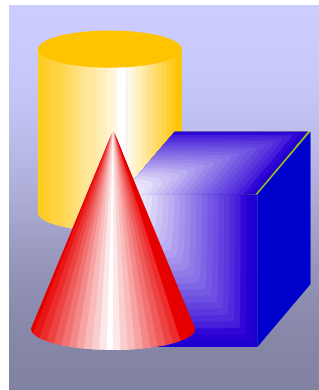


Object Orientation

Unit 3

Thinking in Objects

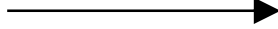


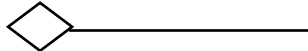
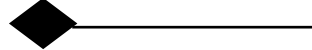
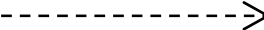


Thinking in Objects

■ Topics

- Is-a, Has-a and Uses relationships
- Multiplicity, navigability & role name
- Case Study
 - Identifying objects real time
 - Analyze object relationships
 - Handle the complexity of these relationships
 - The tricks of designing a complex system with several collaborating objects

Classes and Relationships

- Six types of relationships can exist between classes
 - Generalization 
 - Realization 
 - Association 
 - Aggregation 
 - Composite Aggregation (Composition) 
 - Dependency 

Relationships

- Classification

<<Is-a>>

Generalization
Realization

<<Has-a>>

Association
Aggregation
Composition

<<Uses>>

Dependency

- For all practical purposes we will represent

- Is-a relationship as



- Has-a relationship as



- Uses relationship as



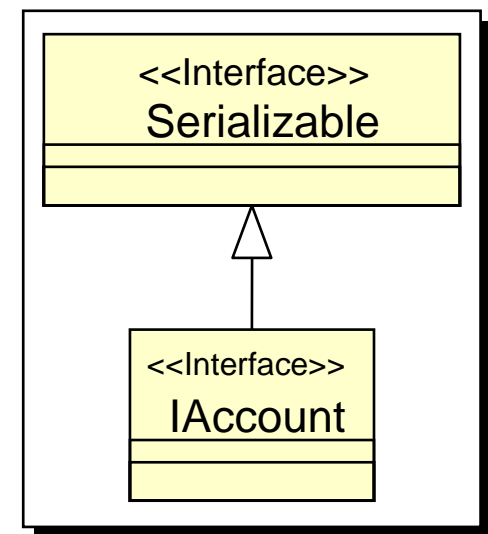
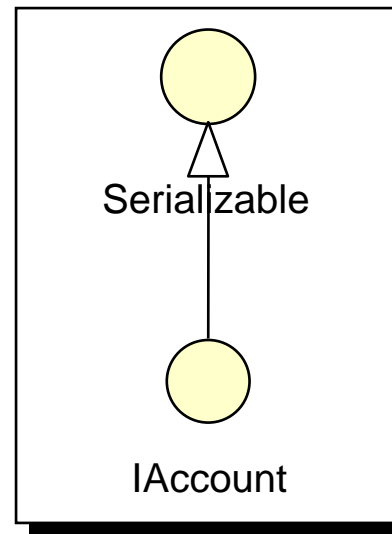
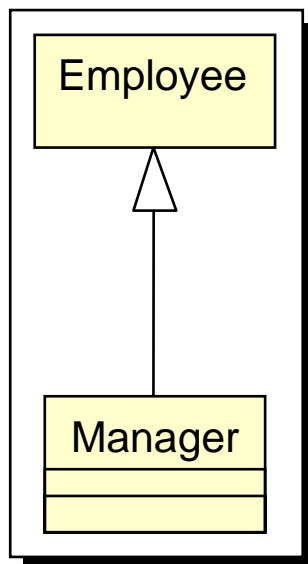
Generalization

Generalization →

- Relationship between two classes or two interfaces
- Represented by 'public' Inheritance in C++
- Represented by 'extends' in Java
- Represented by : in C#

```
class Manager : Employee { }
```

```
struct IAccount : Serializable { }
```

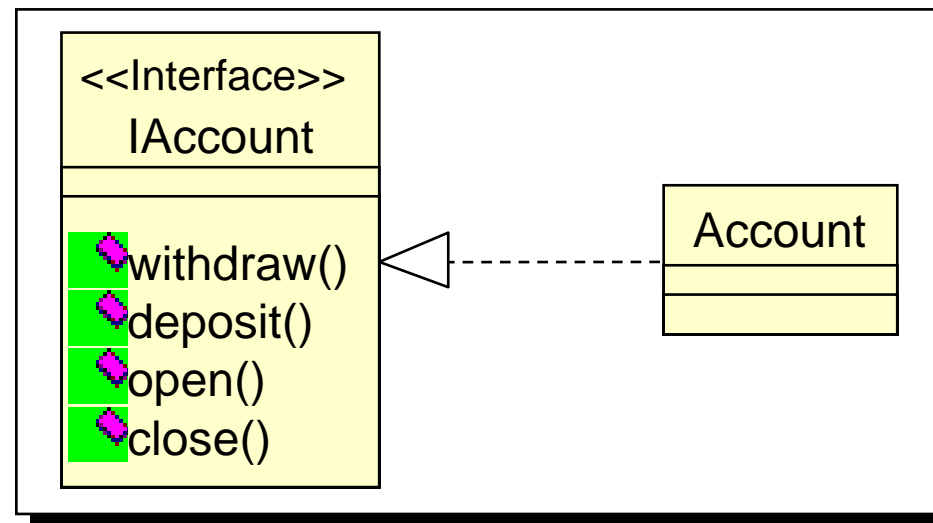
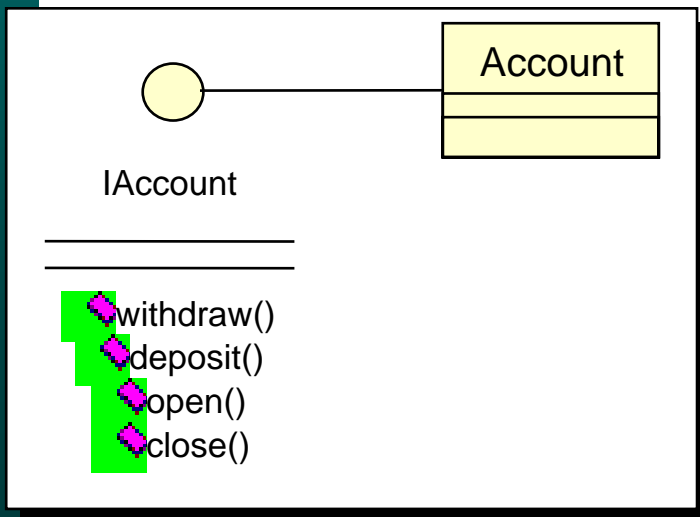


Realization

Realization ----->

- Relationship between a class and an interface
- Represented by 'implements' in Java
- Represented by 'public' inheritance in C++
- Realizing an interface would require a class to provide an implementation for all the inherited method declarations failing which the class becomes abstract

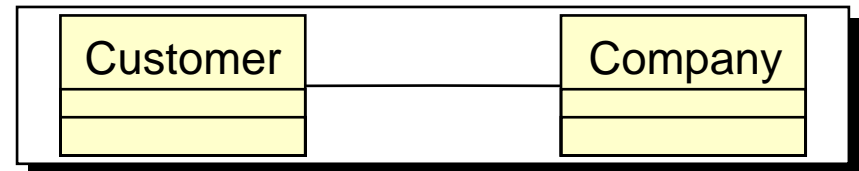
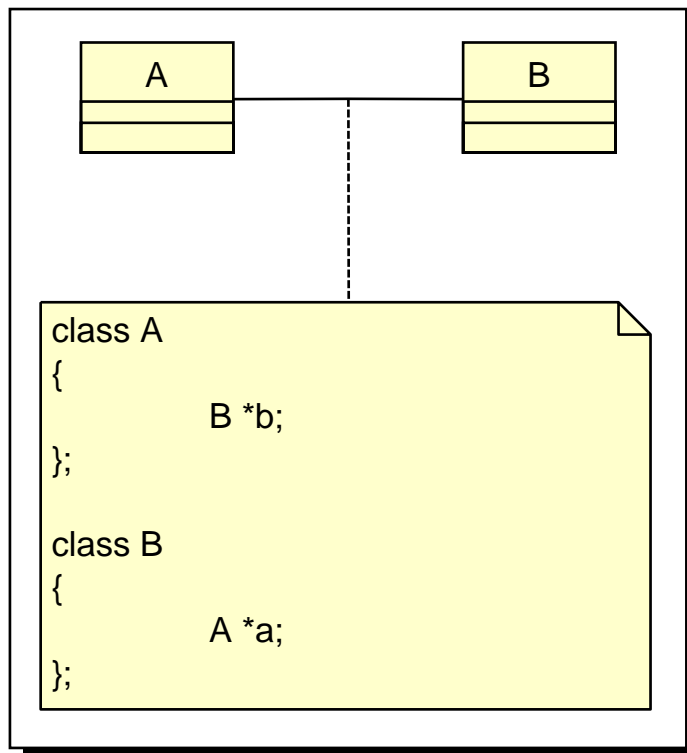
```
class Account : IAccount { }
```



Association

Association

- 'Has-a' relationship
- Semantic relationship between two or more classifiers that involve connections among their instances



