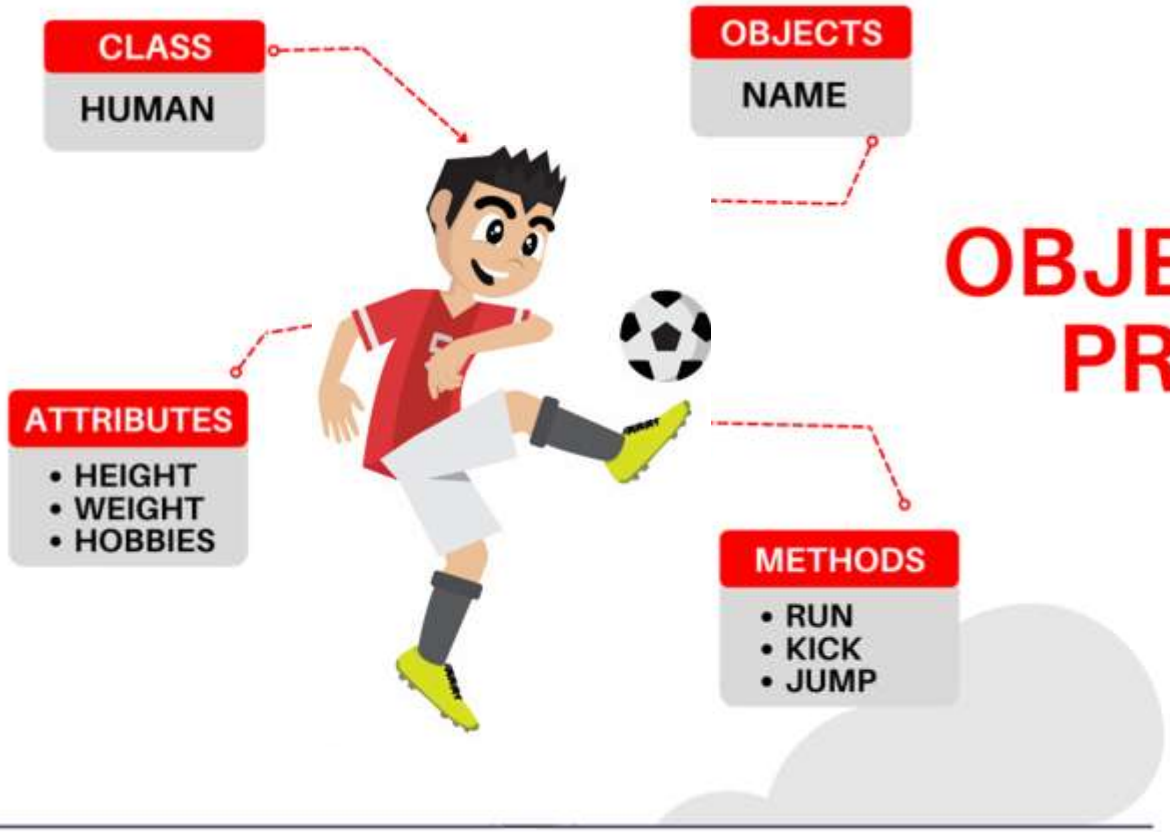
National University of Computer and Emerging Sciences



OBJECT-ORIENTED PRORAMMING

Summer 2023

Pir Sami Ullah Shah

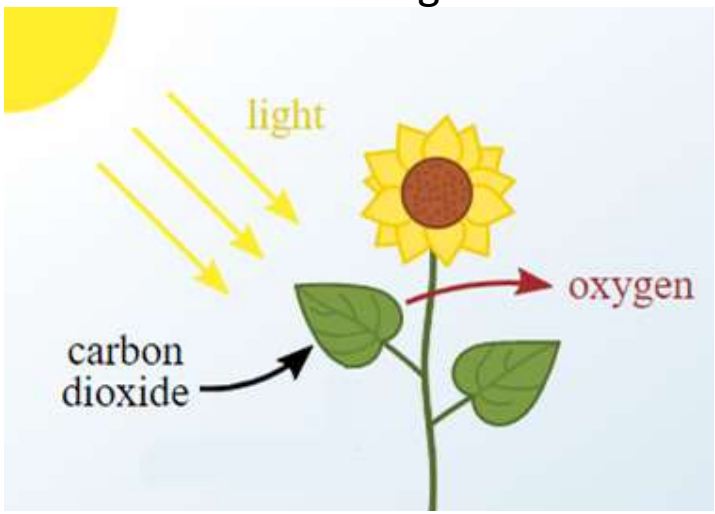
Lecture # 8 Aggregation, Composition and Association

Identifying objects

- Entities in the real world consist of attributes and behaviors
- One motivation for using OOP was that it represents real world entities better than structural programming

- How to identify objects / classes in the real world?

