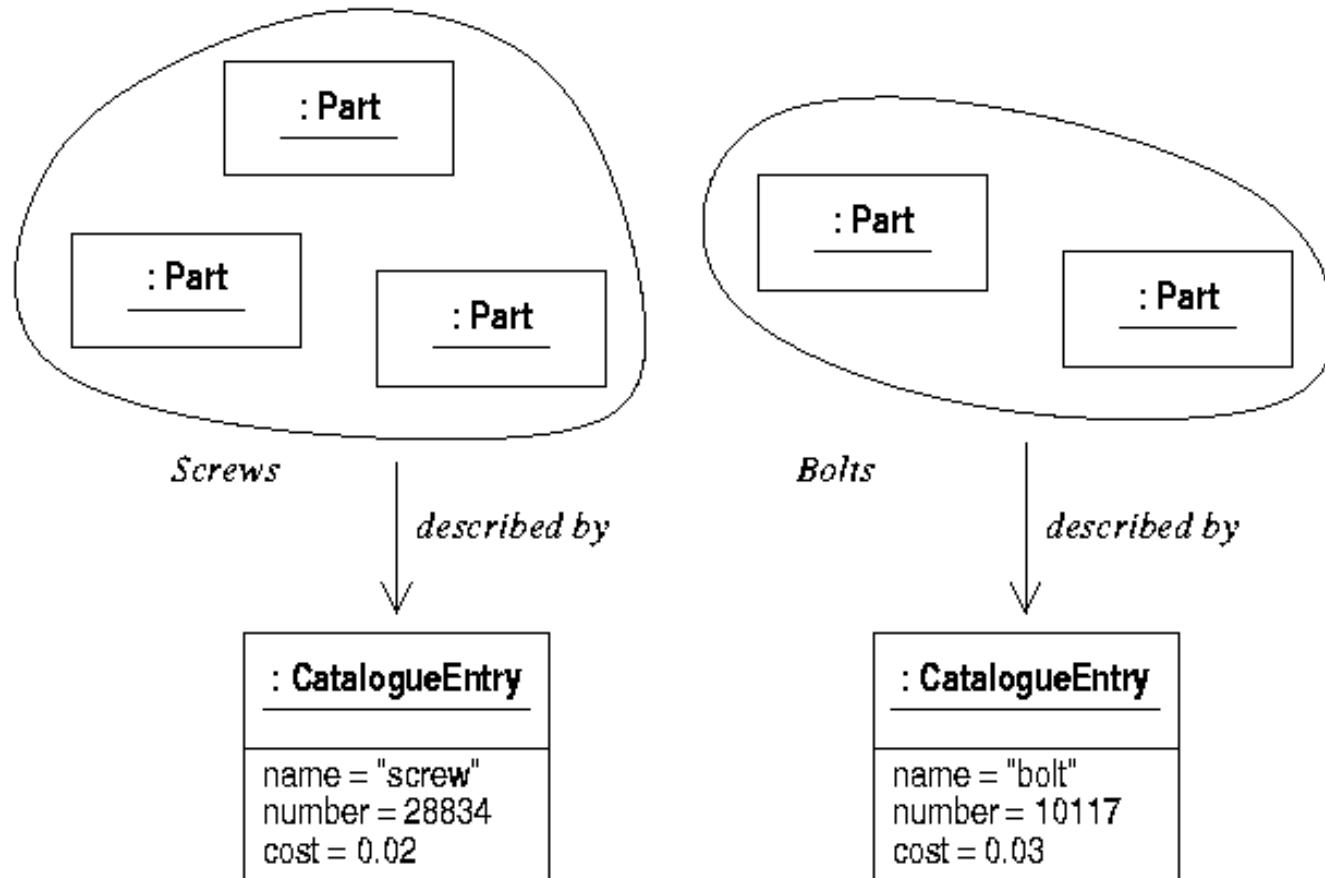
All parts of the same kind have the same attribute values  
(name, number, cost)

- wasteful of resources
- hard to maintain, e.g. if the cost changes

Better to create a new class for shared data

- call this a ‘catalogue entry’
- describes a type of part
- each individual part described by one entry

# PARTS AND CATALOGUE ENTRIES



# IMPLEMENTATION

Catalogue entry class defines shared data

Each part holds a reference to an entry

```
public class Part {  
    private CatalogueEntry entry ;  
    public Part(CatalogueEntry e) { entry =  
        e; }  
}
```

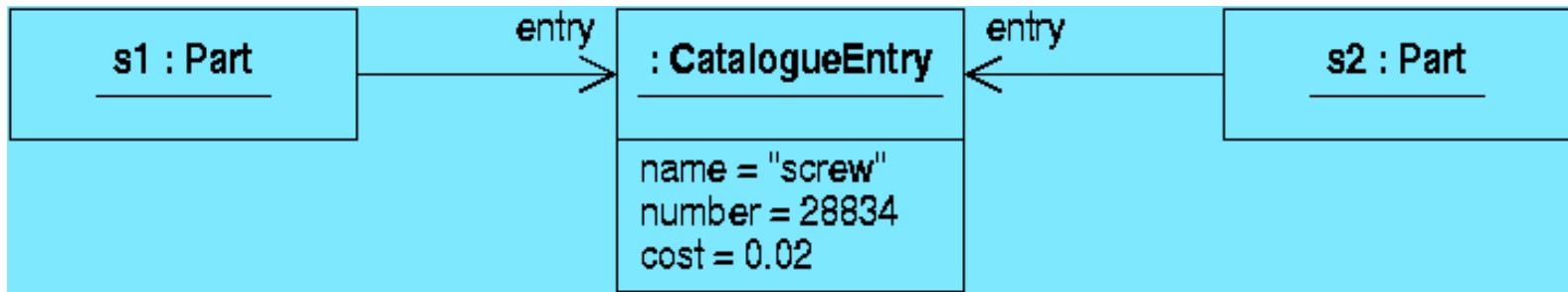
When parts are created, an entry is given

```
CatalogueEntry screw = new  
    CatalogueEntry("screw", 28834, 0.02);  
Part s1 = new Part(screw) ;  
Part s2 = new Part(screw) ;
```

# LINKS

References can be shown in UML as *links*

- object name is name of variable
- field name ‘entry’ used to label end of link



# ASSOCIATIONS

Data relationships between classes

Links are *instances* of associations



# PROPERTIES OF ASSOCIATIONS

Associations can have:

- a *name* (not shown)
- a *rolename* at each end
  - derived from field name in code
- a *multiplicity* at either end
  - this specifies how many links there can be
- an arrowhead at either end showing *navigability*
  - derived from direction of reference in code

# UML DIAGRAM TYPES

*Object diagrams* show objects and links

- this describes the *run-time* properties of a program

*Class diagrams* show classes and associations

- this shows the *compile-time* properties of a program
- ie the logical structure of its source code

# OBJECT INTERACTION

Objects interact by method calls

```
public class Part {  
    private CatalogueEntry entry ;  
    public double cost( ) { return  
        entry.getCost(); }  
}  
...  
Part s1 = new Part(screw) ;  
double s1cost = s1.cost() ;
```

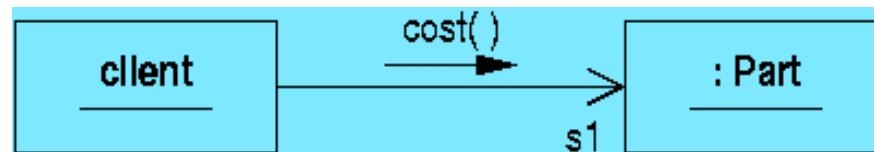
Functionality is distributed among objects which must communicate

# MESSAGES

Object communication is shown in UML as *message passing*

- a message corresponds to a method call

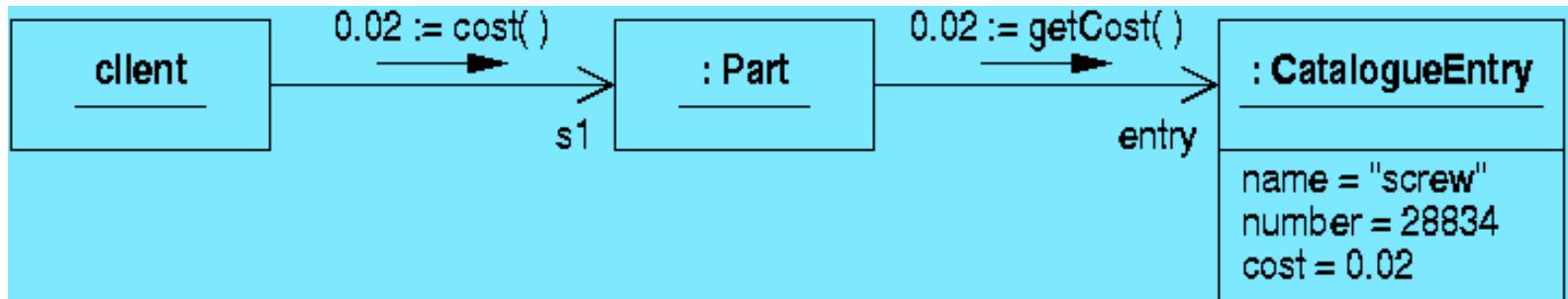
- the class of the client object is not specified in this diagram



# COMMUNICATION DIAGRAMS

Object diagrams with messages show **collaborations**

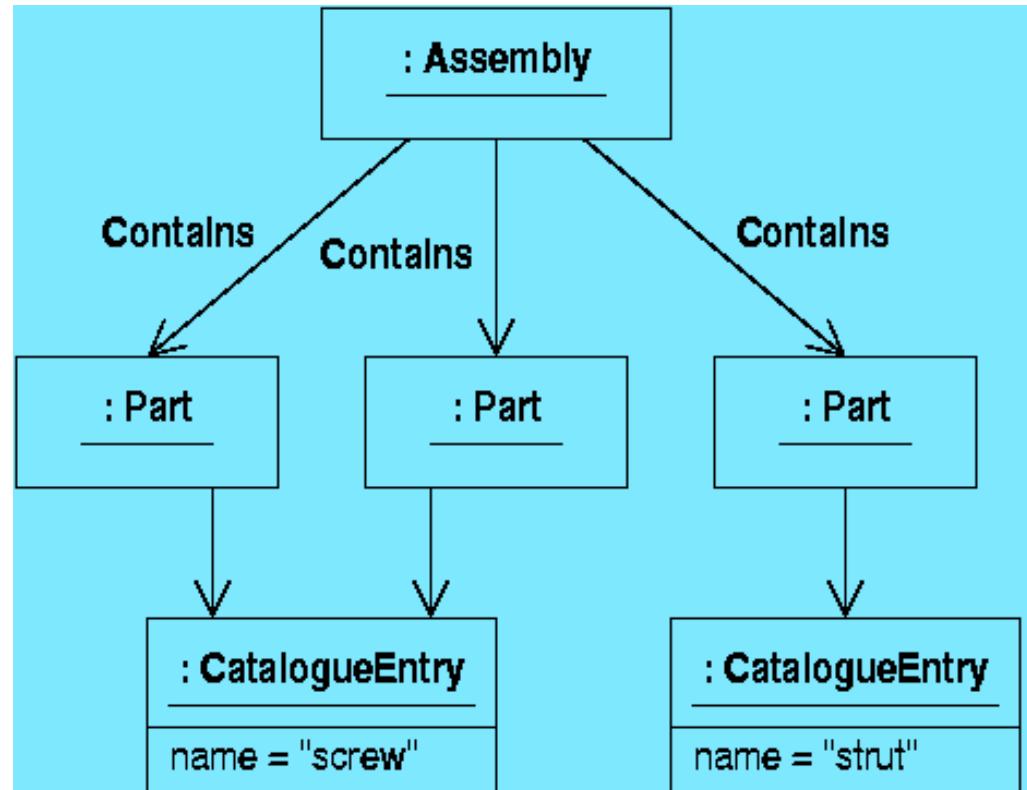
- multiple messages can be shown on one diagram
- return values are shown using the ‘:=’ notation



# ASSEMBLIES

Assemblies contain parts

- Links show which assembly a part belongs to



# IMPLEMENTING ASSEMBLIES

Assemblies could be implemented using a data structure to hold references to parts

```
public class Assembly {  
    private List<Part> parts = new  
        ArrayList() ;  
  
    public void add(Part p) {  
        parts.addElement(p) ;  
    }  
}
```

These links are instances of a new association

# THE 'CONTAINS' ASSOCIATION

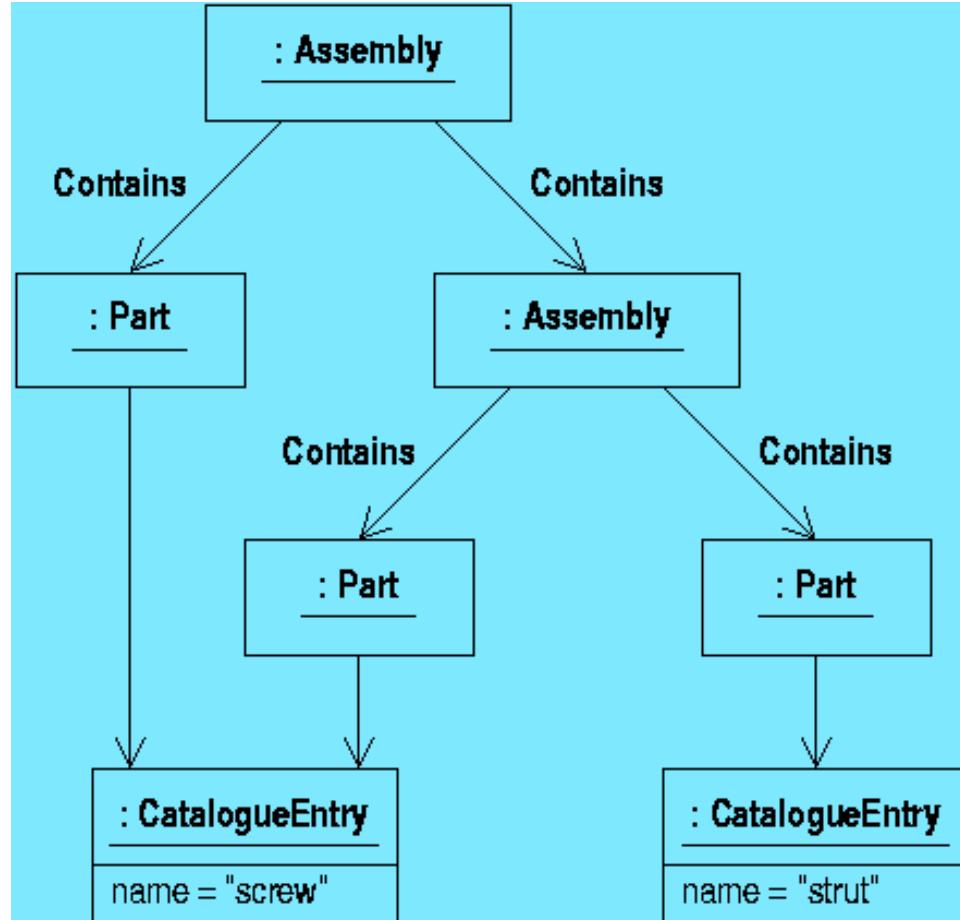
‘Contains’ is the *association name*  
— “an assembly *contains* parts”

‘parts’ is the *rolename*

Assemblies can contain zero or more parts  
— this is documented by the ‘parts’ multiplicity



# SUBASSEMBLIES



# POLYMORPHISM

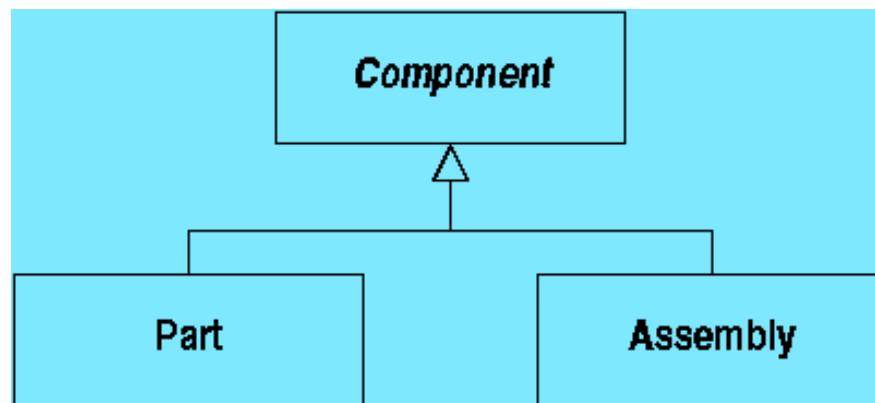
This requires assemblies to hold a mix of pointers to parts and (sub)assemblies

Inheritance is often used to implement this:

```
public abstract class Component { ... }
public class Part extends Component { ...
}
public class Assembly extends Component {
    private List<Component> components =
    new ArrayList() ;
    public void addComponent(Component c)
    {
        components.addElement(c) ;
    }
}
```

# GENERALIZATION

Java's 'extends' relationship is represented by **generalization** in UML  
— Also called *specialization* if viewed 'downwards'



# PROPERTIES OF GENERALIZATION

**Generalization is not an association:**

- does not give rise to links between objects
- usually unlabelled
- never has multiplicity annotations

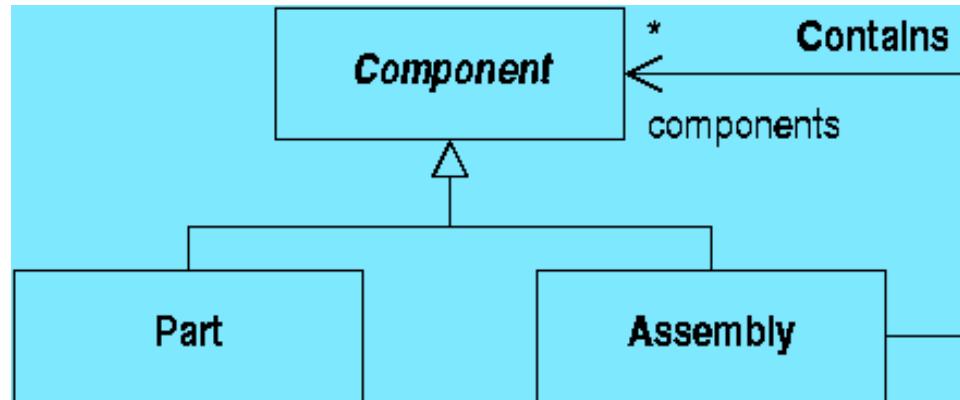
**Explained by ‘substitutability’:**

- an instance of a subclass can be used wherever an instance of a superclass is expected
- similar to ‘reference conversions’ in Java

# A POLYMORPHIC ASSOCIATION

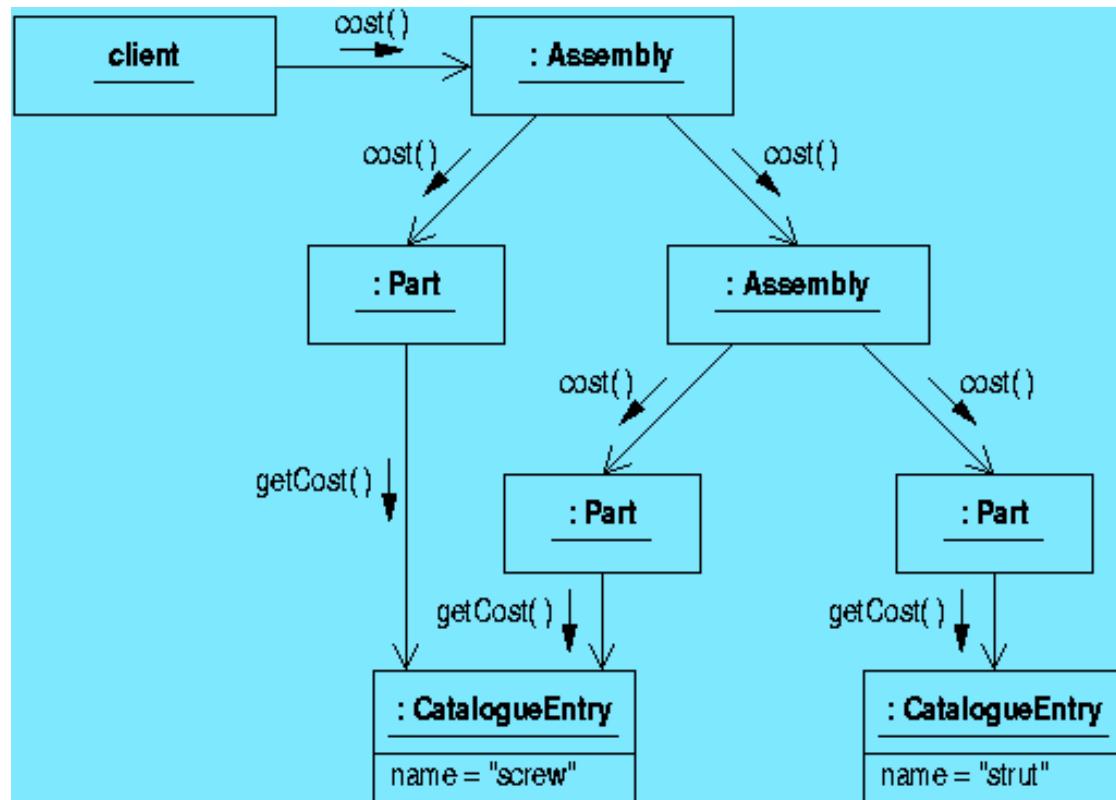
Assemblies contain Components:

- instances of ‘Contains’ can link assemblies to instances of both Component subclasses
- the Component class is *abstract* (no instances)



# THE COST OF AN ASSEMBLY

Add up the cost of all the components



# DYNAMIC BINDING

Both 'Part' and 'Assembly' classes must implement a 'cost' method

An assembly sends the 'cost' message to all its components

The run-time system checks the type of the message receiver and invokes the correct method

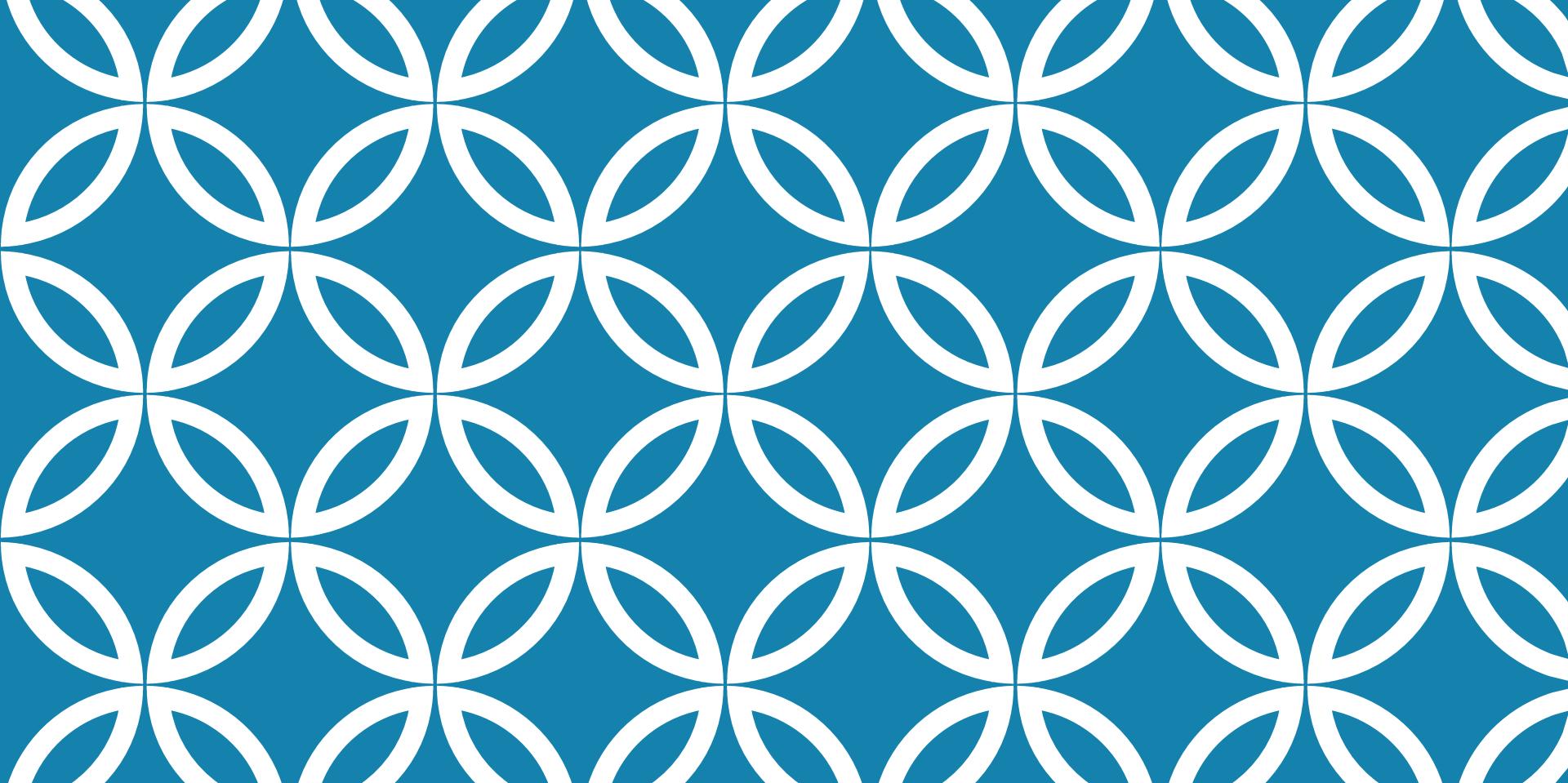
# APPLICABILITY OF OBJECT MODEL

Some classes inspired by real-world objects

Real-world doesn't necessarily provide a good software design

- eg data replication in the original ‘Part’ class

Object-orientation better understood as an effective approach to software architecture



# A OBJECT ORIENTED DESIGN PROCESS

# OO DESIGN

- The chief task of object-oriented program design is to **assign responsibilities** to the objects.
- A **responsibility** is an **obligation** of an object to other objects

# RESPONSIBILITIES

An object may be responsible for knowing:

- **What it knows** – its attributes
- **Who it knows** – the objects associated with it
- **What it knows how to do** – the operations it can perform

# RESPONSIBILITIES

(CONTINUED)

An object may also be responsible for:

- Doing something itself
- Requesting services from other objects
- Controlling and coordinating the activities of other objects

# OO DESIGN STEPS

Step 1. Produce an interaction diagram  
for each system operation  
identified during analysis.

Step 2. Produce a design class diagram  
showing the operations from the  
interaction diagrams.

Step 3. Specify the signature and the algorithm  
for each operation.

# 00 DESIGN STEPS CONTINUED<sub>(CONTINUED)</sub>

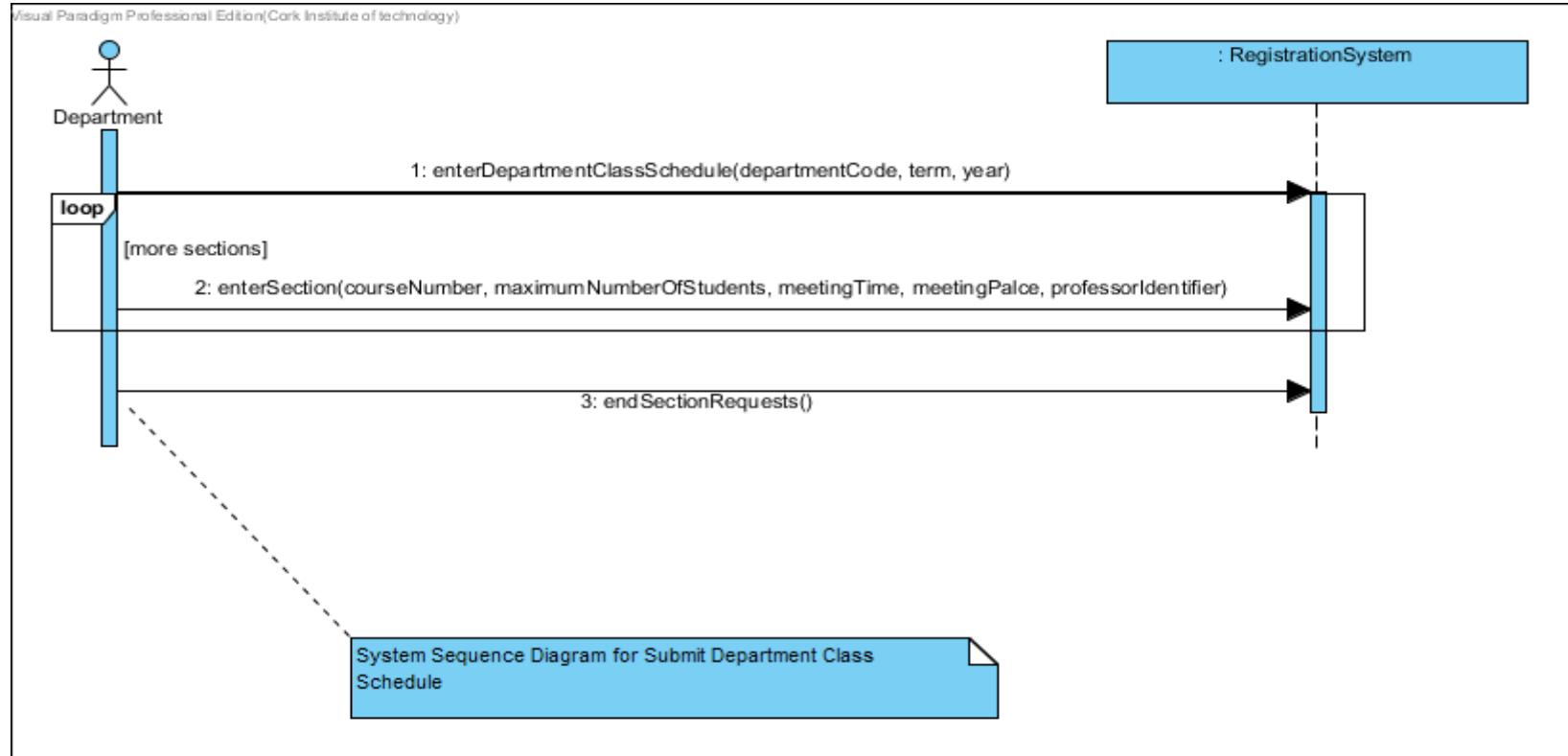
Step 4. Design the graphical user interface.

