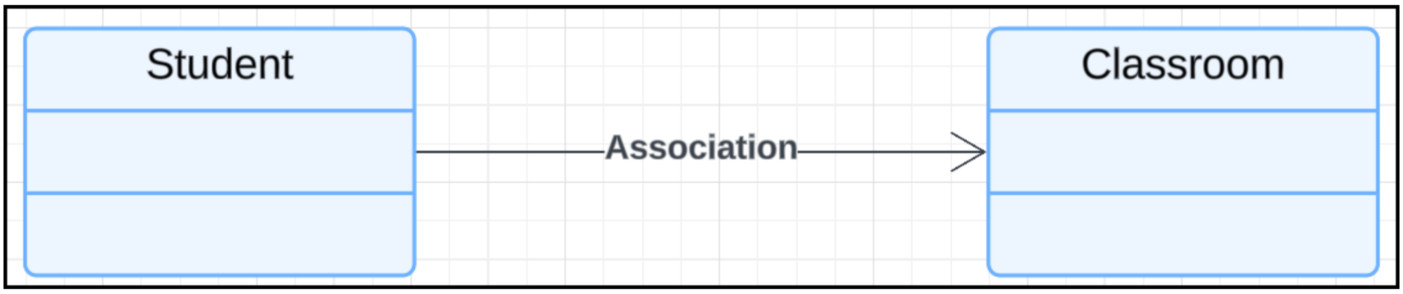
1. Association

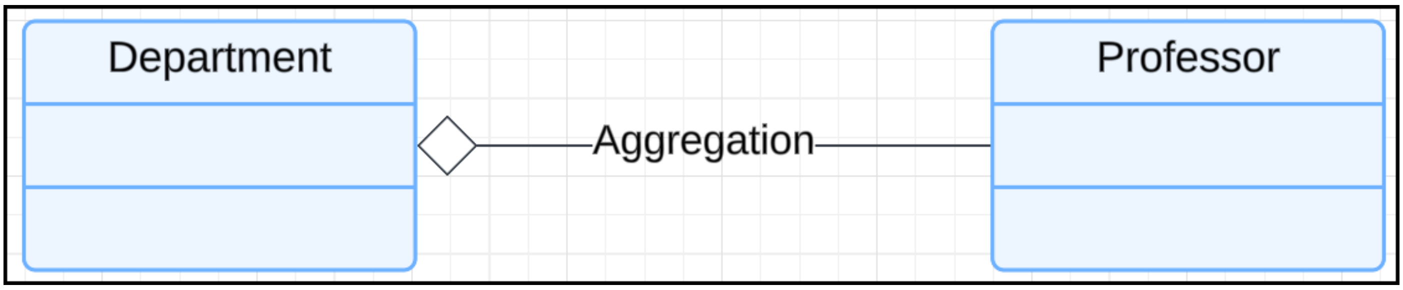


- If the arrow points from Student to Classroom, it means the Student knows about the Classroom.
- If there's no arrow on one end, it indicates a bidirectional association, meaning both classes know about each other.
- Student can "**navigate**" to Classroom but not necessarily the other way around.

```
class Classroom {
    // Classroom details
};

class Student {
private:
    // Student knows about Classroom
    Classroom* classroom;
};
```