Objects
Plant, Flower, Leaves, Sunlight



Objects
Car, Body, Steering wheel, tires, engine, Driver

Identifying objects

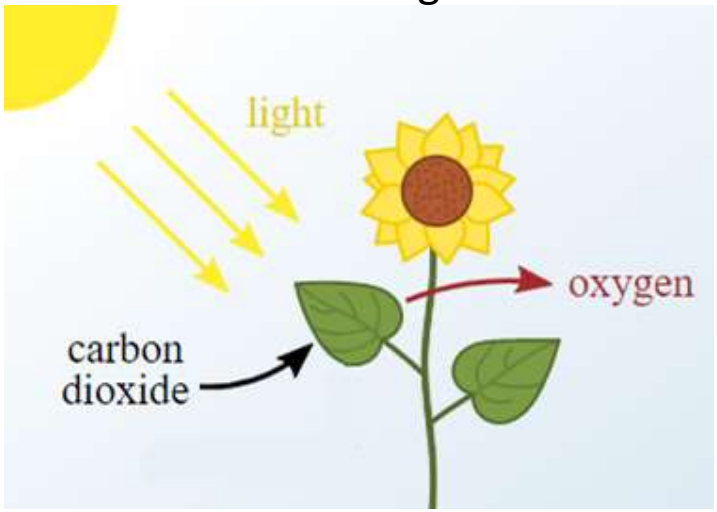
Case Study (Parking lot)
A parking lot is an open area designated for parking cars. We will design a parking lot where a certain number of cars can be parked for a certain amount of time. Each parking slot can have a single vehicle/car parked in it.

Interaction between objects

Plant Example

- Leaves interact with the sunlight

Objects
Plant, Flower, Leaves, Sunlight



Car Example

- Driver interacts with the car

Objects
Car, Body, Steering wheel, tires, engine, Driver

Relationship b/w objects

- Objects in the real world have relationships with each other
  - When an object has a relationship with another object it can interact with it

- What are the ***relationships between objects***?

- Understanding these relationships help in writing reusable and extensible code - HOW?

Relationship between objects

Plant Example

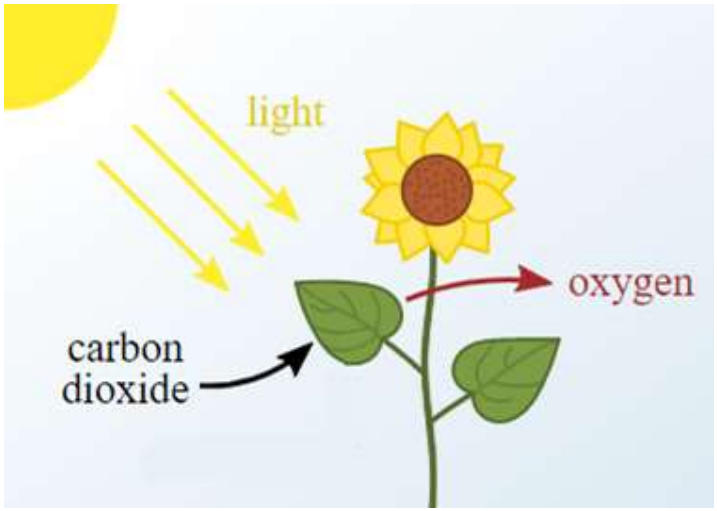
- Plant is composed of leaves and flowers
- Flower's ***existence depends on the plant.***

Car Example

- Car is composed of body, steering wheel, tires, engine etc.

Objects
Plant, Flower, Leaves, Sunlight

Objects
Car, Body, Steering wheel, tires, engine, Driver



Relationship type words

- How can we identify the relationships between objects?

- There are special "relationship type" words to describe these relationships. These are:

  - part-of
  - has-a
  - uses-a
  - depends-on
  - member-of
  - is-a

Can we use these words to describe relationships we identified?