Step 5. Define the interface to the presentation layer.

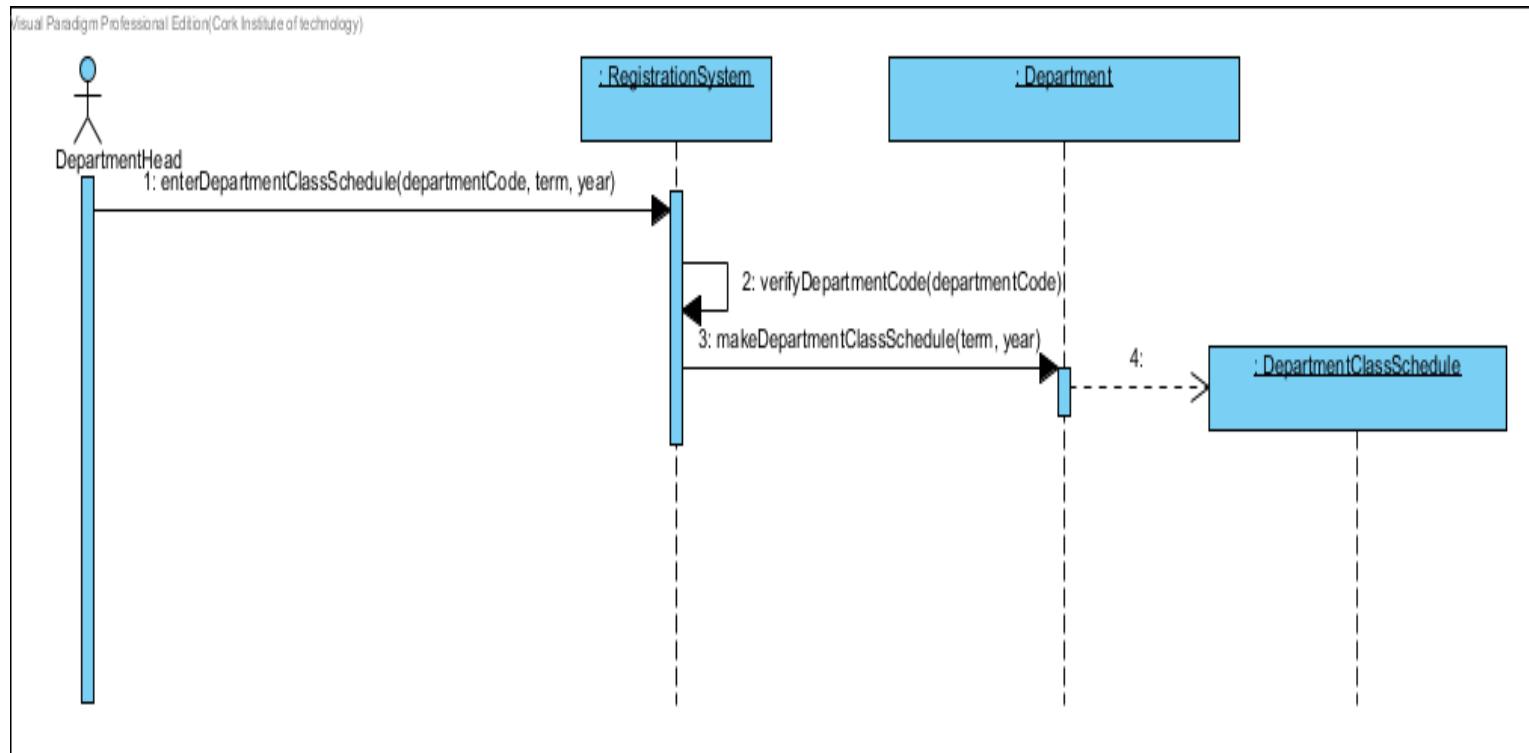
Step 6. Define the interface to the storage layer.

Step 7. Place the classes in packages.

# EXAMPLE SYSTEM SEQUENCE DIAGRAM

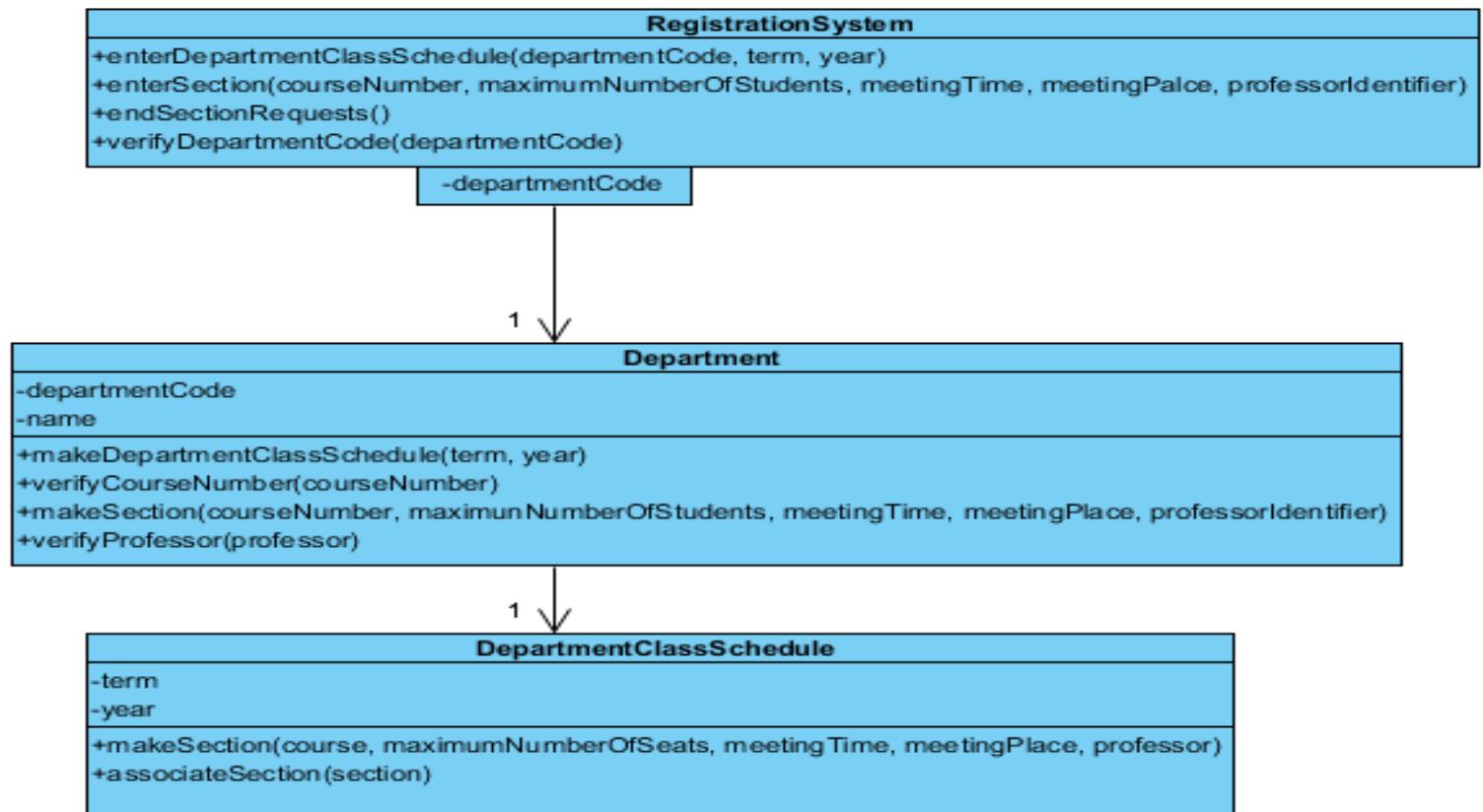


# SEQUENCE DIAGRAM SHOWING REALIZATION OF A SYSTEM OPERATION



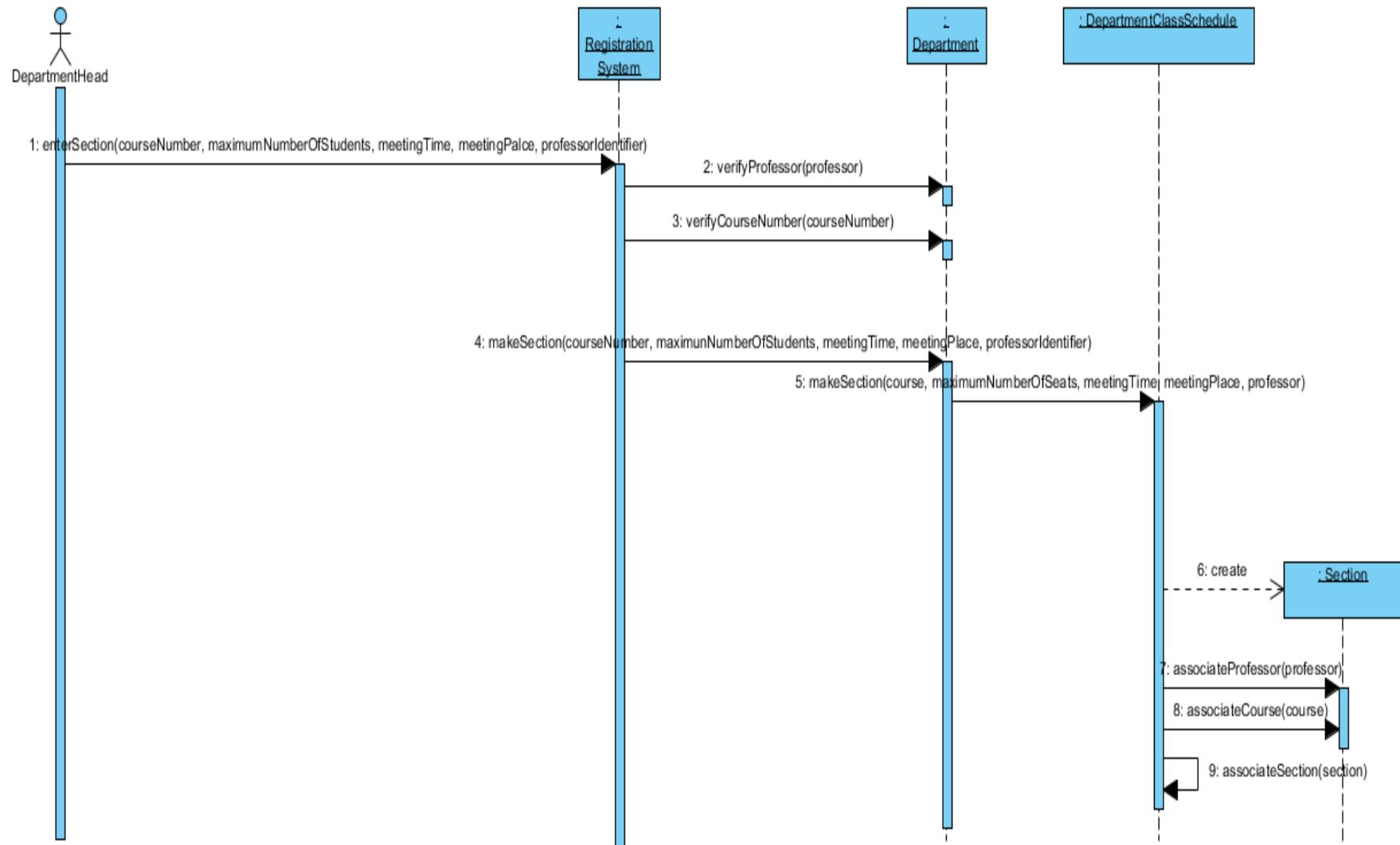
# DESIGN CLASS DIAGRAM

Visual Paradigm Professional Edition(Cork Institute of technology)

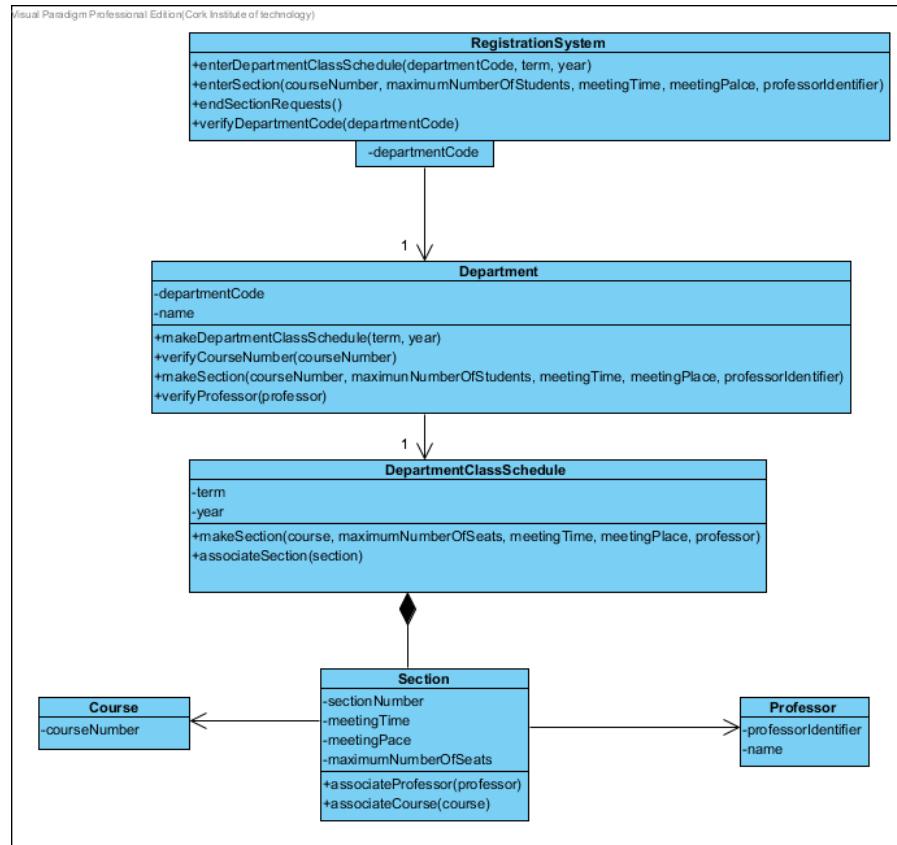


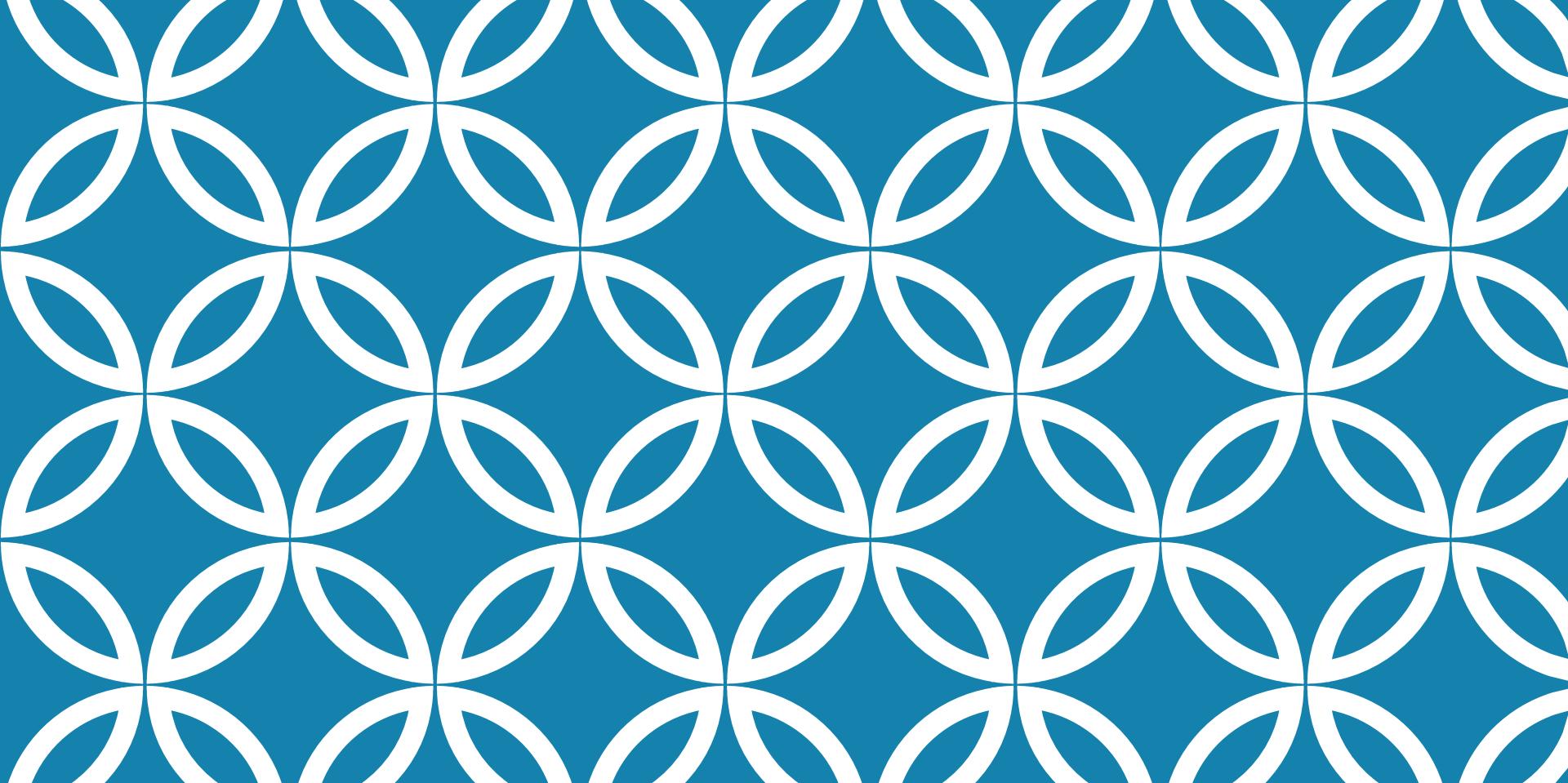
# SEQUENCE DIAGRAM SHOWING REALIZATION OF A SYSTEM OPERATION

Visual Paradigm Professional Edition(Cork Institute of technology)



# UPDATED DESIGN CLASS DIAGRAM





# UML INTERACTION DIAGRAMS

# IN THIS LECTURE YOU WILL LEARN:

- How to develop object interaction from use cases;
- How to model object interaction using an interaction sequence diagram;
- How to cross-check between interaction diagrams and a class diagram.

# GUIDELINE



*Spend time doing object modelling  
with interaction diagrams not just  
static object modelling with class  
diagrams.*

# OO DESIGN

- The chief task of object-oriented program design is to **assign responsibilities** to the objects.
- A **responsibility** is an **obligation** of an object to other objects

# RESPONSIBILITIES

An object may be responsible for knowing:

- What it knows – its attributes
- Who it knows – the objects associated with it
- What it knows how to do – the operations it can perform

# RESPONSIBILITIES

(CONTINUED)

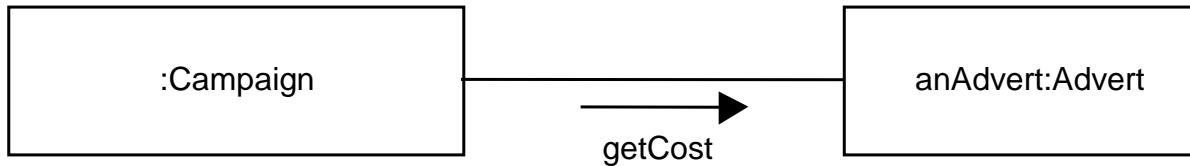
An object may also be **responsible** for:

- Doing something itself
- Requesting services from other objects
- Controlling and coordinating the activities of other objects

# OBJECT MESSAGING

Objects communicate by sending messages.  
Sending the message `getCost()` to an `Advert` object, might use the following syntax.

```
currentadvertCost = anAdvert.getCost()
```



# INTERACTION & COLLABORATION

- A **collaboration** is a group of objects or classes that work together to provide an element of functionality or behaviour.
- An **interaction** defines the message passing between lifelines (e.g. objects) within the context of a collaboration to achieve a particular behaviour.

# MODELLING INTERACTIONS

Interactions can be modelled using various notations

- Interaction sequence diagrams
- Communication diagrams
- Sequence diagrams illustrate interactions in a kind of fence format.
- Communication diagrams illustrate interactions in a graph or network format.

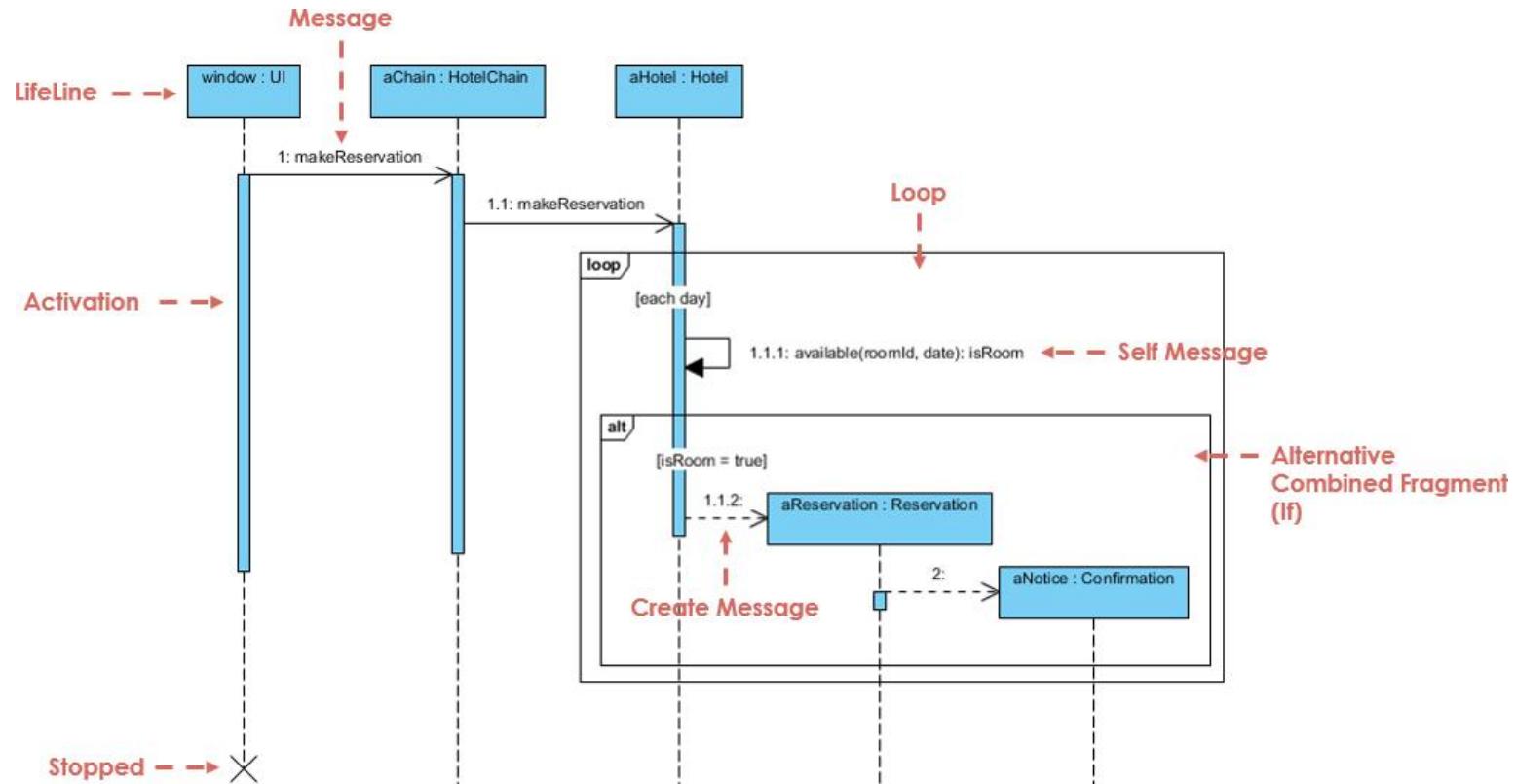
# SEQUENCE DIAGRAMS

- Shows an interaction between lifelines (e.g. objects) arranged in a time sequence.
- Can be drawn at different levels of detail and to meet different purposes at several stages in the development life cycle.
- Typically used to represent the detailed object interaction that occurs for one use case or for one operation.

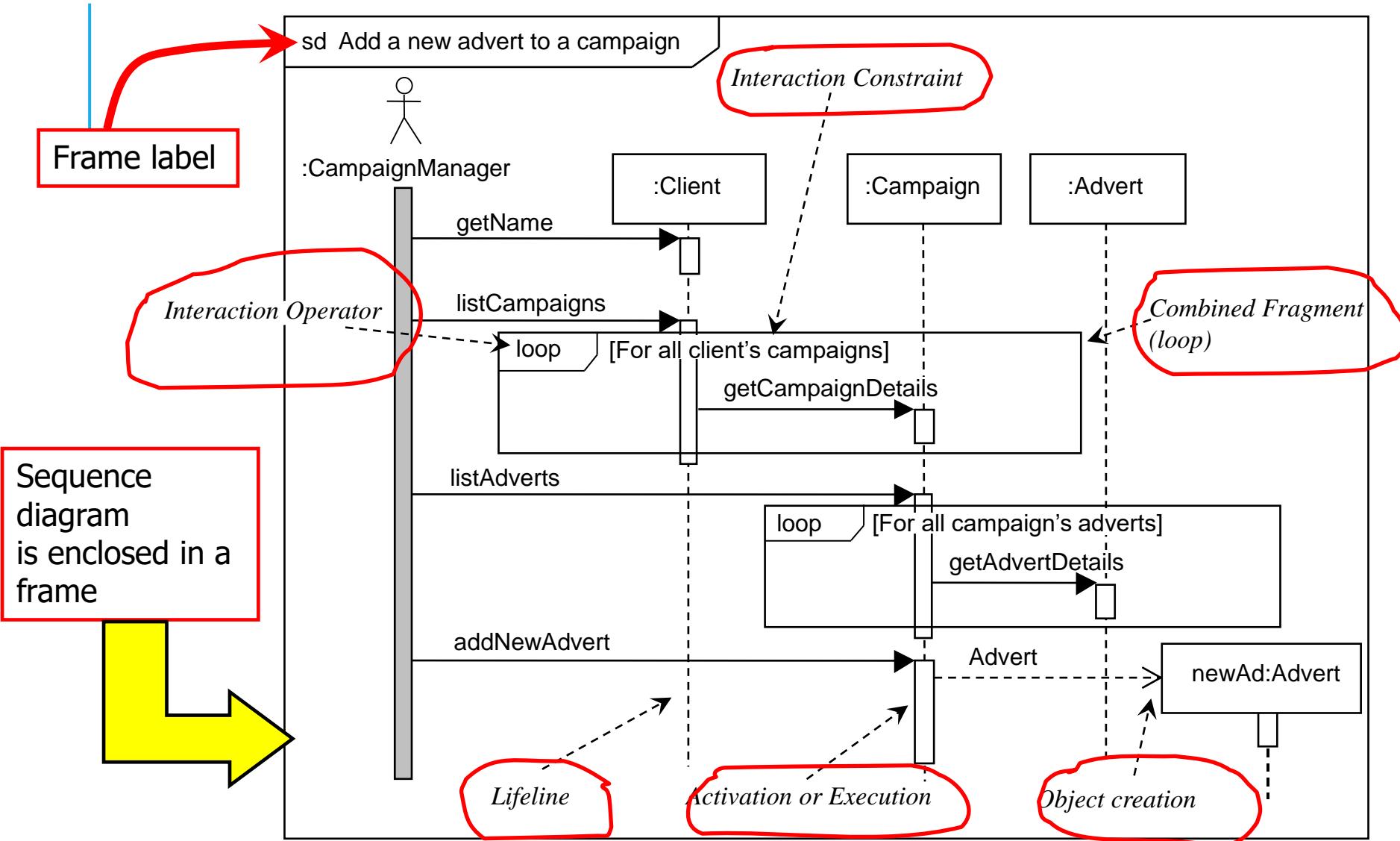
# SEQUENCE DIAGRAMS

- Vertical dimension shows time.
- Objects (or subsystems or other connectable objects) involved in interaction appear horizontally across the page and are represented by lifelines.
- Messages are shown by a solid horizontal arrow.
- The execution or activation of an operation is shown by a rectangle on the relevant lifeline.

