Association, aggregation, and composition in OOP explained

MICROSOFT ARCHITECT

By Joydip Kanjilal, Columnist, InfoWorld | NOV 19, 2018 3:00 AM PST

05506004

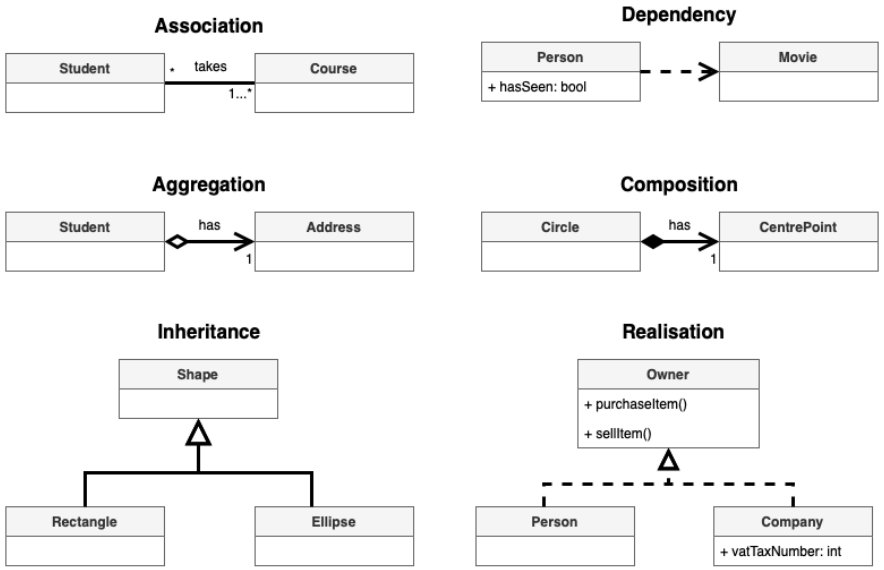
https://www.infoworld.com/article/3029325/exploring-association-aggregation-and-composition-in-oop.html

Introduction (Association)

Association in object oriented programming

Association is a semantically weak relationship (a semantic dependency) between otherwise unrelated objects. An association is a "using" relationship between two or more objects in which the objects have their own lifetime and there is no owner.

As an example, imagine the relationship between a doctor and a patient. A doctor can be associated with multiple patients. At the same time, one patient can visit multiple doctors for treatment or consultation. Each of these objects has its own life cycle and there is no "owner" or parent. The objects that are part of the association relationship can be created and destroyed independently.



https://www.diagrams.net/blog/uml-class-diagrams

Introduction (Aggregation)

Aggregation in object oriented programming

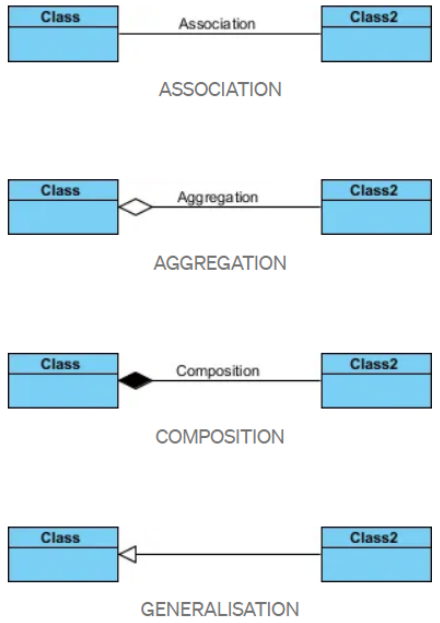
Aggregation is a specialized form of association between two or more objects in which each object has its own life cycle but there exists an ownership as well. Aggregation is a typical whole/part or parent/child relationship but it may or may not denote physical containment. An essential property of an aggregation relationship is that the whole or parent (i.e. the owner) can exist without the part or child and vice versa.

As an example, an employee may belong to one or more departments in an organization. However, if an employee's department is deleted, the employee object would not be destroyed but would live on. Note that the relationships between objects participating in an aggregation cannot be reciprocal—i.e., a department may "own" an employee, but the employee does not own the department.

Aggregation is usually represented in UML using a line with a hollow diamond.