How are these words useful in context of C++ classes?

Other examples of relationship words

- For example:
  - a square "**is-a**" shape
  - a car "**has-a**" steering wheel
  - a computer programmer "**uses-a**" keyboard
  - a flower "**depends-on**" a bee for pollination
  - a student is a "**member-of**" a class
  - Your brain exists as "**part-of**" you

- All of these relation types have useful analogies in C++.

Types of Relationships

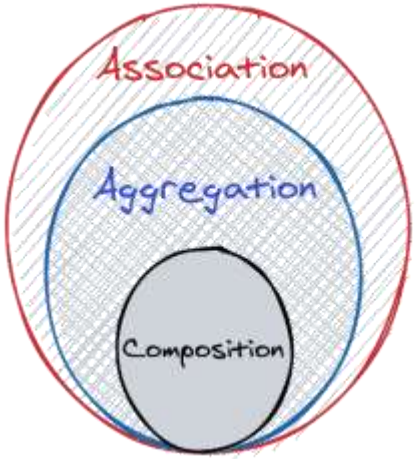
- Association
- Generalization

- Composition
- Aggregation
- Association

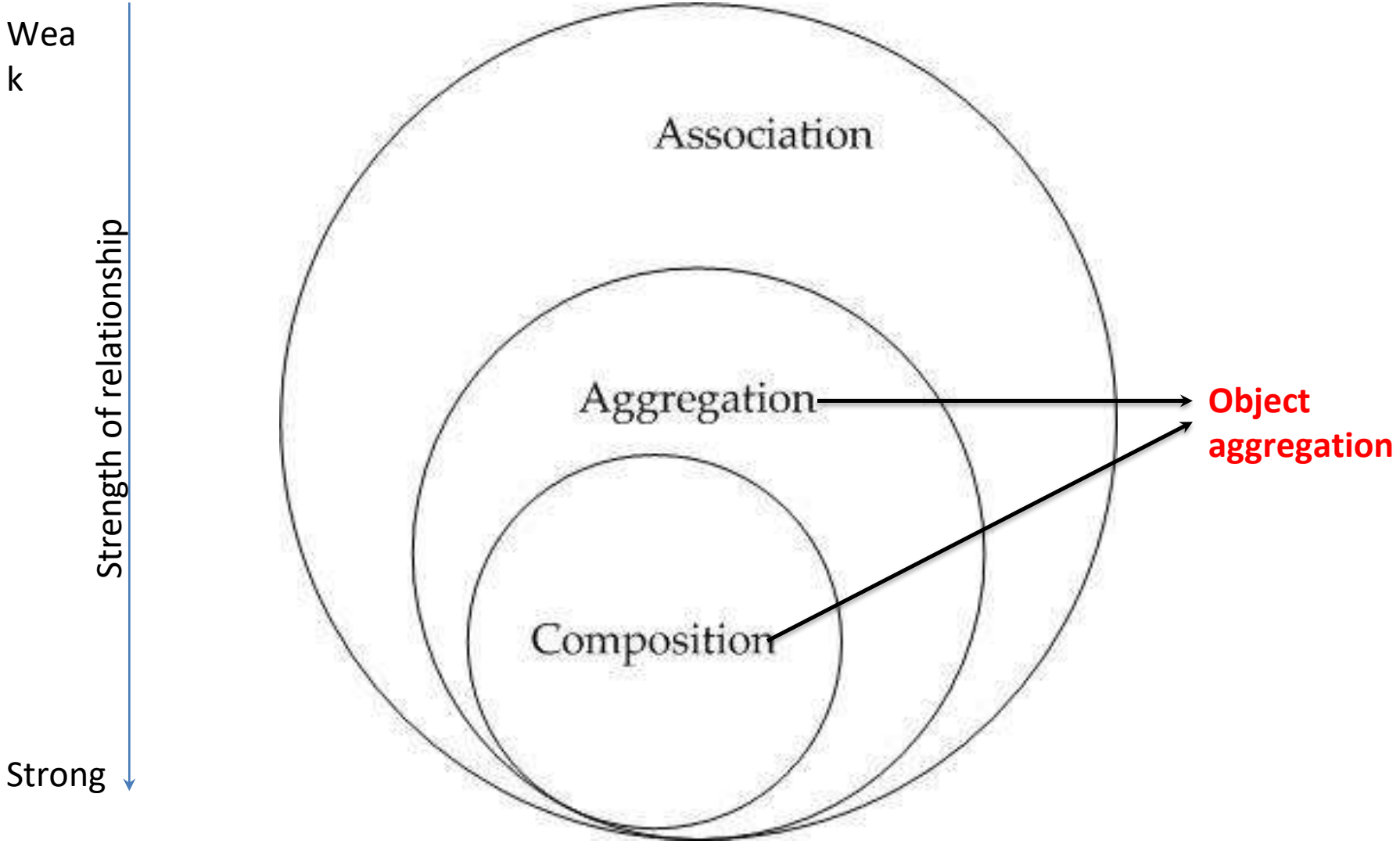
and

In these two objects are not really related!

- Dependency
- Realization



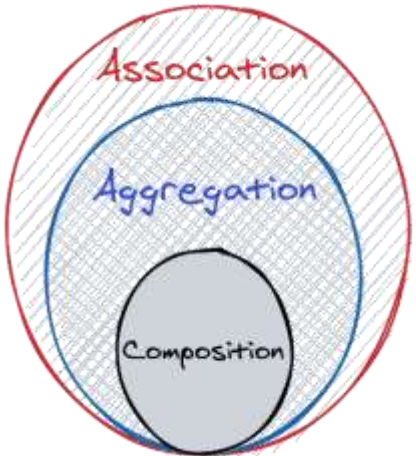