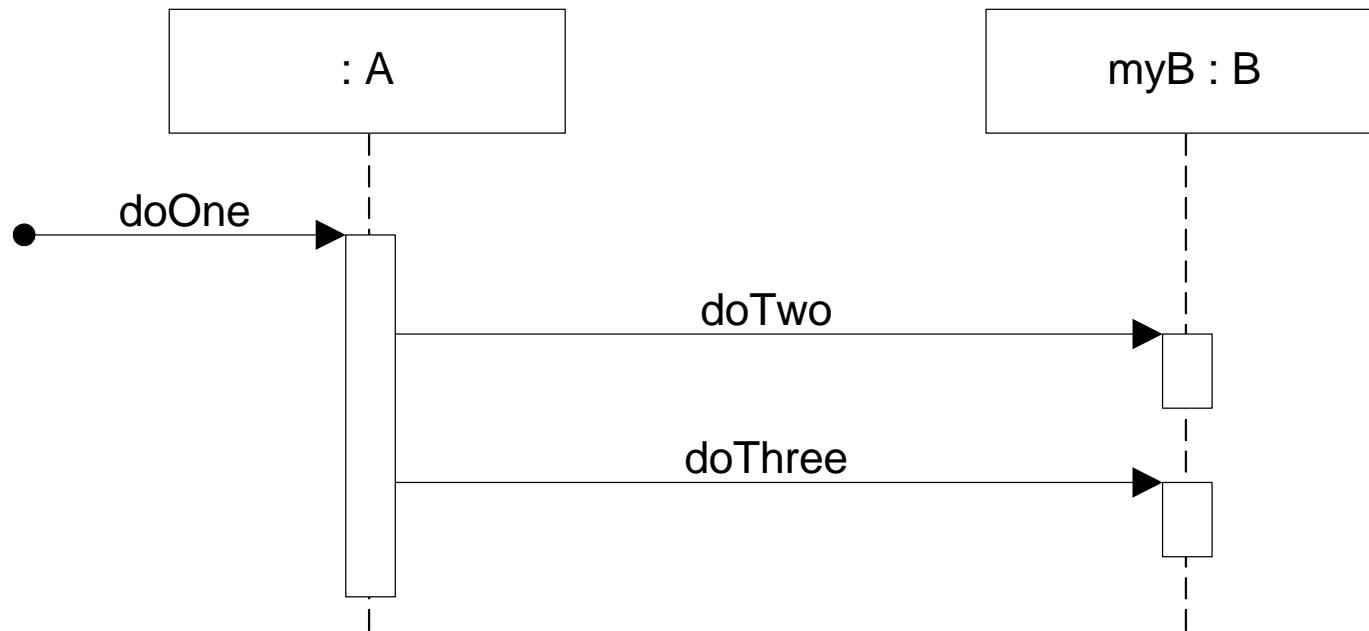
# SEQUENCE DIAGRAM



# SEQUENCE DIAGRAM

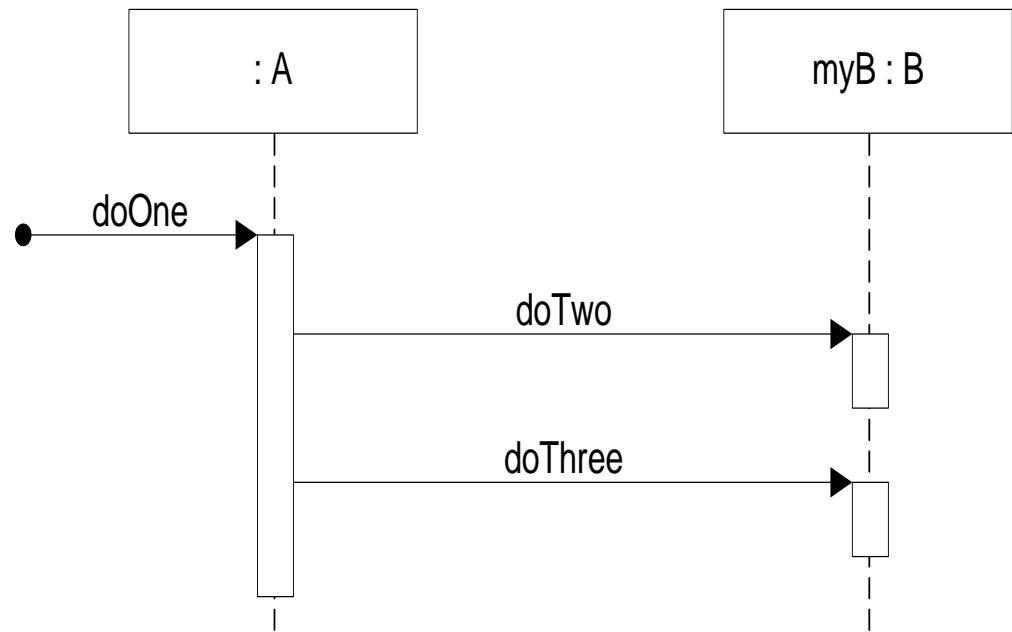


# SEQUENCE DIAGRAM

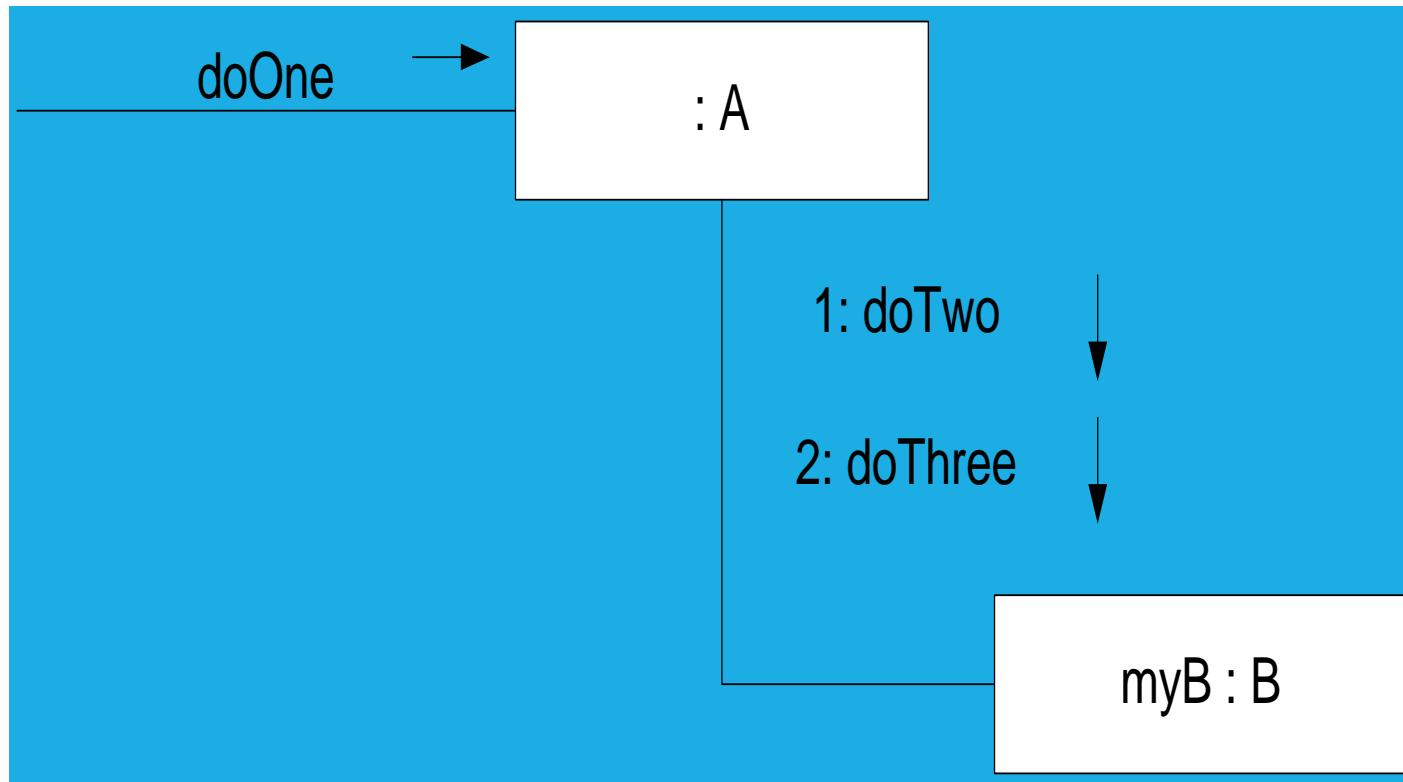


# CODE MAPPING

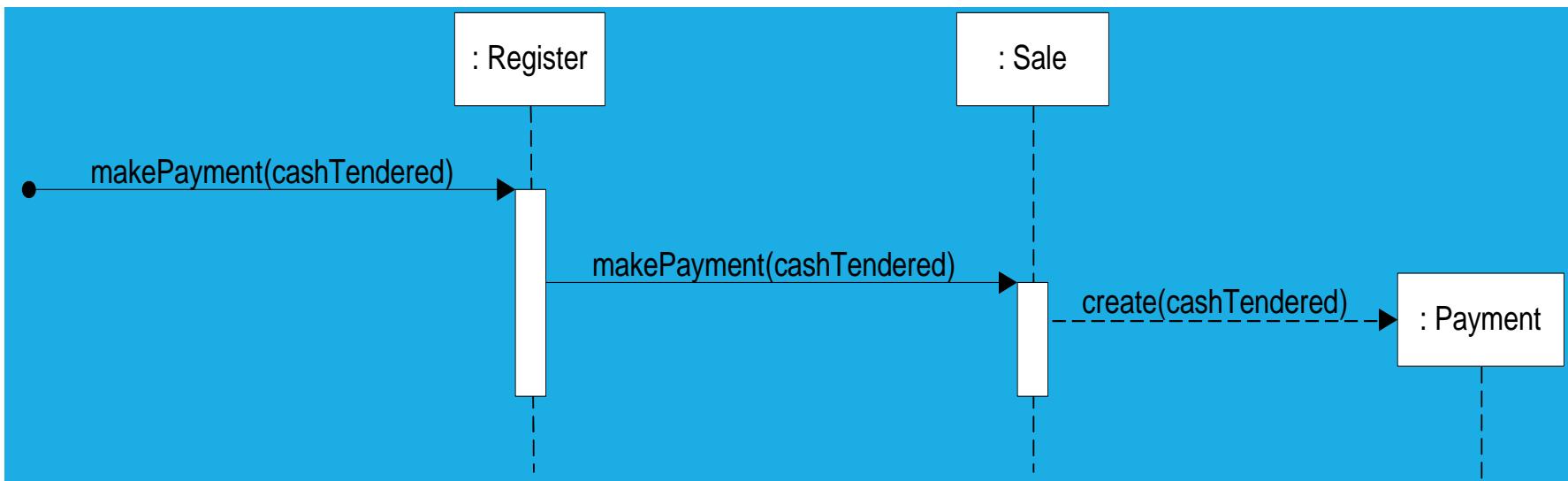
```
Public class A
{
    Private B myB = new B();
    Public void doOne()
    {
        myB.doTwo();
        myB.doThree();
    }
    //....
}
```



# COMMUNICATION DIAGRAM



# SEQUENCE DIAGRAM



# CODE MAPPING

*Public Class Sale*

```
{
```

```
Private Payment payment;
```

```
Public void makePayment(Money cashTendered)
```

```
{
```

```
    payment = new payment(cashTendered);
```

```
//...
```

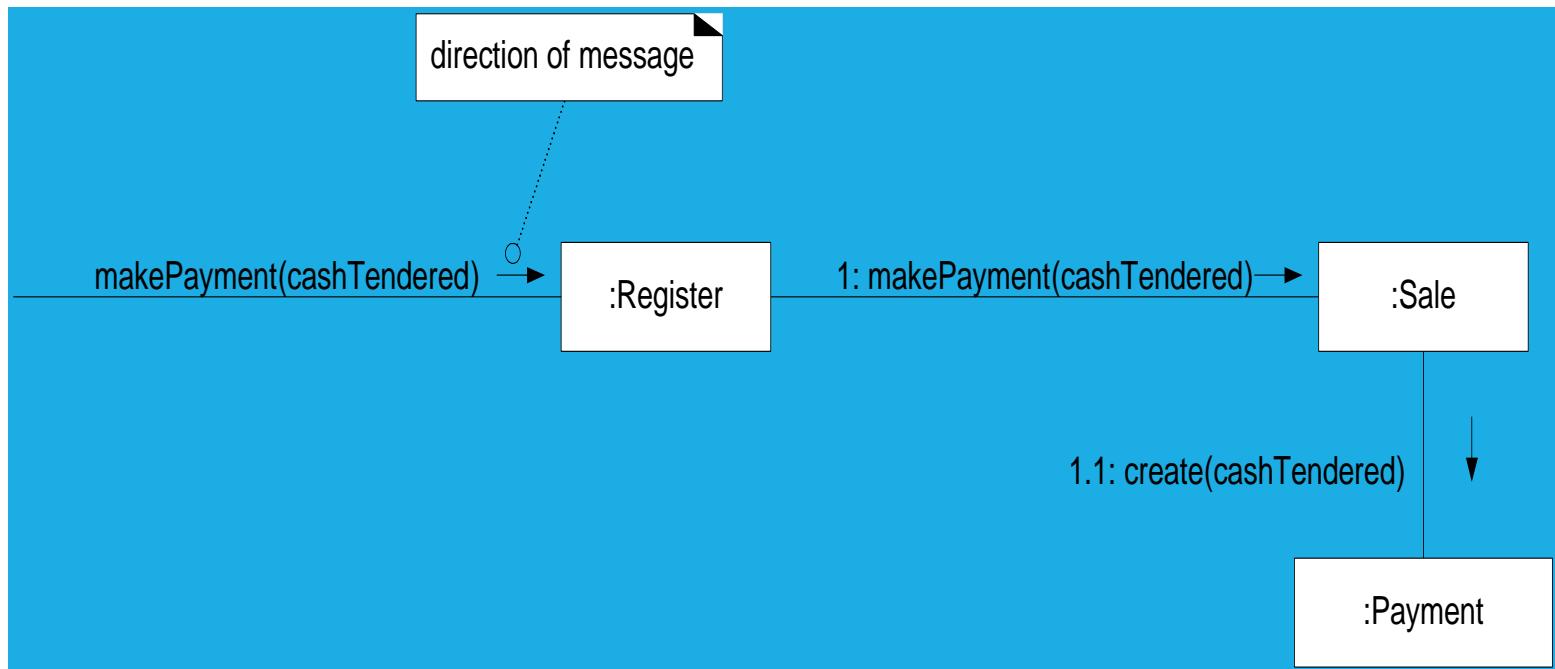
```
}
```

```
//...
```

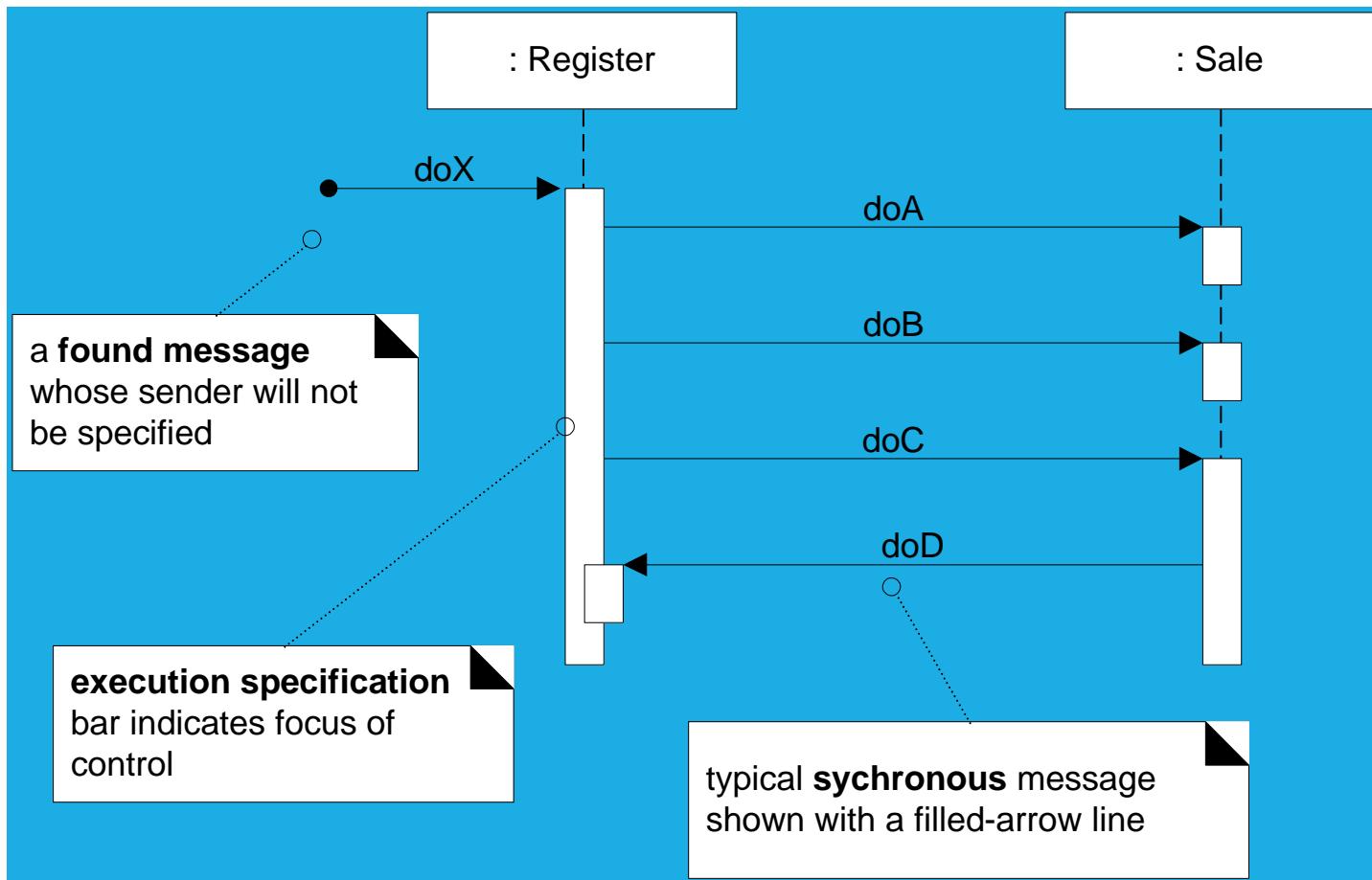
```
}
```



# COMMUNICATION DIAGRAM:MAKEPAYMENT



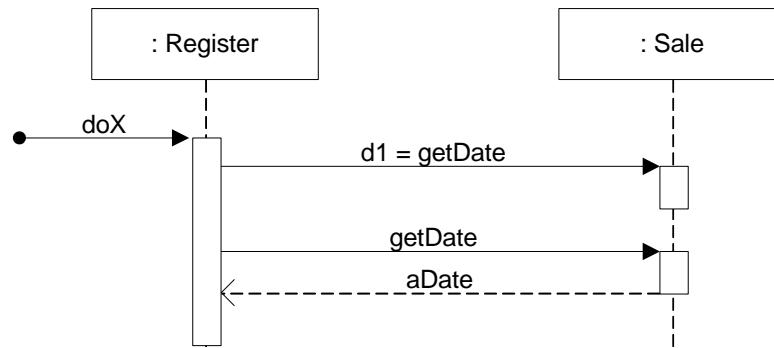
# MESSAGES AND EXECUTION SPECIFICATIONS(ACTIVATIONS)



# EXECUTION SPECIFICATION

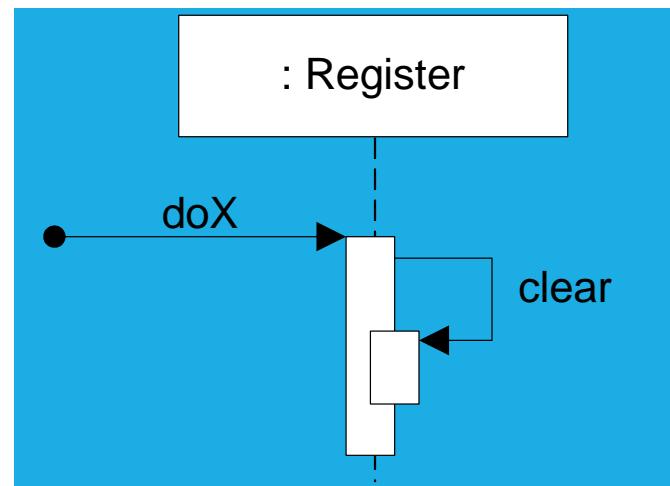
- During the time an object exists , it is shown by a dashed line.
- During the time an execution specification of a procedure on the object is active, the lifeline is drawn as a double line.
- An execution specification(**activation**) is the execution of a procedure , including any time it waits for nested procedures to execute.
- A call is shown by an arrow leading to the top of the execution specification the call initiate.
- A synchronous call is shown with a filled triangular arrowhead.

# TWO WAYS TO SHOW A RETURN RESULT FROM A MESSAGE

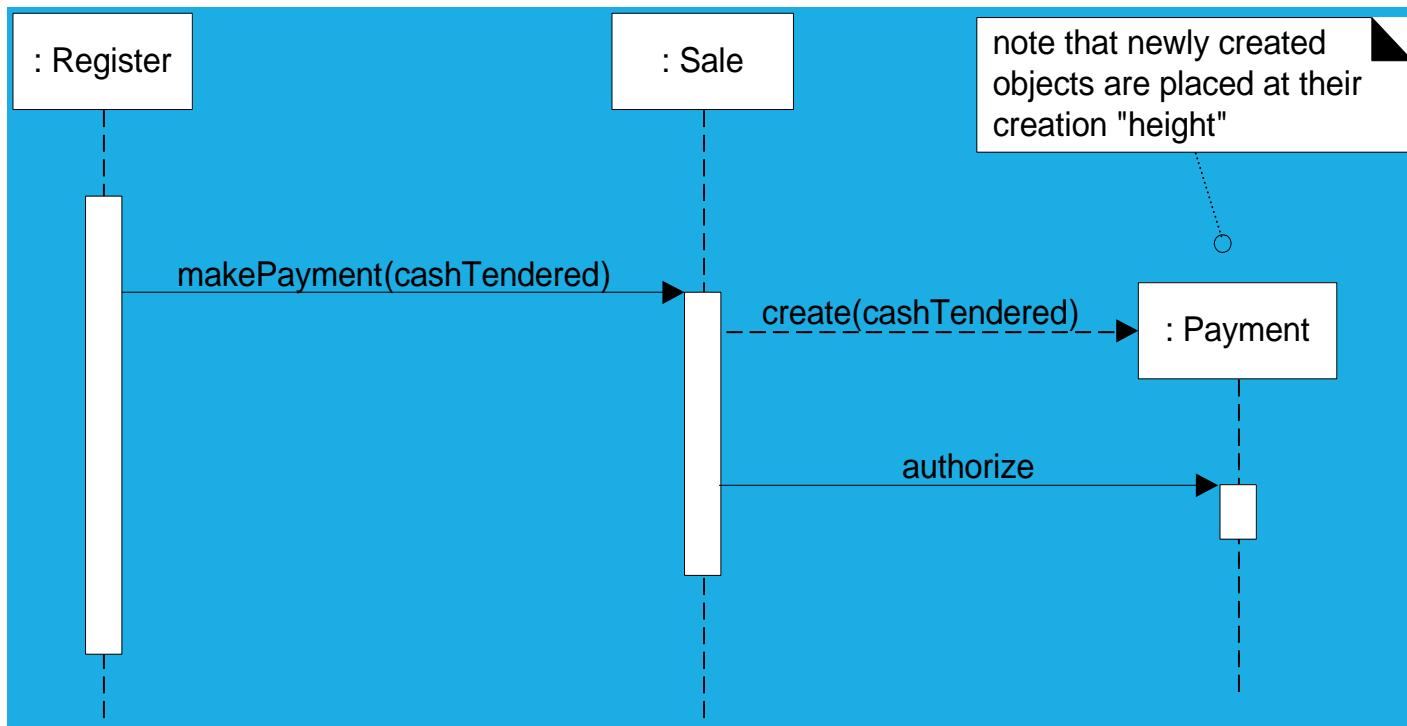


# MESSAGES TO SELF OR “THIS”

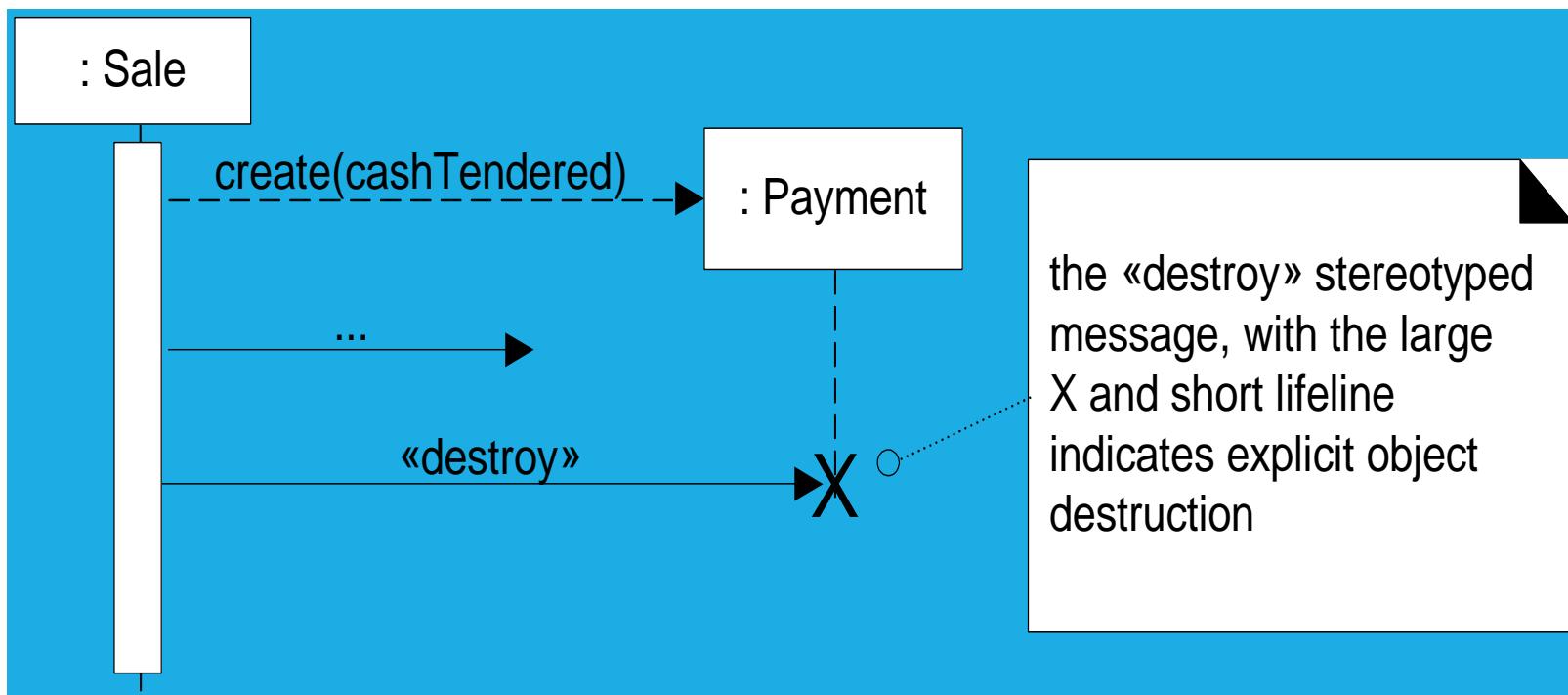
A recursive call occurs when control reenters an operation on an object, but the second call is a separate execution specification from the first



# INSTANCE CREATION .

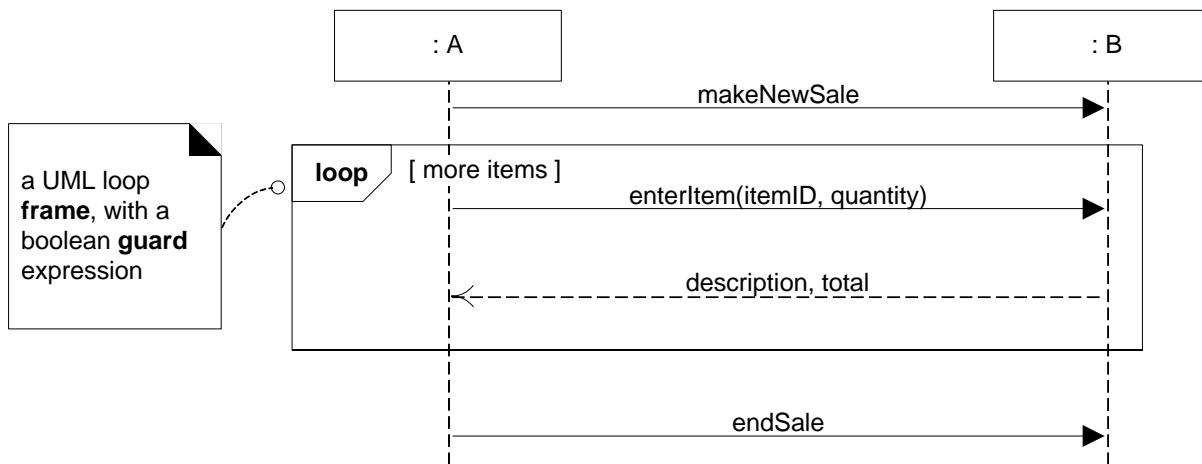


# OBJECT DESTRUCTION.



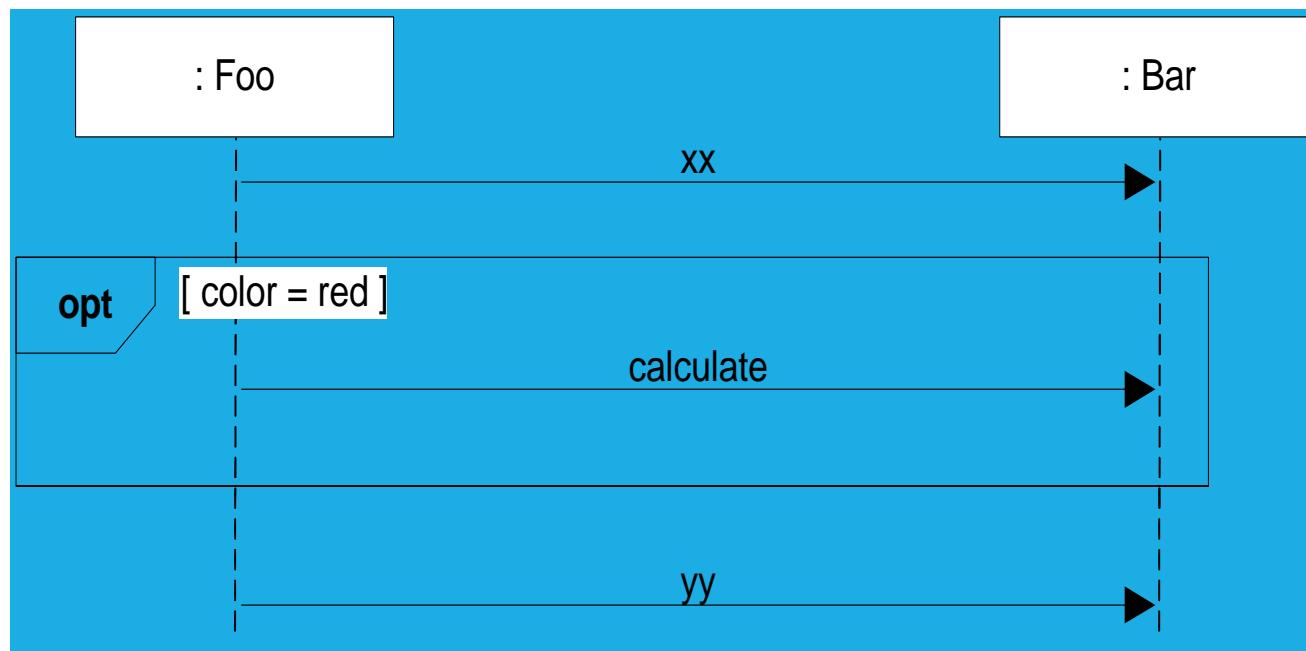
# FRAGMENTS

Fragments are regions of the diagram. They have an operator or label (such as loop) and a guard.