UML Summary:

Below figure shows different types of association connector notation. We shall go over them one by one.



https://blog.devgenius.io/association-aggregation-composition-pattern-w-uml-5c68956fd689

Introduction (Composition)

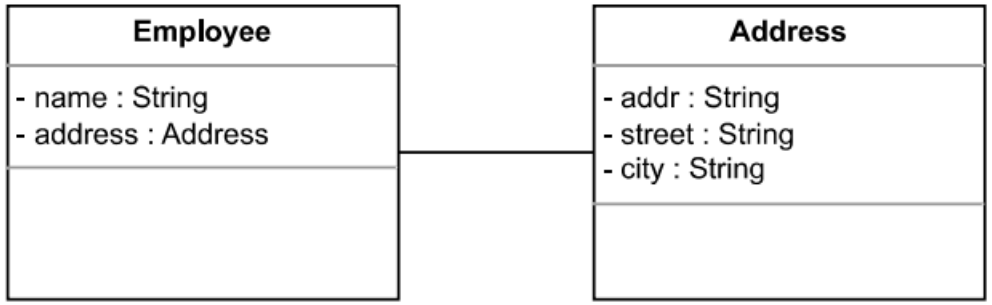
Composition in object oriented programming

Composition is a specialized form of aggregation. In composition, if the parent object is destroyed, then the child objects also cease to exist. Composition is actually a strong type of aggregation and is sometimes referred to as a "death" relationship. As an example, a house may be composed of one or more rooms. If the house is destroyed, then all of the rooms that are part of the house are also destroyed. The following code snippet illustrates a composition relationship between two classes, House and Room.

```
public class House
{
    private Room room;
    public House()
    {
        room = new Room();
    }
}
```

Like aggregation, composition is also a whole/part or parent/child relationship. However, in composition the life cycle of the part or child is controlled by the whole or parent that owns it. It should be noted that this control can either be direct or transitive. That is, the parent may be directly responsible for the creation or destruction of the child or the parent may use a child that has been already created. Similarly, a parent object might delegate the control to some other parent to destroy the child object. Composition is represented in UML using a line connecting the objects with a solid diamond at the end of the object that owns the other object.

Employee – Address




```java
public class Employee {
  private String name;
  private Address address;

  public Employee(String n, Address a) {
      name = n; address = a;
  }

  @Override
  public String toString() {
      return name + " " + address.toString();
  }
}
```

```java
class Address {
  private String addr;
  private String street;
  private String city;

  Address(String a, String s, String c) {
    addr = a;  street = s;  city = c;
  }

  @Override
  public String toString() {
    return "Address [" + addr + " "
          + street + " " + city + "]";
  }
}
```

```java
public static void main(String[] args) {
    Address home1 = new Address(a: "homey", s: "sook", c: "BKK");
    Address home2 = new Address(a: "lieDownAllDay", s: "smile", c: "Lampang");

    Employee yindee = new Employee(n: "Yindee", home1);
    Employee preeda = new Employee(n: "Preeda", home2);

    System.out.println(yindee);
    // Yindee Address [homey sook BKK]

    System.out.println(preeda);
    // Preeda Address [lieDownAllDay smile Lampang]
}
```

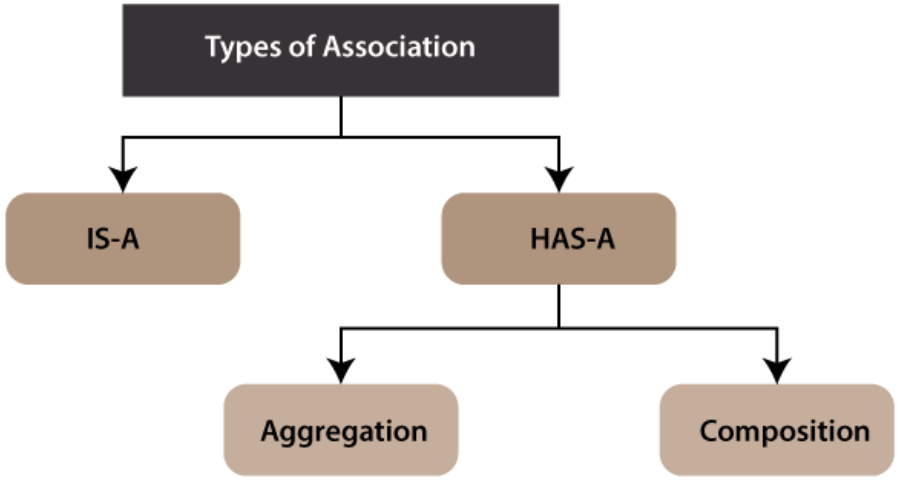
Types of Association

Association - Owner love pets, owner feed pets; in return pets pleases and loves owner

Aggregation / Composition - Tails or legs are part of Dog & Cat class objects.