A Customer is associated with a Company is essentially a 'has-a' relationship between a Customer and a Company

```

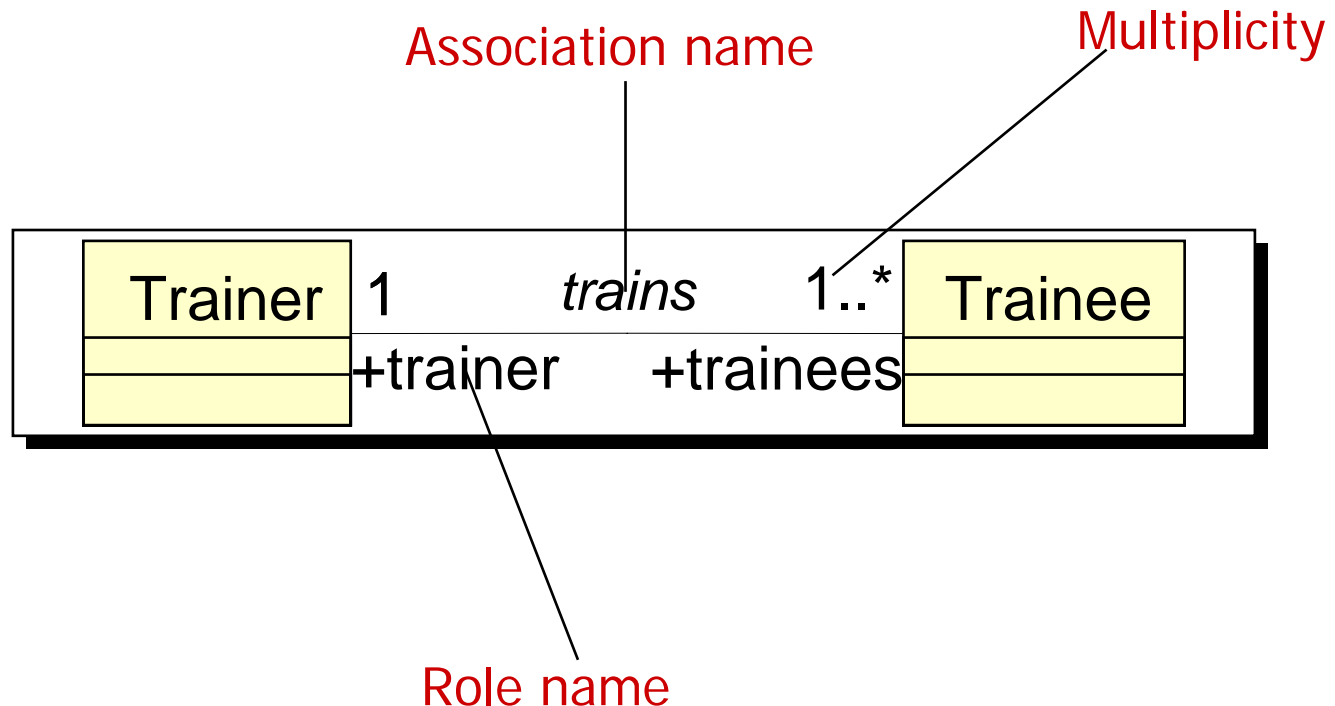
class Customer
{
    Company* company;
};
  
```

Association

- The associations are qualified by
 - Multiplicity
 - The number of instances with which a class is associated
 - Can be 1, 0..1, *, 1..*, 0..*, 2..*, 5..10, etc.
 - Multiplicity is by default 1
 - Navigability
 - Can be unidirectional or bidirectional
 - Navigability is by default bi-directional
 - Role name
 - The name of the instance in the relationship
 - Multiple associations based on different roles are possible

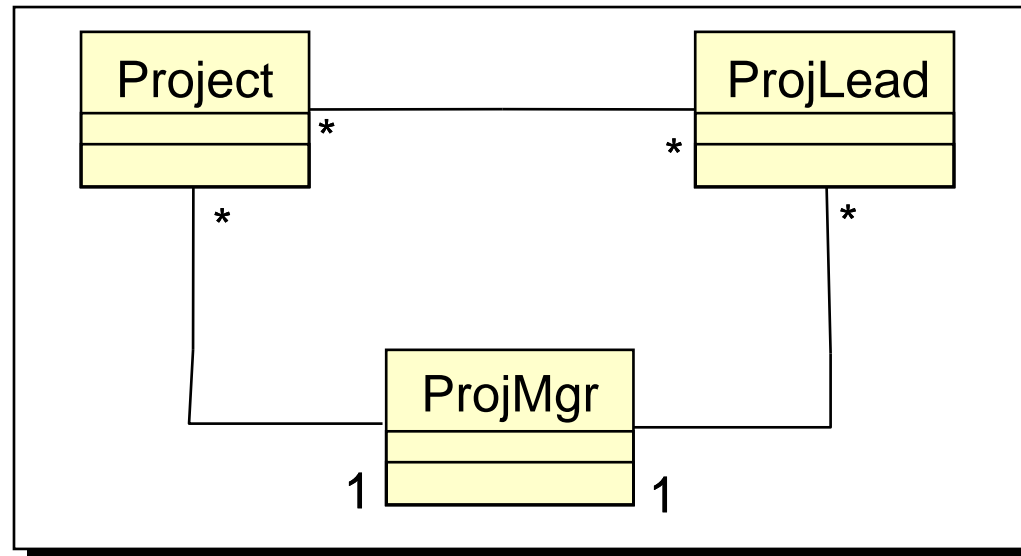
Association

- Role name, navigability and multiplicity



Association

- Examples



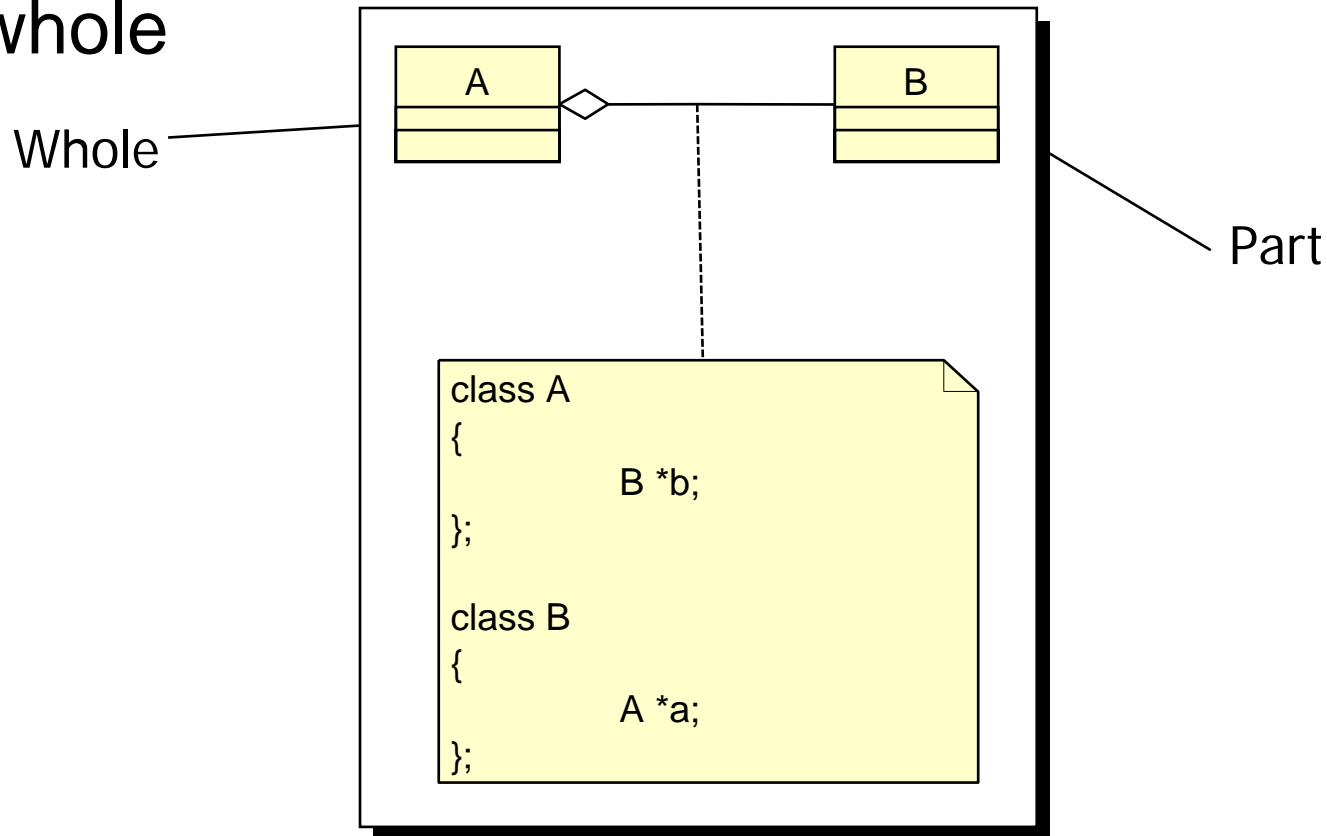
Aggregation

Aggregation 

- 'Has-a' relationship
- Is a special (stronger) form of association which conveys a whole part meaning to the relationship
 - Also known as Aggregate Association
- Has multiplicity and navigability
 - Multiplicity is by default 1
 - Navigability is by default bi-directional

Aggregation

- The hollow diamond is placed towards the whole

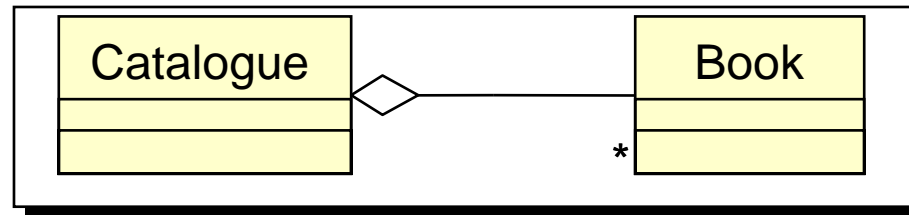


Aggregation

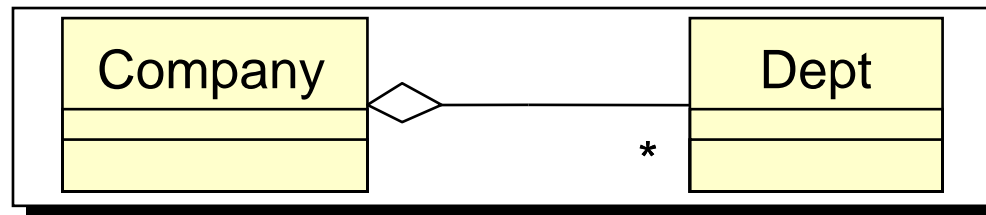
- So what's the difference between Association and Aggregation?
 - When it comes to code – NOTHING!
- Aggregation is a special meaning derived out of Association based on the context
 - Whole Part
 - Lack of independent use and existence
 - Scope of verb is constrained to only 'has' or 'contains'
- **When in doubt, leave aggregation out!**
 - Association and Aggregation exist to add clarity and not to introduce ambiguity