2. Aggregation

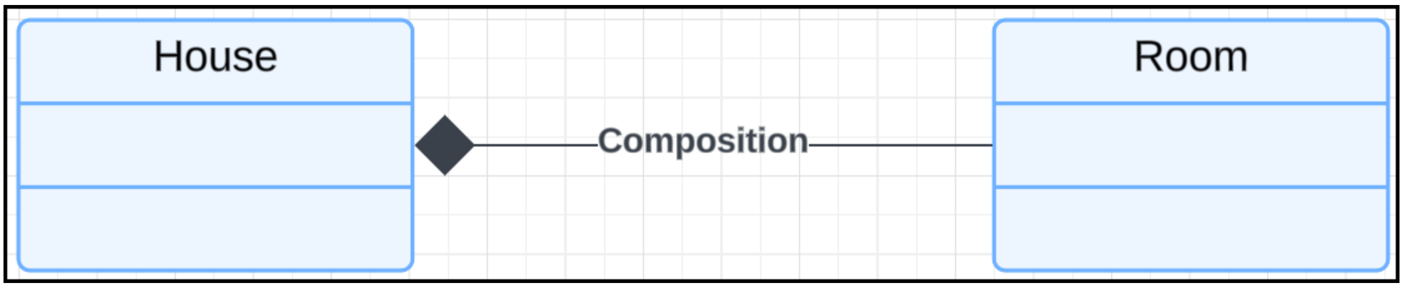


- A "whole-part" relationship where the part can exist independently of the whole.
- **Whole** (Department): Represents the container or aggregator.
- **Part** (Professor): Represents the independent object that can exist outside of the container.
- **Hollow Diamond**: Always resides on the "whole" side (the side that contains the other objects).

```
class Professor {
    // Professor details
};
```

```cpp
class Department {
private:
    // Contains professors, but
    // professors can exist independently
    std::vector<Professor*> professors;
};
```

3. Composition

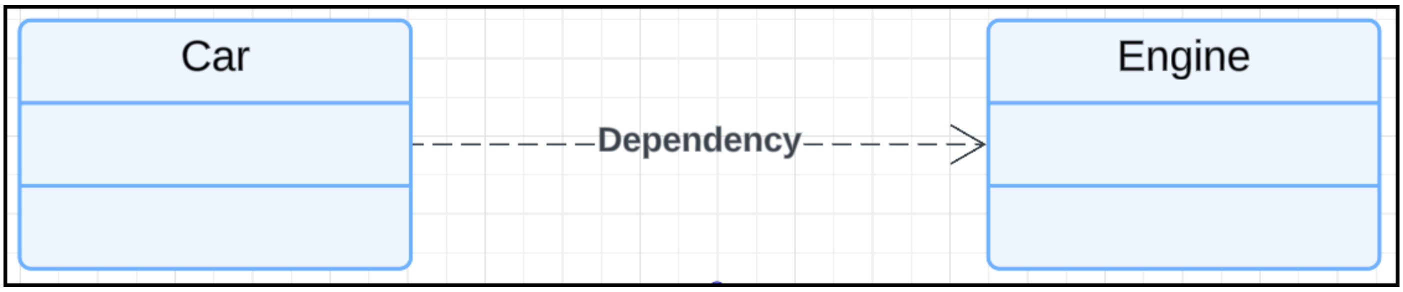


- Composition represents a strong "whole-part" relationship.
- **Whole** (`House`): Represents the container or owner.
- **Part** (`Room`): Represents the dependent object that cannot exist without the container.
- **Filled Diamond**: Always resides on the "whole" side (the side that owns the parts).

```cpp
class Room {
public:
    Room(const std::string& name)
        : name(name) {}
    std::string name;
};

class House {
private:
    // Rooms are part of the house
    std::vector<Room> rooms;
public:
    // Rooms are created as part of the house
    void addRoom(const std::string& roomName) {
        rooms.emplace_back(roomName);
    }
};
```