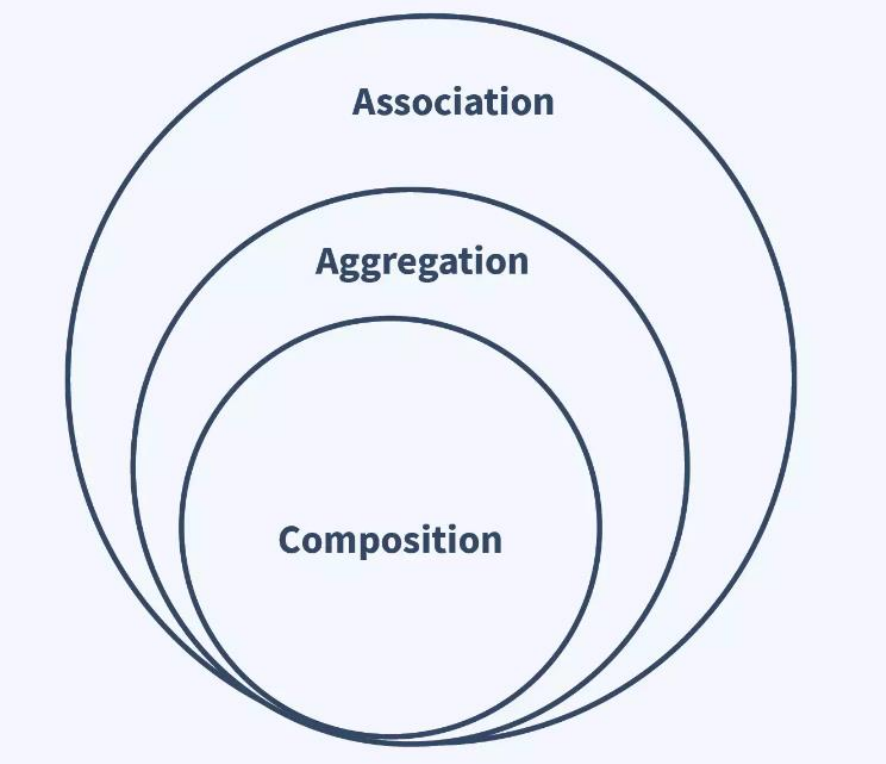
Inheritance / Generalisation - Cat is-a Pet, Dog is-a Pet

https://blog.devgenius.io/association-aggregation-composition-pattern-w-uml-5c68956fd689



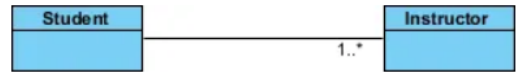
https://www.javatpoint.com/association-in-java

Types of Association
(Association) multiplicity / cardinality

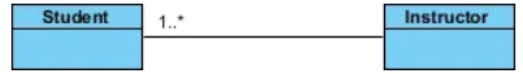


https://www.scaler.com/topics/association-composition-and-aggregation-in-java/

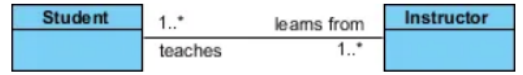
We can indicate the multiplicity/cardinality of an association by adding multiplicity adornments to the line denoting the association. The example below indicates that a Student has one or more Instructors:



each Student can be associated with many Instructor



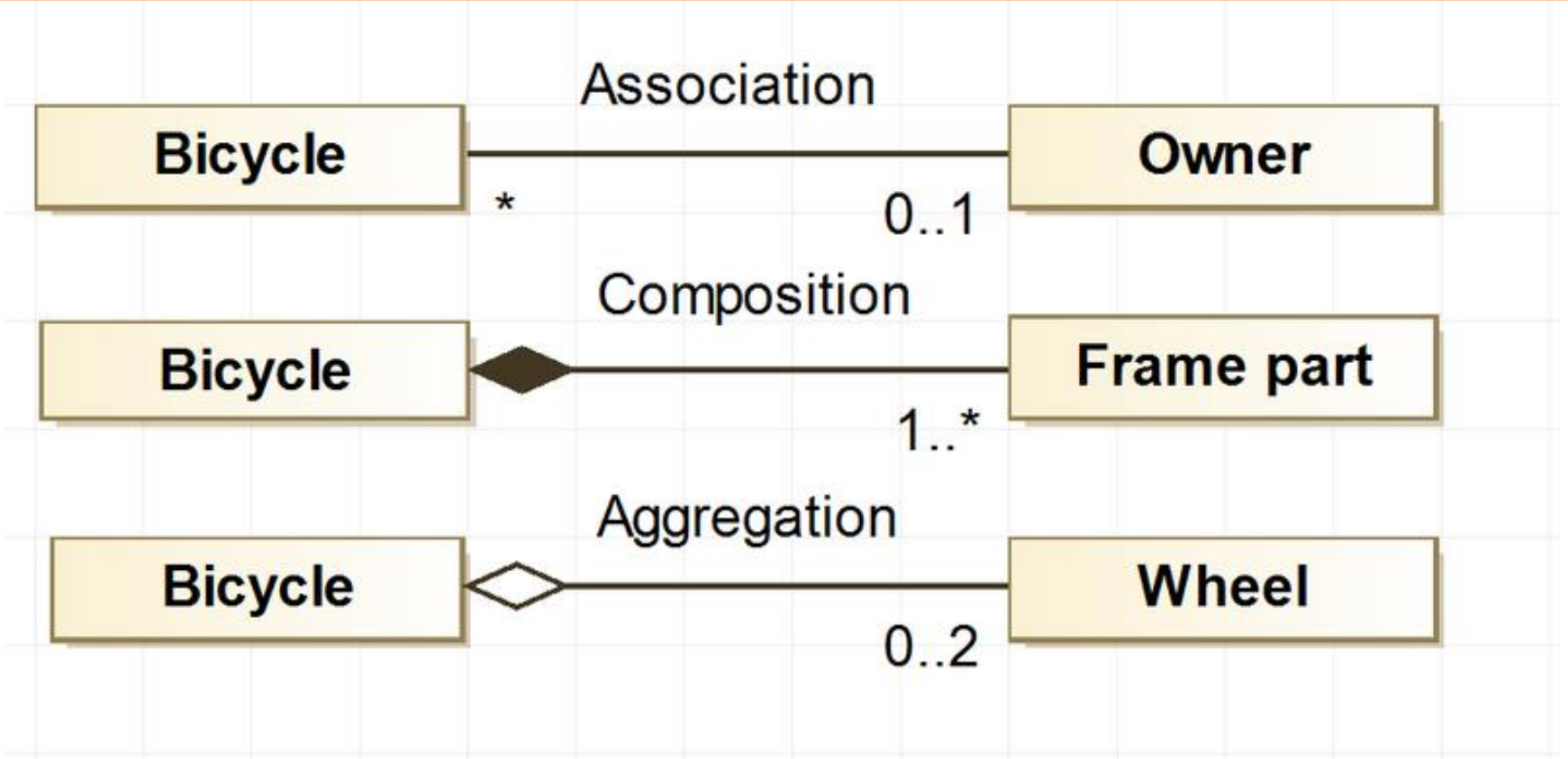
each Instructor has one or more Student



behaviour of objects in an association using role names

https://blog.devgenius.io/association-aggregation-composition-pattern-w-uml-5c68956fd689

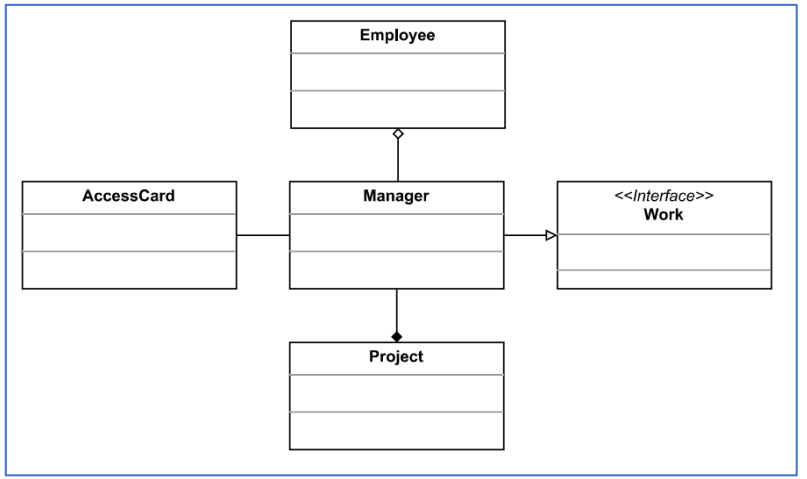
Types of Association



https://en.m.wikipedia.org/wiki/File:UML_association,_aggregation_and_composition_examples_for_a_bicycle.png