Aggregation

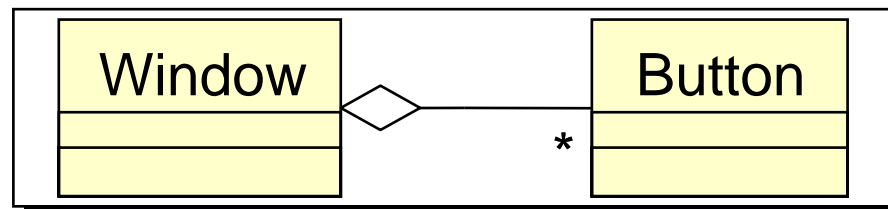
- The Whole-Part nature



- Independent existence / use

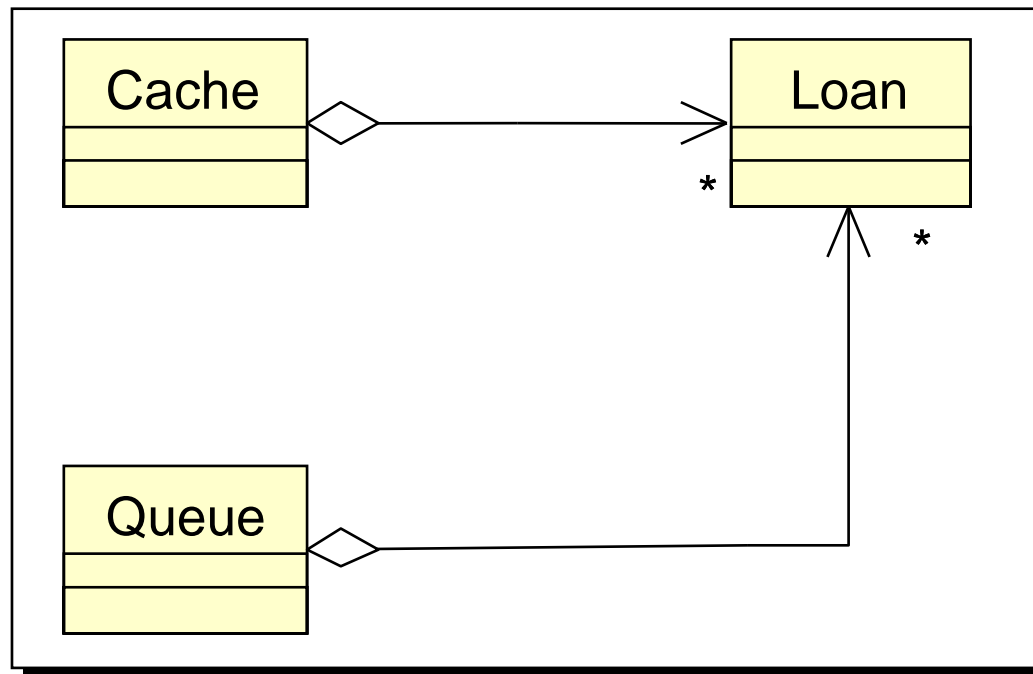


- Scope of verb is constrained to only 'has' or 'contains'



Aggregation

- A part can be shared between many wholes
 - Shared aggregation



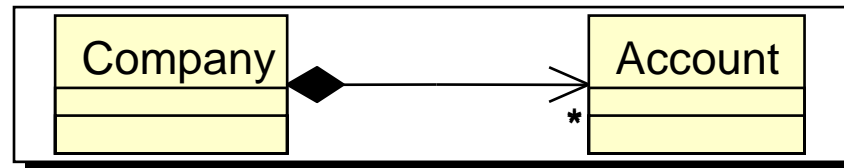
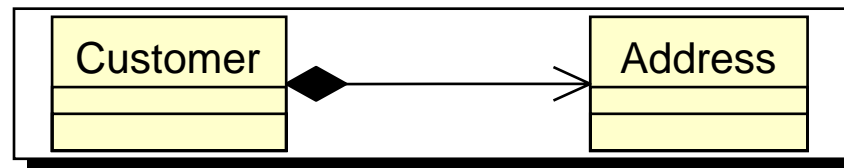
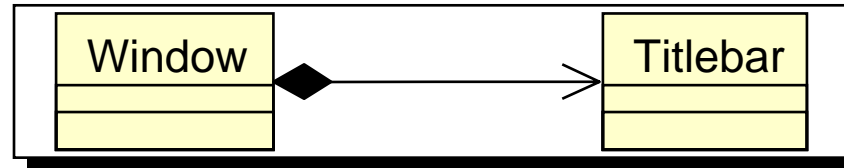
Composite Aggregation

Composite Aggregation 

- 'Has-a' relationship
- Is a stronger form of aggregation
 - Also known as Composition or Containment By Value
- A part cannot be shared between many wholes
 - Unshared aggregation
- Explicit lifetime control
- Exclusive ownership
- Has multiplicity and navigability
 - Multiplicity at the whole end should always be 1
 - Navigability is by default bi-directional

Composite Aggregation

- Examples



Dependency

Dependency ----->

- 'Uses' relationship
- Behavioral relationship
 - Loose coupling
 - Has no impact on class structure (data members)
- A class references another class only within its methods for the purpose of:
 - Invoking a static method
 - Local instantiation
 - Formal argument use
 - Return type

Dependency

- **Invoking a static method**

```
class B {  
    public:  
        static void method1() { }  
};
```

```
class A {  
    public:  
        void f1() {  
            B::method1();  
        }  
};
```

Dependency

- **Local instantiation**

```
class B {  
    public:  
        void method2() { }  
};  
  
class A {  
    public :  
        void f1() {  
            B *b1 = new B;  
            b1->method2();  
        }  
};
```

Dependency

- **Formal argument use**

```
class B {  
    public:  
        void method2() { }  
};
```

```
class A {  
    public:  
        void f1(B &b) {  
            b1.method2();  
        }  
};
```

Dependency

- **Return type**

```
class B { };
```

```
class BFactory {
```

```
public:
```

```
    B* create() {
```

```
        // check some condition
```

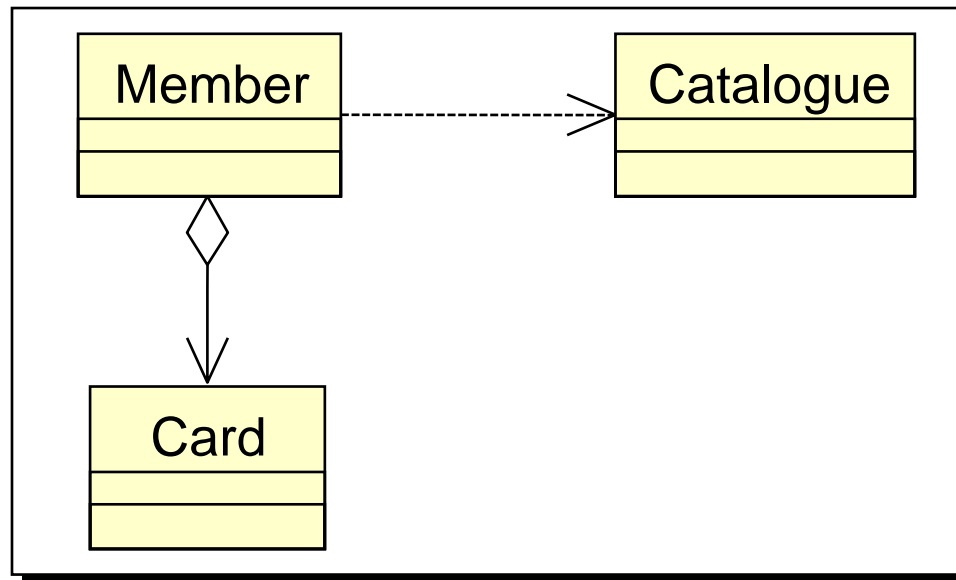
```
        return new B;
```

```
    }
```

