Object Oriented Analysis

- **Class Diagrams – Part2 (Lecture 3)**

Sessional Outcomes

- Relationships Between Classes.
  1. Dependency
  2. Association
  3. Aggregation
  4. Composition
  5. Inheritance.

- Common Misuses of Inheritance

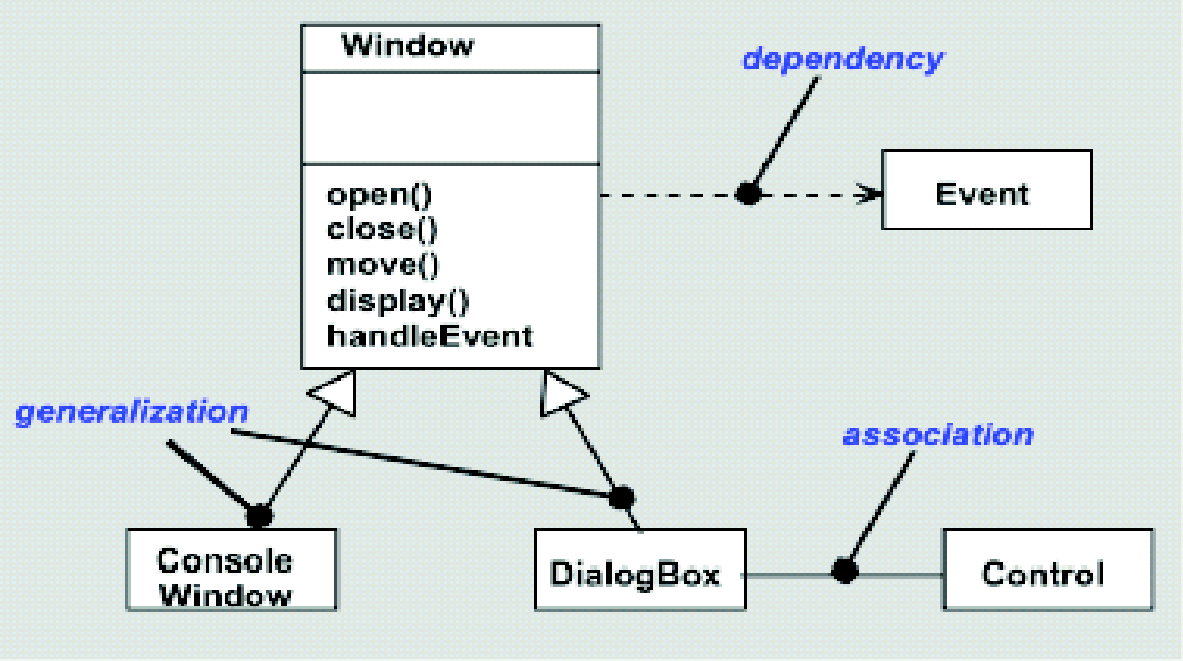
Introduction

There can be four types of relationships between two classes.

1. **Dependency:** one entity depends on the behavior of another in some way (dotted line with open arrow).
2. **Association:** one entity uses another as part of its behavior (solid line, with optional adornments).
3. **Aggregation / Composition:** one entity belongs to another. models "has a" relationships;
4. **Inheritance:** (a form of generalization), which models "is a" relationships.

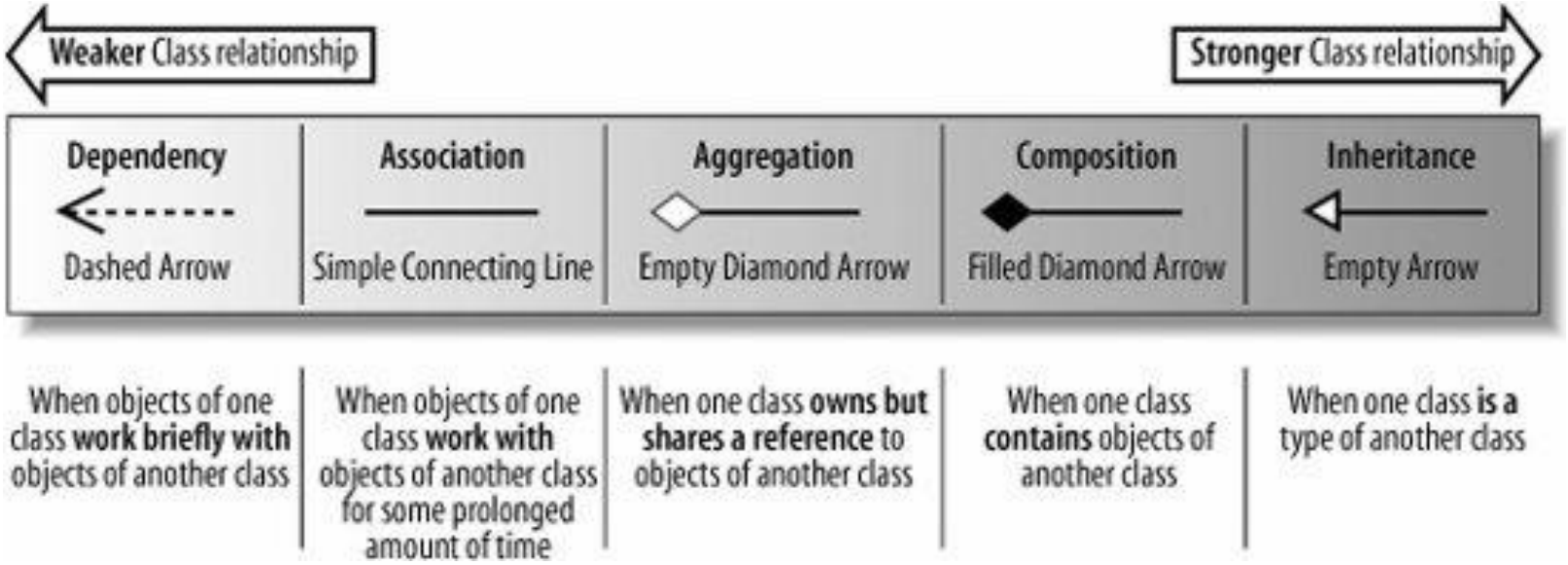
Introduction

Go through the following class diagram:



There are different types of relationships between classes !!!

Strength of Relationships



- Most weakest relationship : *Dependency*
- Most Strongest relationship: *Inheritance*

1. Dependency

Dependency

- It is the simplest form of relationship between two classes.
- It means: one class depends on another class in some way.
- It implies that a change to one class may affect the other but not vice versa.
- It describes a connection at a higher level of abstraction than an association.
- As it may have a broad meaning, it is best not to overuse the dependency relationship.

Dependency

- Graphically, a dependency is rendered as a dashed directed line, directed to the thing being depended on.

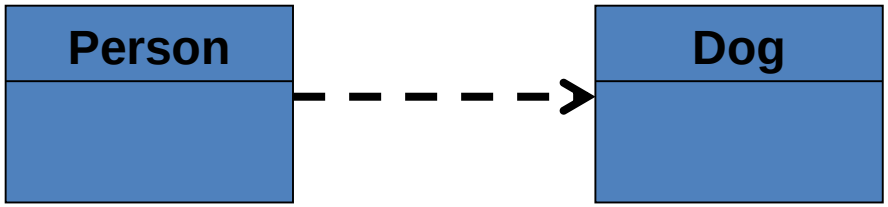
Dashed Arrow

- It means that;
  - A uses B, But A does not contain an instance of B as part of its own state.
  - If B's interface changes, it will likely to impact A and require it to change.