Relationships between Objects



The *HAS - A* Relationship

- Complex objects can be built using simpler ones is called object aggregation

- This relationship is described using ***HAS - A*** word
  - Car – engine, steering wheel, frame etc.
  - Computer – CPU, motherboard, memory etc.
  - Book – page
  - Car – Driver
  - Course – Instructor

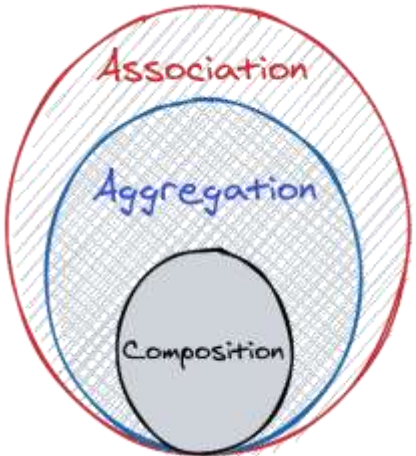


The *HAS - A* Relationship

- Complex objects can be built using simpler ones is called object aggregation

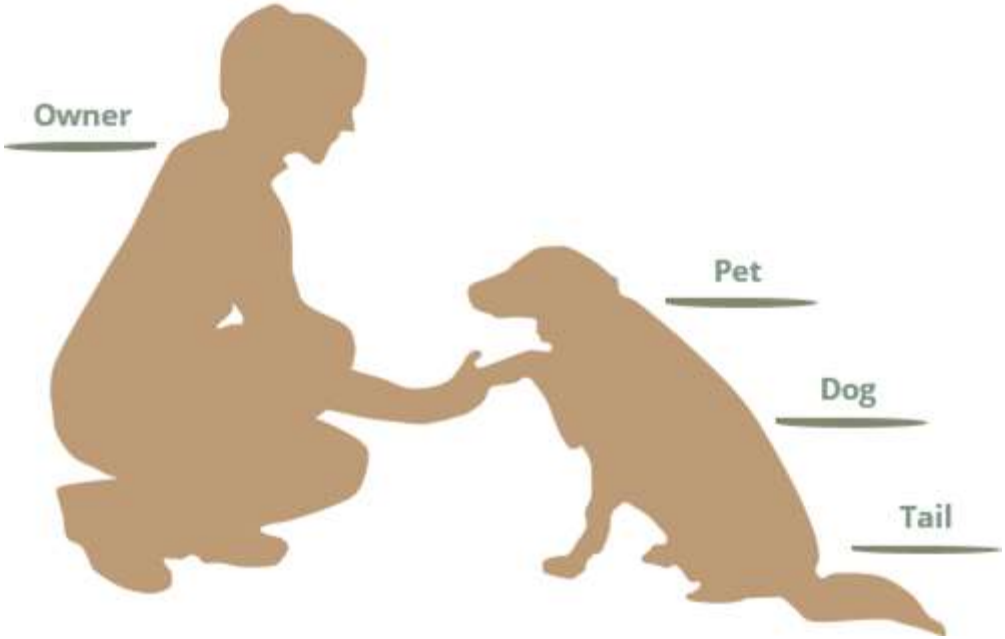
- This relationship is described using ***HAS - A*** phrase
  - Car – engine, steering wheel, frame etc.
  - Computer – CPU, motherboard, memory etc.
  - Book – page
  - Car – Driver
  - Course – Instructor