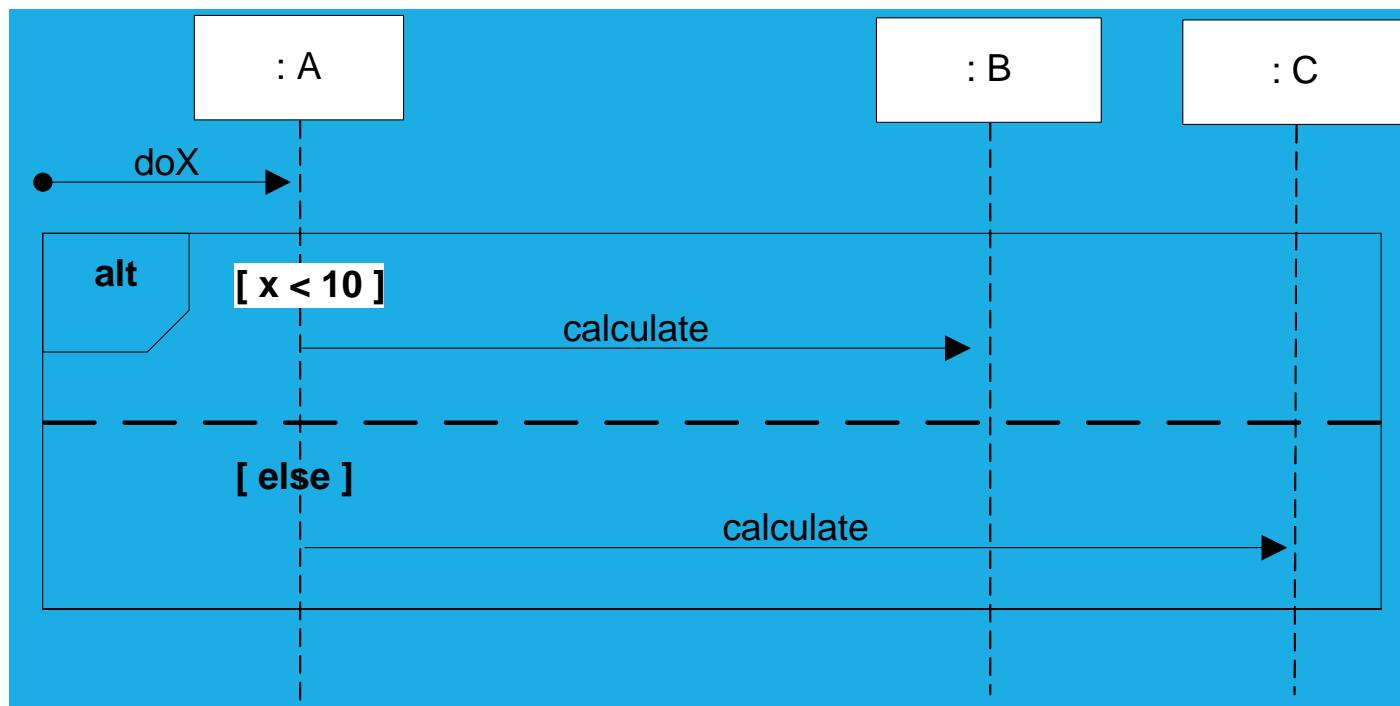


# A CONDITIONAL MESSAGE.

The guard is placed over the related lifelines.

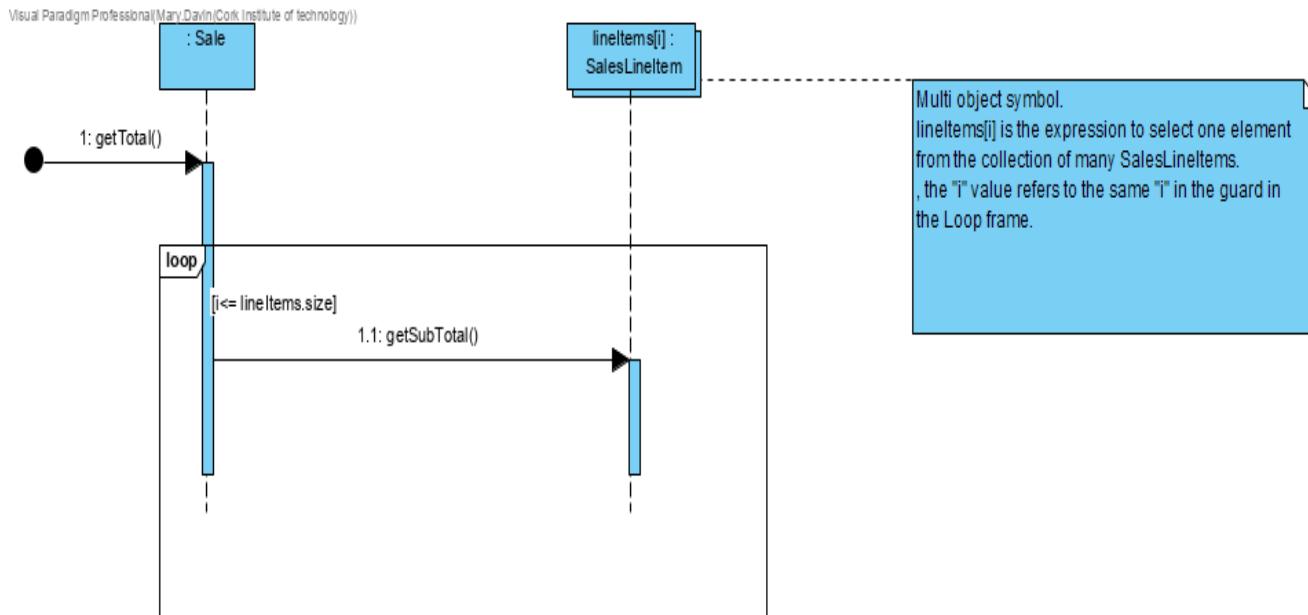


# MUTUALLY EXCLUSIVE CONDITIONAL MESSAGES.



An ALT frame is placed around the mutually exclusive alternatives.

# ITERATION OVER A COLLECTION USING RELATIVELY EXPLICIT.



# POSSIBLE JAVA MAPPING.

*Public class Sale*

{

*Private List<SalesLineItem> lineItems = new ArrayList<SalesLineItem>();*

*Public Money getTotal()*

{

*Money total = new Money(0;*

*Money subtotal = null;*

*For ( SalesLineItem lineItem : lineItems)*

{

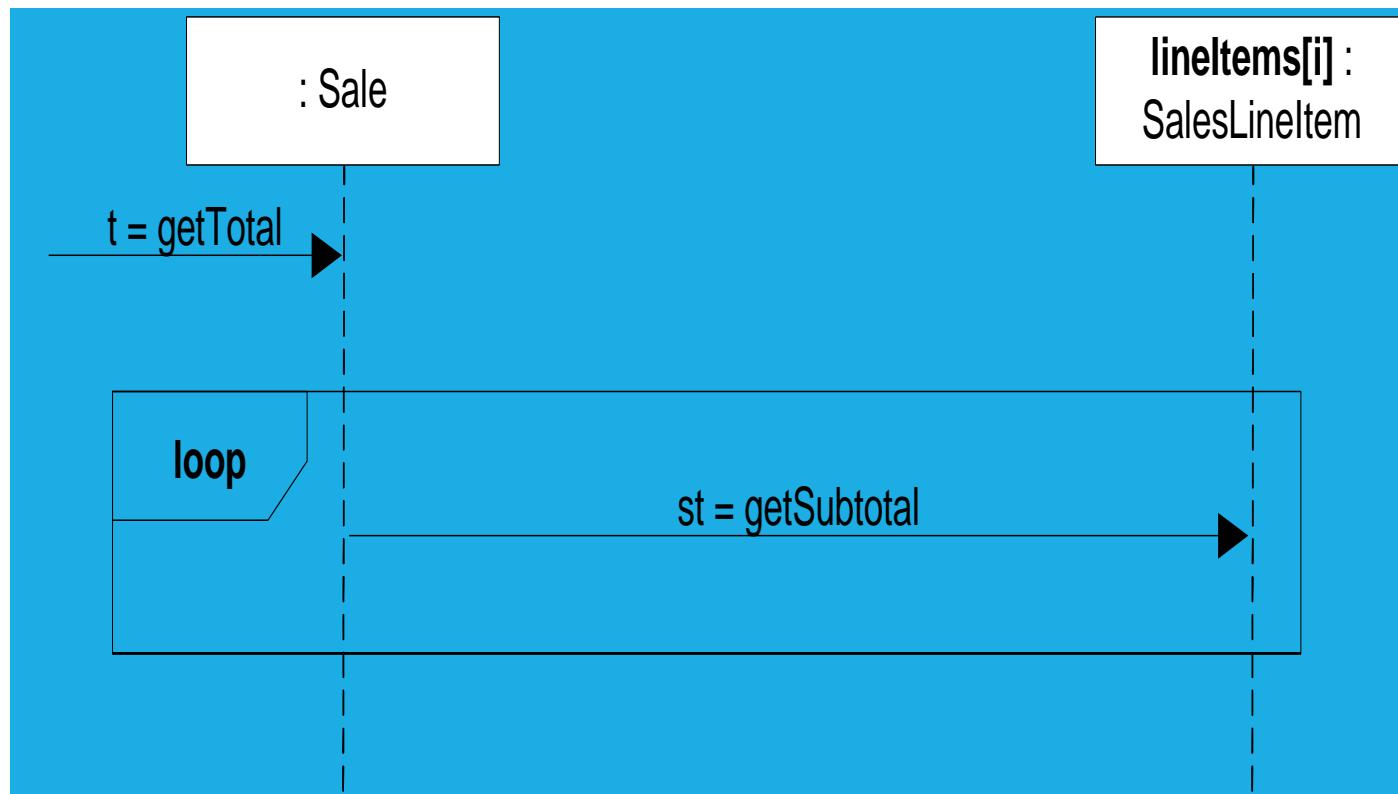
*Subtotal = lineItem.getSubtotal();*

*Total.add(subtotal);*

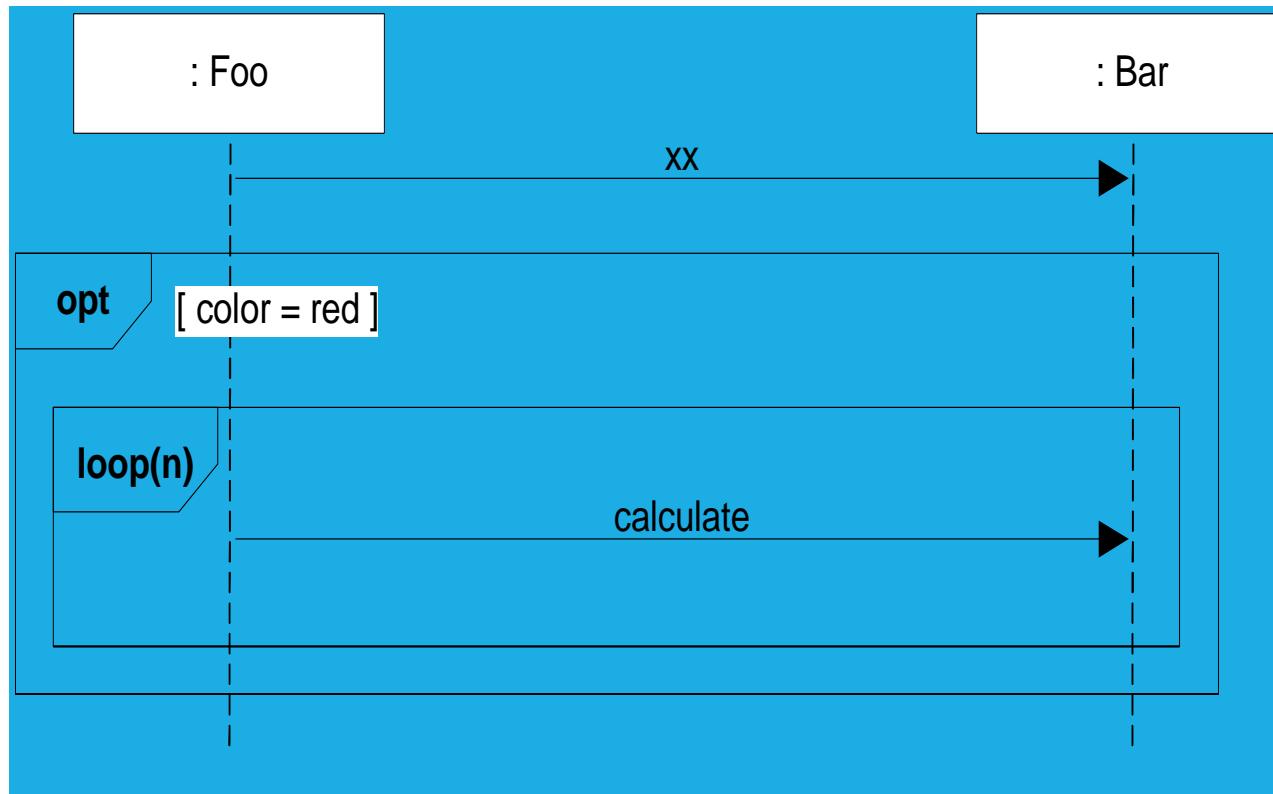
}

*return total;*

# ITERATION OVER A COLLECTION LEAVING THINGS MORE IMPLICIT.



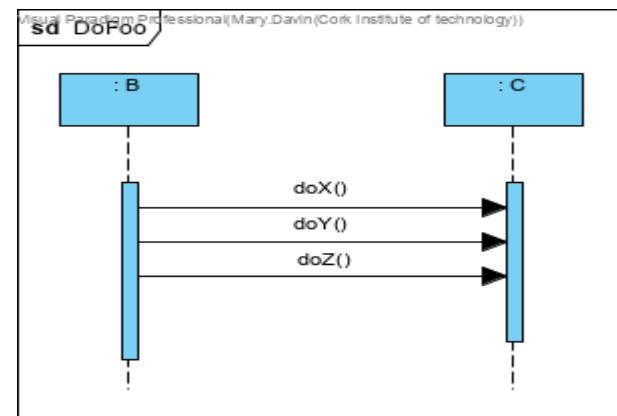
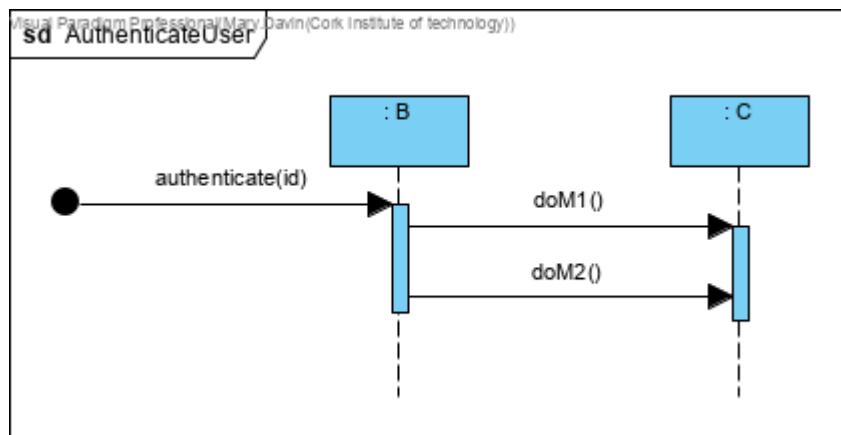
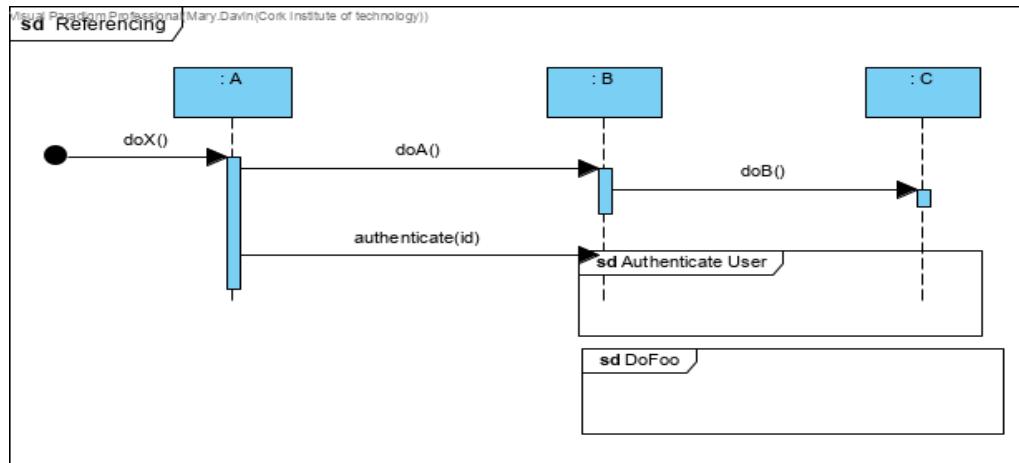
# NESTED FRAMES



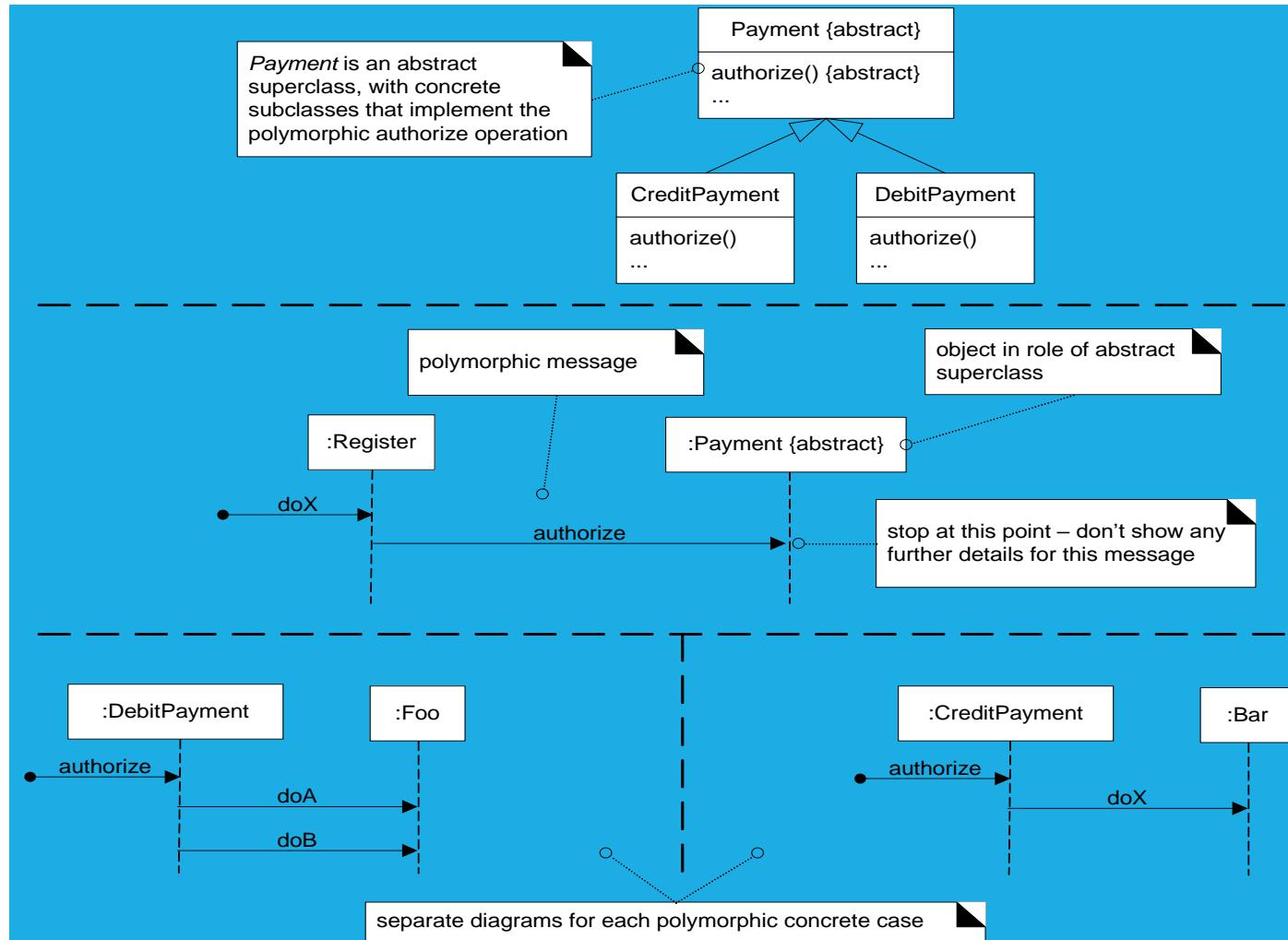
# HOW TO RELATE INTERACTION DIAGRAMS

- They are created with two related frames.
- A frame around an entire sequence diagram , labelled with the tag **sd** and a name.
- A frame tagged **ref**, called a reference,that refers to another named sequence diagram;

# HOW TO RELATE SEQUENCE DIAGRAMS



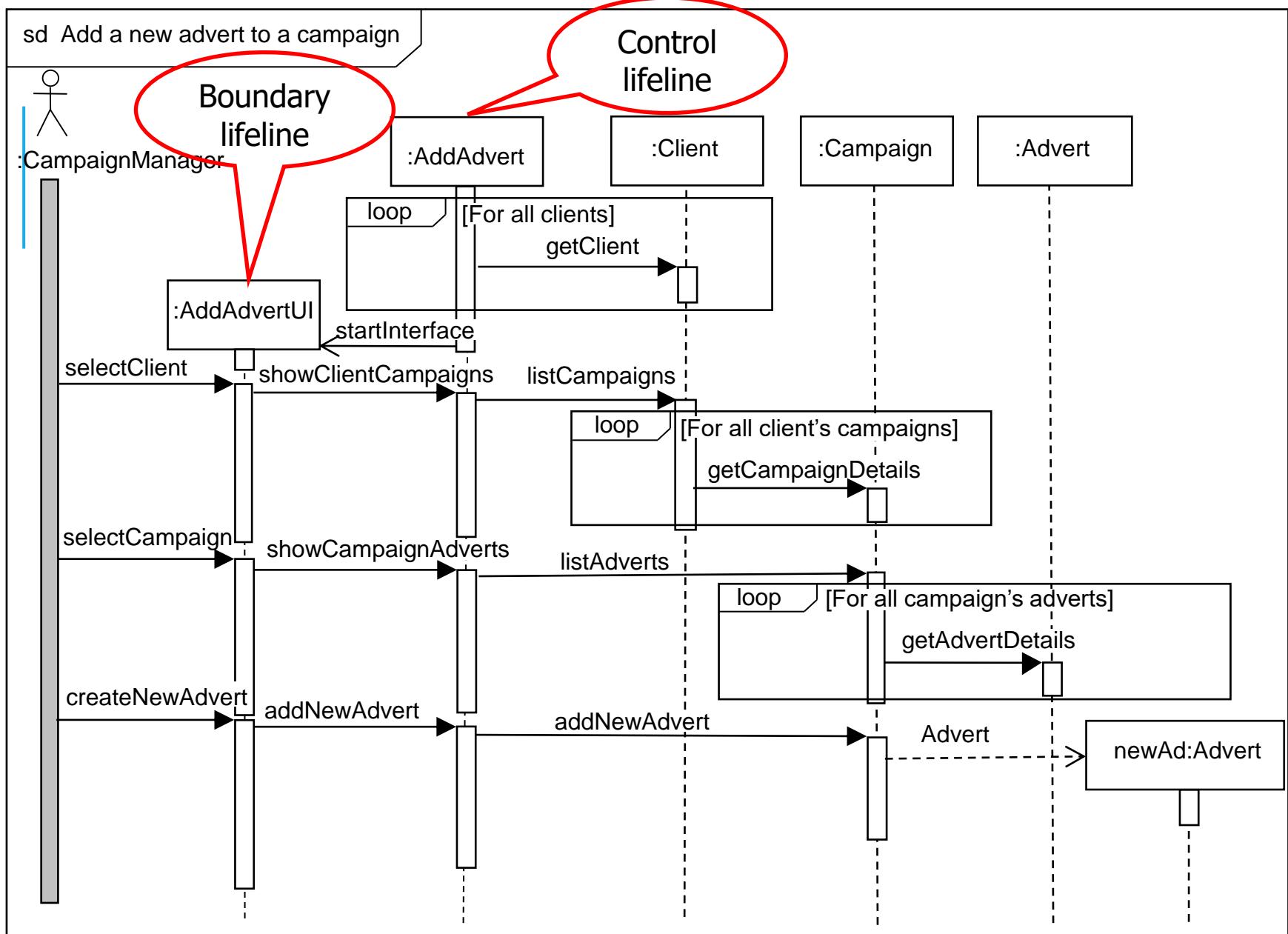
# AN APPROACH TO MODELLING POLYMORPHIC CASES IN SEQUENCE DIAGRAMS



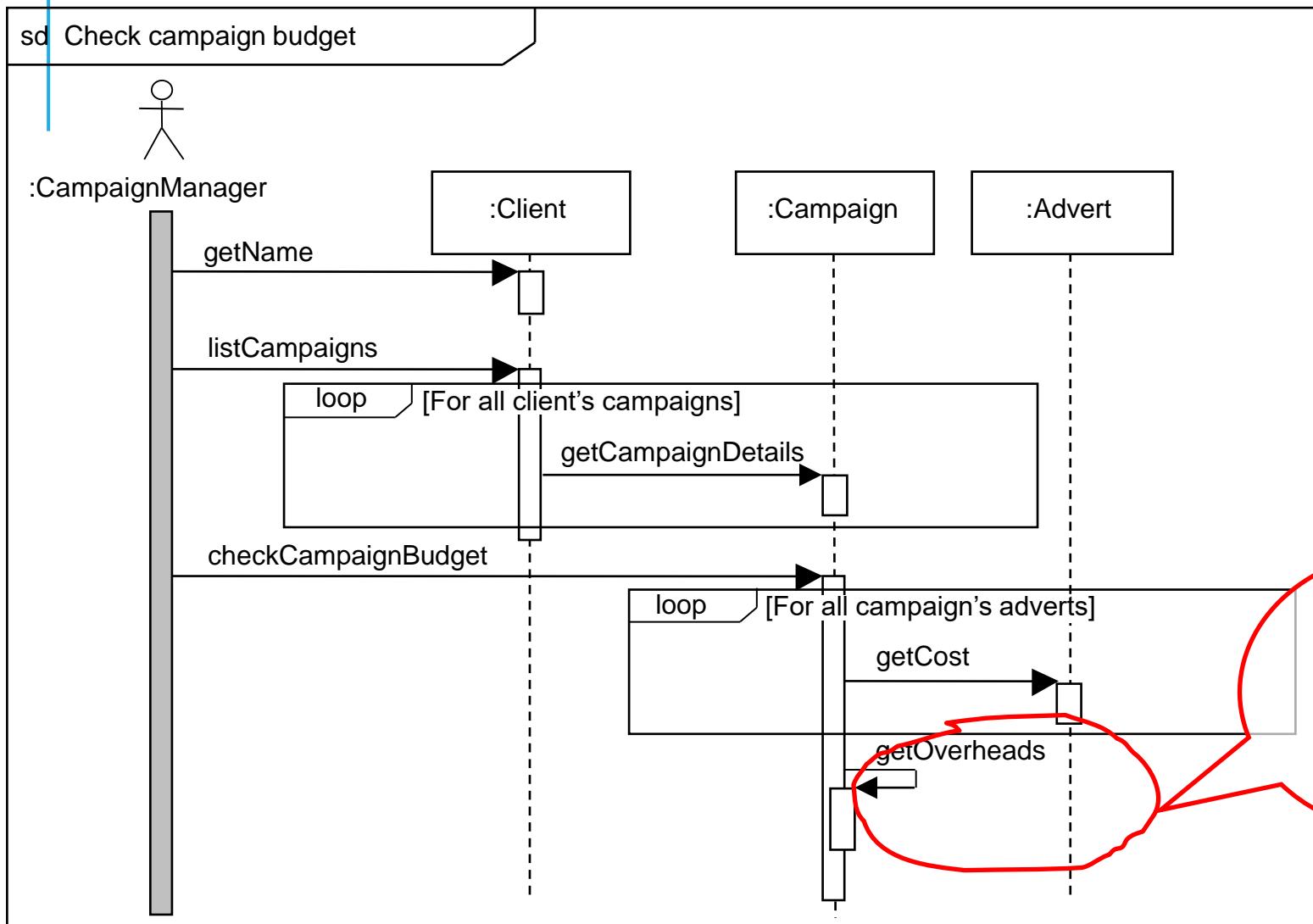
# BOUNDARY & CONTROL CLASSES

Most use cases imply at least one **boundary object** that manages the dialogue between the actor and the system – in the next sequence diagram it is represented by the lifeline :AddAdvertUI

The **control object** is represented by the lifeline :AddAdvert and this manages the overall object communication.