Dependency

- You may use dependency to indicate that,
  - "A" receives an instance of "B" as a parameter to at least one of its methods.
  - "A" creates an instance of "B" local to one of its methods.
- You would not use dependency to indicate that
  - "A" declares an instance variable of "B".
    use *association* to do that (covered next).

Dependency E.g.



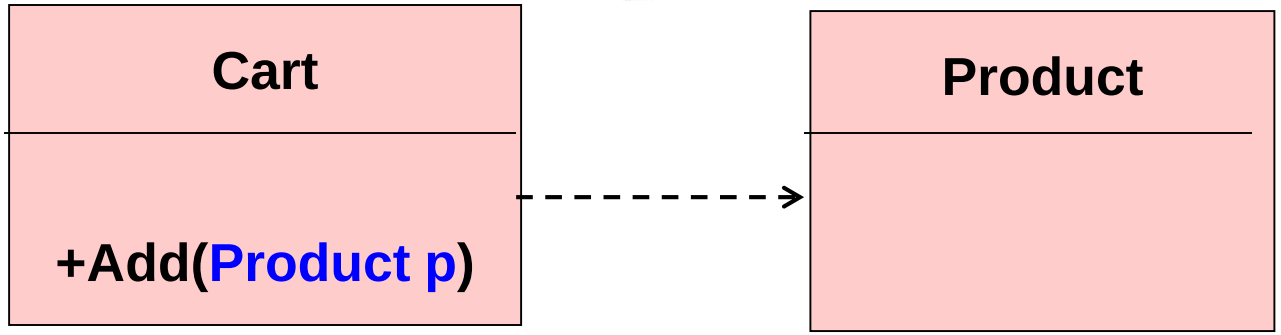
- The “Person” class depends on “Dog” class.
- Changes to “Dog” class may affect the “Person” class, but not vice versa.
- A “Person” object uses a “Dog” object somewhere as a parameter in one of its methods.

```
public class Person
  {
  public void walkDog ( Dog theDog );
  };
```

Dependency E.g.

dependant

independent



Cart class depends on a Product class because the Cart class uses the Product class as a parameter to Add() method.
Changing the Product Class will affect the Cart Class

2. Dependency

Association

- Association defines a much stronger dependency than that described before with the plain *dependency* relationship.

- Association connects one instance of a class with another instance of another class.

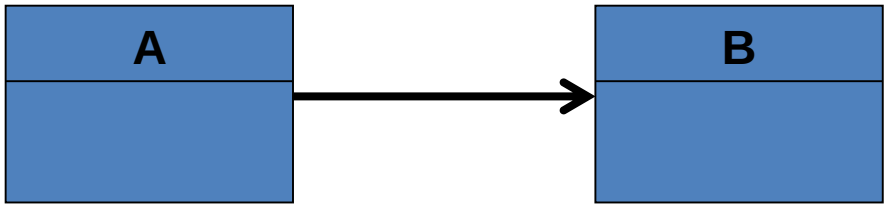
Association

- The relationship can be **bi-directional (two way)** or **uni-directional (one way)**

- If there is an arrowhead, it means there is a one-way relationship.

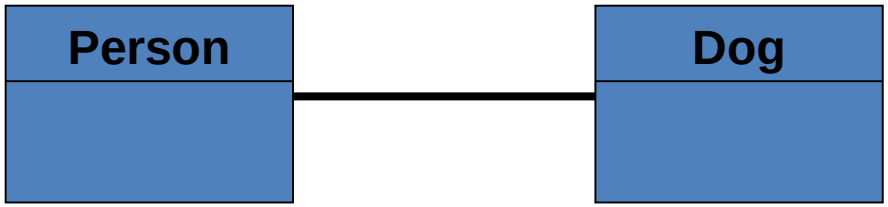
Simple Connected Line

Association (One Way)



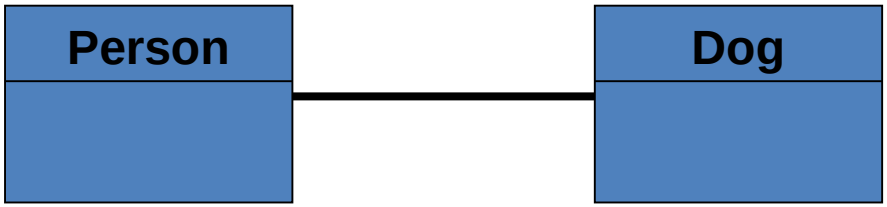
- Here, it means that;
  - Class A is associated with class B.
  - Class A uses and contains one instance of class B, but B does not know about or contain any instances of class A.
- In an association relationship, the dependent class (A) defines an instance of the associated class (B) within its class scope.

Association (Two Way)



- The Person class and the Dog class are associated.
- Here, it is bi-directional.
  - The person is related to the dog in some way.
  - The dog is also related to the person in some way.
  - The exact nature of the association is unknown.

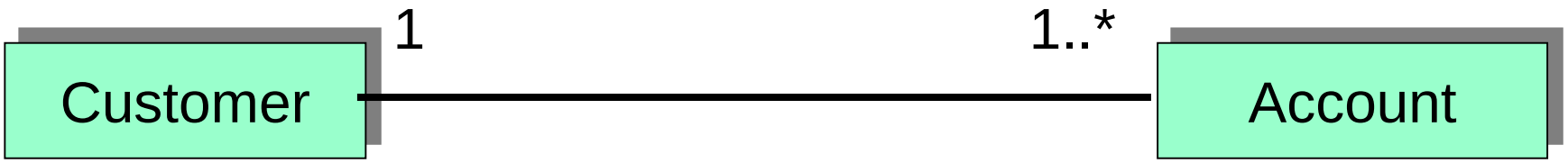
Association (Two Way) E.g.




```
Class person
{
Dog bruno;
Public walkDog();
}
```

```
Class Dog
{
Person owner;
}
```

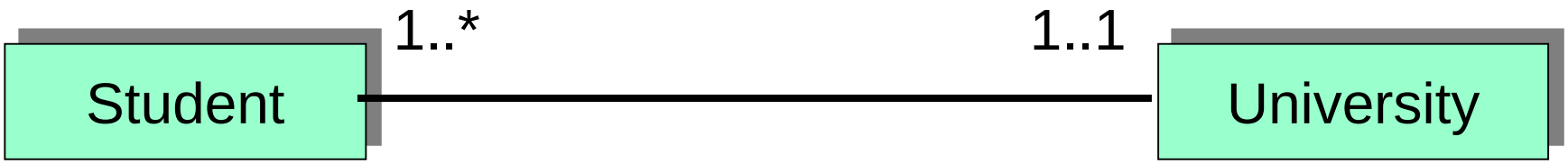
Association (Two Way) E.g.




```
Class Customer
{
Account myAcct1;
Account myAcct2;
}
```

```
Class Account
{
 Customer owner;
}
```

Association (Two Way) E.g.