- Complex part is called the whole
- Simpler object is called the part



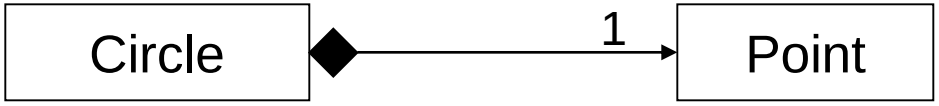
Types of object aggregation

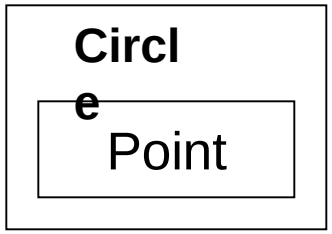
- Two types
  - Composition
  - Aggregation



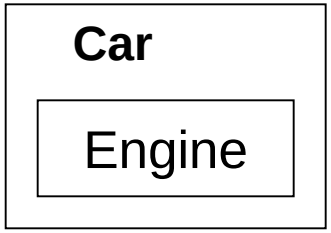
Composition

- Composition models "***part-of***" relationships
- These relationships are ***part-whole*** relationships
- Composition is often used to model physical relationships, where one object is physically contained inside another.
  - Heart is part-of body
  - Fish are part-of the pond
  - Circle is composed-of Point / Point is a part-of Circle





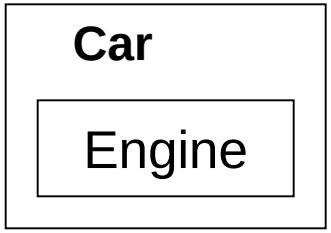
Engine is a part-of Car (Example)




```
class Car
{
    public:
        Car(char* e_No){
            cout << "Car created" << endl;
            ptr_engine = new Engine(e_No); //Engine created
        }
        void disp(){
            cout << ptr_engine->getEngineNumber() << endl;
        }
        ~Car() {
            cout << "\nCar destroyed" << endl;
            delete ptr_engine; //engine destroyed/deleted
        }
    private:
        Engine* ptr_engine;
};
```

Engine is a part-of Car (Example)

```
class Car
{
    public:
        Car(char* e_No){
            cout << "Car created" << endl;
            ptr_engine = new Engine(e_No); //Engine created
        }
        void disp(){
            cout << ptr_engine->getEngine
        }
        ~Car() {
            cout << "\nCar destroyed"
            delete ptr_engine; //e
        }
    private:
        Engine* ptr_engine;
};
```



Car is **composed of** Engine, therefore, the creation and destruction of the Engine object is managed by the Car .

The engine object **CANNOT** exist without the Car object

```cpp
//*********Composition Example************

//Accounts are a "PART-OF" the Bank    OR
//Bank is "COMPOSED-OF" Accounts

class Account {
    double balance;
    int accNum;
public:
    void open(unsigned int bal) {
        balance = bal;
    }
    int getAccNum() {
        return accNum;
    }
    double getBalance() {
        return balance;
    }
    void close() {
        //some closing code
    }
};
```

```cpp
class Bank {
    //Bank is composed of Accounts
    Account *accounts;//can also make as Account accounts[100]
    int numAccounts;
public:
    Bank(int accs=1) {
        //creation of Account object(s) controlled by
        //Bank object
        accounts = new Account[accs];
        numAccounts = 0;
    }
    void OpenAccount(unsigned int openingBalance) {

        //Bank object interacts with Account object
        accounts[numAccounts].open(openingBalance);
        numAccounts++;
    }
    double getBalance(int accNum) {

        //searches an account in the array
        for (int i = 0; i < numAccounts; i++) {
            //Bank interacts with Accounts to get account
            // number and balance
            if (accNum == accounts[i].getAccNum())
                return accounts[i].getBalance();
        }
    }
    ~Bank() {
        //destruction of Account object(s) controlled by
        //Bank object

        //Accounts cannot exist without Bank
        delete[] accounts;
    }
};
```