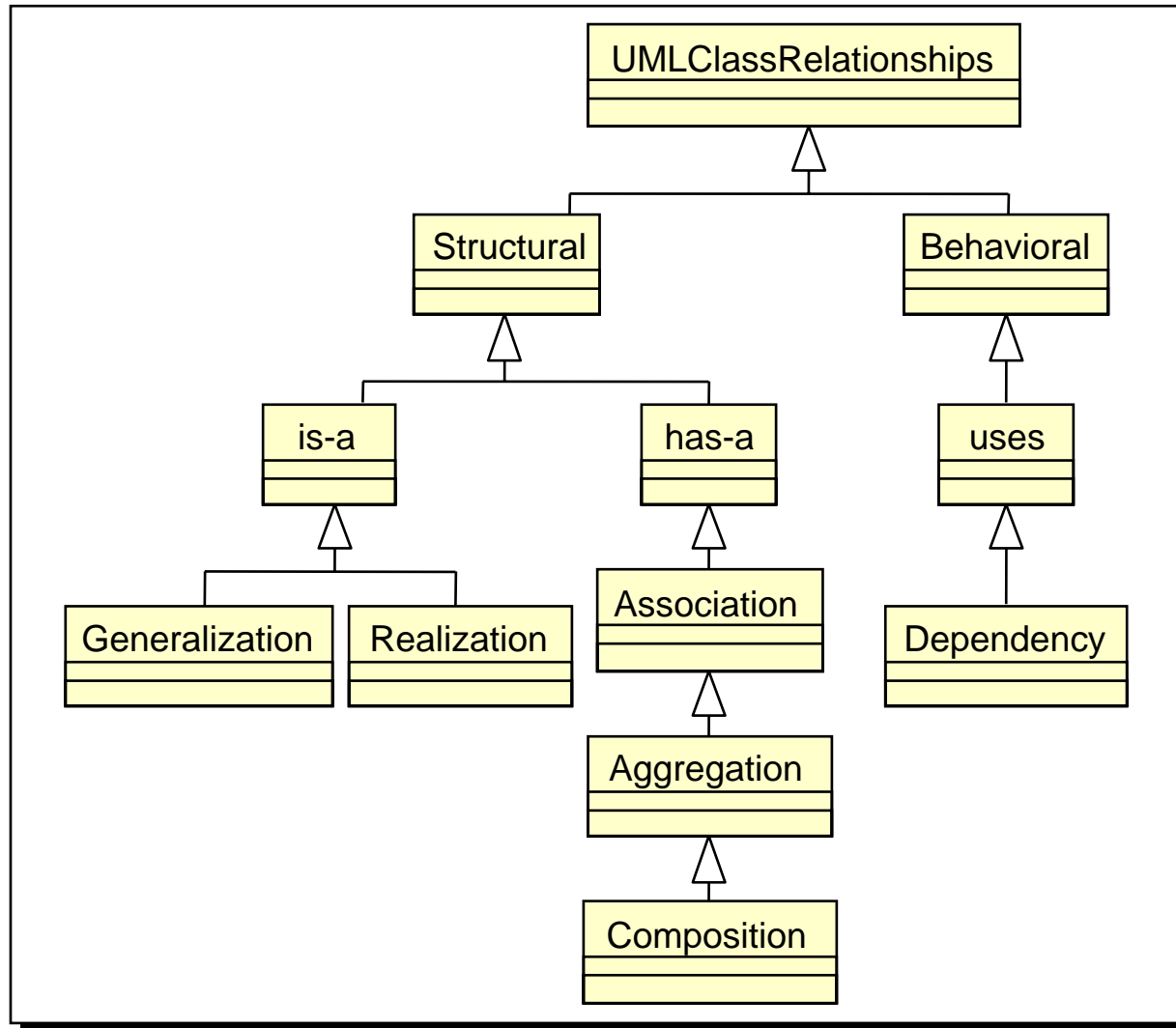
```
};
```

Dependency

- Example
 - A member who has a card can use a Catalogue to view book information



Relationship Meta Model

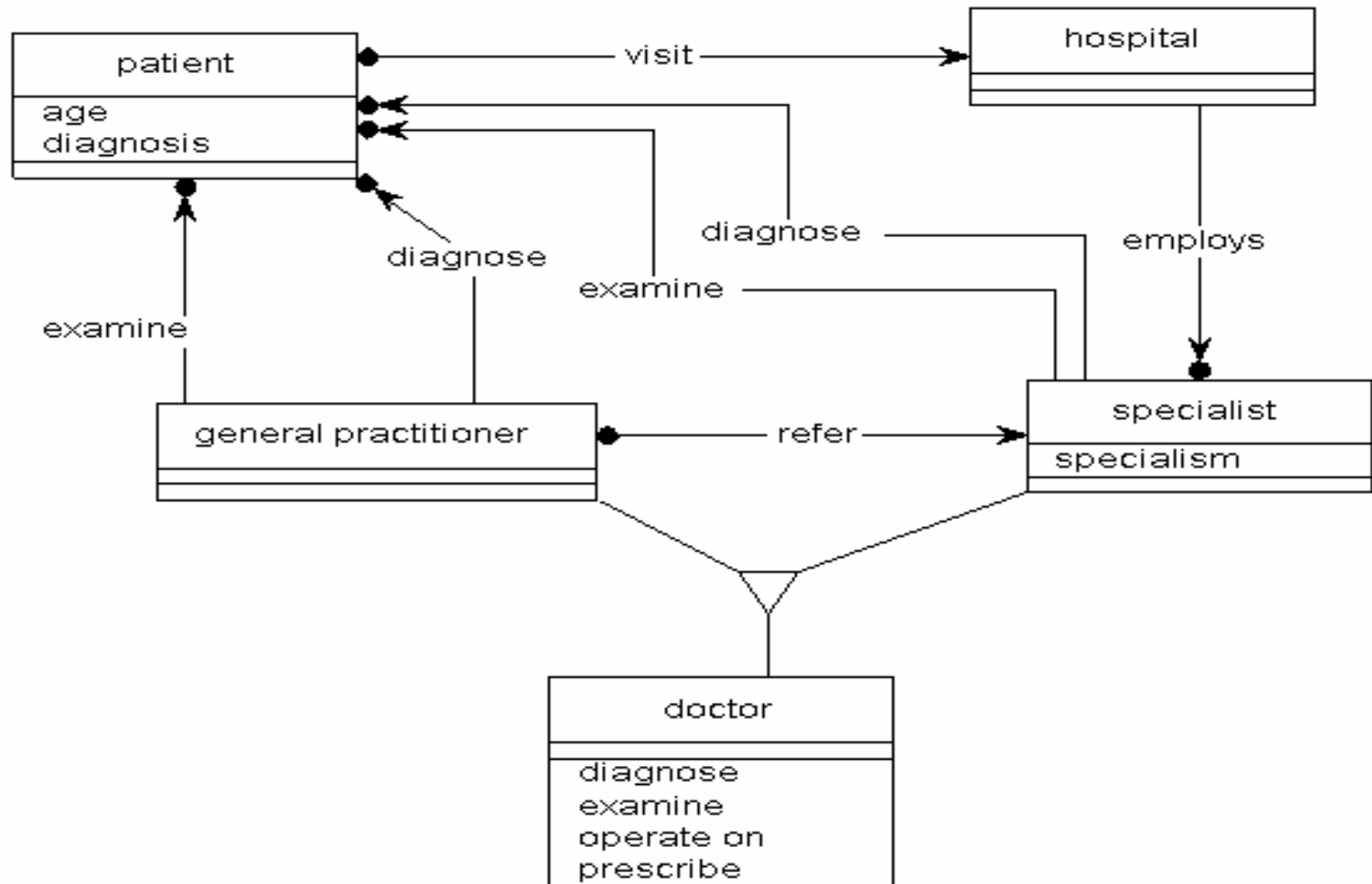


Exercise 1

- **Construct an object model for one of the following:**
 - Patient referrals by GP's to specialists
 - Lift system
 - A Computer System
 - The football league,
 - Star Wars,
 - A personnel system,
 - Cinderella,
 - An automatic washing machine,
 - A video hire shop,
 - A kitchen.

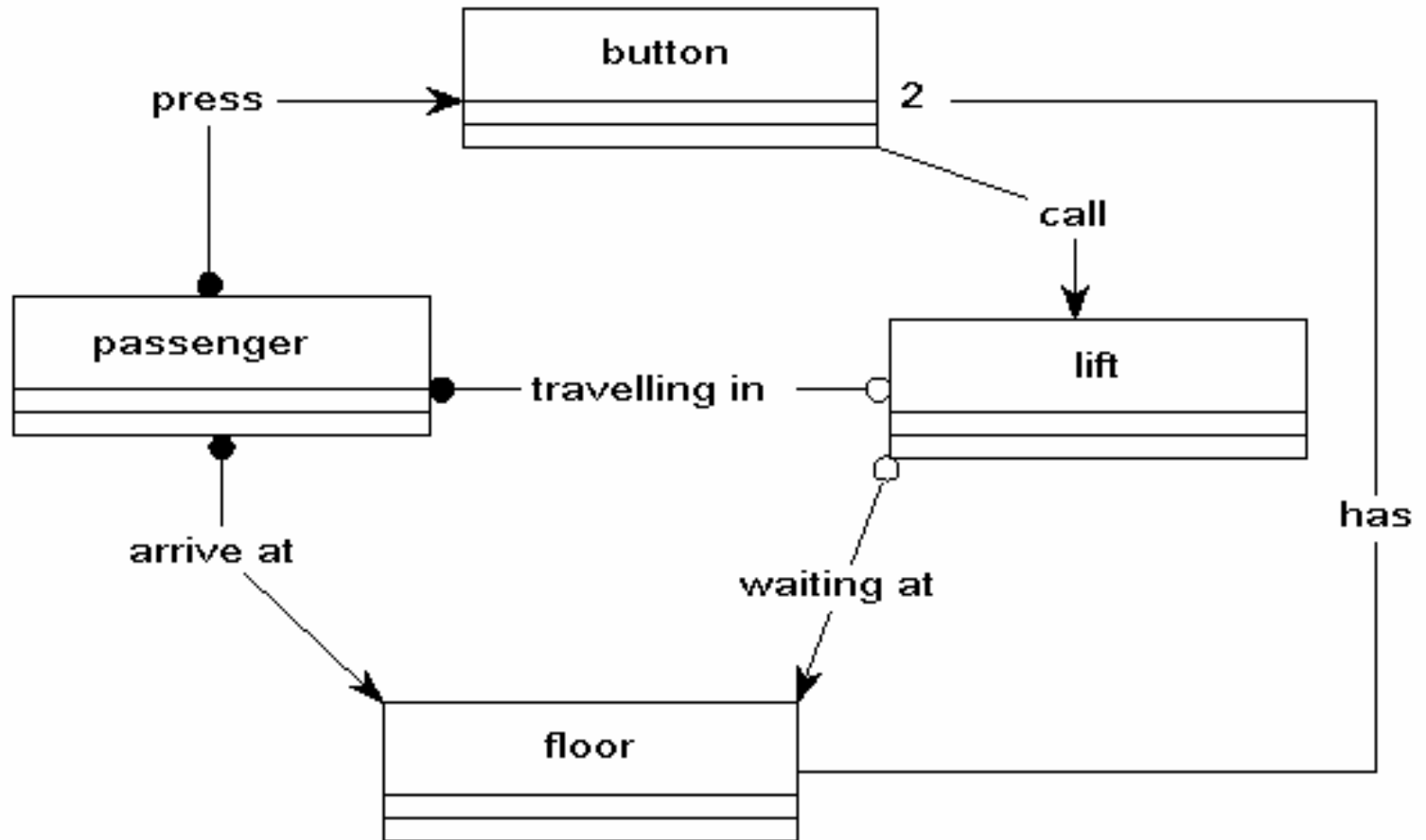
The Object Model of patient referrals by GP's to specialists