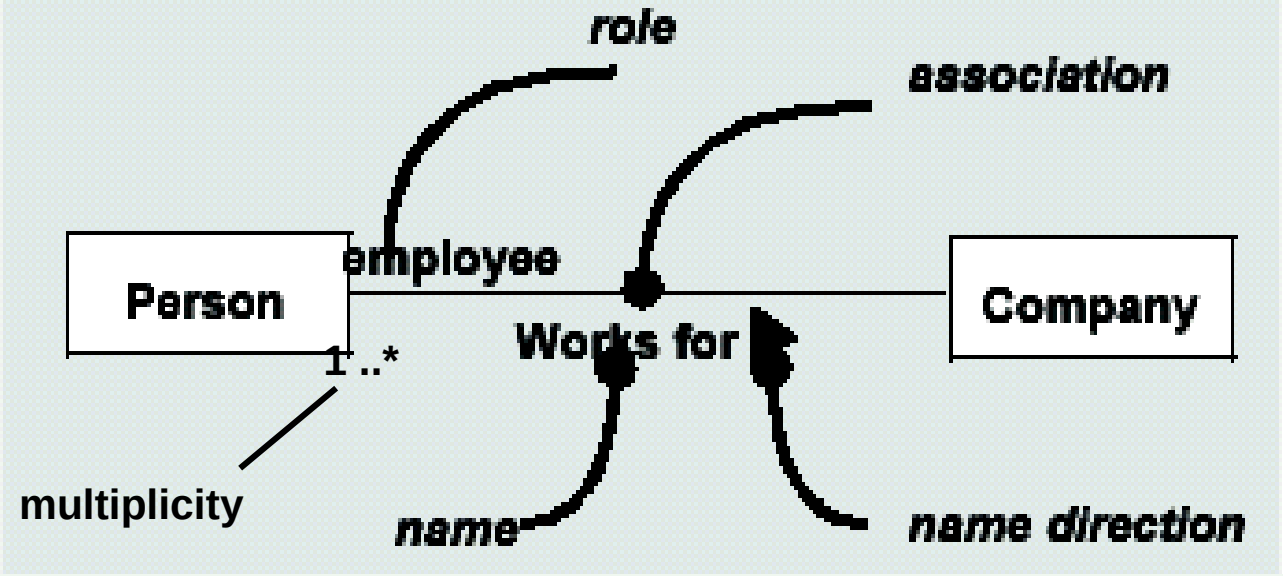



```
Class Student
{
University myUni;
}
```

```
Class University
{
Student[] students;
}
```

Association

- An association can be named, and the ends of an association can be adorned with role and multiplicity.

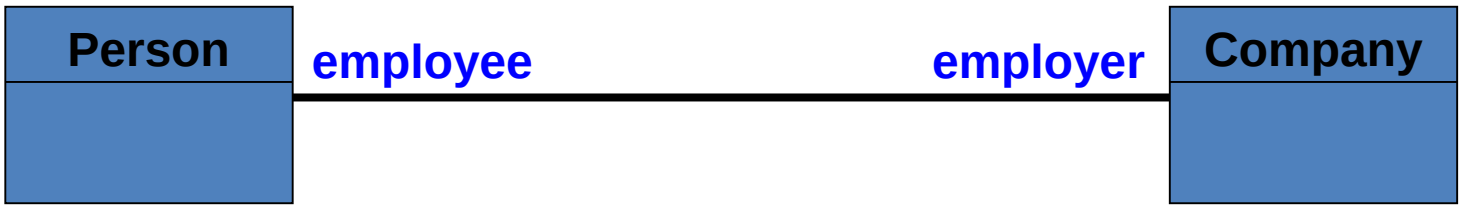


Association

- Association Name:
  - An association can be given a name:
    (This is only shown if it helps to clarify the association)
  - An association can be directed or not:

Association

- Association Role:
  - When a class participates in an association, it has a specific role that it plays in that relationship;
  - A role is just the face the class at the near end of the association presents to the class at the other end of the association.
  - A Person playing the role of employee is associated with a Company playing the role of employer.

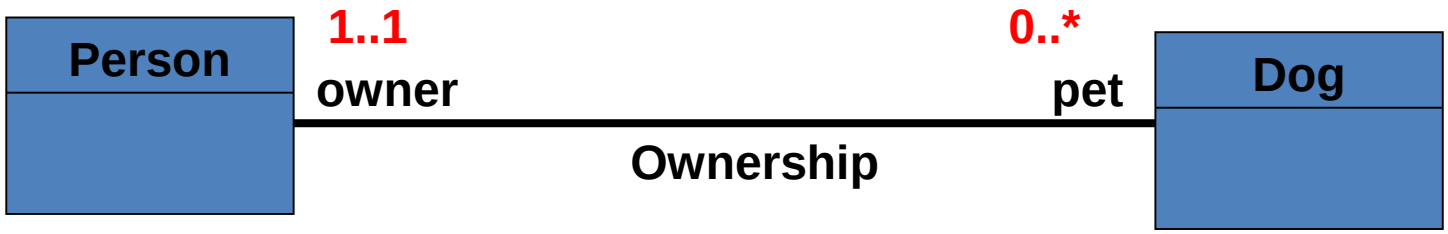


Association

- Association multiplicity :
  - Defines "How many objects of each class are associated in this relationship"
  - This "How many" is the multiplicity of an association's role.

1	Exactly one
n	Exactly n
0 .. 1	Zero or one
n .. m	Between n and m (inclusive)
0 .. *	Zero or more
1 .. *	One or more
0 .. 1, 3 .. 5	Zero or one, or between 3 and 5 inclusive.