Composition Properties

- The whole is composed of the part
  - i.e. the part object is a ***data member*** of the whole class
  - e.g. the engine object is a data member of the Car class

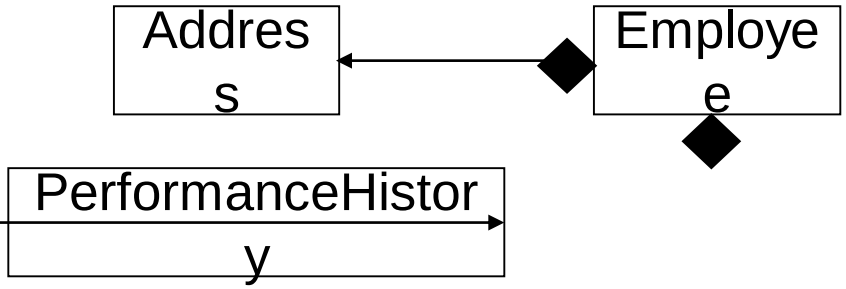
- The part can only belong to one whole at a time
  - i.e. a part object can only be a ***data member of a single whole object at one time***
  - e.g. a heart that is part of one Human cannot be a part of another Human at the same time

- The part has its existence managed by the whole
  - i.e. the existence of the part object ***is controlled by the whole object***.
  - e.g. the creation and destruction of the engine object is managed by the Car object

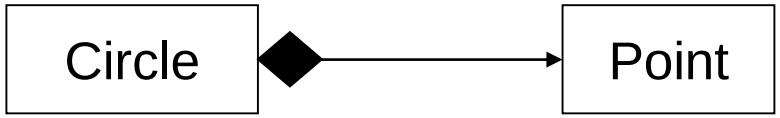
- The part does not know about the existence of the whole – ***unidirectional***
  - i.e. the part object is a data member of the whole class, the part object ***knows nothing about the whole class***, cannot access its functions
  - e.g. the engine object cannot call functions of Car class but Car can interact with the engine object

Composition

- Why create a so many classes instead of direct implementation in just a single class?
  - Car (whole) Engine (part) example

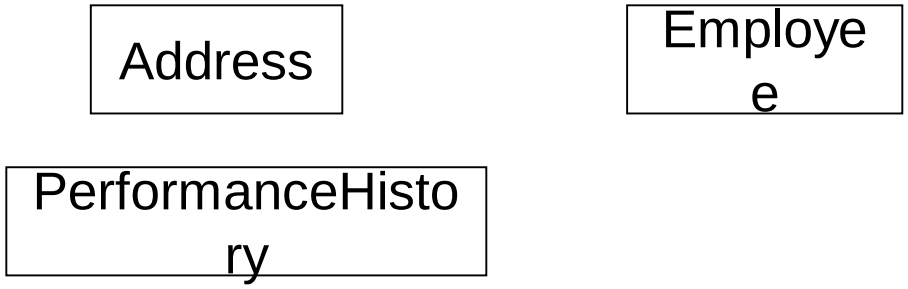


- Composition
  - **Each individual class** should be focused on performing **one task** (simple and straight forward)
  - Each class can be **self-contained**, which makes them **reusable**.
  - The **composing class** can focus only on **coordinating the data flow** between the composed classes.



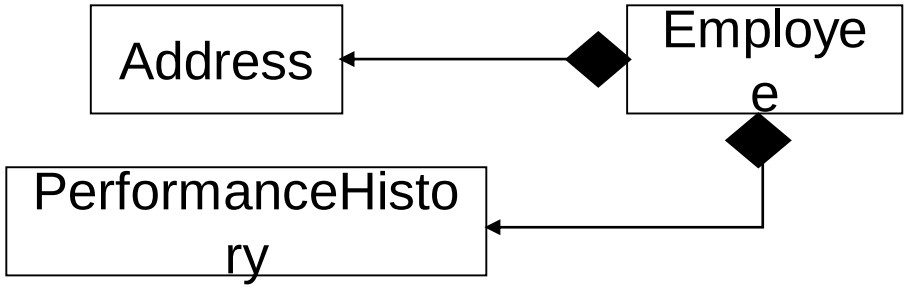
Composition

- Break down into simpler classes or directly implement everything in one class?
  - One class one task