# REFLEXIVE MESSAGES

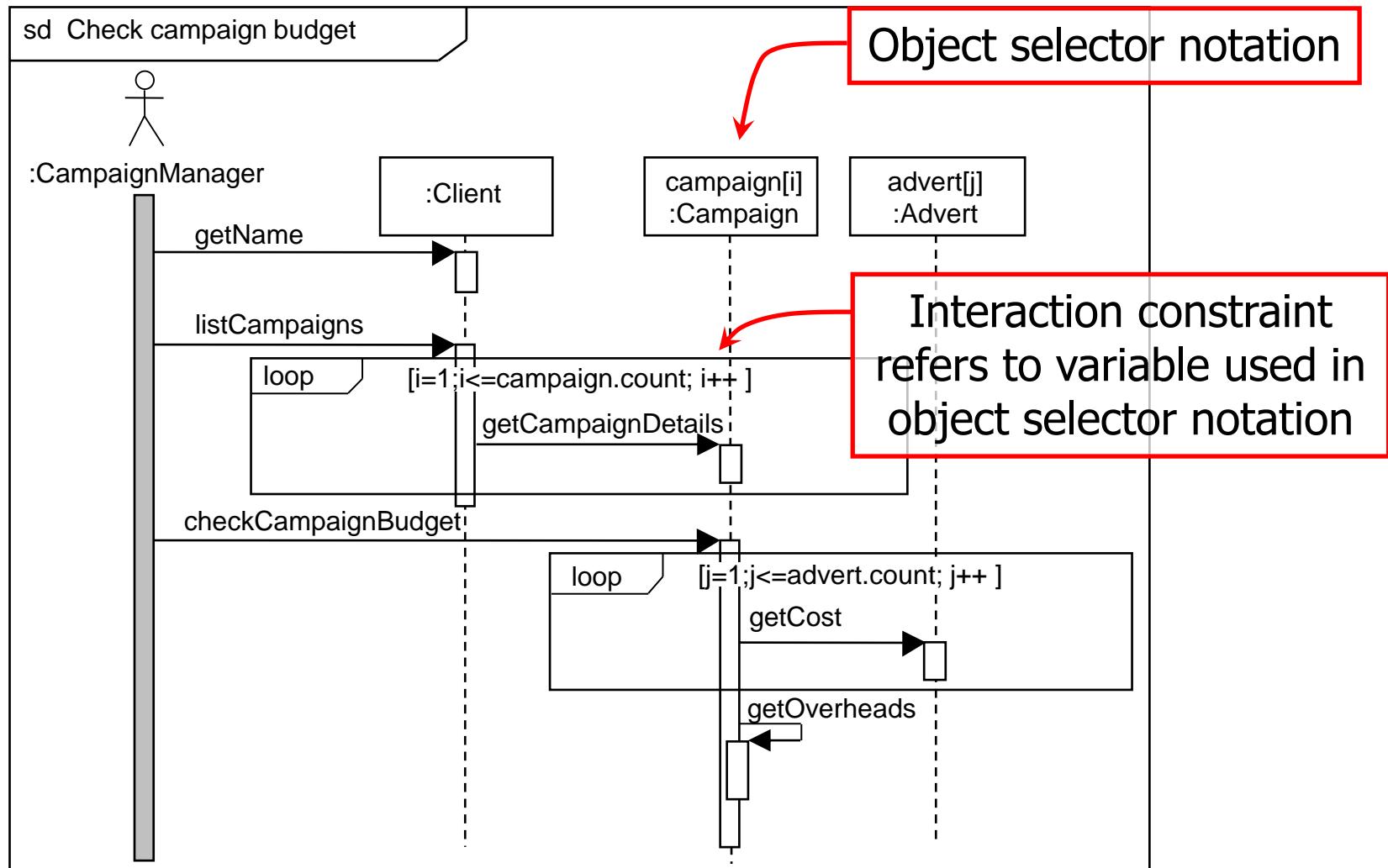


Reflexive message

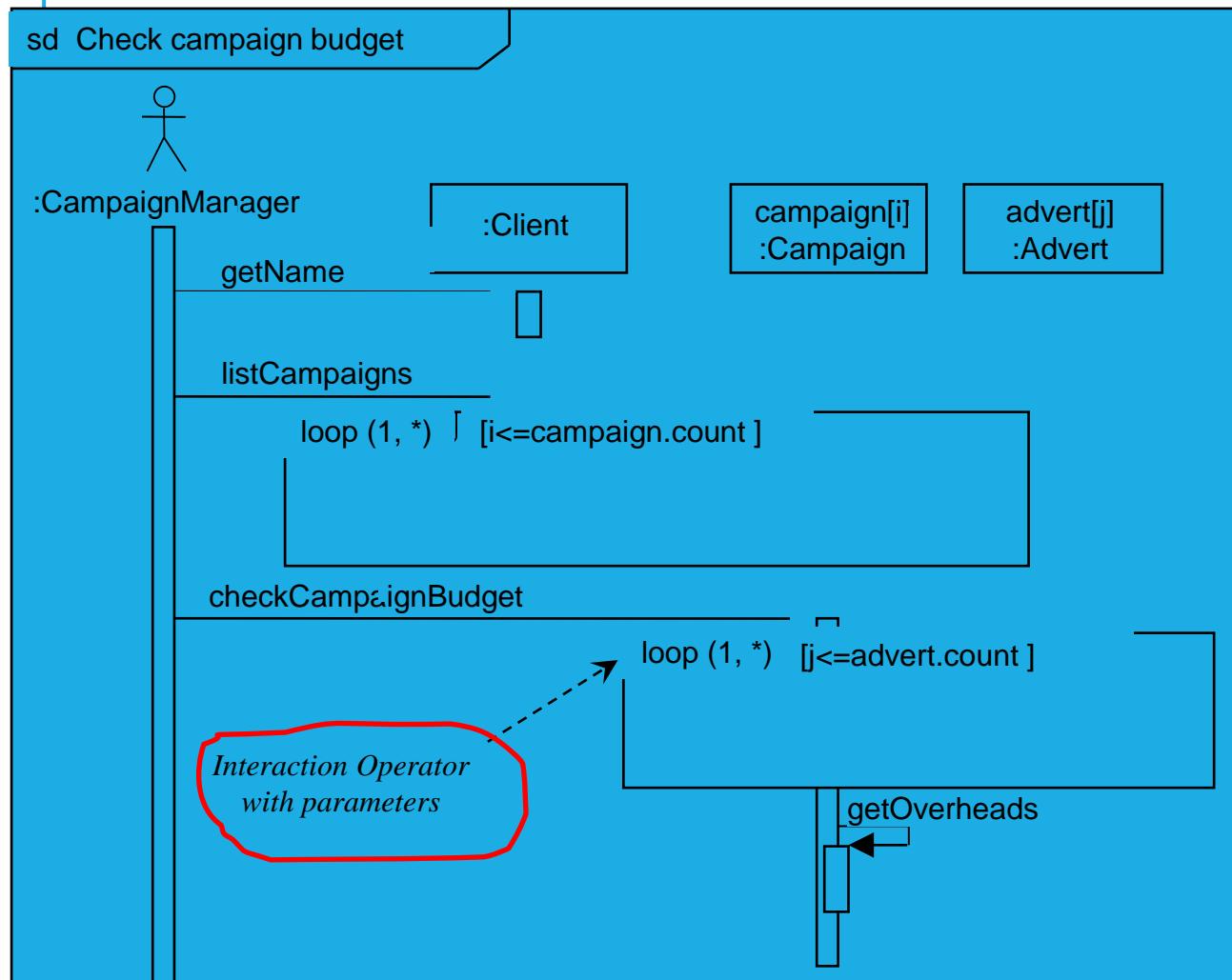
# REPLY MESSAGE

- A *reply* message returns the control to the object that originated the message that began the activation.
- **Reply messages** are shown with a dashed arrow, but it is optional to show them at all since it can be assumed that control is returned to the originating object at the end of the

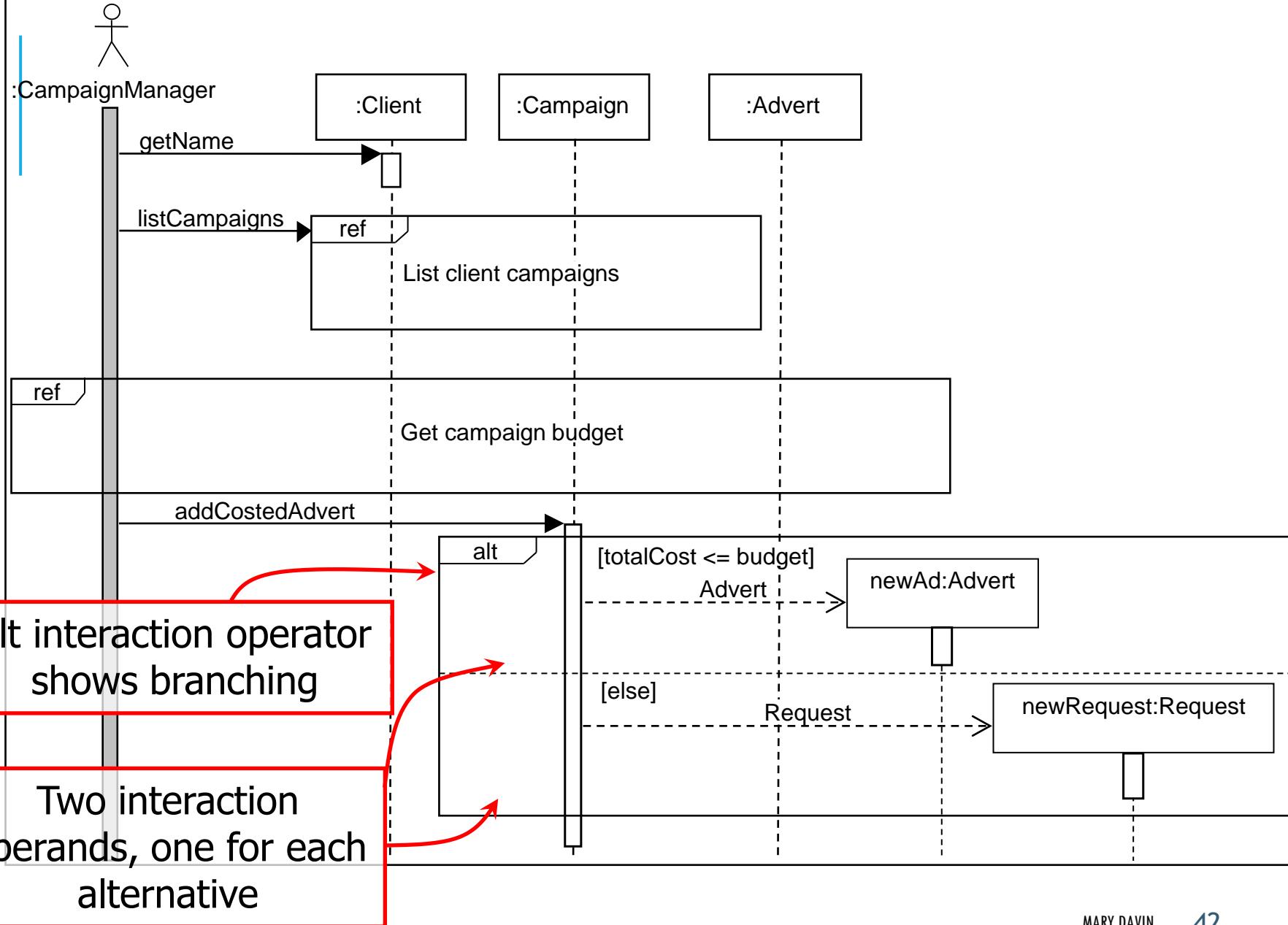
# OBJECT SELECTOR NOTATION



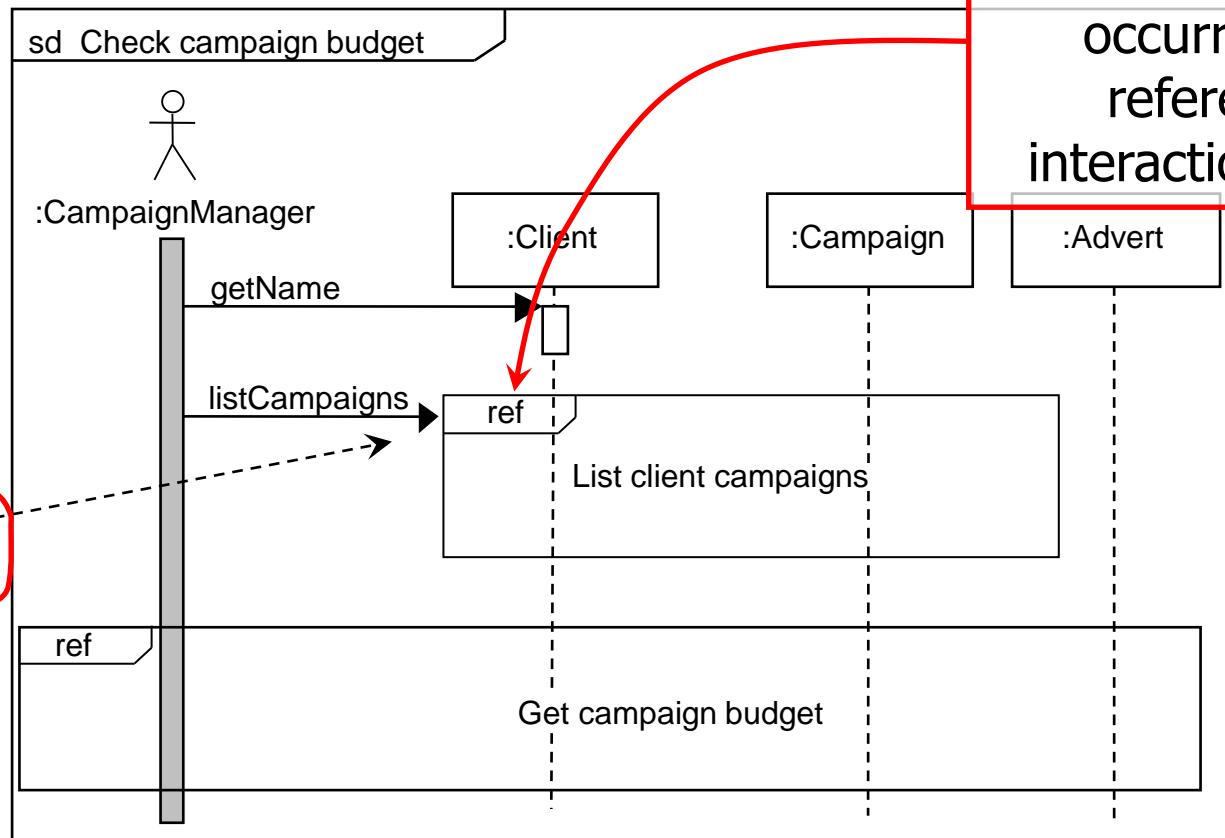
# INTERACTION OPERATORS



sd Add a new advert to a campaign if within budget

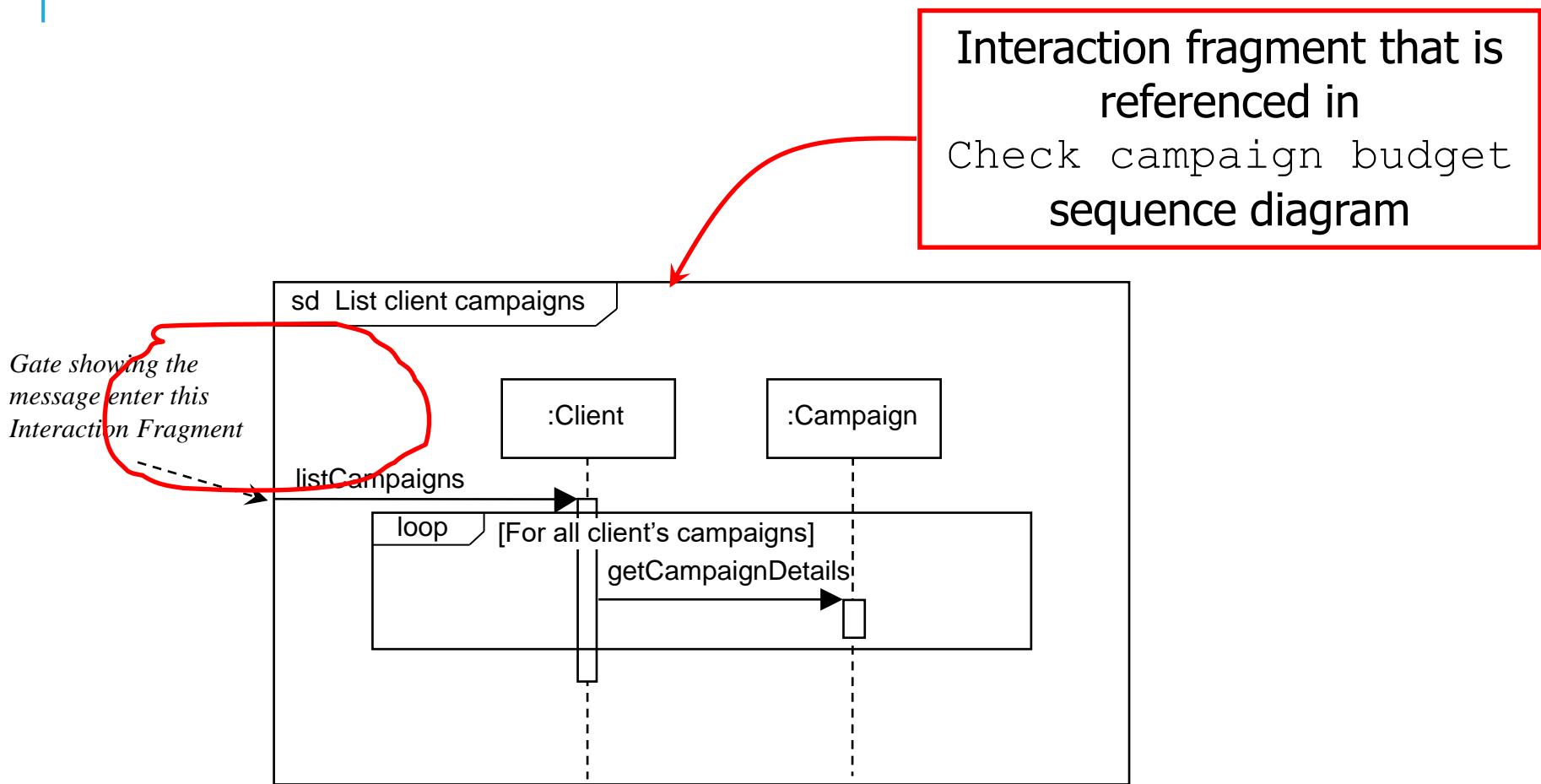


# USING INTERACTION FRAGMENTS

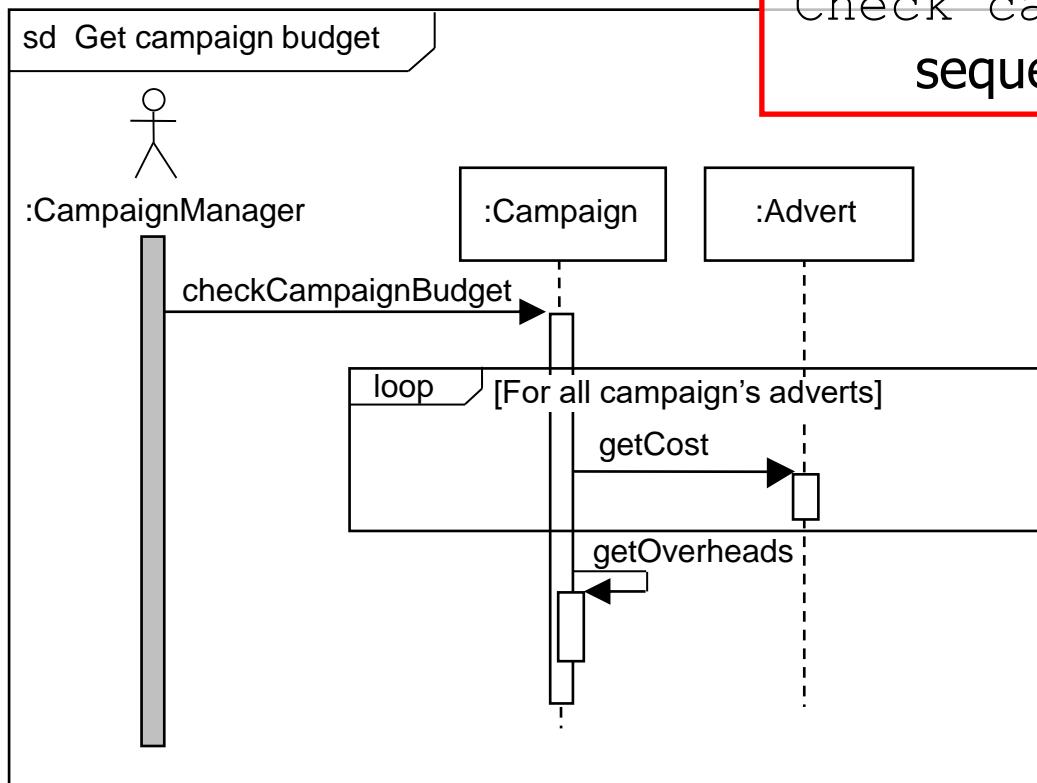


ref interaction operator indicates interaction occurrence that references an interaction fragment

# INTERACTION FRAGMENT

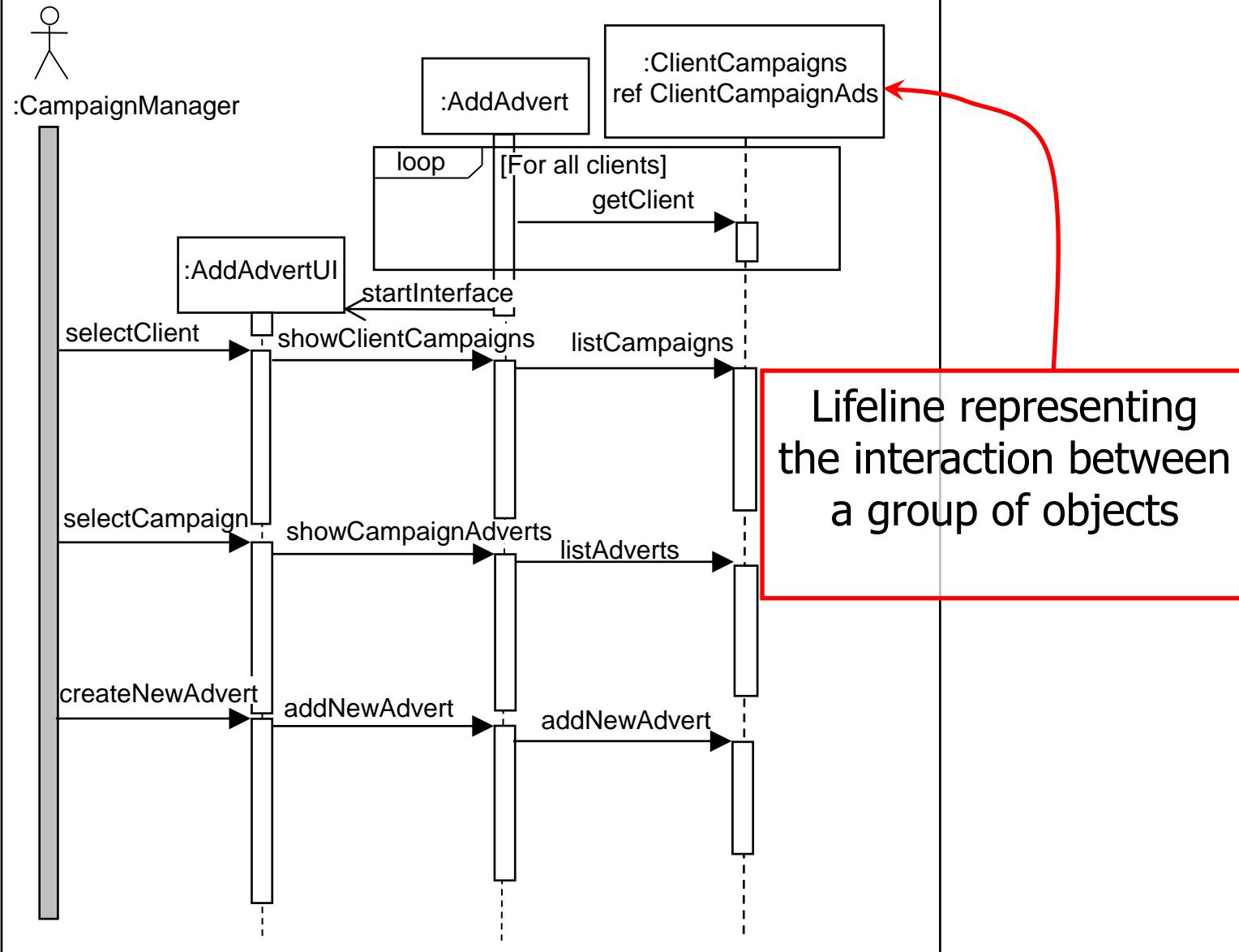


# INTERACTION FRAGMENT

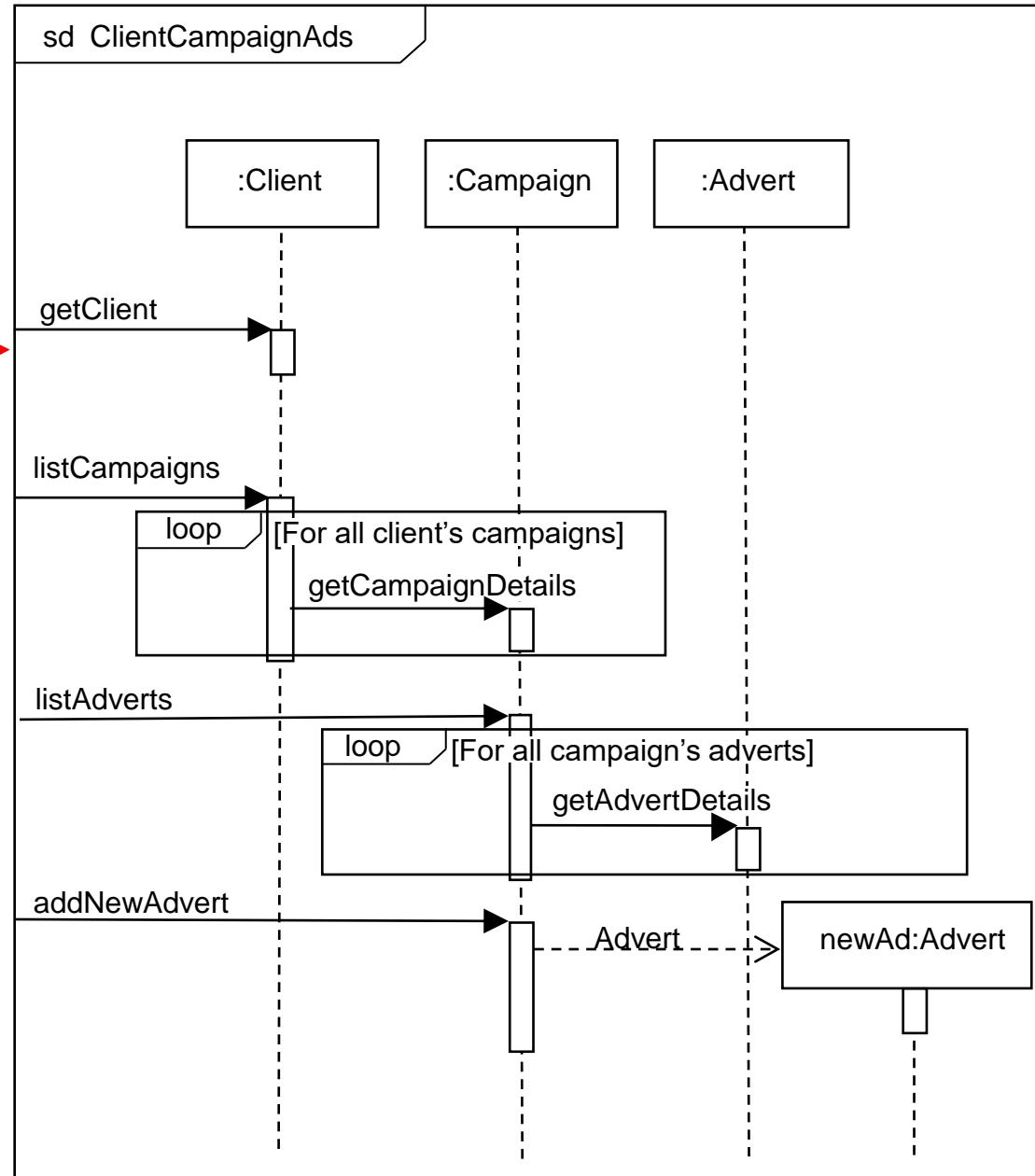


Interaction fragment that is  
also referenced in  
Check campaign budget  
sequence diagram

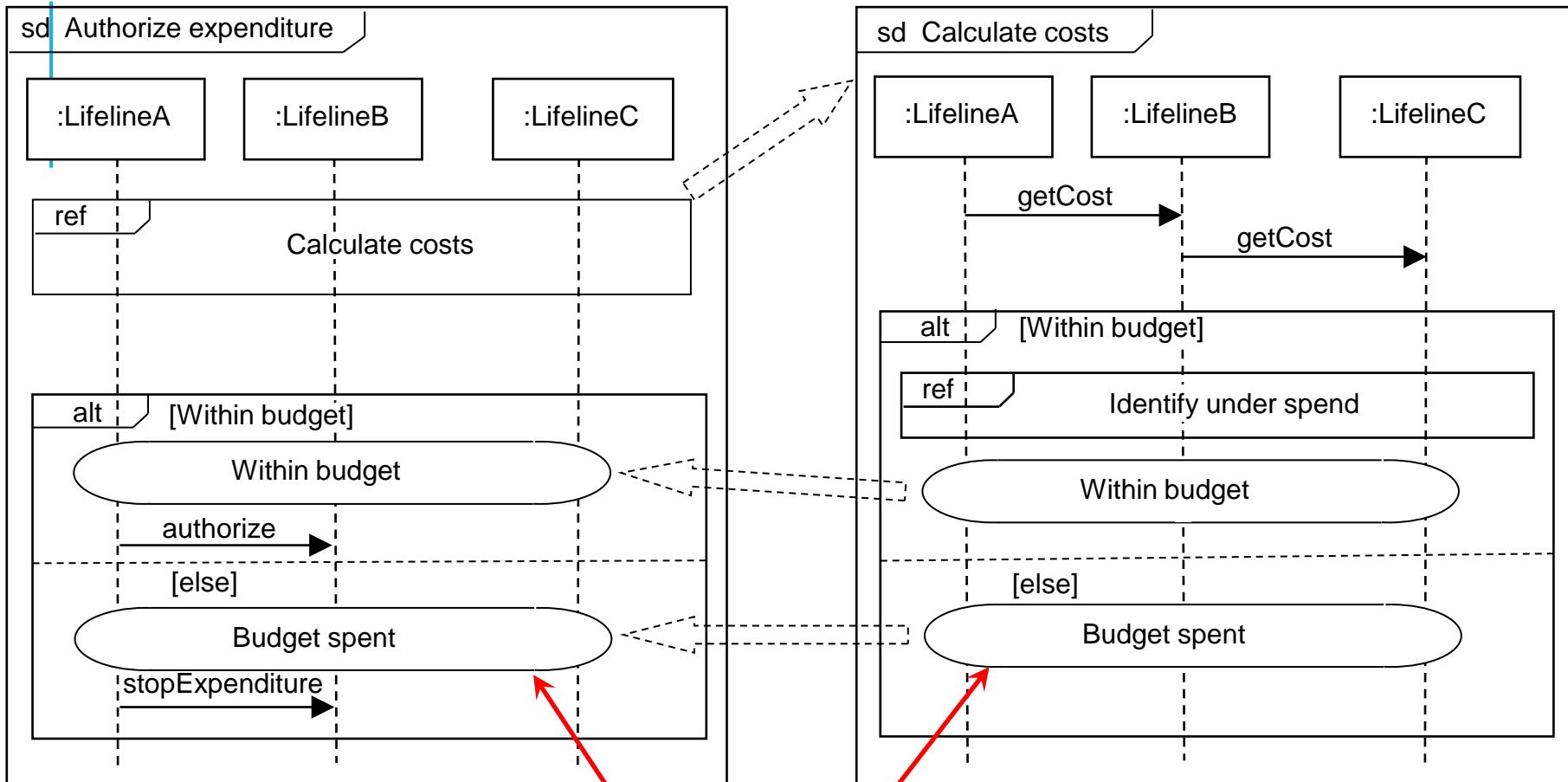
sd Add a new advert to a campaign



Sequence diagram  
referenced in the  
Add a new advert  
to a compaign  
sequence diagram



# USING CONTINUATIONS



Continuations are used to link  
sequence diagrams

## GUIDELINES FOR SEQUENCE DIAGRAMS

- Decide at what level you are modelling the interaction.
- Identify the main elements involved in the interaction.
- Consider the alternative scenarios that may be needed.
- Identify the main elements involved in the interaction.