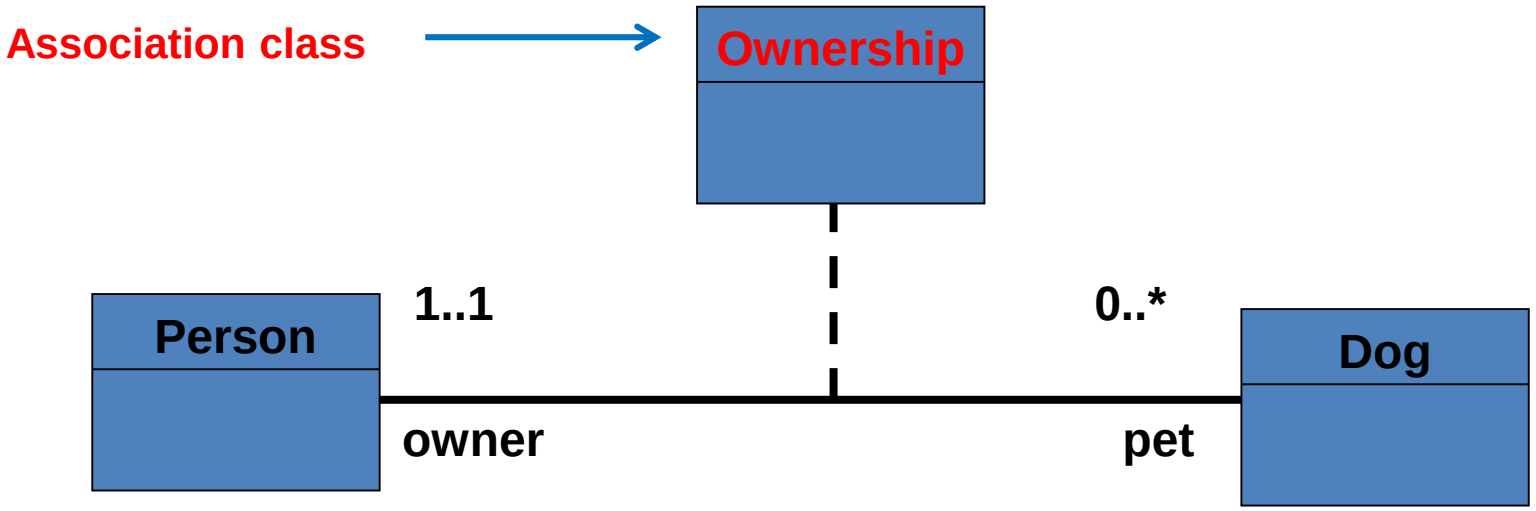
Association



- One Person object can have 0 or more Dog objects (0..*) associated with it.

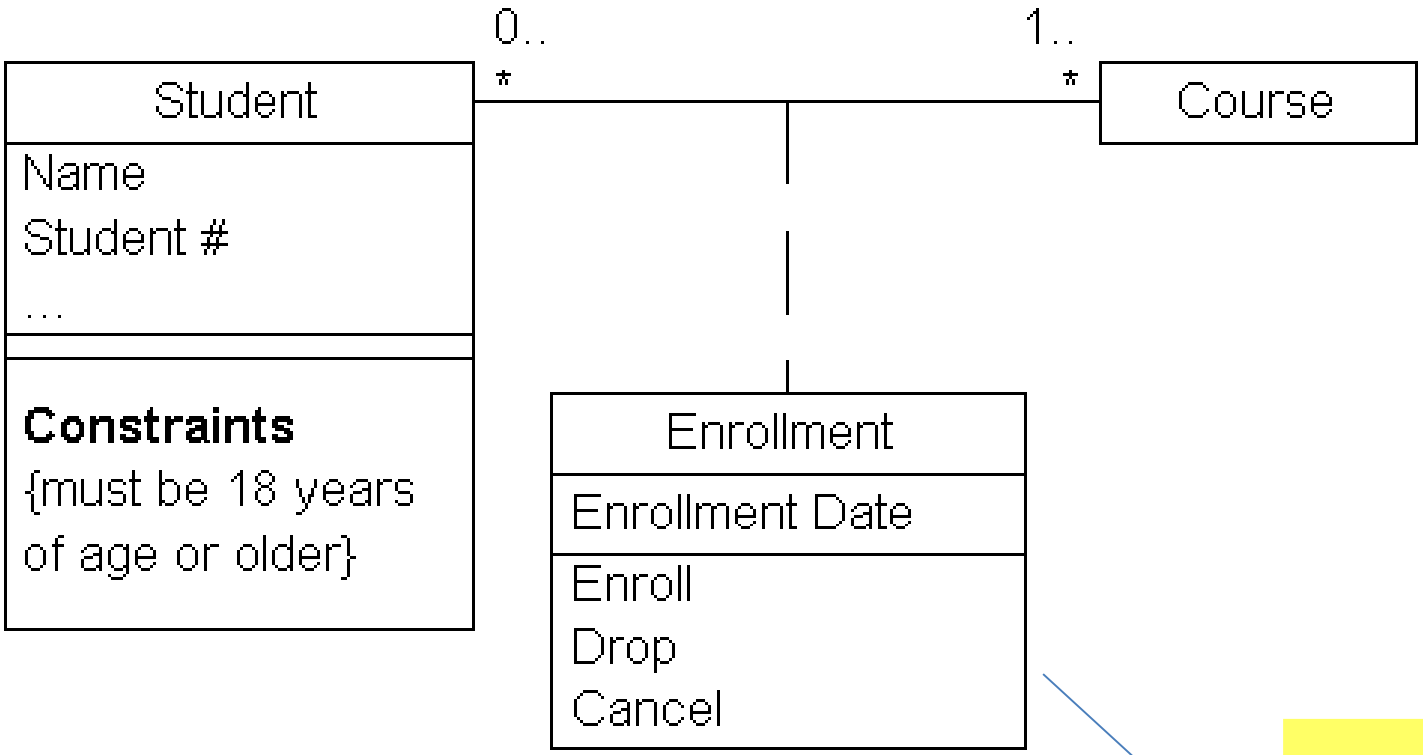
- Each Dog object is associated with exactly one Person object (1..1).

Association



- The association can also be promoted to a class.
- This places the responsibility for maintaining information pertaining to the association with the Ownership class.

Association



"Association Class"

has information related to the association

Association

- Promoting an association to a class allows:
  - The addition of attributes specific to the association into the new class.
    - this keeps the associated classes free of this extraneous information.
  - The addition of methods specific to the association into the new class.
    - this keeps the associated classes free of methods for maintaining the consistency of the association.

Association

Activity 01:

Draw a class diagram for the following description;

- Annie may work for a single Company.
- We need to keep information about the period of time that Annie works for the Company
- (Hint : try to come up with a association class)