Composition

- Break down into simpler classes or directly implement everything in one class?
  - One class one task



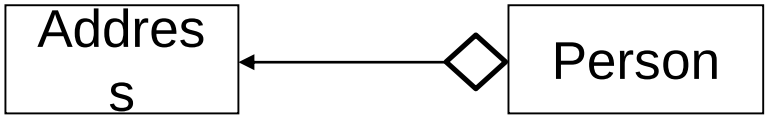
Aggregation

- An aggregation is also a part-whole relationship
- It models ***HAS-A*** relationship

- **Similarities to composition**
  - The parts are contained within the whole
  - It is also a unidirectional relationship

- **Unlike composition**
  - Whole is not responsible for the existence and lifespan of the parts
  - Parts can belong to ***more than one object at a time***

Aggregation



Singular part

- Every person has an address.
- One address can belong to more than one person at a time
- Address existed before the person starting living at the address
- Whole knows of existence (person knows)
- Part doesn't know about the whole

Multiplicative parts

- A address is part of the person.
- Address belongs to a Person,
- The same address can belong to other people as well.
- The Person is not responsible for the creation or destruction of the address.
- Whole knows about existence
- Part doesn't know about the whole

Person has an Address - Example

```
class Address
{
    private:
        int h_No; //house no
        int st_No;  //street no
        string sector; //sector
        string city; //store city
    public:
        //parameterized constructor
        Address(int h, int s, const string& sec, const string& c)
        { }

};
```

Person has an Address - Example

```
class Person
{
    private:
        string p_name; //person name

        //it will get reference to address object (part)
        const Address& p_address; // A person can live at only one address (here)

    public:
        //parameterized constructor
        Person(const string& s, const Address& address) : p_name{s}, p_address{ address }
        { }

        //display person details
        void disp_Person() const{
            cout << "Name: " << p_name << "; ";
            p_address.disp_Address();
        }
};
```

Person has an Address - Example

```
int main()
{
    //part object created
    Address part_Object( 12, 3, "G-20", "Islamabad" );

    //whole object created
    Person whole_Object("Random person", part_Object );

    whole_Object.disp_Person();


    return 0;
}
```

```
//*********Aggregation Example************
//Person is "HAS-A" Address
class Address {
    int h_num;
    int st_num;
    string sector;
    string city;
public:
    Address(int hnum, int stnum,const string&sector,const string&city) {
        cout << "\n Address Constructor called" << endl;
        h_num = hnum;
        st_num = stnum;
        this->sector = sector;
        this->city = city;
    }
    void display() const{
        cout << "House # " << h_num << ", Street # " << st_num
            << ", Sector " << sector << " City " << city << endl;
    }
    ~Address() {
        cout << "\n Address Destructor called" << endl;
    }
};
```

```
class Person {
    string name;
    //const to prevent changes
    const Address& add; //can also do Address *add;
public:
    Person(const string&name,const Address &address):add(address) {
        //initializer list used because const reference
        this->name = name;
        cout << "\n Person Constructor called" << endl;
    }
    void displayPerson() const {
        cout << "Name: " << name << endl << "Address ";
        add.display();
    }
    ~Person() {
        cout << "\n Person Destructor called" << endl;
    }    };
int main()
{  Address a1(20, 16, "Some Sector", "Some City");
    {  Person p1("SomeName", a1); //created in this scope
        p1.displayPerson();
        //Person p1 destroyed here, a1 still exists
    }
    a1.display();
    return 0;
}
```

Aggregation Properties

- The part is part of the whole
  - i.e. the part object is a ***data member*** of the whole class
  - e.g. a Patient object is a data member of the Doctor class

- The part can belong to more than one wholes at a time
  - i.e. the part object can be ***a data member of multiple whole objects***
  - e.g. a single Patient object can be a part of multiple doctor objects

- The part does not have its existence managed by the whole
  - i.e. the creation and destruction of the part is ***not done by the whole***
  - e.g. the doctor will not create or destroy patient objects

- The part does not know about the existence of the whole

Implementing aggregation VS composition

- **Aggregation**
  - **Parts** are **added** as **references** or **pointers**
  - **Whole** is **not responsible** for creation and deletion
  - Whole takes the objects it is going to point to as:

    1) **constructor parameters**;

    2) **parts are added later via access functions like setters**
  - Parts exists **outside the scope** of whole