Object Orientation



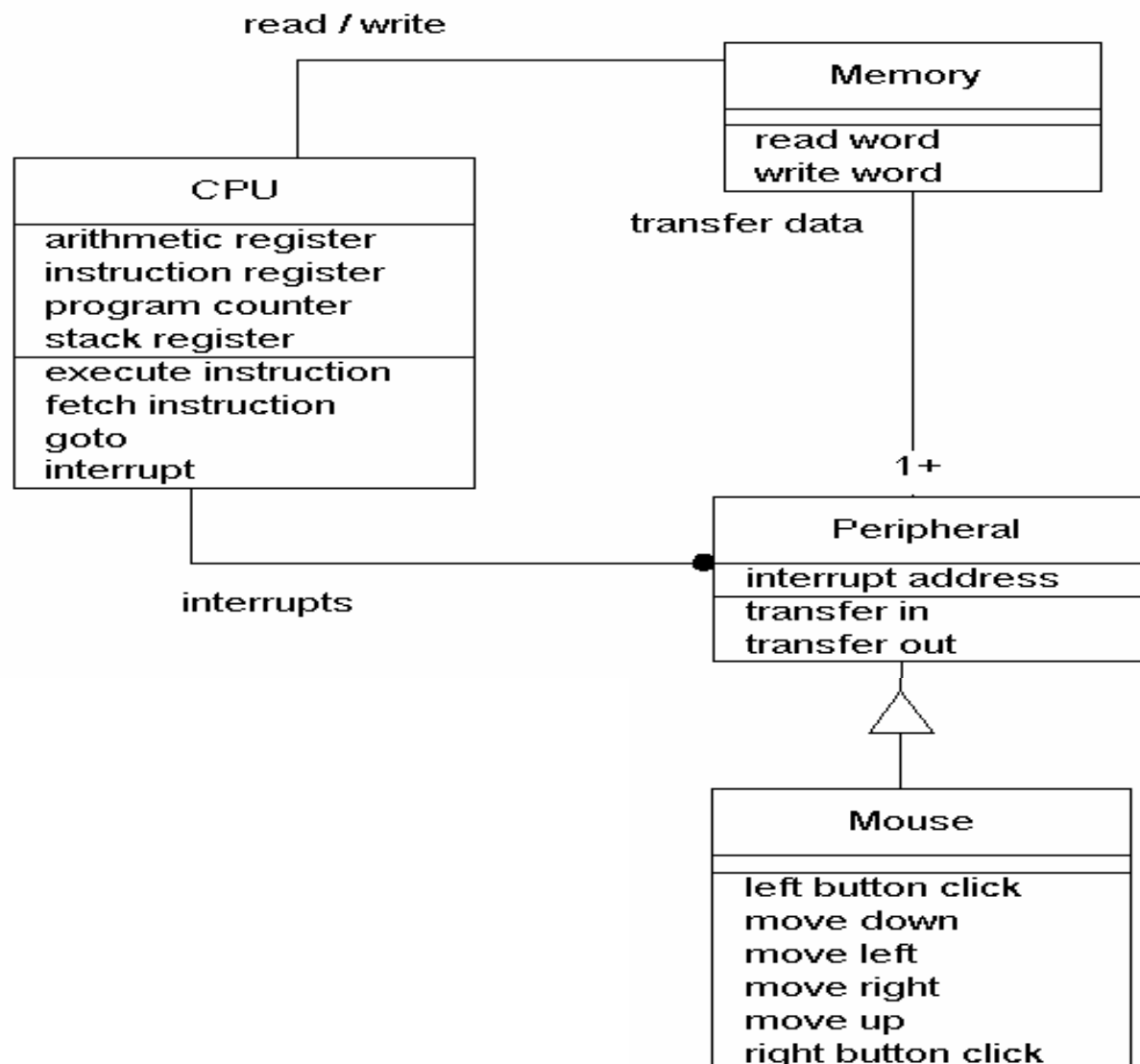
The Object Model of Lift System

Object Orientation



The Object Model of Computer System

Object Orientation



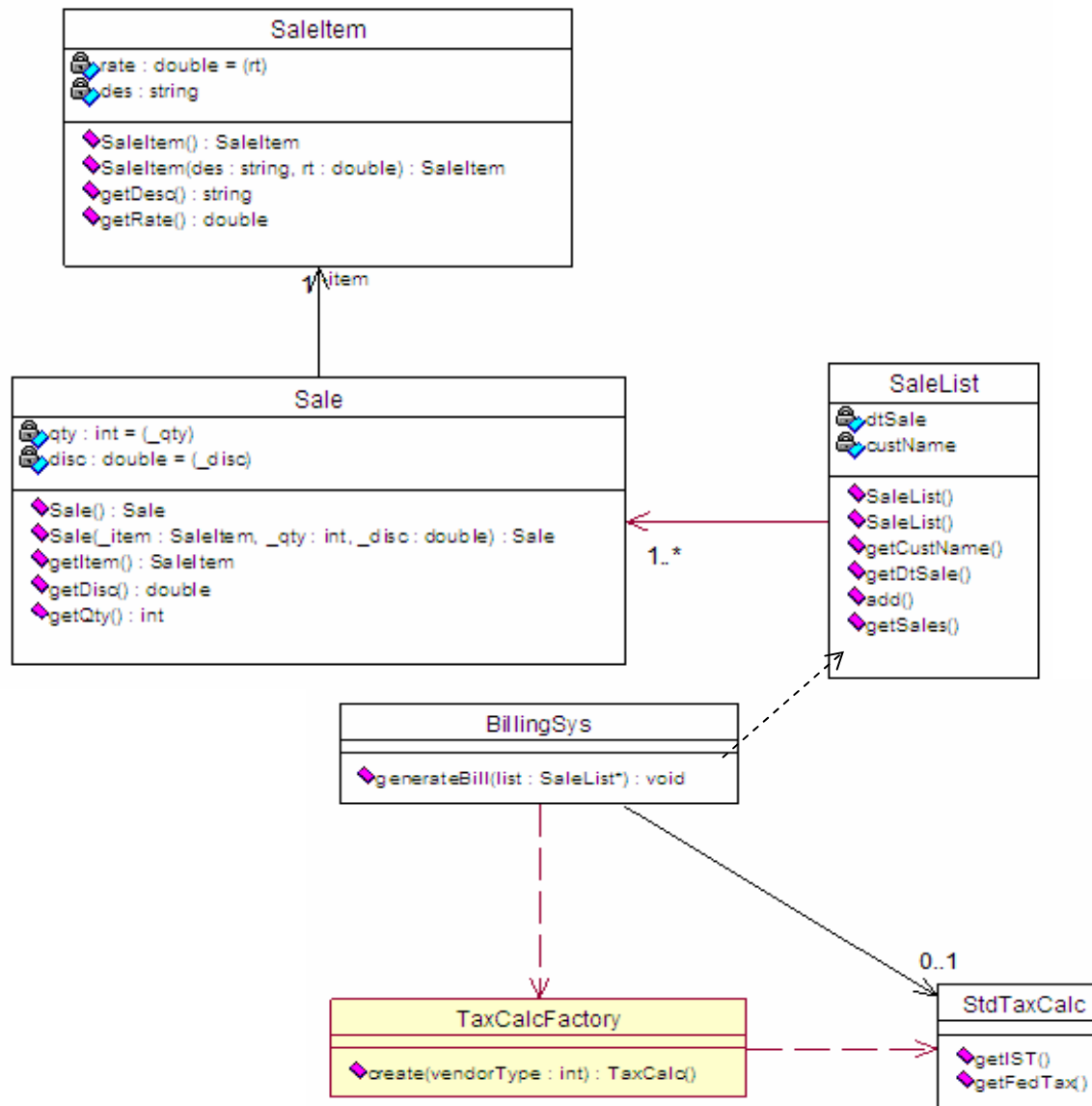
Point of Sale: Exercise 2

Object Orientation

- **In a Point Of Sale (POS) system the following requirements need to be addressed.**
- A SaleList has many Sale objects for a given customer. A Sale object is created whenever a Sale is made and contains information about the SaleItem being sold. The SaleItem is the item that is sold and has rate and description.
- The BillingSys has a method called generateBill to which the object of SaleList is provided. This method finds the grand total for the sale list by considering the qty and discount (from Sale) and the rate (from SaleList).
- Every time a sale is made some taxes need to be levied. For this purpose, there is a StdTaxCalc (a tax calculator) which helps in finding the IST (inter-state tax) and FedTax (Federal Tax).

Point of Sale: Solution

Object Orientation

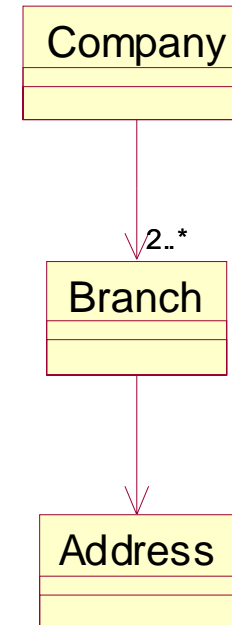


Exercise 3

- See **Exercise1.doc**
- Identify the classes
- Identify the relationships
- Please focus on relevant OO principles
- Convey your ideas using simple 'has-a', 'is-a' or 'uses' notations
- Focus on classes with attributes and methods

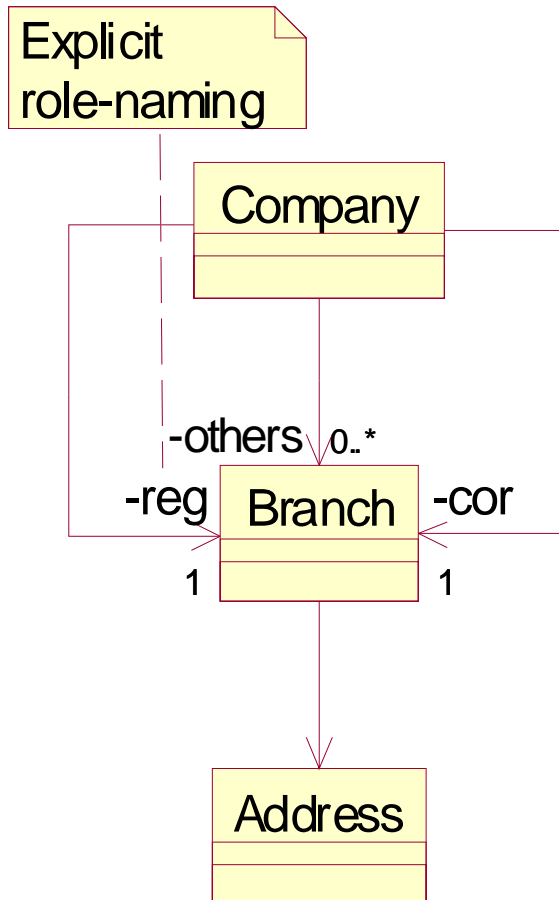
Solution

- Consider the following classes
 - Company
 - Branch
 - Address
- Company has two or more instances of the Branch
- Instances of Branch called corporate office and registered office are mandatory