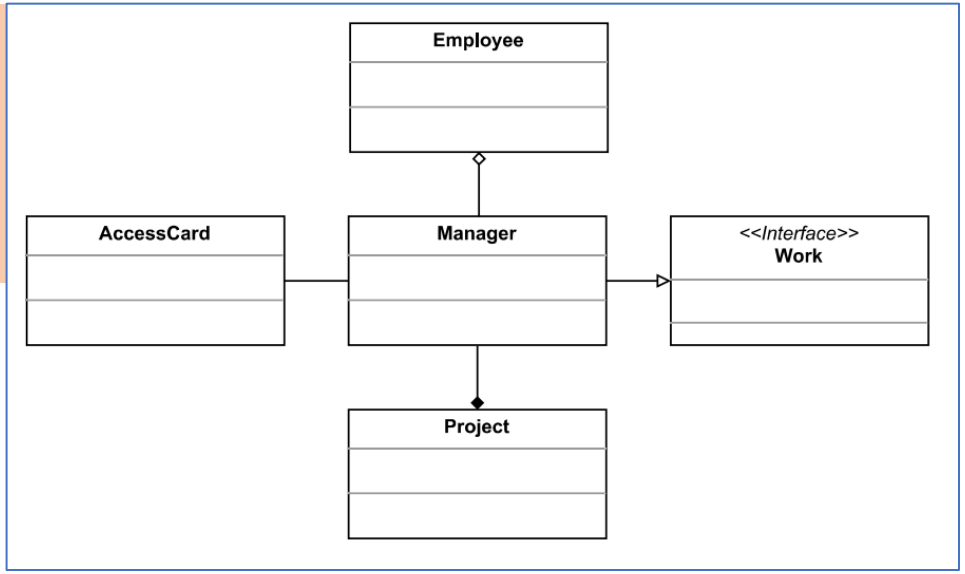
Example

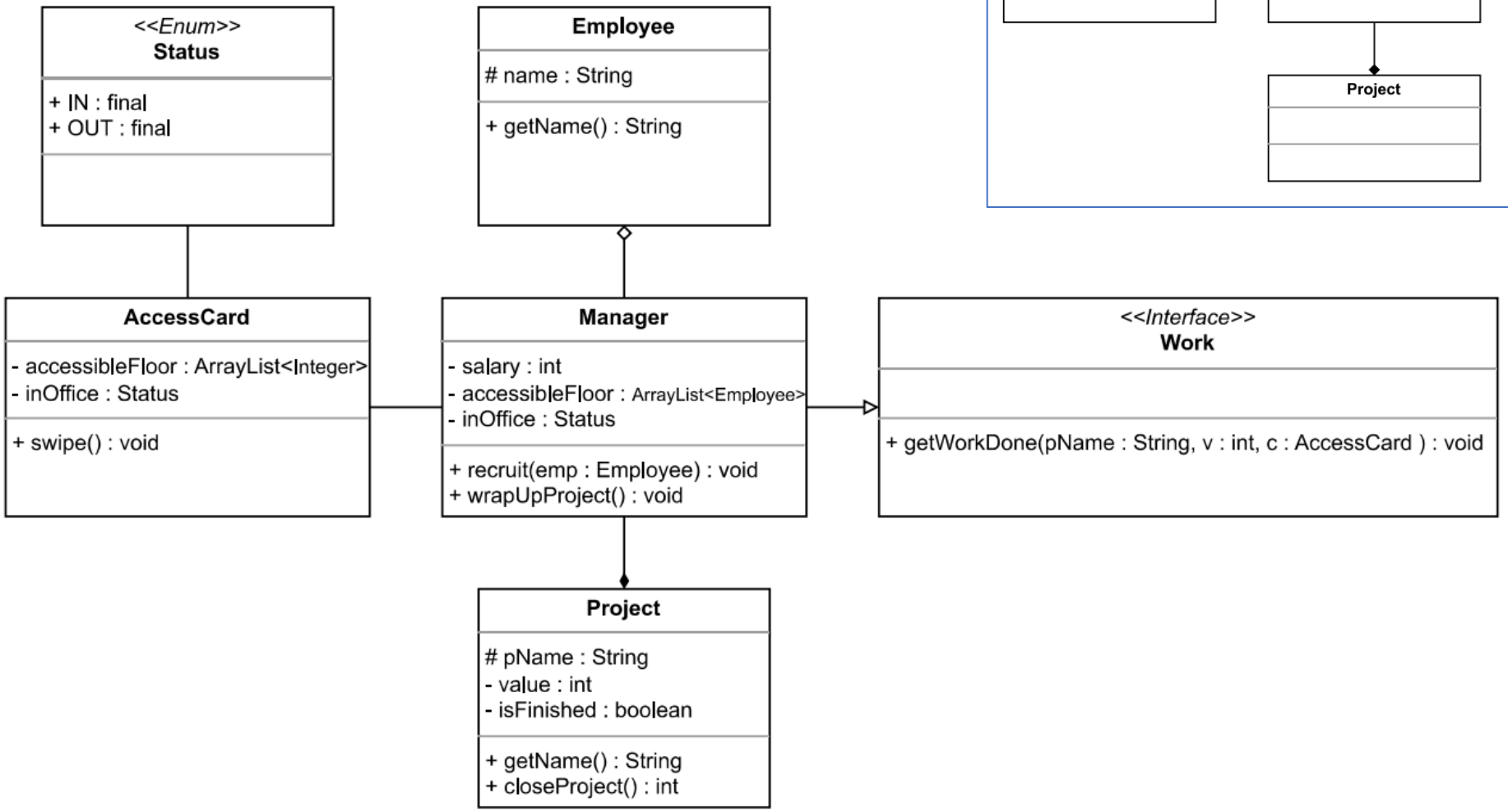
- A manager is a type of employee. -> inheritance
- Manager has a access card to enter company premises. -> association
  - Manager life-time is independent to swipe card.
- Manager has many workers work under him. -> aggregation
  - Worker is part of the manager team. (independent)
- Manager salary depends on project success. (Project success depends on the manager skills.) -> composition
  - Not a good example …cannot able convey the key idea that its life time depends on each other.
  - base + project value