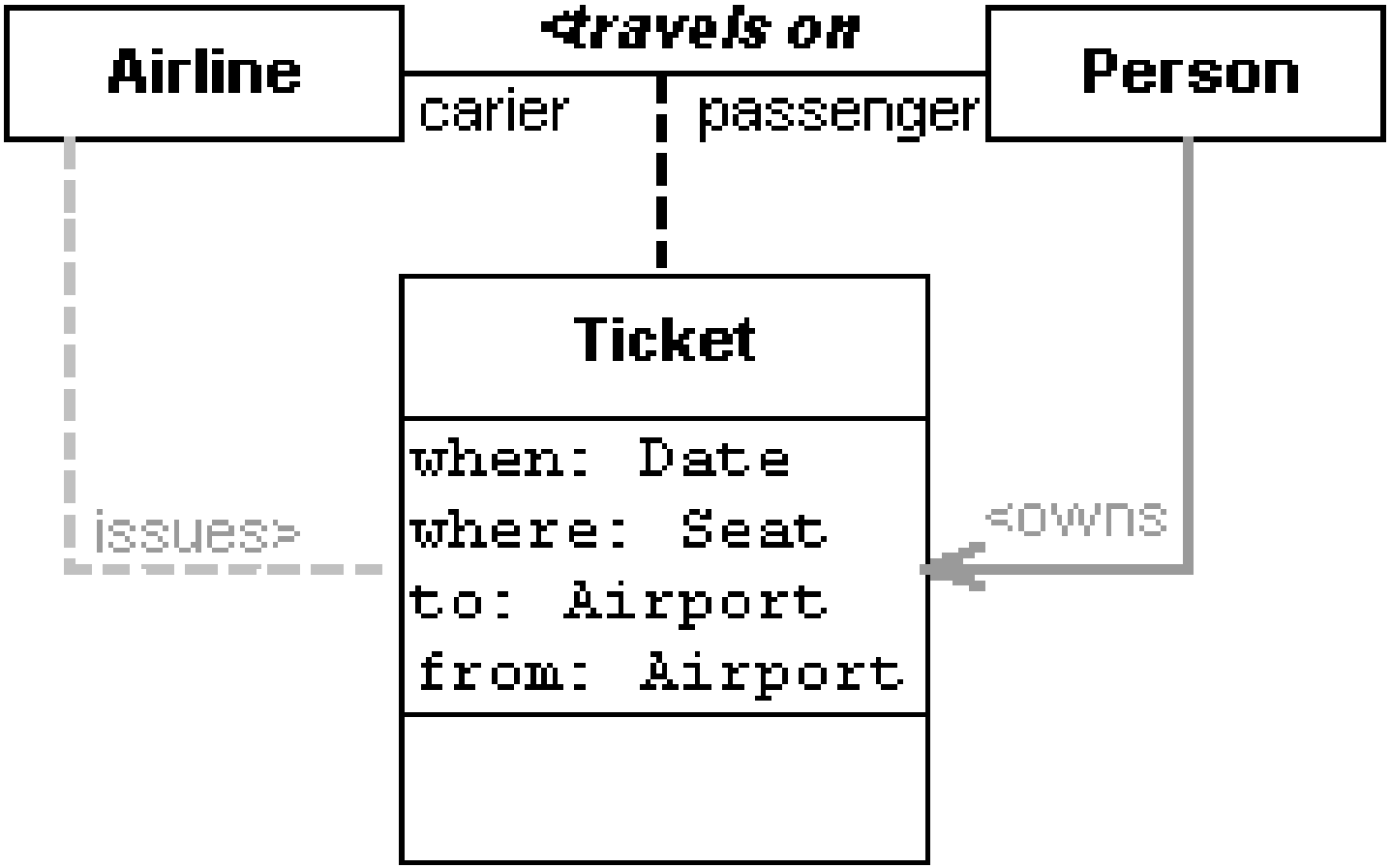
Association

Activity 02:

Improve the previous class diagram for the following conditions;

- Annie at the age of 23 joined Company A.
- She resigned from that Company A at the age of 25 and joined Company B.
- At the age of 29, Annie joined Company A again

Association



Whole-Part Relationships

- Aggregation
- Composition

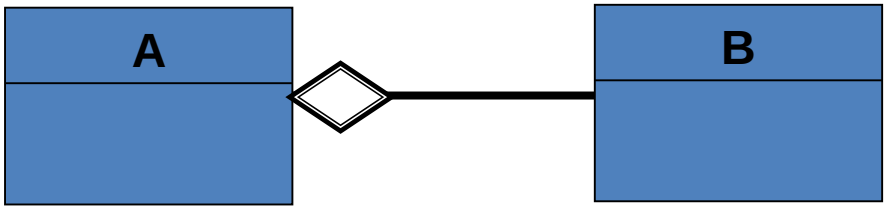
Whole-Part Relationships

- A "**whole/part**" relationship refers to a fairly strong connection between two classes.
- One class represents a larger thing ("whole"), which consists of smaller things ("parts").
- This means "**has a**" relationship. meaning that an object of the whole has objects of the part.
- Two types;
  - **Aggregation (Relatively Weak)**
  - **Composition (Relatively Strong)**

3. Aggregation

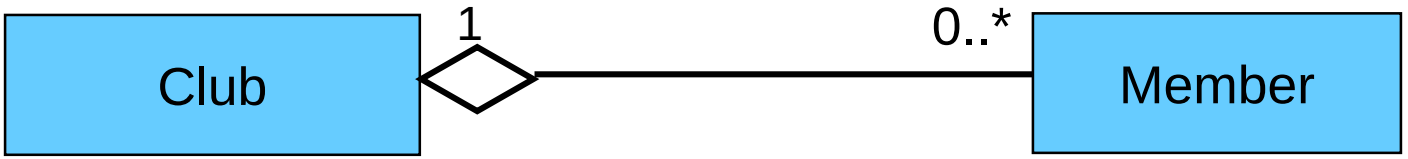
Aggregation

- Aggregation is just a special kind of association.
- Aggregation is a weak form of whole-part relationship
- Graphically, a dependency is rendered as an empty diamond arrow.
- This means that A aggregates B

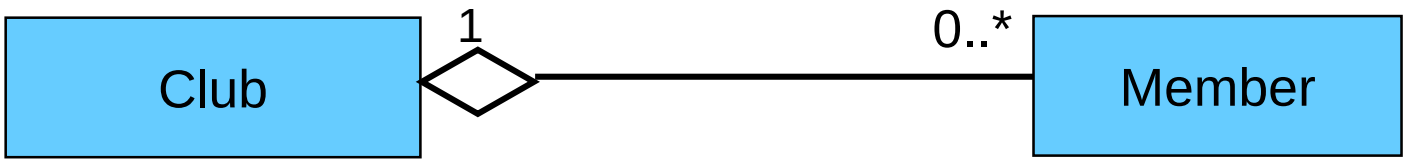


Aggregation

- Multiplicity:
  - The same rule for multiplicity can be applied for aggregation relationship.
  - Aggregation should have a relationship with a multiplicity of "**0..**"
  - In this case, every frequent flyer club consists of **zero or more members.**



Aggregation



- Whole : Club
- Part: Member
- A club consists of zero or more members.
- This Implies that the Club can exist without any Members.

- This Implies that the whole can exist without the parts.