4. Dependency

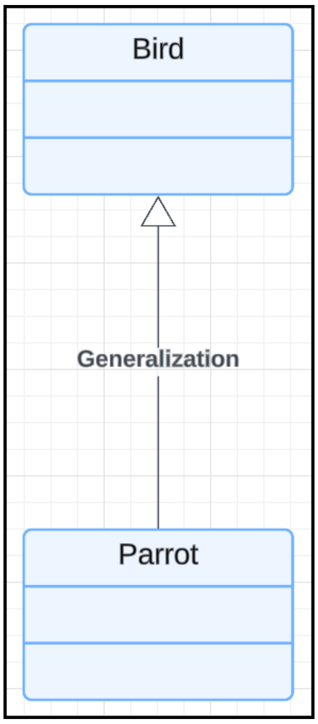


- A weaker, **temporary relationship** where one `class` depends on another for some `behaviour` or `functionality`.
- Use it when one `class` uses another as a `parameter` or during a `method` call but does not maintain a long-term association.
- Example: A `Car` `class` depends on an `Engine` `interface` to calculate fuel efficiency.
- The **dashed arrow** will point **from** the dependent `class` (in this case, the `Car` class) to the `class` or `interface` it depends on.

```cpp
class Engine {
public:
   // Abstract method
   virtual double calculateFuelEfficiency() = 0;
};
class Car {
private:
   // Car depends on Engine
   Engine* engine;
public:
   Car(Engine* e) : engine(e) {}
   // Car uses Engine's method
   void calculateEfficiency() {
      engine->calculateFuelEfficiency();
   }
};
```

5. Generalization/Inheritance

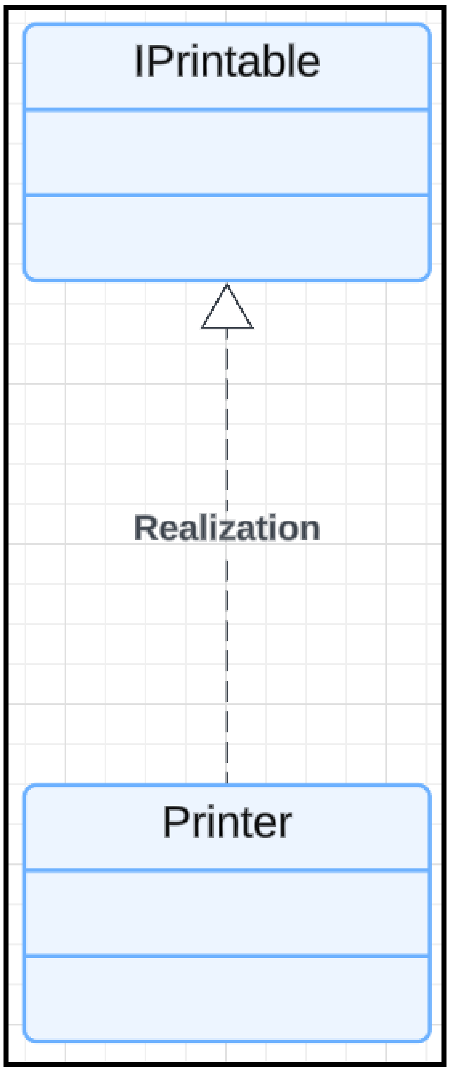
- Represents an inheritance relationship between a general parent class and a more specific child class.
- In a generalization (inheritance) relationship, the empty triangle (arrowhead) points from the subclass (specialized class) to the superclass (generalized class).



```cpp
class Bird {
public:
   virtual void makeSound() const {
      std::cout << "Chirp!" << std::endl;
   }
};
class Parrot : public Bird {
public:
   void mimicSound() const {
      std::cout << "Squawk!" << std::endl;
   }
};
```

6. Implementation

- A relationship where a `class` implements an `interface`.
- Implementation represents a contract where the `class` promises to provide the `behaviour` defined in the `interface`.
- The dashed line with the empty triangle indicates that the `class` implements the `interface`, meaning the `class` provides concrete `behaviour` for the `methods` declared in the `interface`.
- The arrow points towards the `interface` because the `class` is fulfilling the `behaviour` defined in that `interface`.
- `Printer` implements the `IPrintable` interface, so the dashed line with an empty arrow will point from `Printer` to `IPrintable`.



```cpp
class Printable {
public:
   // Pure virtual function
   virtual void print() = 0;
};

class Printer : public Printable {
public:
   void print() override {
      std::cout << "Printing document...\n";
   }
};
```