https://youtu.be/0po_wmSEW1Q

Manager.java





```
public class AccessCard {
  ArrayList<Integer> accessibleFloor
     = new ArrayList<>(List.of(3, 4));
  Status inOffice = Status.OUT;

  public void swipe() {
    inOffice =
inOffice == Status.IN ? Status.OUT : Status.IN;
    System.out.println("logging " + inOffice);
  }
}
enum Status {
    IN, OUT
}
```

```
public class Project {
    private String pName;
    private int value;
    private boolean isFinished;

    public Project(String n, int v) {
        pName = n;  value = v;
        isFinished = false;
    }
    public int closeProject() {
        isFinished = true;
        return value;
    }
    public String getName() { return pName; }
}
```

```
public class Employee {
   protected String name;

   // private Manager boss;
   public Employee(String n) {  name = n;  }

   public String getName() {  return name; }
}
```

Manager.java

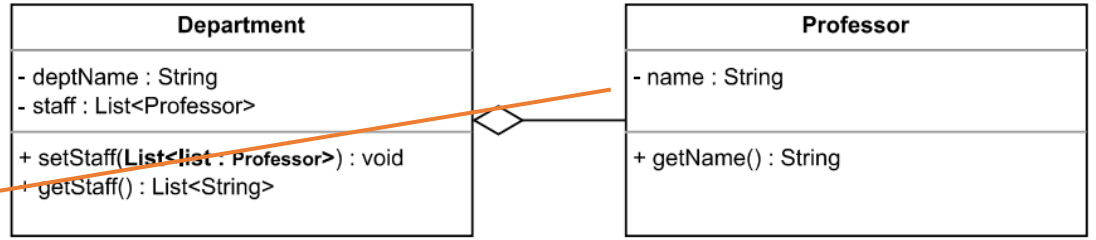
```
public class Manager extends Employee
                        implements Work {
  private int salary;
  private ArrayList<Employee> supervisee;
  private Project currentProject;
  private AccessCard officeAccessCard;

  public Manager(String n) {
      super(n);
      int base = 1000;
      salary = base;
      supervisee = new ArrayList<>();
  }

  public void recruit(Employee emp) {
      supervisee.add(emp); // aggregation
  }

  public void getWorkDone(String projectName,
                          int v, AccessCard card) {
    officeAccessCard = card;
    officeAccessCard.swipe();
    currentProject = new Project(projectName, v);
                          // composition
    for (Employee emp : supervisee)
        println("working with " + emp.name);
  }

  public void wrapUpProject() {
    officeAccessCard.swipe();
    officeAccessCard = null;
    salary += currentProject.closeProject();
    print(currentProject.getName()
                        + " is finished. ");
    println(name + " got paid for " + salary);
  }
}
```

main(String [] args)

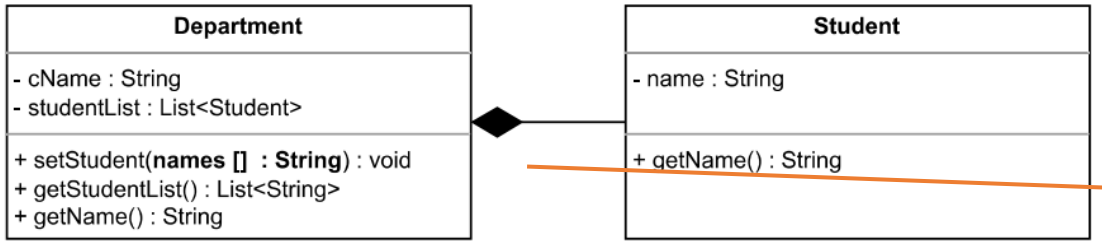
```
public static void main(String[] args) {
    AccessCard onlyCardinCompany = new AccessCard();
    Employee yindee = new Employee("Yindee");
    Employee preeda = new Employee("Preeda");
    Manager haha = new Manager("haha ^.^");
    haha.recruit(yindee);
    haha.recruit(preeda);
    haha.getWorkDone("break-through-app", 500, onlyCardinCompany);
    haha.wrapUpProject();
    System.out.println("--------------");

    Manager suesue = new Manager("Ganbatte");
    suesue.recruit(yindee);
    suesue.recruit(preeda);
    suesue.getWorkDone("another-project", 600, onlyCardinCompany);
    suesue.wrapUpProject();
}
```