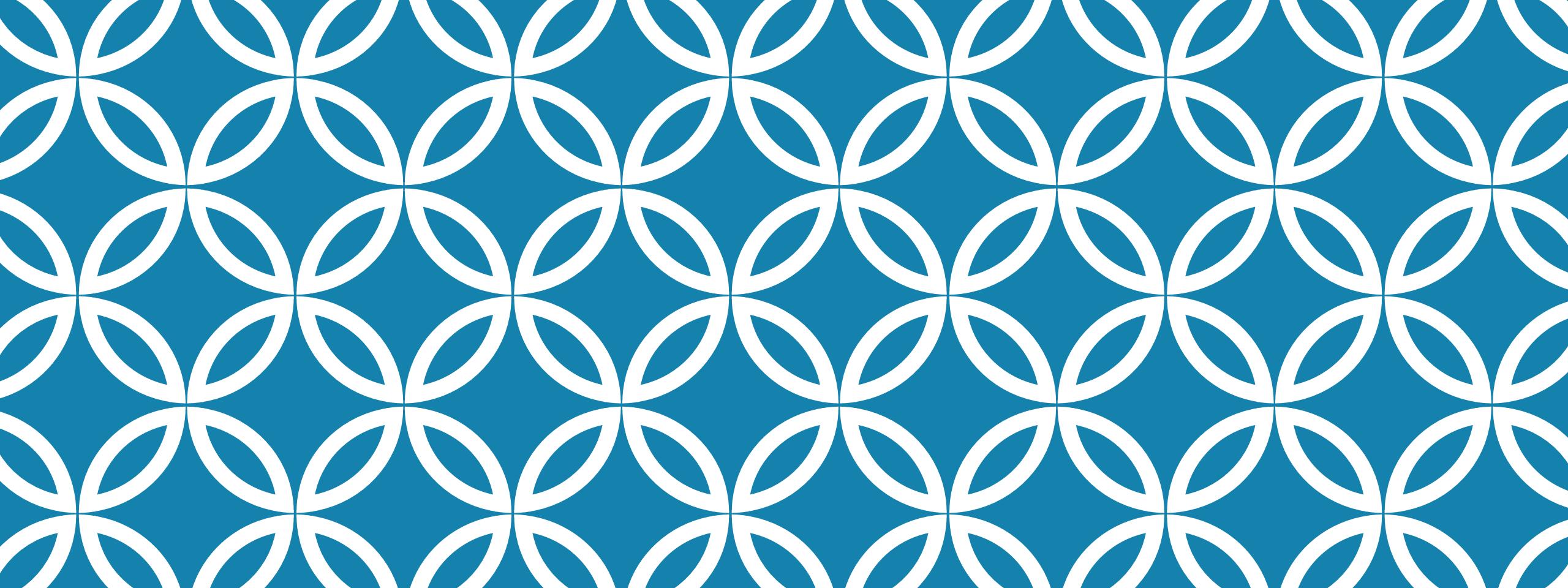
# MODEL CONSISTENCY

The allocation of operations to objects must be consistent with the class diagram and the message signature must match that of the operation.

- Can be enforced through CASE tools.

Every sending object must have the object reference for the destination object.

- Either an association exists between the classes or another object passes the reference to the sender.
- This issue is key in determining association design
- Message pathways should be carefully analysed.



# OPEN CLOSED PRINCIPLE



# OPEN CLOSED PRINCIPLE

As the name suggests, this principle states that software entities should be open for extension but closed for modification.

As a result, when the business requirements change then the entity can be extended, but not modified

# SCENARIO 1: PAYMENT PROCESSING SYSTEM

Suppose we are designing a payment processing system that supports various payment methods, such as credit cards, PayPal, and mobile wallets. We want to ensure that the system can easily accommodate new payment methods in the future without modifying existing code.

## ***Without OCP:***

In a non-compliant design, we might have a single **PaymentProcessor** class that handles all payment methods. Each time a new payment method is introduced, we would need to modify the existing **PaymentProcessor** class to add the new behavior. This approach violates OCP, as it requires modifying existing code.

# WITH OCP

To adhere to OCP, we can design the payment processing system with abstraction and polymorphism.

We create an abstract **Payment** interface that defines a method like **processPayment()**. Each payment method (credit card, PayPal, mobile wallet, etc.) implements this interface with its specific implementation of the **processPayment()** method.

Now, the **PaymentProcessor** class can work with the **Payment** interface instead of concrete implementations. When a new payment method needs to be added, we simply create a new class that implements the **Payment** interface, and the **PaymentProcessor** can use it without any modifications.

The existing code remains untouched, and new functionality is added through extension, adhering to the OCP.



# EXAMPLE JAVA CODE

```
public interface PaymentGateway
    void processPayment(String cardNumber, String amount);
}

public class StripePaymentGateway implements PaymentGateway {
    @Override
    public void processPayment(String cardNumber, String amount) {
        // Use Stripe API to process the payment
    }
}

public class PayPalPaymentGateway implements PaymentGateway {
    @Override
    public void processPayment(String cardNumber, String amount) {
        // Use PayPal API to process the payment
    }
}
```

```
public class PaymentService {  
    private PaymentGateway paymentGateway;  
  
    public PaymentService(PaymentGateway paymentGateway) {  
        this.paymentGateway = paymentGateway;  
    }  
  
    public void processPayment(String cardNumber, String amount) {  
        paymentGateway.processPayment(cardNumber, amount);  
    }  
}
```

# SAMPLE CODE USING CLASSES

```
public class Main {  
    public static void main(String[] args) {  
        PaymentService paymentService = new PaymentService(new StripePaymentGateway());  
        paymentService.processPayment("1234-5678-9012-3456", "100.00");  
  
        paymentService = new PaymentService(new PayPalPaymentGateway());  
        paymentService.processPayment("9876-5432-1098-7654", "200.00");  
    }  
}
```

# SCENARIO 2: NOTIFICATION SYSTEM

Suppose we are building a notification system that supports multiple channels, such as email, SMS, and push notifications. We want to ensure that new notification channels can be added easily without affecting existing channels.

*Without OCP:*

In a non-compliant design, we might have a single **NotificationService** class that handles all notification channels.

Each time a new channel is added, we would need to modify the **NotificationService** class to include the new channel logic.

This approach violates OCP, as it requires modifying existing code.

## *With OCP:*

To follow OCP, we can design the notification system using abstraction and interfaces.

We create a **NotificationChannel** interface with methods like **sendNotification()**.

Each notification channel (email, SMS, push notification, etc.) implements this interface with its specific **sendNotification()** implementation.

Now, the **NotificationService** can work with the **NotificationChannel** interface instead of concrete implementations.

When a new notification channel needs to be added, we create a new class that implements the **NotificationChannel** interface, and the **NotificationService** can use it without modifying existing code.

The system remains open for extension and closed for modification, adhering to the OCP.

# BENEFIT OF OCP COMPLIANT CODE

In this example, the **PaymentService** class is open for extension, but closed for modification.

We can add new payment gateways to the system by simply creating new classes that implement the **PaymentGateway** interface.

We don't need to modify the **PaymentService** class itself. This makes the code more extensible and maintainable.

The **PaymentService** class uses polymorphism to call the `processPayment()` method on the appropriate payment gateway.

This is one way to implement the Open-Closed Principle.

We can add new payment gateways to the system without modifying the `PaymentService` class.

# COMMON VIOLATIONS OF OCP:

***Modifying Existing Code***: One of the most common violations is directly modifying existing code to accommodate new functionality. This defeats the purpose of OCP, as it introduces risks of breaking existing functionality and increases the maintenance effort.

***God Classes or Modules***: Creating large classes or modules that handle multiple responsibilities can lead to a violation of OCP. These "God" classes tend to grow as new functionality is added, and they become difficult to maintain and extend.

***Conditional Statements***: Using extensive conditional statements or switch-case blocks to handle different scenarios can be a sign of a violation of OCP. Adding new cases to the conditional logic requires modifying the existing code.

***Monolithic Interfaces***: Designing interfaces that contain numerous methods for different functionalities can be a violation of OCP. Clients might be forced to depend on methods they do not use, making the interface less cohesive and harder to extend.

***Tight Coupling***: Tight coupling between classes can lead to OCP violations, as changes in one class may necessitate changes in dependent classes. Proper dependency management and inversion of control are essential to prevent this.

# ADDITIONAL NOTES

Please refer to additional material available on canvas.



# MTU

Ollscoil Teicneolaíochta na Mumhan  
Munster Technological University

# PRIMITIVE OBSESSION

# WHAT IS PRIMITIVE OBSESSION?

**Primitive Obsession** refers to an anti-pattern where developers rely heavily on primitive data types (such as integers, strings, or Booleans) instead of encapsulating related data into well-defined domain objects.

# PROBLEMS RESULTING FROM PRIMITIVE OBSESSION

When **primitives** are used extensively throughout a codebase, the resulting code can become cluttered and harder to read.

For example, imagine a system that handles customer data using only string variables for names, addresses, and phone numbers. This approach can quickly lead to tangled code, where strings are passed around and manipulated without any clear structure. Consequently, understanding and maintaining such code becomes increasingly challenging.

**Primitive Obsession** often leads to a lack of type safety, as there is no inherent structure or validation associated with primitive data types.

This increases the likelihood of runtime errors and bugs caused by incorrect data handling. In the example, if a developer mistakenly assigns a phone number to a variable intended for an address, it will go unnoticed until runtime, potentially causing issues that are hard to track down.

# ENTITY

An object defined primarily by its **identity** is called an **ENTITY**.

**Entities** have special modelling and design considerations.

They have lifecycles that can radically change their form and content, but a thread of continuity must be maintained.

Their identities must be defined so that they can be effectively tracked.

Their class definitions, responsibilities, attributes, and associations should resolve around who they are, rather than particular attributes they carry.

# ENTITY

In Object Oriented Programming, we represent related attributes and methods as a Class.

For example, a Person could be a Class within our application. A person will have a name, email address and password as well as many other attributes.

Within our database this person is represented by an id.

This means that the person could change their name, email and password but it would still be the same person. When an object can change its attributes but remain the same object, we call it an Entity. An Entity is mutable because its attributes can change without changing the identity of the object.

# ENTITY

The **Entity object** will maintain the identity because it has an id.

Imagine that our application allows the person to track their current location. When the person can successfully connect to the internet and authenticate with our application a new **location** object is created.

This **Location** object has attributes for longitude and latitude.

The **Location** is a **value object** because we don't care about the specific instance of the object, we only care that it is a location.

# VALUE OBJECTS

**value objects** differ from entities by lacking the concept of identity.

We do not care who they are but rather what they are.  
They are defined by their attributes and should be immutable.

► When a child is drawing, he cares about the color of the marker he chooses, and he may care about the sharpness of the tip. But if there are two markers of the same color and shape, he probably won't care which one he uses. If a marker is lost and replaced by another of the same color from a new pack, he can resume his work unconcerned about the switch.



# VALUE OBJECTS

When the person changes location, we don't have to update the **Location** object, we can simply create a new **Location** object.

The attributes of a **Location** object never change from the moment it is created until the moment it is destroyed. When an object's attributes cannot be changed, it is known as **immutable**

Another **important distinction** is, Value Objects equality **is not based** upon **identity**. For example, when you create two Location objects with the same longitude and latitude attributes those two objects will be equal.

The **Person** object on the other hand does base equality on identity because it is a single representation with an id. If you had two people with the exact same name, they would not be the same person.

# HOW TO IDENTIFY VALUE OBJECTS?

An Entity's attributes can change, but it remains the same representation within our system because of its unique identifier.

Whereas a **value object** is a single instance of an object that is created and then destroyed. We don't care about a specific instance of a Value Object, and we can't change its attributes.

So how do you know when to use an **Entity** and when to use a **Value Object**?

The decision really comes down to the context of the application.

# VALUE OBJECT OR NOT?

Imagine our application is not just a generic social application, it is actually Foursquare now every Location object does have a unique identifier because many different users can check in to that location over time.

Location is an Entity, not a Value Object.

So, whether an object is an Entity or a Value Object really depends on the context of how you are using it within your application.

Generally speaking, objects like location, dates, numbers or money will nearly always be Value Objects, and objects like people, products, files or sales will nearly always be entities

# VALUE CHARACTERISTICS

When you are deciding whether a concept is a Value, you should determine whether it possesses most of these characteristics

- It measures , quantifies, or describes a thing in the domain.
- It can be maintained as immutable.
- It models a conceptual whole by composing related attributes as an integral unit.
- It is completely replaceable when the measurement or description changes.
- It can be compared with others using Value equality.

# CONCEPTUAL WHOLE.

A **Value Object** may possess just one, a few or several individual attributes , each of which is related to the others. Each attribute contributes an important part to a whole that collectively the attributes describe.

Taken apart from the others , each of the attributes fails to provide a cohesive meaning.

Example the value [50,000 euros] has two attributes, the attribute **50,000** and the attribute **euros**.

Together these attributes are a conceptual whole that describes a monetary measure.

# INCORRECTLY MODELLED THING OF WORTH

```
public class ThingOfWorth{  
    private string name;  
    private BigDecimal amount;  
    private string currency;  
}
```

Clients must know how to use amount and currency together because they do not form a conceptual whole.

# BETTER MODEL

```
public final class MonetaryValue {  
    private BigDecimal amount;  
    private String currency;  
    public Monetaryvalue(BigDecimal an amount, String  
aCurrency)  
    {        this.setAmount(anAmount);  
        this.setCurrency(aCurrency)  
    }  
    Public class ThingOfWorth{  
        private ThingName name  
        private MonetaryValue worth  
    }
```

# MAIN CHARACTERISTICS OF VALUE OBJECTS

Value objects have three main characteristics:

## 1. Value Equality

Value objects are defined by their attributes.

They are equal if their attributes are equal.

A value object differs from an entity in that it does not have a concept of identity.

For example, if we consider duration as a value object then a duration of 60 seconds is the same as a duration of 1 minute.

In java , we need to redefine the equals and hashCode methods. By default, Java tests equality against objects' references

# VALUE EQUALITY

Imagine you and your friend pick one card each from two different poker decks and you want to understand if you have picked the same card. How do you do that?

What you would probably do is to check whether the two cards show the same number and are of the same suit. In other words, you check whether they have **the same properties**.

Imagine now that you exchange the two cards. You take the one from your friend, your friend yours. Has anything changed? No. You still have the same card as your friend. Having the same properties, the two cards are indistinguishable one from the other.

Turns out the two cards are in fact Value Objects.

Two Value Objects with same internal values are considered equal. This means you can test for equality by checking if their internal values are all respectively equal

# EXAMPLE

```
1  final class Card {  
2      private Rank rank;  
3      private Suit suit;  
4  
5      public Card(Rank rank, Suit suit) {  
6          this.rank = rank;  
7          this.suit = suit;  
8      }  
9  
10     public boolean sameAs(Card anotherCard) {  
11         return rank.sameAs(anotherCard.rank) &&  
12             suit.sameAs(anotherCard.suit);  
13     }  
14 }
```

## 2. IMMUTABILITY

The only way to change its value is by full replacement. What this means, in code, is to create a new instance with the new value.

We will see later how [immutability](#) allows us to prevent errors from occurring and also how it can help optimise performance.

When implementing a value object, we need to make sure that we remove all setters and that getters return immutable objects or copies to guarantee that nobody can change those values from the outside. In Java, we can use the `final` keyword.

# IMMUTABILITY

A Value Object is immutable.

It means you can't change its internal value(s) after creating it. **No setters allowed.**[private if provided]

After injecting one or more parameters into the constructor, there's no way back. That object will remain the same no matter what until it gets disposed by the garbage collector.

Immutability brings two great advantages:

# ADVANTAGES OF IMMUTABILITY

- 1.Hassel Free Sharing
- 2.Improved Semantics
- 3.Value equality

# HASSLE FREE SHARING

You can share any Value Object by reference because, being immutable, it won't be modified in another part of the code.

This dramatically lowers accidental complexity and cognitive load required to avoid introducing any bug.

This is **gold** especially when your code will run in a multi-threaded environment.

# EXAMPLE

```
final class Name {  
    private String value;  
    public Name(String value) {  
        this.value = value;  
    }  
}
```

No chances to modify this object after construction

# 3 .SELF VALIDATION

A **Value Object** only accepts values which make sense in its context. This means you are not allowed to create a Value Object with invalid values.

A Value Object must check for the consistency of its values when they are injected into the constructor. If one of the values is invalid, a meaningful exception must be thrown.

This means no more *ifs* around the instantiation of an object. Every **formal validation** happens at construction time.

For instance, if we have a concept of Age, it wouldn't make sense to create an instance of age with a negative value.

This enforced validation is also useful to express domain constraints in a meaningful and explicit way.

# EXAMPLE

```
1  final class Rank {  
2      private int value;  
3  
4      public Rank(int value) {  
5          if (value < 1 || value > 13) {  
6              throw new InvalidRankValue(value);  
7          }  
8  
9          this.value = value;  
10     }  
11 }
```

# EXAMPLE OF A VALUE OBJECT

```
public final class EmailAddress {  
    private static final EmailValidator validator = EmailValidator.getInstance();  
  
    private final String value;  
  
    public EmailAddress(String value) {  
        if (!validator.isValid(value)) {  
            throw new InvalidEmailException();  
        }  
        this.value = value;  
    }  
    public EmailAddress change(String value) {  
        return new EmailAddress(value);  
    }  
    @Override  
    public String toString() {  
        return value;  
    }  
    @Override  
    public boolean equals(Object o) {  
        if (o == null || getClass() != o.getClass()) return false;  
        EmailAddress that = (EmailAddress) o;  
        return Objects.equals(value, that.value);  
    }  
    @Override  
    public int hashCode() {  
        return Objects.hash(value);  
    }  
}
```

# BENEFITS: REDUCE PRIMITIVE OBSESSION

Primitive Obsession is a code smell that is easy to create.

It's when we use primitives for storing data or as parameters.

For instance, we use a string for an email address, a number for a weight or a number for a duration.

It usually happens because it seems like too much effort to create a new class just for storing that one attribute.

Before you know it, everything you use is now a primitive!

# VALIDATION

The only way for a value object to exist is to be **valid**.

Once that object exists it can no longer change.

Thanks to these characteristics of self-validation and immutability, we can stop wondering if we need to validate a parameter or not.

# EASIER TO READ

Using value objects improves **readability** of code.

Example `List<PhoneNumber>` instead of `List<String>`.

The code is a lot **more expressive**.

**Value objects** centralize related domain logic making it easier to find and change.

# WHEN TO USE

Use value objects if there is any validation logic associated with a field or group of fields.

# IMPLEMENTING VALUE OBJECTS

*Create an **Immutable** class.*

*Make it **final**.*

*Make all **fields final**.*

*Create a new object when someone wants a value object with different data.*

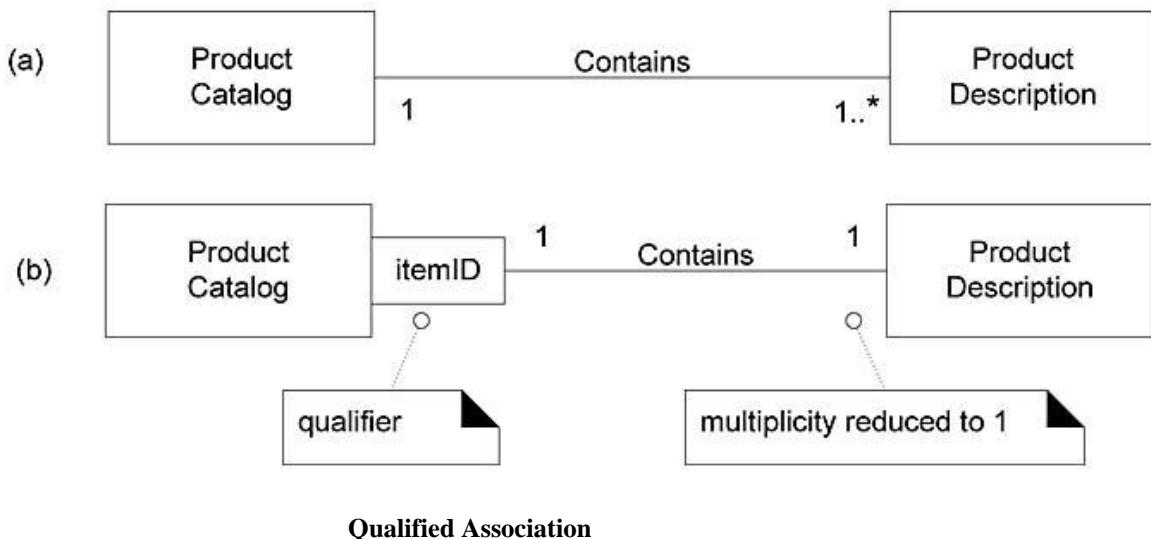
```
5  public DateRange(Date fromInclusive, Date toExclusive) {
6      this.fromInclusive = (Date) fromInclusive.clone();
7      this.toExclusive = (Date) toExclusive.clone();
8  }
9
10 public DateRange extend(long millis) {
11     Date extendedTo = new Date(toExclusive.getTime() + millis);
12     return new DateRange(fromInclusive, extendedTo);
13 }
14
15 public Date getFromInclusive() {
16     return (Date) fromInclusive.clone();
17 }
18
19 public Date getToExclusive() {
20     return (Date) toExclusive.clone();
21 }
22
23 @Override
24 public boolean equals(Object o) {
25     if (this == o) return true;
26     if (o == null || getClass() != o.getClass()) return false;
27
28     DateRange dateRange = (DateRange) o;
29
30     return fromInclusive.equals(dateRange.fromInclusive)
31         && toExclusive.equals(dateRange.toExclusive);
32 }
33
34 @Override
35 public int hashCode() {
36     int result = fromInclusive.hashCode();
37     result = 31 * result + toExclusive.hashCode();
38     return result;
39 }
40 }
```

# Qualified Associations

## 1 Example 1

A **qualified association** is the UML equivalent of a programming concept variously known as associative arrays, maps, and dictionaries.

A **qualifier** may be used in an association; it distinguishes the set of objects at the far end of the association based on the qualifier value. An association with a qualifier is a **qualified association**.



For example, *ProductDescriptions* may be distinguished in a *ProductCatalog* by their *itemID*, as illustrated in Figure b . As contrasted in Figure a , qualification reduces the multiplicity at the far end from the qualifier, usually down from many to one. Depicting a qualifier in a domain model communicates how, in the domain, things of one class are distinguished in relation to another class.

Figure 1 shows a way of representing the association between the Order and Order Line classes that uses a qualifier. The qualifier says that in connection with an Order, there may be one Order Line for each instance of Product.

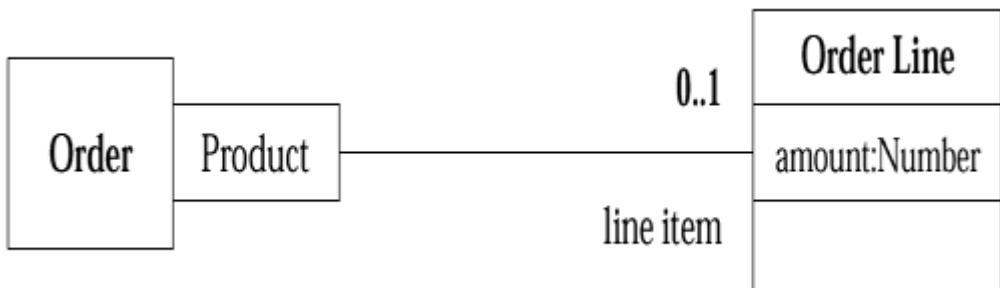


Figure 1

Conceptually, this example indicates that you cannot have two Order Lines within an Order for the same Product. From a specification perspective, this qualified association would imply an interface along the lines of

```
class Order {  
  
    public OrderLine getLineItem  
        (Product aProduct);  
    public void addLineItem  
        (Number amount, Product forProduct);
```

Thus, all access to a given Line Item requires a Product as an argument. A multiplicity of 1 would indicate that there must be a Line Item for every Product; \* would indicate that you can have multiple Order Lines per Product but that access to the Line Items is still indexed by Product.

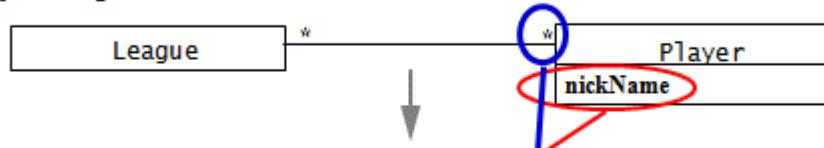
From an implementation perspective, this suggests the use of an associative array or similar data structure to hold the order lines.

```
Class Order {  
  
    private Map _lineItems;
```

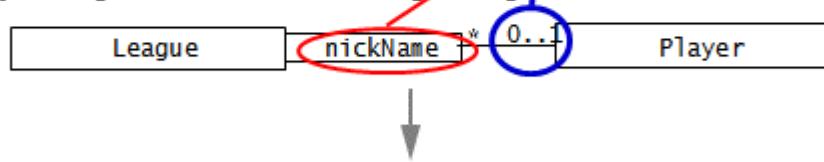
## 2 Example 2

### *Bidirectional qualified association*

#### **Object design model before transformation**



#### **Object design model before forward engineering**



#### **Source code after forward engineering**

*Bidirectional qualified association (continued)*

Source code after forward engineering

```
public class League {           public class Player {  
    private Map players;         private Map leagues;  
  
    public void addPlayer        public void addLeague  
        (String nickName, Player p)   (String nickName, League l)  
    {                           {  
        if                         if (!leagues.containsKey(l))  
            (!players.containsKey(nickName) {  
                players.put(nickName,     leagues.put(l,  
                    p);                  nickName);  
                p.addLeague(nickName,      l.addPlayer(nickName,  
                    this);               this);  
            }                          }  
        }                          }  
    }  
}
```

David J. Barnes & Allen R. Dickey

Object-Oriented Software Engineering: Using UML, Patterns, and Java

24

Sample Implementaton of using a map in Java.

```
import java.util.HashMap;

import java.util.Map;

public class HashMapExample {

    public static void main(String[] args) {

        Map<String, Integer> vehicles = new HashMap<>();

        // Add some vehicles.

        vehicles.put("BMW", 5);

        vehicles.put("Mercedes", 3);

        vehicles.put("Audi", 4);

        vehicles.put("Ford", 10);

        System.out.println("Total vehicles: " + vehicles.size());

        // Iterate over all vehicles, using the keySet method.

        for (String key : vehicles.keySet())

            System.out.println(key + " - " + vehicles.get(key));

        System.out.println();

        String searchKey = "Audi";

        if (vehicles.containsKey(searchKey))

            System.out.println("Found total " + vehicles.get(searchKey) +
" " + searchKey + " cars!\n");

        // Clear all values.

        vehicles.clear();

        // Equals to zero.

        System.out.println("After clear operation, size: " + vehicles.size());

    }
}
```

# SOLID PRINCIPLES



# SINGLE RESPONSIBILITY PRINCIPLE

In object-oriented programming, the single responsibility principle states that every class should have a **single responsibility**, and that responsibility should be entirely **encapsulated** by the class. All its services should be narrowly aligned with that responsibility....