Explicit role-naming technique

- Using this technique it is possible to add greater clarity to the representation



```
class Company
{
    Branch cor;
    Branch reg;
    Branch[ ] others;
};
```

Mixed Instance Cohesion

- Let us now consider the Customer class and its attributes

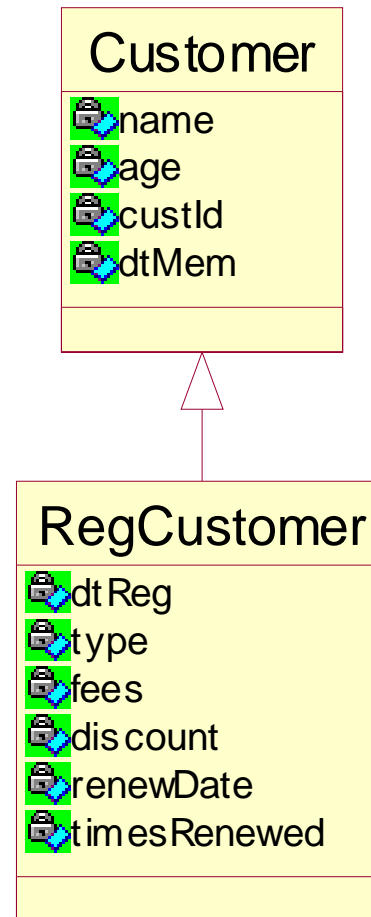
Mixed Instance Cohesion (MIC)



- Not all instances of Customer will need the complete information. Different instances have different cohesion – Mixed Instance Cohesion, which is a problem

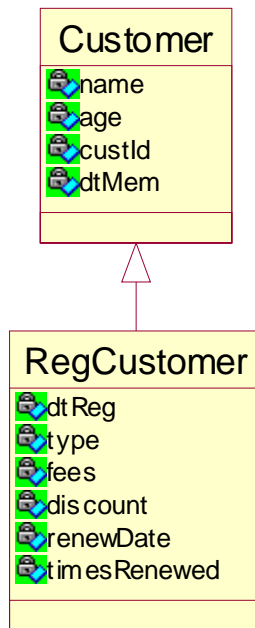
Specialization

- Use Specialization to resolve the problem of Mixed Instance Cohesion



Check for instance level cohesion

- Notice that the information pertaining to type, fees and discount needs to be properly updated across all the instances of the RegCustomer

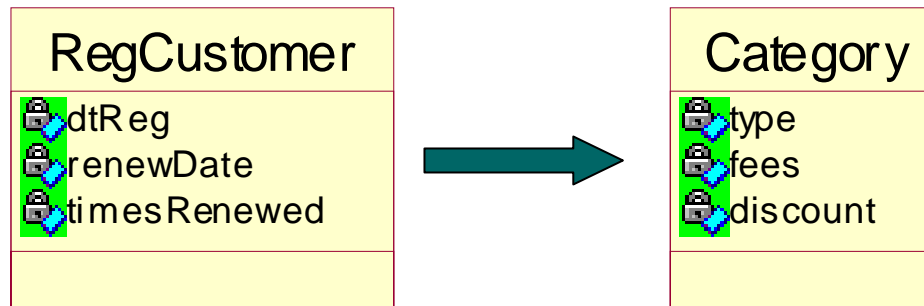


C001	ABC	35	10/10/05	A	2000	5%
C002	ABC	35	10/10/05	A	2000	5%
C003	ABC	35	10/10/05	A	2000	5%
C004	ABC	35	10/10/05	B	5000	10%
C005	ABC	35	10/10/05	B	5000	10%
C006	ABC	35	10/10/05	B	5000	10%
C007	ABC	35	10/10/05	C	15000	15%
C008	ABC	35	10/10/05	C	15000	15%

- There is redundancy in the information contained in the RegCustomer class, which is replicated unnecessarily across all its instances belonging to a given type

Decompose the class

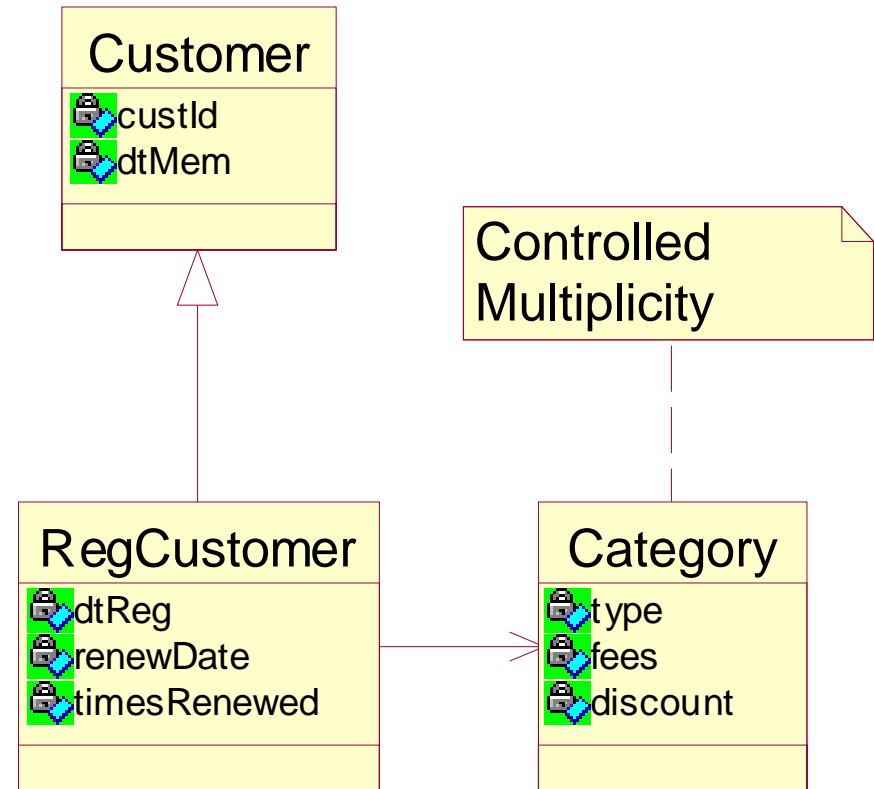
- It is hence important to check for instance level cohesion and eliminate redundancies



- Move the redundant information out of the **RegCustomer** class into the **Category** class
- However, **RegCustomer** needs to still reuse the information. How do you reuse this information – ‘is-a’ or ‘has-a’ ?

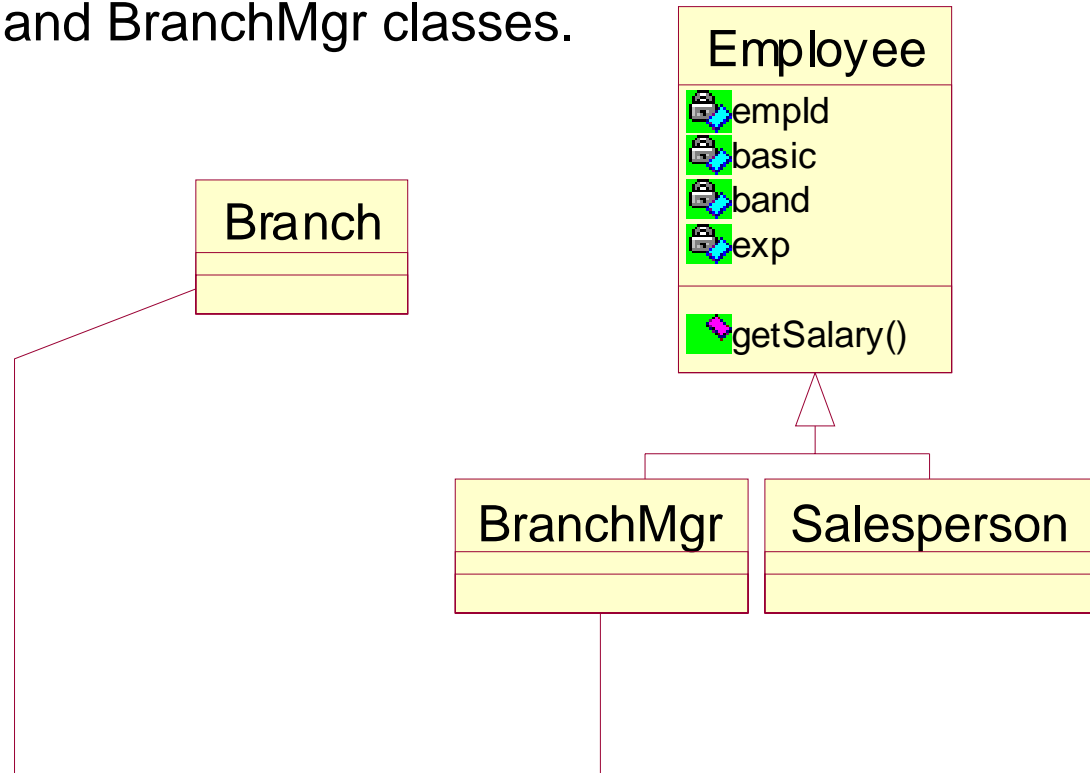
Controlled multiplicity

- Use 'has-a' whenever there is controlled multiplicity
 - There are many instances of RegCustomer
 - There are a few instances of Category
 - But each instance of RegCustomer does not have its own unique copy of category information (controlled multiplicity)



Class Hierarchy

- Consider the Employee classes. A hierarchy is created by the Salesperson and BranchMgr classes.