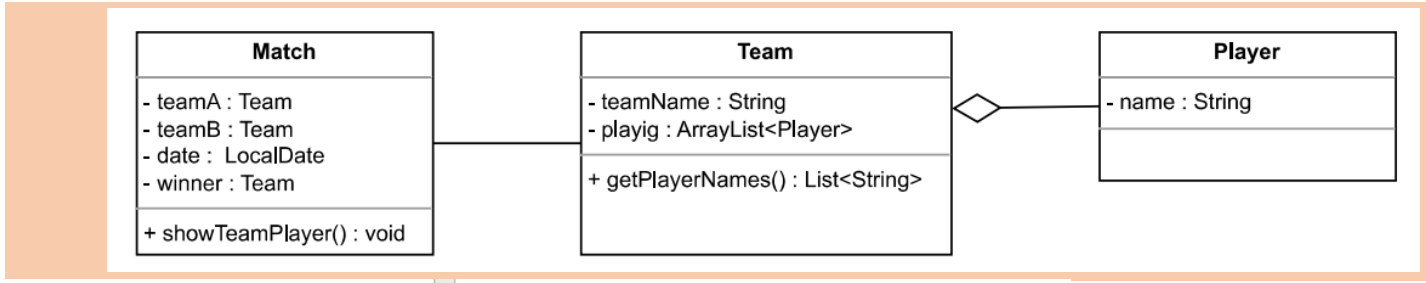
(more) Aggregation / Association

```java
public static void main(String[] args) {
    Professor yindee = new Professor(n: "Yindee");
    Professor preeda = new Professor(n: "Preeda");
    Department happyCS = new Department(dName: "happyCS");

    List<Professor> happyCS_staff = new ArrayList<>();
    happyCS_staff.add(yindee);
    happyCS_staff.add(preeda);

    happyCS.setStaff(happyCS_staff);
    System.out.println("our staff " + happyCS.getStaff());
}
```




```java
public class MainCollegeStudent {
    Run | Debug
    public static void main(String[] args) {
        College col1 = new College(name: "infinite");
        col1.setStudent(...names: "tep moo", "tep ma", "tep ga", "tep gai");
        System.out.println("Teps are " + col1.getStudentList());
    }
}

class College {
    private String cName;
    private ArrayList<Student> studentList;
    void setStudent(String... names) {
        studentList = new ArrayList<>();
        for (String name : names) {
            studentList.add(new Student(name));
        }
    }
```