A class or module should have one, and only one, **reason to change**.



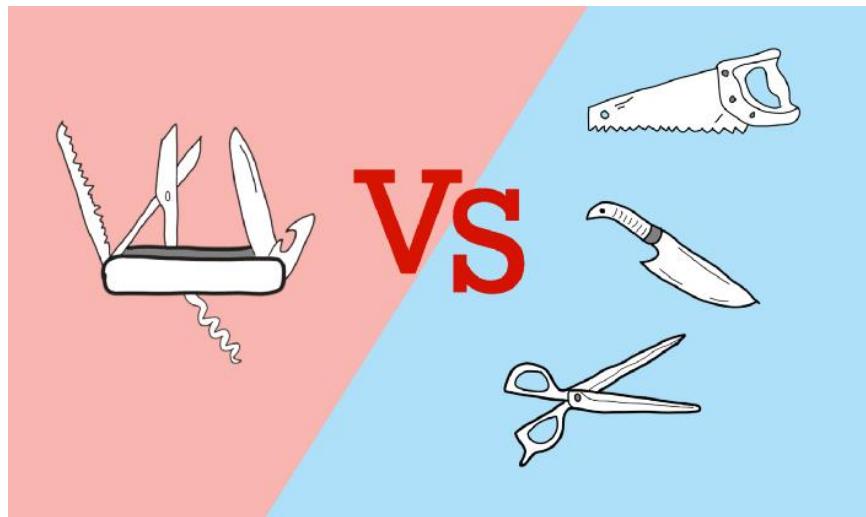
# GOD CLASS

- A **GOD** is a class that keeps track of a lot of information and have several responsibilities.
- A **god object** is an object that *knows too much* or *does too much*. The god object is an example of an anti-pattern.
- Most of such a program's overall functionality is coded into a single "all-knowing" object, which maintains most of the information about the entire program and provides most of the methods for manipulating this data

# SWISS KNIFE?

In the picture, what would you rather own, the Swiss army knife or all the tools on the right? When traveling, portability is the most important thing. The Swiss army knife is the appropriate tool.

But for day-to-day life — would you cut bread, mend the car, or open cans with a Swiss army knife? Probably not. You can do it, but it's much easier to use a bread knife, a set of screwdrivers, or a can-opener.



# SWISS KNIFE

Real-life code should be treated the same.

A do-it-all class (god object) it's ok, it will get the job done when you are on the go (like prototyping or scripting) but not for everyday use.

# SINGLE RESPONSIBILITY PRINCIPLE



# SINGLE RESPONSIBILITY PRINCIPLE

You work as a team leader for one of the software firms in Cork . In your spare time you do some writing, newspaper editing and other various projects. Basically, you have multiple responsibilities in your life.

When something bad happens at your work place, like when your boss scolds you for some mistake, you get distracted from your other work. Basically, if one thing goes bad, everything will mess up.

So if a class (or module) needs to be modified for more than one reason, it does more than one thing i.e. has more than one responsibility.

# WHY SRP

## Helps Organize the code.

Let's imagine a car mechanic who owns a repair shop. He has many tools to work with. The tools are divided into types; Pliers, Screw-Drivers (Phillips / Blade), Hammers, Wrenches (Tubing / Hex) and many more .

How would it be easier to organize the tools?

Few drawers with different types in each one of them?

or

Many small drawers, each containing a specific type?

Now, imagine the drawer as the *module*

This is why many small modules (classes) are more organized than few large ones.

# WHY SRP

## Less fragile

When a class has more than one reason to be changed, it is more fragile.

A change in one location might lead to some unexpected behaviour in totally other places.

## Low Coupling

More responsibilities lead to higher coupling.  
The couplings are the responsibilities.

Higher coupling leads to more dependencies,  
which is harder to maintain.

## Code Changes

Refactoring is much easier for a single responsibility module.

## Maintainability

It's obvious that it is much easier to maintain a small single purpose class, then a big monolithic one.

# WHY SRP

## Testability

A test class for a ‘one purpose class’ will have less test cases (branches). If a class has one purpose it will usually have less dependencies, thus less mocking and test preparing.

## Easier Debugging

In a single responsibility class, finding the bug or the cause of the problem, becomes a much easier task.

## WHAT NEEDS TO HAVE SINGLE RESPONSIBILITY?

Each part of the system.

The methods

The classes

The packages

# AN EXAMPLE TO CLARIFY THIS PRINCIPLE:

Suppose you are asked to implement a UserSetting service where the user can change the settings but before that the user has to be authenticated. One way to implement this would be:

```
public class UserSettingService  
{  
    public void changeEmail(User user)  
    {  
        if(checkAccess(user))  
        { //Grant option to change } }  
  
    public boolean checkAccess(User user)  
    { //Verify if the user is valid. }  
}
```

# PROBLEMS

All looks good, until you would want to reuse the checkAccess code at some other place OR you want to make changes to the way checkAccess is being done OR you want to make change to the way email changes are being approved.

In all the later 2 cases you would end up changing the same class and in the first case you would have to use UserSettingService to check for access as well, which is unnecessary.

One way to correct this is to decompose the UserSettingService into UserSettingService and SecurityService.

And move the checkAccess code into SecurityService.

# REVISED CODE

```
public class UserSettingService
{
    public void changeEmail(User user)
    {
        if(SecurityService.checkAccess(user))
            { //Grant option to change } } }

public class SecurityService
{
    public static boolean checkAccess(User user) {
        //check the access. }

}
```

# BAD DESIGN EXAMPLE

```
public class Task {  
    public void downloadFile(String location) {  
        //...  
    }  
  
    public void parseTheFile(File file) {  
        //...  
    }  
  
    public void persistTheData(Data data) {  
        //...  
    }  
}
```

# DOES CLASS ADHERE TO SRP?

Not at all, because it does a lot of different things:

`downloadFile()` downloads the file, by communicating over the internet

`parseTheFile()` parses the file contents

`persistTheData()` saves the data into a database

A better solution would be to have separate classes for each of the responsibilities currently taken up by Task. Here is one such solution.

# IDENTIFYING THE LOW- LEVEL CLASSES

```
public class FileDownloader {  
  
    public void downloadFile() {  
        • //...}  
  
    }  
  
    public class DataParser {  
  
        public void parseData() {  
  
            //...  
  
        }  
        • }  
  
    public class DatabaseStorer {  
  
        public void storeIntoDatabase() {  
            • //...  
  
        }  
  
    }  
}
```

# WHY IS THIS A BETTER DESIGN

**FileDownloader** only downloads the file from the internet

**DataParser** only parses the downloaded file

**DatabaseStorer** only stores the parsed data, into the database

These are all low-level classes.

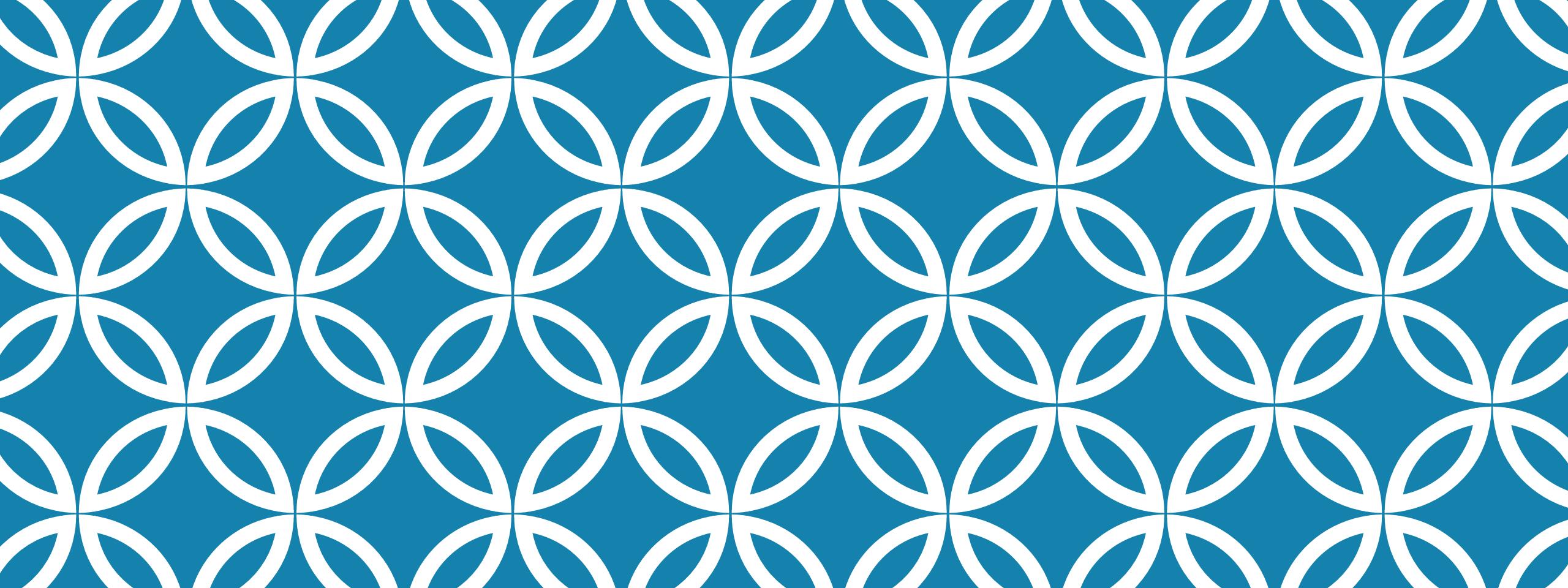
# ADDING A HIGH-LEVEL CLASS

```
public class DownloadAndStore {  
    public void doEverything() {  
        new FileDownloader().downloadFile();  
        new DataParser().parseData();  
  
        new DatabaseStorer().storeIntoDatabase();  
    }  
}
```

Then, you can create a high-level class such as `DownloadAndStore` that invokes the functionality of these low-level classes to perform the complete task. Such an organization ensures that the low-level classes are reusable.

# EXAMPLES

Refer to Examples of SRP applied in sample code in archive file.



# MODELLING ROLES

Analysis Patterns

# OO ANALYSIS

In the object-oriented analysis, we ...

- 1. Elicit requirements:** Define what does the software need to do, and what's the problem the software trying to solve.
- 2. Specify requirements:** Describe the requirements, usually, using use cases (and scenarios) or user stories.
- 3. Create Conceptual model:** Identify the important concepts, attributes and relationships.

We're not going to cover the first two activities, just the **last one**. These are already explained in detail in Requirements Engineering.

# ANALYSIS PATTERNS

Patterns embody the knowledge and experience of many developers.

A common belief is that analysis patterns can improve conceptual modelling quality because they are proven solutions and have been scrutinized by many developers

# ROLES

While performing **object-oriented analysis**, one often encounters problems related to **roles**.

**Roles** are what any concept (or class) would play within the context of its related concepts (or classes).

For instance a “company” would be the “supplier” of some specific “product”. “Supplier” is a **role**.

All **role problems** can be easily solved by selecting one of the following **5 role patterns**:

1. **Role Inheritance**
2. **Association Roles**
3. **Role Classes**
4. **Generalized Role Classes**
5. **Association Role Classes**

Each **role pattern** contains its own blend of power, flexibility and complexity. Together, they offer a complete solution to all role problems.

# ROLE ANALYSIS PATTERNS

Role problems are very frequent, and there exists at least half-a-dozen ways to solve them.

We call them *Role Analysis Patterns*. They are “pre-cooked” solutions for typical Role analysis problems.

Each pattern uses simple UML **static modelling elements**: classes, inheritance, associations and association classes.

# 1 ROLE INHERITANCE

One could easily think of Roles as *types*.

For instance a *company* that *supplies products* would be understood as a *vendor*.

On the other hand, if it *buys products*, then it's known as a *Customer*. That makes two different types of companies: *vendors* and *customers*.

This concept could lead to the following model fragment:

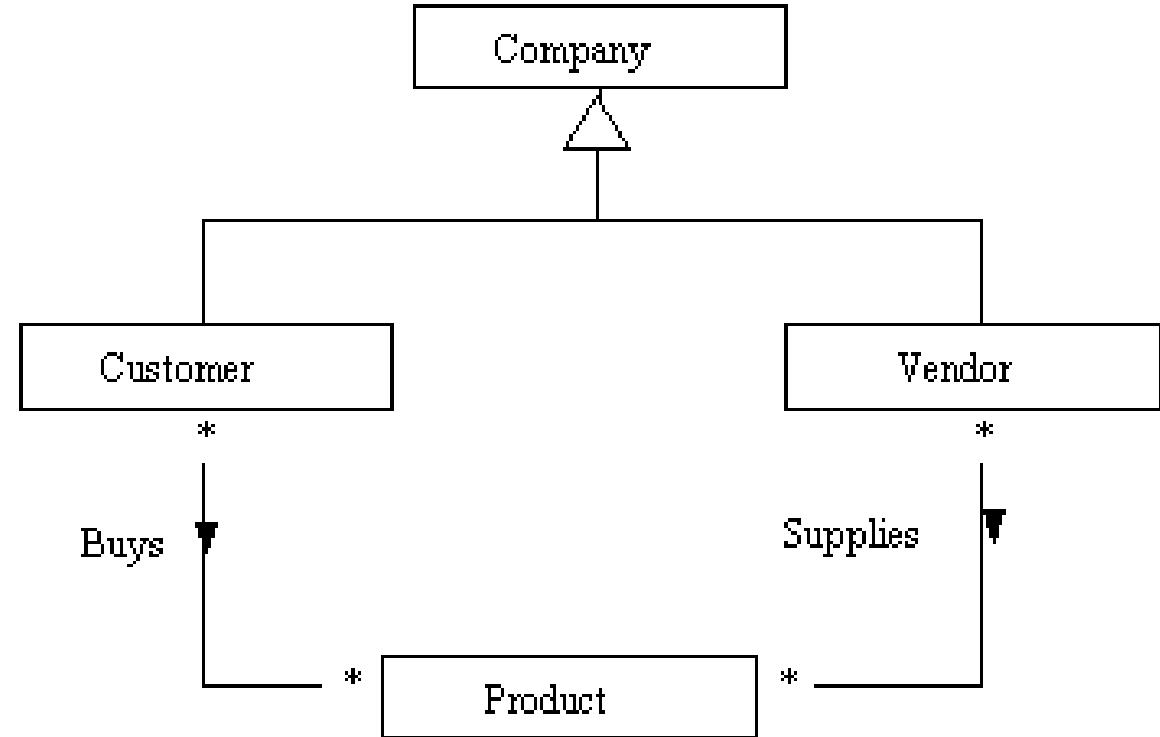


Figure 1

# THE INHERITANCE SOLUTION

In plain English this model tells us that there exists two different types of companies: *customers* and *vendors*. *Customers buy products* whereas *vendors supply them*. Because inheritance is being used, this model also says that a company **cannot be both** a vendor and a customer; it's either one or the other—and once the choice is made, it's final.

Is this good or bad?

It all depends on the problem domain you're trying to solve. Some business rules might dictate such a restriction; for example, one might not be allowed to get supplies from a customer. Other business rules might state the opposite: anyone may buy or supply at any time. We pick the model that best describes the situation.

The model shown above is a concrete example of the inheritance solution. It describes the role restrictions.

# METAMODEL

Now let's express the same solution for this example in an **abstract** way, using a *metamodel*.

Now, the inheritance meta model disallows multiple roles to be played by the same base class. In other words, a company can be either a vendor or a customer, but not both. Similarly, a person can be either a dentist or a landlord, but not both! There are obviously cases where this limitation will be too restrictive to meet your problem domain requirements.

Because metamodels supply a general answer, they are also called *patterns*.

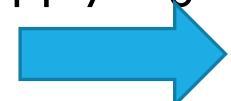
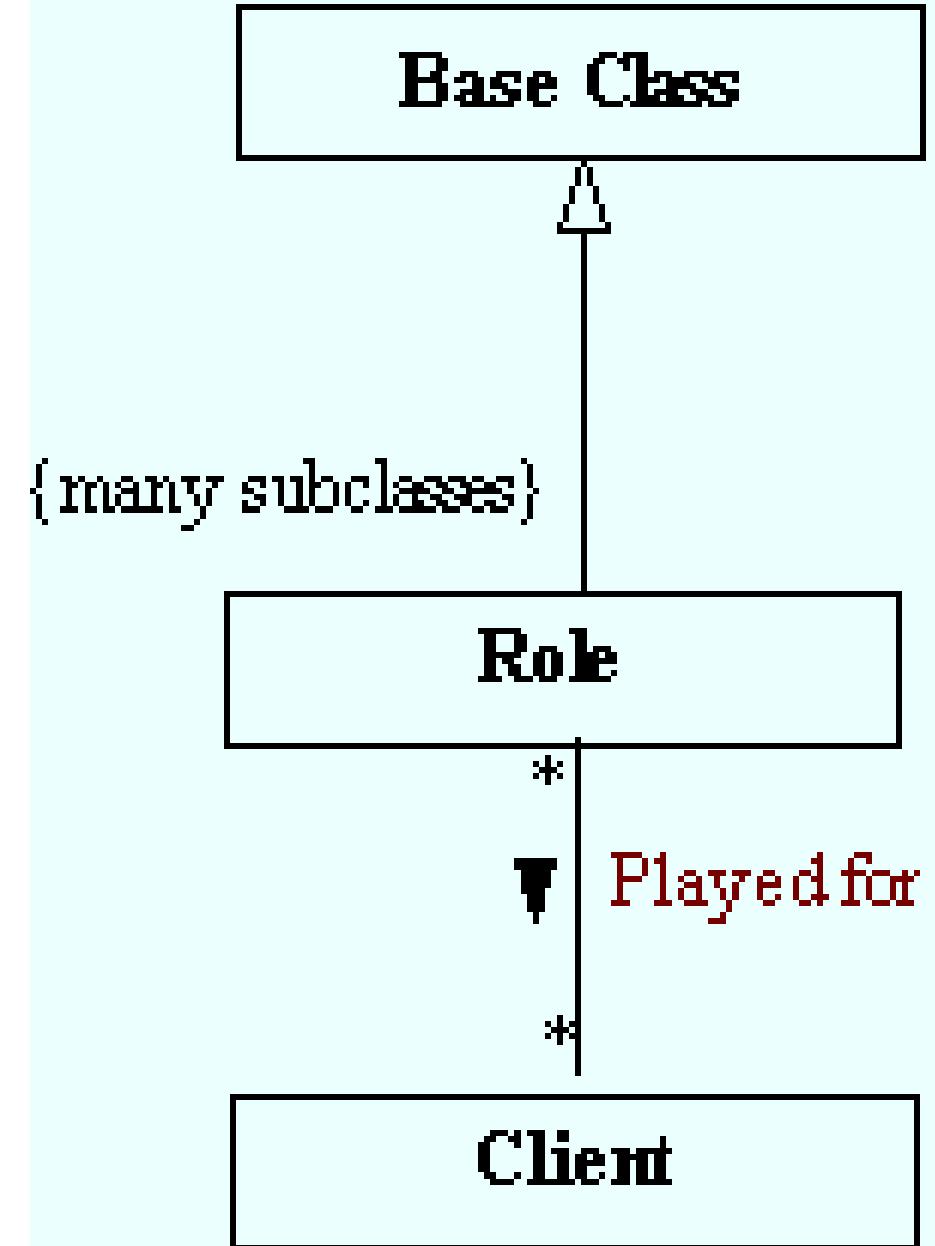


Figure 2



# INHERITANCE SOLUTION RESTRICTIONS

Now, the inheritance metamodel disallows multiple roles to be played by the same **base class**.

In other words, a company can be either a vendor or a customer, but not both.

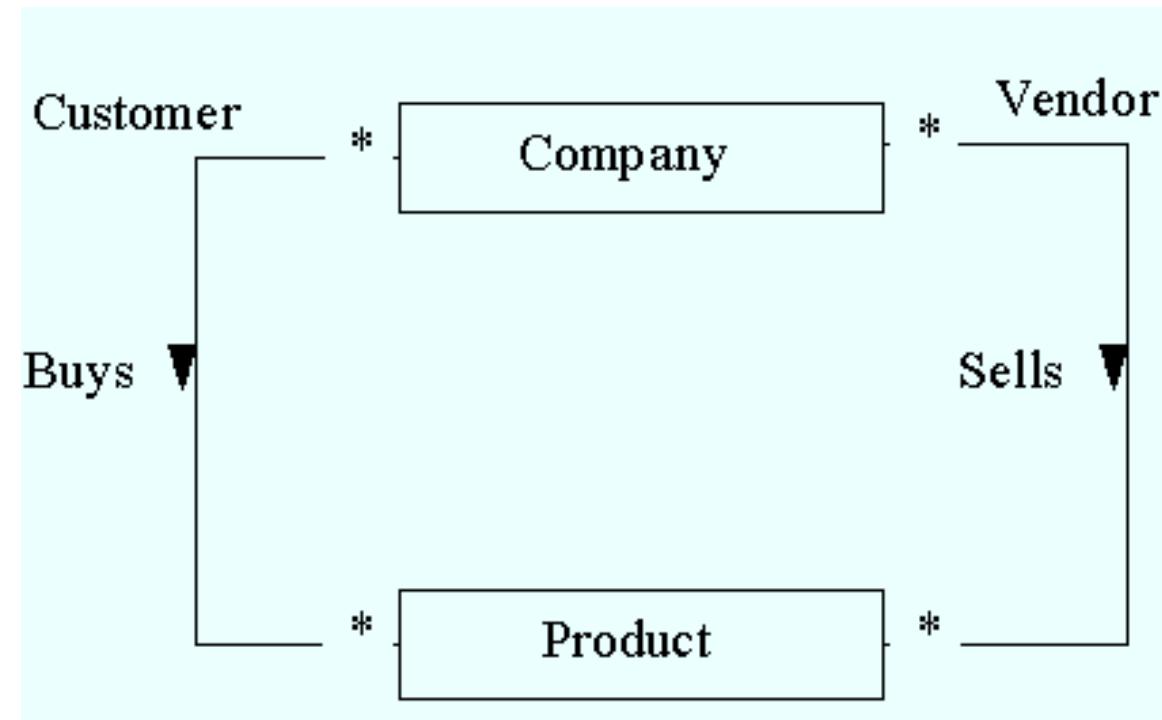
Similarly, a person can be either a dentist or a landlord, but not both! There are obviously cases where this limitation will be too restrictive to meet your problem domain requirements.

## 2. THE ASSOCIATION ROLE SOLUTION

With this solution, a company can either sell or buy any product and therefore be considered as a customer or a vendor, at any time as well as *at the same time*.

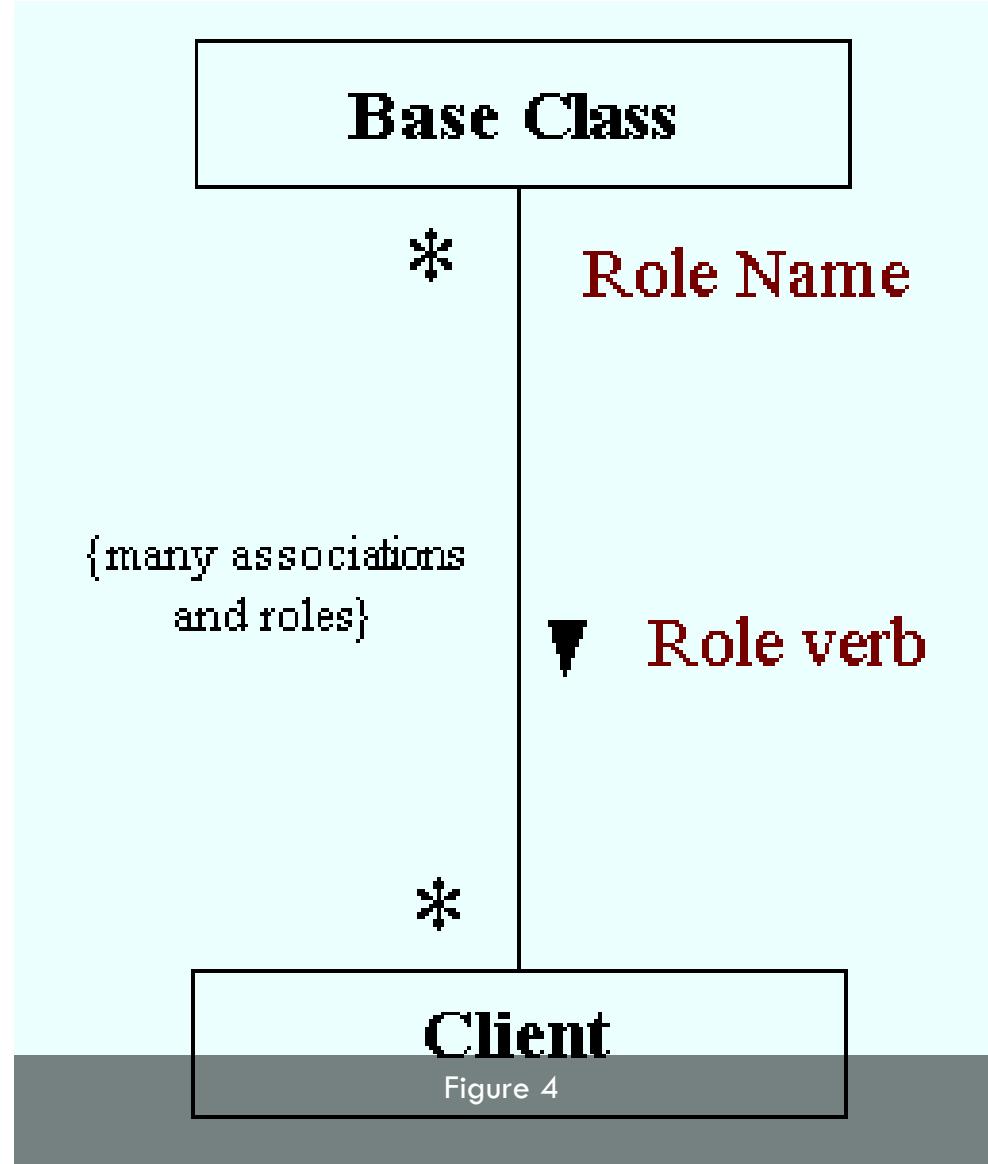
All it takes is a *link* (association instance) to allow a company object to play the role of a customer and/or a vendor for a specific product instance.

Figure 3



## 2 THE ASSOCIATION ROLE PATTERN

The Figure shows *Base Class* as an abstraction of classes like *Company*, and *Client* as an abstraction of *Product*. It allows for as many *Base Class* instances and *Client* instances as needed. The **Role part** of the metamodel is taken care of by the association and its own *Role*. The Association Role metamodel describes our *second role pattern*.



# ASSOCIATION ROLES

This Association Role pattern is clearly a very economical way of dealing with roles, but you'd be amazed to see how many problems can be solved with just that one feature.

Due to their efficiency and usefulness, every modeller should be very familiar with Association Roles.