Aggregation

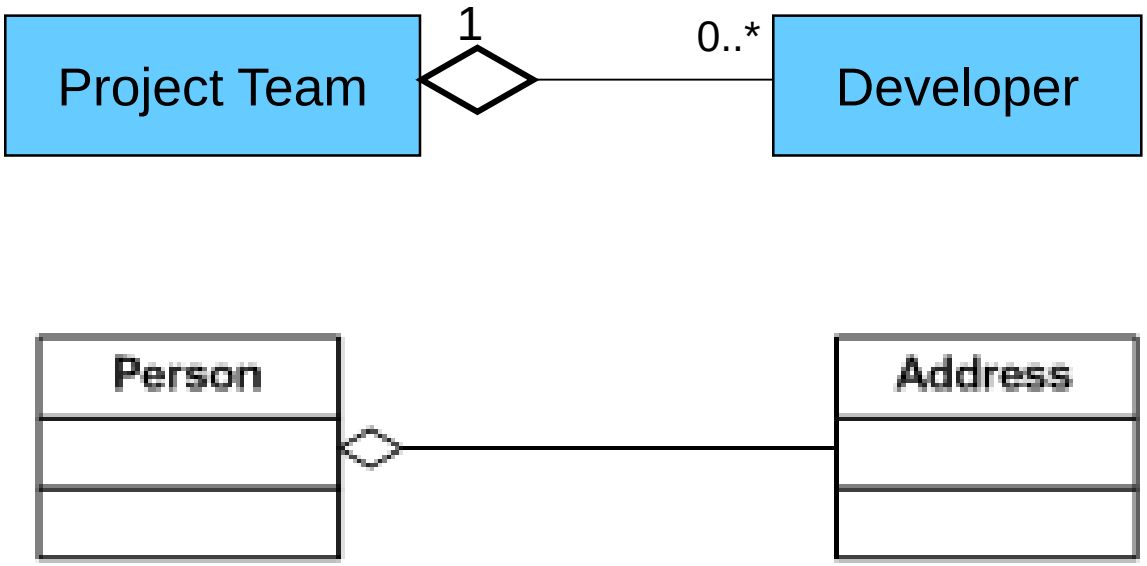


Figure 2 - Aggregation

Aggregation

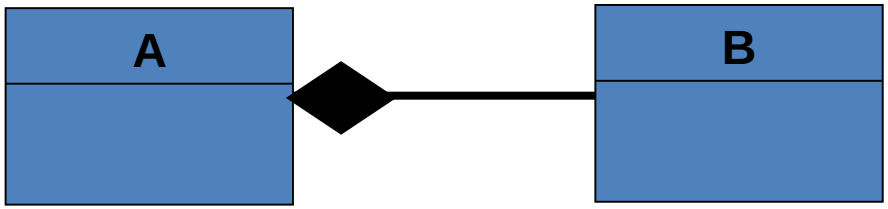
```
class Club {
   private Member member;

    public Club() {
       ..........
    }
    void setMember(Member mem) {
       this.member = mem;
    }
}
```

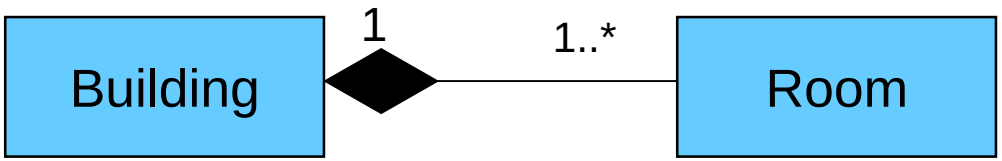
4. Composition

Composition

- Composition is a strong form of whole-part relationship.
- Graphically, a dependency is rendered as a filled diamond arrow.
- Composition should have a relationship with a multiplicity of "**1..**"



Composition



- Whole : Building
- Part : Room
- A Building is composed of at least one Room.
  - If there are no rooms, there is no Building.
- Implies that the "whole cannot exist without the parts".

Composition

```
Class Car {
   Private Engine engine;

     Public Car(EngineSpecs specs) {
        engine = new Engine(specs);
     }
     void move() {
        engine.work();
     }
}
```

Write the java code **assuming** Car and Engine has aggregation relationship

Aggregation

```
class Car {
   private Engine engine;
   public Car(){}
   void setEngine(Engine engine) {
         this.engine = engine;
   }
   void move() {
          if (engine != null)
                  engine.work();
   }
}
```

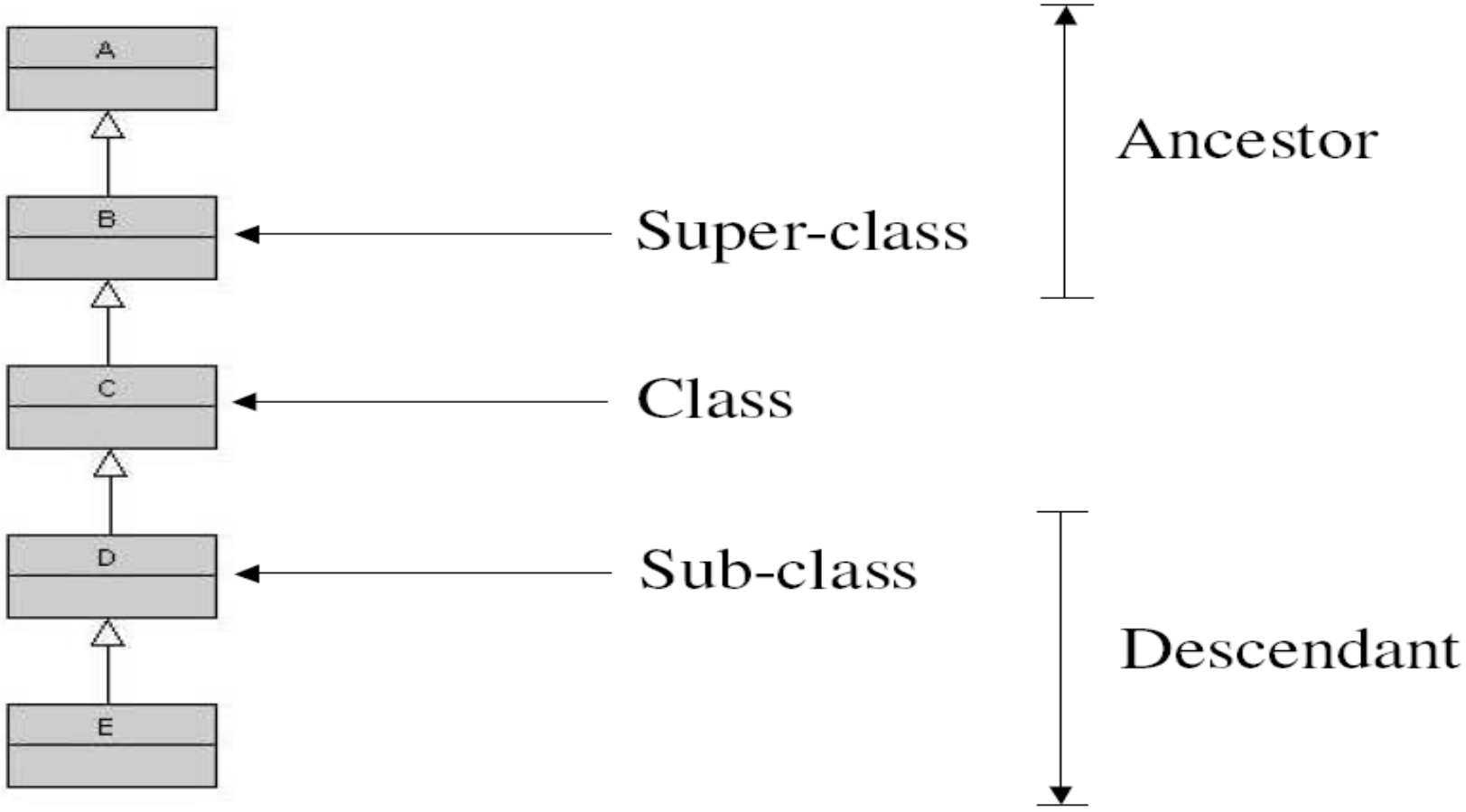
5. Inheritance

Inheritance

- Otherwise Known as Generalization.
- Inheritance represents a “is-a-kind-of" relationship.
- Inheritance is a relationship between a general thing (superclass / parent) and a more specific kind of that thing (subclass / child).
- Graphically, a dependency is rendered as a empty block arrow.