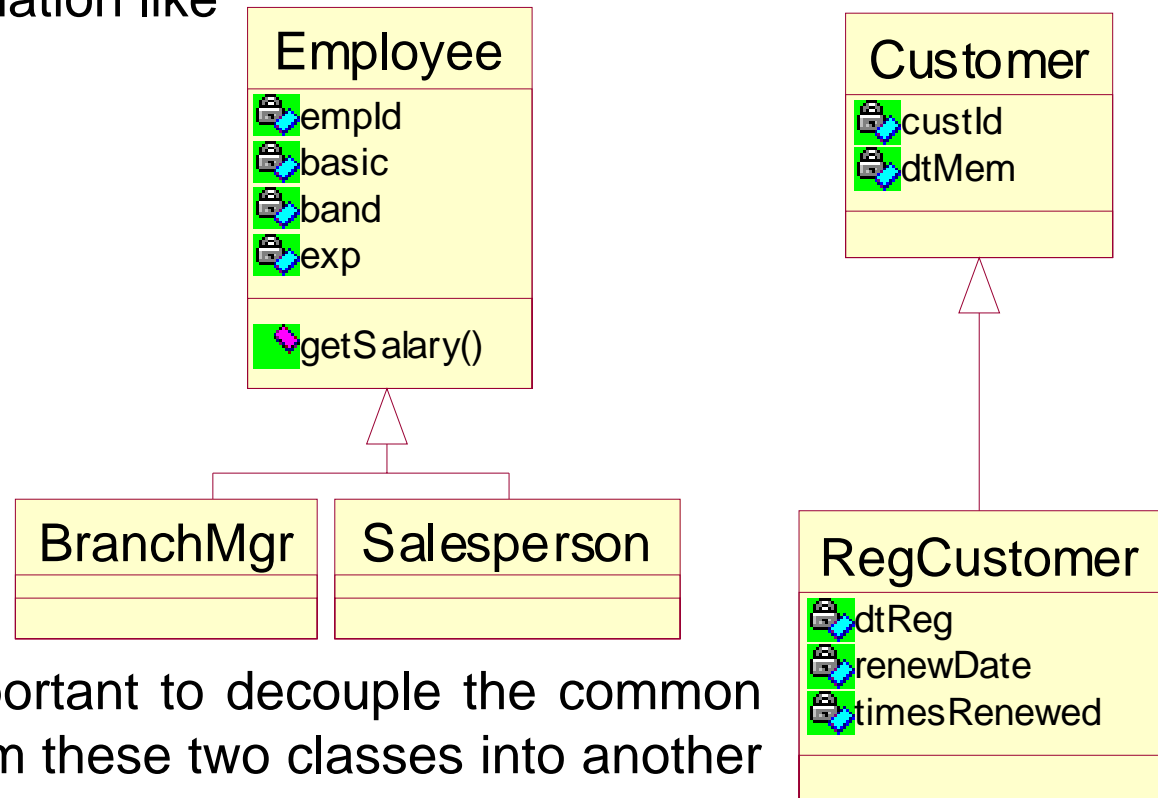


- Every Branch has a BranchMgr
- A BranchMgr manages a Branch

Problem of reuse

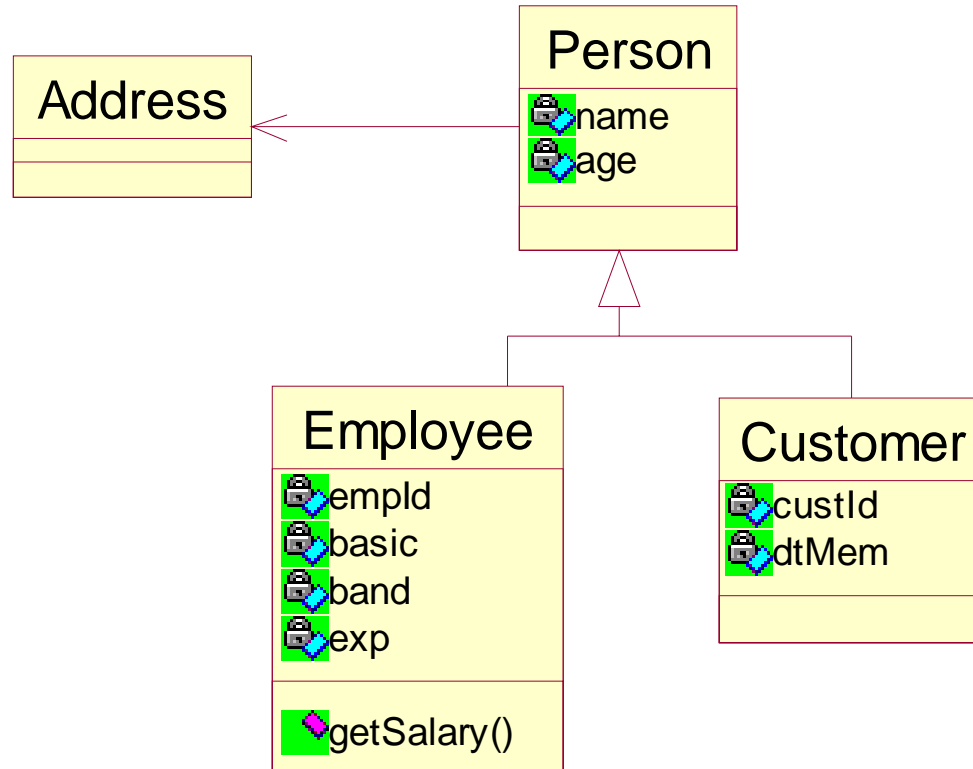
- Both the Employee class and Customer class have some common information like

- Name
- Age
- Address



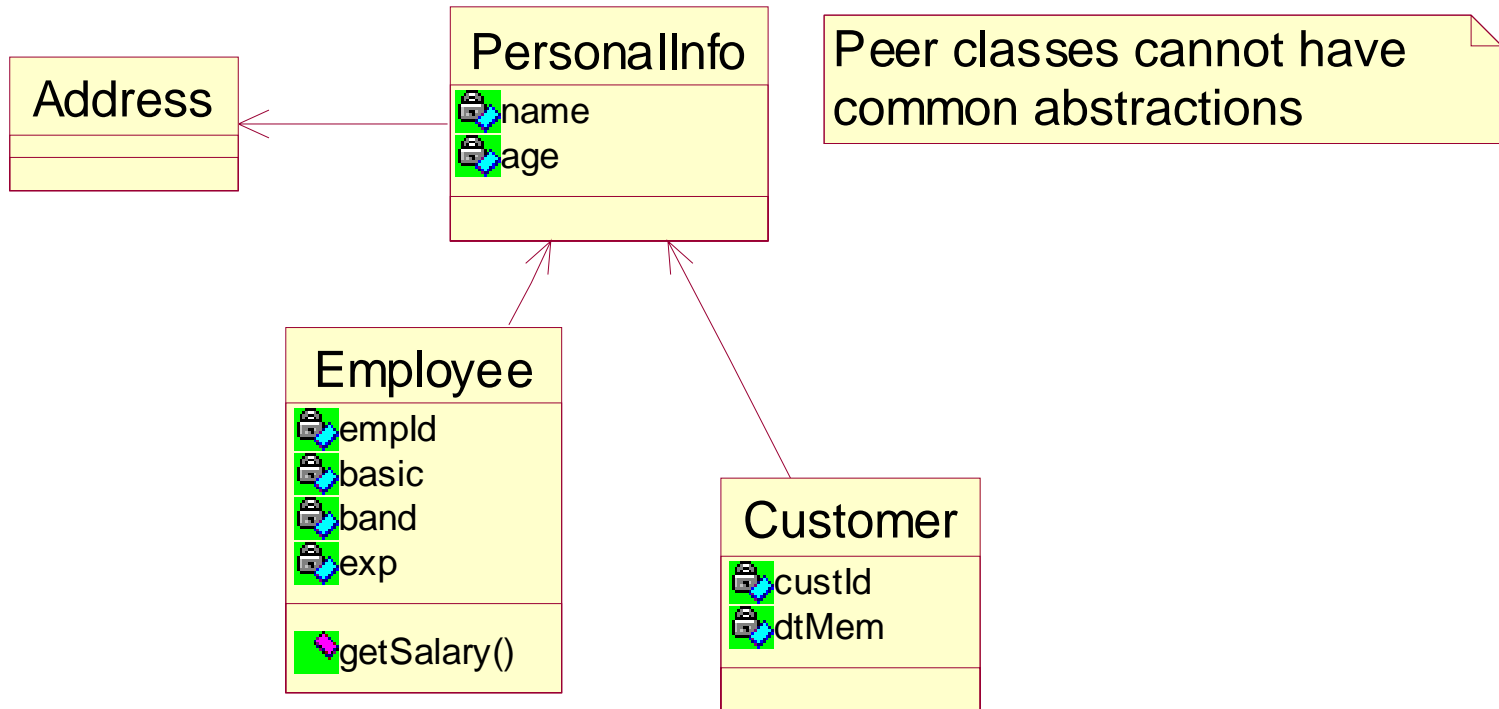
- It is hence important to decouple the common information from these two classes into another class and reuse the same
- How do you promote the reuse (is-a or has-a) ?

Reuse through Inheritance



- Do the **Employee** and **Customer** classes have any common abstractions?