# ASSOCIATION ROLES EXAMPLE

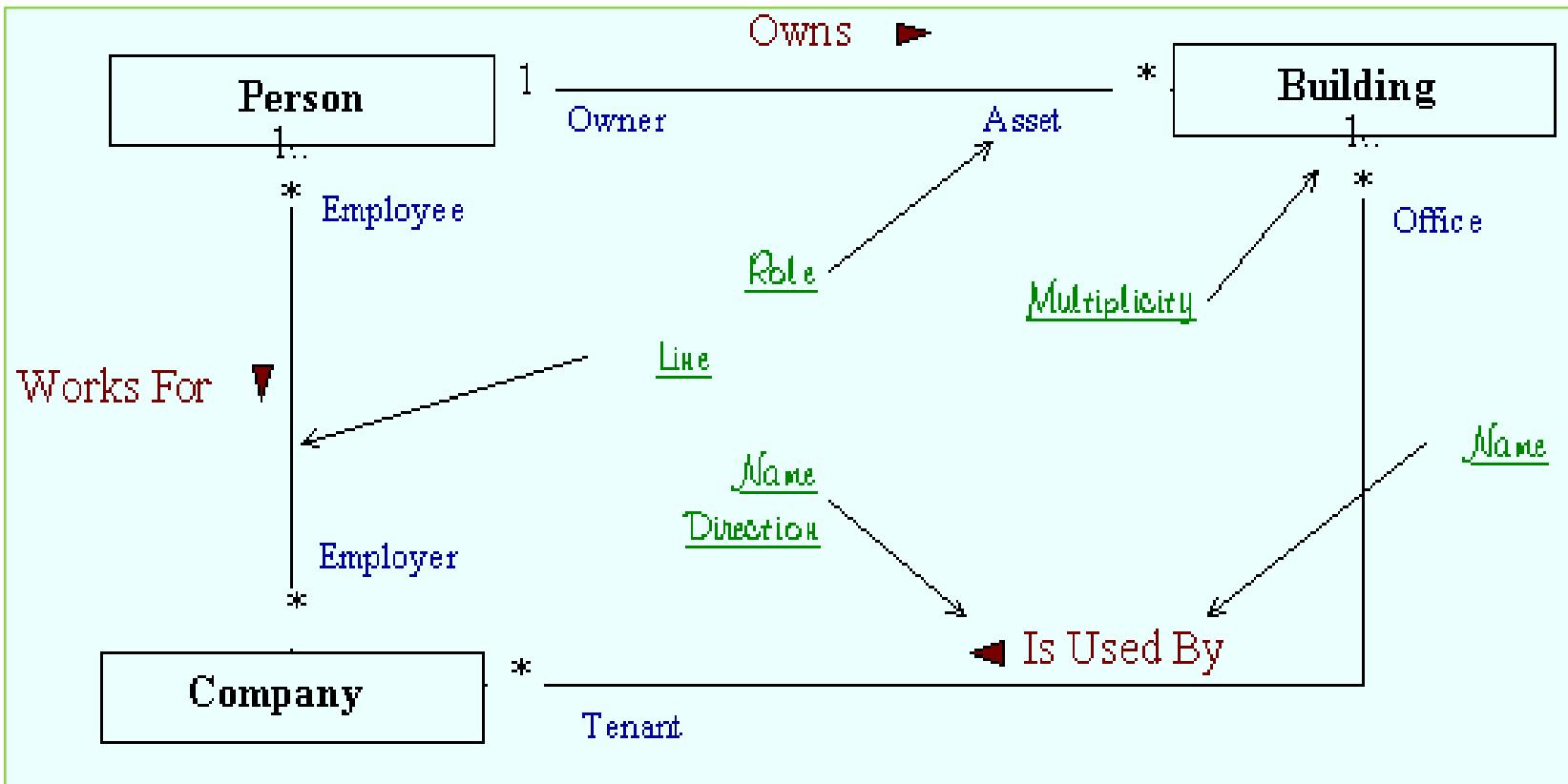


Figure 5

# PROBLEM WITH ASSOCIATION ROLE SOLUTION

An obvious **drawback** of Association Roles is the fact that they **don't allow** any attributes.

Back to our original example, if the vendors or customers would require specific attributes, then we'd have to come up with a class Role solution, since classes can hold attributes. **Role Classes** are a way to accomplish that.

# 3 THE ROLE CLASS

This **model example** allows you to link any company with any of the Roles it may possibly play. Here a role is modeled with a class—as it was in our inheritance solution in

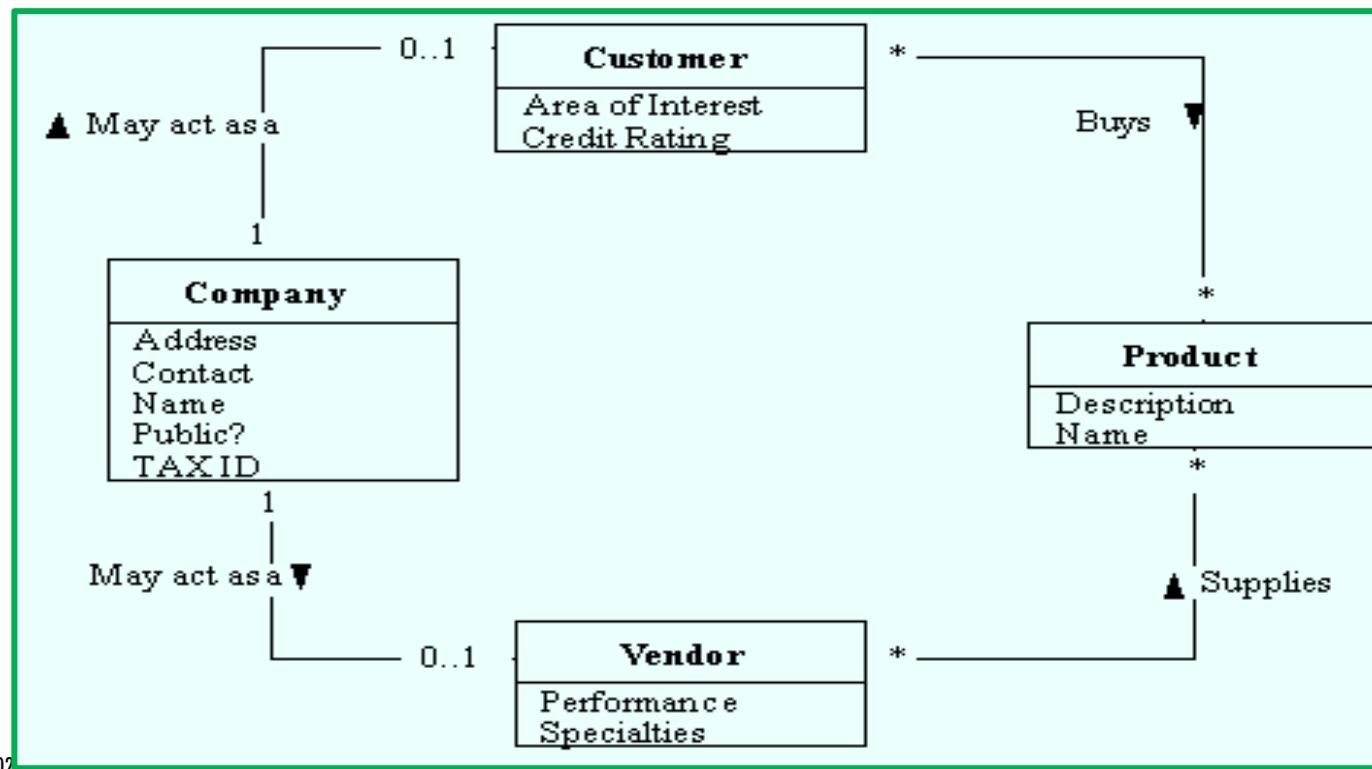


Figure 6

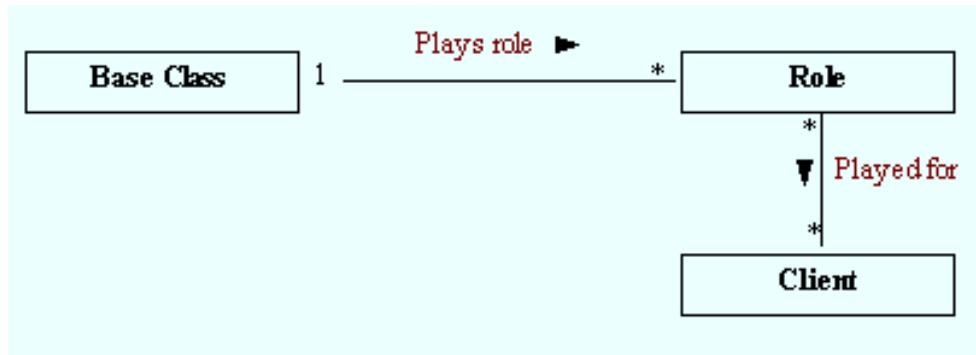
# METAMODEL FOR CLASS ROLE PATTERN

This allows for flexibility of zero or multiple Roles while providing the power of Role-specific attributes and even class operations, if needed. Figure below shows the Role Class Metamodel, our third Role Pattern.

This metamodel shows that the role class is like a middle man between the *Client* and the *Base Class*. Chances are that *Client* will “look” at *Role* as if it were the *Base Class* itself.

# META MODEL FOR CLASS ROLE PATTERN

Figure 7



# ROLE CLASS EXAMPLE

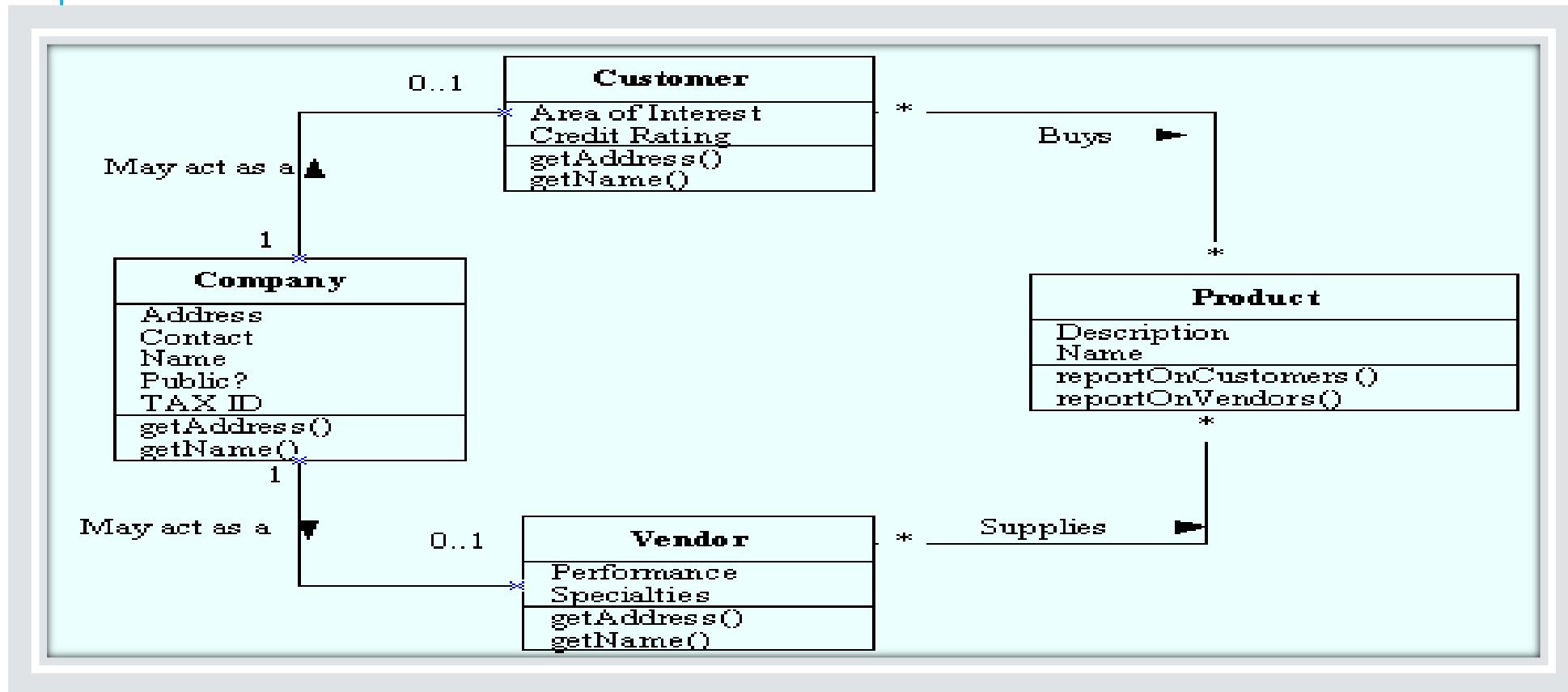


Figure 8

# ROLE CLASS EXAMPLE

In our previous example , a *Product* object might need the address of the *Customer* object that it's linked with.

Then the *Customer* object will need to turn towards the *Company* object and get it's address from there. **Delegation** calls will be quite frequent. The same methods may be found in different classes (like "getAddress" and "getName" in figure 8 on previous slide slide).

In **delegation**, an object handles a request by delegating to a second object (the **delegate**). The **delegate** is a helper object.

# 4 THE ROLE CLASS GENERALIZATION

Figure 8 shows repetition of class operations; for instance, `getAddress()` and `getName()` operations are seen in both *Vendor* and *Customer* Role classes.

Repetition calls for generalization. Figure 9 shows an example—along with an additional class role, *Broker*—that emphasizes the need for generalization.

# 4 THE ROLE CLASS GENERALIZATION

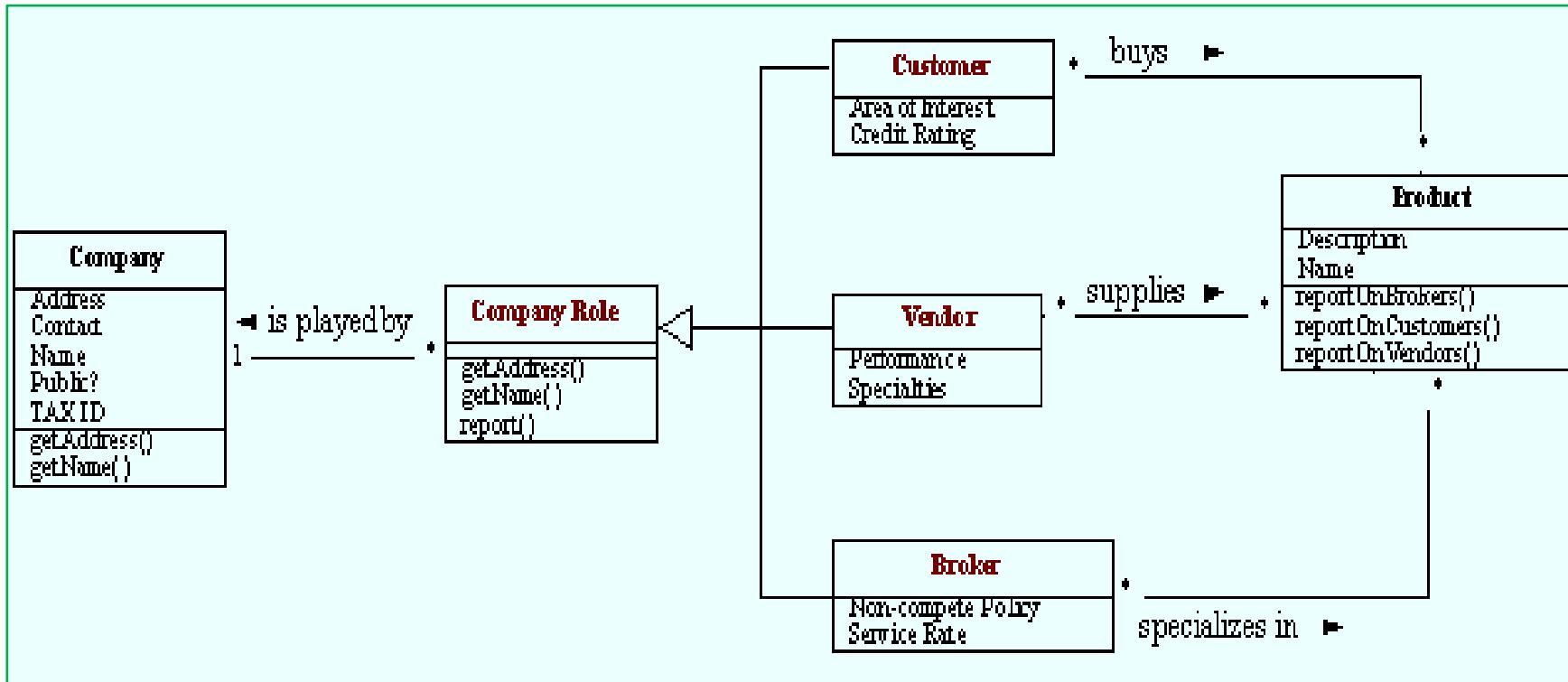


Figure 9

# THE ROLE CLASS GENERALIZATION SOLUTION

With the **Role Class Generalization solution**, all roles may be modeled and dynamically assigned to any Company at any time. Role-specific operations are supported but not repeated, thanks to the UML generalization feature.

These Role-specific operations simply reside in one place: the *Company* Role generalization class. It is the one class that will take care of all delegations to Company.

Only one implementation of its operations (methods) is needed, and it will serve all class Roles by virtue of inheritance. Figure 10 shows the Metamodel for this solution.

# THE ROLE CLASS GENERALIZATION META MODEL

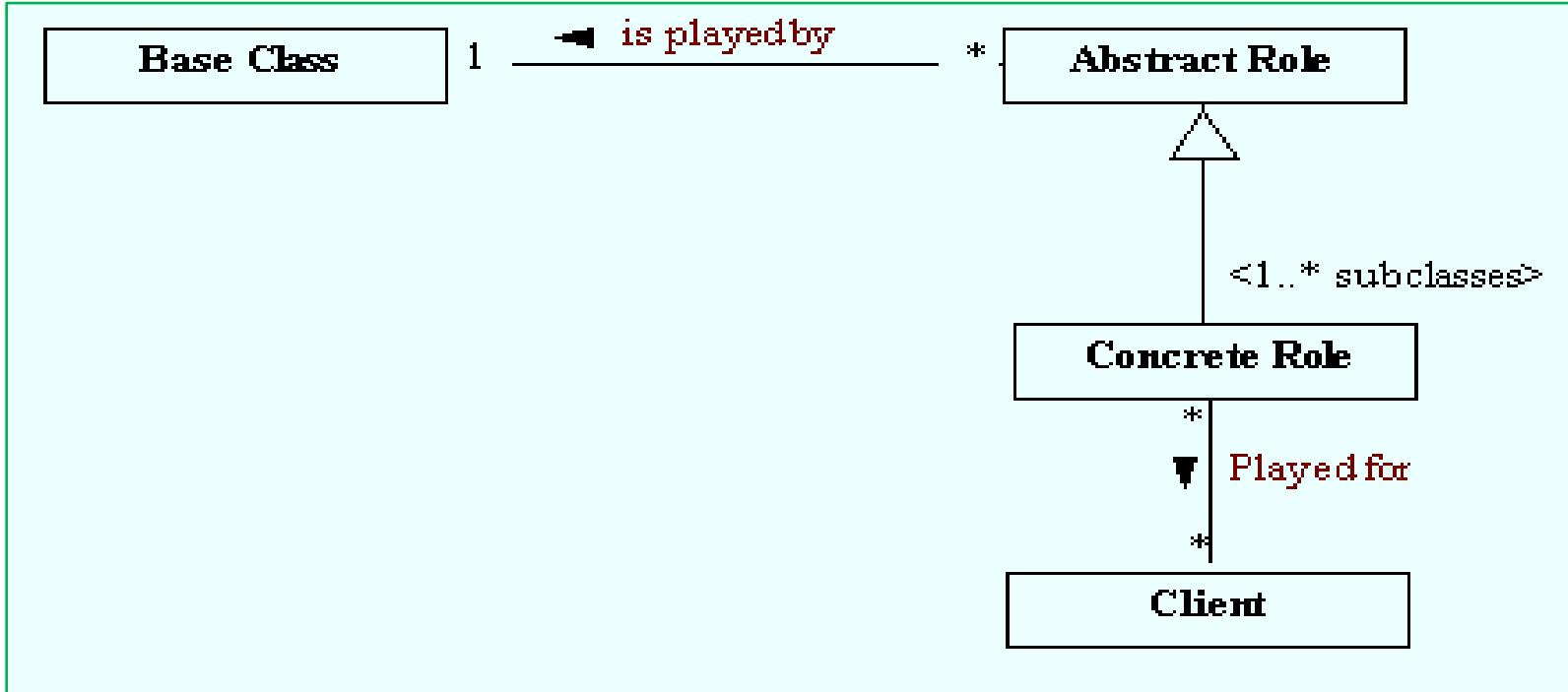


Figure 10

# 5 THE ROLE ASSOCIATION CLASS

The examples we've seen so far assume that all roles are known at the time of the analysis.

When the system is implemented, only *these* roles can be used.

For instance, a company can only assume the role of a vendor, a customer or a broker. These different roles can certainly be dynamically assigned, but not dynamically created.

We can assign any role to Company, but it will have to be one of the roles already present in the system. In other words, it will have to be an existing class.

Classes aren't created dynamically—but objects are. The following example (Figure 11) shows how you can dynamically specify role relationships by using objects.

# THE ROLE ASSOCIATION CLASS SOLUTION

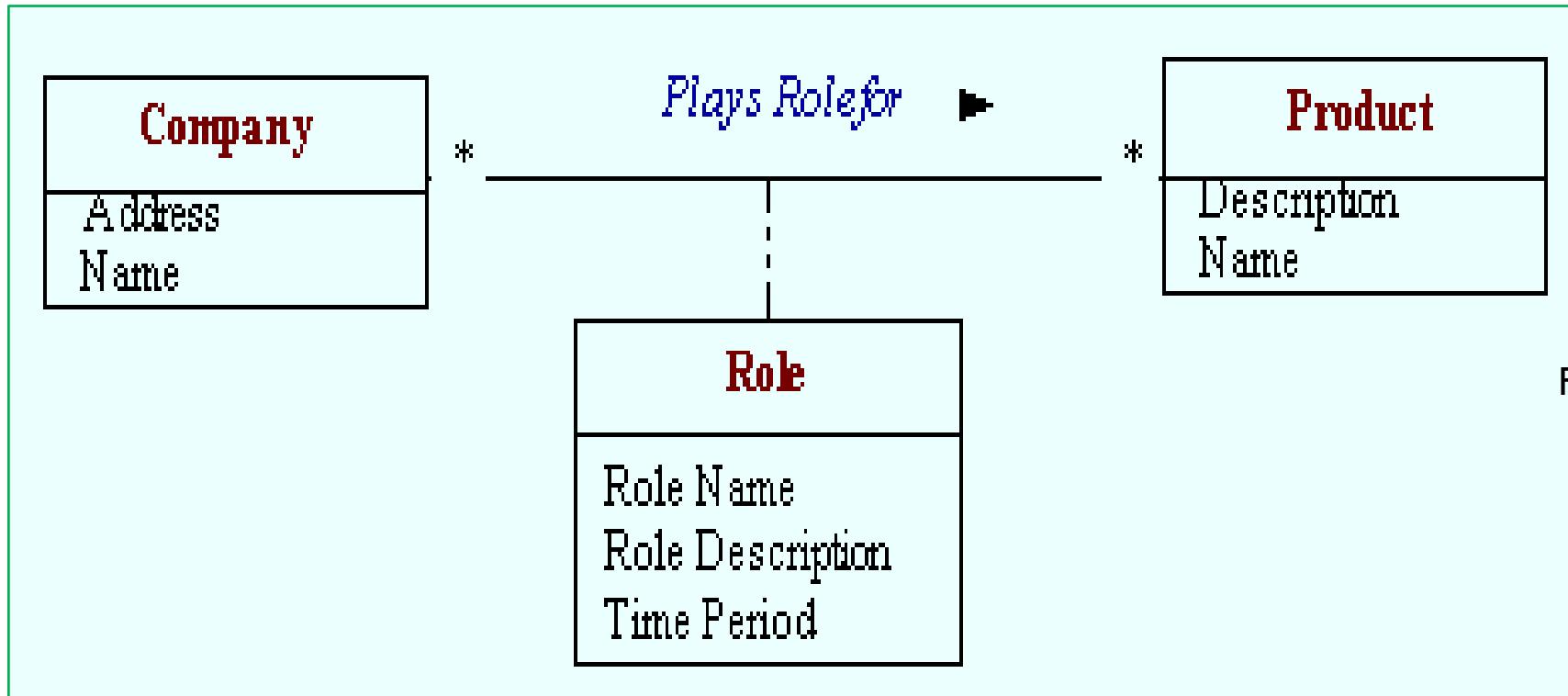


Figure 11

# THE ROLE ASSOCIATION CLASS SOLUTION

In this case, **roles are created on demand**. To specify that a company supplies a specific product, then one just needs to create an instance of Role—a link object—and link the proper product to the right company. That role link object can be named whatever one desires it to be: vendor, customer, broker.

Note another interesting feature of this role association class: *Time Period*. Many business rules require relationship time periods to be specified.

The **association class role** solution is flexible and powerful. Notice that the concept of roles is modeled—not roles themselves. This model is more abstract than models that deal with explicit roles. In general, you will see that **higher levels of abstraction** bring not only greater simplicity, but also greater power and flexibility. However, notice that more abstract models impose greater object manipulation. For instance, to solve the Vendor/Customer/Broker problem we looked in Figure 9, the application user would need to generate three link objects—one for each role.

# THE ROLE ASSOCIATION CLASS SOLUTION

This solution allows for all Roles and Role Types to be dynamically created and used. This model is abstract, powerful, flexible and object-intensive, as opposed to class-intensive. In the earlier models you would need one class per role. In this new model, one class (Role Type) takes care of all cases, but each actual Role Type (e.g.: Customer, Vendor, ...) still requires the creation of a Role Type object.

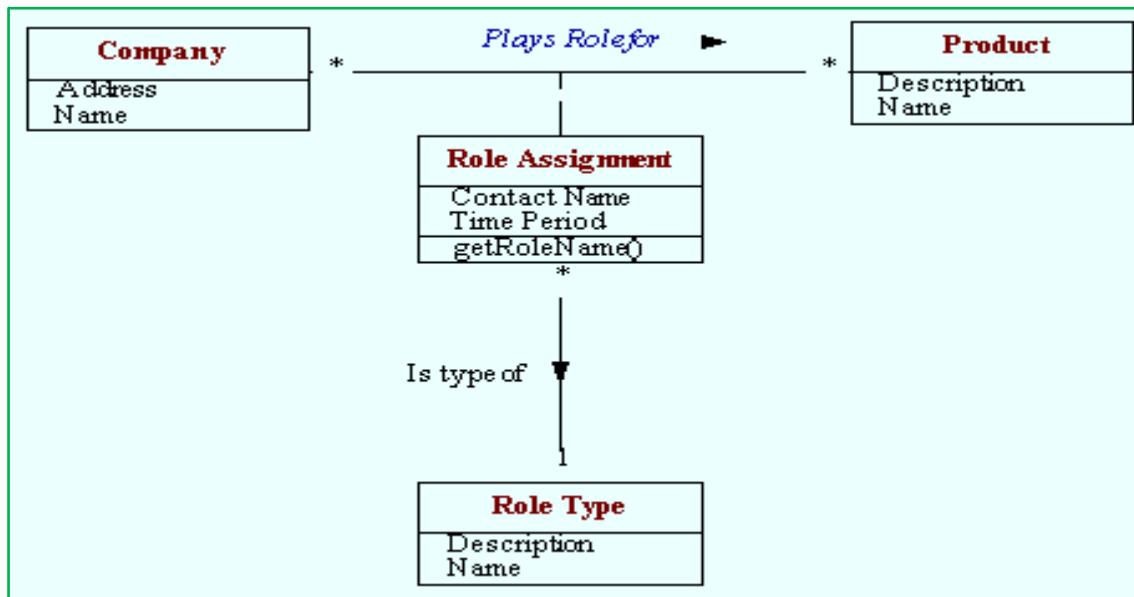
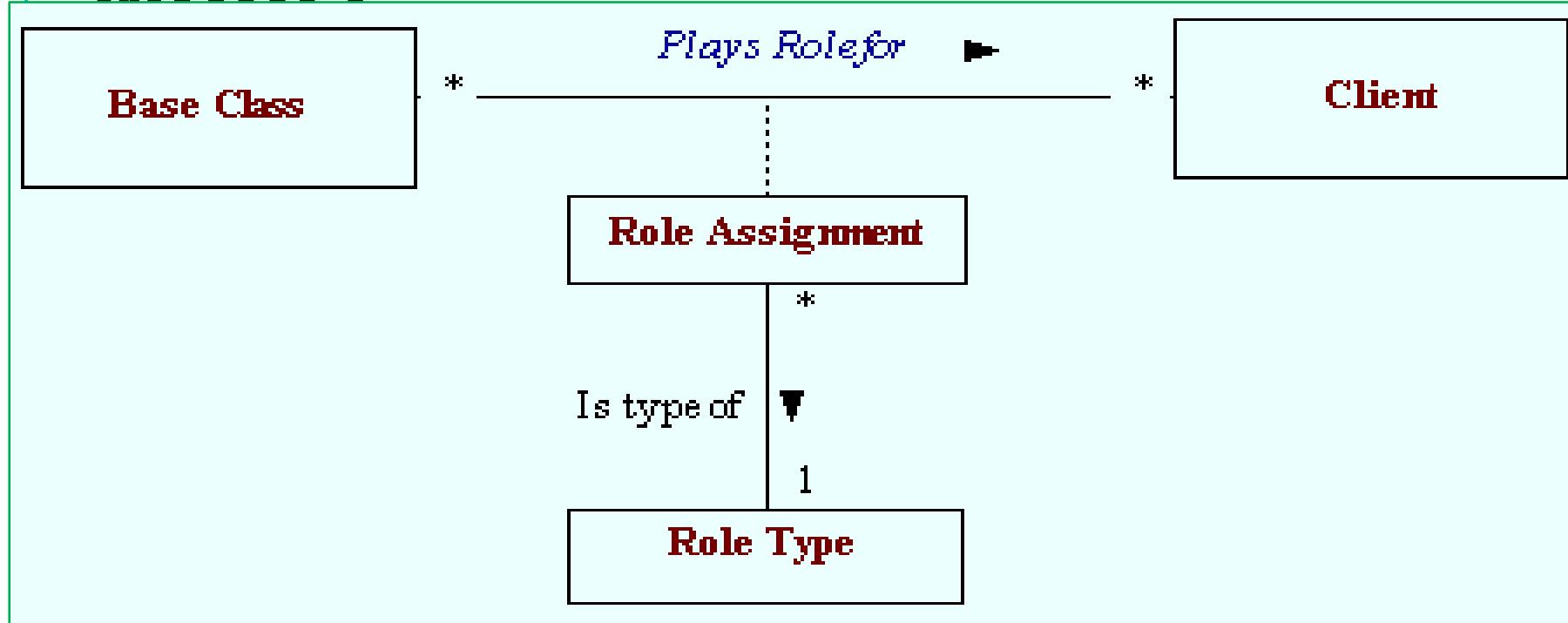


Figure 12

# THE ROLE ASSOCIATION CLASS META MODEL



# CONCLUSION

We have seen a series of role requirements with their corresponding solutions, ranging from simple association roles up to role classes and role association classes.

We've looked at the strengths and weaknesses of each solution. Role problems are very frequent.

By knowing the solutions discussed here—also called patterns or metamodels—and their characteristics, one can quickly come up with the optimum model for any role situations.