Empty block arrow

Inheritance



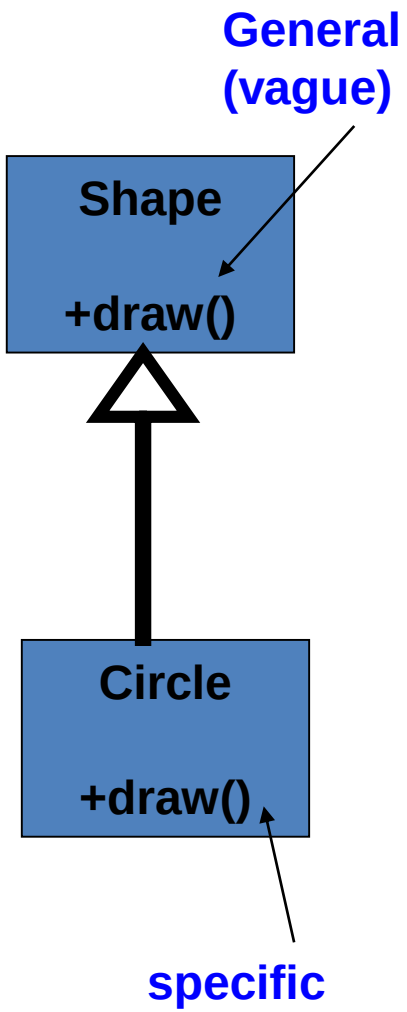
Inheritance

- Class A is a super-class (Parent) of class B if B directly inherits from A.
- Class B is a sub-class (Child) of class A if B directly inherits from A.

- Class A is an ancestor of class B if A is above B in the inheritance hierarchy.
- Class B is a descendant of class A if B is below A in the inheritance hierarchy.

Inheritance

- Generalization takes place from sub-class to super-class: the super-class is a **generalization** of the sub-class.
- Specialization takes place from super-class to sub-class: the sub-class is a **specialization** of the super-class.
- The functionality of the child should be a specialization of the functionality of the parent.
    - e.g. the functionality of the draw method in Circle is more specific than the one in Shape

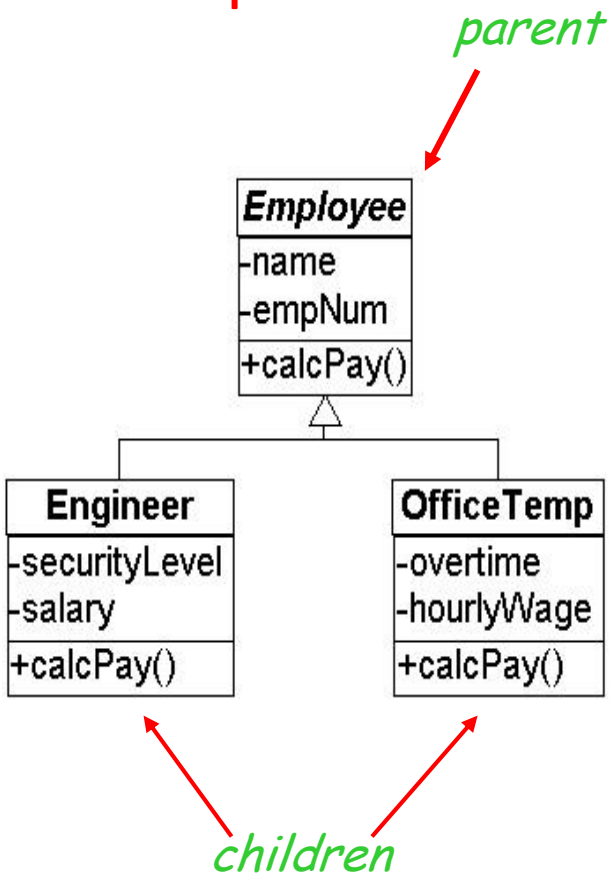


Inheritance

E.g: Go through the following example;

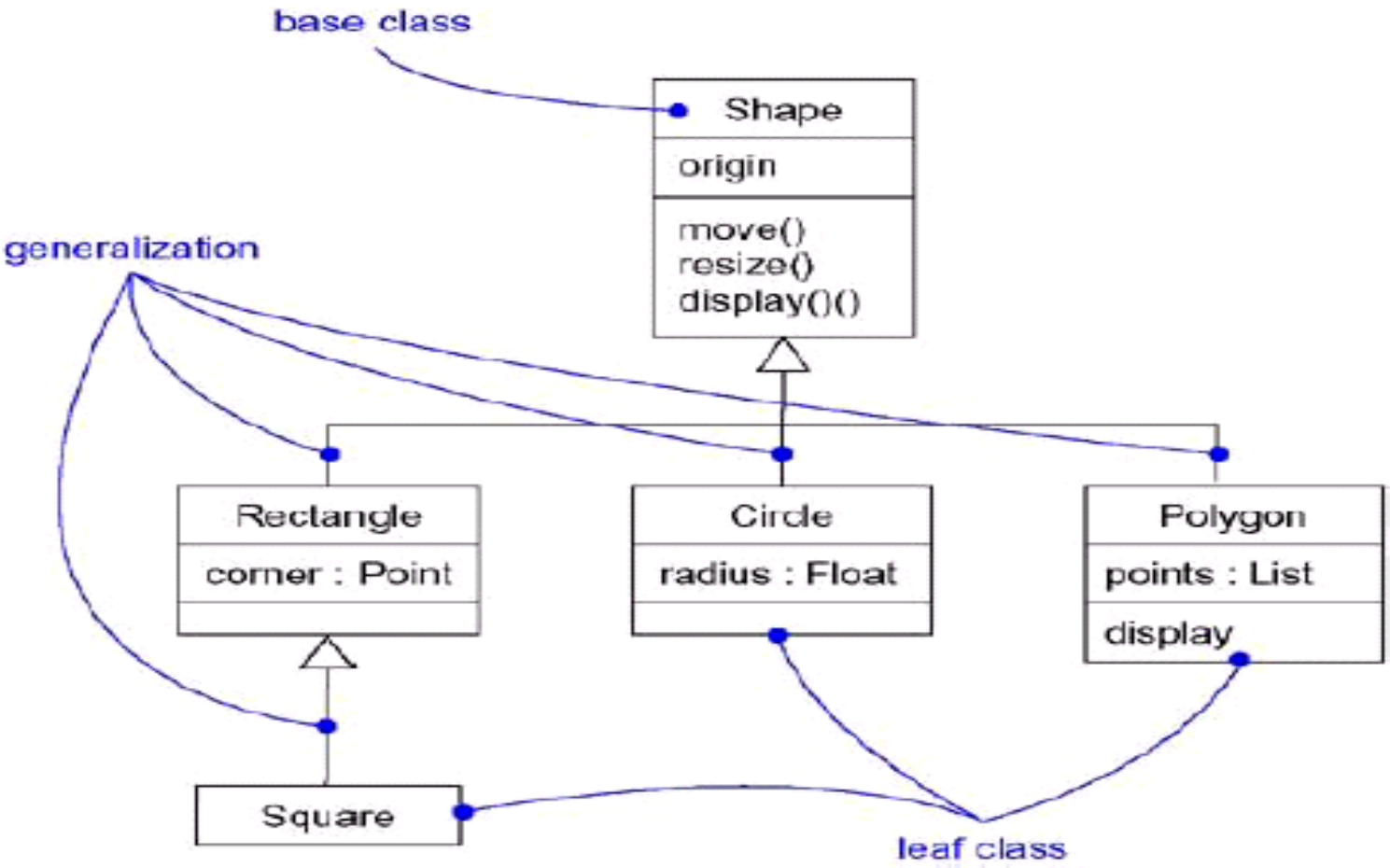
- Super Class / Parent: Employee
- Sub Classes / Children : Engineer, Office Temp

- Children (Engineer and Office Temp) inherit the properties of its parent's attributes, operations, responsibilities, etc.).