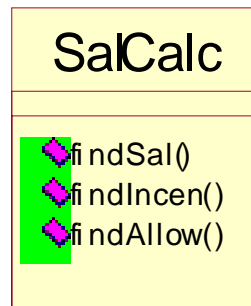
Reuse through 'Has-a'



- The Employee and Customer classes do not have any common abstractions (i.e. behavior), but only common structure
- They are known as peer classes
- Whenever two classes have only common structure (information) and not behavior use 'has-a' relationship

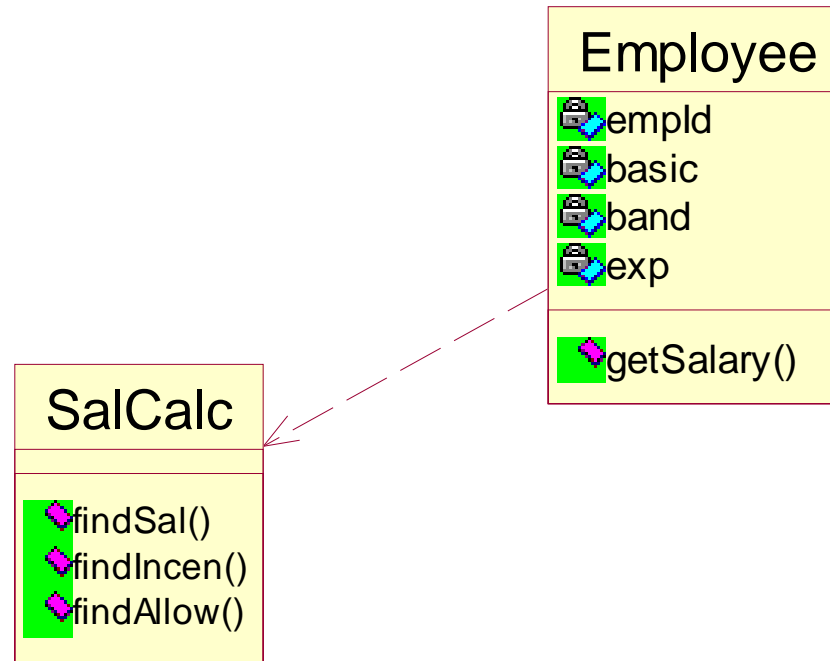
Changing rules

- Consider the implementation of `getSalary()` in the `Employee` class
- What if the rules change frequently?
 - Do not associate rules with an entity class
 - Decouple the rules into a conceptual entity known as the `SalCalc`



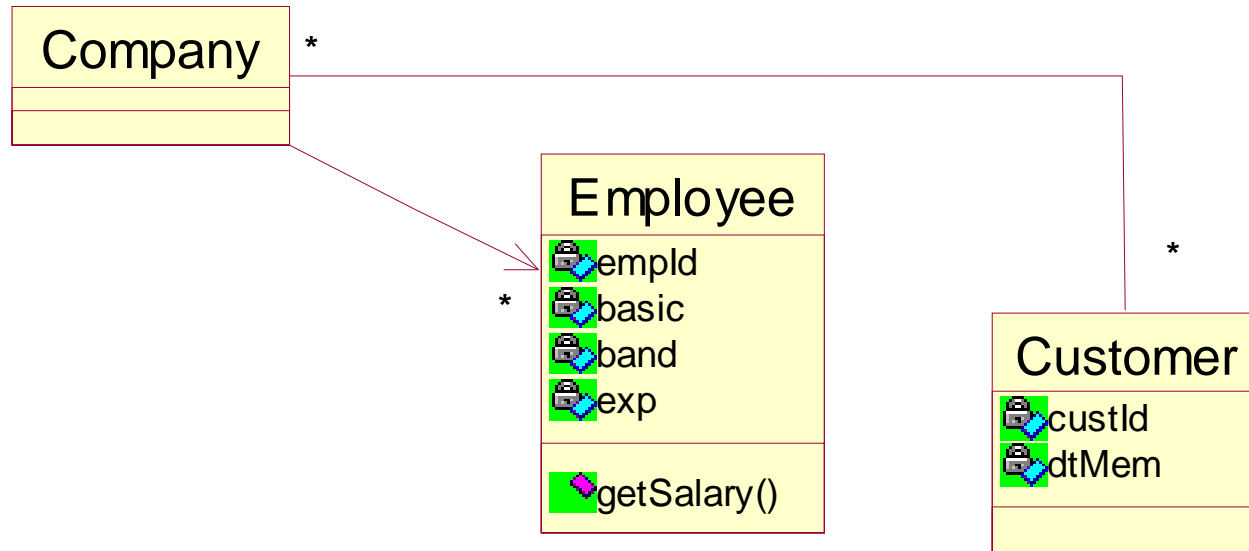
- How will the `Employee` invoke the methods of the `SalCalc` ?
 - Delegation through uses

Delegation through uses



- This delegation need not be done by using 'has-a' as the **SalCalc** does not contain any information, but only methods
 - Consider making the methods static
 - Avoid creating the instance of **SalCalc** for each call made to `getSalary()`

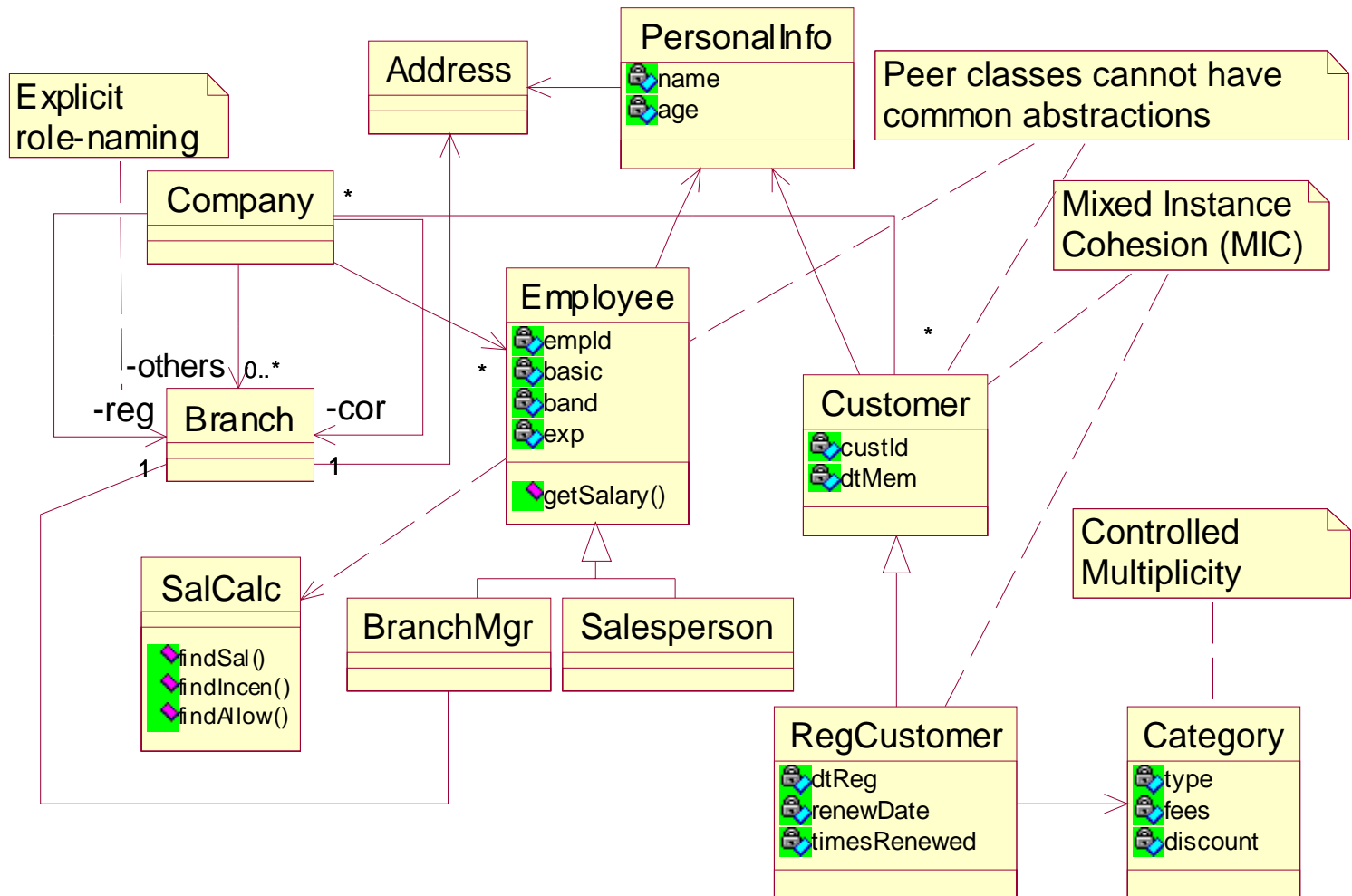
Associations to the Company



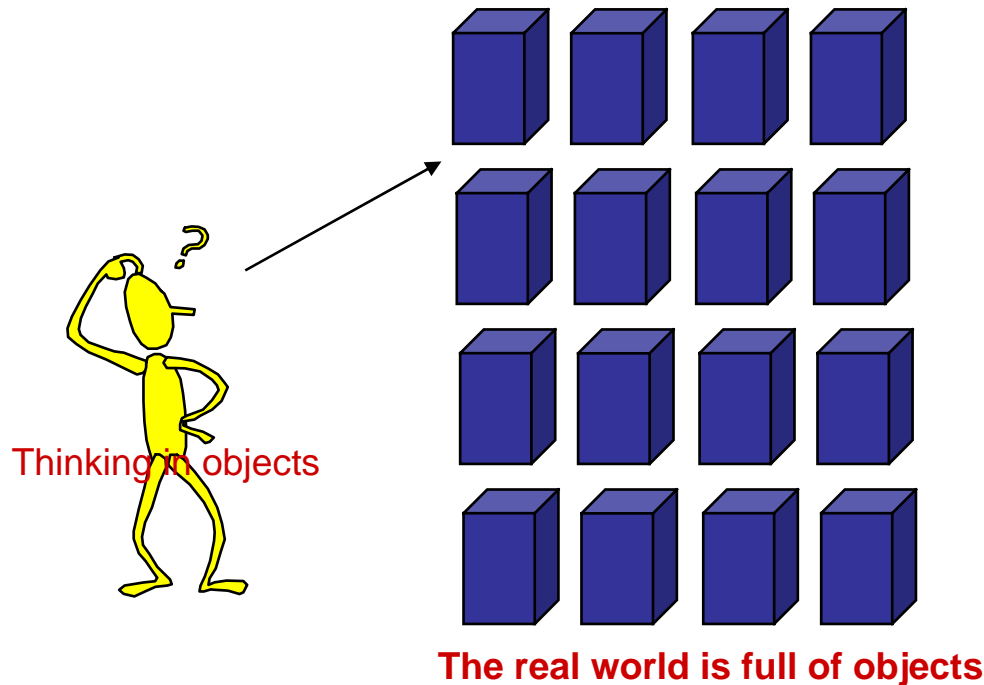
- Company 'has' many Customers
- A Customer is associated with many Companies
- Company has many Employees

Solution

Object Orientation



Conclusion



- Owning a hammer doesn't make one an architect
- Knowing an object-oriented language is a necessary but insufficient first step to create object systems.
- Knowing how to 'think in objects' is critical

Question Time



Please try to limit the questions to the topics discussed during the session.

Participants can clarify other doubts during the breaks.

Thank you.