- **Composition**
  - **Parts** are **added** as **normal variables** (or pointers)
  - **Whole** is **responsible** for **creation** and **deletion**

Examples

Composition

```
class Part{
    //class implementation
};

class Whole {
    private:
        Part* p; //can be normal variable
    public:
        Whole() {
            this->p = new Part();
        }
        ~Whole(){
            delete p;
        }
};

int main()
{
    Whole w;
}
```

Aggregation

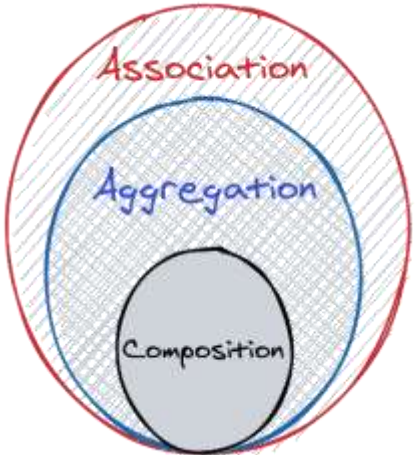
```
class Part{
    //class implementation
};

class Whole {
    private:
        Part* p;
    public:
        Whole(Part *p) {
            this->p = p;
        }
};

int main()
{
    Part* p = new Part();
    Whole w(p);
}
```

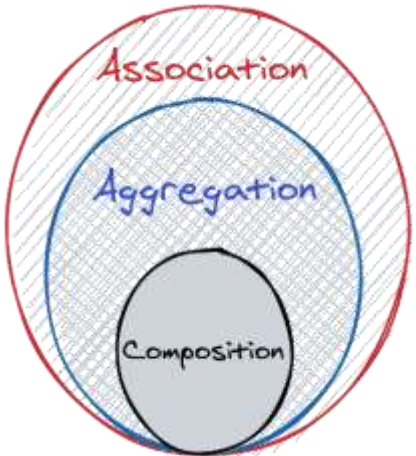
Association

- A weak of relationship
- Two otherwise unrelated objects
- There is no implied whole/part relationship
- Models a uses - a relationship

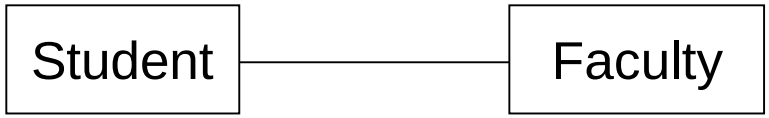


Association

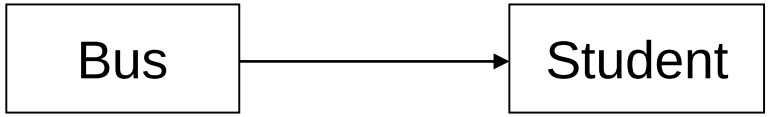
Composition	Aggregation	Association
Whole/Part relationship	Whole/Part relationship	**Associated object is unrelated**
Associated object can belong to only **one object**	Associated object can belong to multiple objects	Associated object can belong to multiple objects
Unidirectional	Unidirectional	**Usually Bidirectional**



Association



- The teacher clearly has a relationship with his students and *vice versa*
- It's not a part/whole (object composition) relationship
- A teacher can teach many students
- A student can study from many teachers
- Neither of the object's lifespans are tied to the other.
- Bidirectional



- A student has a relationship with the route bus
- Its not a part/whole relationship
- Multiple students can be on a certain route
- Neither of the object's lifespans are tied to the other
- Bidirectional

Implementing Association

- Associations are a broad type of relationship
- They can be implemented in many different ways
  - Association implemented using pointers

```
class A{//associated object
    private:
        B* b;
        //private members
    public:
        A(){
        }
};

class B{
    private:
        A* a;
    public:
        //constructors and member functions
};
```

```
class A{//associated object
    private:
        //private members
    public:
        A(){
        }
};

class B{
    private:
        A* a;
    public:
        //constructors and member functions
};
```

Composition vs Aggregation vs Association

Property	Composition	Aggregation	Association
Relationship type	Whole/part	Whole/part	Otherwise unrelated
Members can belong to multiple classes	No	Yes	Yes
Members existence managed by class	Yes	No	No
Directionality	Unidirectional	Unidirectional	Unidirectional or bidirectional
Relationship verb	Part-of	Has-a	Uses-a