Inheritance

Example:



Inheritance - Activity

Draw a class diagram for the following football club. Try to identify different types of relationships between classes.

The football club has two grounds. Each ground consists of two or three pitches and a clubhouse.

The football club has lots of members. A member may be a playing member or a social member, and a playing member may be an adult member or a junior member.

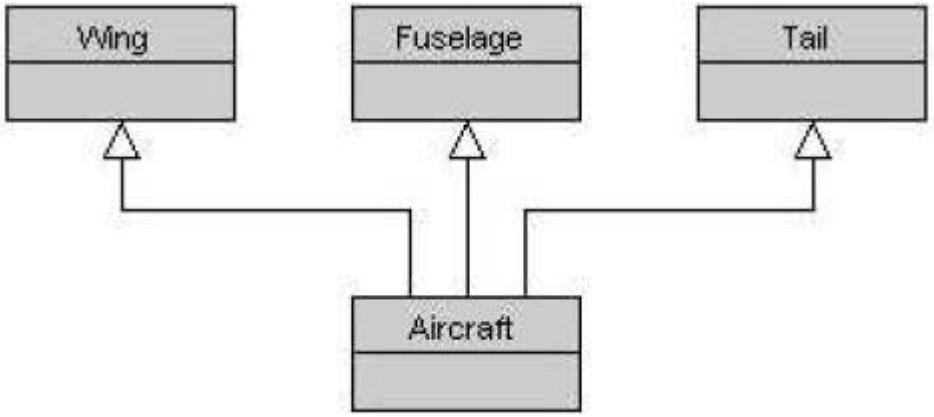
A playing member may be chosen to play for one or more teams, and each team has 11 playing members. Each team plays a number of matches.

Inheritance

- Inheritance is one of the defining concepts of object orientation:
  - it is a powerful technique for organizing and reusing design concepts in an abstracted hierarchy;
  - without inheritance, polymorphism would not be possible!
  - Unfortunately, inheritance is also the most misused and misunderstood relationship:

Misuse of Inheritance

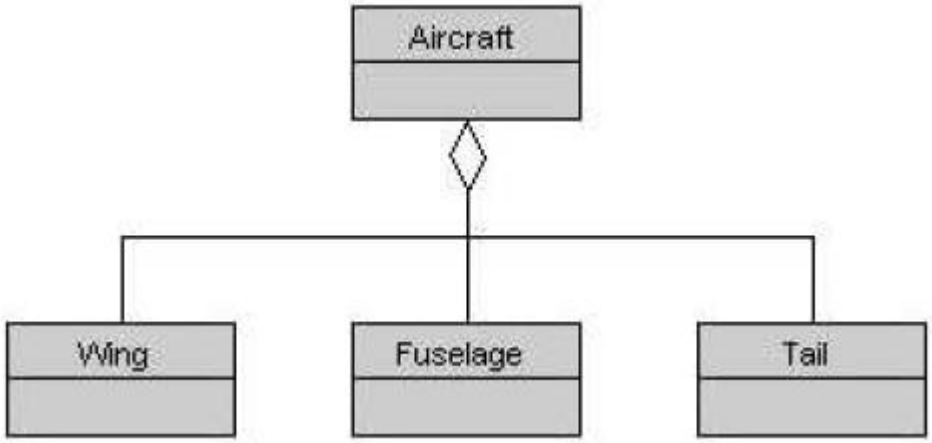
- **Inheritance in place of aggregation:**
  - **using "is a" instead of "has a".**



- The idea is that an aircraft has all the functionality of its component parts:
  - a plane can do all the things a wing, fuselage and tail can do (and more).

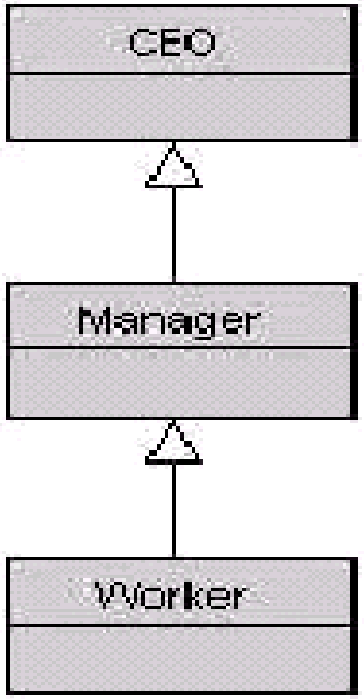
Misuse of Inheritance

- The problem is that an aircraft is **not** a wing:
  - the class hierarchy implies that an aircraft can be used wherever a wing is required.
- A better way to represent the relationship is through aggregation or composition.



Misuse of Inheritance

- **Inverted hierarchy:**
  - the design (right) represents a hierarchy in the real world.
  - the idea is that capturing the real-world hierarchy via inheritance relationships will capture real-world behavior.



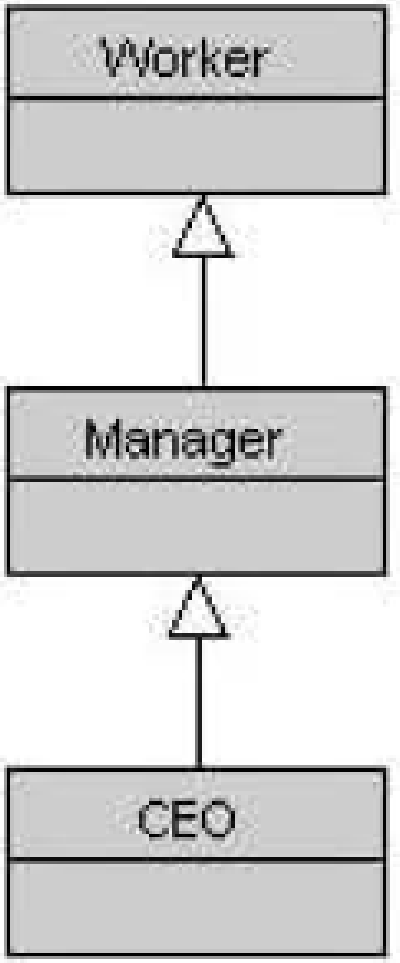
Misuse of Inheritance

- In OO design:
  - the child class has all the attributes and methods of the parent (and more);
  - the functionality of the child class should be more specialized than that of the parent.

- This implies that:
  - a Worker can do everything a CEO can do;
  - a Worker is also “better” at doing these things.

Misuse of Inheritance

- The solution may be simply to reverse the direction of inheritance:
  - In the real-world, the higher entity in the hierarchy has the most power.
  - In object oriented design, the higher entity in the hierarchy has the least power.



References

- UML Distilled by Martin Fowler, chapters 3 and 5
- Fundamentals of Object- Oriented Design in UML by Page-Jones, M (2000). Chapters 4 & 12.