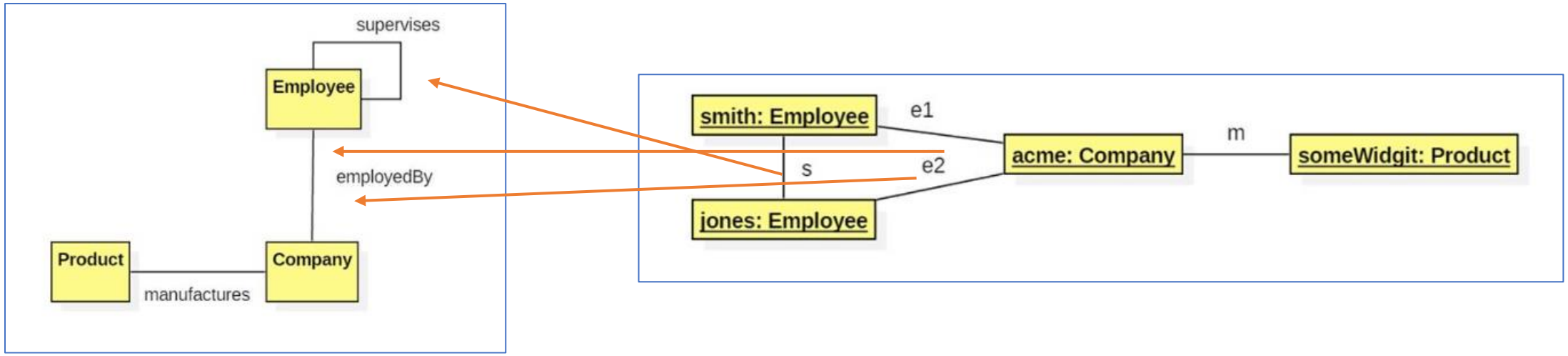

```java
public class Match {
    private MatchTeam teamA;
    private MatchTeam teamB;
    private LocalDate date;
    private MatchTeam winner;
    public Match(MatchTeam a, MatchTeam b,
                 int yy, int mm, int dd) {
        teamA = a;  teamB = b;
        date = LocalDate.of(yy, mm, dd);
    }
    public void showTeamPlayer() {
        System.out.println(x: "Team A");
        for (String s : teamA.getPlayerNames())
            System.out.println(s);
        System.out.println(x: "Team B");
        for (String s : teamB.getPlayerNames())
            System.out.println(s);
    }
}
```

```java
public class MatchPlayer {
    private String name;
    public MatchPlayer(String n) {
        name = n;
    }
    public String getName() {
        return name;
    }
}
```

```java
public class MatchTeam {
    private String tName;
    private ArrayList<MatchPlayer> playingToday;

    public MatchTeam(String n,
                     ArrayList<MatchPlayer> players) {
        tName = n;
        playingToday = players;
    }
    public List<String> getPlayerNames() {
        ArrayList<String> result = new ArrayList<>();
        for (MatchPlayer p : playingToday)
            result.add(p.getName());
        return result;
    }
}
```

Another Example

- Associations represent domain-significant relationships—Employee X supervises Employee Y, Employee X is employed by Company Z, Company Z manufactures Product P, etc.



http://www.cs.sjsu.edu/~pearce/modules/lectures/uml2/relationships/Association.htm

Employee

```
public class Employee {
    private String name;
    private Employee supervisor = null;
    private ArrayList<String> supervisee
                = new ArrayList<>();

    public Employee(String n) {
        name = n;
    }

    public void setSupervisee(Employee emp) {
        supervisee.add(emp.getName());
        emp.setSupervisor(this);
    }

    private void setSupervisor(Employee sup) {
        this.supervisor = sup;
    }

    public String getName() {
        return name;
    }

    public void showSupervisee() {
        if (!supervisee.isEmpty()) {
            System.out.println(getName()
                        + " supervises ");
            for (String s : supervisee)
                System.out.println(s);
        }
    }

    public String getSupervisorName() {
        return supervisor.getName();
    }
}
```

Company

```
public class Company {
    private String title;
    private ArrayList<Employee> empList
                        = new ArrayList<>();
    private Product specialty;

    public void setTitle(String t) {
        title = t;
    }

    public void setSpecialty(Product p) {
        specialty = p;
    }

    public void hire(Employee emp) {
        empList.add(emp);
    }
```

```
  public void showEmployees() {
    System.out.println(title
            + "'s employees are ");
    for (Employee emp : empList) {
      System.out.println(emp.getName());
    }
  }
} //Company
```

```
public class Product {
  private String productName;
  public Product(String n) {
        productName = n;
  }
  @Override
  public String toString() {
    return "[productName="
          + productName + "]";
  }
}
```

main(String [] args)

```
public static void main(String[] args) {
        Product p = new Product("Kayan");
        Company c = new Company();
        c.setTitle("CS");
        c.setSpecialty(p);
        Employee yindee = new Employee("Yindee");
        Employee preeda = new Employee("Preeda");
        Employee pramote = new Employee("Pramote");
        Employee haha = new Employee("HaHa ^.^");
        c.hire(yindee);
        c.hire(preeda);
        c.hire(pramote);
        c.hire(haha);
        haha.setSupervisee(yindee);
        haha.setSupervisee(preeda);
        haha.setSupervisee(pramote);

        c.showEmployees();
        haha.showSupervisee();
        System.out.println(yindee.getSupervisorName());
    }
```

Other Resources

- https://blog.devgenius.io/association-aggregation-composition-pattern-w-uml-5c68956fd689
- https://www.scaler.com/topics/association-composition-and-aggregation-in-java/
- https://www.codeproject.com/Articles/880451/Really-Understanding-Association-Aggregation-and-Composition
- https://javarevisited.blogspot.com/2014/02/ifference-between-association-vs-composition-vs-aggregation.html
- Why COMPOSITION is better than INHERITANCE - detailed Python example (ArjanCodes)
  - https://youtu.be/0mcP8ZpUR38